symmetry

MDPI

Article

Parallel Hybrid Testing Techniques for the Dual-Programming Models-Based Programs

Ahmed Mohammed Alghamdi [1,*], Fathy Elbouraey Eassa [2], Maher Ali Khamakhem [2], Abdullah Saad AL-Malaise AL-Ghamdi [3], Ahmed S. Alfakeeh [3], Abdullah S. Alshahrani [4] and Ala A. Alarood [5]

1 Department of Software Engineering, College of Computer Science and Engineering, University of Jeddah, Jeddah 21493, Saudi Arabia
2 Department of Computer Science, Faculty of Computing and Information Technology, King Abdulaziz University, Jeddah 21589, Saudi Arabia; feassa@kau.edu.sa (F.E.E.); makhemakhem@kau.edu.sa (M.A.K.)
3 Department of Information Systems, Faculty of Computing and Information Technology, King Abdulaziz University, Jeddah 21589, Saudi Arabia; aalmalaise@kau.edu.sa (A.S.A.-M.A.-G.); asalfakeeh@kau.edu.sa (A.S.A.)
4 Department of Computer Science and Artificial Intelligence, College of Computer Science and Engineering, University of Jeddah, Jeddah 21493, Saudi Arabia; asalshahrani2@uj.edu.sa
5 College of Computer Science and Engineering, University of Jeddah, Jeddah 21432, Saudi Arabia; alaa.alarood@mail.com or aasoleman@uj.edu.sa
* Correspondence: amalghamdi@uj.edu.sa

Received: 4 September 2020; Accepted: 18 September 2020; Published: 20 September 2020

Abstract: The importance of high-performance computing is increasing, and Exascale systems will be feasible in a few years. These systems can be achieved by enhancing the hardware's ability as well as the parallelism in the application by integrating more than one programming model. One of the dual-programming model combinations is Message Passing Interface (MPI) + OpenACC, which has several features including increased system parallelism, support for different platforms with more performance, better productivity, and less programming effort. Several testing tools target parallel applications built by using programming models, but more effort is needed, especially for high-level Graphics Processing Unit (GPU)-related programming models. Owing to the integration of different programming models, errors will be more frequent and unpredictable. Testing techniques are required to detect these errors, especially runtime errors resulting from the integration of MPI and OpenACC; studying their behavior is also important, especially some OpenACC runtime errors that cannot be detected by any compiler. In this paper, we enhance the capabilities of ACC_TEST to test the programs built by using the dual-programming models MPI + OpenACC and detect their related errors. Our tool integrated both static and dynamic testing techniques to create ACC_TEST and allowed us to benefit from the advantages of both techniques reducing overheads, enhancing system execution time, and covering a wide range of errors. Finally, ACC_TEST is a parallel testing tool that creates testing threads based on the number of application threads for detecting runtime errors.

Keywords: hybrid-testing tool; MPI + OpenACC; dual-programming model; Exascale systems; ACC_TEST

1. Introduction

Exascale systems will be available in a few years. These systems can be achieved by enhancing hardware ability as well as parallelism in the application by integrating different programming models using dual- and tri-programming models. Exascale systems can achieve 10^{18} floating-point operations

per second with thousands or millions of threads, which will involve several difficulties including runtime errors.

Parallel programs built by using programming models have been targeted by different testing tools. In particular, high-level programming models targeting Graphics Processing Unit (GPU) need more effort. Because of the integration between different programming models, errors will be more frequent and unpredictable and therefore need testing techniques to detect them. As a result, building a testing tool targeting parallel programs is not an easy task, especially when these parallel programs are built using integrated programming models. In addition, there is a shortage of testing tools targeting parallel systems, which use dual-programming models.

Previously, we proposed an architecture for a parallel testing tool targeting programs built with heterogeneous architecture, including the dual-programming models Message Passing Interface (MPI) + OpenACC, and covering different errors [1–3]. In addition, we proposed a static testing approach that targets OpenACC-related programs [4]. We then improved that to create a hybrid testing tool that targets OpenACC-related programs by integrating static and dynamic approaches; we named it ACC_TEST [5,6].

