International Journal of Computers and Applications

ISSN: 1206-212X (Print) 1925-7074 (Online) Journal homepage: https://www.tandfonline.com/loi/tjca20

Concurrent Deadlock Detection In Parallel Programs

W. Haque

To cite this article: W. Haque (2006) Concurrent Deadlock Detection In Parallel Programs, International Journal of Computers and Applications, 28:1, 19-25, DOI: 10.1080/1206212X.2006.11441784

To link to this article: https://doi.org/10.1080/1206212X.2006.11441784

International Journal of Computers and Applications, Vol. 28, No. 1, 2006

CONCURRENT DEADLOCK DETECTION IN PARALLEL PROGRAMS

W. Haque*

Abstract

Many parallel programs have been developed that use message passing for communication. This leads to efficient and portable programs, but their complexity makes them hard to debug. One of the common problems in such programs is the detection of deadlocks. A deadlock detector, MPIDD, has been developed for dynamically detecting deadlocks in parallel programs that are written using C++ and MPI. The detection code for most of the blocking and non-blocking point-to-point and collective routines has been implemented. The code has been tested against an extensive test suite, application programs, and some publicly available benchmarks. The detector takes advantage of the MPI's profiling layer, requires no significant modification of user's code, and incurs very little overhead when invoked. Portability of the detector code is also a key advantage.

Key Words

Parallel programming, distributed memory, shared memory, MPI

1. Introduction

A large number of computational problems, including those in such emerging computing fields as bio-based techniques and quantum computing, are inherently parallel. The architectural advancements, availability of tools, and development of standards for distributed and parallel computing have made such problems more tractable. However, parallel programming does impose a few challenges, such as homogeneous versus heterogeneous platforms and whether the programs run in a shared memory or a distributed memory environment. In either case, when a task is split into many subtasks, which execute on multiple processors, interprocessor communication becomes necessary. Such communication is generally classified as collective communication that involves a group of processors. Collective communication is supported by the Message Passing Interface (MPI) [1–4], which has now emerged as the standard message-passing library specification. This allows developers of parallel software to write parallel programs that are both portable and efficient. Unfortunately, debugging message-passing applications can be difficult [5–8]. This is because the complexity of designing and developing programs using message passing leads to a number of problems, the most nasty of which is the existence of a *deadlock*. A deadlock occurs when one resource is busy trying to acquire some other resource that in turn is trying to acquire yet another resource, and so on, leading to a cycle. Each resource is unable to satisfy requests until the resource ahead in the cycle is free. This can happen in MPI programs when nodes in such a cycle are each waiting to receive a message and so cannot send messages to other nodes.

This document describes the inner workings of a software system, MPI Deadlock Detector (MPIDD), which was developed at the University of Northern British Columbia to detect deadlock situations within the MPI environment and aid users in resolving them. MPIDD uses a central manager to detect deadlocks, but is based on the distributed memory model for the programs being executed. It requires only insertion of a single line in the programmer's original code and uses the MPI profiling layer to efficiently detect deadlocks. It can run on any designated node and, because of the multithreaded approach, the involvement of the host node in a deadlock will not cause the detector to fail.

1.1 Related Work

There are currently two main ongoing projects involved in developing tools to detect deadlocks in an MPI system: the High Performance Computing Group at Iowa State University is working on MPI-Check [9] and Lawrence Livermore National Laboratory is pursuing dynamic software testing using Umpire [10]. Both of these approaches differ from MPIDD in various aspects. MPI-Check runs the original code through a preprocessor and inserts a large amount of its own code into the original source code. In addition, the original calls to MPI routines are modified by adding/replacing the parameters in the argument list. The so-called "deadlock detection" is indirectly achieved by hand-shaking code that is inserted prior to each call to a send or receive routine to ensure matching calls. There are a few things that make this approach somewhat impractical. First, a program must be run on the source code to insert the handshaking protocol. This

