Computers, Materials & Continua
DOI: 10.32604/cmc.2023.033928
Article

Tech Science Press
check for updates

A Parallel Hybrid Testing Technique for Tri-Programming Model-Based Software Systems

Huda Basloom[1,*], Mohamed Dahab[1], Abdullah Saad AL-Ghamdi[2], Fathy Eassa[1], Ahmed Mohammed Alghamdi[3] and Seif Haridi[4]

[1]Department of Computer Science, Faculty of Computing and Information Technology, King Abdulaziz University, Jeddah, 21589, Saudi Arabia
[2]Department of Information Systems, Faculty of Computing and Information Technology, King Abdulaziz University, Jeddah, 21589, Saudi Arabia
[3]Department of Software Engineering, College of Computer Science and Engineering, University of Jeddah, Jeddah, 21493, Saudi Arabia
[4]KTH Royal Institute of Technology, Stockholm, Sweden
*Corresponding Author: Huda Basloom. Email: hbasaloom@kau.edu.sa
Received: 01 July 2022; Accepted: 15 September 2022

Abstract: Recently, researchers have shown increasing interest in combining more than one programming model into systems running on high performance computing systems (HPCs) to achieve exascale by applying parallelism at multiple levels. Combining different programming paradigms, such as Message Passing Interface (MPI), Open Multiple Processing (OpenMP), and Open Accelerators (OpenACC), can increase computation speed and improve performance. During the integration of multiple models, the probability of runtime errors increases, making their detection difficult, especially in the absence of testing techniques that can detect these errors. Numerous studies have been conducted to identify these errors, but no technique exists for detecting errors in three-level programming models. Despite the increasing research that integrates the three programming models, MPI, OpenMP, and OpenACC, a testing technology to detect runtime errors, such as deadlocks and race conditions, which can arise from this integration has not been developed. Therefore, this paper begins with a definition and explanation of runtime errors that result from integrating the three programming models that compilers cannot detect. For the first time, this paper presents a classification of operational errors that can result from the integration of the three models. This paper also proposes a parallel hybrid testing technique for detecting runtime errors in systems built in the C++ programming language that uses the triple programming models MPI, OpenMP, and OpenACC. This hybrid technology combines static technology and dynamic technology, given that some errors can be detected using static techniques, whereas others can be detected using dynamic technology. The hybrid technique can detect more errors because it combines two distinct technologies. The proposed static technology detects a wide range of error types in less time, whereas a portion of the potential errors that may or may not occur depending on the operating environment are left to the dynamic technology, which completes the validation.

Keywords: Software testing; hybrid testing technique; OpenACC; OpenMP; MPI; tri-programming model; exascale computing

1 Introduction

The development of parallel systems is a significant area of study in the modern era, given that this approach is becoming increasingly key to exascale computing. Parallel applications of high-performance computing allow work to be divided among several nodes, central processing unit (CPU) cores, and graphics processing units (GPUs) to decrease the computation time needed to accomplish the program's tasks. The concern is that standard programming languages cannot handle parallelism effectively; hence alternative programming models have been employed to provide parallelism to systems. Programming models can be integrated into standard programming languages such as C, C++, Python, Java, and Fortran, making them capable of executing parallel tasks.

Currently, several programming models are available, each of which has different characteristics. Multiple models can be combined to obtain the different features of each model. Several programming models are commonly used in heterogeneous systems, such as: Open Accelerators (OpenACC) [1], Open Multi-Processing (OpenMP) [2], Compute Unified Device Architecture (NVIDIA CUDA) [3], and Open Computing Language (OpenCL) [4]. The utilization of multi-core processors and shared memory in some models, such as OpenMP, can allow programs to run faster because they can split the work into multiple threads and apply parallelism. Other models support parallelism in GPU accelerators, including OpenCL, OpenACC, and CUDA; these approaches use heterogeneous nodes to achieve high performance and exascale computing. As a result, the efficient use of computer resources and the effective use of inter-node and intra-node parallelism increase functional and energy efficiency, posing a major obstacle to the development of future exascale systems.

This paper focuses on detecting errors in systems that use tri-programming models to increase parallelism in high-performance computing software, including exascale systems. One possible combination is Message Passing Interface (MPI) + OpenMP + OpenACC, which has been implemented in several studies [5–7]. To the best of our knowledge, no testing tool or technique has been developed to discover errors that may arise because of this integration. This study aims to propose a parallel hybrid testing technique for detecting runtime errors in systems built in the C++ programming language that uses a triple programming model that combines MPI, OpenMP, and OpenACC. In addition, we present a classification of errors that appear as a result of the integration of these three models. This is the first time that this category has been provided. This classification is supported by numerous examples that have been fully tested and discussed.

The remainder of this paper is organized as follows: Section 2 provides a brief overview of the software models that were employed in this research. Section 3 discusses related work. Section 4 identifies some runtime errors in the MPI, OpenMP, and OpenACC programming models. Section 5 presents some errors that resulted from integrating the three programming models. Section 6 presents an error classification for the suggested tri-programming model. Section 7 addresses the proposed architecture. In Section 8, we present the discussion. Section 9 includes the conclusions and future works.

2 Background

There are various programming models that can be used with programming languages. This section provides a brief overview of the software models used in this paper: MPI, OpenMP, and OpenACC.

The Parallel Computing Institute created the Message Passing Interface (MPI) library specification for parallel computing architectures in May 1994. High performance computing (HPC) frequently uses the distributed computing library MPI, which can run on any distributed architecture. It distributes work among nodes and clusters across the network. As a result of its widespread use, it significantly contributes to the acceleration of computing. It has been implemented in several research in combination with OpenMP [8–10] and OpenACC [5,11,12].

OpenMP is a shared memory application programming interface. Developers are using it more frequently due to the obvious widespread use of multi-core devices and multi-threaded processors [2,5,8–10]. Using OpenMP, work can be distributed among CPU-core-based threads, and data can be shared or made private for each thread as needed by the program. The performance will be improved when OpenMP capabilities are properly included in a serial application. The program will be able to make use of parallel architectures with shared memory. OpenMP is a simple programming model to understand and use, because the overhead of a parallel program is handled by the compiler directives.

OpenACC, a high-level programming model based on directives like OpenMP, is supported by an initial set of three translator vendors, including Computer Assisted Program System (CAPS), Cray, and Portland Group Inc. (PGI) [13]. It works by including directives in the source code of Fortran, C, or C++, which makes it easy to use. OpenACC is characterized by portability across devices from multiple vendors. It is also more productive than other models because it is a high-level, directive-based programming model that efficiently utilizes the GPU [1,13,14].

Recent studies have also considered merging directive-based programming such as OpenMP and OpenACC, which have gained popularity because of their simplicity of implementation in comparison to low-level programming models such as CUDA and OpenCL. Both OpenACC and OpenMP are directive-based models for accelerators. OpenACC and OpenMP are not equivalent, and each of them has its own advantages and disadvantages. The references [5,11,15–20] provide additional details regarding the comparison of the two programming models. Several of these studies show that OpenACC is efficient at using accelerators. OpenMP, on the other hand, used in shared memory and multi-core CPU parallel programming. The following are the advantages of utilizing both OpenMP and OpenACC together:

- Portability is ensured by using two sets of directives in the same source files.
- Two different runtimes can communicate with one another within the same process.
- Incorporating libraries introduced in either model, such as the OpenACC math libraries.
- Transferring data from one programming model to another.

Many studies [1,5,8–12] have shown the possible benefits of integrating more than one programming model through memory exploitation and accelerator utilization. This may increase throughput and improve performance. In addition, the workload can be distributed across network nodes to boost throughput. Many combinations of programming models are possible, such as X1 + X2 (where X is a programming model) and Message Passing Interface (MPI) + X1 + X2. Some studies have employed integration in different areas of software acceleration, as seen in the following case studies: MPI + OpenMP [2,8–10]; MPI + OpenACC [1]; OpenMP + CUDA [21]; MPI + OpenACC + OpenMP [5–7]; and OpenACC + OpenMP [12,20].

Runtime errors may be actual or potential. When a program encounters an actual error, it occurs each time the program is executed, regardless of whether the programmer's awareness. Potential errors may be difficult to detect, however, because they are not present during every software execution. Depending on the value of the data being processed or the execution order of parallel threads, it may or may not occur, making it difficult for the programmer to identify. These types of runtime errors may lead to race conditions, data races, deadlocks, and other undesirable outcomes. Discovering runtime errors in parallel programs that integrate more than one programming model is more challenging than finding errors in programs that just use one programming model. One reason is that it is not possible to predict the behavior of threads during runtime. Combining two or more models in a single parallel application can result in runtime errors for unknown reasons, particularly if there is no testing technique to identify these runtime errors.

3 Related Work

This section provides an overview of various open source and commercial debugging and testing techniques. This section discusses OpenMP, OpenACC, and MPI models research, along with combination of these models. Because these models are part of the research, the techniques used to detect runtime errors in them, and the current state of the art techniques will be evaluated to determine the most effective strategies.

Recent studies have investigated errors and identified their causes based on the programming models MPI, OpenMP, CUDA, and OpenCL, as well as the integrated programming models MPI with OpenACC, MPI with OpenMP, and many other combinations. There is no testing tool or debugger that has obtained the OpenACC, OpenMP, and MPI combination. In the past few years, there has been a big increase in the number of applications that use dual-programming models and tri-programming models that include OpenACC, OpenMP, and MPI [5–7].

There are numerous classifications for testing techniques such as static, dynamic, and hybrid testing techniques. Static code analysis [22–26] is a method of debugging that examines source code prior to the compilation and execution stages of the program. This sort of test may be used to identify potential runtime errors and actual runtime errors, such as various forms of deadlock and race conditions, which might occur during the execution of the program.