Parallelism and performance can be enhanced by integrating several programming models along with the ability to work in heterogeneous platforms. Using hybrid programming models helps move towards Exascale, which needs programming models for supporting massively parallel systems. The programming models can be classified into three levels: single-level programming model: MPI [7]; dual-level programming model: MPI + OpenMP [8]; and tri-level programming model: MPI + OpenMP + CUDA [9].

In this paper, the dual-programming model MPI + OpenACC has been chosen, as combining MPI and OpenACC enhances parallelism and performance and reduces programming efforts. OpenACC can be compiled to work in different hardware architecture as well as multiple device types, and MPI will be used for exchanging data between different nodes. The dual-programming model combines the advantages of both programming models, including programmability and portability from OpenACC and high performance, scalability, and portability from MPI [10].

We chose this combination because OpenACC has allowed non-computer science specialists to parallelize their programs, which can possibly lead to errors when OpenACC directives and clauses are misused. OpenACC uses high-level directives without considering much detail, which is one of its main features. In addition, OpenACC has the ability to work on any GPU platform, which gives it high portability. We believe that this combination needs to be considered for testing because there is a high possibility of runtime errors occurring when programmers use MPI + OpenACC. Some researchers claim that MPI + CUDA [9], a hybrid programming model, can be applicable on heterogeneous cluster systems. However, this combination only works on NVIDIA GPU devices and CUDA is considered a low-level programming model that needs various details. In addition, the combination of MPI + OpenMP [8] targets homogenous architecture through shared memory, which does not serve the purpose of using the GPU in accelerating the user codes.

The first part of the chosen combination is Message-Passing Interface (MPI), the first version of which was released in 1994 [11]. MPI supports parallelism in C, C++, and Fortran by using message passing to establish standard efficient, portable, and flexible message passing programs. MPI has the ability to be integrated with other programming models, including shared memory-related programming models like OpenMP and GPU-related programming models like OpenACC. In addition, MPI has two types of communication, including point-to-point and collective, which can be blocking or non-blocking. Finally, MPI has released the latest version MPI 4.0, which has the ability to better support hybrid-programming models [11].

The second part is OpenACC, which has been released to support parallelism and accelerate C, C++, and Fortran codes for heterogeneous CPU/GPU systems by using high-level directives. OpenACC has three directive types, including compute, data management, and synchronization directives. In November 2019, the latest OpenACC version 3.0 [12] was released to include many

features to support parallelism, including portability, flexibility, compatibility, and less programming effort and time.

In this paper, the ACC_TEST has been improved upon to have the ability to test programs built using MPI + OpenACC dual-programming models and detect related errors. Our solution aimed at covering a wide range of errors that occur in the dual-programming models MPI + OpenACC with less overhead and better system execution time. Finally, our testing tool works in parallel by detecting runtime errors with testing threads created based on application threads numbers.

This paper is structured as follows: Section 2 provides a related work, including testing tools classified by the testing techniques that were used. We explain our techniques for testing the programs based on dual-programming models in Section 3. We discuss our implementation, testing, and evaluation of ACC_TEST in Section 4 and show some results from our experiments. The conclusion and future scope will be discussed in Section 5.

2. Related Work

In our study, approximately 30 different tools and techniques were reviewed, varying from open-source to commercial tools. Including different types of testing techniques targeted at programming models led to discovering runtime errors for different purposes when finding errors or tracking the cause of these errors (debuggers). Only parallel systems-related testing has been included in our study, in which we survey parallel systems testing techniques. In addition, we focus on the testing techniques used to detect runtime errors that occur on parallel systems. The tools and techniques in our study were chosen from a wide range of available testing tools and techniques in our full published survey [13]. We eliminated any tool or technique that did not meet our objectives. We aimed to survey testing tools and techniques that detect runtime errors in parallel systems using programming models. Therefore, we classified the used testing techniques into four categories, namely static, dynamic, hybrid, and symbolic testing techniques. Further, we classified these techniques into two subcategories to determine the targeted programming model level, single- or dual-level programming models. The following subsections will discuss our classifications.

2.1. Static Testing Techniques

Five testing tools were classified as using the static testing technique to detect errors in parallel programs that parallelize using programming models. The testing tools [14] and GPUVerify [15] used the static technique to detect data race in CUDA, OpenCL, and OpenMP programming models individually. For OpenMP, the testing tools [16] and ompVerify [17] were used to detect data race. Finally, MPI-Checker [18] used static techniques to detect MPI mismatching errors.

