ISSN 2278-3091
Volume 9, No.5, September - October 2020
International Journal of Advanced Trends in Computer Science and Engineering
Available Online at http://www.warse.org/IJATCSE/static/pdf/file/ijatcse65952020.pdf
https://doi.org/10.30534/ijatcse/2020/65952020

Design and Implementation of a Message Passing Interface (MPI) Dynamic Error Detection System

Rana Abdul Razaq Alnemari[1] , Dr. Mai Fadel[2]

[1]Computer Science Department, Faculty of Computing and Information Technology, King Abdulaziz University, Jeddah, Saudi Arabia, 1ranaalnemary@gmail.com

[2]Computer Science Department, Faculty of Computing and Information Technology, King Abdulaziz University, Jeddah, Saudi Arabia, mfadel@kau.edu.sa

ABSTRACT

This paper presents the design and implementation of Message Passing Interface (MPI) -Dynamic Error Detection mechanisms contributing to the early detection of some scenarios of errors during static analysis by defining a novel representation of the target application based on stack structures. The representation is an extension of the one used by the clang compiler. This fine-grained representation allows for analyzing the flow of concurrent messages being exchanged, which is important for deadlock errors and race conditions detection. We detect these kinds of errors in point-to-point and collective communication. In a broader context, this paper aims to improve the performance of error detection in MPI applications by integrating static analysis and dynamic analysis, where potential problematic constructs that are reported by the static part of our tool are further checked during program execution. Hence, we focus our analysis to consider only the highlighted paths, to be able to reduce time overhead of dynamic analysis. Several mini-programs are selected from a benchmark that contain examples of the errors identified by our work. These mini programs are then tested by our tool. The experimental results show that our tool is capable of finding deadlocks and race conditions.

Key words: Parallel computing, message passing interface, static analysis, deadlocks, race conditions detection.

1. INTRODUCTION

The message passing interface (MPI) has become one of the more commonly used models for application development in high performance computing (HPC). MPI implementation in HPC is error prone, and testing tools can greatly increase MPI programmers' productivity. However, MPI programs can be checked using either a static approach (i.e., during compile time) or a dynamic approach (i.e., during runtime). Static approach can in fact considerably save computing 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. For instance, the process rank cannot be obtained easily in some cases. On the other hand, dynamic MPI analysis 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 affects the performance of the running code. In fact, checking algorithms need to be executed almost during every communication between two or more processes. Despite such an efficiency drawback, the majority of available MPI checkers are based on this dynamic checking method. This paper presents an approach of designing and implementing an MPI checking technique that combines static and dynamic approaches. Our static checking approach is based on the Clang/LLVM [1] framework, which intercepts 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. Moreover, we support detecting basic race conditions that can result from inappropriate structure of the MPI code. To overcome the shortcomings of the static approach, we have also implemented a dynamic approach for detecting deadlock and race conditions. To reduce the computational overhead due to the checking procedure, we have combined the two approaches to implement a hybrid scheme for detecting deadlock and race conditions in MPI programs. The remainder of the paper is organized as follows: Section II presents a detailed description of the types of MPI errors. Section III presents the system architecture design. Section IV presents a description of the static and dynamic analysis. As well as representative parts of the implementation of our tool. Then, section V presents the performance assessment results, and section VI concludes the paper.

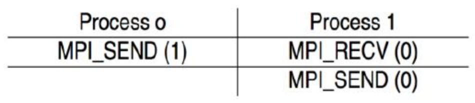
2.MPI DYNAMIC USAGE ERRORS

It is clear, that all debugging problems of sequential codes are inherited by parallel programs. Communication and synchronization of parallel task can be source of additional errors. parallel programs do not always have reproducible behavior and can be completely different from run to run. So, debugging of a parallel program cannot rely on cycling running the code with the comparison of results. The use of MPI is a potential source of additional hard-to-detect errors. There are lot of possible errors related to usage of MPI like deadlocks, race condition erroneous buffer and type mismatches.

Deadlocks typically happen in MPI programs when there is a sender or receiver call not found a match [2]. There are two types of deadlock: actual deadlock that should occur and potential deadlock that occurs in some cases.

The following are the categories of deadlock situations that occur when using point-to-point MPI routines:

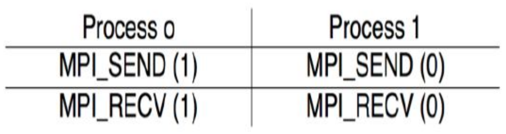
(a) deadlock due to send receive mismatch [3]: the number of send and receive calls is not the same. Figure 1 illustrates a case of actual deadlock, in this example, processes 0 sends message to process 1 and process 1 receives that message. After that process 1 sends message to process 0 but no corresponding call in process 0, leading to a deadlock.