* University of Northern British Columbia, 3333 University Way, Prince George, BC, Canada V2N 4Z9; e-mail: haque@unbc.ca
(paper no. 202-1383)

may also mean that the programmer has to keep two copies of the code, the original version and the handshaking version. In contrast, to use our detector, MPIDD, no change to the code is required; instead, the program only needs to be linked to the "*MPIDDC.a*" library, which contains support for both C and C++. Another problem with handshaking is that it may be very unreliable. If an MPI command is not matched within the timeout period, the program will assume that it is deadlocked. MPI-Check also does not provide any support for C++. Finally, the overhead associated with using MPI-Check can be phenomenal. Umpire is more similar to our program. Both MPIDD and Umpire utilize the MPI profiling layer, and they both dynamically find deadlocks as the program is run. However, Umpire runs as a separate process using shared memory to communicate between processes. This is very efficient, but the problem is that it will only work in an SMP environment. Umpire assumes and makes use of shared memory to keep a central state of the system, including a wait-for-graph that it can then search for cycles to detect deadlocks. In contrast, MPIDD acts as another MPI process, and is sent information using the actual MPI function calls. This allows its use in both a distributed or shared-memory environment. Finally, MPIDD allows runtime monitoring of data structures and also provides a graphical interface to aid in resolving deadlocks.

1.2 Portability

MPIDD has been tested on two distinct platforms in order to demonstrate its portability. The code was initially designed to run on a heterogeneous set of machines all running Local Area Multicomputer (LAM)/MPI 6.5.6 under Linux. It was then ported over and tested on a 28-node SGI Origin 3400 running SGI MPI under Irix.

1.3 Organization of This Paper

We begin by describing the architecture of MPIDD and detection algorithm in Section 2. This section also includes a description of various components of the detector and the wrapper header file. In Section 3 we present our testing methodology and results. We conclude the paper in Section 4.

2. System Architecture

The system architecture for our deadlock detection model is shown in Fig. 1. There are two main parts of the MPIDD system, the detector and the MPIDD wrappers. The detector is a separate program that receives input from the client programs about MPI commands being issued. It uses this information to construct a state and determine if there is a deadlock in the client programs. The information to the detector is provided via the MPI function call wrappers that simulate the client program's original calls. MPI's profiling layer is used to create a wrapper containing information needed by the detector, and the actual function call is embedded inside this wrapper. As indicated earlier, the client only needs to include the file *MPIDD.h* in the original code, and all of this functionality is obtained transparently. The new *wrapped* routines basically act as a spy, seeing what commands are being issued and relaying this information to the deadlock detector program. These

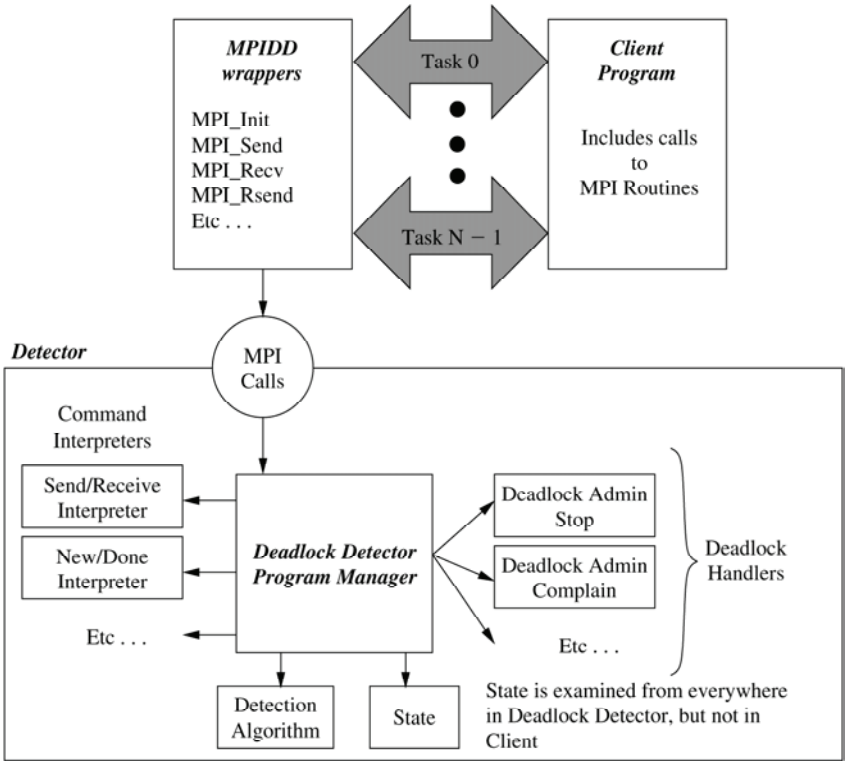

Figure 1. System architecture.

wrapped routines also contain some synchronization code that is used to avoid detecting false deadlocks. An example of the structure of the wrapped routines is provided later.