2.2. Dynamic Testing Techniques

In our study, many testing tools use dynamic testing to detect runtime errors in parallel programs. Regarding detecting errors in the MPI programming model, fourteen testing tools use a dynamic technique that targets MPI. The testing tools MEMCHECKER [19], MUST [20], STAT [21], Nasty-MPI [22], and Intel Message Checker [23] were used to detect MPI runtime errors, including deadlocks, data race, and mismatching. For detecting deadlocks and mismatching, MPI-CHECK [24], GEM [25], and Umpire [26] were used. The tools PDT [27], MAD [28], and [29] were used for detecting MPI deadlocks and race conditions. For deadlocks, MOPPER [30] and ISP [31] were used. Finally, MPIRace-Check [32] was used for detecting MPI race condition.

For detecting data race in CUDA using dynamic techniques, the testing tool in [33] was used. Regarding data race detection, there are several testing tools for different programming models, including GUARD [34], RaceTM [35], and KUDA [36] for CUDA. For detecting errors in a heterogeneous programming model by using dynamic testing, WEFT [37] was used to detect deadlocks and race conditions. Finally, for testing the hybrid MPI/OpenMP programming model using dynamic testing, the testing tools MARMOT [38] were used for detecting deadlocks, race conditions, and mismatching.

2.3. Hybrid Testing Techniques

Several reviewed testing tools used hybrid-testing techniques, which combine static/dynamic testing. In our survey, five tools used hybrid-testing techniques. These tools are classified into tools targeting single- and dual-level programming models.

In terms of testing tools designed for the single-level programming model, four testing tools targeted single-level programming models, including OpenMP, OpenCL, and CUDA. ARCHER [39] and Dragon [40] are testing tools that use hybrid testing techniques to detect data race in the OpenMP programming model. GMRace [41] and GRace [42] use hybrid testing techniques to detect data race in the CUDA programming model. Finally, GRace is also used to test the OpenCL programming model for detecting data race. All the previous testing tools used static/dynamic hybrid testing techniques to detect runtime errors.

Two testing tools used static/dynamic hybrid testing techniques to detect runtime errors in MPI + OpenMP dual programming model. These tools are PARCOACH [43] and [44], which used the hybrid model to detect deadlocks and other runtime errors resulting from the dual programming model. Even though combining two programming models is beneficial, it creates complex runtime errors that are difficult to detect and determine.

It is noticeable that dynamic testing used, mainly for detecting runtime errors for different programming models. Single-level programming models were targeted to be tested, especially MPI and OpenMP because of their wide use and their history in programming models. Regarding heterogeneous programming models in the reviewed testing tools, CUDA is the most targeted programming model, while OpenACC has not been targeted in any reviewed testing tools for detecting runtime errors, despite its benefits and trending use. However, there are some OpenACC-related studies, including compilers' evaluation of OpenACC 2.0 in [7] and OpenACC 2.5 in [8]. In the published paper [9], an evaluation comparative study was conducted for the compilers, comparing and evaluating PGI, CRAY, and CAPS compilers. For testing high numerical programs, PGI Compiler Assisted Software Testing (PCAST) [45] was released as a feature in PGI compilers and runtime. PCAST is useful to detect errors when numerical results diverge between CPU and GPU versions of code and when they run on different processor architectures within the same code. However, PCAST cannot detect runtime errors, including errors in the OpenACC data directives, race conditions, and deadlock. Finally, we believe that a lot of work needs to be done in creating and developing testing tools for massively parallel systems, especially heterogeneous parallel systems, which will be needed when the Exascale systems are applied in different fields.

3. Our Techniques for Testing Dual-Programming Models Related Programs

We designed ACC_TEST for detecting errors in parallel programs created by using the integrated model MPI + OpenACC. Our solution integrates the static and dynamic analysis for checking the actual and potential errors with lower overheads and covering a wide range of errors. In our solution, we use the static analysis for discovering as many errors as possible as well as annotate any potential errors for further dynamic testing. This method helps us reduce any unnecessary code instrumentation and minimize the dynamic testing to cover only the code parts that need to be tested during runtime. The dynamic analysis will be used to check and detect any errors that cannot be detected during our static analysis and to detect thread interactions during runtime for any possible errors like race condition and deadlock. An overview of our solution is shown in Figure 1.

