

Received March 21, 2020, accepted May 9, 2020, date of publication May 12, 2020, date of current version May 28, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.2994172

ACC_TEST: Hybrid Testing Techniques for MPI-Based Programs

ABDULLAH S. ALMALAISE ALGHAMDI¹, AHMED MOHAMMED ALGHAMDI²,
FATHY ELBOURAEY EASSA³, AND MAHER ALI KHEMAKHEM³

¹Department of Information Systems, Faculty of Computing and Information Technology, King Abdulaziz University, Jeddah 21589, Saudi Arabia

²Department of Software Engineering, College of Computer Science and Engineering, University of Jeddah, Jeddah 21493, Saudi Arabia

³Department of Computer Science, Faculty of Computing and Information Technology, King Abdulaziz University, Jeddah 21589, Saudi Arabia


This project was funded by the Deanship of Scientific Research (DSR), King Abdulaziz University, Jeddah, under grant No. (RG-9-611-40). The authors, therefore, gratefully acknowledge the DSR technical and financial support.

ABSTRACT Recently, MPI has become widely used in many scientific applications, including different non-computer science fields, for parallelizing their applications. An MPI programming model is used for supporting parallelism in several programming languages, including C, C++, and Fortran. MPI also supports integration with some programming models and has several implementations from different vendors, including open-source and commercial implementations. However, testing parallel programs is a difficult task, especially when using programming models with different behaviours and types of error based on the programming model type. In addition, the increased use of these programming models by non-computer science specialists can cause several errors due to lack of experience in programming, which needs to be considered when using any testing tools. We noticed that dynamic testing techniques have been used for testing the majority of MPI programs. The dynamic testing techniques detect errors by analyzing the source code during runtime, which will cause overheads, and this will affect the program's performance, especially when targeting massive parallel applications generating thousands or millions of threads. In this paper, we enhance ACC_TEST to have the ability to test MPI-based programs and detect runtime errors occurring with different types of MPI communications. We decided to use hybrid-testing techniques by combining both static and dynamic testing techniques to gain the benefit of each and reduce the cost.

INDEX TERMS MPI, MPI testing tool, hybrid testing techniques, parallel programming, ACC_TEST.

I. INTRODUCTION

Message-Passing Interface (MPI) is one of the most widely used programming models for parallelizing most scientific applications. This programming model is used for supporting parallelism in sequential programming languages by adding MPI directives to control data movements between processes. MPI also supports integration with some programming models and has several implementations from different vendors, including open-source and commercial implementations. However, testing parallel programs is a difficult task, especially when using programming models with different errors behaviours and types based on the programming model type. In addition, the increased use of these programming models by non-computer science specialists can cause several errors due to lack of experience in programming, which needs to be considered when using any testing tools.

The associate editor coordinating the review of this manuscript and approving it for publication was Fan Zhang .

As a part of our previous work [1]–[3], we proposed and created a parallel hybrid testing tool named ACC_TEST that targeted programs built in a heterogeneous architecture and covering different errors. In addition, we aim to develop hybrid-testing techniques for detecting errors in the dual-programming models MPI + OpenACC at the ends of our project. In this paper, we enhance ACC_TEST to have the ability to test MPI-based programs and detect runtime errors occurring with different types of MPI communications. We also focus on the interaction between MPI and the other programming models, especially high-level programming models such as OpenACC.

The rest of this paper is structured as follows. Section 2 briefly gives an overview of MPI, and section 3 will discuss the related work. In section 4, we explain our techniques for testing MPI-based programs covering different types of MPI communications, including point-to-point and collective communications. In sections 5 and 6, we explain and discuss implementing, testing, and evaluating our

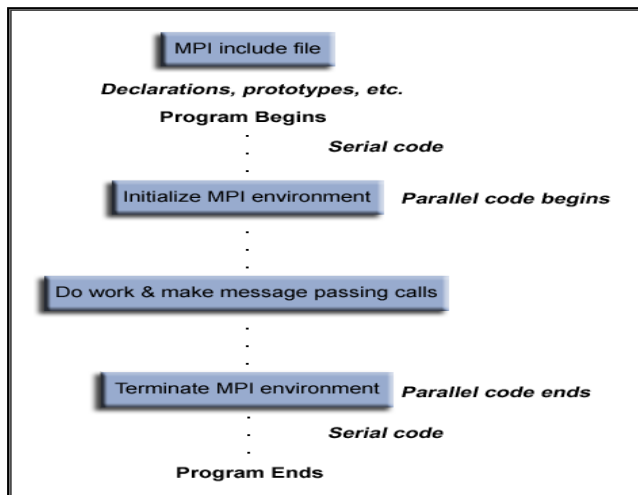


FIGURE 1. General MPI program structure [4].

techniques for testing MPI. Finally, conclusions and future work will be discussed in section 7.

II. MPI PROGRAMMING MODEL

Message-Passing Interface (MPI) [4] is a programming model used for message-passing techniques for supporting parallelism in traditional programming languages, including Fortran, C, and C++. The first official version of MPI was released in May 1994. Data is moved from a process address space to another process by using cooperative operations in each process. The aim of MPI is to establish a standard for creating message-passing programs to be portable, efficient, and flexible. MPI has several implementations, including open-source implementations such as Open MPI [5] and MPICH [6], and commercial implementations such as IBM Spectrum MPI [7] and Intel MPI [8]. MPI is considered as a standard, portal programming model that can be implemented in several platforms, hardware, systems, and programming languages. Additionally, MPI can work perfectly with several programming models and with heterogeneous networks. In addition, MPI has various versions of MPI implementations from different vendors and organizations that are available as open-source and commercial implementations.

An MPI program has a specific structure, starting with MPI including the file in the header and then MPI environment initialization, which considers the beginning of the parallel code. After that, the main message-passing calls take place, and in the end terminate the MPI environment. Figure 1 demonstrates the general MPI program structure with respective examples of MPI codes.

There are two types of MPI communications, including point-to-point and collective communications. MPI point-to-point communication typically involves message passing between two and only two different MPI tasks. The first task is to perform a send operation, and the other task is to perform a matching receive operation. There are different types

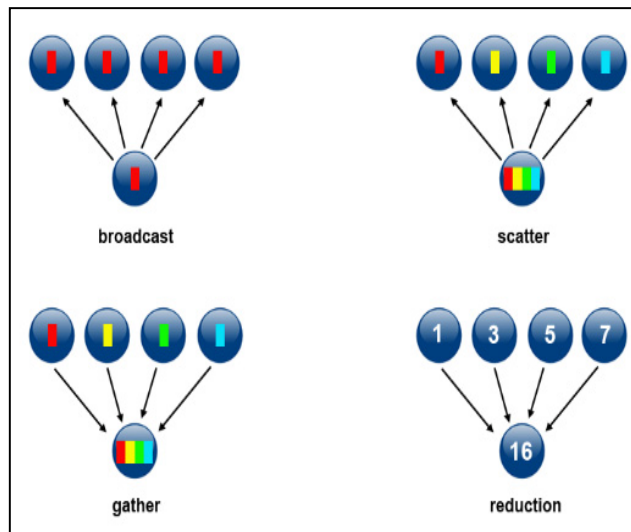


FIGURE 2. Example of MPI collective communications [4].

of send-and-receive operations used for different purposes, including:

- 1) Blocking send and receive.
- 2) Non-blocking send and receive.
- 3) Buffered send.
- 4) Combined send/receive.

In terms of collective communication, there are three types: synchronization, data movement, and collective computation. Also, MPI collective communications can be blocking or non-blocking, just like MPI point-to-point communications. Figure 2 demonstrates the collective computation and data movements for MPI collective communications.

The newest MPI standardization version 4.0 [4] is currently available, which aims to add new techniques, approaches, and concepts to the MPI standard to help MPI address the needs of current and next-generation applications and architectures. The new version extends to better support hybrid-programming models, including hybrid MPI+X concerns and support for fault tolerance in MPI applications.

III. RELATED WORK

There are many testing tools that target MPI using different testing techniques and covering different types of error. In our survey, we cover more than 20 testing tools that target MPI programming models. Using static testing techniques, the testing tool MPI-Checker [9] has been used for detecting mismatching errors in MPI-related programs.

For the dynamic testing techniques, we studied more than 15 testing tools that use dynamic testing for MPI-related programs to detect various errors. For detecting runtime errors including mismatching, data race, and deadlocks, many testing tools have been used such as MUST [10], [11], MEMCHECKER [5], Intel Message Checker [12], STAT [13], Nasty-MPI [14], and MARMOT [15]. In addition, the testing tools that have been used for detecting mismatching

and deadlocks include Umpire [16], GEM [17], and MPI-CHECK [18]. For race condition and deadlocks, MPI detections MAD [19], [20], and PDT [21] have been used. MPIRace-Check [22] has been used to detect race conditions in MPI. Finally, MOPPER [23] and ISP [24] are used for deadlock detection.

In the case of hybrid testing techniques, we did not find enough research tools that used the hybrid testing techniques for building a testing tool for MPI-related programs covering wide range of errors. However, MPI collective communication has been validated by using a two-phase analysis for detecting collective patterns in an MPI program that can cause deadlock [25]. Also, for detecting errors in the MPI/OpenMP dual programming model, two hybrid testing tools have been used, including PARCOACH [25] and [26] for detecting deadlocks and other runtime errors.

In our study, we noticed that some tools considered as debugging tools, not testing tools, including AutomaDeD [27], ALLINEA DDT [28], TotalView [29], PDT [21] and MPVisualizer [30]. We have noticed that these five debuggers do not help to test or detect errors, but instead are used for finding the causes of the errors.

In our literature review [31], we note that dynamic testing techniques have been used for testing the majority of MPI programs. The dynamic testing techniques detect errors by analyzing the source code during runtime, which will cause overheads, and this will affect the program's performance, especially when targeting massive parallel applications generating thousands or millions of threads. Also, dynamic testing needs some insertion mechanisms to perform the testing and get better results, which also comes with its own cost. On the other hand, only one testing tool that used the static testing has less execution and size overheads, but does not detect all errors. Finally, in this version of our testing tool ACC_TEST, we decided to use hybrid testing techniques by combining both static and dynamic testing techniques to gain the benefit of each and reduce the cost. Also, we decided to cover errors from each type of MPI communications because the previously mentioned testing tools did not cover some errors or only focused on race condition and deadlocks.

