Available Online at www.ijcsmc.com

International Journal of Computer Science and Mobile Computing

A Monthly Journal of Computer Science and Information Technology

ISSN 2320–088X
IMPACT FACTOR: 6.017

IJCSMC, Vol. 7, Issue. 4, April 2018, pg.141 – 147

Integrating Static and Dynamic Analysis Techniques for Detecting Dynamic Errors in MPI Programs

[1]Rana A. Alnemari, [2]Mai A. Fadel, [3]Fathy Eassa

Faculty of Computing and Information Technology, King Abdul-Aziz University, KSA
[1] ranaalnemary@gmail.com; [2] mfadel3@yahoo.com; [3] fathy55@yahoo.com

Abstract*: Message passing interface is the de-facto standard for programming high performance computing applications and it is ready for scaling to extreme scale system with millions of nodes and billions of cores. With this huge number of components MPI will be error prone. Many types of errors can occur with MPI implementation such as deadlock. Testing tools can assist application developers in the detection of such errors. This thesis provides an integration of static and dynamic analysis to enable a precise testing of MPI applications. This paper presents a new algorithm that is able to detect deadlock for point –to-point communication in two phases. By using static phase the overhead in this dynamic phase will limited. The experimental results show that our verification tool verifies several benchmarks and finds deadlock.
Keywords— High performance computing – Message passing interface –Static testing –Dynamic testing

I. INTRODUCTION

Parallel and distributed computing environments are becoming increasingly common. These environments require parallel programming models and tools to enable the development of application software that uses the hardware resources effectively. Message passing interface (MPI) has become one of the more commonly used models for application development in such environments. MPI is the de-facto standard for programming HPC (High Performance Computing) applications, most of recent HPC systems implement the MPI as the part of their software [1]. MPI is ready for scaling to extreme scale systems like exascale systems. Exascale systems refer to computers that can perform 10^{18} floating point operations per second which are predicted to emerge with several hundred thousand nodes and each node itself may have large numbers of concurrent cores/threads [2],[10].Achieving the expected level of performance in future exascale systems will require many millions of components, like increases in communication networks, in memory modules, and in storage devices. With this huge growth in the number of components the number of faults will increase .MPI implementation in this high-performance computing is error prone and testing tools can greatly increase MPI programmers' productivity. Many types of errors can occur with MPI implementation including errors in type matching, invalid arguments, race conditions, deadlocks and portability errors. Unfortunately, debugging MPI can be difficult. Some accounts of HPC software development report that testing and debugging can consume almost 50% of application development time [3] The main value of the Testing and model checking is to find errors in programs. Further,

© 2018, IJCSMC All Rights Reserved

if no errors are to be found these techniques will increase the confidence that the program is correct. This paper presents an integration of static and dynamic analysis tool that is able to detect deadlock in two phases: the first phase statically verifies the program at the source code level and outlines all execution paths that may leading to actual and potential deadlock. With this information of this phase the program transformed to dynamic phase to check only execution paths with potential errors instead of checking all processes. For that the overhead in this phase is limited.
The rest of the paper is organized as follows: Section II discusses the related work; Section III presents the background of deadlock in point-to-point communication, Section IV presents the proposed framework; Section gives the experimental results; and Section VI is the conclusion.

II. RELATED WORK

There are a number of tools that can detect MPI errors in large scale systems. These tools can be classified under following analysis: Static Analysis, Dynamic Analysis and model based

A. *Static Analysis*

The static analysis is a process of source code without real running the program. It can be useful to present detailed and full coverage of the analyzed code. This approach can detect errors that may not manifest during real program execution. Static analysis suffers from false positive report. There are many approaches like TASS [11, 12], MPI- checker [13] use this analysis to detect errors in MPI applications. TASS rely on symbolic execution and explicit state space enumeration. TASS can check a number of properties as it explores the state space: absence of deadlocks (actual or potential), buffer overflows, reading uninitialized variables, division by zero, memory leaks, and assertion violations. MPI-Checker uses Clang’s Static Analyzer [14]. It employs a combination of approaches to perform static analysis of c \c++ code: path sensitive and non-path sensitive checks. MPI- Checker’s path- sensitive perform a full symbolic execution of the code. MPI- Checker’s non-path-sensitive checks are based on traversing the abstract syntax tree (AST). MPI-Checker can detect many errors like unmatched point-to-point calls, unreachable calls, type mismatch, invalid argument type, collective call in rank branch, double request usage of non-blocking calls and missing wait errors.