2.1 Detector

The detector is made up of five main parts: the manager, the state object, the detection algorithm, the command interpreters, and the deadlock handlers. The routine *main()* simply extracts some command line arguments, adds all the deadlock handlers and command interpreters into the manager, and then calls the manager's member function *run()* to start the detection process. All clients must register with the detector before sending any MPI calls; these clients then de-register upon completion.

2.1.1 Manager

The manager is started with the method *Manager::run()*. This method starts a loop that does not end until all the clients have signed off. The manager sits in the loop waiting for input from one of the clients (through the routines defined in MPIDD.h). These inputs represent commands that are then built into a command object and passed to the command interpreter designated to add that command to the *state*. Then, satisfied that the command was handled properly, the manager checks to see if all of the clients have finished and exited by checking a Boolean variable contained within *state*. If not, it then checks a status variable in *state* to see if there is a deadlock. If the variable indicates a deadlock, the manager activates the deadlock handler and waits for input again.

2.1.2 State

State is a simple object that contains the current state of the system. It keeps track of all currently running clients and the commands issued by those clients that may cause a deadlock. It contains variables representing the state of a deadlock (or lack thereof) and whether or not all clients have exited. Most of the *state* is represented by an array of *ProcessInfo* objects (Fig. 2). Each one of these objects denotes a client process. Also defined in *State.h* is the *Command* class. It represents an MPI command that the client has issued. Each *Command* object has six fields: *from* (the client who issued the command), *to* (the client/clients who will receive the command), *type* (the type of command), *tag* (tag of a command, which is used by MPI to match point-to-point commands), *comm* (communicator used by the command), and *extra*, which just holds any extra information that may be needed.

Fig. 2 shows the state object for the given example. In this case, process 0 is waiting to receive a message from process 2 and process 2 is waiting to receive a message from process 1, which in turn is waiting to receive a message from process 0. This leads to a cycle that represents a deadlock. Meanwhile, processes 3 and 4 have issued *send* calls to process 0 that will never be received because of the deadlocked status.