In dynamic analysis [1,27–32] bugs are detected during runtime. This type can detect the actual errors that occur during execution. The test cases used are the key to the reliability of this technique. This creates a challenge because these errors might be caused by a variety of circumstances and complex conditions. In fact, certain instruments can only define a race after it has already occurred. Thus, with this approach, one cannot be certain that all errors are reported or hidden. The performance of this technique may also be poor due to its complexity. Moreover, some errors can be buried because it is impossible to monitor all possible program responses during execution. The hybrid testing technique combines more than one testing technique together, for example, the integration of static and dynamic testing techniques. When multiple techniques are used, a broader range of errors can be detected than if each approach were employed alone. This integration may increase the time required to debug errors, but the intended advantage of identifying the largest number of potential and actual runtime errors will boost their use. Meany research used hybrid testing techniques such as: [30–34]. As a result, the type and behavior of runtime errors decide the procedures that should be employed. For example, some errors are difficult to discover using static analysis techniques, whereas others cannot be detected using dynamic analysis processes.

Model checking [35–38] is another approach that is similar to the static technique in that it checks for errors statically by determining runtime states before runtime. This technique checks the parallel system in a finite state. It replicates the circumstances of race and deadlock and guarantees that there are no races or deadlock. It is necessary to change the program into a modeling language to use this method. However, due to the vast number of codes that may be tested, it is quite difficult to manually extract the model. In addition, the potential number of possible occurrences is just too huge to be considered in any detail. It is only suitable for testing small and critical applications, and it is not advised for testing larger projects.

According to the findings of this study, no technique for identifying runtime errors has been published in the tri-Programming model, particularly when using MPI, OpenACC, and OpenMP. A hybrid testing technique for identifying runtime errors in the tri-programming model implemented in C++ by MPI, OpenACC, and OpenMP is proposed because of this study. The proposed technique will be discussed in further detail in a subsequent section. There is no existing study that is similar to what this paper proposes, thus it is impossible to compare it with earlier research that offered a technique for the triple programming model of MPI, OpenMP, and OpenACC. The three models used for this study were picked with the goal of increasing the system's speed to attain the exascale. Specifically, the research recommends a hybrid testing technique that combines static testing techniques with dynamic testing techniques to discover as many errors as possible in C++ programs combing the three programming models. The hybrid programming models MPI, OpenACC, and OpenMP will be discussed in the following section.

4 Runtime Errors

Several different varieties of runtime errors can occur after a program has been compiled, and the compiler may not be able to detect them, which is particularly critical when designing parallel applications. To take advantage of parallelism in software, the programmer must identify and address runtime errors to avoid losing the desired benefits of parallelism as a result of the errors that occur. When hybrid models are used, the percentage of these mistakes increases, and the reasons for these errors vary. Some of the most common runtime errors in C++ programs that use the MPI, OpenMP, and OpenACC programming models will be discussed in this section.

4.1 Deadlock

A deadlock is a common problem, and many previous studies have addressed it, particularly in parallel programming [39,40]. A deadlock usually occurs when two or more threads are waiting for input from one another; a loop forms that prevents either thread from progressing. Some reasons for deadlock in the OpenMP, OpenACC, and MPI programming models will be discussed.

First, in OpenMP, the incorrect usage of asynchronous features leads to a deadlock. For example, setting locks in the wrong order in OpenMP can cause two or more threads to wait for each other, as seen in Listing 1. Two or more OpenMP threads are stuck in this example, and both threads wait for each other to lock a variable that has been obtained by the other thread. The example has two sections, each of which is executed by a distinct thread. If thread one runs the first section, it will lock 'locka' at line 6. Assuming that thread two locks 'lockb' in line 15. In line 7, thread one needs to lock 'lockb' to complete the work, but it will wait for the second thread to release the lock from 'lockb'. At the same time, thread two is waiting for thread one to unset the lock 'locka' while thread one is waiting for thread two. This waiting will continue forever, resulting in a deadlock. Many of the tools presented in the literature review were designed to detect such types of errors.

Listing 1: Incorrect use of the lock. The same variable 'locka' is locked in different sections. If each thread runs a section, then thread 1 locks the variable 'locka', the other thread locks the variable 'lockb' and each of them waits for each other forever to release the hold lock.

```
#pragma omp parallel
  #pragma omp sections nowait
    {
    #pragma omp section
      {
      omp_set_lock(&locka);
      omp_set_lock(&lockb);
      do_work();
      omp_unset_lock(&lockb);
      omp_unset_lock(&locka);
        }
      #pragma omp section
        {
        omp_set_lock(&lockb);
        omp_set_lock(&locka);
        do_work();
        omp_unset_lock(&locka);
        omp_unset_lock(&lockb);
        }   }       }
```

A barrier directive is another potential cause of deadlock in OpenMP if it is not used correctly. This directive requires any threads that arrive at the barrier to wait until all other threads have also arrived. If a barrier exists, either all threads or none of them must encounter the barrier; otherwise, a deadlock will occur. It should not be contained within an 'if', 'else', 'section', or 'for loop'. Such errors cannot be detected by the compiler, so a debugger or testing tool must be used to detect them. In Listing 2, an example of the wrong use of the barrier is given. According to the OpenMP standard, a barrier cannot be utilized within the loop.

Listing 2: An example of a deadlock in OpenMP caused by the use of the 'barrier' within a 'for loop'. This is not allowed according to the OpenMP specification.

```
#pragma omp parallel
{ int i;
    #pragma omp parallel for
      for(i = 0; i < N;i++)
            {
                        //–do work—
                        #pragma omp barrier
                        //–do work—

              }}
```

Despite the many advantages provided by the MPI programming model, serious issues such as deadlocks, can occur because of programming errors that are not indicated by the compiler. The send and receive functions in MPI can be performed in either synchronous or asynchronous mode. A synchronous operation causes a process to be suspended until the operation is completed. When performing an asynchronous operation, the operation is not blocked and is initiated only once. This means that the 'send' will be blocked until it reaches its destination. In contrast, a blocking receive action, returns only once the message is received. Higher parallelism can be achieved using asynchronous message passing. Because a process does not block, it can perform certain computations while the message is being transmitted.

Understanding the right manner of transmitting and receiving data is necessary to fully understand the reasons for the existence of deadlock or race in MPI. This programming model is based on the messages that are sent and received. 'MPI_Send' is a function that transmits a buffer from one sender to one receiver. 'MPI_Recv' is a function that receives data sent by the sender (see Listing 3). The type of data sent should match the type of data received to avoid a deadlock from occurring. For example, if data type 'MPI_FLOAT' is received and the destination selects a different data type, such as 'MPI_UNSIGNED_CHAR', a deadlock will occur. A deadlock will also occur if the size of the data being sent is bigger than the size of the data being received.

Listing 3: MPI send and receive parameters. 'MPI_Send' takes six parameters in order: a pointer to the buffer holding the data to be sent, a pointer to the data to be sent, the number of items stored in the buffer, the data type for the elements, the destination rank, a message tag used to differentiate between different message kinds, and the communication device. In 'MPI_Recv', two more arguments that convey status information about the received message and the rank of the transmission process is added.

```
MPI_Send(void∗ data, int count, MPI_Datatype datatype, int destination, int tag, MPI_Comm
communicator)
MPI_Recv(void∗ data, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm
communicator, MPI_Status∗ status)
```

In blocking point-to-point MPI routines, a deadlock occurs if message passing cannot be completed. 'MPI_Send' is a blocking operation. It may not be complete until an 'MPI_Recv' has received the data. For example, if Process 0 carries out 'MPI_Send' without a corresponding receipt 'MPI_Recv' in Process 1, a potential deadlock will be encountered. Another scenario in which a deadlock occurs is illustrated in Listing 4. In this example, Process 0 attempts to send a message to Process 1, but Process 1 does not have an 'MPI_Recv' that matches the message sent by Process 0. The solution is straightforward: simply replace the command 'MPI_Send' with the 'MPI_Recv'. More research has developed tools or debuggers for detecting such errors. Another cause of deadlock in MPI is the presence of 'MPI_Recv' data on one side without the presence of 'MPI_Send' on the other side. For example, the existence of 'MPI_Recv' corresponds to 'MPI_Recv' rather than to the presence of 'MPI_Send'. This is called a "receive-receive" deadlock. In addition, when a process sends a message to itself, a deadlock situation may arise.

Listing 4: An example of a deadlock in the MPI. In line 2, Process 0 sends data; yet, in line 5, Process 1 has ‘send’ instead of ‘receive’. Consequently, Process 0 would be blocked due to the absence of a corresponding ‘receive’ at Process 1.

```
1   if (process == 0) {
2       MPI_Send( . . . , 1, tag, comm);
3       MPI_Recv( . . . , 1, tag, comm, &status);
4     }else if (process == 1) {
5       MPI_Send( . . . , 0, tag, comm);
6       MPI_Recv( . . . , 0, tag, comm, &status);} }
```

A programmer may make certain runtime errors, but these errors do not arise every time the software is run, but they do appear from time to time. Although MPI supports the receive feature (from: MPI_ANY_SOURCE), which allows the receiving process to match a transmission from any process when a sender is not specified, this might result in a potential deadlock, as will be illustrated in the following example. In the programming example given in Listing 5, Process 1 takes advantage of (MPI_ANY_SOURCE in MPI). After receiving the message from Process 0, this method will be able to receive the message from Process 2 at the second recipient of Process 1, guaranteeing that no conflicts arise. If Process 1 receives the message from Process 2 before the first reception is completed, the second reception will not be completed because Process 2 does not send another message before invoking the barrier to prevent communication between processes. Using MPI_ANY_SOURCE in the incorrect manner, as seen in this example, may result in a deadlock. According to this example, the potential runtime errors do not arise every time the program is executed; rather, they depend on the sequence in which messages are sent between the processes.