B. *Dynamic analysis*

Dynamic software analysis approaches execute an instrumented MPI application, observe MPI operations at runtime, and then apply correctness checks to this data. There are many approaches like MPI-check [15, 16], Umpire [5], Marmot [6], Must [7,8] use this analysis to detect errors in MPI applications.
MPI-check is automatically finds many of the errors made when writing F77 and F90 MPI programs. MPI-check performs run-time checking by instrumenting the original program and then compiling and linking the resulting program to produce an instrumented executable. The detection of deadlocks on the base of a decentralized handshaking approach was added in MPI-check 2. It can detect “actual” and “potential” deadlock situations involving blocking and non-blocking point-to-point MPI routines and also deadlocks caused by the incorrect use of collective MPI routines. In Umpire MPI call information is collected using a central manager and checked with a verification algorithm. The central manager controls the execution of the MPI program and communicates with all MPI processes via its shared memory buffers. Umpire’s deadlock detection function tracks blocking point-to-point and collective MPI communication calls and detects cycles in dependency graphs prior to program execution by using dependency graph and timeout mechanism. Umpire lack of the scalability that needed for large High-performance computing systems. Marmot checks conformance to MPI-1.2 standard for both C and F90 codes. It checks all the parameters passed to MPI calls and detects usage errors like deadlocks. In order to detect global errors like deadlocks an additional MPI process is used. Marmot does not require source code modification, it works as a wrapper for compiler and automatically performs instrumentation of the source code. MUST overcomes the scalability Limitations of Umpire and Marmot by relying on a tree-based layout [9]. Must can detect wide range of MPI usage errors such as MPI datatype matching violations, incorrect communication buffer usage and deadlocks. For deadlock detecting Must combine a time out, graph based deadlock and transition system.

C. *Based analysis*

It is a methodology in which test cases are gotten from a model that describes the program under test? The most model checking tools operate on model of the program, not at on the program code itself. For MPI-based programs, such method would require that programmers build, either manually or automatically, a model of their applications in a special language and then verify these models against various properties. There are many

© 2018, IJCSMC All Rights Reserved

approaches like MPI-spin [18] and ISP [17] use this analysis to detect errors in MPI applications. MPI-Spin is an extension of the popular model checker Spin. It adds to Spin's input language a number of commonly-used MPI functions, types, and constants for modeling parallel programs that use the Message Passing Interface. It contains functions used for blocking and non-blocking communications. The algorithm used in MPI-spin can point out deadlocks, assertion violations, MPI object leaks. ISP extracts automatically the model during run time. ISP detects all deadlocks, and local assertion violations in MPI programs. Also, it can detect MPI object leaks and communication races. ISP uses MPI profiling interface to intercept MPI operations and to enforce particular outcomes for non-deterministic ones. The ISP scheduler uses an MPI-semantics aware algorithm that reorders MPI operations before sending them into the MPI runtime. This allows ISP to fully explores the area of non-deterministic outcomes for a given input, although the control flow decisions related to that input still limit overall testing coverage. The disadvantage of ISP is low scalability of its centralized scheduling algorithm and applying ISP to large process number is infeasible.

our tool is hybrid analysis. We integrate static analysis with dynamic analysis to take advantages and avoid disadvantages from every one of them. Static MPI can in fact considerably save computer resources during the runtime since no additional overhead is induced during the program execution. However, it is a challenge to obtain all the required information for the checking algorithm only during compile time. And it suffers from false positive.

On the other hand, dynamic MPI checking approaches can easily provide all the required information since this data is already available whenever processes communicate with each other peers. However, injecting the checking algorithm during the execution of the MPI programs affecting the performance of the running code.