ACC_TEST classifies the targeted source code into several parts for ensuring efficient error detection so that only the parts that needed to be investigated will be covered. This classification includes OpenACC data and compute regions, MPI point-to-point and collective communications, as well as non-parallel code parts. ACC_TEST detects errors based on the OpenACC error classification published in [4] that classified OpenACC errors into OpenACC data managements errors, race condition, deadlock, and livelock. In addition, MPI errors are detected by ACC_TEST based on [46] that classified MPI errors into deadlock, data race, mismatches, resource handling, and memory errors.

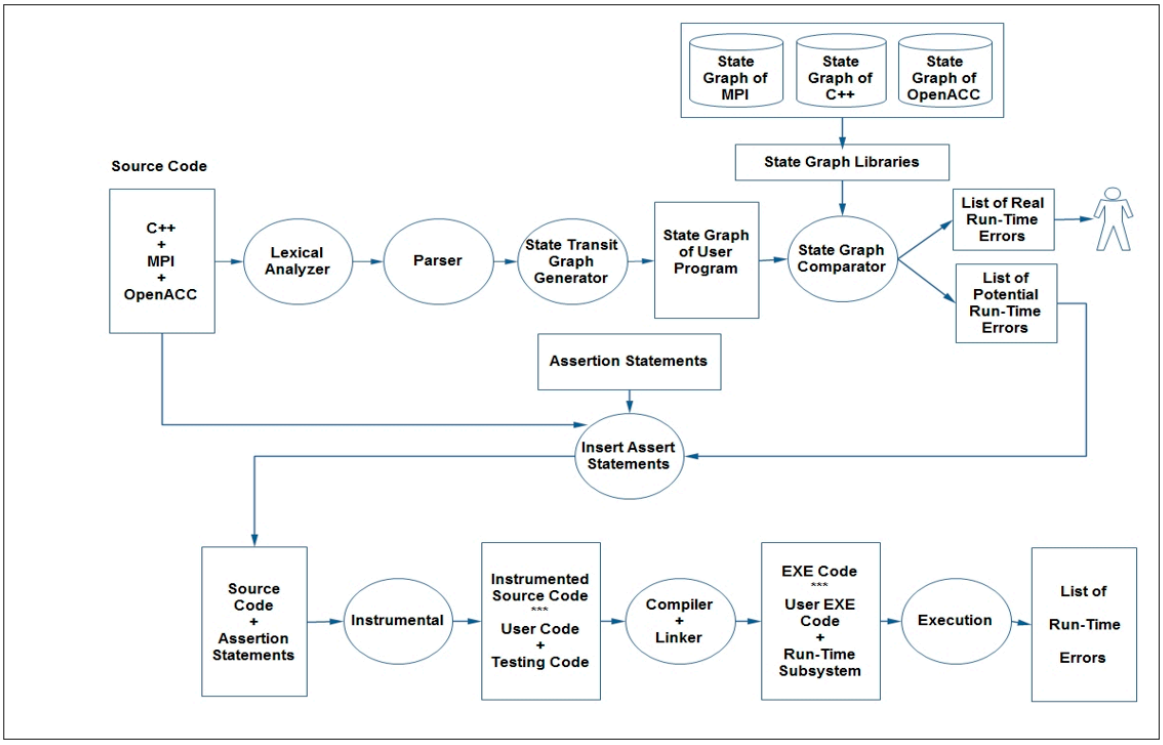

Figure 1. Our hybrid testing tool architecture.

OpenACC data clause detection is built based on the assumption that the OpenACC data clause can be used inefficiently, which makes it error-prone and the errors cannot be detected by a compiler. As a result, ACC_TEST investigates all OpenACC data clauses in the source code using our static approach to ensure their correctness, including their syntax and semantics. There are two main types of OpenACC data clauses, structured and unstructured data clauses, which need to be checked differently based on their nature. Algorithm 1 was built to check the structure data clauses in both data and compute regions because of their similar behavior and role. This algorithm used our static approach to examine any variable related to OpenACC data clause directives and determine their locations in three places in the targeted source code, including data region, before and after this region. In case of any error detected by our static approach, an error message will be written to the error list resulting from ACC_TEST.