Listing 5: The Recv(from:ANY) method in Process 1 uses (MPI_ANY_SOURCE), which allows it to receive data from any process. After receiving a message from Process 0, Process 2 can send its message too, and all three processes can continue to operate. However, if Process 1 receives the message from Process 2 first, it will not be able to complete its second reception, given that Process 2 will not transmit until the barrier call is received. Process 1 will never be able to send a message to Process 0, and both processes will always be blocked.

```
if (process == 0){
        MPI_Send(sendbuf, count, MPI_INT, 1, tag, comm);
        Barrier();}
else if (process == 1){
        MPI_Recv(recvbuf, count, MPI_INT, MPI_ANY_SOURCE, tag, comm);
        MPI_Recv(recvbuf, count, MPI_INT, 2, tag, comm);
        Barrier();}
else if (process == 2){
        MPI_Send(sendbuf, count, MPI_INT, 1, tag, comm);
        Barrier();}
```

These concerns can be address through various solutions, whether they are implemented using a blocked or an unblocked connection. However, the problem does not exist here. Rather, when systems of enormous importance are built with such errors, it will be quite expensive. We discovered that a tool that can detect such mistakes and report them to the programmer.

Some previous studies have developed testing tools to detect deadlocks in MPI; these tools include MPI-SV [22], MPI-SPIN [27], MUST [39], MPI-CHECK [40], Marmot [41], and MOPPER [42]. Some tools can also detect a race condition in MPI: [43] and MPIRace-Check [44]. These tools can look for problems, such as having a receive without a send and other errors.

As a result of the nature of the OpenACC implicit barrier, as seen in Listing 6, a live lock in the GPU can result in a CPU deadlock. Every computation region has an implicit barrier that exists at the end of each calculation region. While the GPU is always occupied as a result of a live lock, the CPU's execution will be halted until all threads on the GPU have completed their parallel processing region.

Listing 6: Line 7 has a live lock, given that the value of 'i' will never change and it is true. The accelerator's parallel region has an implicit barrier. Because of the implicit barrier, the compute region will never finish, and the result will not be returned to the CPU. As a result, a deadlock occurs.

```
1   #pragma acc data copy(B[0:N], A[0:N])
2   {
3   #pragma acc parallel loop data copy(B[0:N])
4       for (int i = 0; i < N; i++)
5       {
6           B[i] = i;
7           while (i > 5)
8               B[i] = 5;
9       }
10   #pragma acc parallel loop copy(A[0:N])
11       for (int i = 0; i < N; i++)
12           A[i] = i;
13   }
```

An asynchronous clause can be used in OpenACC to specify which regions of the accelerator or parallel kernel can be run asynchronously and which areas cannot. The wait directive is used to keep the local thread running until it reaches this directive. Until all activities have been performed and the relevant asynchronous queue (s) have been completed, the traffic will not begin to be processed for transmission. However, as seen in the example in Listing 7, this may result in a deadlock. As asynchronous routing will enable the host to complete its task by passing the implicit barrier, it will come to a halt when the wait directive is applied, resulting in a deadlock in the program. The error classification in programming model OpenACC can be found in reference [1].

Listing 7: Line 7 has a live lock, but the inclusion of 'async' clauses allows the host to do the work until the use of the 'wait' directive permits the host to wait for asynchronous device tasks to be completed before continuing.

```
1   #pragma acc data copy(A[0:N],B[0:N])
2   {
3   #pragma acc parallel loop data copy(B[0:N]) async
4       for (int i = 0; i < N; i++)
5       {
6           B[i] = i;
7           while (i > 5)
```

(Continued)

Listing 7: Continued

```
                B[i] = 5;
        }
#pragma acc parallel loop copy(A[0:N])
        for (int i = 0; i < N; i++)
                A[i] = i;
}
#pragma acc wait
```

4.2 *Race Condition*

A race condition may develop as a result of many threads running operations in parallel. such as when accessing a shared resource, such as a variable in memory. The outcome can differ depending on the sequence in which the threads are performed, as demonstrated by [43]. Access to common variables should be synchronized to prevent race-related errors from occurring. Various techniques can be used to detect race conditions in each programming model on its own. This includes [43], and MPIRace-Check [44], which check for races in MPI, and OpenMP race checks, which include [45], ROMP OpenMP [46,47], and ACC_TEST [1], for checking the race condition in OpenACC. This subsection of the research will discuss some of the reasons for the race condition in each of the following programming models: OpenMP, MPI, and OpenACC.

Shared data read and write can cause race conditions in parallel programming when using the OpenMP programming model. The following situations can result in a race condition: while running read-after-write, write-after-write, or write-after-read on shared data among multiple threads [45,46]. In Listing 8, multiple threads change the shared variable 'Max_val' at the same time. This results in a race condition.

Listing 8: This example demonstrates code executed by many threads while a shared variable is changed by multiple threads. The ultimate value of 'Max_val' is determined by the thread execution order. This is known as a race condition.

```
#pragma omp parallel
{
   #pragma omp for nowait shared(Max_val)
   for(i = 0; i < N; i++){
      if(Max_val < A[i])
         Max_val = A[i]; }
}
```

In OpenMP, the inner 'for loop' variable in a nested 'for loop' is shared by default with the outer 'for loop' variable. In code example Listing 9, only the variable 'i' is private, but the variable in the inner loop 'j' is shared, and this is where the race condition will occur. This example of nested directives is not consistent with the specification. The solution to such issues is straightforward, but the programmer must be alerted over the progress of the program's execution for potential errors to prevent them and use alternate options. One alternative in this case would be to simply add 'private (j)' at the end of the first line. This problem may be resolved by adding the following: '#pragma omp parallel for' before the inner loop, which causes the inner loop to bind to a distinct parallel area.

Listing 9: The interference between the internal and external ‘for’ directives belonging to the same parallel directive causes the race condition.

```
#pragma omp parallel for
for (i = 0; i < N; i++)
   for (j = 0; j < N; j++)
      A[i][j] = B[i][j] + C[i][j]
```

OpenMP overcomes race conditions by using synchronization to access shared variables. Synchronization restricts demand for shared data such as ‘critical’, ‘atomic’, ‘barrier’, and ‘ordered’. Each of these directives has unique qualities that must be utilized carefully to avoid losing parallel properties. The ‘barrier’ directive is used to synchronize all threads by requiring them to wait at the barrier until all threads have completed their tasks. To avoid deadlock, the ‘barrier’ directive should be used carefully, as demonstrated in the previous subsection. As an alternative, the ‘ordered’ directive may be used to solve the problem, but the parallel performance will be reduced as if the execution were carried out in a sequential fashion. The use of a ‘critical’ directive, on the other hand, might eliminate the benefits of OpenMP and have a negative impact on performance. This directive allows only one thread to be active at a time, like the ‘ordered’ directive, preventing the simultaneous execution of codes across multiple threads from occurring. The ‘atomic’ directive outperforms the ‘critical’ directive while reducing the possibility of a race condition in the OpenMP programming model. It applies only to read and write operations in memory.

Data dependency can lead to a race condition in OpenMP, as seen in Listing 10. Because of the data dependence in the loop —where the value of A[i] depends on A[i + 1] — a race condition may occur if two threads attempt to read or write the same element of array A.

Listing 10: A race condition occur because of data dependencies in the program. ‘i’ is the loop counter, and “A” is the array. The value of the array is updated in each iteration, and its value depends on ‘i’ and ‘i + 1’ in line 3.

```
1   #pragma omp parallel for
2         for (i = 1; i < N; i++)
3               A[i] += A[i + 1];
```

When the ‘nowait’ directive is used incorrectly, another race condition is created (see Listing 11). The ‘nowait’ directive notifies the threads not to stop at the end of the ‘for loop’. The implicit barrier has no impact because of the ‘wait’ directive. For this reason, some threads may skip over the previous parallel region and move on to the next one. These threads can change array B before array A on which array B depends.

Listing 11: The usage of the ‘nowait’ directive causes a race condition in this code. One of the threads to may reach B before A has finished updating.

```
#pragma omp parallel
{
#pragma omp for nowait
      for (i = 1; i < N; i++)
            A[i] = A[i]/2.0;
```

(Continued)

Listing 11: Continued

```
    #pragma omp for nowait
        for (i = 0; i < M; i++)
            B[i] = sqrt(A[i]);
    }
```

In MPI, a race condition may occur when the buffer is used in multiple non-blocking communications. Understanding non-blocking communication is necessary to understand the race conditions in the MPI programming model. Sending data in a non-blocking way is accomplished with the 'MPI_Isend' command. to the nonblocking function 'MPI_Irecv' should be used to receive data. Listing 12 shows their definitions. When the two processes are ready to synchronize, it does nothing but set up the send. The buffer can only be used by one communication at a time, but it can be overwritten in this case.

Listing 12: MPI_ISend like MPI_Send but with extra implementation: 'buffer' which contains the data in the buffer to be sent, and 'request' which checks if the operation has finished and the data have really been sent.

```
int MPI_Isend(void *buffer, int count, MPI_Datatype datatype, int dest, int tag,
MPI_Communicator comm, MPI_Request *request);

int MPI_Irecv(void *buffer, int count, MPI_Datatype datatype, int source, int tag,
MPI_Communicator comm, MPI_Request *request);
```

Whenever the same buffer is being read from and written to at the same time by different non-blocking messages, a race condition is created. For example, in Listing 13, the buffer is used in multiple non-blocking communications. The two processes (Process 0 and Process 1) utilize the same buffer, but they use different values in the buffer. The buffer value may change while the thread is sleeping. As a result, race conditions may occur.