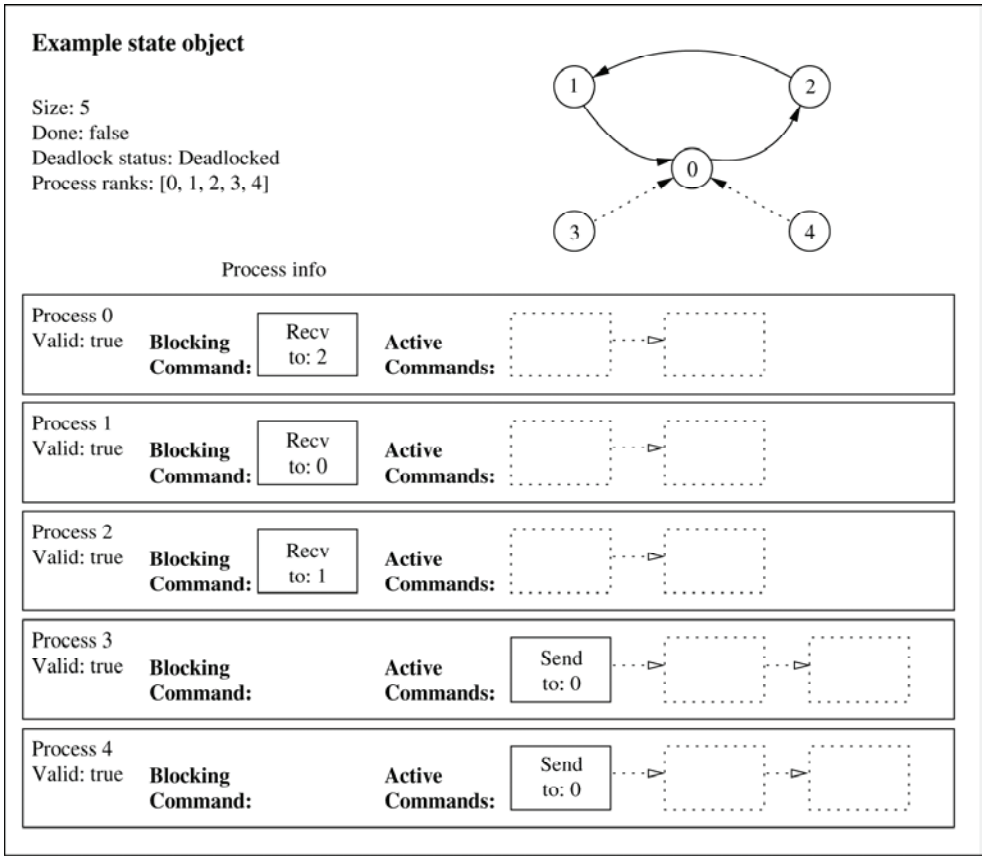

Figure 2. The state object.

2.1.3 Detection Algorithm

Generally, the command interpreters invoke the detection algorithm.[1] However, for some calls such as MPI_Init or MPI_Comm_size, it is not necessary for the command interpreter to invoke that algorithm. The detection algorithm uses the state object intensively, and its correctness depends upon the accurate representation of the current state in the MPI environment. Additionally, the only commands we need to check are blocking commands. Non-blocking commands do not cause deadlocks.[2]

The detection algorithm is simply a depth-first search for cycles in the dependency graph represented by the state object. The algorithm follows dependencies signified by blocking commands of each process, and detects a cycle by simply memorizing where the detection started. Detection is done by a recursive member function of the DeadlockDetector class, *isCycle()*. The function returns *true* if a cycle is found and *false* otherwise; it also records all nodes that are involved in the deadlock. The pseudocode for *isCycle()* is shown below.

```
//MPIDD_IGNORE represents dependencies that
   should be ignored
if (current node is MPIDD_IGNORE)
   return false;
for (i=0 to num of dependencies of blocking command
   of current node)
    // source is the proc_id where the algorithm
       started
    if (source == dependency[i] or (isCycle(source,
    dependency[i]))
    {
    record current node as part of deadlock;
    return true;
    }
return false;
```

This algorithm is efficient because it does not look for all cycles in the state, but only for cycles involving the source node. Other cycles are assumed to have already been detected at the time the command that completed the deadlock cycle is added into *state*. Note that this is not an assumption of the system, but rather an assumption of one deadlock pass about the behaviour of other deadlock detection passes. This can be further explained as follows. If a blocking call waits long enough to cause a deadlock detection pass, but the node that was blocked was not involved in a deadlock, then no deadlock will be detected in that pass (correctly). If a deadlock had concurrently arisen involving other nodes, then another deadlock detection pass will be invoked when the other nodes have been waiting for long enough (specified as a parameter). This second deadlock detection pass will detect the deadlock.

The commands that can be satisfied by a corresponding command from any or multiple nodes are also handled. These blocking commands have an array of "*to*" fields, and only cause a deadlock if every node that the current node is waiting on is deadlocked or cycled. If a single node has a chance to fulfill the request, it is not deadlocked. Also, to identify which nodes are causing a deadlock and which ones are merely blocked by a deadlock, vectors are passed down through the recursive calls. Whenever a recursive call is made, a new vector is passed into the call.

[1] Refer to "Concurrency and Optimization" section for an explanation of how this overhead is reduced.

[2] Note that non-blocking calls that lead to a deadlock are handled by the corresponding blocking calls.

2.1.4 Command Interpreters

Command interpreters take commands as input and use these commands to update the state object. The command interpreters are a group of classes derived from an abstract *CommandInterpreter* class. Each interpreter registers itself with all the types of commands it can handle when it is added to the manager. Whenever a command of that type is received by the manager it is passed to the interpreter for processing. Generally, this involves updating the state object and calling the deadlock detection algorithm, if necessary. As noted earlier, for some MPI routines it is not necessary to invoke the detection algorithm.