IV. OUR TECHNIQUES FOR TESTING MPI-BASED PROGRAMS

In this section, we will explain our techniques for detecting some errors related to MPI applications. As we discussed earlier, there are many testing tools related to MPI; therefore, we only focus on detecting some errors that occur in MPI programs, which include GPU-related programming models. On the MPI side, we tried to minimize the overhead and the slowdown that can occur during dynamic testing, as we will explain later. We will cover some errors from each type of MPI communication, including point-to-point and collective communications.

In terms of MPI point-to-point, we cover two different cases: blocking and non-blocking point-to-point communications. In the blocking type, our hybrid testing technique will

examine the targeted source code by analyzing the code, collecting the related information to MPI_Send, MPI_Recv, and MPI_Sendrecv for detecting any actual or potential errors. In addition, in the non-blocking type, ACC_TEST will deal with MPI_Isend and MPI_Irecv by collecting their related information and analyze their behavior to detect any errors. In the following, we will explain how ACC_TEST detects an MPI-related program and classify them into three sections:

A. POINT-TO-POINT BLOCKING COMMUNICATION DETECTION

ACC_TEST will be responsible for detecting any point-to-point blocking communication, including MPI_Send, MPI_Recv, and MPI_Sendrecv. We chose the previous three MPI directives because of their popularity, and they are the most-used MPI blocking calls in several related programs. In ACC_TEST, we focus on the effects of OpenACC and MPI directives for each other and what types of error could be caused by this interaction.

Our static analysis will analyze the source code to find any MPI-related calls and determine their place in the source code, what type of MPI calls they are, and what arguments they have including source, destination, data type, communicator, and rank, as well as their relationship to the OpenACC regions. Our static analysis also will lexically analyze the MPI calls, as well as parsing them to ensure they are following the MPI call rules. Our static analysis will start by determining each MPI send-and-recv and their locations in the source code, storing their information in a data structure and also ascertaining to which MPI rank they belong for determining the message direction. Our static analysis will create several tables for storing the related send-and-recv calls based on their type and storing their related information, including the rank and type, communicator, tag, and line number. Then we compare this information in the static phase, searching for any missing potential for race condition or deadlock as well as mismatching. In addition, our static analysis will check for any illegal MPI calls before MPI_Init and after MPI_Finalize. Race conditions caused by reading and writing to the same MPI buffer can be detected by our static tool. Also, in the same rank, if there is MPI_Send and MPI_Recv, the MPI_Recv should precede the MPI_Send; if not, our static testing will detect that and issue a warning message to the programmer.

Additionally, ACC_TEST will determine the MPI_Send/MPI_Recv pair, which will be used to detect any differences between the numbers of send-and-receives. This pairing will also examine the message tag to detect any unmatched message pairing. Our static analysis will check any message leaks (messages that were sent but never received) or inconsistent types on the sender and receiver for the same message.

In the case of having more senders than receivers, which is considered a lack of resources that can also lead to a potential race condition, which will be detected by our dynamic tester to know the exact error. On the other hand, when the number

of receivers is more than the number of senders, this will lead to potential deadlock because they will be waiting for a message without receiving it; this will also be annotated for further testing in our dynamic phase.

In addition, our static analysis has the ability to detect any mismatching in data types and message sizes. Our static analyzer will also analyze the relationship between OpenACC and MPI directives to determine the mismatch in the data movement between the OpenACC and MPI directives. This will detect the data type mismatching not only between the MPI_Send and MPI_Recv calls, but also in the code; for example, if the programmer defines a variable as an INT in his code and passes this variable in an MPI call as MPI_CHAR, ACC_TEST will detect this error.

In terms of race condition detection, our static analysis will detect any case of several messages being sent to the same destination with the same tag number, which can cause a race condition.

In terms of deadlock detection, our static phase has the ability to detect actual and potential deadlock based on our static analysis of the targeted source code. One of the potential error situations is using the wildcard receive. Additionally, our static analysis will detect any wildcard receive with any source or any tag and examine them to avoid any potential deadlock or race condition and annotate them to be detected in our dynamic phase.

Another case of point-to-point blocking communication is the MPI_Sendrecv calls, which will be examined and analyzed like the previous MPI_Send and MPI_Recv calls. Additionally, the error detection will be as described in previous MPI calls because they show the same behavior but with a different structure.

In our dynamic phase, deadlock and race condition will be detected using the annotation of our static phase and insertion of the appropriate statements for detecting the actual error during runtime. ACC_TEST tests only the connections that have potential errors as determined by our static testing analysis, which saves time and enhances testing performance by testing only the part that needs to be tested and minimizing overhead and slowdown from dynamic testing.

For detecting deadlock in point-to-point blocking communication, ACC_TEST will reference the marked MPI_Recv that has potential errors as determined by our static analysis. Then, our insertion mechanism will replace the MPI_Recv with MPI_Irecv and define new MPI_Request and MPI_Status for testing purposes. A timer will be set for a specific time, determined by calculating the average of the required times. Finally, we test the MPI_Irecv by using the ACC_TEST to see if the connection is completed or not, and if it is completed, determine whether there is matching between the source and the tag of this connection. However, if the connection is not received, this indicates that this connection has deadlock, and if it uses the MPI_Recv, the program will freeze. In Figure 3, ACC_TEST inserts several statements for detecting deadlock in point-to-point blocking communication for testing MPI_Send and MPI_Recv.

```
// Insert test code for testing point-to-point blocking communication (MPI_Send/MPI_Recv)

// THE FOLLOWING LINE IS HAVE THE INSERTED TESTING CODE AFTER THE MPI_RECV ****

//ASSERT* MPI_Request request_var_name;
//ASSERT* MPI_Status status_var_name;
//ASSERT* int flag_var_name;
//ASSERT* MPI_Irecv(Buffer, Count, Data_Type, Source, Tag, MPI_Communicator, request_var_name);
//ASSERT* Set a timer;
//ASSERT* MPI_Test(&request_var_name, &flag_var_name, &status_var_name);
//ASSERT* if (!flag_var_name)
//ASSERT* {
//ASSERT* cout << "ERROR :: ==> THERE IS A DEADLOCK IN THE MPI_RECV No: " << MPI_RECV_ID << " IN RANK NO:" << Rank << " IN LINE NO:" << line_no << endl;
//ASSERT* }
//ASSERT* else
//ASSERT* {
//ASSERT* cout << "The Tested MPI_RECV NO:" << MPI_RECV_ID << " is Completed" << endl;
//ASSERT* cout << "The Flag Value: " << flag_var_name << endl;
//ASSERT* cout << "Received From Process No: " << status_var_name.MPI_SOURCE << endl;
//ASSERT* cout << "With TAG No: " << status_var_name.MPI_TAG << endl;
//ASSERT* }
}
```

FIGURE 3. Insert test code for testing deadlock in point-to-point blocking communication (MPI_Send/MPI_Recv).

```
// Insert test code for testing race condition in point-to-point blocking communication (MPI_Send/MPI_Recv)

//ASSERT* if (status_var_name.MPI_TAG != Static_Tag_Value | status_var_name.MPI_SOURCE != Static_Source_Value)
//ASSERT* {
//ASSERT* cout << "XXX MPI BLOCKING COMMUNICATION ERROR RACE CONDITION XXX" << endl;
//ASSERT* cout << "XX ERROR : In Line " << line_no << " THERE IS A RACE CONDITION IN THE MPI_RECV No: " << MPI_RECV_ID << endl;
//ASSERT* cout << "IN RANK NO:" << rank << " EQUALS THE MPI_RECV No." << MPI_RECV_ID << "RECEIVED FROM RANK:" << rank << " WITH TAG: " << Tag << endl;
//ASSERT* cout << " BUT IT SHOULD BE RECEIVED FROM RANK:" << Static_Source_Value << " WITH TAG:" << Static_Tag_Value << endl;
//ASSERT* }
}
```

FIGURE 4. Insert test code for testing race condition in point-to-point blocking communication (MPI_Send/MPI_Recv).

In the case of race condition detection, when all calls arrive ACC_TEST will compare the actual message exchange with the information from our static analysis for detecting any potential race condition, as shown in the insert test code in Figure 4. ACC_TEST will insert the values from our static testing and compare them to the resulting values from the actual runtime values by using the following insert statements:

Similarly, MPI_Sendrecv will be tested by dividing each MPI_Sendrecv into MPI_Send and MPI_Recv, as we explained earlier. The following Figure 5 displays the insertion mechanism of the MPI_Sendrecv calls.

Additionally, for testing race condition in point-to-point blocking communication in (MPI_Sendrecv), ACC_TEST will also use the same insertion mechanism of the previous test as shown in Figure 4 by comparing the actual message exchange information with the information from our static analysis.

In addition, to distinguish between actual and potential deadlock in our dynamic tester, we will test our inserted code multiple times where:

- If all tested cases detect the same error, this indicates actual deadlock.
- If some cases detect errors and some not, this indicates potential deadlock, which can be affected by the execution environment and order.
- If all cases have no error, that indicates this code is deadlock-free.

```

// Insert test code for testing point-to-point blocking communication (MPI_Sendrecv)
// THE FOLLOWING LINE IS HAVE THE INSERTED TESTING CODE AFTER THE MPI_SENDRECV ****

/**ASSETRY*/ MPI_Request request_sendrecv_var_name;
/**ASSETRY*/ MPI_Status status_sendrecv_var_name;
/**ASSETRY*/ int flag_sendrecv_var_name;

// THE FOLLOWING LINE IS HAVE THE MPI_RECV INSERTED CODE ****
/**ASSETRY*/ MPI_Irecv(Buffer, Count, Data_Type, Source, Tag, MPI_Communicator, request_sendrecv_var_name);

// THE FOLLOWING LINE IS HAVE THE MPI_SEND INSERTED CODE ****
/**ASSETRY*/ MPI_Send(Buffer, Count, Data_Type, Destination, Tag, MPI_Communicator);
/**ASSETRY*/ Set a Timer
/**ASSETRY*/ MPI_Test(&request_sendrecv_var_name, &flag_sendrecv_var_name, &status_sendrecv_var_name);
/**ASSETRY*/ if (!flag_sendrecv_var_name)
/**ASSETRY*/ cout << "ERROR !! ==> THERE IS A DEADLOCK IN THE MPI_SENDRECV No: " << MPI_RECV_ID << " IN RANK NO:" << Rank << " IN LINE NO:" << line_no << endl;
/**ASSETRY*/ }
/**ASSETRY*/ else
/**ASSETRY*/ {
/**ASSETRY*/ cout << "The Tested MPI_SENDRECV NO: << MPI_SENDRECV_ID << "is Completed" << endl;
/**ASSETRY*/ cout << "The Flag Value: " << flag_sendrecv_var_name << endl;
/**ASSETRY*/ cout << "Received From Process No: " << status_sendrecv_var_name.MPI_SOURCE << endl;
/**ASSETRY*/ cout << "With TAG No: " << status_sendrecv_var_name.MPI_TAG << endl;
/**ASSETRY*/ }
}

```

FIGURE 5. Insert test code for testing deadlock in point-to-point blocking communication (MPI_Sendrecv).

