Exascale Message Passing Interface based Program Deadlock Detection

Raed Al Dhubhani*, Fathy Eassa*, Faisal Saeed**
* Faculty of Computing and Information Technology, King Abdul-Aziz University, KSA
** Faculty of Computing, Universiti Teknologi Malaysia, Malaysia

Article Info

Article history:

Received Oct 4, 2015
Revised Dec 27, 2015
Accepted Jan 16, 2016

Keyword:

Deadlock detection
Exascale systems
Message Passing Interface

ABSTRACT

Deadlock detection is one of the main issues of software testing in High Performance Computing (HPC) and also inexascale computing areas in the near future. Developing and testing programs for machines which have millions of cores is not an easy task. HPC program consists of thousands (or millions) of parallel processes which need to communicate with each other in the runtime. Message Passing Interface (MPI) is a standard library which provides this communication capability and it is frequently used in the HPC. Exascale programs are expected to be developed using MPI standard library. For parallel programs, deadlock is one of the expected problems. In this paper, we discuss the deadlock detection for exascale MPI-based programs where the scalability and efficiency are critical issues. The proposed method detects and flags the processes and communication operations which are potential to cause deadlocks in a scalable and efficient manner. MPI benchmark programs were used to test the proposed method.

Corresponding Author:

Raed Al Dhubhani,
Faculty of Computing and Information Technology,
King Abdul-Aziz University, KSA
Email: raedsaeed@gmail.com

1. INTRODUCTION

Exascale Computing is considered as one of the recent research topics in HPC computing area. Exascale computing refers to the capability to process 1 exaflop (10^{18} floating point operations per second). The computation capability of the current supercomputers is in the petaflops level, where 1 petaflop is equivalent to 10^{15} floating point operations per second. Manufacturing machines with this ambitious computation capability depends on using hundreds of millions of cores to achieve that computational target, which are expected to be in operation in 2020 [1]. The scientific and big data processing applications are planned to be run in these machines. So, one of the challenges is how to develop reliable applications for this parallel-based computation environment.

MPI is a standard library which is frequently used in the HPC. It is a standard library for HPC, which is considered by [2] as the de facto standard for parallel programming in the HPC. According to [3], MPI provides a set of functions or commands which are used by the parallel programs to facilitate the communication between the processes in the runtime. The simple scenario of using the MPI library by a parallel program is achieved by using the MPI_Send operation by one process to send a message to another one in the same program, where the destination process receives the message using the MPI_Recvoperation. To provide a rich communication environment for the applications, MPI library provides two different types of communication: blocking and non-blocking communication. In the blocking communication, the sender and receiver must wait for the communication operations to match each other before theycan proceed to execute the next instruction.In the non-blocking communication, the sender and receiver can issue the operations of the communication –MPI_Isend and MPI_Irecv- and proceed directly to execute the next operation without waiting for the communication operationsto match. The result of the non-blocking communication can be checked later by using the MPI_Test operation to check whether the issued non-blocking operation is already matched to a corresponding operationor not. MPI_Wait operation can also be used to enforce the process to wait for a non-blocking communication operation to finish, and once that communication operation is matched to an appropriate operation, the process can continue its execution.

MPI provides also the wildcard receive feature MPI_Source_Any, such that the process can receive the message from any source, which leads to the message race. Because of this message race, the execution of these MPI-based programs is considered as nondeterministic, which means that the behavior of the program during the runtime may differ from one run to another.

As a result of the non deterministic execution, MPI program testing is not an easy task. As similar to the sequential programs, the parallel programs have the same types of programming errors, like buffer overflow, division by zero, etc. In addition, the parallel programs have errors related to the concurrent communication process among their different processes.

Communication deadlock is one of these parallel-based errors, where one process is executing a communication operation –send or receive-, and this process doesnot find a match for that communication operation, and this leads to a communication deadlock. Therefore, MPI application developers need a mechanism to detect any potential deadlock in their applications. For exascale programs whichare expected to consist of millions of processes communicating with each other, detecting the deadlock requires scalable and efficient techniques. This paper presents a scalable and efficient method for communication deadlock detection for exascaleMPI-based programs.