Listing 13: Race conditions may be encountered in this code, given that both processes (Process 0 and Process 1) share the same buffer with different values. While Process 0 is idle, given the sleep function call, the value of the buffer may change.

```
// Process 0 transmits the message, and Process 1 receives it.
if (process == 0) {
  // Adding data to the buffer
  for (int i = 0; i < buffer_size; ++i)
    buffer[i] = i * i;

  // 10 s have elapsed because the data was sent.
  MPI_Isend(buffer, buffer_size, MPI_INT, 1, 0, MPI_COMM_WORLD, &request);
  sleep(10);
 }
 else if (process == 1) {
  // The buffer has been reset
   for (int i = 0; i < buffer_size; ++i)
```

(Continued)

Listing 13: Continued

```
            buffer[i] = 0;
        // Receiving and sleeping for a total of ten seconds
        MPI_Irecv(buffer, buffer_size, MPI_INT, 0, 0, MPI_COMM_WORLD, &request);
        sleep(10);
    }
```

A race situation may occur when many threads are executed without taking the sequence into consideration. This is particularly tricky if the sequence is critical, as in OpenACC. However, race circumstances may arise because of the synchronization between the host and the device. The code in Listing 14 illustrates this by showing how a data race may occur as a result of an update command, that occurs while data are being sent between the host and the device. A 'kernel' directive, as opposed to a 'parallel loop', can be used in line 6 to eliminate this problem entirely. Using a kernel will ensure that instructions are executed in a sequential manner and will prevent a data race from occurring.

Listing 14: A race condition occurred during updating data between the host (CPU) and device (GPU).

```
1     #pragma acc data copyout(A[0:N]) copyin(B[0:N])
2     {
3        #pragma acc parallel loop
4              for (int i = 0; i < N; i++)
5                       A[i] = i;
6        #pragma acc parallel loop
               for (int i = 0; i < N; i++)
8                      {
9                      #pragma acc update
10                     B[i] = A[i] * 2;
11                     }
12    }
```

Data dependency, as in OpenMP, is another reason for the race condition in OpenACC. For example, as seen in Listing 15, data dependencies may result in race conditions. Because the value of A[i] depends on A[i + 1] a race condition will arise even if a kernel directive is used instead of a parallel loop to accomplish the task. The 'reduction' clause can be used to prevent a race condition from occurring before updating the value of array A.

Listing 15: A race condition develops in OpenACC because of data dependency. 'i' refers to the loop counter, whereas 'A' refers to the array. The value of the array is updated with each iteration, and its value is determined by the values of the variables 'i' and 'i + 1' in line 3.

```
1      #pragma acc parallel loop
2               for (i = 1; i < N; i++)
3                       A[i] + = A[i + 1];
```

4.3 Data Race

A data race occurs when two or more execution threads in a multi-threaded application attempt to access the same shared variable at the same time and at least one writes access to the shared variable [46,47]. A data race results in erroneous execution of the program and faults in the expected outputs. This is because the same data are shared by more than one thread at the same memory address, resulting in memory corruption. Data races are quite likely to occur when, for example, a loop invokes 'update data', and when those data are simultaneously read in parallel with the loop.

Data races can be avoided by integrating synchronization mechanisms into instructions or operations that occur in threads, depending on the programming model that is being used. Another method is to use a global shared flag, which allows just one writer thread to be active at the same time as many readers. The writer thread is responsible for setting the flag that permits the other threads to access the data. When the writer thread is done, it is available for reading by all other threads.

An example of a data race in the OpenMP programming model is demonstrated in Listing 16. Array A is shared across the parallel region, which is updated on line 5 and read on line 8. Furthermore, because of the usage of the 'nowait' directive, no implicit barrier occurs at the end of line 5. Because an element of array A may be read before or after an update, there will be a race will occur between reading and updating this element. Simply removing the 'nowait' directive will solve this problem.

Listing 16: A data race in OpenMP because of updating and reading the shared array A.

```
1   #pragma omp parallel shared (A[0:N],B[0:N],C[0:N])
2   {
3     #pragma omp for nowait
4     for(i = 0; i < N; i++)
5       A[i] = B[i] + A[i];
6     #pragma omp parallel for nowait
7     for(i = 0; i < N; i++)
8       C[i] = A[i] + B[i];
9   }
```

One further example can be found in Listing 17 of the OpenACC Programming Model, in which two loops are operating in the same parallel region and are using the same shared arrays A and B. As demonstrated in line 1, the arrays are not private to each loop in this example; rather they are shared by both loops. Given that arrays A and B can be accessed by many different threads, a data race can occur if the loop goes through its steps in an unexpected way.

Listing 17: A data race may occur if a shared array exists or if the array is updated in one loop and read in another and is shared between them.

```
1   #pragma acc data copy (A[0:N],B[0:N])
2   { #pragma acc loop gang
3    {
4     #pragma acc loop worker
5      for (int i = 0; i < N; i++)
6       A[i] = B[i] + A[i];
7     #pragma acc loop seq
8      for (int i = 0; i < N; i++)
```

(Continued)

Listing 17: Continued

```
        B[i] = A[i]/2;
    } }
```

This paper highlights some of these mistakes to demonstrate the significance and importance of having tools to detect these errors, particularly when merging more than one software model.

5 Hybrid Programming Models

Modern parallel computing systems include multi-core CPUs, accelerators such as GPUs, and Xeon processors, which may increase performance if they are utilized efficiently. C, C++, Java, and Fortran may all benefit from the addition of programming models (such as OpenMP, OpenCL, OpenACC, and NVIDIA CUDA) because they add parallelism and allow greater speed and efficiency for systems. However, there is no debugger or testing tool that can detect runtime errors in C++ system builds made by MPI, OpenACC, and OpenMP. Debuggers and testing tools are crucial when more than one model is merged. In addition to the errors that happen in each model on its own, there is a chance that more errors will happen after integration. Single-level programming models, dual-level programming models, and tri-level programming models are the three types of hybrid programming models that may be found in applications. There are more than three models that may be combined, but they will not be covered in this paper. The purpose of this section is to highlight errors that occur during execution but are not detected by the compiler because they are hidden.

Integrating the three programming models (MPI, OpenMP, and OpenACC) increases the speed by utilizing all available resources and splitting the work in an efficient manner. Utilizing MPI permits the distribution of work across multiple nodes on a network. MPI will keep track of how much memory is available at each stage. Both the use of inter-core memory in the CPU, which distributes tasks between multiple processor cores to allow parallel processing across multiple nodes, and the creation of multiple threads to execute tasks at high speed through the OpenMP programming model. Investigating GPU accelerators by OpenACC.

The MPI programming model is well-known and essential because it distributes work among several devices in a distributed computing environment. OpenACC has been used in five of the thirteen programs that have been created to increase performance on the world's top supercomputers [1,33,34,48]. Furthermore, OpenACC is used by three of the top five high-performance computing applications. It is anticipated that the use of OpenACC will rise. As a result, it is even more critical to have tools to rectify these programs to assure the accuracy of the results that will be retrieved by these applications. OpenMP has been widely used for parallel thread-based programming on shared memory systems, such as multi-core CPUs. With the introduction of running accelerators in OpenMP version 4, the protocol has become more efficient. OpenACC is typically preferred for GPU systems, whereas OpenMP is typically preferred for CPU systems.

The hierarchical structure of the hybrid tri-programming model is displayed in Fig. 1. MPI is responsible for distributing work throughout the network among nodes. OpenMP is utilized for parallel programming on multiple CPU cores. The OpenACC work carried out by the accelerator.

However, despite the potential gains from this integration, there is currently no debugging or testing technique available to detect such runtime errors, particularly those that arise because of the integration that has been implemented. Here are some examples of runtime errors that were found when code written using these three different programming models was put into action.

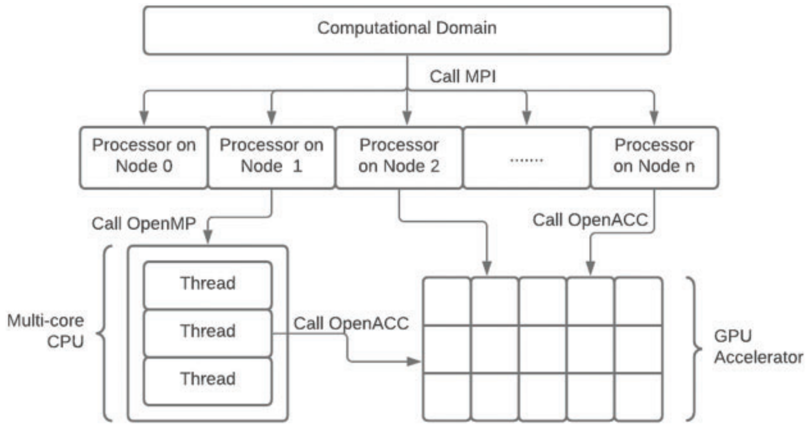

Figure 1: A hybrid programming hierarchy consisting of MPI, OpenMP, and OpenACC. MPI distributes the work over the network across multiple nodes. OpenMP makes advantage of the cores of the CPU and can use the OpenACC code inside OpenMP sections. The OpenACC code can also be called straight from the nodes, allowing it to operate on the GPU

According to Listing 18, there is no error in any of the programming models used in this example, yet the inclusion of the 'nowait' clause in OpenMP allowed the implicit barrier to be negated, as seen in the following example. At the same time, we discover that the data from array A is used in the computation of the array's value C in OpenACC in line 14. In this case, the race condition will provide a result because the array data is incomplete at the time, and we require its value to be included in the array. To address this problem, either the 'nowait' clause should be removed or a 'barrier' should be added before the start of the OpenACC code.