For example, if we test the same connection 5 times where if the number of detected errors is 5, this indicates actual deadlock. If the detected error(s) is 1, 2, 3, or 4 out of 5, that indicates potential deadlock. Finally, if the number of detected errors is 0, this indicates deadlock-free. Our dynamic testing will inform the programmer about the error type, line number, which rank, and what MPI call has caused this error.

Finally, by using this approach, ACC_TEST minimizes the overhead from using the dynamic analysis and enhances our testing performance by decreasing slowdown, as well as testing accuracy, without extra unnecessary testing operations or inserting codes that actually cause overhead without getting accurate results.

B. POINT-TO-POINT NON-BLOCKING COMMUNICATION DETECTION

In this section, we will discuss how our testing techniques will examine and detect runtime errors related to point-to-point non-blocking communication, including MPI_Isend and MPI_Irecv. Similar to the previous point-to-point communication detection, our static approach will also collect information related to MPI_Isend and MPI_Irecv and store them for detecting some errors similar to the previous class, including mismatch and different numbers of senders and receivers, as well as analyzing the MPI_Isend/MPI_Irecv pairs.

Unlike the blocking communication, non-blocking communication has an object called request, used to identify a communications operation and its properties. This feature needs to be detected by our static phase to avoid any potential error. Our static analysis will detect request lost, that if the same request variable is used in different MPI_Isend, MPI_Irecv in the same rank, this can cause request overwrite and should be detected before it can cause further errors in related operations.

In addition, non-blocking communication will cause potential race condition, especially in the case of operations needing to be completed before sending or receiving. Therefore, the MPI_Wait calls need to be used for completing the non-blocking communication. As a result, our static tool will

investigate the targeted source code and detect any missing MPI_Wait calls. However, in the case of using MPI_Wait while the source code has a deadlock, the program will freeze, and therefore our static analysis will annotate this situation to be tested by our dynamic tester. Also, if there is a deadlock and the MPI_Wait is not used, the program will complete running with wrong results that cannot be detected by our static phase and need further dynamic testing.

In terms of detecting errors in our dynamic phase, our testing detects the deadlock in the non-blocking point-to-point connection (MPI_Isend/MPI_Irecv) by adding MPI_Test before any MPI_Wait to avoid any program freeze, because in this case the deadlock will occur in the MPI_Wait call. Also, we can detect the race condition if we found Isend and Irecv without using MPI_Wait or MPI_Test because we cannot ensure the arrival order of the threads; therefore, any potential race condition message will be issued to the programmer. The insertion mechanism of detecting deadlock will be similar to that used in the point-to-point blocking communication shown in Figure 3.

Similar to our approach of detecting race condition in the blocking communication, our dynamic tester of the non-blocking communication will also compare the actual message-receiving information with the information from our static analysis for detecting any potential race condition, as shown in Figure 4.

C. COLLECTIVE COMMUNICATION DETECTION

The third class of detection that our testing tool can target is the MPI collective communication, including blocking and non-blocking, which in our case will be MPI_Bcast and MPI_Ibcast. Our static phase will be responsible for collecting the related information needed to test and detect any runtime errors related to MPI collective communication codes. Additionally, ACC_TEST will lexically analyze and parse the targeted source code to ensure the correctness of the MPI calls, as well as detecting some errors that can be resolved during our static analysis.

The type-matching conditions for the collective operations are stricter than the corresponding conditions between sender and receiver in point-to-point [4]. Therefore, our static phase will be responsible for detecting any data type and size mismatching errors, as discussed earlier. The collective operation order will also be examined in our static phase to avoid any potential errors resulting from incorrectly ordered collective operations in the same MPI communicator, such as the example shown in Figure 6. Finally, our static phase will detect any potential deadlock that occurs as a result of not calling the MPI collective operation by all processes in the MPI communicator.

In our dynamic phase, ACC_TEST will test the MPI_Bcast during runtime to avoid any deadlock. Because of the behavior of MPI_Bcast in the case of deadlock, ACC_TEST will use inserted statements as shown in Figure 7 to test the data exchange between the broadcasts, even in the case of deadlock without facing the effect of deadlock, which causes

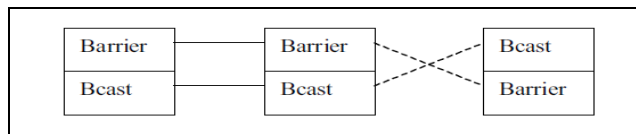


FIGURE 6. Incorrectly ordered collective operations.

```

// Insert test code for testing collective communication (MPI_Bcast)

// THE FOLLOWING LINE IS HAVE THE INSERTED TESTING CODE AFTER THE MPI_Bcast ****

//ASSERT* MPI_Request request_Bcast_var_name;
//ASSERT* MPI_Status status_Bcast_var_name;
//ASSERT* int flag_Bcast_var_name;

//ASSERT* MPI_Ibcast(&data, 1, MPI_INT, 0, MPI_COMM_WORLD, &Bc_request_2);
//ASSERT* Set a Timer
//ASSERT* MPI_Test(&Bc_request_2, &Bc_flag_2, &Bc_status_2);
//ASSERT* if (!Bc_flag_2)
//ASSERT* {
//ASSERT* cout << "ERROR :: ==>> THERE IS A DEADLOCK IN THE MPI_Bcast No:2 IN RANK NO: 2 IN LINE NO: 32 " << endl;
//ASSERT* }
//ASSERT* else
//ASSERT* {
//ASSERT* cout << "The Tested MPI_Bcast No: 2 is Completed" << endl;
//ASSERT* cout << "The Flag Value: " << Bc_flag_2 << endl;
//ASSERT* cout << "Received From The Root No: " << Bc_status_2.MPI_SOURCE << endl;
//ASSERT* }
    
```

FIGURE 7. Insert test code for testing collective communication (MPI_Bcast).

the program to freeze without knowing the actual reasons behind it.

Our dynamic phase will use the annotation from our static analysis to replace each blocking broadcast (MPI_Bcast) with a non-blocking broadcast (MPI_Ibcast) to avoid any blocking behavior, and our dynamic testing will set a timer for waiting for the broadcast calls before testing their situations. Our dynamic testing will then use MPI_Test for each MPI broadcast call and extract the actual information, including the broadcast’s source.

V. IMPLEMENTATION AND TESTING

Many experiments have been conducted to test and simulate runtime errors that can occur in MPI, and their behaviour has been studied to understand them better and discover their causes and effects on the applications. Also, several experiments have been conducted to test our proposed solution and ensure ACC_TEST’s ability to detect different types of errors in MPI, as well as covering errors from different types of MPI connections. To perform our experiments, we used an Intel(R) Core(TM) i7-7700HQ CPU (2.80GHz) with 16 GB main memory, with an NVIDIA GeForce GTX 1050 Mobile GPU, which has 768 NVIDIA CUDA cores, 4 GB GDDR5 RAM, and memory speed of 7 Gbps.

More than 40 MPI benchmarks from four different benchmark suites have been used to evaluate ACC_TEST, including NAS Parallel Benchmarks [32], OSU Micro-Benchmarks [33], EPCC [34], and mpiBench [35]. Table 1 shows some statistics from the chosen MPI benchmarks, including the number of MPI calls and each class of

TABLE 1. MPI statistics from the chosen benchmarks.

Benchmarks	# lines	# MPI Calls	# Point-to-Point Calls		# Collective Calls	
			Blocking	Non-Blocking	Blocking	Non-Blocking
NAS						
DT	755	17	8	0	0	0
IS	1,186	21	2	1	9	0
EPCC						
PingPong	468	19	12	0	5	0
PingPing	437	16	6	3	5	0
Broadcast	203	8	0	0	6	0
ParallelEnvironmet	393	27	10	1	2	0
mpiBench						
mpiBench	1,132	49	0	0	23	0
OSU						
osu_latency_mt	351	24	8	0	2	0
osu_mbw_mr	332	28	4	2	5	0
osu_bcast	131	17	0	0	6	0
osu_ibcast	179	28	2	0	5	2

communication to include blocking, non-blocking point-to-point, and collective communications.

In this section, our implementation and testing for testing MPI-based programs will be explained to show our tool’s ability to detect some errors related to MPI applications. We will show examples of some errors from each type of MPI communication.

A. POINT-TO-POINT BLOCKING COMMUNICATION DETECTION

Our static analysis will start by determining each MPI send-and-recv and their locations in the source code. Also, we store their information in a data structure as shown in Figure 8, and we also determine to which MPI rank they belong for determining the message direction. Our static analysis will create several tables for storing the related send-and-recv calls based on their type, storing their related information such as rank and type, communicator, tag, and line number. Then we compare this information to the static phase, searching for any missing or potential for race condition or deadlock as well as mismatching.

ACC_TEST will determine the MPI_Send/MPI_Recv pair, which will be used to detect any differences between the numbers of send-and-receives, as shown in Figure 9. This pairing will also examine the message tag to detect any unmatched message pairing. Our static analysis will check any message leaks (sending message without receive) or inconsistent types on the sender and receiver for the same message.

