Errors Classification and Static Detection Techniques for Dual-Programming Model (OpenMP and OpenACC)

HUĐA SALEH BASLOOM1, MOHAMED YEHIA DAHAB1,2, AHMED MOHAMMED ALGHAMDI3, FATHY ELBOURAEY EASSA1, ABDULLAH SAAD AL-MALAISE AL-GHAMDI4, AND SEIF HARIDI5

1Department of Computer Science, Faculty of Computing and Information Technology, King Abdulaziz University, Jeddah 21514, Saudi Arabia
2Agricultural Research Center (ARC), Giza 12619, Egypt
3Department of Software Engineering, College of Computer Science and Engineering, University of Jeddah, Jeddah 21493, Saudi Arabia
4Department of Information Systems, Faculty of Computing and Information Technology, King Abdulaziz University, Jeddah 21589, Saudi Arabia
5Department of Computer Science, KTH Royal Institute of Technology, 114 28 Stockholm, Sweden

Corresponding author: Huda Saleh Basloom (hbasaloom@kau.edu.sa)

This work was supported by the Deanship of Scientific Research (DSR), King Abdulaziz University, Jeddah, under Grant KEP-PHD-20-611-42.

ABSTRACT Recently, incorporating more than one programming model into a system designed for high performance computing (HPC) has become a popular solution to implementing parallel systems. Since traditional programming languages, such as C, C++, and Fortran, do not support parallelism at the level of multi-core processors and accelerators, many programmers add one or more programming models to achieve parallelism and accelerate computation efficiently. These models include Open Accelerators (OpenACC) and Open Multi-Processing (OpenMP), which have recently been used with various models, including Message Passing Interface (MPI) and Compute Unified Device Architecture (CUDA). Due to the difficulty of predicting the behavior of threads, runtime errors cannot be predicted. The compiler cannot identify runtime errors such as data races, race conditions, deadlocks, or livelocks. Many studies have been conducted on the development of testing tools to detect runtime errors when using programming models, such as the combinations of OpenACC with MPI models and OpenMP with MPI. Although more applications use OpenACC and OpenMP together, no testing tools have been developed to test these applications to date. This paper presents a testing tool for detecting runtime errors using a static testing technique. This tool can detect actual and potential runtime errors during the integration of the OpenACC and OpenMP models into systems developed in C++. This tool implement error dependency graphs, which are proposed in this paper. Additionally, a dependency graph of the errors is provided, along with a classification of runtime errors that result from combining the two programming models mentioned earlier.

INDEX TERMS Parallel programming, high-performance computing, OpenMP, OpenACC, deadlock, runtime errors, race condition, static testing, software testing, testing tools classifications, exascale systems, programming models.

I. INTRODUCTION Working on high-performance devices and exploiting all sources to speed up a system are some of the means to achieving exascale computing even with the end of Dennard Scaling and the slowing of Moore’s law [1], [2], [3], [4]. Traditional programming languages such as C, C++, and Fortran support memory-level parallelism in modern versions, but they are unable to distribute work to accelerators and multi-core processors. Therefore, programmers add one
or more programming models to achieve parallelism, speed up computation, and efficiently exploit computer resources. Open Accelerators (OpenACC) [5], [6], Compute Unified Device Architecture (CUDA) [7], Open Multi-Processing (OpenMP) [8], and Open Computing Language (OpenCL) [9] are examples of programming models that can be incorporated into programming languages and allow systems to run faster by utilizing parallelism and exploiting all available device resources, such as graphics processing unit (GPU)s, memory sharing, and multi-core CPUs. Each programming model has its own method of applying parallelism as well as the directives that implement parallelism in the system.

It is possible to combine more than one model to increase the speed and efficiency of a system as a whole. Each programming model has unique characteristics. For instance, one programming model can be used to exploit the GPU, while another model can be used to exploit the memory. When merging two models, care must be taken to prevent programming issues that the compiler cannot identify. The testing of the program to ensure that it is error-free will cost more time if more than one programming model is integrated into it; however, if the integration of these models is done correctly, the speed and efficiency of the program will increase as a result. This is due to the efficient use of the resources available in the computer, such as the GPU and memory, to distribute up the task and complete it in a shorter period of time. There is a wide range of studies from numerous fields of science that have utilized more than one programming model and merged the advantages of each of these models into a single system. Heterogeneous systems are used in the areas of high-performance computing (HPC), machine learning, and embedded computing. Heterogeneous systems are composed of more than one type of hardware (e.g., multi-core CPUs, GPUs, and digital signal processors). The greatest possible value may be obtained through effective resource management in these systems, as well as through the integration of more than one programming model. For example, in the systems designed [8], [10], [11], [12], [13], two programming models, OpenACC and OpenMP, were implemented in parallel. It is possible to improve the performance and portability of heterogeneous systems by using devices that are supported. Additionally, the two programming models can communicate with one another within the same process and run faster as a result of the work being distributed across multiple resources. Parallel systems are prone to runtime errors during execution, such as deadlocks, race conditions, data races, livelocks, and other types of runtime errors. This is due to the nature of parallel systems, where more than one thread can run simultaneously, and their behaviour cannot be predicted.

This research aims to develop static techniques for detecting errors that result from the merger of OpenACC and OpenMP in systems that are built with C++. This tool also implements the error dependency graph that is proposed in this paper. This paper also includes a classification of runtime errors that can occur as a result of this merger and includes an explanation of these errors’ causes with examples. Compilers are unable to detect these kinds of runtime issues. The capacity of the tool that we designed was also evaluated using available benchmarks, and the results were compared to other testing tools. The latest versions of the OpenMP and OpenACC programming models—5.2 and 3.2 respectively—were used in our testing, and version 20.4 of the Portland Group Inc. (PGI) compiler was used for our experiments and application. This research is the first to identify and classify runtime errors on systems that use both OpenACC and OpenMP. This is also the first research to develop a static testing tool to detect these errors. To the best of our knowledge, there is no testing tool that identifies OpenACC and OpenMP runtime errors, nor is there a classification for errors that result from merging these two models.

The remainder of this paper is organized in the following manner: Section II provides a brief overview of the utilized programming models and the available testing techniques. Section III addresses the dual-level programming model, Section IV presents related works, Section V illustrates some of the errors that resulted from the combination of the two models that were employed, Section VI presents a classification of runtime errors that occur from this integration, and Section VII includes the algorithms that were developed to detect the runtime errors that occurred as a result of the merging of the two models. The results of testing using this tool are presented in Section VIII, and a summary of what has been accomplished is presented in Section IX.

II. BACKGROUND

In this section, we present a simple background on the OpenMP and OpenACC programming models, and available testing techniques.

OpenMP is a programming interface that is used to create node parallelism in high-performance computing (HPC). OpenMP 1.0 was first launched in 1997. For many years, the OpenMP programming paradigm was well known for being a directive-based programming model, which made it simple for programmers to learn and easy for them to incorporate into their programs. It is used to parallelize the shared memory amongst several simultaneously executing threads. Originally intended to support only multi-core systems with shared memory, OpenMP has been extended to include offload loops for other GPUs and accelerators [14], [15], which is also the primary target of OpenACC [5], [16]. OpenMP has supported the accelerator since its fourth version, which was released in 2013. In November 2021 [15], the latest version, 5.2, was made available for use, and it is compatible with the programming languages C, C++, and Fortran, among others. There are a number of OpenMP implementations available, including [17], [18], [19].

OpenACC was introduced in 2010 as a programming model that supported work on heterogeneous accelerators [20], [21]. OpenACC, which was developed by organizations such as Nvidia, Portland Group, Inc.(PGI), and Cray [20], has quickly gained popularity as one of the most popular
programming models for high-performance computing systems today [8], [16]. The most recent version of the OpenACC standard is 3.2 [22], which was released in November 2021. It is designed to help software developers transform applications that need to distribute work in parallel on heterogeneous HPC devices and allow them to use the accelerator with less programming effort than low-level programming paradigms, such as CUDA and OpenCL. The data management and transformation between a host and an accelerator is hidden from the programmer. There are a number of OpenACC implementations available, including [21], [22], [23], [24], [25], [26], [27], [28], [29]. OpenACC is compatible with the C, C++, and Fortran programming languages, and a variety of hardware architectures, including X86 and POWER processors, as well as NVIDIA GPUs.

Building parallel applications in heterogeneous systems is a challenging task. Parallel applications can manifest uncertain behaviors if they are poorly programmed. Runtime errors vary from one programming model to another, and it is difficult to guarantee that the source is error-free, especially when potential runtime errors emerge. There are two types of runtime errors: actual runtime errors and potential runtime errors. Actual runtime errors appear each time the program is run, but potential do not. Potential runtime errors depend on the runtime environment and the data being processed. It is difficult to ensure that there are no potential errors in a system. There has been research conducted to explore and identify parallel application errors, along with the reasons for these failures.