III. DEADLOCK IN POINT-TO-POINT COMMUNICATION

In the context of this paper deadlock is defined as “a situation in which each member process of the group is waiting for some member process to communicate with it, but no member is attempting to communicate with it” [19] deadlock may be actual deadlock that should occur or potential deadlock that occurs in some case depend on implementation. There are categories of deadlock situations that occur when using point-to-point as following:

A. **Deadlock due to send receive mismatch**:
The number of send and receive calls is not the same. some calls wait to match another call that will not post. Therefore, the execution deadlocks.

B. **Head-to-head deadlock:**
First situation, if two processes issue send calls to each other before receiving calls that causes potential deadlock (if the size of MPI runtime buffer is insufficient or if standard send is implemented as a not buffer send). Second situation, if two processes issue receive calls that only complete and return to overflow if a matching sends calls issued (that causes an actual deadlock). In fig.1 The first process cannot return from the sending procedure, because the second does not issue the receive procedure, and the second does the same. Arising of deadlock in such case depends on implementation of sending procedure.

Process 0	Process 1
MPI_SEND (1) MPI_RECV (1)	MPI_SEND (0) MPI_RECV (0)

Fig. 1. Head-to-head deadlock (send-send deadlock)

In fig.2 both processes first issue the blocking procedure MPI_RECV, and then proceed to send the data, the deadlock will actually arise because the receive calls wait send calls that will never issue.

Process 0	Process 1
MPI_RECV (1) MPI_SEND (1)	MPI_RECV (0) MPI_SEND (0)

Fig. 2. Head-to-head deadlock (Recv-Recv deadlock)

© 2018, IJCSMC All Rights Reserved

IV. PROPOSED FRAMEWORK

Our tool detects deadlock in two phases: Static phase and Dynamic tool.

A. *Static phase:*

The main idea of detecting deadlock in MPI calls using static analysis is to use stacks to arrange the MPI calls in each rank case. Afterwards, trying to match MPI calls located at different stacks. first trying to Match call with a one located at the top of another stack. If there is no matching, trying to match it with a one that is not at the top of any stacks. In the latter case if a matched call is found that is not at the stack top, it has to match all the calls located at the top of the found one before deciding that found call can match the original one. We create a number of stacks equal to the number of processes. The last requirement is challenging when considering the static checking because we do not have a direct access of the number of processes during compile time. There is a valuable information inside the MPI programs that can help us in achieving this goal. This valuable information is hide inside each MPI case. Generally, when an MPI program is written, it is usually subdivided into a number of MPI cases in the form of (if ... else statement) as presented in figure 3. Each conditional statement that include a rank inside its condition can be considered as an MPI case. Moreover, each MPI call is associated with an MPI case such that we can extract all the number of calls that belong to the same MPI case. Eventually each MPI case will represent an MPI process. After we have identified all or most of the process via their corresponding MPI case. We insert all the MPI calls belonging to the same process into its corresponding stack. However, they are inserted in the reverse order of their calls. That is the first MPI call will be the top call in the stack while the last function call will be the first function in the stack. Figure 4 represent as schematic diagram of the stack representation of the MPI processes in Fig. 1.

```
#include <stdio.h>
#include <mpi.h>
void main (int argc, char **argv) {
int myrank;
MPI_Status status;
double a [100], b [100];
MPI_Init (&argc, &argv); /* Initialize MPI */
MPI_Comm_rank (MPI_COMM_WORLD, &myrank); /* Get rank */
if (myrank == 0) {
MPI_Send (a, 100, MPI_DOUBLE, 1, 17, MPI_COMM_WORLD);
MPI_Recv (b, 100, MPI_DOUBLE, 1, 19, MPI_COMM_WORLD, &status);
}
else if (myrank == 1) {
MPI_Recv (b, 100, MPI_DOUBLE, 0, 17, MPI_COMM_WORLD,
MPI_Send (a, 100, MPI_DOUBLE, 0, 19, MPI_COMM_WORLD);
&status);
}
MPI_Finalize (); /* Terminate MPI */
}
```

Fig. 3. An example of MPI program

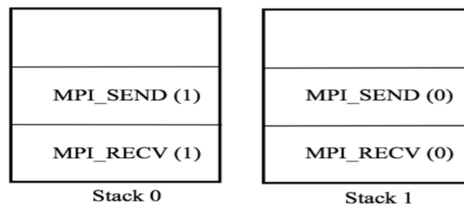

Fig.4. Stack representation of Example in Fig.2

1) Stack scenarios

When trying to search match the stack calls, we may encounter one of the following scenarios:

- No deadlock exists

  This happens when all the stacks are empty. The stacks will be empty if there is no MPI calls in the individual MPI rank cases or the deadlock detection algorithm has popped all the MPI calls from the stack after each MPI send/receive calls match a corresponding MPI calls.