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

Figure 1: Deadlock due to send receive mismatch

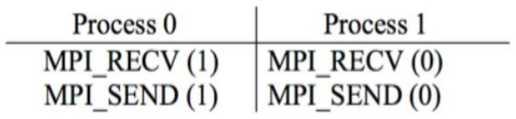
(b) Send -send deadlock [4]: if two processes issue send calls to each other before receiving calls that may leading to deadlock in two cases: if the size of MPI runtime buffer is insufficient or if standard send is implemented as a non-buffer send, but if it implemented in buffered mode and the size of MPI runtime buffer is sufficient this call will not deadlock. Figure 2 illustrates a case of potential deadlock .in this example, both processes of rank 0 and 1 issue send calls to each other, Deadlock may not occur if the call to MPI_send copies the message to a buffer and execution continues, and may it occurs if the size of message buffer bigger than amount of buffering the MPI routine provides. Arising of deadlock in such case depends on implementation of sending procedure (causes a potential deadlock).



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

Figure 2: Send-Send Deadlock

(c) Receive -receive deadlock [4]: if two processes issue receive calls that only complete and return to overflow if a matching sends calls issued, which is never happened as a result all processes wait for each other send calls in a cycle, and for that can't continue execution (causes an actual deadlock). Figure 3 illustrates a case of actual deadlock .in this example, both processes of rank 0 and 1 issues receive operations before send operations since no send operation is available. Both processes will wait each other. so, the program will block in a finite waiting state.

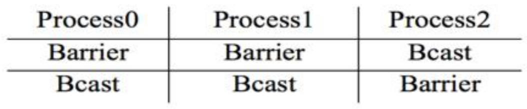


Process 0	Process 1
MPI_RECV (1)	MPI_RECV (0)
MPI_SEND (1)	MPI_SEND (0)

Figure 3: Recv-Recv Deadlock

The following points describe the categories of deadlock situations that occur when using collective routines:

(a) Deadlock due to mismatched collective operations: All members of the communicator may not call the same collective routines in the same order. The MPI standard requires "A correct, portable program must invoke collective communications so that deadlock will not occur, whether collective communications are synchronizing or not." [2].Figure 4 illustrates this situation of potential Deadlock when one of processes executes distinct collective. For a correct MPI program, all processes must execute an identical type of collectives with consistent arguments in the same order; If the operation is synchronizing then a deadlock will occur. Collective operations must be executed in the same order at all members of the communication group



Process0	Process1	Process2
Barrier	Barrier	Bcast
Bcast	Bcast	Barrier

Figure 4: Deadlock due to mismatched collective operation

(b) Deadlock due to incorrectly ordering of collective and point to point routines. MPI standard requires "The relative order of execution of collective operations and pointto- point operations should be such, so that even if the collective operations and the point-to-point operations are synchronizing, no deadlock will occur" [2]. Figure 5 illustrate this situation, one of processes executes distinct call, Process0 first executes a broadcast, followed by a blocking send call. process1 first executes a blocking receive call that matches the send call in process0, followed by broadcast call that matches the broadcast of process zero. This program may deadlock. The broadcast call on process 0 may block until process1 executes the matching broadcast call, so that the send is not executed. Process one will block on the receive call while waiting for the send call.

Process 0	Prpcess1
MPI_BCAST	MPI_RECV (0)
MPI_SEND (1)	MPI_BCAST

Figure 4: Deadlock due to Incorrectly ordering of collective and point-to-point routines

A message race [4],[5] occurs if two or more send calls send messages over communication channels to process have wildcard receive (MPI-ANYSOURCE) simultaneously in transit without guaranteeing the order of their arrivals. In an example in Figure 6 all operation may run correctly if the send call of process 0 matches the wildcard receive from process 2, but if the send of process 1 matches the wildcard receive of process 2 instead that will lead to deadlock since no receive call available for the receive from process 0, that depending of run time choices of implementation.

Process 0	Process 1	Process 2
MPI_SEND (2)	MPI_SEND (2)	MPI_RECV (Any source)
		MPI_RECV (1)

Figure 5: An example of message race

Using of wild card receives (such as MPI-ANY-SOURCE) not always lead to message race. There are some cases that there are no message races even though using of (MPI-ANY-SOURCE). In Figure 7(A) P0 and P1 send messages to P2 with different tags, every receive call in P2 called with (MPI-ANY-SOURCE) but with different tag, first receive call will receive message sent by P0 with tag=1 and second receive call will receive message sent by P1 with tag=2. In this example, even though receive calls are called with MPI any source, two messages being sent by P0 and P1 will be received deterministically because of the different tags. In Figure 7(B), P0 and P1 send messages to P2 with different tags. In this example, two messages will be received deterministically because first receive in P2 will just receive message from P0 and second receive which is called with MPI any source and MPI any tag will receive message sent by P1. In Figure 7(C), P0 send messages to P1. Every receive call in P2 called with (MPI-ANY-SOURCE) but with different tag, two receive calls will receive the messages sent by P0 respectively.