Testing parallel applications may be a complicated task, particularly when combining more than one programming model into a single application. There are many runtime errors that are not detected by the compiler and arise after the program is run. When combining more than one programming model, a new runtime error can emerge, which results in undefined behavior during the running process. Consequently, it is difficult to detect parallel errors when they occur, and even after these errors are modified, there may be hidden errors that do not manifest themselves every time the program is executed. Even if these errors are discovered and the source code has been modified, it can be difficult to determine whether the errors have been corrected or whether they are still hidden within the code.

There are various testing methods [5], [30], including static methods that test the system before runtime. The source code is examined using this technique to find a variety of errors, whether they are actual runtime errors or potential runtime errors. Compared to dynamic testing methods and model-checking testing methods, this method is faster. The dynamic technologies test the system during runtime. This technique requires more time to test the system, as additional codes are added to the user source code for testing. The new code is bigger, and consequently, it takes longer to find these errors [5], [30], [31]. Additionally, the dynamic method detects fewer errors than the static method. In order to benefit from both technologies, hybrid testing techniques can combine two testing methods, such as static and dynamic testing methods. There is also a model checking method, which is similar to the static method but requires the development of test cases using a different application and requires more time to find all probably means [32], [33]. This technique is suitable for small systems but not for large systems; thus, its use in this study was avoided. As revealed in the literature review, the static testing technique is faster than dynamic testing methods and model-checking testing methods. In future research, the combination of them with dynamic technologies to make a hybrid tool can be explored.

### III. Dual-Level Programming Model (OpenMP + OpenACC)

A ‘dual-level programming model’ is a programming model that combines the features of two different programming models [5], [30]. This combination will aid in the transition to exascale systems, which will require the development of more powerful programming models that allow supercomputers to operate in parallel. OpenMP and OpenACC each have their own set of advantages, and the combination of the two will allow for increased parallelism, improved performance, and reduced programming effort required because they will run on heterogeneous devices. Through the use of OpenACC, it is possible to compile it into a variety of devices, including numerous GPUs. OpenMP will be used for threading, task distribution, and memory sharing in multi-core CPUs, as well as in a variety of hardware architectures.

Both OpenMP and OpenACC, can be programmed using C, C++, and Fortran, and parallelism is supported by all of them in the accelerator. Each has its own advantages, and they are not interchangeable. References [10], [11], [12], [34], [35], [36], [37], [38], and [39], present more information about the comparison of the two programming models. OpenACC was found to be more efficient in accelerators than OpenMP, which was found to be more efficient in memory management. Therefore, combining the two programming models can provide benefits from both of them while increasing the speed of the system. In addition, the use of libraries provided by both models, such as the mathematics library in OpenACC, provides another benefit.

An application built using the hybrid parallel programming paradigm can run on a computer cluster using both OpenMP and OpenACC, where OpenMP is used for parallelism within a node (multi-core) while OpenACC is used to distribute work on the GPU. The main objective of this integration is to speed up the process and effectiveness of applications by proficiently exploiting computer resources, using the libraries of each programming model, and maximizing the use of heterogeneous systems.

### IV. Related Work

There are many different sorts of testing techniques, including static testing techniques, dynamic testing techniques, and hybrid testing approaches. In static techniques, the source code is examined before the system is executed, whereas in
dynamic technologies, the source code is checked while the system is running. The disadvantages of dynamic techniques, on the other hand, are the large number of test cases that can be expected and the fact that some errors are hidden and therefore cannot be detected every time the system is started up. Hybrid technologies, on the other hand, are ones that incorporate more than one testing method, such as a combination of static and dynamic technologies. The topic has been covered in our previous study [30].

For example, OmpVerify [40], PolyOMP [41], and DRACO [43] are static techniques that are used in design testing tools to detect runtime errors in systems that contain OpenMP directives. There have also been a number of studies conducted with dynamic techniques to detect runtime errors in systems containing OpenMP directives, such as Helgrind [44], ADAT [45], Valgrind [46], ROMP [47] and TSan [48]. Dynamic technology is also applied in testing systems that combine the MPI with OpenMP, such as Marmot [49] and hybrid technology, such as [50].

In systems that contain the OpenACC directive, the ACC_TEST is an example of hybrid technology that is meant to test runtime faults in systems that contain the OpenACC. It has also been tested with systems that use the MPI in conjunction with OpenACC in ACC_TEST [6]. During this study, it was discovered that there had been little research into testing tools for the OpenACC; however, it was also found that there was an increasing number of research projects that were going to use it, which inspired us to create a testing tool for the OpenACC with OpenMP, which served as inspiration for the creation of a testing tool for the OpenACC with OpenMP in the present study.

Another method that is comparable to static techniques is model checking [51], [52]. Model checking determines runtime states before runtime and checks for errors statically. It ensures that there are neither races nor deadlocks by simulating their conditions. Non-deterministic decisions need to be thoroughly investigated. Among many other possibilities, non-deterministic options can include program inputs, typical external system behavior, thread similarities in a multi-threaded application, and message instructions in a distributed system. The program must be converted into a modeling language to be used in this technique. However, it is quite challenging to manually extract the model due to the large number of codes that may be tested. Furthermore, there are simply too many possible outcomes to take into account in any depth. As a result, the testing of this software takes longer as test cases increase in size for large projects. Additionally, testing requires effort to plan and ensure that every scenario is taken care of. If any potential test cases are overlooked, it will have an impact on the test. As a result, testing a program takes longer as its size increases. Only small, essential applications should be tested using it. It is not recommended for larger projects [32], [33].

There are a number of paid debugging tools but it is not testing tool, including Archer [42], Totalview [53], Arm DDT [54], Intel Inspector [55], that concentrate on finding memory errors while they are being run. It contains a substantial number of programming models, including MPI, OpenMP, OpenACC, and CUDA. These debuggers track memory errors during runtime. However, there are numerous runtime errors that need to be investigated and identify their real causes, including deadlock, data race, and live lock.

From a review of the literature, it was found that there is no testing tool for testing runtime errors that develop from merging OpenMP and OpenACC programming models. The present research focuses on employing static techniques to detect a significant number of runtime errors that occur as a result of integrating OpenMP and OpenACC in the C++ programming language. This technique examines the software source code before it is executed and identifies runtime errors, whether they are actual or potential, and displays them in a report to allow the programmer to fix them. One of the advantages of this technology is that it is faster and more efficient than dynamic techniques and it also reveals a large number of errors, including actual and potential errors. One of the main objectives of the present study is to classify these errors first, because there is currently no classification for errors that result from the combination of the two OpenMP OpenACC programming models. This categorization will assist us in creating the tool we suggested. As part of our ongoing research, we will enhance this tool by incorporating dynamic error detection to cover a greater number of possible faults. In addition to this, codes are added to complete the fundamental project in order to carry out the dynamic test. This enables the creation of a hybrid tool that consists of both the static test component and the dynamic test component.

V. RUNTIME ERRORS IN DUAL-PROGRAMMING MODEL

OpenMP AND OpenACC

In parallel systems, there are various sorts of runtime errors that can arise after program run and the compiler is incapable of identifying them. When hybrid models are used, the percentage of these mistakes increases and the reasons for these errors are different from those of traditional models [5], [6], [8], [16], [30], [36], [56], [57], [58], [59]. Parallel systems have many advantages, but in order to reap the benefits of these advantages, they must be error-free. This section discusses the errors that are encountered when integrating the OpenMP and OpenACC programming models into C++ applications and illustrates these concerns using actual examples. The OpenACC and OpenMP directives are used throughout the following codes, which are written in the C++ programming language. Some mistakes are formed as a consequence of an error in one of the two models, while others are produced despite the absence of an error in either of the two models. To illustrate this dependency in errors, a dependency graph has been created.

There has been much research on deadlock and its causes, particularly in parallel programming, [16], [27], [28], [29], [58] and many of these works have addressed it in further detail. Typically, a deadlock happens when two or more threads are waiting for input from each other; an infinite
loop occurs between these threads, which prohibits any thread from proceeding. One of our earlier research studies [30] highlighted some of the reasons for deadlock in the OpenMP, OpenACC, and MPI programming paradigms.

A race condition may develop as a result of a large number of threads operating in the background. Accessing a shared resource, such as a variable in memory, is an example of a race condition. The outcome can change depending on the order in which the threads are performed. Access to shared variables must be synchronized in order to avoid the occurrence of race conditions. The techniques to identify race conditions in each programming model are different, and examples of these techniques are described in [10], [11], [12], [14], and [13]. More information may be found in the reference [30]. A data race occurs when two or more execution threads in a multi-threaded application attempt to simultaneously access the same shared variable and there is at least one write access to the shared variable [10], [12].