In the case of having more senders than receivers, which is considered as a lack of resources as shown in Figure 10, this can also lead to a potential race condition that will be detected by our dynamic tester to know the exact error. On the other hand, when the number of receivers is more than

```
##### MPI SEND INFORMATION #####
----- MPI SEND LIST -----
-ID - -Line - -Data - -Count - -Data Type - -From - -To - -Tag - -Communicator - -Match Recv - -Recv Line -
0 44 4a 8 MPI_INT 0 1 0 MPI_COMM_WORLD yes 56
1 47 4b 8 MPI_DOUBLE 0 2 0 MPI_COMM_WORLD yes 92
2 76 4a 8 MPI_INT 1 3 0 MPI_COMM_WORLD yes 144
3 79 4sum_1 1 MPI_INT 1 4 0 MPI_COMM_WORLD No 0
4 112 4b 8 MPI_INT 2 3 0 MPI_COMM_WORLD yes 144
5 115 4sum_2 1 MPI_INT 2 4 0 MPI_COMM_WORLD No 0
##### MPI RECEIVE INFORMATION #####
----- MPI RECEIVE LIST -----
-ID - -Line - -Data - -Count - -Data Type - -To - -From - -Tag - -Communicator - -Status - -Match Send -
0 56 4a 8 MPI_INT 1 0 0 MPI_COMM_WORLD testatus yes
1 92 4b 8 MPI_INT 2 0 0 MPI_COMM_WORLD testatus yes
2 144 4a 8 MPI_INT 3 1 0 MPI_COMM_WORLD testatus1 yes
3 144 4b 8 MPI_INT 3 2 0 MPI_COMM_WORLD testatus2 yes
#####
```

FIGURE 8. Information from MPI_Send and MPI_rcv-Related information.

```
#####
#### STATIC TESTING LIST ####
#### MPI ERROR NUMBER ----> (1) <----
#### ERROR TYPE ----> MPI BLOCING COMMUNICATION ERROR
#### THIS ERROR IS --> : DIFFERENT NUMBE OF MPI_SEND/MPI_RECV ERROR
#### THIS ERROR OCCURS IN LINE NO.(**)*****
#### Extra Info. ###
THE TOTAL NUMBER OF SENDERS IS NOT EQUAL TO THE TOTAL NUMBER OF RECEIVERS
-- TOTAL NUMBER OF SENDERS IS MORE THAN THE TOTAL NUMBER OF RECEIVERS, THEREFORE,
THEY WILL BE MESSAGES SENDED WITHOUT BEEN RECEIVED, WHICH CAN CAUSE POTENTIAL ERRORS
#####
```

FIGURE 9. Error message in the case of different numbers of MPI_Send and MPI_Recv.

```
#####
#### STATIC TESTING LIST ####
#### MPI ERROR NUMBER ----> (3) <----
#### ERROR TYPE ----> MPI BLOCING COMMUNICATION ERROR
#### THIS ERROR IS --> : RESOURCE LEAKS : SENDER WITHOUT RECEIVER
#### THIS ERROR OCCURS IN LINE NO.(79)*****
#### Extra Info. ###
The SENDER ID: ( 3 ) At Line No. ( 79 ) IS NOT MATCHING WITH ANY RECEIVER WHICH CAUSE LEAK OF RESOURCE
#####
#### STATIC TESTING LIST ####
#### MPI ERROR NUMBER ----> (4) <----
#### ERROR TYPE ----> MPI BLOCING COMMUNICATION ERROR
#### THIS ERROR IS --> : RESOURCE LEAKS : SENDER WITHOUT RECEIVER
#### THIS ERROR OCCURS IN LINE NO.(115)*****
#### Extra Info. ###
The SENDER ID: ( 5 ) At Line No. ( 115 ) IS NOT MATCHING WITH ANY RECEIVER WHICH CAUSE LEAK OF RESOURCE
#####
```

FIGURE 10. Leak of resource error caused by having sender without receiver.

the number of senders, this will lead to potential deadlock because they will be processes waiting to receive a message without receiving it; this will be annotated for further testing in our dynamic phase.

In addition, our static analysis will detect any mismatching in data types and message sizes, as shown in Figure 11. Additionally, our static analyzer will examine the relationship between OpenACC and MPI directives to determine the mismatch between the data movement between the OpenACC and MPI directives, not only between the MPI_Send and MPI_Recv calls, but also in the code; for example the programmer defines a variable as an INT in his code and passes this variable in an MPI call, as MPI_CHAR ACC_TEST will detect this error.

```
#####
#### STATIC TESTING LIST ####
#### MPI ERROR NUMBER ----> (2) <----
#### ERROR TYPE ----> MPI BLOCING COMMUNICATION ERROR
#### THIS ERROR IS --> : MISMATCH DATA TYPE ERROR
#### THIS ERROR OCCURS IN LINE NO.(19)*****
#### Extra Info. ###
MPI_SEND/MPI_RECV MISMATCH DATA TYPE ERROR
MPI UNMATCH DATA TYPE BETWEEN SENDER ID: (0) AT LINE (37) AND RECEIVER ID: (0) AT LINE (49)
THE SENDER DATA TYPE IS : MPI_BYTE, AND THE RECVER DATA TYPE IS : MPI_INT
#####
#### STATIC TESTING LIST ####
#### MPI ERROR NUMBER ----> (3) <----
#### ERROR TYPE ----> MPI BLOCING COMMUNICATION ERROR
#### THIS ERROR IS --> : MISMATCH DATA TYPE ERROR
#### THIS ERROR OCCURS IN LINE NO.(22)*****
#### Extra Info. ###
MPI_SEND/MPI_RECV MISMATCH MESSAGE SIZE ERROR
MPI UNMATCH MESSAGE SIZE BETWEEN SENDER ID: (2) AT LINE (67) AND RECEIVER ID: (2) AT LINE (110)
THE SENDER MESSAGE SIZE IS : 100, THE RECVER MESSAGE SIZE IS : 20
#####
```

FIGURE 11. Unmatched errors detected by our static analysis.

```
#####
#### STATIC TESTING LIST ####
#### MPI ERROR NUMBER ----> (2) <----
#### ERROR TYPE ----> MPI BLOCING COMMUNICATION ERROR
#### THIS ERROR IS --> : POTENTIAL ERROR
#### THIS ERROR OCCURS IN LINE NO.(41)*****
#### Extra Info. ###
THERE IS A PROCESS SEND 2 MSGESS TO THE SAME DESTINATION WITHOUT TAG THEM --
MPI_SEND AT LINE : (38) AND LINE : (41) HAVE BEEN SEND FROM PROCESS : (0) TO PROCESS: (3)
WHICH HAVE BEEN RECEIVED AT LINE : (116) AND LINE: (117)
#####
```

FIGURE 12. Potential race condition detected by our static tester.

In addition, our static analysis will detect any mismatching in data types and message sizes, as shown in Figure 11. Additionally, our static analyzer will examine the relationship between OpenACC and MPI directives to determine the mismatch between the data movement between the OpenACC and MPI directives, not only between the MPI_Send and MPI_Recv calls, but also in the code; for example the programmer defines a variable as an INT in his code and passes this variable in an MPI call, as MPI_CHAR ACC_TEST will detect this error.

In terms of race condition detection, our static analysis will detect any case of several messages sent to the same destination with the same tag number, which can cause a race condition. Figure 12 shows an error message indicating potential race condition, and further detection by our dynamic phase is needed.

In terms of deadlock detection, our static phase detects actual and potential deadlock based on our static analysis of the targeted source code. One potential error situation is using the wildcard receive. Our static analysis will also detect any wildcard receive with any source or any tag and examine them to avoid any potential deadlock or race condition and annotate them to be detected by our dynamic phase. Figure 13 shows error messages related to deadlock detection in our static phase.

Figure 14 also shows a potential deadlock due to wildcard receives that need further investigation by our dynamic tester to determine the exact error type, including deadlock or race condition based on the execution environment and the source code analysis. Another case of potential deadlock can be caused by data exchange between two processes, which can

```
*****
*** STATIC TESTING LIST ***
*** MPI ERROR NUMBER ----> (2) <----
*** ERROR TYPE ----> MPI BLOCING COMMUNICATION ERROR
*** THIS ERROR IS --> : DEADLOCK
*** THIS ERROR OCCURS IN LINE NO. (50)*****
*** Extra Info. ***
THE MPI_RECV ID (0) AT LINE (50) IS NOT MATCHING WITH ANY SENDER WHICH WILL CAUSE ACTUAL DEADLOCK
*****
```

FIGURE 13. Actual deadlock due to receiving without match sender.

```
*****
*** STATIC TESTING LIST ***
*** MPI ERROR NUMBER ----> (3) <----
*** ERROR TYPE ----> MPI BLOCING COMMUNICATION ERROR
*** THIS ERROR IS --> : POTENTIAL ERROR
*** THIS ERROR OCCURS IN LINE NO. (115)*****
*** Extra Info. ***
WILDCARD RECEIVE POTENTIAL ERROR NEED FURTHER INVESTIGATION BY OUR DYNAMIC TESTER
BECAUSE OF THE WILDCARD (ANY SOURCE) IN MPI_RECV ID (2) AT LINE (115)
*****
*** STATIC TESTING LIST ***
*** MPI ERROR NUMBER ----> (4) <----
*** ERROR TYPE ----> MPI BLOCING COMMUNICATION ERROR
*** THIS ERROR IS --> : POTENTIAL ERROR
*** THIS ERROR OCCURS IN LINE NO. (117)*****
*** Extra Info. ***
WILDCARD RECEIVE POTENTIAL ERROR NEED FURTHER INVESTIGATION BY OUR DYNAMIC TESTER
BECAUSE OF THE WILDCARD (ANY SOURCE) IN MPI_RECV ID (3) AT LINE (117)
*****
```

FIGURE 14. Potential deadlock because of the wildcard receive.

```
*****
*** STATIC TESTING LIST ***
*** MPI ERROR NUMBER ----> (1) <----
*** ERROR TYPE ----> MPI BLOCING COMMUNICATION ERROR
*** THIS ERROR IS --> : POTENTIAL ERROR
*** THIS ERROR OCCURS IN LINE NO. (51)*****
*** Extra Info. ***
DATA EXCHANGE BETWEEN TWO PROCESSES LEAD TO POTENTIAL DEADLOCK ERROR
SENDER ID ( 2 ) AT LINE (51) THE MESSAGE SEND FROM PROCESS ( 1 ) TO PROCESS ( 2 )
AND SENDER ID ( 3 ) AT LINE (82) THE MESSAGE SEND FROM PROCESS ( 2 ) TO PROCESS ( 1 )
*****
```

FIGURE 15. Data exchange leads to potential deadlock.

```
***** MPI SENDRECV INFORMATION *****
----- MPI SENDRECV LIST -----
ID - Line - SData - SCount - SType - To - STag - RData - RCount - RType - From - STag - Communicator - Status -
0 19 buffer 10 MPI_INT 2 222 buffer2 10 MPI_INT 1 123 MPI_COMM_WORLD setatus
1 20 buffer 10 MPI_INT 1 123 buffer1 10 MPI_DOUBLE 2 212 MPI_COMM_WORLD setatus
2 21 buffer 10 MPI_INT 5 123 buffer2 10 MPI_INT 3 123 MPI_COMM_WORLD setatus
3 22 buffer 100 MPI_INT 4 333 buffer2 10 MPI_INT 5 123 MPI_COMM_WORLD setatus
4 23 buffer 10 MPI_INT 3 123 buffer2 10 MPI_INT 4 333 MPI_COMM_WORLD setatus
*****
```

FIGURE 16. MPI_Sendrecv information from Our static analysis.

be detected by our static analysis in Figure 15, and this error will be annotated to be tested by our dynamic tester.