2.1.5 Deadlock Handler

The deadlock handler decides what to do if a deadlock is found. The user can specify one of the three options: Complain, Stop, Debug. The first two options should be used when the overhead is to be kept to a minimum. The action taken under each one of these options is described below. *Complain* is used if the user wishes to attempt to continue in spite of the deadlocks. *Stop* is used if the user wishes to halt the program as soon as a deadlock is detected. Both Complain and Stop can be enhanced by using the *Monitor* option. The *Debug* option is to be used when the user wishes a more interactive role in the detection and resolution process.

- *Complain.* If a deadlock is detected, Complain outputs a warning and attempts to continue the detection process. A sample LAM MPI application schema file that uses Complain is shown below:

  ```
  n0 -s n0 ../Deadlock0 -d 2
  n1 -s n0 ../Deadlock0 -d 2
  n2 -s n0 ../DLDetector -d 2 -h complain
  ```

- *Stop.* If a deadlock is detected, Stop outputs a warning and halts the program, which causes all of the clients to exit as well.
- *Monitor.* This option (when specified with Complain or Stop) allows graphical monitoring of the data structures as they progressively change during the execution of a program. The state object contains most of the information needed to generate this output.
- *Debug.* This is the most interactive part of our development. When this option is activated, upon detecting a deadlock, the control is transferred to a graphical user interface (as in the monitor option described above), which allows tracking of the state and interactive termination of processes in an attempt to resolve deadlocks. A *wait-for* graph is also generated.

The LAM MPI application schemas for Stop and Debug options are similar to the one shown above where Complain is replaced by the appropriate keyword.[3]

2.1.6 Concurrency and Optimization

Instant invocation of the deadlock detector upon receipt of every single MPI command can become an unacceptable overhead. Our detector, therefore, uses a concurrent protocol to optimize its use. First, the parent thread creates a child for every blocking command that is added to the state. These children then sleep for a specified amount of time. If the corresponding command unblocks before the child wakes up, a signal is sent to terminate the child. On the other hand, if the time runs out and the child wakes up, it starts the detection algorithm and looks for deadlocks in the state. As soon as a deadlock is detected, the child signals its parent of this event and then exits. After receiving the signal, the parent thread invokes the deadlock handler specified. This protocol improves efficiency dramatically because only the commands failing to resolve themselves within a given amount of time are checked for deadlocks.

What is interesting with this algorithm is that each child receives only a snapshot of the state at the time it is created; this snapshot is not further updated. Thus it is possible for the child to be detecting cycles according to a state that is not consistent with the current state, due to the state being updated while the child was sleeping. This is done simply because updating all copies of the state belonging to each child is not feasible. Fortunately, this does not effect the detection of deadlocks. To explain this we look at three possible scenarios.

1. *Scenario 1.* A child is spawned due to a blocking command being added to the state. This blocking command completes a dependency cycle and thus creates a deadlock.
2. *Scenario 2.* A child is spawned due to a blocking command being added to the state. This blocking command does not create a deadlock; however, it becomes part of a deadlock after a few more commands are added to the state.
3. *Scenario 3.* A child is spawned due to a blocking command being added to the state. The blocking command does not create a deadlock, nor is it part of a future deadlock.

In scenario 1, the command completes a dependency cycle and creates a deadlock. Inconsistency of the snapshot with the current state will not affect the correctness of the detection in this case, because newly added or removed dependencies would not resolve, simplify, or complicate the deadlock created. Because our goal is only to detect deadlocks, we are only interested in the status of the processes involved in the deadlock, but not in the rest of the state. Thus newly added or removed commands since the snapshot are irrelevant. In scenario 2, the command does not complete a cycle but does become part of a deadlock before the child detection thread wakes up. On the surface, this might seem like a problem due to inconsistencies between the snapshot and the current state, because it is now impossible for the child to detect the already existing deadlock. However, if we look into the future a little, it is not hard to see that the deadlock will be detected by the detection thread created when the blocking command completing the dependency cycle was added into the state. According to this analysis, the problem with scenario 2 will eventually become scenario 1 and be solved in the same manner. Finally, scenario 3 does not really create a problem, as the command is not involved in a deadlock in any way.