When it comes to the OpenMP programming model, ‘lock’ routines are some of the most sensitive instructions available. There are two types of locks to consider: nested and simple. Both lock types should be used with extreme caution and in the exact order. The first step is to initialize the ‘lock’ before using it. For a simple ‘lock’, this is accomplished by using ‘omp_init_lock’ and for a ‘nested lock’, this is accomplished by calling ‘omp_init_nest_lock’. It is not possible to describe the outcome if the programmer fails to initialize the ‘lock’ [58]. Another method of setting a lock is to use the commands ‘omp_test_lock’ for a basic lock or ‘omp_test_nest_lock’ for a nested lock. If the test lock has not yet been set, it will take over and acquire it. If the lock has already been set, the function will return 1; if the lock has not been set, the function will return 0. When testing, the test lock should be utilized after initialization rather than before, as otherwise the outcome would be unpredictable.

The second step is to set the ‘lock’ and begin working on the variable. For a basic ‘lock’, the command ‘omp_set_lock’ should be used, while for a nested lock, the command ‘omp_set_nest_lock’ should be used. The nested lock can be set several times to accomplish the same operation before being released, but the basic lock must be released before it can be set a second time. The third step is to release the lock after completion of the job by using ‘omp_unset_lock’ for a basic lock or ‘omp_unset_nest_lock’ for a nested lock upon completion of the task.

The lock must be destroyed in the final step so that no more space in the memory is occupied by the lock. With a basic lock, the command ‘omp_destroy_lock’ can be used to release it from memory. For nested locks, the command ‘omp_destroy_nest_lock’ can be used instead. It will have no effect on the program if the unset and destroy functions are not invoked; nonetheless, the lock will continue to occupy the memory space allocated for it.

In OpenMP, a deadlock occurs if the same variable is locked more than one time before the ‘unset’ is performed. An example of code for lock the same variable ‘lock_a’ before unlock (see Listing 1). It is necessary to ‘unset’ the lock in line 5 before it can be re-locked in line 7.

A similar problem occurs in the code sample shown in listing Listing 2 as well. The variable ‘lock_a’ is initialized in line 6, and there is no unlock in the same for loop. The same variable is locked in the first round, then it is acquired again in succeeding turns, without the variable being unlocked in between. To resolve this issue, unset can be included within the loop.

The ‘Section’ directive in OpenMP specifies which code should be distributed among all threads. The “Sections” directive may have zero or more section directives. One of the threads runs each structured block once. The sets and unosets for the same variable should be in the same Section of the code block. Listing 3 shows an example of a variable lock_a being set in line 6 in the first Section, but being unset and deleted in lines 11 and 12, which are placed in a different Section. A mistake in OpenMP resulted in a

---

Listing 1. Deadlock occurs across the system as a result of the lock occurring on the same variable in lines 5 and 7 without an unset before line 7.

```c
omp_lock_t lock_a;
omp_init_lock(&lock_a);
#pragma omp parallel shared(lock_a)
{
  omp_set_lock(&lock_a);
  /*-----Do Work-----*/
  omp_set_lock(&lock_a);
  omp_unset_lock(&lock_a);
}
omp_destroy_lock(&lock_a);
#pragma acc kernels
for (i=0;i<N;i++)
  y[i]=x[i]*(i+1);
}
```

Listing 2. A deadlock occurs across the system as a result of a lock being applied to the same variable in line 6 and not being unlocked in the same for loop.

```c
omp_lock_t lock_a;
omp_init_lock(&lock_a);
#pragma omp section
{
  omp_set_lock(&lock_a);
  for (i=0;i<N;i++)
    x[i]=i*2;
  omp_unset_lock(&lock_a);
  omp_destroy_lock(&lock_a);
}
#pragma acc kernels
for (i=0;i<N;i++)
  y[i]=x[i]*(i+1);
```
Errors Classification and Static Detection Techniques for Dual-Programming Model

Listing 3. The system as a whole becomes stuck as a result of an OpenMP error, which occurs when the same variable is set in one region and released in another.

```c
1 omp_lock_t lock_a;
2 omp_init_lock(&lock_a);
3 #pragma omp sections
4 {
5     #pragma omp section
6         { omp_set_lock(&lock_a); /*-----Do Work-----*/
7             }
8 #pragma omp section
9     { omp_unset_lock(&lock_a);
10         }
11 #pragma acc kernels
12 for (i=0; i<N; i++)
13     y[i]=x[i]*(i+1);
14 }
```

program deadlock. Furthermore, when using OpenMP, one of the possible reasons of deadlock is the destruction of the lock without initializing it first. However, unset operations cannot be carried out without first initializing and setting the variable. Additionally, this causes a system deadlock.

When using locks, it is not possible to lock or unlock a variable inside a conditional statement (such as ‘if’, ‘else’, or ‘switch case’) unless those statements are combined in the same conditional statement. A system is vulnerable to a deadlock if a lock is created or released without them being together. In Listing 4, a possible deadlock may occur due to an OpenMP logical error that affects the system as a whole. If the OpenMP lock_a is set in the first section but the thread does not enter the ‘if’ statement in line 10, then the threads operating in the second round will be forced to wait indefinitely for the lock to be released by the first section. It is impossible to predict which threads may interact with the if statement, which means that the lock will never be released.

In ‘if’ or ‘else’ statements, it is not possible to set or unset the lock. If they are required, the Set_lock and Unset_lock functions should be combined together in an ‘if’ statement or an ‘else’ statement.

In Listing 5, suppose there are two threads that are independently occupying each section. In this example, there are two ‘Sections’, each with ‘locks’ on the same variables. These two threads each operate a ‘lock’ and wait for each other to receive the second ‘lock’. The first thread sets a lock for the variable ‘lock_a’ and the second thread sets a lock for the variable ‘lock_b’. However, although the first thread requires the lock ‘lock_b’ to finish the task, the second thread has already obtained the lock through its own threading operations. While this is going on, the second thread is also waiting for the function ‘lock_a’ to complete its task. In a blocked state, each of them awaits the other thread to release its ‘lock’ on the variable before proceeding with the action. It is possible to escape this problem by simply altering the order of locks in the second section. For example, if ‘lock_a’ is locked first before ‘lock_b’, both threads will attempt to lock ‘lock_a’ if it is not already locked. Someone threads will arrive and pick the lock and complete the task. The second thread will not lock ‘lock_b’ until the first thread has finished and released the lock, at which point the other thread will wait and deadlocks will be avoided.

Critical directives with the same name are restricted from nesting inside of each other in OpenMP. The nested critical region, as indicated in Listing 6, is another possible cause risk of deadlocks. By appending a name to the crucial region, such as ‘#pragma omp critical region1’, the critical region may be identified. A deadlock does not occur in the event that there is another vital area with a different name. The code deadlocks if the names of the nested crucial areas are the same. In addition, when critical regions that are nested together do not have a name, the system freezes and treats them as if they all have the same name.

In OpenMP, the ‘barrier’ directive must be accessible by all threads in order to avoid deadlock. This is why programmers should avoid using the barrier directive in ‘if’, ‘else’, and ‘if
Listing 5. Incorrect placement of the OpenMP lock leads to a deadlock in this example. It is the same variable ‘lock_a’ that has been locked in the other section. Each thread runs a section, and thread 1 locks the variable ‘lock_a’ and the other thread locks the variable ‘lock_b’; both threads wait for the other to release the lock for an indefinite amount of time, which results in a deadlock.

Listing 6. In this example, there are two critical zones that overlap with each other. As a result of this issue, which was caused by the abuse of OpenMP directives, the system as a whole has been brought to a deadlock.

Listing 7. An OpenMP barrier directive inside for loop leads to deadlock in the whole system.

Listing 8. In OpenMP, using the master directive inside a parallel region for initializing variables lead to deadlock or data race depending on the thread execution order.

else’ clauses as well as in ‘master’ and ‘ordered’ ‘sections’ and ‘critical’ and ‘single’ regions, because not all threads can enter these areas. Listing 7 is an example of a deadlock that might result from the use of a ‘barrier’ inside ‘for’. Only one thread is allowed to enter a ‘single’ zone. In a ‘critical’ region, only one thread can enter at a time.

The ‘master’ directive in OpenMP restricts access to just the master thread. In this situation the data cannot be synchronized. The remainder of the code is run by other threads, and all other threads bypass the master area entirely. It is possible that other threads will begin executing the rest of the code before the master thread arrives and begins initializing the variables, thus resulting in a deadlock. This is also likely to result in a data race owing to a lack of synchronization. Listing 8 demonstrates a variable x that was defined before the parallelism logic in line 1 but was not initialized. Initialization is intended to take place either before or inside the parallel zone or through adding barriers after the critical region to force all thread to wait until all threads arrive.

The ‘single’ directive in OpenMP is similar to the ‘master’ directive in that only one thread of the threads team is responsible for running this area. However, this thread is not necessarily the master thread; with the remaining threads skipping it in order to complete the remainder of the work. In order to avoid deadlock, the ‘master’ directive must not be used inside the ‘single’ construct. The thread that enters the ‘single’ area may or may not be the master thread, depending on how the region is constructed. The thread that enters the ‘single’ region must be the ‘master’ thread in this example; otherwise, if it is not the master thread, the thread will not enter the master region and the system will wait indefinitely.
Because the master region is contained within a single region, there is a possibility of a system-wide deadlock. The master region can only be accessed by the master thread, whereas the single region may be accessed by any singular thread. In the event that a thread accesses a single region but is not the master thread, the master region will never be executed.