Another case of point-to-point blocking communication is the MPI_Sendrecv calls, which will be examined and analyzed just like the previous MPI_Send and MPI_Recv calls. Figure 16 displays the information collection for the MPI_Sendrecv calls. The error detection will be conducted as described in the previous MPI calls because they display the same behavior but with different structures. An example of detected error in MPI_Sendrecv is shown in Figure 17.

This is for detecting deadlock in point-to-point blocking communication, as we explained previously. Figure 18 shows

```
*****
*** STATIC TESTING LIST ***
*** MPI ERROR NUMBER ----> (1) <----
*** ERROR TYPE ----> MPI BLOCING COMMUNICATION ERROR
*** THIS ERROR IS --> : MISMATCH DATA TYPE ERROR
*** THIS ERROR OCCURS IN LINE NO. (19)*****
*** Extra Info. ***
MPI UNMATCH DATA TYPE BETWEEN SENDRECV ID: 0 IN LINE NO. (19 ) AND SENDRECV ID: 1 IN LINE NO. (20)
THE SENDER DATA TYPE IS : MPI_INT, AND THE RECEIVER DATA TYPE IS : MPI_DOUBLE
*****
*** STATIC TESTING LIST ***
*** MPI ERROR NUMBER ----> (2) <----
*** ERROR TYPE ----> MPI BLOCING COMMUNICATION ERROR
*** THIS ERROR IS --> : MISMATCH DATA SIZE ERROR
*** THIS ERROR OCCURS IN LINE NO. (22)*****
*** Extra Info. ***
MPI UNMATCH MESSAGE SIZE BETWEEN SENDRECV ID: 3 IN LINE NO. (22 ) AND SENDRECV ID: 4 IN LINE NO. (23)
THE SENDER MESSAGE SIZE IS: 100, THE RECEIVER MESSAGE SIZE IS : 10
*****
```

FIGURE 17. MPI_Sendrecv unmatched errors.

```
// THE FOLLOWING LINE IS HAVE THE INSERTED TESTING CODE AFTER THE MPI_RECV ****
MPI_Request request_0;
MPI_Status status_0;
int flag_0;

 ierr = MPI_irecv(&A, N, MPI_INT, 0, 0, MPI_COMM_WORLD, &request_0);
 int mpi_count_down_0 = 0;
 MPI_Test(&request_0, &flag_0, &status_0);
 if (!flag_0)
 {
  cout << "ERROR :: ==> THERE IS A DEADLOCK IN THE MPI_RECV No:0 IN RANK NO: 1 IN LINE NO: 56 " << endl;
 }
 else
 {
  cout << "The Tested MPI_RECV NO: 0 is Completed" << endl;
  cout << "The Flag Value: " << flag_0 << endl;
  cout << "Received From Process No: " << status_0.MPI_SOURCE << endl;
  cout << "With TAG No: " << status_0.MPI_TAG << endl;
 }
 }
```

FIGURE 18. Instrumented inserted test code for testing point-to-point blocking communication.

```
*****
*** DYNAMIC TESTING LIST ***
*** MPI ERROR NUMBER ----> (2) <----
*** ERROR TYPE ----> MPI BLOCING COMMUNICATION ERROR
*** THIS ERROR IS --> : DEADLOCK
*** THIS ERROR OCCURS IN LINE NO. (92)*****
*** Extra Info. ***
THERE IS A DEADLOCK IN THE MPI_RECV No:1 IN RANK NO: 2 IN LINE NO: 92
*****
```

FIGURE 19. Deadlock detected by our dynamic tester.

```
The Tested MPI_RECV NO: 3 is Completed
The Flag Value: 1
Received From Process No: 1
With TAG No: 0

The Tested MPI_RECV NO: 4 is Completed
The Flag Value: 1
Received From Process No: 2
With TAG No: 0

THERE IS NO ERROR BECAUSE THE MPI_RECV No.(3) SHOULD BE RECEIVED FROM RANK (1) WITH TAG (0)
THERE IS NO ERROR BECAUSE THE MPI_RECV No.(4) SHOULD BE RECEIVED FROM RANK (2) WITH TAG (0)
```

FIGURE 20. The actual information of the received message from our dynamic tester.

the instrumented inserted code used for testing the deadlock in the point-to-point blocking communications for each MPI_Recv, and Figure 19 displays the related error message.

In the case of race condition detection, when all calls arrive, ACC_TEST will compare the actual message exchange with the information from our static analysis to detect any potential race condition. Figure 20 shows the actual information from the dynamic tester and the process of comparing them with the static analyzer information, which will be shown in our


```
*****
***# DYNAMIC TESTING LIST ***#
***# MPI ERROR NUMBER ----> (1) <----
***# ERROR TYPE --> MPI BLOCING COMMUNICATION ERROR
***# THIS ERROR IS --> : RACE CONDITION
***# THIS ERROR OCCURS IN LINE NO.(143)*****
***# Extra Info. ***#
THERE IS A RACE CONDITION IN THE MPI_RECV No: 3 IN RANK NO: 3 IN LINE NO: 143
BECAUSE THE MPI_RECV No.(3) RECEIVED FROM RANK (2) WITH TAG (0)
BUT IT SHOULD BE RECEIVED FROM RANK (1) WITH TAG (0)
*****
```

FIGURE 21. Race condition detected by our dynamic tester.

```
// THE FOLLOWING LINE IS HAVE THE INSERTED TESTING CODE AFTER THE MPI_SENDRECV ****
MPI_Request request_sendrecv_0;
MPI_Status status_sendrecv_0;
int flag_sendrecv_0;
// THE FOLLOWING LINE IS HAVE THE MPI_RECV INSERTED CODE ****
MPI_Irecv(buffer2, 10, MPI_INT, 1, 123, MPI_COMM_WORLD, &request_sendrecv_0);
// THE FOLLOWING LINE IS HAVE THE MPI_SEND INSERTED CODE ****
MPI_Send(buffer, 10, MPI_INT, 2, 222, MPI_COMM_WORLD);
MPI_Test(&request_sendrecv_0, &flag_sendrecv_0, &status_sendrecv_0);
if (!flag_sendrecv_0)
{
    cout << "ERROR : --> THERE IS A DEADLOCK IN THE MPI_SENDRECV No:0 IN RANK NO: 1 IN LINE NO: 19 " << endl;
}
else
{
    cout << "The Tested MPI_SENDRECV NO: 0 is Completed" << endl;
    cout << "The Flag Value:" << flag_sendrecv_0 << endl;
    cout << "Received From Process No:" << status_sendrecv_0.MPI_SOURCE << endl;
    cout << "With TAG No: " << status_sendrecv_0.MPI_TAG << endl;
}
}
```

FIGURE 22. Instrumented inserted code for testing deadlock MPI_Sendrecv.

historical log file. In case of an error, the error message will be displayed in the dynamic error file, as shown in Figure 21.

Similarly, MPI_Sendrecv will be tested by dividing each MPI_Sendrecv into MPI_Send and MPI_Recv, and test them as we explained earlier. Figure 22 shows the instrumented inserted test code to be used by our dynamic tester.

B. POINT-TO-POINT NON-BLOCKING COMMUNICATION DETECTION

In this section, we will discuss how our testing tool will examine and detect runtime errors related to point-to-point non-blocking communication, including MPI_Isend and MPI_Irecv. Figure 23 shows the collective data from our static analysis for non-blocking communication, and Figure 24 shows the mismatching error message from our static tester for non-blocking communication. Figure 25 displays a lack of resources by having a sender without a receiver. Finally, the wildcard receive will be tested as the previous detection technique in the blocking communication.

Unlike blocking communication, non-blocking communication has an object called request, which is used to identify a communication operation and its properties. This feature must be detected by our static phase to avoid any potential error. Our static analysis will detect request lost, that is, if the same request variable is used in different MPI_Isend, MPI_Irecv in the same rank, this can cause request overwrite and should be detected before it can cause further error when it needs to be used for related operations, as displayed in Figure 26.