In terms of the OpenACC unstructured data clause, Algorithm 2 is responsible for detecting any error related to the unstructured data clause as well as detecting the data movements between the enter and exit region. In addition, some behaviors that are not considered an error, but could cause memory-related problems, will be detected by our static approach when variables are moved to the GPU without being used or a temporary GPU variable is created and not deleted at the exit region.

ACC_TEST dynamic approach uses the information resulting from our static testing to determine the place of the present data clause in the source code and its variable list. Then, our dynamic instrumentation uses the API function (acc_is_present), which indicates if the variable is present on the device to test the present clause. In addition, an if-statement will be inserted to test the result of the OpenACC API function and issue error message, indicating the error type and place to the programmer. The following code in Figure 2 is an example of our insertion statements before being instrumented, where the var_name, var_size, and var_type will be determined by our static testing and inserted with the comment character, followed by the word "ASSERT" to be distinguished by our instrumenter.

Algorithm 1: Structured Data Clause Check Algorithm

```
    START
1:  RECEIVED ACC_DClauseVar, ACC_DClauseType
2:  SEARCH for ACC_DClauseVar Location in the source code
3:  STORE ACC_DClauseVar, ACC_DClauseType, ACC_DClauseVarLineNo, ACC_DClauseVarFileName,
    ACC_DClauseVarLocation IN ACC_DClauseVarList
4:  WHILE ACC_DClauseVar in ACC_DClauseVarList
5:    CASE ACC_DClauseVarLocation:
6:      In_ ACCDataRegion:
7:        IF ACC_DClauseVar founded
8:          IF ACC_DClauseVar part of equation
9:            IF ACC_DClauseVar is define && ACC_DClauseType = copyin
10:             ERROR REPORTED
11:           ELSE IF ACC_DClauseVar is reused && ACC_DClauseType = copyout
12:             ERROR REPORTED
13:           ELSE IF ACC_DClauseVar is reused && ACC_DClauseType = copyin && the
    ACC_DefineVar not in copy clause or copyin clause
14:             ERROR REPORTED
15:           ELSE IF ACC_DClauseVar is define && ACC_DClauseVar is reused && (ACC_DClauseType
    = copyin or ACC_DClauseType = copyout)
16:             ERROR REPORTED
17:           END IF
18:         ELSE IF ACC_DClauseVar is CALLED by function
19:           IF ACC_DClauseVar is CALLED by value
20:             ERROR REPORTED
21:           ELSE IF ACC_DClauseType = copyout
22:             ERROR REPORTED
23:           END IF
24:         END IF
25:       ELSE
26:         ERROR REPORTED
27:       END IF
28:     Before_ ACCDataRegion:
29:       IF ACC_DClauseVar founded
30:         IF ACC_DClauseVar initialized
31:           PASS
32:         ELSE IF ACC_DClauseVar used as define && (ACC_DClauseType = create or
    ACC_DClauseType = copyout)
33:           ERROR REPORTED
34:         ELSE
35:           ERROR REPORTED
36:         END IF
37:       ELSE
38:         ERROR REPORTED
39:       END IF
40:     After_ ACCDataRegion:
41:       IF ACC_DClauseVar founded
42:         IF ACC_DClauseType = create or ACC_DClauseType = copyin
43:           ERROR REPORTED
44:         ELSE
45:           PASS
46:         END IF
47:       ELSE
48:         ERROR REPORTED
49:       ENDIF
50: END CASE
51: END WHILE
    END
```