[3] The SGI MPI does not allow use of such schema files. This functionality was attained by inserting additional code to identify placement of the detector.

2.1.7 Synchronization

In order to avoid detection of "false" deadlocks resulting from out-of-sequence/delayed arrival of messages to the detector, a simple synchronization protocol is used that requires that all messages sent to the detector must be acknowledged. This is also accomplished via the profiling layer and is included with the wrappers. It does affect the overall performance of the detector, but the overhead becomes insignificant as the processing time becomes much larger than the communication time. To explain the synchronization process, consider the following code segment:

```
# include "MPIDD.h"
    ...
if(currentNode == 0)
    MPI_Ssend (message, 100, MPI_CHAR, 1, tag,
      MPI_COMM_WORLD);
    MPI_Recv (message, 100, MPI_CHAR, 1, tag,
      MPI_COMM_WORLD, status);
else if(currentNode == 1)
    MPI_Recv (message, 100, MPI_CHAR, 0, tag,
      MPI_COMM_WORLD, status);
    MPI_Ssend (message, 100, MPI_CHAR, 0, tag,
      MPI_COMM_WORLD);
    ...
```

This segment of code simply communicates between two nodes and should not deadlock; however, without the synchronization mechanism, the detector might detect a false deadlock. Let us first look at the way this code should be interpreted by the detector. Concurrently, node 0 and node 1 send out two messages to the detector announcing the invocation of MPI_Ssend and MPI_Recv, respectively. These two messages reach the detector and cancel each other out in the state of the detector. Then node 0 and node 1 each send out another message to the detector, announcing the next two commands being invoked; these commands again cancel in the state of the detector. Everything is well, and there is no deadlock. Now imagine that the message from node 0 announcing the invocation of MPI_Ssend is delayed on the way (Fig. 3). But the actual MPI call does complete on time, not knowing the delay of the message to the detector. Now, as expected, each node will again send to the detector announcing its next commands. The delay persists causing the second message from node 1 to reach the detector before the first message from node 0. This is when the problem occurs. As

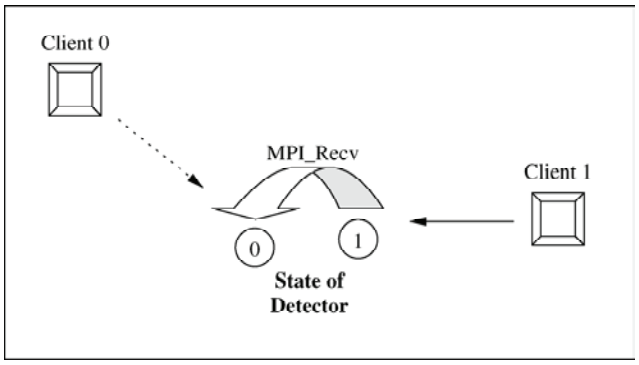

Figure 3. Message from client 0 is delayed on the way to the detector.

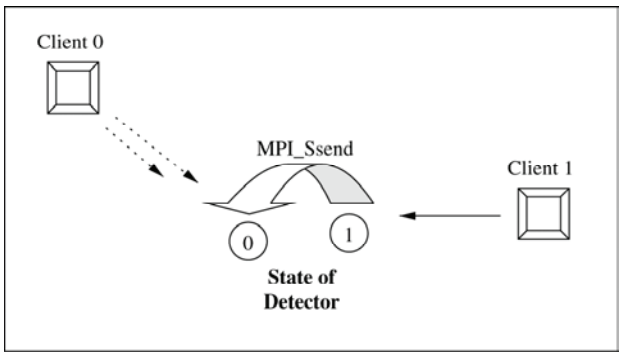

Figure 4. Message 2 from client 1 reaches detector before message 1 from client 0; thus new blocking command replaces old blocking command before cancelling occurs.