Also, non-blocking communication will cause a potential race condition, especially in the case of operations to be

```
***** MPI SEND INFORMATION *****
----- MPI SEND LIST -----
ID - Line - Data - Count - Data Type - From - To - Tag - Communicator - Request - Match Recv - Recv Line -
0 39 4A N MPI_INT 0 1 0 MPI_COMM_WORLD &request1 yes 54
1 41 4B N MPI_INT 0 2 0 MPI_COMM_WORLD &request2 yes 66
2 72 4A N MPI_INT 1 3 0 MPI_COMM_WORLD &request yes 118
3 104 4B N MPI_INT 2 3 0 MPI_COMM_WORLD &request yes 119
*****
***** MPI RECEIVE INFORMATION *****
----- MPI RECV LIST -----
ID - Line - Data - Count - Data Type - To - From - Tag - Communicator - Request - Match Send -
0 54 4A N MPI_INT 1 0 0 MPI_COMM_WORLD &request yes
1 56 4B N MPI_INT 2 0 0 MPI_COMM_WORLD &request yes
2 118 4A N MPI_INT 3 1 0 MPI_COMM_WORLD &request1 yes
3 119 4B N MPI_INT 3 2 0 MPI_COMM_WORLD &request2 yes
*****
```

FIGURE 23. Collected information related to non-Blocking sends and receives.

```
*****
***# STATIC TESTING LIST ***#
***# MPI ERROR NUMBER ----> (1) <----
***# ERROR TYPE --> MPI NONBLOCING COMMUNICATION ERROR
***# THIS ERROR IS --> : MPI_ISEND/MPI_IRecv MISMATCH DATA TYPE ERROR
***# THIS ERROR OCCURS IN LINE NO.(19)*****
***# Extra Info. ***#
MPI UNMATCH DATA TYPE BETWEEN SENDER ID: 0 AT LINE: 19 AND RECVER ID: 0 AT LINE: 25
THE SENDER DATA TYPE IS : MPI_DOUBLE, AND THE RECVER DATA TYPE IS : MPI_INT
*****
```

FIGURE 24. Mismatching error message in non-blocking communication.

```
*****
***# STATIC TESTING LIST ***#
***# MPI ERROR NUMBER ----> (3) <----
***# ERROR TYPE --> MPI NONBLOCING COMMUNICATION ERROR
***# THIS ERROR IS --> : RESOURCE LEAKS : SENDER WITHOUT RECEIVER
***# THIS ERROR OCCURS IN LINE NO.(29)*****
***# Extra Info. ***#
The SENDER ID: ( 0 ) At Line No. ( 29 ) IS NOT MATCHING WITH ANY RECEIVER WHICH CAUSE LEAK OF RESOURCE
*****
```

FIGURE 25. Leak of resources in non-blocking communication.

```
*****
***# STATIC TESTING LIST ***#
***# MPI ERROR NUMBER ----> (1) <----
***# ERROR TYPE --> MPI NONBLOCING COMMUNICATION ERROR
***# THIS ERROR IS --> : RESOURCE LEAKS : REQUEST LOST
***# THIS ERROR OCCURS IN LINE NO.(29)*****
***# Extra Info. ***#
THERE IS TWO REQUESTS BY THE SAME NAME IN THE SAME RANK NO:0
WHICH CAN CAUSE REQUEST OVERWRITE AND CAUSE FURTHER ERROR IF THEY USED IN ANY RELATED OPERATIONS
THESE TWO REQUESTS ARE IN THE MPI_ISEND AT LINE : 29 AND AT THE MPI_IRecv AT LINE NO: 38
*****
```

FIGURE 26. Request lost potential error.

```
*****
***# STATIC TESTING LIST ***#
***# MPI ERROR NUMBER ----> (2) <----
***# ERROR TYPE --> MPI NONBLOCING COMMUNICATION ERROR
***# THIS ERROR IS --> : Deadlock
***# THIS ERROR OCCURS IN LINE NO.(38)*****
***# Extra Info. ***#
The IRECV ID: ( 0 ) At Line No. ( 38 ) IS NOT MATCHING WITH ANY SENDER WHICH CAUSE ACTUAL DEADLOCK
THIS DEADLOCK WILL HAPPENED IN THE USE OF MPI_WAIT, HOWEVER, IF THERE IS NO WAIT, IT WILL CAUSE RACE CONDITION
*****
```

FIGURE 27. Deadlock detected in non-Blocking communication.

completed before sending or receiving. Therefore, the MPI_Wait calls needed to be used for completing the non-blocking communication. Figure 27 shows deadlock detection by our static testing and indicates not having MPI_Wait call.

In terms of detecting errors in our dynamic phase, our dynamic testing detects the deadlock in the non-blocking point-to-point connection (MPI_Isend/MPI_Irecv) by adding MPI_Test before any MPI_Wait to avoid any program freeze

```

MPI_Wait(&Rrequest31, &Rstatus31);
MPI_Wait(&Rrequest32, &Rstatus32);

// test the first recv
MPI_Test(&Rrequest31, &Rflag31, &Rstatus31);
if (Rflag31 == 0)
{
    cout << "The receive operation 1 is not yet complete" << endl;
}
else
{
    cout << "The receive operation 1 is complete" << endl;
    cout << "The flag Value: " << Rflag31 << endl;
    cout << "The received message tag is : " << Rstatus31.MPI_TAG << endl;
    cout << "The received message Source is : " << Rstatus31.MPI_SOURCE << endl;
}
// test the second recv
MPI_Test(&Rrequest32, &Rflag32, &Rstatus32);
if (Rflag32 == 0)
{
    cout << "The receive operation 2 is not yet complete" << endl;
}
else
{
    cout << "The receive operation 2 is complete" << endl;
    cout << "The flag Value: " << Rflag32 << endl;
    cout << "The received message tag is : " << Rstatus32.MPI_TAG << endl;
    cout << "The received message Source is : " << Rstatus32.MPI_SOURCE << endl;
}
    
```

FIGURE 28. Instrumented inserted test code for detecting errors in non-blocking communication.

```

***** Collective Communications *****
The MPI BROADCAST )
The MPI Bcast No. ( 0 ) found at Line No. --> 19
The MPI Bcast No. ( 1 ) found at Line No. --> 25
The MPI Bcast No. ( 2 ) found at Line No. --> 32
The MPI Bcast No. ( 3 ) found at Line No. --> 38
The Total Number of MPI Bcast = 4

***** MPI BCAST INFORMATION *****

----- MPI_BCAST LIST -----
- ID - - Line - - Data - - Count - - Type - - Root - - Communicator - - Rank -
0 - 19 - sdata - 1 - MPI_INT - 0 - MPI_COMM_WORLD - 0
1 - 25 - sdata - 1 - MPI_INT - 0 - MPI_COMM_WORLD - 1
2 - 32 - sdata - 1 - MPI_INT - 0 - MPI_COMM_WORLD - 2
3 - 38 - sdata - 1 - MPI_INT - 0 - MPI_COMM_WORLD - 2
    
```

FIGURE 29. MPI collective communication information collected from our static analysis.

because in this case, deadlock will occur in the MPI_Wait call. We can also detect the race condition if we found Isend and Irecv without using MPI_Wait or MPI_Test because we cannot ensure the arrival order of the threads; therefore, any potential race condition message will be issued to the programmer. The insertion mechanism of detecting deadlock will be similar to that used in the point-to-point blocking communication shown in Figure 3. Figure 28 shows the instrumented inserted code in the case of having two MPI_Irecv.

Similar to our approach to detecting a race condition in the blocking communication, our dynamic tester of the non-blocking communication will also compare the actual message receiving information to that from our static analysis for detecting any potential race condition, as shown in Figure 4.

C. COLLECTIVE COMMUNICATION DETECTION

Our static phase will be responsible for collecting information needed to test and detect any runtime errors related to MPI collective communication codes, as shown in Figure 29. ACC_TEST will also lexically analyze and parse the targeted source code to ensure the correctness of the MPI calls, as well as detecting errors that can be resolved during our static analysis.

```

*****
*** STATIC TESTING LIST ***
*** MPI ERROR NUMBER ----> (1) <----
*** ERROR TYPE ----> MPI COLLECTIVE COMMUNICATION ERROR
*** THIS ERROR IS --> : POTENTIAL ERROR
*** THIS ERROR OCCURS IN LINE NO.(**)*****
*** Extra Info. ***
THERE IS A POTENTIAL DEADLOCK IN RANK NO: 3 BECAUSE THE MPI_BCAST HAS NOT BEEN
CALLED IN THIS PROCESS WHICH CAN CAUSE POTENTIAL ERROR.
*****
    
```

FIGURE 30. Potential deadlock in MPI collective communication.

TABLE 2. Our hybrid testing tool error coverage for MPI.

MPI Errors	Detected by Our Static Approach*	Detected by Our Dynamic Approach*	Detected by Our Hybrid Approach*
Point-to-point Blocking/ Non-blocking Communication			
Illegal MPI Calls	√	X	√
Data Type Mismatching	√	X	√
Data Size Mismatching	√	X	√
Resource Lacks	√	X	√
Request Lost	√	X	√
Inconsistence Send/Recv Pairs	√	X	√
Wildcard Receive	√	X	√
Race Condition	P	P	√
Deadlock	P	P	√
Collective Blocking/ Non-blocking Communication			
Illegal MPI Calls	√	X	√
Data Type Mismatching	√	X	√
Data Size Mismatching	√	X	√
Incorrect Ordered Collective Operation	P	√	√
Race Condition	P	P	√
Deadlock	P	P	√

* The symbols are √: Fully Detected; P: Partially Detected; X: Not Detected.

Finally, our static phase will detect any potential deadlock that occurs as a result of not calling the MPI collective operation by all processes in the MPI communicator; an example of the error message for detecting this error is shown in Figure 30.

As we explained in Section 4, our dynamic phase will use the annotation from our static analysis to replace each blocking broadcast (MPI_Bcast) with a no-blocking broadcast (MPI_Ibcast). Figure 31 demonstrates an example of the instrumented inserted test code for detecting errors in the MPI collective communications, and Figure 32 shows an MPI collective communication deadlock error detected in our dynamic phase.

VI. DISCUSSION AND EVALUATION

Table 2 demonstrates our hybrid testing tool’s ability to detect MPI errors based on their types, similar to the approach used to evaluate ACC_TEST’s ability to detect OpenACC errors. We noticed in Table 2 that ACC_TEST tried to detect errors by our static approach as much as possible to decrease overhead from the dynamic testing approach and therefore enhanced our testing performance. However, race condition and deadlock are partially detected by our static testing

```
// THE FOLLOWING LINE IS HAVE THE INSERTED TESTING CODE AFTER THE MPI_Bcast ****
MPI_Request Bc_request_0;
MPI_Status Bc_status_0;
int Bc_flag_0;

MPI_Ibcast(&data, 1, MPI_INT, 0, MPI_COMM_WORLD, &Bc_request_0);
MPI_Test(&Bc_request_0, &Bc_flag_0, &Bc_status_0);
if (!Bc_flag_0)
{
    cout << "ERROR :: ==>> THERE IS A DEADLOCK IN THE MPI_Bcast No:0 IN RANK NO: 0 IN LINE NO: 19 " << endl;
}
else
{
    cout << "The Tested MPI_Bcast No: 0 is Completed" << endl;
    cout << "The Flag Value: " << Bc_flag_0 << endl;
    cout << "Received From The Root No: " << Bc_status_0.MPI_SOURCE << endl;
}
}
```

FIGURE 31. Instrumented inserted statements for detecting MPI collective communication errors.

```
#####
*** DYNAMIC TESTING LIST ***
*** MPI ERROR NUMBER ----> (1) <----
*** ERROR TYPE ----> MPI COLLECTIVE COMMUNICATION ERROR
*** THIS ERROR IS --> : Deadlock
*** THIS ERROR OCCURS IN LINE NO. (38)*****
*** Extra Info. ***
THERE IS A DEADLOCK IN THE MPI_Bcast No:3 IN RANK NO: 2 IN LINE NO: 38
#####
```

FIGURE 32. Deadlock Detected in our dynamic phase.