Algorithm 2: Unstructured Data Clause Check Algorithm

```
START
RECEIVE ACC_DClauseVar, ACC_DClauseType
SEARCH for ACC_DClauseVar Locations in the source code
STORE ACC_DClauseVar, ACC_DClauseType, ACC_DClauseVarLineNo, ACC_DClauseVarFileName,
ACC_DClauseVarLocation IN ACC_DClauseVarList
WHILE ACC_DClauseVar in ACC_DClauseVarList
  IF (ACC_DClauseType = copyin or ACC_DClauseType = create) && (ACC_DClauseVar not founded at
exit data region directive)
    ERROR REPORTED
  ELSE IF (ACC_DClauseType = copyout or ACC_DClauseType = delete) && (ACC_DClauseVar not
founded at enter data region directive)
    ERROR REPORTED
  END IF
  CASE ACC_DClauseVarlocation:
    In_ ACCDataRegion:
      IF ACC_DClauseVar founded
        IF ACC_DClauseVar part of equation
          IF ACC_DClauseVar is define
            IF ACC_DClauseType = delete
              ERROR REPORTED
            ELSE IF (ACC_DClauseType = create or ACC_DClauseType = copyin) &&
ACC_DClauseVar not founded in copyout at exit
              ERROR REPORTED
            END IF
          ELSE IF ACC_DClauseVar is reused
            IF ACC_DClauseType = copyout
              ERROR REPORTED
            ELSE IF ACC_DClauseType = create or ACC_DClauseType = copyin
              IF ACC_Definedvar not in copyout at exit
                ERROR REPORTED
              ELSE IF ACC_DClauseVar not in delete at exit
                ERROR REPORTED
              END IF
            END IF
          ELSE IF ACC_DClauseVar is define and reused
            IF ACC_DClauseType = delete
              ERROR REPORTED
            ELSE IF ACC_DClauseType = copyout && ACC_DClauseVar not at enter region
              ERROR REPORTED
            ELSE IF (ACC_DClauseType = copyin or ACC_DClauseType = create) &&
ACC_DClauseVar is not in copyout at exit region
              ERROR REPORTED
            END IF
          END IF
        ELSE IF ACC_DClauseVar is CALLED by function
          IF ACC_DClauseVar is CALLED by value
            ERROR REPORTED
          ELSE IF ACC_DClauseVar is CALLED by reference
            IF (ACC_DClauseType = copyin or ACC_DClauseType = create) && ACC_DClauseVar not
in delete at exit
              ERROR REPORTED
```

```
                ELSE IF ACC_DClauseType = copyout
                  ERROR REPORTED
                END IF
              END IF
            END IF
          ELSE
            ERROR REPORTED
          END IF
        Before_ ACCDataRegion:
          IF ACC_DClauseVar founded
            IF ACC_DClauseVar initialized
              PASS
            ELSE IF ACC_DClauseVar used in the equation as define
              IF ACC_DClauseType = create
                ERROR REPORTED
              ELSE IF (ACC_DClauseType = copyout or ACC_DClauseType = delete) &&
ACC_DClauseVar not in copyin or create at enter
                ERROR REPORTED
              END IF
            ELSE
              ERROR REPORTED
            END IF
          ELSE
            ERROR REPORTED
          END IF
        After_ ACCDataRegion:
          IF ACC_DClauseVar founded
            IF ACC_DClauseType = delete
              ERROR REPORTED
            ELSE IF (ACC_DClauseType = copyin or ACC_DClauseType = create) && ACC_DClauseVar
not in copyout at exit
              ERROR REPORTED
            END IF
          ELSE IF ACC_DClauseVar not founded
            IF ACC_DClauseType = copyout
              ERROR REPORTED
            ELSE IF (ACC_DClauseType = copyin or ACC_DClauseType = create) && ACC_DClauseVar
not in delete at exit
              ERROR REPORTED
            ELSE IF ACC_DClauseVar in delete and copyout at exit
              ERROR REPORTED
            END IF
        END IF
      END CASE
    END WHILE
    END
```

ACC_TEST identifies a variable to receive the value from the API function (acc_is_present), which takes the variable name, size, and type as shown in the previous code. This OpenACC API function tests if the data are present on the GPU. In C and C++, the function returns nonzero value if the data are present and zero if not present. If (acc_is_present) returns a nonzero value, this shows that the targeted variable is present in the current GPU, which can then complete the program. Our instrumenter will

remove any comment character followed by our inserted label "ASSERT" to keep the insert test code to be compiled, when the user code and the inserted test code move to the instrumenter phase.

```
// Inserted statements for testing the present clause during run-time
//"ASSERT" int present = acc_is_present (var_name, var_size * sizeof (var_type));
//"ASSERT" if (present)
//"ASSERT" {
//"ASSERT" cout << "This variable is present in the GPU with value = " << var_name << endl;
//"ASSERT"}
//"ASSERT" else
//"ASSERT" {
//"ASSERT" cout << "XXX ERROR :: This variable is not present in the current GPU XX" << endl;
//"ASSERT" }
```

Figure 2. Our inserted statements for testing the OpenACC present clause before the instrumenter.