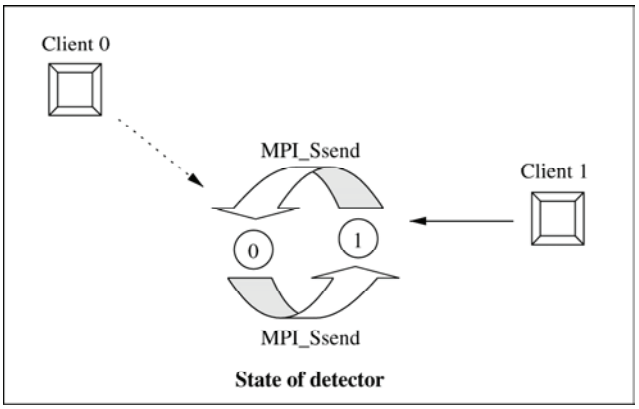

Figure 5. Clients 1 and 0 appear to be deadlocked when the delayed message finally gets to the detector.

each process can have only one blocking command, node 1's MPI_Recv command would be overwritten from the state by the new blocking command MPI_Ssend (Fig. 4). Finally, when the first message from node 0 arrives, there is no command for it to match with and cancel; thus the state now consists of two MPI_Ssends having dependencies to each other, causing a false deadlock (Fig. 5). The synchronization mechanism solves this problem by forcing an acknowledgement from the detector. This ensures that the next command will not be invoked until the detector has received the message announcing the invocation of the current command.

2.2 MPIDD.h Wrappers

All of the MPI function wrappers are contained in a header file called *MPIDD.h.* Basically, we need to have wrappers around any MPI routine that can help us determine if there is a deadlock. Also, we need to redefine many of the communicator and collective routines so that the client programs do not accidently send messages to the deadlock detector. Recall that the detector runs as another MPI process in our implementation. The general structure of the wrappers is as follows:

```
type MPI_call(argument_list)
{
  synchronization code
  our spy code for the detector
  PMPI_call(argument_list)
  more of our spy code
}
```

Another important data structure that was created is called *requestList.* As the name implies, it holds special information on requests. This is needed for commands such as MPI_Send_init that do not actually affect the state of the system until MPI_Wait is called. The *requestList* stores all the information about the requests contained therein, so that when a call like MPI_Wait is handled, the detector is informed about the corresponding command involved.

3. Testing and Results

There are two phases to our testing process. First, an extensive test suite has been developed to test the functionality of our deadlock detector. The suite contains a variety of cases ranging from simple deadlock between a pair of processes to more complex cases involving a subset of processes or multiple cycles. The tests are grouped into categories depending upon the complexity of a test case. The tests are generated using scripts that define rules and the desired complexity. A separate suite has been developed for point-to-point routines and collective routines. The tests are then run automatically, once again using scripts. After the basic functionality is verified, we proceed to the second phase in which publicly available application codes, such as NAS Parallel Benchmark [11], were used to test the limits of our detector. These codes are mature codes and are generally deadlock free. However, the codes are large and complex enough to stress MPIDD programming structures. In some cases, deadlocks were intentionally introduced by modifying the code; MPIDD was able to detect all deadlock situations. An MPICH test suite obtained from Argonne National Lab was also used for testing [12].[4] This test suite was primarily used for stress-testing and to test the functionality of the detector.

[4] This test suite is originally written in C, but needed only minor modifications for the C++ compiler.

The code was systematically altered to introduce deadlocks. For performance testing, the SKaMPI benchmark [13] was used, and the reports generated are available on the author's website.

3.1 MPIDD Overhead and Performance

The extensive optimizations used in the design of MPIDD are demonstrated by noticeably low overhead, particularly when the Complain or Stop option is used without the monitor. The detector introduces two types of overhead, one constant and the other associated with each MPI call. Besides the time required to construct the detector itself, each process must register itself with the detector before any MPI routines are invoked and then de-register upon completion. This constitutes the constant overhead that does not increase as programs become more complicated. The overhead associated with each MPI call also becomes insignificant when processing time at each node becomes very large.

Because MPIDD will be used in an environment that requires a large amount of processing, the performance overhead of MPIDD falls well within acceptable limits. As an example, the IS benchmark from NAS Parallel Benchmark suite shows only a 1% overhead when run with the MPIDD on a five-node cluster. Similarly, the matrix multiplication code for 500×500 matrices showed an overhead of 10.78% in the same environment. The comprehensive reports from the SKaMPI benchmark are too large to include in this paper; these are available on the author's website [14].