Listing 18: This code contains a data race because the value of array C depends on the calculation of array A. An update of array A is performed in the OpenMP programming model, which uses a 'nowait' clause, causing some threads to complete their work while the update is being executed. The programming model OpenACC starts by updating array C without finishing the array A update.

```
1    if (process == 0) {
2       MPI_Send (sendbuf, count, MPI_INT, 1, tag, comm);
3       MPI_Recv (recvbuf, count, MPI_INT, 1, tag, comm, &status);
4    } else if (process == 1) {
5       MPI_Recv (recvbuf, count, MPI_INT, 0, tag, comm, &status);
6       MPI_Send (sendbuf, count, MPI_INT, 0, tag, comm);
7    }
8    #pragma omp parallel for nowait
9    for (int i = 0; i < x;i++)
10     A[i] = B[i]*2;
11
12     #pragma acc kernels
13     for (int i = 0; i < x;i++)
14       C[i] = A[i]/0.2;
```

A programming model that encounters an error while running has an impact on the entire system because of the error. In the case of a deadlock caused by one of these programming models, the entire system is affected. For example, a deadlock in Listing 19 was generated by MPI because a message was sent from Process 0 and no corresponding receive was received in Process 1. This affected the entire system.

Listing 19: A hybrid programming model with a deadlock caused by the MPI send-send deadlock in blocking communication, resulting in the communication being blocked in line 3. The matching receive must be used in line 6 rather than the send.

```
1    if (process == 0) {
2       MPI_Send (sendbuf, count, MPI_INT, 1, tag, comm);
3       MPI_Recv (recvbuf, count, MPI_INT, 1, tag, comm, &status);
4    } else if (process == 1) {
5       MPI_Send (sendbuf, count, MPI_INT, 0, tag, comm);
6       MPI_Recv (recvbuf, count, MPI_INT, 0, tag, comm, &status);
7    }
8    #pragma omp parallel for
9    for (int i = 0; i < x;i++)
10      A[i] = B[i]*2;
11    #pragma acc kernels
12    for (int i = 0; i < x;i++)
13      C[i] = A[i]/0.2;
```

Therefore, the system becomes entirely deadlocked as a result of the OpenMP barrier directive in the case provided in the Listing 20. when a thread encounters a barrier, the thread must wait until all other threads have also encountered the barrier. By default, two separate threads are executed using the two OpenMP sections (lines 14 and 16). Each of them includes a barrier in the call to the ‘print_result’ function, and each thread will wait for the other threads to finish, which will never happen. Due to that, the system enters a deadlock.

Listing 20: The barrier in the print-result function at line 28 causes the entire system to deadlock. It is possible that several threads will execute this function, but they should all hit the same barrier, which will never occur.

```
1    void main()
2    {
3    //code . . . . . .
4    if (process == 0) {
5       MPI_Send (send_buf, count, MPI_CHAR, 1, tag, communication);
6       MPI_Recv (recvbuf, count, MPI_CHAR, 1, tag, communication, &status);
7    } else if (process == 1) {
8       MPI_Recv (recv_buf, count, MPI_CHAR, 0, tag, communication, &status);
9       MPI_Send (sen_dbuf, count, MPI_CHAR, 0, tag, communication);
10    }
11    #pragma omp parallel sections
12    {
13    #pragma omp sectoin
14      print_result(A, 1);
```

(Continued)

Listing 20: Continued

```
#pragma omp sectoin
  print_result(A, 2);
  }
  #pragma acc kernels
  for (int i = 0; i < x;i++)
    C[i] = A[i]/0.2;
  }//end main
  void print_result(float A[n], int section_no)
  {#pragma omp critical
     { for(i = 0; i < N; i++)
          A[i] = A[i]*.02;}
       #pragma omp barrier
     }
}//end print result
```

In Listing 21, it was observed that, depending on the behavior of the program during runtime, livelock in OpenMP generates deadlock and, in certain experiments, a data race. Consider that the value of the variable ‘x’ is less than 5. The system will enter a state of livelock, which may produce incorrect results in array C, especially if there are threads that have begun processing the section of the accelerator in line 14 before updating ‘x’ is complete. In the worst-case scenario, the system will be in a livelock or deadlock.

Listing 21: Livelock at line 5 result in a deadlock in the whole system. The value of array A is not modified within the while loop.

```
1    if (process == 0) {
2         MPI_Send (send_buf, count, MPI_CHAR, 1, tag, communication);
3         MPI_Recv (recvbuf, count, MPI_CHAR, 1, tag, communication, &status);
4      } else if (process == 1) {
5         MPI_Recv (recv_buf, count, MPI_CHAR, 0, tag, communication, &status);
6         MPI_Send (sen_dbuf, count, MPI_CHAR, 0, tag, communication);
7      }
8      #pragma omp parallel for nowait
9      while ( x < 5)
10        {A[i] = x*0.2;
11        if (i < N)
12        i++;}
13       #pragma acc kernels
14       for (int i = 0; i < N;i++)
15         C[i] = A[i]/0.2;
```

Two programming models in Listing 22 contain two different errors. The first error in the MPI programming model is that sending a message without corresponding receive results in a deadlock in the system. The second issue is caused by the inclusion of ‘nowait’ in OpenMP, which causes a race condition to occur in OpenACC, causing the data to be corrupted. The combination of these two errors resulted in a system deadlock.

Listing 22: Two errors deadlock and race condition result in a deadlock in the whole system.

```
if (process == 0) {
    MPI_Send (sendbuf, count, MPI_INT, 1, tag, comm);
    MPI_Recv (recvbuf, count, MPI_INT, 1, tag, comm, &status);
} else if (process == 1) {
    MPI_Send (sendbuf, count, MPI_INT, 0, tag, comm);
    MPI_Recv (recvbuf, count, MPI_INT, 0, tag, comm, &status);
}
#pragma omp parallel for nowait
for (int i = 0; i < x;i++)
    A[i] = B[i]*2;
 #pragma acc kernels
 for (int i = 0; i < x;i++)
    C[i] = A[i]/0.2;
```

There are three faults in the last example (see Listing 23), each of which is caused by one of the three programming models used in this research. The first error, caused by the MPI programming model, where there is a sending message without corresponding receiving, results in a deadlock in the system. The second issue is caused by the inclusion of 'nowait' in OpenMP, which causes a race condition to occur in OpenACC, causing the data to be corrupted. The last error occurred because of a data race in OpenACC. As a result of these three runtime errors, the system became stuck in a deadlock.

Listing 23: Three errors in the three programming models result in a deadlock in the whole system. The first error is deadlock result from programming error in MPI, The second error result from miss use of 'nowait' directive in OpenMP. The last error is a data race result from programming error in OpenACC.

```
if (process == 0) {
    MPI_Send (sendbuf, count, MPI_INT, 1, tag, comm);
    MPI_Recv (recvbuf, count, MPI_INT, 1, tag, comm, &status);
} else if (process == 1) {
    MPI_Send (sendbuf, count, MPI_INT, 0, tag, comm);
    MPI_Recv (recvbuf, count, MPI_INT, 0, tag, comm, &status);
}
#pragma omp parallel for nowait
for (int i = 0; i < x;i++)
    A[i] = B[i]*2;
    #pragma acc parallel for
 for (int i = 0; i < x;i++)
    C[i] = C[i + 1] + A[i]/0.2;
```

6 Error Classification for Software Using Hybrid Programming Models

Based on the findings of the research conducted in Section 3, as well as the errors that were identified throughout the experiments in Sections 4 and 5, we have classified these errors into three categories: one-level programming errors, two-level programming errors, and three-level programming errors. The first level occurs when there is an error in one of the programming models. The second level is reached when two programming models contain errors. When errors from each of the three programming models occur at the same time, this is the third level.

Based on our research, we have discovered that it is not always necessary for a program to be right when there is no error in any programming model used. For example, in Listing 18 a deadlock occurred in the system because the threads were permitted to complete the task while the array was being initialized. There was an error because the initialization of this array had not been finished before any additional actions on it were performed. To solve this problem, a 'barrier' should be inserted before updating the array.

Table 1 demonstrates that runtime errors occur when one of the programming models is responsible for a runtime error. According to our findings, the presence of a deadlock in one of the programming models causes the system to stop working. If there is a race condition or a data race, it may only cause a race condition or it may also cause a deadlock, depending on the production environment and the number of threads that are running. A deadlock may occur when there is a data race and many threads have already passed the initialization step. In addition, the presence of a livelock will disrupt the work, and we will become trapped in the same process, resulting in a deadlock.

Table 1: The table shows the errors that result in the first level, where errors are found in one of the three programming models. The models used are MPI, OpenMP, and OpenACC

OpenACC	MPI	OpenMP	Tri-Programming Model
No Error	No Error	No Error	Depending on the environment and the operating sequence sometimes it may not produce an error and other times it may cause a deadlock or a race condition.
OpenACC Race Condition	No Error	No Error	Depending on the environment and the running sequence, a deadlock or a race condition may occur.
OpenACC Deadlock	No Error	No Error	Deadlock.
OpenACC Potential Deadlock	No Error	No Error	Depending on the environment and the operating order a deadlock may or may not occur.
OpenACC Livelock	No Error	No Error	Deadlock.
OpenACC Potential Livelock	No Error	No Error	Depending on the environment and the operating order, livelock may or may not occur.
No Error	MPI Deadlock	No Error	Deadlock.
No Error	MPI Potential Deadlock	No Error	Depending on the environment and the order in which the programs are run, a deadlock may or may not happen.

(Continued)

Table 1: Continued

OpenACC	MPI	OpenMP	Tri-Programming Model
No Error	MPI Race Condition	No Error	Race Condition.
No Error	MPI Potential Race Condition	No Error	Depending on the context and running order, a race condition may or may not develop.
No Error	No Error	OpenMP Deadlock	Deadlock
No Error	No Error	OpenMP Potential Deadlock	Depending on the environment and the order in which the programs are run, a deadlock may or may not happen.
No Error	No Error	OpenMP Race Condition	Race Condition.
No Error	No Error	OpenMP Potential Race Condition	Depending on the context and running order, a race condition may or may not develop.
No Error	No Error	OpenMP Livelock	Deadlock.
No Error	No Error	OpenMP Potential Livelock	Depending on the environment and the order in which the programs are run, a deadlock may or may not happen.