Listing 9. Because the master region is contained within a single region, there is a possibility of a system-wide deadlock. The master region can only be accessed by the master thread, whereas the single region may be accessed by any singular thread. In the event that a thread accesses a single region but is not the master thread, the master region will never be executed.

for the master thread to execute the master region. In Listing 9, the potential for a system deadlock emerges as a result of the inclusion of the ‘master’ directive within a ‘single’ directive. In contrast, a ‘single’ region can be accessed by just one thread, and it does not have to be the master thread, but the ‘master’ region can only be accessed by the master thread. If a thread other than the master thread enters the ‘single’ region and then the master region will never be performed. A deadlock will occur if another thread skips the master region, and the variable ‘percent’ is not updated since it is inside the ‘master’ region. However, if the data had already been initialized before the parallel region began, the results of the operations will not be valid since the right data is meant to be computed in the ‘master’ region before the operations could be performed.

The majority of computer programs make use of loops of various forms, including ‘for’, ‘while’, and ‘do while loops’. In both OpenMP and OpenACC, loops are used to continuously execute a series of statements that can be distributed across several threads at the same time. Both approaches allow these loops to be conducted in parallel. The OpenMP programming paradigm prohibits the nesting of loops, especially when they are in the same parallel region. This can be found in the OpenMP programming paradigm specification, where the outcomes are incorrect due to race conditions, as demonstrated in Listing 10 and Listing 11. The same issue was repeated in both examples, where the variable ‘i’ in the first loop is considered private by default, while the variable ‘j’ in the second loop is not considered private by default, which result in a race condition. Defining the variable ‘j’ as private is one of the solutions that may be used to avoid the race condition in this situation. Another option is to assign each loop to a distinct parallel zone.

Listing 10. A race condition occurred in the system as a result of the nested for loops in lines 3 and 6 in OpenMP, as both loops belong to the same parallel region in line 1. The variable ‘j’ is not private by default, but the variable ‘i’ is private by default.

race condition in this situation. Another option is to assign each loop to a distinct parallel zone.

When a for loop is nested within another for loop, it is considered to be in a different parallel region. Each for loop in the example given in Listing 12 corresponds to a distinct parallel area, as can be seen in the code below. Because each loop is associated with a distinct parallel zone, the variables ‘i’ and ‘j’ are private; nevertheless, the internal variable ‘sum’ must also be private in order to function correctly. In this example, the variable ‘sum’ is not private. In this situation, a race condition may occur, and in order to resolve it, the variable ‘sum’ can be included in the parallel operation using reduce (+: sum). Another source of race conditions is the data dependence, which is illustrated in Listing 13. The repetitions of the loop in this example are fully reliant on each other. The index of ‘a’ on the left side differs from the index of ‘a’ on the right side in line 6. More solutions to this problem can be found in our research [30].

The parallel region of OpenACC cannot be nested within the parallel region of OpenMP. In this circumstance, a deadlock will develop, as seen in Listing 14. In making some progress, the problem cannot be avoided even when the ‘kernel’ directive is used instead of the ‘parallel’ one. When
Listing 12. Data race developed in this scenario because the variable ‘sum’ in OpenMP cannot be used in parallel, and in particular because it is read from and written to by more than one thread at the same time. This is an exception. In order to overcome this problem, the expression `reduction(+: sum)` is included in the first line.

```c
1 $\#pragma omp parallel$
2 \{ 
3 $\#pragma omp for$
4 \{ 
5 \text{for} (i=0;i<N;i++) 
6 \} 
7 $\#pragma omp parallel for$
8 \{ 
9 \text{for} (j=0;j<N;j++) 
10 \} 
11 $\#pragma acc kernels$
12 \text{for} (i=0;i<N;i++) 
13 \text{sum}+=i*j;
14 \} 
15 \text{percent}=(\text{sum}/\text{total})*100;
16 \}
```

Listing 13. A data race developed in this scenario because of data dependency in the OpenMP portion. The value of a in index i depends on the next index i+1, which leads to a data race.

```c
1 $\#pragma omp parallel$
2 \{ 
3 $\#pragma omp for$
4 \{ 
5 \text{for} (i=0;i<N;i++) 
6 \} 
7 \text{a}[i]=\text{a}[i+1]*\text{b}[j];
8 $\#pragma acc kernels$
9 \text{for} (i=0;i<N;i++) 
10 \text{c}[i]=\text{a}[i]+\text{b}[i];
11 \}
```

Listing 14. It is not possible to nest parallel OpenACC for loops within a parallel OpenMP section. As a result, the whole system comes to a deadlock.

```c
1 $\#pragma omp parallel$
2 \{ 
3 $\#pragma omp single$
4 \{ 
5 $\#pragma acc parallel loop$
6 \{ 
7 \text{for} (i=0;i<N;i++) 
8 \text{k}[i]=\text{sinf}(i);
9 \}
10 \}
11 \}
```

Listing 15. Entering the OpenACC parallel code within the single does not completely guarantee success. Although the deadlock has been broken, the findings are still incorrect.

```c
1 \text{int sum}=0, i=0, \text{total}= 3459;
2 \text{float percent;}
3 $\#pragma omp parallel$
4 \{ 
5 $\#pragma omp for$ \text{reduction}(+: \text{sum})
6 \{ 
7 \text{for} (i=0;i<10;i++) 
8 \text{sum}+=i;
9 $\#pragma omp single$
10 \{ 
11 \text{printf("sum is %d\n", sum);}
12 $\#pragma omp master$
13 \{ 
14 \text{percent}=\text{sum}/\text{total}*100;
15 \text{printf("percent is %d\n", percent);}
16 \}
17 $\#pragma acc kernels$
18 \{ 
19 \text{for} (i=0;i<N;i++) 
20 \text{a}[i]+=\text{a}[i]*\text{percent;}
21 \}
22 \}
```

Listing 16. The code shows an OpenMP error where a master region within a single region can cause a deadlock. The OpenACC code has a data race. This part of the program has a deadlock, and if it does not, the data has a race.

entering the parallel OpenACC code inside the ‘single’, ‘atomic’, ‘critical’, or ‘master’ OpenMP regions, the programmer can prevent a deadlock situation. In this scenario, only one thread can be running at a time. As a result, because the outcome of this computation is incorrect, it does not completely address the issue. An example of this situation can be seen in code in Listing 15. The OpenACC code is included within a parallel code. No deadlock exists, yet the outcome is incorrect. When creating a parallel area in OpenMP, the inclusion of the OpenACC region without any parallel routing must be ensured.

An OpenMP error is displayed in Listing 16, and it indicates that there is a possibility of a deadlock due to the interference of a ‘master’ within a ‘single’ region in this problem, as explained in Listing 9. In addition, there is a data race in the OpenACC section because of data dependency in line 14. The software encounters a deadlock in this example depending on inputs and the running environment, and if the deadlock does not occur, a data race will occur. According to the error dependency graph presented in Figure 1, Figure 2, and Figure 3, a deadlock occurs in the whole system.

According to Listing 17, another test was performed by placing the OpenACC structured data area code in a single and critical region. As a consequence, the system as a whole came to a deadlock, despite the fact that all of the codes were accurate in both programming models. Additionally, this was one of the problems that was discovered throughout the merging process. As a result, overlapping a parallel OpenACC structured data region inside an OpenMP region which is run by only one thread will not succeed. Since the master needs only the master thread to be executed, but when the thread that enters the single region is not the master, the master will not run for ever.

Consider the following example in Listing 18, in line 2 there is a while loop. If the value of ‘x’ is greater than 5, the program will be unable to exit the while loop until the value of
Despite the utilization of the structured data from OpenACC to overcome the deadlock issue inside a single region of OpenMP, stalemate was still encountered. Consequently, it is not possible to include an OpenACC structured data area inside a single region of the OpenMP structured data.

Listing 18. A livelock in line 2 results in a deadlock in the whole system. The value of array A is not updated inside the while loop.

```c
#pragma omp parallel for nowait
while (x < 5) {
    A[i] = x * 0.2;
    if (i < N)
        i++;
}
```

Listing 17. Despite the utilization of the structured data from OpenACC to overcome the deadlock issue inside a single region of OpenMP, stalemate was still encountered. Consequently, it is not possible to include an OpenACC structured data area inside a single region of the OpenMP structured data.

```c
#pragma omp parallel
{ 
    #pragma acc enter data copyin(a[0:n])
    #pragma acc parallel loop
    for (i = 0; i < N; i++)
        k[i] = sinf(a[i]) + cosf(a[i]);
    #pragma acc exit data copyout(k[0:n])
}
```

‘x’ is less than 5. At the same time, another error is discovered in the system: a data race in line 8. This circumstance results in a deadlock. These errors cause a deadlock in the whole system as the value of array ‘A’ is not updated while the while loop is running. Additionally, array ‘C’ requires array ‘A’ to complete the calculation and update the array ‘C’ values at the same time. A data race bug in OpenACC combined with a livelock in OpenMP results in a deadlock in the entire system. The errors identified in the present study are classified and listed in the next section.