2. RELATED WORK

In [4], In-Situ Partial (ISP) is a deadlock detection tool which is implemented by investigating all the possible interleaving in the MPI program by running the program multiple times. This tool provides complete coverage for all possible execution paths, and hence provides a guaranteed result for the deadlock detection.

According to [5], this tool produces an exponential number of communication interleaving cases which makes it difficult to test MPI programs that have a large number of processes.

In [6], AND-OR Wait-For graph (WFG) is used to detect deadlocks in MPI programs. The ‘AND’ operation is used in this graph to represent the communication pair which is required to match each other. On the other hand, the ‘OR’ operation is used to represent the communication expected between sender nodes and a receiver node which hasthe wildcard receive feature. According to [7], using AND-OR WFG for deadlock detection is time consuming and requires high performance.

A modified version of AND-OR graph is used for the deadlock detection [5]. Marmot Umpire Scalable Tool (MUST) provides a scalable and efficient technique for deadlock detection. However, it doesnot provide complete support for testing the MPI programs which have wildcard receives [8].

Model checking technique is used in [9] to detect MPI deadlocks. It explores all the possible matching and interleaving of the tested MPI program and supports the wildcard communication. The limitation of this technique is the need to construct the model of the MPI program manually.

In [10], a deadlock detection technique is suggested which does not require the testing model to be constructed manually. But, the limitation of this technique is the need to re-run the entire program many times to detect the deadlocks.

Therefore, deadlock detection in exascale MPI-based program requires an efficient and scalable technique which should be able to test millions of processes created by the program. Current deadlock detection techniques of MPI-based programs suffer from the exponential growth of the number of possible communication interleaving cases which makes the testing process of exascale MPI-based program impractical. For the other techniques which do not suffer from this exponential growth, incomplete support for the MPI communication features is provided, or the testing model construction is done manually. In both cases, deadlock detection for exascale MPI-based program cannot be achieved with these limitations.

The motivation for this research is to provide a scalable and efficient technique which does not suffer from the exponential growth of the number of possible communication interleaving cases. At the same time, the technique should have a complete support for the MPI communication features, and the ability to run the deadlock detection process without the need to construct a manual testing model.

3. METHOD

This paper presents Exascale MPI-based Program Deadlock Detection (EMPDD) as a scalable and efficient method for detecting MPI deadlocks in the O(m*n) magnitude, where m is the number of processes in the program, and n is the number of communication operations in each process. The EMPDD method is a static-based, and supports the wildcard receives. In addition, it investigates all the possible matching and interleaving for the MPI program communication operations.

The EMPDD method consists of three algorithms: MatchProcesses, MatchOperations and DetectDeadlocks. MatchProcesses algorithm is used to apply the matching rules between the different processes of the program. To investigate the possible matching between each send operation and all the potential receive operations, the MatchOperations algorithm is used. After the processes and operations matching are done, the DetectDeadlocks algorithm is used to flag all the possible deadlocks based on the produced potential matches.

In comparison to the MPI deadlock approaches which provide all interleaving cases investigation with order of exponential magnitude, EMPDD method is considered more efficient and scalable. In addition, it does not suffer from the problem of the optimized approaches which try to minimize the order of magnitude required to detect the deadlock by limiting the number of possible interleaving visited, where such limitation does not provide complete coverage to the possible execution paths.

The limitation of EMPDD method is the lack of providing a graphical notation for the execution paths which lead to the deadlock. Instead, it is capable to flag all the processes and communication operations which are responsible to produce the deadlock case. However, EMPDD method is useful to present an efficient and scalable approach to check the existence of deadlock in the MPI programs that consist of millions of processes, which is the case of the exascale MPI-based programs.

The EMPDD method includes four stages: 1) extracting the MPI program communication log file 2) parsing the MPI program communication log file 3) matching the communication operations 4) deadlock detection.