P0	P1	P2
MPI_SEND (2, tag=1)	MPI_SEND (2, tag=2)	MPI_RECV (0, tag=1)
		MPI_RECV (Any source, tag =2)

(A)

P0	P1	P2
MPI_SEND (2, tag=1)	MPI_SEND (2, tag=2)	MPI_RECV (Any source, tag=1)
		MPI_RECV (Any source, tag =2)

(B)

P0	P1
MPI_SEND (1, tag=1)	MPI_RECV (Any source, tag =Any)
MPI_SEND (1, tag=2)	MPI_RECV (Any source, tag =Any)

(C)

Figure 6: No Message races with MPI ANY SOURCE

In Figure 8 we present a simple example of an MPI code. Its focuses on point-to-point communication calls; send to and receive from calls. This example will be referred to in subsequent sections, when describing the steps of the static analysis part of our system.

```
...
if (rank == 0) {
MPI_Send (to 1);
MPI_Recv (from 1);
else if (rank == 1) {
MPI_Recv (from 0);
MPI_Send (to 0);
}
...
```

Figure 7:MPI code Example

3.SYSTEM ARCHITECTURE

The system generally takes the Abstract Syntax Tree (AST) as its input and generates a report of its analysis. AST is used to collect information from the code and detect MPI communication patterns in the code. All collected information are stored in two main data structures MPIRankCase and MPICall. The MPICall class store the information related to a single MPI call (send/receive) including the number and values of each argument. On the other hand, MPIRankCase represents a list of processes, it also keeps a record of all MPI calls that are related to each process. The word ‘rank’ has the same meaning of ‘process’ and from this point onward they will be used interchangeably. Figure 9 presents the system architecture of our error detection system. It consists of 3 modules: The Stacks builder, the Static analyzer and the dynamic analyzer. The Stacks builder creates a call stack for each rank (every rank in program refers to process) in the program. The Static analyzer is responsible for matching each send call with its corresponding receive call in the whole program, since in MPI to fulfil a communication between two ranks, the sender rank should issue a send to call and the receiver rank should issue a receive from call. It can signal out errors when the matching process flow faces problems. The static analyzer either terminates the analysis and reporting the existence of actual deadlocks or race condition or completes the analysis of the whole program. The dynamic analyzer takes as input the executable file and the report generated by the static analyzer. The report points out the type of potential error – whether it is a deadlock or race condition – and the ID of the problematic rank. The error is considered as potential since there is not enough information during static analysis to confirm its existence. During the execution of the program, the dynamic analyzer captures actual call information, i.e. a call that has been issued by a rank - and makes necessary checks to detect errors. It is noteworthy to explain that only certain types of incoming calls that are analyzed, which are send, receive and finalize. The finalize call signifies that a process has ended. This information is used during the application of the time-out mechanism.

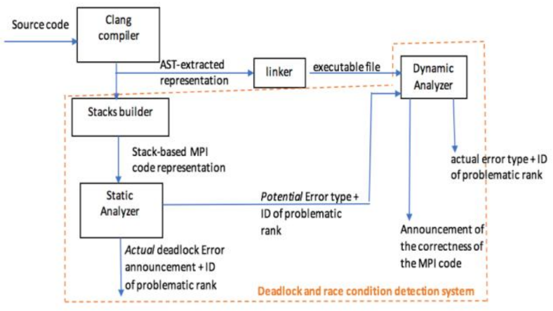

Source code
Clang compiler
AST-extracted representation
linker
executable file
Dynamic Analyzer
Stacks builder
Stack-based MPI code representation
actual error type + ID of problematic rank
Static Analyzer
Potential Error type + ID of problematic rank
Announcement of the correctness of the MPI code
Actual deadlock Error announcement + ID of problematic rank
Deadlock and race condition detection system

Figure 8: System Architecture

4.STATIC AND DYNAMIC ANALYSIS

In this section, we first describe the general process of analysing the code during compile time. Then, we present detailed description of the design decisions related to static analysis made in certain parts of the process. Figure 10 shows a schematic description of the steps of the process.

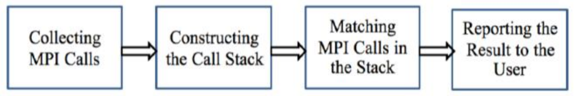

Collecting MPI Calls
Constructing the Call Stack
Matching MPI Calls in the Stack
Reporting the Result to the User

Figure 9: Main steps in static checking phase of MPI programs

The steps of the process are as follows:

1- The call information within all ranks are collected in the rank and call constructs to generate the AST-extracted representation of the MPI code.