VI. ERROR CLASSIFICATION
A classification of errors caused by the integration between OpenACC and OpenMP is provided in this section. The most recent documentation is examined in the OpenACC 3.2 [22] and OpenMP 5.2 [58] programming models. Additionally, a number of experiments have been undertaken. As previously stated in the earlier sections, a number of runtime errors were discovered based on experience. Among the documents are some guidelines that explain how to use each programming model to achieve parallelism. Some runtime errors can result in serious problem during the execution of the program. Although runtime errors are not detected by the compiler, there is a need for a testing tool to identify these runtime errors. As explained in the previous section, merging OpenACC with OpenMP results in new errors. If each model was programmed separately, these errors would not have manifested. There are three degrees of mistakes that may be distinguished. At the first level, there are no errors in both models, as seen in Listing 14, 15 and 16, which may or may not result in errors depending on the application. When there is an error in one of the two programming models (see Listing 1 - 13), the second level is triggered. It is possible to reach the third level if there are faults in both programming models (see Listing 17). The following categorization graphics have been created in accordance with the results of the tests conducted in the preceding section.

Figure 1 and Figure 2 show the absence of errors at the first level in the case of both programming models being written correctly, and the errors at the second level when there is an error in one of the two models. In the event that there is no error in any model, there are two possibilities. The first is that the system as a whole is working properly, which is the goal. Another possibility is that there is a deadlock in the system due to an overlap of the two models, as shown in the list. Figure 1 emphasizes the defects in the system, which incorporates OpenACC and OpenMP. The faults that resulted were caused by OpenMP issues, and the system did not have any errors in the OpenACC area of the code. It was found that when an OpenMP race condition occurs, the system as a whole is placed in a race condition, and the estimated output will be incorrect. Additionally, there is a potential deadlock that could result in a system deadlock depending on thread orders. In the case of a livelock, the system may freeze or it may continue with the livelock.

The errors in the system caused by OpenACC errors are the primary focus of Figure 2. The systems in this case did not have any errors in the OpenMP regions. It was found that if there is a race condition in OpenACC, the system as a whole is placed in a race condition, resulting in the estimated output being incorrect. Furthermore, in some cases, the system as a
A deadlock in OpenACC codes results in the system as a whole being in a deadlock. In the case of a livelock, the system may freeze or it may continue in the livelock state. Figure 3 focuses on runtime errors in the system that are produced by defects in both OpenMP and OpenACC when they are combined. It is probable that mistakes in the results will occur if there is a race situation in both OpenMP and OpenACC, which will result in a deadlock in the whole system at some point. In the event that one of the models experiences a race situation while the other experiences a livelock, the system will be suspended. In the case that one of the programming models becomes stuck in a deadlock, the system as a whole is put into a state of suspension. The system is halted if there is a livelock in either of the two models.

VII. STATIC TESTING APPROACH IMPLEMENTATION

It is very difficult to test parallel systems due to the challenges that arise during the implementation process due to a variety of circumstances, such as complex situations and the nature of the applications being tested. As a result of these considerations, more work is spent developing the test tool in order to cover all possible scenarios for the test cases and data. A static test tool was developed in this study to detect errors that occur in programs written in C++, including the OpenACC and OpenMP programming models. This tool also takes into account the dependency error described in Figure 1, Figure 2, and Figure 3. Our previous study [30] included a discussion of the proposed tool design. This tool is able to identify problems in the source code before the code is run, log the problem details, and display them in a separate log file that contains all the information to help the programmer locate the errors. All of this is done before the program is compiled. This makes it easier for the programmer to understand the problem and fix it before the system is actually put into place. Our tool also gives a summary of the runtime error based on the dependency graph discussed in the previous section.

A. ANALYSIS PHASE

The C++ programming language was used to create our tool. Initially, the source code to be tested was entered into C or C++ programming environments. One or both of these programming modules (OpenMP and OpenACC) were included in this source code. The code that was going to be tested was analyzed by our tool, and the data that were be used for runtime error detection was extracted. The following information was gathered by our tool:

- OpenMP and OpenACC ‘process’ data collection and storage in vectors.
- Information about OpenMP and OpenACC regions such as start line number and end line number for each regions, and save them in structured data.
- Gather information about explicit barriers in the OpenMP.
- Information about the explicit and implicit end lines of OpenACC compute region.
- Initial and final values of the variables in all parallel regions in both programming models.
- The equations and the variables they contain are preserved whether read or write.
- The ‘for’, ‘while’, and ‘do while’ loops data are collected, along with the dependent variables and the initial and final values of these loops.
- Information about OpenMP locks, detailing their beginning and ending in lines, and whether they were configured or not, in order to detect the deadlock that causes them.

All this information is saved in a log file and presented to the programmer in an organized manner. For example, Figure 4 shows a log file for data that was gathered from a program in test. The first section summarizes the number and type of regions in the source code being examined. The second part depicts the start and end of each region.

Figure 5 shows an example of data that is collected from test code that combines OpenMP and OpenACC. This example illustrates a set of equations and their dependent variables, as well as their type, and indicates whether they are an array or not. Additionally, the status of the variable is determined whether it is now reading ‘R’ or writing ‘W’. In addition, the kind of operation, such as ‘greater than’, ‘less than’, or ‘equal to’, is shown in the table. The value that has been increased and the loop that it is a part of are both given.

Figure 6 is also present in part of the log file that was reported to the programmer. In this particular example, the OpenMP ‘lock’ parameters are extracted from the source
code. It is essential to report the start line number and end line number for each ‘lock’, as well as their interactions, in order to identify the causes of any deadlocks that may occur. The information includes the name of the variable that is being locked and details about the type of ‘lock’, such as whether it is nested or single and if this ‘lock’ was initialized before any process and destroyed after it was finished. This information allows for the identification of runtime issues related to deadlock and livelock. In the second part, a summary of the variables that are present in each of the OpenMP regions is presented. This helps in locating race conditions and gathering data on races. The table lists the variables and the areas to which they belong, and indicates whether or not the variables are an array or a simple variable. Additionally, the number of the equation to which this variable belongs to is listed as well, along with whether or not variable is included inside a loop. In addition, the presence of any dependencies inside the array may cause a race condition. The final section provides information regarding loops, including the start line and end line of the loops, the initial and final values of any variables contained within the loops, and the operation for incremental changes.

In the next section, the data and the constructed algorithms are used to record the report and present it to the programmer with the errors that were discovered.

### B. DETECTION PHASE

This subsection will discuss algorithms for detecting runtime errors that are associated with either the OpenMP programming model only or both the OpenMP and the OpenACC programming models together. Regarding the algorithms related to the OpenACC programming model, refer to reference [5], where the same algorithms were used. Parallel programming necessitates numerous conditions that some programmers may ignore, particularly the interventions between various regions, such as the critical and single regions, the sections, shared variables, the reduction clause, and many other conditions. To ensure that there are no issues encountered during use, each of these must be examined carefully by tool developed in the present research. Obviously, these runtime
errors are not detectable in the compilation stage, which results in trouble further down the pipeline. Depending on the significance of the program, the consequences may be severe. We are eager to find these errors and present them in a manner that is clear to the programmer.

Algorithm 1 is capable of determining whether or not a lock has resulted in a deadlock. If the lock does exist, it will be tested inside the ‘for loop’, and if it is set, it will either be ‘unset’ or ‘destroyed’. If the lock is not destroyed, the loop will eventually lead to a deadlock. This is because it is essential to make sure that the lock is closed and that the process is terminated once it is finished. When a loop is being used, the lock will be re-requested at the beginning of each cycle. Different causes of loop deadlock have been collected. This algorithm documents these occurrences and writes the error message to the log file.

Additionally, algorithm 2 investigates the deadlock produced by a lock. The algorithm investigates two successive locks for the same variable, but it leaves the first lock without release before it is used again. It also determines if there is no lock initialization, which is a violation of standards that may result in a deadlock situation. Additionally, prior to the end of the program, each lock must be unset and destroyed once it has served its purpose. This code ensures that the simple locks do not overlap each other in any way. It is necessary to finish the currently active lock on a variable before beginning to set a new lock on the same variable. Because of the data that had previously gathered, line numbers can tell us where each lock starts and ends. The data related to locking is stored in a table in the same sequence as it occurs in the line numbers. Each piece of lock data is located in the same row as the lock itself, including the line of the lock. To identify any potential interference, the algorithm compares each lock to the lock that comes immediately after it. After that, the information that is gathered is recorded in the log file, as shown by the messages that are printed out.