The architecture of EMPDD is shown in Figure 1.

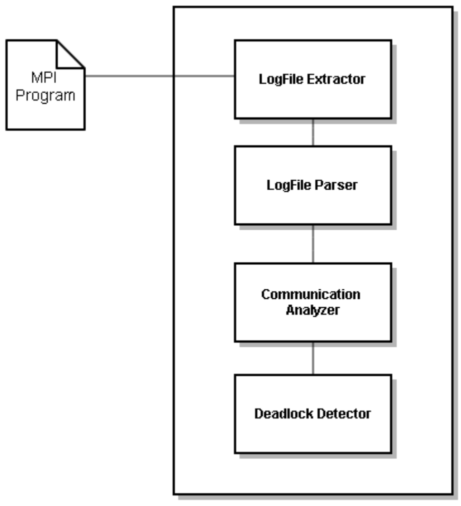

Figure 1. EMPDD Architecture

Algorithm 1: MatchProcesses

Definitions:

1. p: MPI process
2. Processes = {p_1,p_2,p_3, .., p_m} where Processes represents the set of MPI program processes
3. op: communication operation
4. p = {op_1, op_2, op_3, ..., op_n}, where op_i is communication operation
5. op = {OperationType, SourceProcess, DestinationProcess, IsMatched, Counter, IsDeadlock} where:
    a. OperationType$\in$ {Send, Isend, Recv, Irecv}
    b. SourceProcess$\in$ {p_1, p_2,… , p_m}
    c. IsMatched$\in$ {true, false}
    d. IsDeadlock$\in$ {true, false}
6. For each process p, Pointers = {$pointer_1$, $pointer_2$, …, $pointer_m$} where $pointer_i$ is the index of the next operation of p_i to match with respect to p

```
Input:
Processes = {$p_1$,$p_2$,$p_3$, …, $p_m$}
Output:
Processes (after matching)

Steps:
1. Initialization:
    a. for each process p in Processes
        i. initialize Pointers of p to zeros
        ii. for each operation op in p
            1. op.IsMatched = false
            2. op.Counter = 0
            3. op. IsDeadlock = false
2. for each process p in Processes
    a. set currentProcess = p
    b. for each operation op in currentProcess
        i. set currentOperation = op
        ii. set destProcess = Process which currentOperation wants to match
        iii. set currentPointers = Pointers of currentProcess
        iv. set destPointer = pointer of destProcess in currentPointers
        v. if currentOperation is Recv and matched: move to the next operation
        vi. if currentOperation is Recv and not matched: exit the current process and move to
            the next process
        vii. if currentOperation is Irecv: move to the next operation
        viii. if currentOperation is Send:
            1. result = MatchOperations(currentOperation, destPointer)
            2. if result = true: move to the next operation
            3. if result = false: exit the current process and move to the next process
        ix. if currentOperation is Isend:
            1. MatchOperations(currentOperation, destPointer)
            2. move to the next operation
3. return Processes
```

```
Algorithm 2: MatchOperations
Definitions:
1. op: communication operation
2. p = {$op_1$, $op_2$, $op_3$, ..., $op_n$}, where $op_i$ is communication operation
3. op = {OperationType, SourceProcess, DestinationProcess, IsMatched, Counter, IsDeadlock} where:
    a. OperationType$\in$ {Send, Isend, Recv, Irecv}
    b. SourceProcess$\in$ {$p_1$, $p_2$,… , $p_m$}
    c. IsMatched$\in$ {true, false}
    d. IsDeadlock$\in$ {true, false}
4. For each process p, Pointers = {$pointer_1$, $pointer_2$, …, $pointer_m$} where $pointer_i$ is the index of the next
   operation of $p_i$ to match with respect to p

Inputs:
sourceOperation
destPointer
Output:
MatchingStatus$\in$ {true, false}

Steps:
1. Set flag = false
2. while (flag = false)
    a. set currentOperation = destinationProcess[destPointer]
    b. if currentOperation.OperationType = Send and currentOperation.IsMatched = false:
            exit the while loop
```

```
        c. if currentOperation.OperationType = Send and currentOperation.IsMatched = true:
                destPointer++
        d. if currentOperation.OperationType =  ISend:
                destPointer++
        e. if currentOperation.OperationType = Recv and currentOperation.IsMatched = true and
           currentOperation.DestProcess != MPI_Source_Any:
                destPointer++
        f. if currentOperation.OperationType =  Irecv and currentOperation.IsMatched = true and
           currentOperation.DestProcess != MPI_Source_Any:
                destPointer++
        g. if currentOperation.OperationType = Recv:
                i. if currentOperation.DestProcess = MPI_Source_Any
                        1. sourceOperation.IsMatched = true
                        2. currentOperation.IsMatched = true
                        3. currentOperation.Counter ++
                        4. destPointer++
               ii. if currentOperation.DestProcess != sourceOperation.DestProcess:
                        exit the while loop
              iii. if currentOperation.DestProcess = sourceOperation.DestProcess:
                        1. sourceOperation.IsMatched = true
                        2. currentOperation.IsMatched = true
                        3. currentOperation.Counter ++
                        4. destPointer++
                        5. flag = true
                        6. if currentOperation.Counter = 1
                                exit the loop
        h. if currentOperation.OperationType = Irecv:
                i. if currentOperation.DestProcess = MPI_Source_Any
                        1. sourceOperation.IsMatched = true
                        2. currentOperation.IsMatched = true
                        3. currentOperation.Counter ++
                        4. destPointer++
               ii. if currentOperation.DestProcess != sourceOperation.DestProcess:
                        destPointer++
              iii. if currentOperation.DestProcess = sourceOperation.DestProcess:
                        1. sourceOperation.IsMatched = true
                        2. currentOperation.IsMatched = true
                        3. currentOperation.Counter ++
                        4. destPointer++
                        5. flag = true
                        6. if currentOperation.Counter = 1
                                exit the loop
3. return flag
```