2- For each rank in the MPI program a stack is built and all calls issued by the specific rank are pushed in the stack. The calls are organized in the stack in an order that ensures that the top of the stack will be the call that will be the first to be executed. Figure 11 shows the stack representation of the code example described in Figure 7.

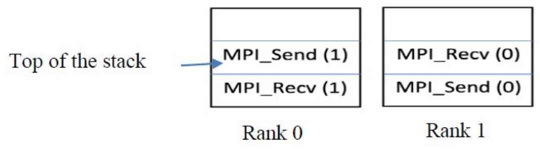


Figure 10: Stack representation of the MPI code example of Figure 7

3- Creating the expected MPI communication pattern for each call, by matching each send call with its corresponding receive call and each receive call with its corresponding send call. This should be completed for the whole program to consider it as a correct piece of code. Any error in the flow of this matching will indicate a deadlock error. In this step, certain checks are also done to detect race conditions.

4- Whenever an error is found the analysis process will stop and a report describing the type of error and the problematic rank is presented to the user. If the deadlock errors found are actual then the dynamic analyzer will not be triggered, where as potential errors found, whether it be a deadlock or a race condition, then the same report along with the executable file of the MPI program will be fed to the dynamic analyzer, and then the dynamic analysis will start.

Before commencing the process of checking the MPI code, we have to collect all MPI calls in the source code. Fortunately, there are many tools that can parse the C++ code and provide us with a syntax tree that can be further processed by our algorithm. Clang is a well-known tool that can achieve this goal. It is a frontend compiler that exposes the generated commencing information by the backend compiler to the application programmers such that they can make use of the generated parsing to check C++ code or even to adjust legacy C++ code by automatically replacing certain constructs. This feature is indeed very useful compared to the manual amend of the code. For the purpose of our application, we have adopted the clang infrastructure to collect MPI calls.

During the process of extracting and collecting the MPI calls, we follow a similar approach in [6] by classifying call statements into different categories depending on the statement type. The MPI statements can be classified into: declarative statement versus executive statements, blocking versus non-blocking, point-to-point versus collective calls, or send versus receive calls. It should be noted that this list is not exclusive, i.e., a call can belong to more than one category. For instance, an MPI_Send can belong to the executive, blocking, and point-to-point communication group at the same time. Declarative MPI statements are used only to specify some parameters to the MPI programs. They do not involve any sort of communication when they are called. For example, the MPI_Comm_rank and MPI_Comm_size statements are used to populate a variable with the current process rank as well as the total number of processes, respectively, that take place on the application. This type of calls is imperative to the checking procedure, and they have to be collected during the parsing stage as the information collected by such calls will be required during the checking phase to help inferring the process rank. In contrast, the executive calls perform communication between two or more processes when they are called. It is also important to distinguish such calls from the declarative calls such that the checking algorithm is not applied to a call that belongs to the former group. Furthermore, MPI calls can be classified into blocking and non-blocking calls. During the collection phase, we also mark the call type so that we can make use of such information during the checking process. The call type is determined by comparing the call text to the list of calls in this group. A map can be filled with the calls that belong to one type. This map is then searched using the call on hand. As a further classification of the MPI calls, we examine if the call is a collective or a point-to-point call. Point-to-point calls communicate directly to a single node process while collective calls communicate with more than one processes when the call is issued. It is important to note that the check can be implemented by storing all the calls that belong to a certain class in a standard container (e.g., map, vector, or a list), and then, we search this list each time we have a call whose type needs to be determined. Besides, this list of checks can be extended by adding further checks. Once the call category has been determined, it can be stored into the corresponding call structure. We assume that each call has a special structure that encapsulates all the required information by a certain call. To store the different categories of calls, one solution is to add a Boolean field representing which category this call belongs to. After all of the required information has been collected, we move to the actual check of the MPI code. In fact, it is not an easy process to perform the check statically since a lot of information is missing. Here, we classify the required information into two categories: identification of the MPI process ranks and deciding the calling order of the MPI calls. As there is no runtime information available, there is no direct way to easily obtain the process ranks. Therefore, we have to use an algorithm that can extract as much of such information as possible so that it can assist our MPI checkers when the rank information is required. Similarly, the call order of the MPI functions is not available during compile time. Unfortunately, this information is imperative to the operation of the deadlock detection algorithm since deadlock might be caused by a direct call of two MPI processes. Generally, when an MPI program is written, it is usually subdivided into a number of MPI cases in the form of (if ...else) statements. Each conditional statement that includes a rank inside its condition can be considered as an MPI case, according to the discussion given at [6]. Therefore, we can make use of this information by collecting each MPI case while collecting the MPI calls. 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. Of course, there are some cases when the MPI case might refer to range of processes (e.g., when the if condition consists of less than or greater than logical operator). Ideally, the MPI case can represent one process when the condition operator inside the if statement employs the equal conditional operator. If we have identified all or most of the process via their corresponding MPI case. In this 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. So, the top will be the first to be executed. When we start the checking procedure, we try to match the top element from a stack with another one from any of the other stacks. However, while we are trying to match the stack top with another call, the matched call might be not at the top of the stack. In this case, we might need to search the whole stack for the matched call. If the matched call is not located at the top of stack, then we have to recursively match all calls that proceed or goal one.