When there are errors in two of the three programming models used in this investigation, runtime errors occur, as shown in Listing 22 and Table 2. When a deadlock occurs in one of the programming models, we discovered that the system stops working. Similarly, when a race condition or a data race occurs, it may cause a race condition and may also cause a deadlock based on the operating environment. It is possible that the race will not be noticed by the programmer, especially if the data on which the race occurred is not utilized in the subsequent actions, but if the system is dependent on its outcome, then the system will most likely become stuck in deadlock mode.

Table 2: The table shows the errors that result in the second level, where errors are found in two of the three programming models. The models used are MPI, OpenMP, and OpenACC

OpenACC	MPI	OpenMP	Tri-Programming Model
OpenACC Race Condition	MPI Deadlock	No Error	Deadlock.
OpenACC Race Condition	MPI Race Condition	No Error	A deadlock or race condition might occur based on the environment and operating sequence.
OpenACC Deadlock	MPI Deadlock	No Error	Deadlock.

(Continued)

Table 2: Continued

OpenACC	MPI	OpenMP	Tri-Programming Model
OpenACC Deadlock	MPI Race Condition	No Error	Deadlock.
OpenACC Livelock	MPI Deadlock	No Error	Deadlock.
OpenACC Livelock	MPI Race Condition	No Error	Deadlock.
OpenACC Race Condition	No Error	OpenMP Deadlock	Deadlock.
OpenACC Race Condition	No Error	OpenMP Race Condition	A deadlock or race condition might occur based on the environment and operating sequence.
OpenACC Race Condition	No Error	OpenMP Livelock	Deadlock.
OpenACC Deadlock	No Error	OpenMP Deadlock	Deadlock.
OpenACC Deadlock	No Error	OpenMP Race Condition	Deadlock.
OpenACC Deadlock	No Error	OpenMP Livelock	Deadlock.
OpenACC Livelock	No Error	OpenMP Deadlock	Deadlock.
OpenACC Livelock	No Error	OpenMP Race Condition	Deadlock.
OpenACC Livelock	No Error	OpenMP Livelock	Deadlock.
No Error	MPI Deadlock	OpenMP Deadlock	Deadlock.
No Error	MPI Deadlock	OpenMP Race Condition	Deadlock.
No Error	MPI Deadlock	OpenMP Livelock	Deadlock.
No Error	MPI Race Condition	OpenMP Deadlock	Deadlock.
No Error	MPI Race Condition	OpenMP Race Condition	A deadlock or race condition might occur based on the environment and operating sequence.
No Error	MPI Race Condition	OpenMP Livelock	Deadlock.

Runtime errors happen when there are errors in all the three programming models as indicated in Listing 23 and Table 3. In our investigation, we discovered that many of the mistakes are deadlocks, which are caused by errors occurring in all the programming models. The proposed technique for detecting such errors will be reviewed in the next section.

Table 3: This table displays the errors that occur at the third level, as errors exist in all three programming paradigms

OpenACC	MPI	OpenMP	Tri-Programming Model
OpenACC Race Condition	MPI Deadlock	OpenMP Deadlock	Deadlock.
OpenACC Race Condition	MPI Deadlock	OpenMP Race Condition	Deadlock.
OpenACC Race Condition	MPI Deadlock	OpenMP Livelock	Deadlock.
OpenACC Race Condition	MPI Race Condition	OpenMP Deadlock	Deadlock.
OpenACC Race Condition	MPI Race Condition	OpenMP Race Condition	Based on the environment and operating sequence, a deadlock or race condition might occur.
OpenACC Race Condition	MPI Race Condition	OpenMP Livelock	Deadlock.
OpenACC Deadlock	MPI Deadlock	OpenMP Deadlock	Deadlock.
OpenACC Deadlock	MPI Race Condition	OpenMP Race Condition	Deadlock.
OpenACC Deadlock	MPI Deadlock	OpenMP Livelock	Deadlock.
OpenACC Deadlock	MPI Race Condition	OpenMP Deadlock	Deadlock.
OpenACC Deadlock	MPI Deadlock	OpenMP Race Condition	Deadlock.
OpenACC Deadlock	MPI Race Condition	OpenMP Livelock	Deadlock.
OpenACC Livelock	MPI Deadlock	OpenMP Deadlock	Deadlock.
OpenACC Livelock	MPI Race Condition	OpenMP Race Condition	A deadlock or live lock might occur based on the environment and operating sequence.
OpenACC Livelock	MPI Deadlock	OpenMP Livelock	Deadlock.
OpenACC Livelock	MPI Race Condition	OpenMP Deadlock	Deadlock.
OpenACC Livelock	MPI Deadlock	OpenMP Race Condition	Deadlock.
OpenACC Livelock	MPI Race Condition	OpenMP Livelock	Depending on the environment and running order, the system will either deadlock or livelock.

7 Proposed Architecture

A review of the current state of the tri-programming model, which combines OpenACC, OpenMP, and MPI, revealed that there is no testing technique or debugging tool for detecting runtime errors. This paper proposes a hybrid testing technique for the tri-programming model (MPI + OpenMP + OpenACC), which is integrated into the C++ programming language and is illustrated in Fig. 2. The suggested technique combines dynamic and static testing techniques for the purpose of detecting errors before and during runtime. Combining static and dynamic testing approaches gives it the flexibility to discover potential and actual runtime. Static testing was employed to identify and report issues prior to running. The assertion language is used to automatically identify these errors during execution. A list of detected runtime errors is notified to the programmer.

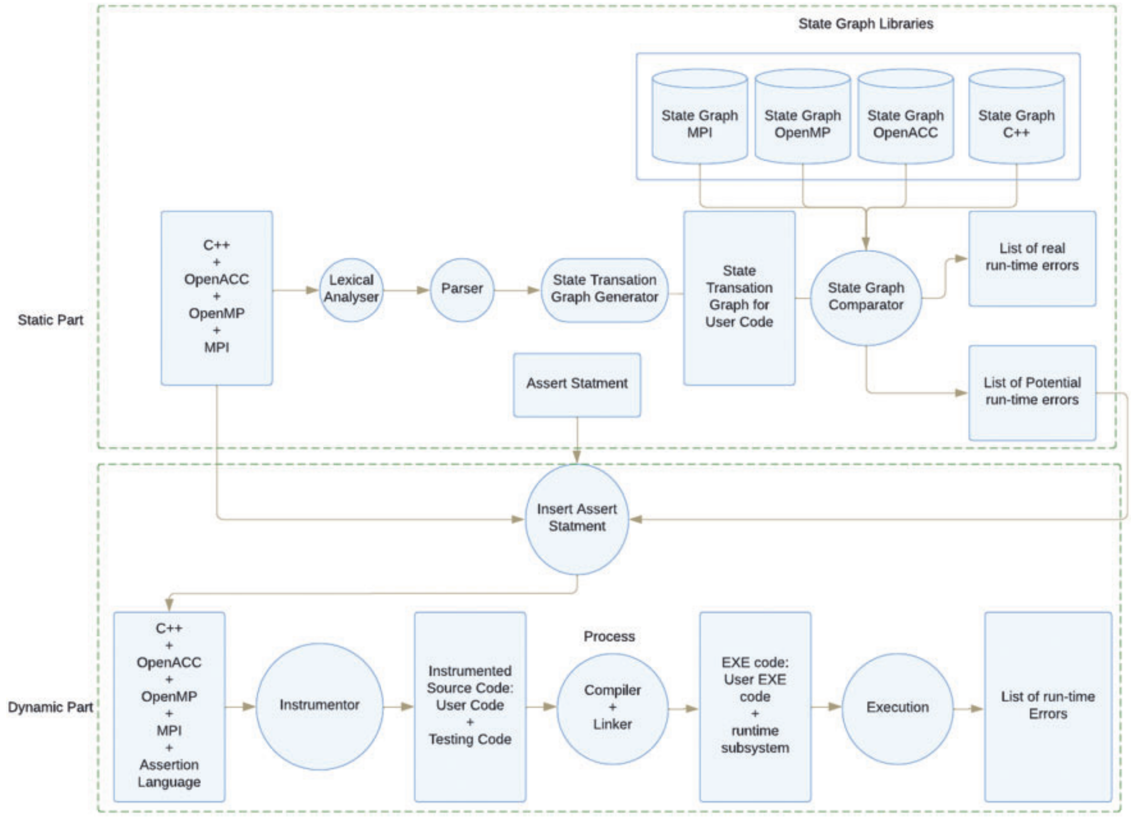
State Graph Libraries
State Graph MPI
State Graph OpenMP
State Graph OpenACC
State Graph C++
Static Part
C++ + OpenACC + OpenMP + MPI
Lexical Analyser
Parser
State Transation Graph Generator
State Transation Graph for User Code
State Graph Comparator
List of real run-time errors
List of Potential run-time errors
Assert Statment
Insert Assert Statment
Dynamic Part
C++ + OpenACC + OpenMP + MPI + Assertion Language
Instrumentor
Instrumented Source Code: User Code + Testing Code
Process
Compiler + Linker
EXE code: User EXE code + runtime subsystem
Execution
List of run-time Errors

Figure 2: The proposed architecture

The code written by the programmer is the input for the static section. The static part is responsible for inspecting each line of code and searching for runtime errors. It identifies both potential and actual errors and handles them appropriately (see Fig. 2). Detecting runtime errors during program execution will take less time if the programmer fixes these errors before running the program. Dynamically identifying errors takes longer without the static component. The dynamic stage must receive the static stage's outputs (potential errors) to identify them during program execution. The following elements comprise the proposed architecture's static component:

- Lexical analyzer: During this step, the programmer's C++ code, which includes MPI, OpenACC, and OpenMP, is the input. After scanning the source code as a stream of characters, this step converts the code into a stream of tokens. Unnecessary comments and spaces are ignored, and data is stored in the table as tokens instead. Examples of errors that can be identified during this step include meeting characters that are not available in the programming language's alphabet or forgetting to close the instruction with a semicolon. A table is used by the lexical analyzer to represent these tokens. The token table is the result of this phase.