Algorithm 3: DetectDeadlocks

<u>Definitions:</u>

1. p: MPI process
2. Processes = {p_1,p_2,p_3, .., p_m} where p_i represents a process and Processes represents the set of MPI program processes
3. op: communication operation
4. p = {op_1, op_2, op_3, ..., op_n}, where op_i is communication operation
5. op = {OperationType, SourceProcess, DestinationProcess, IsMatched, Counter, IsDeadlock} where:
    a. OperationType∈ {Send, Isend, Recv, Irecv}
    b. SourceProcess∈ {p_1, p_2,… , p_m}
    c. IsMatched∈ {true, false}
    d. IsDeadlock∈ {true, false}

```
Input:
Processes
Output:
Processes (after marking the potential deadlocks operations)
Steps:
1. for each process p in Processes
    a. set currentProcess = p
    b. for each operation op in currentProcess
        i. set currentOperation = op
        ii. if currentOperation.IsMatched = false
                currentOperation.IsDeadlock = true
        iii. if (currentOperation.OperationType = Recv or currentOperation.OperationType =
             Irecv) and currentOperation.Counter> 1
            1. Count = number of operations in the currentProcess which have the same
               destination
            2. If currentOperation.Counter> Count
                    currentOperation.IsDeadlock = true
2. return Processes
```

4. RESULTS AND DISCUSSION

To evaluate EMPDD method, four benchmark MPI programs were used to check its capability to detect deadlocks. The four benchmark programs are: Diffusion, DTG, Integrate, and Floyd.

Diffusion program is an MPI program which solves the 2-dimensional diffusion equations.There are 40 communication operations in this program.This program does not contain wildcard receive operations, but it has barrier operations. The result of applying MatchProcesses and MatchOperations algorithms of the EMPDD method leads to matching all the communication operations of the program to the corresponding operation. DetectDeadlock algorithm identifies no deadlocks in the program. The communication operations of the program are shown in Table 1.