Similar to deadlock detection using the static approach, we can also detect race conditions by searching the stacks for the necessary conditions that can lead to races between one or more MPI send calls. The main condition that might lead to races between MPI calls is the use of MPI_ANY_SOURCE constant as well as MPI_ANY_TAG. In this case, the receive call is prepared to receive from any send call. However, the usage of such constants inside MPI code does not imply that a race exists. Therefore, additional search of the calling semantics of the MPI function is required to arrive at a final conclusion. The general idea for detecting a race condition using our stack-based approach is to match the MPI calls in a similar manner as in the case of deadlock detection. However, instead of just matching the MPI calls, we will also try to detect if there is a race between two calls. While we are matching the call, we repeatedly check the tops of all stacks and not only one stack, as in the case of deadlock detection. If there is more than one matched call in the stack tops that match the one in consideration, then we signal that a race has occurred. If all the stacks are empty and we did not detect any race condition, this will mean that there is no race condition in the MPI program under investigation. During the visiting of the stacks, we make use of the previous procedures described for detecting deadlock. Moreover, it is mandatory for the race checking procedure to evaluate the rank of all the MPI senders so that it can discover the race.

In general, it is not possible to obtain the required values for the ranks of MPI calls, since at this point the compiler may be dealing with just symbols during the compilation phase. Figure 12 presents the situation where x and y are thedestinations of the send call.

Process 0	Process 1	Process 2
MPI_SEND (x)	MPI_RECV(MPI_ANY_SOURCE)	MPI_SEND (y)

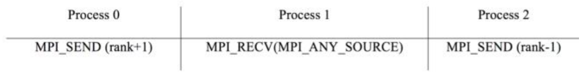
Figure 11: An example using (receive –from –any) with symbol values

To conclude that there is a race or not, we need to know the numeric value of x and y. If this value is evaluated to be 1, then there will be a race. If not, there is no race in these calls. If we are lucky, we can find the set of calls written as shown in Figure 13. In this figure, the destination rank of the send calls in processes 0 and 2 are obvious, and we can extract them directly as they are evaluated to constants.

Process 0	Process 1	Process 2
MPI_SEND (1)	MPI_RECV(MPI_ANY_SOURCE)	MPI_SEND (1)

Figure 12: An example using (receive –from –any) with obvious values

However, in Figure 14, which is similar to the previous one, the destination ranks are given in terms of the process ranking. In this example, the destination rank is given in terms of the process rank itself. Although here rank is a variable, we can detect that there will be a race condition if we can know the relative ordering of each process. For instance, the destination of the send call in process 0 can be determined if we know the value of the rank variable (so the destination will be 1 in this case, since the process rank is 0). Similarly, the destination of the send call in process 2 can be determined (the destination will be also 1, since the process rank is 2).

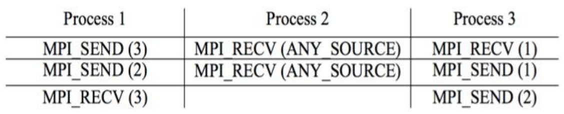


Process 0	Process 1	Process 2
MPI_SEND (rank+1)	MPI_RECV(MPI_ANY_SOURCE)	MPI_SEND (rank-1)

Figure 13: An example using (receive –from –any) depending on Rank values

Therefore, a race can be detected in this simple MPI program. Similar heuristics can also be used to assist in discovering and evaluating the destination of the send call. However, if we fail to apply one of these heuristics to the send call under consideration, our checker will not arrive to a conclusion as towhether this MPI code involves a race condition.