Additionally, the insertion of the lock in the OpenMP parallel region and the unset of the lock in another region is one of the reasons why the system deadlocks. This condition is revealed in algorithm 3. According to the data gathered about the locks, we can know in which region the lock was established and in which region it was terminated. In this algorithm, each lock and its starting line are examined, and the region that each lock is associated with is explored. It is also possible to discover the line number of locks destroyed. In the event that lock set and destroy line numbers belong to the region that each lock is associated with is explored. It is also possible to discover the line number of locks destroyed. In the event that lock set and destroy line numbers belong to a distinct region, then a deadlock will occur and this error will be 'unset' or 'destroyed'. If the lock is not destroyed, the loop will eventually lead to a deadlock. This is because it is essential to make sure that the lock is closed and that the process is terminated once it is finished. When a loop is being used, the lock will be re-requested at the beginning of each cycle. Different causes of loop deadlock have been collected. This algorithm documents these occurrences and writes the error message to the log file.

Algorithm 4 searches all of the equations for dependencies and data dependencies, and it also finds the race conditions that are caused by data dependency. This issue frequently arises in situations where the same equation is applied to the same array, each of which has a different index. Algorithm 5 also reveals the cases of racing caused by writing after writing, reading after writing, and writing after reading, as explained in Section 5. The previous information collected allows for the determination of each variable; whether in the case of writing or reading,
and regardless of the parallel region, this variable is checked along with the cases in which it is used. The variable that is to the left of the ‘equal sign’ is referred to as the writing variable, while the variables that are to the right of the equal sign are referred to as the reading variables. It is possible to confirm the presence of race conditions that take place as a result of the utilization of multiple threads that are used during execution. The causes of livelocking that might occur in loops are investigated by algorithm 6. In either of the two programming approaches, a livelock might be caused by the absence of a condition or when it is True or False forever. In this case, the loop will not stop. It is also possible for the system to freeze if the condition is a constant number. Consequently, the data is gathered and then recorded into the log file for the programmer to examine. A list of the circumstances that lead to a livelock has been created, and source code has been inspected in an effort to find faults. In the event that it is present, the livelock is accurate will result in a runtime error. However, if the condition is not obvious, a warning message is printed so that the programmer can review the condition to avoid a livelock.

Algorithm 4 Read-Write Race Condition Detection Algorithm

1: SEARCH for equation in OpenMP Parallel Region
2: if (Equation1\_d1 = Equation1\_d2 Equation1\_a1 = Equation1\_a2)
3: if (Equation1\_a1,tatus = Read Equation1\_a2,tatus = Write )
4: "Display error message Read after Write Race Condition"
5: else if (Equation1\_a1,tatus = Write Equation1\_a2,tatus = Read )
6: "Display error message Write after Read Race Condition"
7: else if (Equation1\_a1,tatus = Write Equation1\_a2,tatus = Write )
8: "Display error message Write after Write Race Condition"
9: end if

Algorithm 5 Detect Race Condition Caused by Data Dependency

1: SEARCH for Loop Inside OpenMP Region
2: SEARCH for Equation inside For Loop
3: if (equation1\_region = equation2\_region)
4: if (equation1\_variable\_name = equation2\_variable\_name)
5: if (equation1\_variable\_type =array equation2\_variable\_type =array)
6: if (equation1\_variable\_index =equation2\_variable\_index)
7: "Display Error Message Data Race Due to Data Dependency"

Algorithm 6 Livelock Detection Algorithm

1: SEARCH for (While Loop)
2: if value = True
3: "Display message livelock"
4: else if (value = number)
5: "Display message caution be aware potential Livelock"
6: SEARCH for (For Loop)
7: if (parameter(); )
8: "Display message Livelock"
9: else if (ConditionVariable = number)
10: "Display message caution be aware potential Livelock"
11: else if (IncrementalVariableisempty)
12: "Display message caution be aware potential Livelock"

Algorithm 7 Detect Deadlock Caused by Barrier Directive

1: SEARCH for Barrier Inside OpenMP Region
2: if (critical1\_tartine,o < barrier1\_tartine,o < critical1\_ndine,o)
3: "Display message Deadlock"
4: else if (single1\_tartine,o < barrier1\_tartine,o < single1\_ndine,o)
5: "Display message Deadlock"
6: else if (master1\_tartine,o < barrier1\_tartine,o < master1\_ndine,o)
7: "Display message Deadlock"

Algorithm 8 This Algorithm Find Overlaps Between OpenACC and OpenMP Regions

1: SEARCH For OpenACC Inside OpenMP Parallel Region
2: SEARCH For OpenACC Parallel region
3: if (OpenACC\_parallelregion Not Inside OpenACC\_singleregion OR Not Inside OpenACC\_criticalregion)
4: "Display Error Message Deadlock Could Not Insert OpenACC Parallel Region Inside OpenMP Parallel Region"
5: SEARCH For OpenMP Inside OpenACC Parallel Region
6: "Display Error Message Deadlock Could Not Insert OpenMP Parallel Region Inside OpenACC Parallel Region"

The algorithm that examines a barrier and states that cause a system to deadlock is expressed by 7. As was previously indicated, the barrier’s placement is crucial since all of all threads must pass through it for the task to be finished. The barrier cannot be positioned in regions such as the critical and single region that only permit specific threads to enter them, as verified by the algorithm. Overlaps between regions in the OpenACC and OpenMP programming paradigms might result in unexpected behavior. The existence of a parallel section of the OpenACC model within an OpenMP parallel region immediately creates a
deadlock in the system. During the course of multiple experiments run between these two models, this issue presented itself. If the region is not parallel, it can function properly. OpenACC parallel regions can be entered if they are contained inside a ‘single’ region or ‘critical’ area in OpenMP. In addition, one of the requirements for using the accelerator is to exclude the OpenMP parallel zone from OpenACC. Algorithm 8, detects this interference. The line numbers that appear for the beginning and end of each region can show overlaps, in addition to the kind, which indicates whether or not the region is parallel.

Algorithm 9 Dependency Error Between OpenMP and OpenACC
1: if \( \text{omp4deadlock}_\text{count} \geq 1 \)
2: "Display message: According to Error Dependency Graph, there is Deadlock in the whole System Because there is Deadlock in OpenMP"
3: else if \( \text{acc4deadlock}_\text{count} \geq 1 \)
4: "Display message: According to Error Dependency Graph, there is Deadlock in the whole System Because there is Deadlock in OpenACC"
5: else if \( \text{mpi4deadlock}_\text{count} \geq 1 \)
6: "Display message: According to Error Dependency Graph, there is Deadlock in the whole System Because there is Deadlock in OpenACC"
7: if \( \text{omp4livelock}_\text{count} \geq 1 \)
8: "Display message: According to Error Dependency Graph, there is Deadlock in the whole System Because there is Livelock in OpenMP"
9: else if \( \text{acc4livelock}_\text{count} \geq 1 \)
10: "Display message: According to Error Dependency Graph, there is Deadlock in the whole System Because there is Livelock in OpenACC"
11: else if \( \text{mpi4livelock}_\text{count} \geq 1 \)
12: "Display message: According to Error Dependency Graph, there is Deadlock in the whole System Because there is Livelock in OpenACC"
13: else if \( \text{omp4ac4count} \geq 1 \) AND \( \text{acc4ac4count} \geq 1 \)
14: "Display message: According to Error Dependency Graph, there is Race Condition and Potential Deadlock in the whole System Because there is Race Condition in OpenACC and OpenMP"
15: else if \( \text{omp4ac4count} \geq 1 \) AND \( \text{mpiall44count} \geq 1 \)
16: "Display message: According to Error Dependency Graph, there is Race Condition and Potential Deadlock in the whole System Because there is Race Condition in MPI and OpenMP"
17: else if \( \text{acc4ac4count} \geq 1 \) AND \( \text{mpiall44count} \geq 1 \)
18: "Display message: According to Error Dependency Graph, there is Race Condition and Potential Deadlock in the whole System Because there is Race Condition in OpenACC and MPI"
19: else if \( \text{omp4ac4count} \geq 1 \)
20: "Display message: According to Error Dependency Graph, there is Race Condition and Potential Deadlock in the whole System Because there is Race Condition in OpenMP"
21: else if \( \text{acc4ac4count} \geq 1 \)
22: "Display message: According to Error Dependency Graph, there is Race Condition and Potential Deadlock in the whole System Because there is Race Condition in OpenACC"
23: else if \( \text{mpiall44count} \geq 1 \)
24: "Display message: According to Error Dependency Graph, there is Race Condition and Potential Deadlock in the whole System Because there is Race Condition in MPI"

Once the analysis of the source code is complete and a list of the runtime errors of each programming model is generated, the issue with the system can be determined based on the error dependency graph, which was proposed in Section 1. According to the dependency graph, the occurrence of a deadlock in one of the two models or a livelock leads to a complete deadlock in the system as a whole. An error dependency scheme has been implemented in algorithm 9. In addition, the occurrence of a race condition or data race in either of the programming models results in the occurrence of a race condition or data race, and the probability of a deadlock is dependent on the running environment and thread behavior.