© 2018, IJCSMC All Rights Reserved

- Explicit deadlock exists

  This can happen in point to point communication when all the top calls in the stacks are MPI receive/send calls (Fig.1, Fig.2). In this case the MPI program will not be able to make any further executions

- No direct match exists

  This the main step of the deadlock matching algorithm. When we arrive to this case it means that we are not able to match send/receive calls in the top of the stack with receive/send calls on the top of the stack. So, we have to search the stacks for matching receive calls. If one is found, we have to match first the send calls above it or a deadlock will be found if we could not do such match. This is mandatory since our target might be hidden inside other calls in the stack. To match the found call in the middle of the stack we have to repeat the search process using all the previous steps. Moreover, when try to match the candidate call, may also come again to this step and repeat the recursion process. It can happen that we are not able to match this call in the middle of the stack and in this case, we have to search in the other stacks. If we search all stacks and there is no match, we return a failure code.

B. Dynamic phase

In this section, we discuss the dynamic approach where the checking is performed during the runtime. In this case we have all the required information to perform the check. Therefore, this approach would be more powerful than the static one. However, it has the disadvantage of introducing computational overhead each time the program is executed which is solved by predicting potential errors in static phase. so, Dynamic phase only checking execution paths with potential errors instead of checking all paths. for that the overhead in this phase is limited. Dynamic phase checks only execution processes with potential deadlock by using timeout mechanism. user defines limit time to these processes to wait in an MPI call. If this time exceeds limit time defined by user on that processes at the same time, a deadlock warning issue.

V. EXPERIMENTAL RESULT

At compile time, a warning is returned to the programmer when an error situation is detected. User can know what processes have potential or actual errors. At runtime, we have experimented with Umpire [5] test cases, and in tests that have deadlocks, our tool can detect the deadlocks. Table 1 provide the runtime information when the MPI code is executed with and without our tool. From the presented data in this table it can be shown the additional overhead can be accepted. In table 2 we compare our tool with two other tools. The experiments have been designed by repeating it for more than one run. If there is no deadlock on the code under examination then no deadlock will be detected despite repeating the experiment many times. On the contrary, if the code that require checks contains a deadlock, we might need to execute the code and performing several runs until we can detect deadlocks. Thus, during the dynamic check of the MPI code we repeat the experiments a number of time and we count how many of the experiment a deadlock has been detected. This gives us a rough idea of how often the deadlock can occur during the overall number of experiments.

Table 2 summarizes the outcome of the performed experiments. The first column represents the adopted benchmarks, the second column provides the outcome detection when the ISI tool is used, the third columns report the output of the Marmot tool, and the last column gives the outcome of our tool.

Umpire benchmarks	With our tool	Without Our tool	Execution-Time Overhead
basic-deadlock.c	1.87	1.43	30.7%
basic-deadlock-comm_create.c	0.523	0.434	20%
basic-deadlock-comm_dup.c	0.4324	0.324	33%
basic-deadlock-comm_split.c	0.943	0.854	38%

Table1: An execution time overhead

© 2018, IJCSMC All Rights Reserved

Case Study	ISP	Marmot	Our tool
send-recv-ok.c	No deadlock	0/20	0/20
basic-deadlock-comm_create.c	deadlocked	10/10	10/10
basic-deadlock-comm_split.c	deadlocked	10/10	10/10
basic-deadlock.c	deadlocked	10/10	10/10
basic-deadlock-comm_dup.c	deadlocked	10/10	10/10

Table 2: A comparison between a number of tools and our tool

VI. CONCLUSION

In this paper, we presented our approach of designing and implementing an MPI checking technique that employs static as well as dynamic approaches. Our static checking approach is based on the Clang/LLVM framework [14] which intercept the syntax tree during the program compilation. We also propose an efficient approach to trace MPI calls while they are translated by clang by employing the call stack idea to match MPI calls while checking the program statically. To overcome the shortcoming of the static approach we have also implemented a dynamic approach for detecting deadlock. To reduce the computational overhead due to the checking procedure we have combined the two approaches for implementing a hybrid scheme for detecting race conditions in MPI programs. The experimental results show that our tool efficiently verifies several benchmarks and detect deadlock in acceptable overhead.