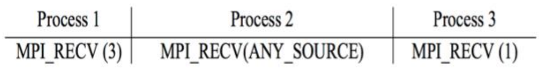
After we have provided our solution to the problem of checking MPI programs statically, we move to the dynamic approach where the checking is performed during runtime. In this case, we have all the required information to perform the check. Therefore, this approach would be more powerful than the static one. However, it has the disadvantage of introducing computational overhead each time the program is executed. From static analysis, the dynamic part will receive the following: (a) Type of error: deadlock or race condition, (b) Problematic process. The Static phase outlines all execution paths that may lead to potential deadlock. The dynamic phase checks only the execution processes with potential deadlock by using the timeout mechanism. The user defines a limit time to these processes to wait in an MPI call. If this time exceeds the limit time defined by the user on that process, a deadlock warning is issued. Our proposed approach of detecting the race dynamically involves creating a central entity that will be responsible for collecting the data from the application processes in order to analyze it. This entity is very important since detecting the race dynamically will involve analyzing information that is not available on a single process. Although this might introduce a new communication bottleneck for the application process, it may be acceptable because the single entity can be extended to include different copies that communicate to each other. The number of copies will depend on the user preference, and they can be dynamically created and destroyed depending on the application requirement. As mentioned earlier, message race means we have two send calls that have a race condition with one receive call (receivefrom- any). Obviously, only one of the two calls will be matched, and the other may be deadlocked. The order of which of them will be matched first is nondeterministic, and this is the problem with detecting the race conditions. Therefore, the idea of detecting race conditions is based on how the race takes place. In particular, when a receive call is matched with any send call, we save the last receive call that has been matched. When a new send call comes, we first try to match it with the last matched receive call. If we can match it, then a race condition is detected. Otherwise the matching procedure will continue. This idea can be implemented inside the server by creating a call object that stores the last matched call. This object will be created inside each process that resides inside the server. Each time a receive call is matched, we record such a call as the last matched call. When a new send call arrives, it will be checked for a match with the last matched call. If the new send call can also be matched, then there will be a race. The main proposal here is that if one receive call is matched, there should be no other send calls that can be matched with this receive call. If this condition cannot be fulfilled, it means a race exists in this MPI call. Figure 15 presents an example of detecting a race condition. Process 1 sends to process 3, and the last matching receive will be the first call in process 3. Process 1 will then issue the next MPI call, which will be matched with the receive call in process 2 (the first MPI_Recv(MPI_ANY_SOURCE)). This receive call will be matched and marked as the last matched receive call in this process. Now suppose that process 3 will issue the next send call (MPI_Send (1)), which will be matched with the receive call in process 1. The last matched receive call in process 1 will be MPI_Recv (3).



Process 1	Process 2	Process 3
MPI_SEND (3)	MPI_RECV (ANY_SOURCE)	MPI_RECV (1)
MPI_SEND (2)	MPI_RECV (ANY_SOURCE)	MPI_SEND (1)
MPI_RECV (3)		MPI_SEND (2)

Figure 14: An example of race condition

The order of the matching between of the send calls belonging to different processes is not important because each process request will be queued in the destination process when they are received. For instance, suppose that the second receive call in process 2 (Recv _ANY_SOURCE)) has arrived at the server before any send call. It will be then queued in the receive queue maintained by the process 2 data structure in the server side. When a new send call arrives to process 2, it will be matched with the last matched receive call first and then matched with one of the receive calls in the queue. Thecurrent situation is shown in Figure 16. This figure shows that each process will maintain the last matched receive call. Consider that process 3 issued its last send request (MPI_Send (2)). When this request arrives to the server, it will route it the process data structure representing process 2. When this send call arrives at process 2, it will be matched first with the last matched receive call in process 2 (MPI_Recv(MPI_ANY_SOURCE) ;). In this case, it will be matched, and therefore, a race will be detected. If the send call of process 3 was not matched with the last receive call in process 3, no race would be detected.



Process 1	Process 2	Process 3
MPI_RECV (3)	MPI_RECV(ANY_SOURCE)	MPI_RECV (1)

Figure 15: Last matched receive

The order of the matching between of the send calls belonging to different processes is not important because each process request will be queued in the destination process when they are received. For instance, suppose that the second receive call in process 2 (Recv _ANY_SOURCE)) has arrived at the server before any send call. It will be then queued in the receive queue maintained by the process 2 data structure in the server side. When a new send call arrives to process 2, it will be matched with the last matched receive call first and then matched with one of the receive calls in the queue.

5 OVERALL ASSESSMENT.

In this section, we present a brief discussion of the implementation of our framework for dynamic race detection. The MPI race server is the central component of our framework. It is responsible for each MPI call issued by any of the application processes. Once a call has been received, it is handled by one of the analysis modules that resides inside the server. When an MPI call is received by an analysis module, a complete record about this call is stored. This information is useful when the analysis module starts working. Many entries related to the process can be located via this information (e.g., the call process, the call data type, etc.).

The instrumentation library links between the application code and the remote server. It provides a way to collect the required information and send it automatically to the server. Inside the library, we designate our own wrapper to the MPI calls. Inside our wrapper, we collect the necessary information before issuing the MPI call. We call the requested MPI function on behalf of the application itself. The application code represents the user application that needs to be checked by our framework. It is called by the user, and it communicates with the server via the instrumentation library. The number of processes is mainly controlled by the user when the application command is issued. There is no limit on the number of processes that can be created by the application when it is started. Using our tool to analyze the programs in the Umpire benchmarks [7], we have successfully analyzed a lot of programs, i.e., either no deadlock detected or detecting a deadlock as expected.