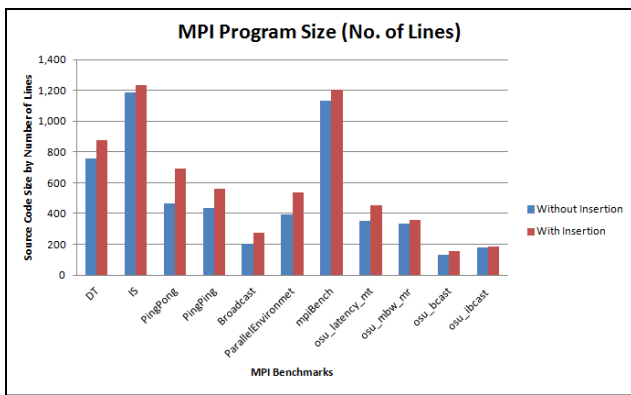


FIGURE 33. MPI program size overhead (by number of lines).

approach and need further investigation by our dynamic testing approach due to their behavior, and they were affected by the execution environment and sequence. As a result, these errors were detected using our hybrid testing approach.

Figure 33 shows the size overhead that resulted from the insertion mechanism for executing the dynamic testing to detect runtime errors that cannot be detected by the static approach by using Equation 1, as shown at the bottom of this page, which measures size overhead.

Also, Figure 35 shows the testing time needed to conduct the static testing approach on the MPI-related program with our hybrid-testing tool. The average overhead in the number of lines added is 22%, and 29% size overhead in bytes, and the average testing time for the MPI-related program is 17 milliseconds.

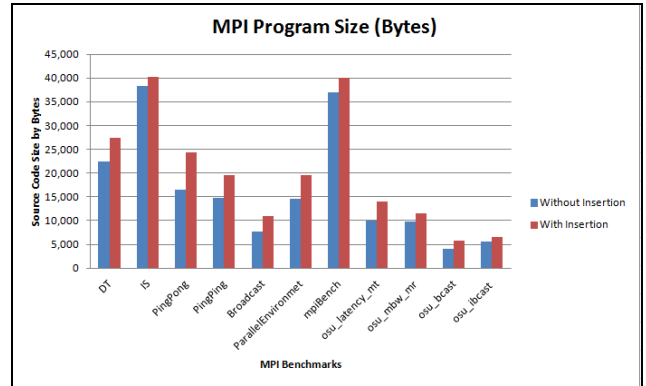


FIGURE 34. MPI program size overhead (by bytes).

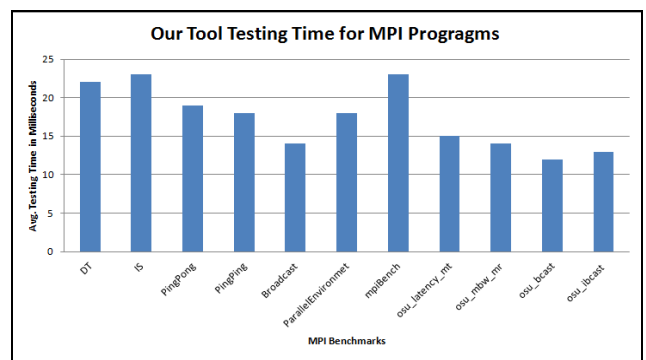


FIGURE 35. Testing time for MPI related program in milliseconds.

As we noticed in the previous Figures 33 and 34, the benchmark PingPong has the largest overhead at 48%; that is because it has the largest number of MPI point-to-point blocking calls to be tested by our dynamic approach. Based on our results, the range of overheads size varies based on the behavior of the insertion statements.

ACC_TEST minimizes the size overhead when testing MPI-related programs because we only add the insertion statements when needed and only on the MPI receiver side. We avoid adding unnecessary messaging (communications) to test the connection between senders and receivers to detect deadlock, unlike the research that suggested adding (MPI_Isend) before any send and (MPI_Irecv) before any receive [36]. Even if the connection seems to be deadlock-free after testing, the connection for any reason (non-programmatic fault) can cause the message not to arrive, which means the detected message itself has not been tested. MPI_Isend and MPI_Irecv can also cause deadlock or a race condition if there is any error or if the MPI_Wait has not been used, while using them in the insertion mechanism. Therefore, we choose to test the arrival of the detected message

$$Size\ Overhead = \frac{Size\ with\ inserted\ test\ code - Size\ without\ inserted\ test\ code}{Size\ without\ inserted\ test\ code} \quad (1)$$

without adding overhead or sending unnecessary messages, which will affect system performance and testing time.

VII. CONCLUSIONS AND FUTURE WORK

Despite the fact that there are many testing tools that target MPI, there is still much work to be done, primarily for covering more errors as well as reducing the execution and size overheads resulting from dynamic testing techniques.

Our testing tool ACC_TEST has used hybrid testing techniques combining both static and dynamic techniques for detecting errors at lower cost and overheads. ACC_TEST can cover errors from each type of MPI communication because the testing tools previously mentioned in our related work did not cover some errors or only focused on race condition and deadlocks. Finally, ACC_TEST can be integrated for testing the dual-programming model MPI + X.

In our future work, we will create a hybrid testing tool for the dual-programming model MPI + OpenACC. Our new version of ACC_TEST will have the ability to detect run-time errors when using the hybrid programming model in a heterogeneous architecture.

ACKNOWLEDGMENT

This project was funded by the Deanship of Scientific Research (DSR), King Abdulaziz University, Jeddah, under grant No. (RG-9-611-40). The authors, therefore, gratefully acknowledge the DSR technical and financial support.