The second benchmark program is DTG programwhich is an MPI dependence transition group program. It has 10 communication operations. There are 3 wildcard receive operations. Applying MatchProcesses and MatchOperations algorithms for this program shows that all the communication operations are matched. Although all the communication operations of the program are matched, DetectDeadlock algorithm shows that there is a potential deadlock related to the first send operation in process 1. This send operation matches the receive operation of process 0 that has two wildcard receive operations. For these two receive operations, there are two corresponding send operations: one in process 1 and the other in process 2. In one of the potential interleaving scienarios, the send operation of process 1 matches the first receive operation of process 0. Hence, the send operation of process 2 matches the second receive operation of process 0. In this scienario, there is no deadlock situation, when each communication operation matches its corresponding operation, and the program terminates normally. For the second potential interleaving scienario, the send operation of process 2 matches the first receive operation of process 0. The result of this match is the send operation of process 1 needs to match the second receive operation of process 0. But the second receive operation of process 0 comes after the send operation which needs to match the receive operation of process 3. At this moment, process 3 can not match its receive operation to the send operation of process 0 because it is waiting to match its receive operation with process 1. So, it is clear that there is a deadlock in this situation. So, DetectDeadlock algorithm reports the send operation of process 1 as an operation may lead to deadlock.The communication operations of the program are shown in Table 2.

Integrate program is an MPI program which calculates the integral value of cosine or sin function for a given range. There are 60 communication operations,15 of them are wildcard receive operations. Each communication operation in the program matches its cooresponding operation, so this program has no deadlock.The communication operations of the program are shown in Table 3.

The last benchmark program is Floyd program which uses the Floyd-Warshall algorithm to solve the problem of all-pairs shortest-path. It has 1400 communication operations, and it contains 700 wildcard receive operations. The program has no deadlock, and testing it by EMPDD does not report any deadlock situation. Due to the large number of communication operations of this program, it is not feasible to show them in a table.

The experimental results showed that EMPDD could successfully detected the deadlockin the DTG benchmark program. For the other benchmark programs which donot contain deadlocks, EMPDD reports them as deadlock free.

Table 1. Communication operations of Diffusion MPI program

P0	P1	P2	P3
Recv(1,1)	Send(0,1)	Recv(3,1)	Send(2,1)
Send(1,2)	Recv(0,2)	Send(3,2)	Recv(2,2)
Recv(3,3)	Send(2,3)	Recv(1,3)	Send(0,3)
Send(3,4)	Recv(2,4)	Send(1,4)	Recv(0,4)
Barrier()	Barrier()	Barrier()	Barrier()
Recv(1,1)	Send(0,1)	Recv(3,1)	Send(2,1)
Send(1,2)	Recv(0,2)	Send(3,2)	Recv(2,2)
Recv(3,3)	Send(2,3)	Recv(1,3)	Send(0,3)
Send(3,4)	Recv(2,4)	Send(1,4)	Recv(0,4)
Barrier()	Barrier()	Barrier()	Barrier()

Table 2. Communication operations of DTG MPI program

P0	P1	P2	P3	P4
Recv(*,0)	Send(0,0)	Recv(*,0)	Recv(1,0)	Send(2,0)
Send(3,0)	Send(3,0)	Send(0,0)	Recv(0,0)	
Recv(*,0)				

Table 3. Communication operations of Integrate MPI program

P0	P1	P2	P3	P4	P5	P6	P7
Send(1,1)	Recv(0,*)	Recv(0,*)	Recv(0,*)	Recv(0,*)	Recv(0,*)	Recv(0,*)	Recv(0,*)
Send(2,1)	Send(0,3)	Send(0,3)	Send(0,3)	Send(0,3)	Send(0,3)	Send(0,3)	Send(0,3)
Send(3,1)							
Send(4,1)							
Send(5,1)							
Send(6,1)							
Send(7,1)							
Send(8,1)							
Send(9,1)							
Send(10,1)							
Send(11,1)							
Send(12,1)							
Send(13,1)							
Send(14,1)							
Send(15,1)							
Recv(*,3)							
Recv(*,3)							
Recv(*,3)							
Recv(*,3)							
Recv(*,3)							
Recv(*,3)							
Recv(*,3)							
Recv(*,3)							
Recv(*,3)							
Recv(*,3)							
Recv(*,3)							
Recv(*,3)							
Recv(*,3)							
Recv(*,3)							
Recv(*,3)							