Table 1 and Figure 17 summarize the outcome of the performed experiments. The first column presents the Umpire benchmarks, the second column provides the execution time with static phase information and the third column provides the execution time without static phase information. We execute all benchmarks in this experiment with 8 processes. From the presented data in this table it can be shown the time overhead decrease about 60% when we use static phase information. Table 2 and Figure 18 show that the time overhead in different number of processes. From this table, we can observe that, for all the selected benchmarks, the execution time does not increase exponentially with respect to the number of processes. It justifies that our tool avoids the exponential increasing of execution.

Table 1: Time overhead in run time with and without static phase

Umpire benchmark	Execution time with static phase information	Execution time without static phase information
any_src-can-deadlock.c	0.1101	0.176
any_src-can-deadlock3.c	0.3230	0.556
basic-deadlock.c	0.2210	0.332
Bcast-deadlock.c	0.1873	0.310
head-to-head.c	0.3801	0.689
collective-misorder mod.c	0.2810	0.4496

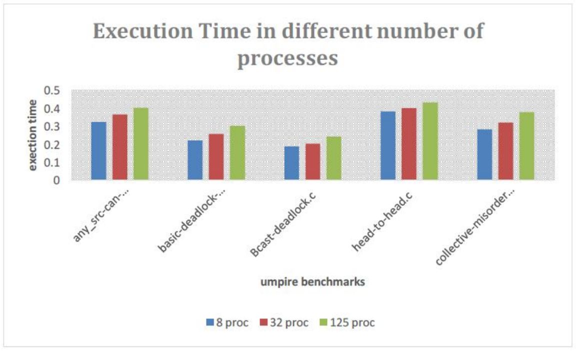


Figure 16: The experimental results with and without Static phase

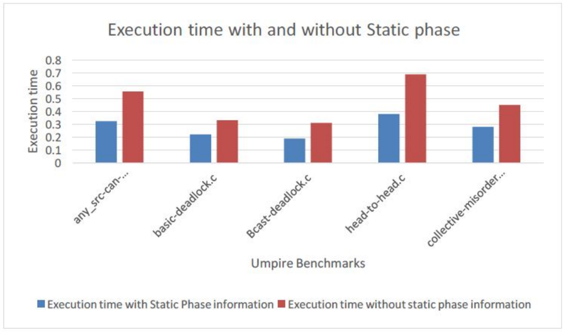
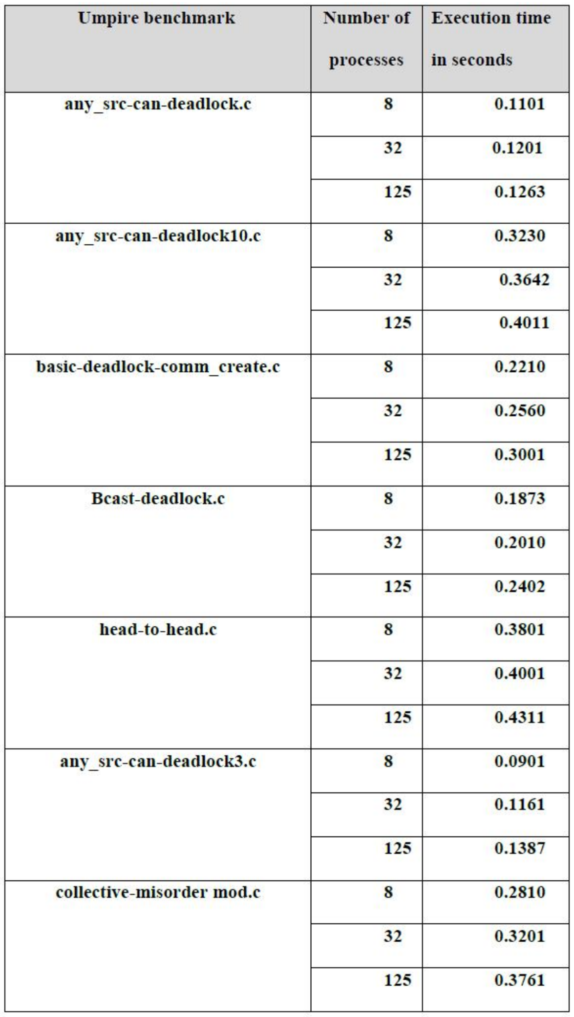