REFERENCES

[1] Judicael, A., et al. "Extreme-Scale Computing Services over MPI: Experiences, Observations and Features Proposal for Next Generation Message Passing Interface."
[2] Schulte, Michael J., et al. "Achieving exascale capabilities through heterogeneous computing." *IEEE Micro* 35.4 (2015): 26-36.
[3] *FY 1998 Blue Book: Computing, Information, and Communications: Technologies for the 21st Century*. .
[4] Luecke, Glenn R., et al. "Deadlock detection in MPI programs." *Concurrency and Computation: Practice and Experience* 14.11 (2002): 911-932.
[5] Vetter, Jeffrey S., and Bronis R. De Supinski. "Dynamic software testing of MPI applications with Umpire." *Supercomputing, ACM/IEEE 2000 Conference*. IEEE, 2000.
[6] Krammer, Bettina, et al. "MARMOT: An MPI analysis and checking tool." *Advances in Parallel Computing*. Vol. 13. North-Holland, 2004. 493-500.
[7] Hilbrich, Tobias, et al. "MPI runtime error detection with MUST: advances in deadlock detection." *Scientific Programming* 21.3-4 (2013): 109-121
[8] Hilbrich, Tobias, et al. "MUST: A scalable approach to runtime error detection in MPI programs." *Tools for high performance computing 2009*. Springer, Berlin, Heidelberg, 2010. 53-66.
[9] Vakkalanka, Sarvani S., et al. "ISP: a tool for model checking MPI programs." *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*. ACM, 2008.
[10] Sharma, Subodh, Ganesh Gopalakrishnan, and Greg Bronevetsky. "MAPPED: Predictive dynamic analysis tool for MPI applications." *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion.* IEEE, 2012.
[11]T. Hilbrich, M. S. Müller, and B. Krammer, “MPI Correctness Checking for OpenMP/MPI Applications,” *Int. J. Parallel Program.*, vol. 37, no. 3, pp. 277–291, 2009.
[10]“Message Passing Interface Forum. MPI-2: Extensions to the Message-Passing Interface, 1997.” احط الجديد
[11] Siegel, Stephen F., et al. "Combining symbolic execution with model checking to verify parallel numerical programs." ACM Transactions on Software Engineering and Methodology (TOSEM) 17.2 (2008): 10
[12] Siegel, Stephen F., and Timothy K. Zirkel. "Automatic formal verification of MPI-based parallel programs." *ACM SIGPLAN Notices* 46.8 (2011): 309-310
[13] Droste, Alexander, Michael Kuhn, and Thomas Ludwig. "MPI-checker: static analysis for MPI." *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*. ACM, 2015.
[14]Clang. http://clang.LLVM.org/docs/index.html
[15] Luecke, Glenn, et al. "MPI-CHECK: a tool for checking Fortran 90 MPI programs." *Concurrency and Computation: Practice and Experience* 15.2 (2003): 93-100.
[16] Luecke, Glenn R., et al. "Deadlock detection in MPI programs." *Concurrency and Computation: Practice and Experience* 14.11 (2002): 911-932.

© 2018, IJCSMC All Rights Reserved

[17] Hilbrich, Tobias, et al. "Runtime MPI collective checking with tree-based overlay networks." Proceedings of the 20th European MPI Users' Group Meeting. ACM, 2013.
[18] Siegel, Stephen F. "Model checking nonblocking MPI programs." *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer, Berlin, Heidelberg, 2007.
[19] N. Natarajan. A distributed algorithm for detecting communication deadlocks. In Foundations of Software Technology and Theoretical Computer Science, Fourth Conference, Bangalore, India, December 13-15, 1984, Proceedings, pages 119–135, 1984.

© 2018, IJCSMC All Rights Reserved