- Syntax Analysis: In syntax analysis or parsing, the token table is the input in this stage. At this phase, all syntactic errors have been identified, a diagnostic message has been generated in response to the error, and the program's parse tree has been produced because of the process.
- State transition graph generator: At this point, a parse tree is inserted and a state graph for the user application is generated, which incorporates C++, MPI, OpenACC, and OpenMP. This graph can be represented by an array or a linked list.
- State graph comparator: During this step, the state graph created in the previous stage is inserted and compared to the state graphs of the programming models MPI, OpenACC, and OpenMP, as well as the state graph of C++, which were all created in the previous stage. The output contains all encountered errors. It is added to the list of potential runtime errors, as well as the list of actual runtime problems, if there are any differences between the outcomes of these comparisons. The result of this step that contains actual runtime errors must be addressed by the developer because they will surely occur if they are not addressed. For example, incompatible data types, addition of a string to an integer, multiple declarations of the same variable, reference to a variable before it is declared, and the inclusion of too many arguments in the method call are discovered at this stage. The potential runtime errors will be used as input for the next phase of the proposed architecture's dynamic component.

The dynamic phase of the proposed approach is the second component (see Fig. 2). It is composed of the following components:

- Instrumentor: This part adds the necessary codes to the user code to detect errors during runtime. The inputs include MPI, OpenACC, and OpenMP, as well as C++, the assertion language, and potential runtime errors that were collected during the static phase. Based on the semantics of the assertion language, the instrumentor will create instrumented source code. The instrumented source code consists of two types of code: user codes and testing codes. They are both written in the same programming language as the user code. The result will be a larger code base because it will include both source code and assertion statements that have been added as comments.
- Compiler and linker: The resulting code from the previous step will be generated and linked together, resulting in EXE files that include user executable code as well as additional instructions for testing the code. The last step is to run these EXE programs, which will make a list of runtime errors.

The assertion statement was used to discover errors that occurred during the execution of the program. Before executing the user code, an assertion statement was inserted to detect deadlock errors and other runtime errors as a comment. The instrumentor was responsible for converting the assertion statement into a C++ alternative. In this case, the size of the program is large. To reduce the size and speed up the operation of the program, it is possible to ignore the assertion statement and use only the user code. In this case the added assertion statement will not affect the performance or speed of the system because it will be inserted as a comment. However, it should be used to identify runtime errors during testing.

Future work will include the implementation of the proposed technique and an evaluation of its ability to identify runtime errors that resulted from the tri-programming model, including OpenACC, MPI, and OpenMP, which were implemented in the C++ programming language. Given that it will be the first of its type to integrate three programming models—OpenACC, MPI, and OpenMP—we will be unable to compare it to previous work because no similar work has been discovered so far.

8 Discussion

Many debugging and testing tools are available for identifying runtime errors in parallel applications in a wide range of MPI, CUDA, OpenMP, and OpenACC modules, either individually or in pairs, all of which are discussed in this research. These programming models have become more popular in recent years across a wide range of academic domains working toward the development of exascale computing systems. Debugging and testing techniques are not provided by some integrations, such as OpenACC, MPI, and OpenMP integration, even though they have become increasingly popular in recent years. In addition, many applications in mathematics and software specialize in modeling surgical operations and climate events, wherein errors are difficult to identify due to the large amount of complexity in which they are performed.

In a distributed computing environment, the MPI programming model is essential because it distributes work across multiple scalable devices. Five of the thirteen applications designed to enhance the performance of the world's most powerful supercomputer utilized OpenACC. that the usage of OpenACC is anticipated to increase. In recent years, numerous academics have cited OpenMP as a crucial tool for parallel thread-based programming on shared memory systems, such as multi-core CPUs, and its widespread use continues to this day. OpenACC is typically preferred for GPU systems, whereas OpenMP is typically preferred for CPU systems. A coding error may result in runtime errors that do not manifest during compilation but that have severe consequences and high costs. In large programs, runtime errors such as deadlock, data race, and livelock cannot be discovered by the compiler.

Static and dynamic testing are the two testing techniques that we have included in our research. These two types have been integrated to gain benefits from both techniques. The first component of the hybrid technique is a static testing approach that evaluates the source code prior to execution to discover runtime errors. As a result, the programmer may acquire a list of runtime errors and correct them before the actual run; otherwise, identifying the errors will take a longer time. Potential errors are forwarded to the second component of the system, which is the dynamic portion, which identifies errors that occur during runtime. In addition, given the structure and behavior of parallel programs, examining these programs in real time is a challenging task. Because of this, the amount of effort required by the testing tool to cover every possible scenario involving the test cases and data would rise significantly.

Furthermore, because dynamic techniques are affected by the events that may occur during execution and the possible different test cases, they may have a detrimental impact on the overall execution time of the system. Nevertheless, by detecting these errors using static techniques, we can minimize the amount of time required to find them. The kind and behavior of runtime errors, as well as other factors, dictate which approaches should be employed given that static analysis and other techniques cannot identify some errors that can be detected through dynamic techniques, whereas dynamic techniques cannot detect some errors that can be detected through static techniques.

9 Conclusions

In recent years, increasing interest in high-performance computing has been seen. Building massively parallelized huge computer systems based on heterogeneous architecture is becoming increasingly vital for improving performance and enabling exascale computing. One possible way to achieve this is by merging two or more programming models, such as MPI, OpenMP, and OpenACC, into a single application. Although the integration delivers several benefits and accelerates the system's performance, it is rendered ineffective if it is accompanied by runtime errors that are concealed from the programmer. Testing such complicated systems by programmers is difficult, and they cannot guarantee that the program is error-free.

In this paper, we propose a hybrid parallel testing technique for detecting runtime errors in systems built in the C++ programming language using a tri-programming model, that combines MPI, OpenMP, and OpenACC. To the best of our knowledge, no testing technique for detecting runtime errors in systems that incorporate these three programming models has been developed. Static and dynamic testing techniques are combined in this suggested solution. When the two methods are used together, a wide range of errors can be found.

A classification of errors that result from this integration is provided. The proposed classification has three levels. The first level is reached when an error exists in one of the programming models mentioned. The second level is reached when two errors occur in two of the programming models, and the third level is reached when all three models contain errors. In future work, the suggested technique will be applied, and its ability to find runtime errors caused by this integration will be evaluated.

Acknowledgement: This work was funded by the Deanship of Scientific Research (DSR), King Abdulaziz University, Jeddah, under Grant No. KEP-PHD-20-611-42. The authors would like to gratefully acknowledge DSR for technical and financial support.

Funding Statement: This research was funded by [King Abdulaziz University] [Deanship of Scientific Research] Grant Number [KEP-PHD-20-611-42].

Conflicts of Interest: The authors declare that they have no conflicts of interest to report regarding the present study.

References

[1] M. Alghamdi and F. Eassa, "OpenACC errors classification and static detection techniques," *IEEE Access*, vol. 7, pp. 113235–113253, 2019.

[2] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. Mcdonald *et al., Parallel Programming in OpenMP*, NY: Morgan Kaufmann Publishers, 2000.

[3] C. Warren, A. Giannopoulos, A. Gray, I. Giannakis, A. Patterson *et al.,* "A CUDA-based GPU engine for gprmax: Open source FDTD electromagnetic simulation software," *Computer Physics Communication*, vol. 237, pp. 208–218, 2019.

[4] A. Munshi, B. Gaster, T. Mattson and D. Ginsburg, "An introduction to OpenCL," in *OpenCL Programming Guide*, 1st ed., Upper Saddle River N.J.: Addison-Wesley, pp. 3–5, 2011.

[5] M. Aldinucci, V. Cesare, I. Colonnelli, A. Martinelli, G. Mittone *et al.,* "Practical parallelization of scientific applications with OpenMP, OpenACC and MPI," *Journal of Parallel and Distributed Computing*, Elsevier, vol. 157, pp. 13–29, 2021.

[6] Z. Chen, H. Yu, X. Fu and J. Wang, "MPI-SV: A symbolic verifier for mpi programs," in *2020 IEEE/ACM 42nd Int. Conf. on Software Engineering: Companion Proc. (ICSE-Companion)*, IEEE, Seoul, South Korea, pp. 93–96, 2020.

[7] F. Cabral, C. Osthoff, M. Kischinhevsky and D. Brandão, "Hybrid MPI/OpenMP/OpenACC implementations for the solution of convection-diffusion equations with the HOPMOC method," in *Proc. of XXIV Int. Conf. on Computational Science and Its Applications (2014)*, IEEE, Guimaraes, Portugal, pp. 196–199, 2014.

[8] P. Ouro, B. Fraga, U. Lopeznovoa and T. Stoesser, "Scalability of an eulerian-lagrangian large-eddy simulation solver with hybrid MPI/OpenMP parallelisation," *Computers & Fluids*, vol. 179, pp. 123–136, 2019.

[9] A. Eghtesad, T. Barrett, K. Germaschewski, R. Lebensohn, R. McCabe *et al.,* "OpenMP and MPI implementations of an elasto-viscoplastic fast Fourier transform-based micromechanical solver for fast crystal plasticity modeling," *Advances in Engineering Software*, vol. 126, pp. 46–60, 2022.

[10] W. Kwedlo and P. Czochanski, "A hybrid MPI/OpenMP parallelization of means algorithms accelerated using the triangle inequality," *IEEE Access*, vol. 7, pp. 42280–42297, 2019.