Figure 17: The experimental results under different numbers of processes

In the sequential programs, static analysis has been successful at analyzing programs. However, analyzing MPI programs is difficult for many reasons: no runtime information (as in the dynamic case) is available like the number of MPI processes and MPI provides several nondeterministic calls (such as MPI _ ANY _ SOURCE and MPI MPI_ANY_ TAG). Some testing approaches treat MPI programs as sequential codes, making it possible to determine simple usage errors (such as type and expression errors). However, these approaches cannot represent the programs' communication topology. Another testing approaches (such as MPIChecker [6]) that extends traditional dataflow analyses to MPI programs, extracting the program's communication topology and trying to match the send and receive calls that may communicate at runtime. If they found partners, they are marked. At the end, the algorithm returns the set of unmarked MPI Calls which are considered not matched. So, they just detect one type of deadlock (Deadlock duo send receive mismatch) but our Static approach extends the program representation used in the LLVM and clang compilers, to enable the extraction of fine-grained information about exchanging messages, during static analysis. Our approach represents the execution of multiple processes using stacks idea, keeping track of any send and receive calls and predict many types of deadlock error and race condition. And we can detect many types of deadlock in Static phase (Deadlock duo send receive mismatch, Head-to-Head deadlock and collective deadlock). Unlike existing static analysis our tool can detect race condition in Static phase. Existing dynamic analysis tools (like Umpire, MARMOT and MUST) suffers from time overhead at runtime, but our dynamic analysis checks only execution processes with potential errors instead of checking all processes, for that the overhead in this phase is limited.

Table 2: Time overhead in different number of processes



Umpire benchmark	Number of processes	Execution time in seconds
any_src-can-deadlock.c	8	0.1101
	32	0.1201
	125	0.1263
any_src-can-deadlock10.c	8	0.3230
	32	0.3642
	125	0.4011
basic-deadlock-comm_create.c	8	0.2210
	32	0.2560
	125	0.3001
Bcast-deadlock.c	8	0.1873
	32	0.2010
	125	0.2402
head-to-head.c	8	0.3801
	32	0.4001
	125	0.4311
any_src-can-deadlock3.c	8	0.0901
	32	0.1161
	125	0.1387
collective-misorder mod.c	8	0.2810
	32	0.3201
	125	0.3761

There are several approaches for detecting message races such as MARMOT[8], MAD [9], and MPVisualizer [10]. However, those approaches are not suitable for debugging MPI programs because they do not provide information to locate and debug message races. Also, some of them detect message races just by identifying the use of wild card receives (like MPI_ANY_SOURCE and MPI_ANY _TAG) as sources of race conditions. Using of wild card not always lead to message race. Therefore, programmers can be overwhelmed by the incorrect information or be incapable of finding where the races occurred in a huge source code. Our tool by using Stack idea in Static Phase and last matched receive idea in Dynamic Phase can detect race condition that may happened between calls without depending on identifying the use of wild card receives, as the only source of race condition. Existing hybrid analysis tools (which are few) as [11] does not coverall MPI communication routines, our tool can detect deadlock and race condition in point-to-point and collective MPI routines.

6.CONCLUSIONS

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 [1] 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. Moreover, we support detecting basic race conditions that can result from inappropriate structure of the MPI code. To overcome the shortcoming of the static approach we have also implemented a dynamic approach for detecting deadlock and race conditions. 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 and race condition.

Acknowledgement

We would like to express our gratitude and appreciation to Prof. FathyEassa the head of our research group in the department of Computer Science at KAU. He always provides us with promising research directions and continuous guidance.

REFERENCES

1. Clang. **http://clang.LLVM.org/docs/index.html**
2. W. Gropp, E. Lusk, and A. Skjellum, **Using MPI: portable parallel programming with the message-passing interface**, vol. 1. MIT press, 1999.
3. M. Schulz and B. R. De Supinski, “**A flexible and dynamic infrastructure for MPI tool interoperability**,” in Parallel Processing, 2006. ICPP 2006, International Conference on, 2006.

4. D. R. Shires and L. Pollock, **“Program flow graph construction for static analysis of explicitly parallel message-passing programs,”** Army Research Lab Aberdeen Proving Ground MD, 2000.

5. A. P. Cláudio, J. D. Cunha, and M. B. Carmo, “**MPVisualizer : a General Tool to Debug Message Passing Parallel Applications**,”.

6. A. Droste, M. Kuhn, and T. Ludwig, **“MPI-checker: static analysis for MPI,”** in Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC, 2015, p. 3.

7. J. S. Vetter and B. R. De Supinski, “**Dynamic Software Testing of MPI Applications with Umpire,”** ACM/IEEE SC 2000 Conf., pp. 12–15, 2000.

8. Krammer, Bettina, MARMOT: **An MPI analysis and checking tool.** In: Advances in Parallel Computing. North-Holland, 2004.

9. D. Kranzlmüller, C. Schaubschläger, and J. Volkert, “**A brief overview of theMAD debugging activities,**” arXivPrepr. cs/0012012, 2000.

10. A. P. Cláudio, J. D. Cunha, and M. B. Carmo, “MPVisualizer: **A general toolto debug message passing parallel applications,”** in International Conferenceon High-Performance Computing and Networking, 1999.

11. E. Saillard, P. Carribault, and D. Barthou, “**Combining static and dynamic validation of MPI collective communications,**” in Proceedings of the 20th European MPI Users’ Group Meeting, pp. 117–122 2013.