P8	P9	P10	P11	P12	P13	P14	P15
Recv(0,*)	Recv(0,*)	Recv(0,*)	Recv(0,*)	Recv(0,*)	Recv(0,*)	Recv(0,*)	Recv(0,*)
Send(0,3)	Send(0,3)	Send(0,3)	Send(0,3)	Send(0,3)	Send(0,3)	Send(0,3)	Send(0,3)

5. CONCLUSION

Deadlock detection for MPI programs is very important. There are many approaches which can detect the deadlocks of the MPI programs. Although some of them provide complete coverage for the possible execution paths, they are not efficient and cannot be used for complicated MPI programs.

Alternative approaches solve the scalability problem and provide efficient performance to detect the MPI deadlock by limiting the number of the investigated execution paths. The cost of such limitation is the lack of the guarantee that all the execution paths are visited, and hence there is no guarantee that all the possible deadlocks are detected. In this paper, we presented an efficient and scalable method for deadlock detection in exascale MPI-based programs. The proposed method (EMPDD) is implemented to detect and flag the processes and communication operations which are potential to cause deadlocks. The limitation of this method is its lack to specify the execution paths which lead to the deadlock.

ACKNOWLEDGEMENTS

This work is supported by the Research Management Centre (RMC) at the Universiti Teknologi Malaysia (UTM) under Research University Grant Category (Cost Center No: Q.J130000.2728.01K73).

REFERENCES

[1] Greenough, C., Worth, D.J. and Chin, L.S. “Thoughts on Software Engineering for ExaScale Software Development,” Science and Technology Facilites Council, UK, 2011.
[2] Fu, X., Chen, Z., Zhang, Y., Huang, C., Dong, W. and Wang, J. “MPISE: Symbolic Execution of MPI Programs,” Cornell University Library, Ithaca, NY, USA, 2014.
[3] Message Passing Interface. [Online] www.mpi-forum.org/docs/mpi-3.0.
[4] Vakkalanka, S., Sharma, S., Gopalakrishnan, G. and Kirby, R. “ISP: A Tool for Model Checking MPI Programs,” In Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming, ACM New York, NY, USA, 2008.
[5] Hilbrich, T., Protze, J., Schulz, M., Supinski, B. and Müller, M. “MPI Runtime Error Detection with MUST: Advances in Deadlock Detection,” In Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, IEEE Computer Society Press, Los Alamitos, CA, USA, 2012.
[6] Hilbrich, T., Supinski, B., Schulz, M. and Müller, M. “A Graph Based Approach for MPI Deadlock Detection,” In Proceedings of the International Conference on Supercomputing, ACM New York, NY, USA, 2009.
[7] Deniz, E., Sen, A. and Holt, J. “Verification and Coverage of Message Passing Multicore Applications,” Journal of ACM Transactions on Design Automation of Electronic Systems, vol. 17, pp. 1-31, 2012.
[8] Forejt, V., Kroening, D., Narayanaswamy, G. and Sharma, S.*Precise Predictive Analysis for Discovering Communication Deadlocks in MPI Programs.*Springer International Publishing Switzerland, pp. 263-278, 2014.
[9] Siegel, S. *Model Checking Nonblocking MPI Programs*. Springer Verlag Berlin Heidelberg, pp. 44-58, 2007.
[10] Vo, A., Aananthakrishnan, S., Gopalakrishnan, G., Supinski, B., Schulz, M., and Bronevetsky, G. A Scalable and Distributed Dynamic Formal Verifier for MPI Programs. In Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, 2010.