Additionally, codes for dynamic testing will be added for more tests, particularly those especially with regard to potential errors in future. In order to ensure that there is no instance of the system entering a deadlock, we will count the number of threads that enter the parallel region and check to see that they have all exited the region. In addition to that, these codes that are responsible for dynamic testing will be added to the header file, and they will be embedded alongside the other target codes. Additionally, the dynamic component is responsible for accurately recording all issues and runtime errors, as well as writing a report to the user about the test status of the program.

FIGURE 7. Total time for evaluating the OpenACC static testing approach.

FIGURE 8. Total time for evaluating the OpenMP static testing approach.

VIII. TESTING AND EVALUATION

In this section, the methodology used in the study is evaluated based on a predetermined set of benchmarks. We discuss our experiments and compare our tool to similar tools in terms of the runtime errors it is able to identify. The testing was conducted on a laptop with an 11th Generation Intel® Core™ i7-11800H @ 2.30GHz x 16 CPU, 16GB of main memory, and an NVIDIA GeForce RTX 3060 GPU running Ubuntu.
TABLE 1. Comparison of runtime error handled by OpenMP and OpenACC testing tools. The symbol ✓ means fully detected, P means partially detected and X means cannot be detected.

| Programming Model | Our Tool (Static Approach) | ACC_Test (static approach) | ROMP | LLOV | ARCHER | MPR ACER |
|-------------------|-----------------------------|-----------------------------|------|------|--------|----------|
| Race Condition    |                             |                             |      |      |        |          |
| Loop Parallelization Race | ✓                           | ✓                           |      |      |        |          |
| Data Read and Write Race   | ✓                           | ✓                           |      |      |        |          |
| Nowait clause                | ✓                           | ✓                           |      |      |        |          |
| Shared Clause                | ✓                           | X                           |      |      |        |          |
| Barrier Construct            | ✓                           | X                           |      |      |        |          |
| Atomic Construct            | ✓                           | X                           |      |      |        |          |
| Critical Construct          | ✓                           | X                           |      |      |        |          |
| master Construct            | P                           | X                           |      |      |        |          |
| Single Construct            | P                           | X                           |      |      |        |          |
| SIMD Directives             | X                           | X                           |      |      | ✓      | X        |
| Teams Construct            | X                           | X                           |      |      | ✓      | X        |
| Deadlock                    |                             |                             |      |      |        |          |
| Lock Routines               | ✓                           | X                           |      |      |        |          |
| Section construct           | ✓                           | X                           |      |      |        |          |
| Barrier Construct           | ✓                           | X                           |      |      |        |          |
| Nowait clause               | P                           | X                           |      |      |        |          |
| LiveLock                    | P                           | X                           |      |      |        |          |
| Device Directives           | X                           | X                           |      |      |        |          |
| Data Clause related errors |                             |                             |      |      |        |          |
| Create Clause              | ✓                           | X                           |      |      |        |          |
| Copy Clause                 | ✓                           | X                           |      |      |        |          |
| Copyin Clause               | ✓                           | X                           |      |      |        |          |
| Copyout Clause              | ✓                           | X                           |      |      |        |          |
| Delete Clause               | ✓                           | X                           |      |      |        |          |
| Memory Errors               | ✓                           | X                           |      |      |        |          |
| Data Transmission Errors    | ✓                           | X                           |      |      |        |          |
| Present Clause              | X                           | X                           |      |      |        |          |
| Race Condition              |                             |                             |      |      |        |          |
| Independent Clause Race    | ✓                           | ✓                           | X    |      |        |          |
| Data Read and Write Race    | ✓                           | ✓                           | X    |      |        |          |
| Reduction Clause Race       | ✓                           | ✓                           | X    |      |        |          |
| Loop Parallelization Race   | P                           | P                           | X    |      |        |          |
| Host/Device Synchronization Race | X                     | X                           | X    |      |        |          |
| Asynchronous Directive Race | X                           | X                           | X    |      |        |          |
| Deadlock                    |                             |                             |      |      |        |          |
| Device Deadlock             | ✓                           | ✓                           | X    |      |        |          |
| Host Deadlock               | P                           | P                           | X    |      |        |          |
| LiveLock                    | P                           | P                           | X    |      |        |          |

20.04.4 LTS. In addition, 40 criteria from five distinct benchmark sets were employed, including TORMENT [60], EPCC-OpenACC benchmarks [61] and PolyBench-ACC [62] for testing on OpenACC. DataRaceBench 1.4.0 C++ [63], and NPB-CPP Benchmark 3.3.1 C++ [64] were used for testing OpenMP.

The outcomes of the static testing strategy are displayed in Figures 7 and 8. The x-axis indicates the codes that have been tested based on the selected benchmarks, the y-axis on the left reflects the number of lines in each code, and the y-axis on the right represents the amount of time spent during the test using the tool we developed. Figure 8 also displays the amount of time required to test these codes based on two distinct benchmarks, as well as the number of lines that were examined. The unit of time was Clocked per Second.

Figure 9 is an example of the log file is presented; it demonstrates errors that can be reported during testing but cannot be detected by the compiler. This screenshot is taken from the file of one of the programs tested by the developed tool.

Figure 10 displays the errors that were found in one of the codes during the test. The final summary highlights the primary issue that arises from integrating these programming models. To illustrate the error that occurs in the system, the provided tool displays these errors at the end of the user’s report. Potential errors can be verified in the dynamic stage, which will be established in future work.
The proposed tool is the first tool of its sort to test C++ programs with the OpenACC and OpenMP programming paradigms. Therefore, we decided to compare it to similar tools that employ static testing methodologies to examine the OpenACC or OpenMP programming models for runtime errors. Table 1 displays a collection of runtime testing tools. It was found that our tool was the only one capable of detecting faults in either of the programming models. This tool finds runtime defects in C++ programs.

IX. CONCLUSION AND FUTURE WORK

During the course of this study, we developed a tool that is capable of identifying runtime errors in C++ programs that are caused by race conditions, data races, deadlocks, and livelocks that are based on the OpenACC and OpenMP programming models. In the process of this research, static testing techniques were used to detect errors. Using this method, errors can be found before the actual running of the code being tested. The developed tool is capable of identifying faults in either of the programming models. This tool also implements an error dependency graph for runtime errors in OpenMP and OpenACC. In addition to this, a number of algorithms designed to track these errors were reviewed. The research also proposed a classification for runtime errors that arise upon merging OpenMP and OpenACC programming models. Additionally, the functionality of the proposed tool was evaluated based on a collection of benchmarks written in C and C++. According to the results, the tool covers a greater number of faults when compared to other tools by a significant margin. By contrasting the tool with other tools and determining their capabilities, the developed tool was found to be the only one of its kind that could detect deadlock faults in OpenMP. To our knowledge, it is the only tool that can identify errors in the integration that takes place between OpenACC and OpenMP. This will contribute to increasing the system’s reliability and guarantee that there will be no faults in high-quality systems. The test run by our tool will not have any impact on the program’s core code or its execution speed; rather, it will display a list of faults for the programmer to correct.

In future work, we would like to extend our tool by supporting a greater number of OpenMP directives and constructs to detect more causes of runtime errors, some of which are not currently supported by our tool. Moreover, we intend to enhance our tool by transforming it into a hybrid version, which will give us the opportunity to recognize runtime issues during runtime. The dynamic component is responsible for achieving this goal.

ACKNOWLEDGMENT

The authors would like to thank DSR for technical and financial support.