4. Conclusion

Currently, no comprehensive tools exist that would aid a programmer to dynamically locate errors in message-passing parallel programs. Most debugging is done manually using interactive techniques which are time consuming, error prone, and complicated. In this paper, we have presented MPIDD, which is a tool for dynamically detecting and resolving a very common problem in parallel programming, that is, deadlocks. The key advantages of our tool are its portability, very low overhead, and the use of MPI's profiling layer that allows the use of this tool without modification of the original source code. The reason for this is that the detector runs as another MPI process. Further, the key to low overhead is the delayed initiation of the deadlock detection pass, which automatically excludes the calls that resolve themselves within a specified amount of time. A comprehensive test suite has also been designed to test a variety of deadlock situations, and most blocking and non-blocking point-to-point and collective routines have been implemented and tested. In addition, the detector has also been tested with application codes that, in some cases, were modified to include deadlocks. Even though we use a centralized manager, our detector is not restricted to run in an SMP environment.

References

[1] W. Gropp, E. Lusk, & A. Skjellum, *Using MPI: Portable parallel programming with the message passing interface*, 2nd ed. (Cambridge, MA: MIT Press, 1999).

[2] M. Snir, S. Otto, S. Huss-Ledeman, D. Walker, & J. Dongarra, *MPI: The complete reference*, 2nd ed. (Cambridge, MA: MIT Press, 1998).

[3] P.S. Pacheco, *Parallel programming with MPI* (San Francisco: Morgan Kaufman, 1997).

[4] Message Passing Interface Forum, http://www.mpi-forum.org/docs/docs.html.

[5] R.H.B. Netzer, Optimal tracing and replay for debugging shared-memory parallel programs, *Proc. ACM/ONR Workshop on Parallel and Distributed Debugging*, San Diego, 1993, 1–11.

[6] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, & T. Anderson, Eraser: a dynamic data race detector for multithreaded programs, *ACM Trans. on Computer Systems, 15*(4), 1997, 391–411.

[7] S. Huband & C. MacDonald, Debugging parallel programs using incomplete information, *Proc. 1st IEEE Computer Society Int. Workshop on Cluster Computing*, Melbourne, Australia, 1999, 278–286.

[8] J. Zhou & K. Tai, Deadlock analysis of synchronous message-passing programs, *Proc. Int. Symp. on Software Engineering for Parallel and Distributed Systems*, Los Angeles, 1999, 62–71.

[9] G. Luecke, Y. Zou, J. Coyle, J. Hoekstra, & M. Kraeva, *Deadlock detection in MPI programs*, Concurrency and Computation: Practice and Experience, *14*(11), 2002, 911–932.

[10] J.S. Vetter & B.R. de Supinski, Dynamic software testing of MPI applications with Umpire, *Proc. Supercomputing 2000*, Dallas, TX, 2000.

[11] MPICH Test Suite available at http://www-unix.mcs.anl.gov/mpi/mpi-test/tsuite.html.

[12] NAS Parallel Benchmarks available at http://www.nas.nasa.gov/Software/NPB.

[13] R. Reusser, P. Sanders, L. Prechelt, & M. Muller, SKaMPI: A detailed, accurate MPI benchmark, *Proc. EuroPVM/MPI,* University of Liverpool, UK, 1998, 52–59.

[14] http://web.unbc.ca/~haque/mpidd_perf.

Biography

Waqar Haque is Associate Professor and Chair of the Department of Computer Science, University of Northern British Columbia (UNBC), Canada. Haque joined UNBC in 1995 after completing a post-doctoral fellowship with the High Performance Computing Group at Iowa State University, USA. During that time, he worked on various projects for Hitachi Data Systems, IBM, and Cray Research. His research interests include high-performance computing, real-time transaction processing, software design and testing, VLDBs, and e-commerce applications. Haque received his M.Sc. and Ph.D. in computer science from Iowa State University. He is a member of IEEE Computer Society and IASTED and also serves on IEEE Technical Committees on Parallel and Distributed Computing and High Performance Computing.