REFERENCES

- [1] A. M. Alghamdi and F. E. Eassa, "OpenACC errors classification and static detection techniques," *IEEE Access*, vol. 7, pp. 113235–113253, 2019, doi: [10.1109/ACCESS.2019.2935498](https://doi.org/10.1109/ACCESS.2019.2935498).
- [2] A. M. Alghamdi and F. Elbouraey, "A parallel hybrid-testing tool architecture for a dual-programming model," *Int. J. Adv. Comput. Sci. Appl.*, vol. 10, no. 4, pp. 394–400, 2019, doi: [10.14569/IJACSA.2019.0100448](https://doi.org/10.14569/IJACSA.2019.0100448).
- [3] A. M. Alghamdi and F. E. Eassa, "Parallel hybrid testing tool for applications developed by using MPI + OpenACC dual-programming model," *Adv. Sci., Technol. Eng. Syst. J.*, vol. 4, no. 2, pp. 203–210, 2019, doi: [10.25046/aj040227](https://doi.org/10.25046/aj040227).
- [4] Message Passing Interface Forum. (2017). *MPI Forum*. [Online]. Available: <http://mpi-forum.org/docs/>
- [5] The Open MPI Organization. (2018). *Open MPI: Open Source High Performance Computing*. [Online]. Available: <https://www.open-mpi.org/>
- [6] MPICH Organization. (2018). *MPICH*. [Online]. Available: <http://www.mpich.org/>
- [7] IBM Systems. (2018). *IBM Spectrum MPI*. [Online]. Available: <https://www.ibm.com/us-en/marketplace/spectrum-mpi>
- [8] Intel Developer Zone. (2018). *Intel MPI Library*. [Online]. Available: <https://software.intel.com/en-us/intel-mpi-library>
- [9] A. Droste, M. Kuhn, and T. Ludwig, "MPI-checker: Static analysis for MPI," in *Proc. 2nd Workshop LLVM Compiler Infrastruct. HPC LLVM*, 2015, pp. 1–10, doi: [10.1145/2833157.2833159](https://doi.org/10.1145/2833157.2833159).
- [10] *MUST: MPI Runtime Error Detection Tool*, RWTH Aachen Univ., Aachen, Germany, 2018.
- [11] T. Hilbrich, M. Schulz, B. R. de Supinski, and M. S. Müller, "MUST: A scalable approach to runtime error detection in MPI programs," in *Tools for High Performance Computing*. Berlin, Germany: Springer, 2010, pp. 53–66.
- [12] J. DeSouza, B. Kuhn, and B. R. de Supinski, "Automated, scalable debugging of MPI programs with Intel message checker," in *Proc. 2nd Int. Workshop Softw. Eng. High Perform. Comput. Syst. Appl. SE-HPCS*, 2005, pp. 901–908.
- [13] D. C. Arnold, D. H. Ahn, B. R. de Supinski, G. L. Lee, B. P. Miller, and M. Schulz, "Stack trace analysis for large scale debugging," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, Mar. 2007, pp. 1–10, doi: [10.1109/IPDPS.2007.370254](https://doi.org/10.1109/IPDPS.2007.370254).
- [14] R. Kowalewski and K. Furlinger, "Nasty-MPI: Debugging synchronization errors in MPI-3 one-sided applications," in *Proc. Eur. Conf. Parallel Process. Euro-Par*, 2016, pp. 51–62, doi: [10.1007/978-3-319-43659-3_4](https://doi.org/10.1007/978-3-319-43659-3_4).
- [15] 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, Jun. 2009, doi: [10.1007/s10766-009-0099-4](https://doi.org/10.1007/s10766-009-0099-4).
- [16] M. K. Ganai, "Dynamic livelock analysis of multi-threaded programs," in *Runtime Verification*. San Diego, CA, USA: IEEE, 2013, pp. 3–18.
- [17] A. Humphrey, C. Derrick, G. Gopalakrishnan, and B. Tibbitts, "GEM: Graphical explorer of MPI programs," in *Proc. 39th Int. Conf. Parallel Process. Workshops*, Sep. 2010, pp. 161–168. [Online]. Available: <http://ieeexplore.ieee.org/document/5599207/>
- [18] G. Luecke, H. Chen, J. Coyle, J. Hoekstra, M. Kraeva, and Y. Zou, "MPI-CHECK: A tool for checking fortran 90 MPI programs," *Concurrency Comput., Pract. Exper.*, vol. 15, no. 2, pp. 93–100, 2003.
- [19] D. Kranzlmüller, C. Schaubachlaeger, and J. Volkert, "A brief overview of the MAD debugging activities," in *Proc. 4th Int. Workshop Automated Debugging (AADEBUG)*, 2000, pp. 234–299.
- [20] A.-T. Do-Mai, T.-D. Diep, and N. Thoi, "Race condition and deadlock detection for large-scale applications," in *Proc. 15th Int. Symp. Parallel Distrib. Comput. (ISPDC)*, Jul. 2016, pp. 319–326, doi: [10.1109/ISPDC.2016.53](https://doi.org/10.1109/ISPDC.2016.53).
- [21] C. Clemenccon, J. Fritscher, and R. Ruhl, "Visualization, execution control and replay of massively parallel programs within annai's debugging tool," in *Proc. High-Perform. Comput. Symp. (HPCS)*, 1995, pp. 393–404.
- [22] M.-Y. Park, S. J. Shim, Y.-K. Jun, and H.-R. Park, "MPIRace-check: Detection of message races in MPI programs," in *Proc. Int. Conf. Grid Pervasive Comput. GPC*, 2007, pp. 322–333, doi: [10.1007/978-3-540-72360-8_28](https://doi.org/10.1007/978-3-540-72360-8_28).
- [23] V. Forejt, S. Joshi, D. Kroening, G. Narayanaswamy, and S. Sharma, "Precise predictive analysis for discovering communication deadlocks in MPI programs," *ACM Trans. Program. Lang. Syst.*, vol. 39, no. 4, pp. 1–27, Sep. 2017, doi: [10.1145/3095075](https://doi.org/10.1145/3095075).
- [24] G. Gopalakrishnan, R. M. Kirby, S. Vakkalanka, A. Vo, and Y. Yang, "ISP (*in-situ* partial order): A dynamic verifier for MPI programs," Ph.D. dissertation, School Comput. Univ. Utah, Salt Lake, Utah, 2009. [Online]. Available: <http://formalverification.cs.utah.edu/ISP-release/>
- [25] E. Saillard, P. Carribault, and D. Barthou, "PARCOACH: Combining static and dynamic validation of MPI collective communications," *Int. J. High Perform. Comput. Appl.*, vol. 28, no. 4, pp. 425–434, Nov. 2014, doi: [10.1177/1094342014552204](https://doi.org/10.1177/1094342014552204).
- [26] H. Ma, L. Wang, and K. Krishnamoorthy, "Detecting thread-safety violations in hybrid OpenMP/MPI programs," in *Proc. IEEE Int. Conf. Cluster Comput.*, Sep. 2015, pp. 460–463, doi: [10.1109/CLUSTER.2015.70](https://doi.org/10.1109/CLUSTER.2015.70).
- [27] G. Bronevetsky, I. Laguna, S. Bagchi, B. R. de Supinski, D. H. Ahn, and M. Schulz, "AutomaDeD: Automata-based debugging for dissimilar parallel tasks," in *Proc. IFIP Int. Conf. Dependable Syst. Netw. (DSN)*, Jun. 2010, pp. 231–240.
- [28] Allinea Software Ltd. (2018). *ALLINEA DDT*. ARM HPC Tools. [Online]. Available: <https://www.arm.com/products/development-tools/hpc-tools/cross-platform/forge/ddt>
- [29] RWS Inc. (2018). *TotalView for HPC*. [Online]. Available: <https://www.roguewave.com/products-services/totalview>
- [30] A. P. Claudio, J. D. Cunha, and M. B. Carmo, "Monitoring and debugging message passing applications with MPVisualizer," in *Proc. 8th Euromicro Workshop Parallel Distrib. Process.*, Jan. 2000, pp. 376–382, doi: [10.1109/EMPDP.2000.823433](https://doi.org/10.1109/EMPDP.2000.823433).
- [31] A. M. Alghamdi and F. E. Eassa, "Software testing techniques for parallel systems: A survey," *Int. J. Comput. Sci. Netw. Secur.*, vol. 19, no. 4, pp. 176–186, 2019.
- [32] NAS Parallel Benchmarks Team. (2018). *NAS Parallel Benchmarks Version 3.4*. [Online]. Available: <https://www.nas.nasa.gov/publications/npb.html>
- [33] Network-Based Computing Laboratory. (2019). *OSU Microbenchmarks*. The Ohio State University, Columbus, OH, USA. [Online]. Available: <http://mvapich.cse.ohio-state.edu/benchmarks/>
- [34] J. M. Bull, J. Enright, X. Guo, C. Maynard, and F. Reid, "Performance evaluation of mixed-mode OpenMP/MPI implementations," *Int. J. Parallel Program.*, vol. 38, nos. 5–6, pp. 396–417, Oct. 2010.

- [35] D. Grove and P. Coddington, "Precise MPI performance measurement using MPIBench," in *Proc. HPC Asia*, 2001, pp. 24–28.
- [36] G. R. Luecke, Y. Zou, J. Coyle, J. Hoekstra, and M. Kraeva, "Deadlock detection in MPI programs," *Concurr. Comput. Pract. Express*, vol. 14, no. 11, pp. 911–932, Aug. 2002, doi: [10.1002/cpe.701](https://doi.org/10.1002/cpe.701).



ABDULLAH S. ALMALAISE ALGHAMDI received the B.Sc. degree in computer science from the University of Southern Mississippi, USA, in 1990, the M.Sc. degree in management information systems from the University of Illinois at Springfield, IL, USA, in 1992, and the Ph.D. degree in computer science from George Washington University, USA, in 2003. He is currently a Full Professor with the Information Systems Department, Faculty of Computing and Information Technology, King Abdulaziz University, Saudi Arabia. His research interests include collaborative software, distributed systems, conflict measurements, workflow, information systems, and artificial intelligence.

received the B.Sc. degree in computer science from the University of Southern Mississippi, USA, in 1990, the M.Sc. degree in management information systems from the University of Illinois at Springfield, IL, USA, in 1992, and the Ph.D. degree in computer science from George Washington University, USA, in 2003. He is currently a Full Professor with the Information Systems Department, Faculty of Computing and Information Technology, King Abdulaziz University, Saudi Arabia. His research interests include collaborative software, distributed systems, conflict measurements, workflow, information systems, and artificial intelligence.



AHMED MOHAMMED ALGHAMDI received the B.Sc. degree in computer science and the first M.Sc. degree in business administration from King Abdulaziz University, Jeddah, Saudi Arabia, in 2005 and 2010, respectively, the second master's degree in internet computing and network security from Loughborough University, U.K., in 2013, and the Ph.D. degree in computer science from King Abdulaziz University. He is an Assistant Professor with the Department of Software Engineering, College of Computer Science and Engineering, University of Jeddah, Saudi Arabia. He has also over 11 years of working experience before attending the academic carrier. His research interests include high-performance computing, big data, distributed systems, programming models, software engineering, and software testing.

received the B.Sc. degree in computer science and the first M.Sc. degree in business administration from King Abdulaziz University, Jeddah, Saudi Arabia, in 2005 and 2010, respectively, the second master's degree in internet computing and network security from Loughborough University, U.K., in 2013, and the Ph.D. degree in computer science from King Abdulaziz University. He is an Assistant Professor with the Department of Software Engineering, College of Computer Science and Engineering, University of Jeddah, Saudi Arabia. He has also over 11 years of working experience before attending the academic carrier. His research interests include high-performance computing, big data, distributed systems, programming models, software engineering, and software testing.



FATHY ELBOURAEY EASSA received the B.Sc. degree in electronics and electrical communication engineering from Cairo University, Egypt, in 1978, and the M.Sc. and Ph.D. degrees in computers and systems engineering from Al-Azhar University, Cairo, Egypt, in 1984 and 1989, respectively, with joint supervision with the University of Colorado, USA, in 1989. He is currently a Full Professor with the Computer Science Department, Faculty of Computing and Information Technology, King Abdulaziz University, Saudi Arabia. His research interests include agent-based software engineering, cloud computing, software engineering, big data, distributed systems, and exascale system testing.

received the B.Sc. degree in electronics and electrical communication engineering from Cairo University, Egypt, in 1978, and the M.Sc. and Ph.D. degrees in computers and systems engineering from Al-Azhar University, Cairo, Egypt, in 1984 and 1989, respectively, with joint supervision with the University of Colorado, USA, in 1989. He is currently a Full Professor with the Computer Science Department, Faculty of Computing and Information Technology, King Abdulaziz University, Saudi Arabia. His research interests include agent-based software engineering, cloud computing, software engineering, big data, distributed systems, and exascale system testing.



MAHER ALI KHEMAKHEM received the B.Sc. degree in physics from the University of Tunis, Tunisia, in 1982, and the M.Sc. degree in digital electronics and computer science and the Ph.D. degree in digital electronics and computer science from the University of Paris 11, Orsay, France, in 1984 and 1987, respectively, and the Habilitation Accreditation (HDR) degree in computer science from the University of Sfax, Tunisia, in 2008. He is currently a Full Professor with the Computer Science Department, Faculty of Computing and Information technology, King Abdulaziz University, Saudi Arabia. His research interests include distributed systems, performance analysis, network security, and pattern recognition.

received the B.Sc. degree in physics from the University of Tunis, Tunisia, in 1982, and the M.Sc. degree in digital electronics and computer science and the Ph.D. degree in digital electronics and computer science from the University of Paris 11, Orsay, France, in 1984 and 1987, respectively, and the Habilitation Accreditation (HDR) degree in computer science from the University of Sfax, Tunisia, in 2008. He is currently a Full Professor with the Computer Science Department, Faculty of Computing and Information technology, King Abdulaziz University, Saudi Arabia. His research interests include distributed systems, performance analysis, network security, and pattern recognition.

• • •