REFERENCES

[1] M. Zahran, “The future of high-performance computing,” in Proc. 17th Int. Comput. Eng. Conf. (ICENCE), Cairo, Egypt, Dec. 2021, pp. 129–134, doi: 10.1109/ICENCE49852.2021.9698918.
[2] J. M. Shalf and R. Leland, “Computing beyond Moore’s law,” Computer, vol. 48, no. 12, pp. 14–23, Dec. 2015, doi: 10.1109/MC.2015.374.
[3] J. Shalf, “HPC interconnects at the end of Moore’s law,” in Proc. Opt. Fiber Commun. Conf. (OFC), San Diego, CA, USA, 2019, pp. 1–3, doi: 10.1364/OFC.2019.Th3A.1.
[4] D. Milojicic, P. Faraboschi, N. Dube, and D. Roweth, “Future of HPC: Diversifying heterogeneity,” in Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE), Feb. 2021, pp. 276–281, doi: 10.23919/DATE51398.2021.9474063.
[5] 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.
[6] A. S. A. Alghamdi, A. M. Alghamdi, F. E. Eassa, and M. A. Khemakhem, “ACC-TEST: Hybrid testing techniques for MPI-based programs,” IEEE Access, vol. 8, pp. 91488–91500, 2020, doi: 10.1109/ACCESS.2020.2994172.
[7] K. Hamidouche, A. A. Awan, A. Venkatesh, and D. K. Panda, “CUDA M3: Designing efficient CUDA managed memory-aware MPI by exploiting GDR and IPC,” in Proc. IEEE 23rd Int. Conf. High Perform. Comput. (HiPC), Dec. 2016, pp. 52–61, doi: 10.1109/HiPC.2016.016.
[8] S. R. M. Rostami and M. Ghaffari-Miab, “Finite difference generated transient potentials of open-layered media by parallel computing using OpenMP, MPI, OpenACC, and CUDA,” IEEE Trans. Antennas Propag., vol. 67, no. 10, pp. 6541–6550, Oct. 2019, doi: 10.1109/TAP.2019.2920253.
[9] Z. Hamid Reza, M. Naoya, S. Aaron, M. Motohiko, and M. Satoshi, “Evaluating and optimizing OpenCL kernels for high performance computing with FPGAs,” in Proc. Int. Conf. High Perform. Comput. Netw., Storage Anal., Nov. 2016, pp. 409–420, doi: 10.1109/SC.2016.34.
[10] M. Aldinucci, V. Cesare, I. Colonelli, A. R. Martinelli, G. Mittone, B. Cantalupo, C. Cavazzoni, and M. Drocco, “Practical parallelization of scientific applications with OpenMP, OpenACC and MPI,” J. Parallel Distrib. Comput., vol. 157, pp. 13–29, Nov. 2021.
[11] S. Blair, C. Albing, A. Grund, and A. Jocksch, “Accelerating an MPI lattice Boltzmann code using OpenACC,” in Proc. 2nd Workshop Accel. Program. Directives (WACCPD), 2015, pp. 1–9.
[12] J. A. Herdman, W. P. Gaudin, O. Perks, D. A. Beckingsale, A. C. Mallinson, and S. A. Jarvis, “Achieving portability and performance through OpenACC,” in Proc. 1st Workshop Accel. Program. Directives, Nov. 2014, pp. 19–26.
[13] F. L. Cabrall, C. Osthoff, M. Kischinhevsky, and D. Brandao, “Hybrid MPI/OpenMP/OpenACC implementations for the solution of convection-diffusion equations with the HOPMOC method,” in Proc. 14th Int. Conf. Comput. Sci. Appl., Jun. 2014, pp. 196–199.
[14] B. Swain, Y. Li, P. Liu, I. Laguna, G. Georgakoudis, and J. Huang, “OMP Racer: A scalable and precise static race detector for OpenMP programs,” in Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal., Nov. 2020, pp. 1–14.
What is OpenACC?

OpenACC is a high-level Directives-based programming model for accelerators. It provides a set of compiler directives that allow developers to explicitly offload sections of their code to accelerators, such as GPUs, without the need for deep knowledge of the underlying hardware or manual memory management. OpenACC directives are inserted into existing application source code, allowing for the acceleration of compute-intensive kernels while maintaining the same control structures and data access patterns as in the original sequential code.

The OpenACC Application Programming Interface (API) was introduced in 2012 and has since evolved to support multi-platform programming, enabling developers to write portable code that can run on a variety of hardware architectures. OpenACC is designed to work seamlessly with other programming models like OpenMP and MPI, making it a powerful tool for creating hybrid applications that leverage the strengths of different computing models.

OpenACC is widely used in scientific computing, machine learning, and other domains requiring high-performance computing. Its simplicity and ease of use make it an attractive choice for developers looking to accelerate their applications without the complexity of low-level APIs like CUDA or OpenCL.
S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos, “The EPCC OpenACC benchmark suite,” in Proc. 25th Euromicro Int. Conf. Parallel, Distrib. Netw. Process. (PDP), 2017, pp. 246–250.

N. Johnson and A. Jackson, “The EPCC OpenACC benchmark suite,” in Proc. Exascale Appl. Softw. Conf., Edinburgh, U.K., Apr. 2013, pp. 1–15. [Online]. Available: https://github.com/EPCCed/epcc-openacc-benchmarks

S. Grauer-Gray, L. Xu, R. Searles, S. Ayala-somayajula, and J. Cavazos, “Auto-tuning a high-level language targeted to GPU codes,” in Proc. Innov. Parallel Comput. (InPar), San Jose, CA, USA, May 2012, doi: 10.1109/InPar.2012.6339595.

C. Liao, P.-H. Lin, J. Asplund, M. Schordan, and I. Karlin, “DataRaceBench: A benchmark suite for systematic evaluation of data race detection tools,” in Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal., Denver, CO, USA, Nov. 2017, pp. 1–14.

D. Griebl, J. Loff, G. Mencaglì, M. Danelluto, and L. G. Fernandes, “Efficient NAS benchmark kernels with C++ parallel programming,” in Proc. 26th Euromicro Int. Conf. Parallel, Distrib. Netw. Process. (PDP), Cambridge, U.K., Mar. 2018, pp. 733–740.

HUDA SALEH BASLOOM received the bachelor’s and master’s degrees in software engineering from King Abdulaziz University, Jeddah, Saudi Arabia, in 2002 and 2011, respectively, where she is currently pursuing the Ph.D. degree. She is also a Staff Member at the Deanship of Information Technology, King Abdulaziz University. She has more than nine years of practical experience in systems programming and software testing. Her research interests include high-performance computing, information security, distributed systems, programming models, software engineering, and software testing.

MOHAMED YEHIA DAHAB was a Knowledge Engineer at the International Center for Agricultural Research in Dry Areas (ICARDA), from 2005 to 2006. He was also the Chairperson of the Agricultural Expert Systems Development Department, The Central Laboratory for Agricultural Expert Systems (CLAES), Ministry of Agriculture, Egypt, for two years, from 2006 to 2009. He is currently an Assistant Professor at the Department of Computer Science, Faculty of Computing and Information Technology, King Abdulaziz University (KAU), Jeddah, Saudi Arabia. His main research interests include pattern recognition, prediction, natural language processing, expert systems, knowledge bases, text mining, and information retrieval.

AHMED MOHAMMED ALGHAMDI received the B.Sc. degree in computer science and the M.Sc. degree in business administration from King Abdulaziz University, Jeddah, Saudi Arabia, in 2005 and 2010, respectively, the master’s degree in internet computing and network security from Loughborough University, Loughborough, U.K., in 2013, and the Ph.D. degree in computer science from King Abdulaziz University, in 2019. He is currently an Assistant Professor with the Department of Software Engineering, College of Computer Science and Engineering, University of Jeddah, Saudi Arabia. His research interests include high-performance computing, big data, distributed systems, programming models, software engineering, and software testing.

FATHY ELBOURAYE 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, joint supervision with the University of Colorado at Boulder, Boulder, CO, USA. He is currently a Full Professor at Department of Computer Science, Faculty of Computing and Information Technology, King Abdulaziz University, Saudi Arabia. His research interests include agent-based software engineering, the IOT security, software engineering, big data management and security, distributed systems security, and exascale systems testing.

ABDULLAH SAAD AL-MALAISE AL-GHAMDI received the Ph.D. degree in computer science from George Washington University, Washington, DC, USA, in 2003. He is currently a Professor in software and systems engineering and AI, and associated with the Department of Information Systems (IS), Faculty of Computing and Information Technology (FCIT), King Abdulaziz University (KAU), Jeddah, Saudi Arabia. He is also working as the Head of the Consultant’s Unit; the Vice President for Development Office, KAU; and the Head of the Department of Computer Skills, FCIT.

SEIF HARIDI is currently the Chair Professor of computer systems specialized in parallel and distributed computing and the Head of Distributed Computing Group (DC), KTH Royal Institute of Technology, Stockholm, Sweden. He is also the Chief Scientific Advisor of RISE SICS, until December 2019. He led a European Research Program on cloud computing and big data at EIT-Digital, from 2010 to 2013, and the Co-Founder of a number of start-ups in the area of distributed and cloud computing, including HiveStreaming and LogicalClocks. He is a Co-Designer of SICStus Prolog, the most well-known logic programming system, and the Mozart Programming System, and a high-quality open-source development platform based on the Oz multi-model programming language. For more than ten years, he has been teaching two popular courses on distributed algorithms and peer-to-peer computing at KTH. Recently, together with his research group, he has been contributing to the design of Apache Flink, a big data distributed engine for streaming analytics, and HOPS the first European complete platform for data-analytics and machine learning. He is well-known for the textbook Concepts, Techniques, and Models of Computer Programming that explains many difficult programming concepts in a simple and insightful way. His research interests include combination of systems research and theory in the areas of programming systems and distributed computing. HOPS-FS the file system of HOPS won IEEE Scale Prize 2017 as the most scalable HDFS file system. He is also with LogicalClocks and Hopsworks won the award for European Data Science Technology Innovation, in 2019.

* * *