[11] S. Blair, C. Albing, A. Grund and A. Jocksch, "Accelerating an MPI lattice boltzmann code using OpenACC," in *Proc. of the Second Workshop on Accelerator Programming Using Directives*, Austin Texas, pp. 1–9, 2015.

[12] S. Alshahrani, W. AlShehri, J. Almalki, A. Alghamdi and A. Alammari, "Accelerating spark-based applications with MPI and OpenACC," *Complexity*, Hindawi, vol. 2021, pp. 1–17, 2022.

[13] H. Zhang, J. Zhu, Z. Ma, G. Kan, X. Wang *et al.,* "Acceleration of three-dimensional tokamak magnetohydrodynamical code with graphics processing unit and OpenACC heterogeneous parallel programming," *International Journal of Computational Fluid Dynamics*, vol. 33, no. 10, pp. 393–406, 2019.

[14] J. Kraus, M. Schlottke, A. Adinetz and D. Pleiter, "Accelerating a C++ CFD code with OpenACC," in *2014 First Workshop on Accelerator Programming Using Directives*, New Orleans, LA, USA, IEEE, pp. 47–54, 2014.

[15] S. Pophale, S. Boehm, L. Vergara and G. Veronica, "Comparing high performance computing accelerator programming models," in *Int. Conf. on High Performance Computing*, New York, NY, USA, 11887 LNCS, pp. 155–168, 2019.

[16] S. Wienke, C. Terboven, J. Beyer and M. Müller, "A pattern-based comparison of OpenACC and OpenMP for accelerator computing," in *European Conf. on Parallel Processing*, Porto, Portugal, Springer, pp. 812–823, 2014.

[17] J. Lambert, S. Lee, A. Malony and J. Vetter, "CCAMP: OpenMP and OpenACC interoperable framework," in *European Conf. on Parallel Processing*, Cham, Springer, pp. 357–369, 2019.

[18] J. Herdman, W. Gaudin, O. Perks, D. Beckingsale, A. Mallinson *et al.,* "Achieving portability and performance through OpenACC," in *2014 First Workshop on Accelerator Programming Using Directives*, IEEE, New Orleans, LA, USA, pp. 19–26, 2014.

[19] R. Usha, P. Pandey and N. Mangala, "A comprehensive comparison and analysis of OpenACC and OpenMP 4.5 for NVIDIA GPUs," in *2020 IEEE High Performance Extreme Computing Conf. (HPEC)*, Boston, IEEE, pp. 1–6, 2022.

[20] A. Goyal, Z. Li and H. Kimm, "Comparative study on edge detection algorithms using OpenACC and OpenMPI on multicore systems," in *2017 IEEE 11th Int. Symp. on Embedded Multicore/Many-Core Systems-on-Chip (MCSoC)*, Seoul, South Korea, IEEE, pp. 67–74, 2017.

[21] T. Hilbrich, J. Protze, M. Schulz, B. Supinski and M. Müller, "MPI runtime error detection with MUST: Advances in deadlock detection," *Scientific Programming*, IOS Press, vol. 21, no. 3, pp. 109–121, 2013.

[22] Z. Chen, H. Yu, X. Fu and J. Wang, "MPI-SV: A symbolic verifier for MPI programs," in *Proc.-2020 ACM/IEEE 42nd Int. Conf. on Software Engineering: Companion, ICSE-Companion 2020*, Seoul, South Korea, pp. 93–96, 2020.

[23] H. Ma, S. Diersen and L. Wang, "Symbolic analysis of concurrency errors in openmp programs," in *2013 42nd Int. Conf. on Parallel Processing*, Lyon, France, IEEE, pp. 510–516, 2013.

[24] V. Basupalli, T. Yuki, S. Rajopadhye, A. Morvan, S. Derrien *et al.,* "OmpVerify: Polyhedral analysis for the OpenMP programmer," in *Int. Workshop on OpenMP*, Berlin, Heidelberg, Springer, vol. 6665, pp. 37–53, 2011.

[25] P. Chatarasi, J. Shirako, M. Kong and V. Sarkar, "An extended polyhedral model for SPMD programs and its use in static data race detection," in *Int. Workshop on Languages and Compilers for Parallel Computing*, Cham, IEEE, vol. 10136, pp. 106–120, 2017.

[26] F. Ye, M. Schordan and C. Liao, "Using polyhedral analysis to verify openmp applications are data race free," in *2018 IEEE/ACM 2nd Int. Workshop on Software Correctness for HPC Applications*, Dallas, TX, USA, IEEE, pp. 42–50, 2018.

[27] S. Siegel, "Verifying parallel programs with MPI-spin," in *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, New York, NY, US: LNPSE, vol. 4757, pp. 13–14, 2007.

[28] E. Saillard, P. Carribault and D. Barthou, "Combining static and dynamic validation of MPI collective communications," *The International Journal of High Performance Computing Applications*, New York, NY, US, vol. 28, no. 4, pp. 425–434, 2014.

[29] G. R. Luecke, Y. Zou, J. Coyle, J. Hoekstra and M. Kraeva, "Deadlock detection in MPI programs," *Concurrency and Computation: Practice and Experience*, vol. 14, no. 11, pp. 911–932, 2002.

[30] H. Ma, L. Wang and K. Krishnamoorthy, "Detecting thread-safety violations in hybrid OpenMP/MPI programs," in *2015 IEEE Int. Conf. on Cluster Computing*, Chicago, IL, USA, IEEE, pp. 460–463, 2015.

[31] M. Ali, P. Strazdins, B. Harding, J. Southern and P. Strazdins, "Application level fault recovery: Using fault-tolerant open MPI in a PDE solver," in *2014 IEEE Int. Parallel & Distributed Processing Symp. Workshops*, Phoenix, AZ, USA, IEEE, pp. 1169–1178, 2014.

[32] Z. Yang, Y. Zhu and Y. Pu, "Parallel image processing based on CUDA," in *2008 Int. Conf. on Computer Science and Software Engineering*, Washington, DC, US, IEEE, vol. 3, 2008.

[33] F. Eassa, A. Alghamdi, S. Haridi, M. Khemakhem, A. Alghamdi *et al.,* "ACC_TEST: Hybrid testing approach for OpenACC-based programs," *IEEE Access*, vol. 8, pp. 80358–80368, 2020.

[34] A. Alghamdi, F. Eassa, M. Khamakhem, A. Alghamdi, A. Alfakeeh *et al.,* "Parallel hybrid testing techniques for the dual-programming models-based programs," *Symmetry*, vol. 12, no. 9, pp. 15–55, 2020.

[35] M. Park, S. Shim, Y. Jun and H. Park, "MPIRace-check: Detection of message races in MPI programs," *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 4459, pp. 322–333, 2007.

[36] S. Vakkalanka, S. Sharma, G. Gopalakrishnan and R. Kirby, "ISP: A tool for model checking MPI programs," in *Proc. of the 13th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, Salt Lake, UT, US, pp. 285–286, 2008.

[37] M. Zheng, V. Ravi, F. Qin and G. Agrawal, "Gmrace: Detecting data races in gpu programs via a low-overhead scheme," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 1, pp. 104–115, 2013.

[38] K. Vorobyov and P. Krishnan, "Comparing model checking and static program analysis: A case study in error detection approaches," *Proceedings of SSV*, vol. 1, pp. 1–7, 2010.

[39] H. Ma, S. Diersen and L. Wang, "Symbolic analysis of concurrency errors in openmp programs," in *2013 42nd Int. Conf. on Parallel Processing*, Lyon, France, IEEE, pp. 510–516, 2013.

[40] G. Luecke, H. Chen, J. Coyle, J. Hoekstra, M. Kraeva *et al.,* "MPI-CHECK: A tool for checking fortran 90 MPI programs," *Wiley Online Library*, vol. 15, no. 2, pp. 93–100, 2003.

[41] B. Krammer, M. Müller and M. Resch, "MPI application development using the analysis tool MARMOT," *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 3038, pp. 464–471, 2004.

[42] V. Forejt, D. Kroening, G. Narayanaswamy and S. Sharma, "A precise predictive analysis for discovering communication deadlocks in MPI programs," in *Int. Symp. on Formal Methods*, Singapore, Springer, vol. 39, no.4, pp. 263–278, 2014.

[43] M. Park, "Detecting race conditions in one-sided communication of MPI programs," in *Eighth IEEE/ACIS Int. Conf. on Computer and Information Science*, Shanghai, China, IEEE, pp. 867–872, 2009.

[44] M. Park and S. Chung, "MPIRace-check v 1.0: A tool for detecting message races in MPI parallel programs," *The KIPS Transactions: PartA, Korea Information Processing Society*, Korea, vol. 15, no. 2, pp. 87–94, 2008.
[45] Y. Kim, S. Song and Y. Jun, "Adat: An adaptable dynamic analysis tool for race detection in OpenMP programs," in *2011 IEEE Ninth Int. Symp. on Parallel and Distributed Processing with Applications*, NW Washington, DC US, pp. 304–310, 2011.
[46] Y. Gu and J. Mellorcrummey, "Dynamic data race detection for openmp programs," in *SC18: Int. Conf. for High Performance Computing, Networking, Storage and Analysis*, Dallas, Texas, IEEE, pp. 767–778, 2018.
[47] Y. Lin, "Static nonconcurrency analysis of OpenMP programs," in *Int. Workshop on OpenMP*, Berlin, Heidelberg, Springer, vol. 4315, pp. 36–50, 2005.
[48] B. Swain, Y. Li, P. Liu, I. Laguna, G. Georgakoudis *et al.,* "Ompracer: A scalable and precise static race detector for OpenMP programs," in *SC20: Int. Conf. for High Performance Computing, Networking, Storage and Analysis*, Atlanta, Georgia, IEEE, pp. 1–14, 2020.