In terms of detecting OpenACC race condition, there are several situations that cause race condition in OpenACC, including host and device synchronization, loop parallelization, shared data read-and-write, asynchronous directives, reduction clause, and independent clause races. Our instrumentation mechanism will use the static approach annotation to insert codes into the targeted source code for collecting information during runtime. In the code given in Figure 3, our insertion mechanism will insert data structure for collecting actual information during runtime and use them for several test cases in our dynamic testing phase after the instrumentation phase.

```
// Inserted data structure to collect some important information during run-time
//"ASSERT" struct mem_table
//"ASSERT" {
//"ASSERT" int id[1024];              //the Struct id
//"ASSERT" int line_no;               //the line Number of the Variable
//"ASSERT" int Compute_region_id;     //the Compute Region Number
//"ASSERT" int* Device_Addr[1024];    // the address on GPU (Device)
//"ASSERT" int Gang_id[1024];         //the gang id = block id
//"ASSERT" int Vector_id[1024];       // the vector id = thread id
//"ASSERT" int Total_Gangs;           // Total Number of Gang in the compute region
//"ASSERT" int Total_Vectors;         // Total Number of Vector in the compute region
//"ASSERT" int Tested_Total_Gangs;    // Total Number of Gang in the compute region from the testing tool
//"ASSERT" int Tested_Total_Vectors;  // Total Number of Vector in the compute region from the testing tool
//"ASSERT" };
```

Figure 3. Inserted data structure for collecting actual information during runtime.

For detecting data dependency in OpenACC compute region loops, Algorithm 3 shows the process of detecting loop race condition, finding any dependency, and the possibility of having race condition within each loop threads.

For ensuring there are worked parallel codes, we detect the threads generated in OpenACC, including gangs and vectors. Our static testing phase generates tested gangs and vectors for each compute region for further comparison with the actual number of gang and vectors generated by the original source code. By using this compassion, the user code performance could be enhanced, because users assume that some compute regions work in parallel when they actually work sequentially. For each thread, some information will be collected during runtime, as shown in the following inserted test code in Figure 4, which will be used in our dynamic testing phase after instrumentation. These inserted statements will be added to each OpenACC compute region.

Algorithm 3: Race Detection: Data Dependency Check Algorithm

START
SEARCH Loops inside ACCComputeRegion
STORE ACC_LoopID, ACC_LoopLineNo, ACC_LoopIterationVar, ACC_LoopStartValue, ACC_LoopEndValue, ACC_LoopComputeRegion,ACC_LoopIndependant
SEARCH Equation in loop
STORE ACC_EquationID, ACC_EquationComputeRegion, ACC_EquationLoopID, ACC_EquationArrayVarName, ACC_EquationArrayVarStatus, ACC_EquationArrayVarIndex, ACC_ EquationArrayVarType, ACC_EquationArrayVarPlace
IF (ACC_EquationArrayVarName_1 = ACC_EquationArrayVarName_2) && (ACC_EquationComputeRegion_1 = ACC_EquationComputeRegion_2)
  IF ACC_EquationArrayVarType = Array
    IF (ACC_EquationArrayVarIndex_1 != ACC_EquationArrayVarIndex_2) && (ACC_EquationArrayVarPlace_1 != ACC_EquationArrayVarPlace_2)
      ERROR REPORTED
    ELSE IF Equation In ACC_IndependentClause
      ERROR REPORTED
    END IF
  END IF
END IF
END

```
// Inserted Test code for collecting information from each thread and store them in the data structure
//"ASSERT" Data_Structure.Thread_id = Thread_ID;
//"ASSERT" Data_Structure.Compute_region_id = Compute_region_id;
//"ASSERT" Data_Structure.Gang_id[Thread_ID] = __pgi_gangidx();
//"ASSERT" Data_Structure.Vector_id[Thread_ID] = __pgi_vectoridx();
//"ASSERT" Data_Structure.Tested_Total_Gangs = Tested_Total_Gangs;
//"ASSERT" Data_Structure.Tested_Total_Vectors = Tested_Total_Vectors;
//"ASSERT" Data_Structure.line_no = Line_Number;
//"ASSERT" Data_Structure.Device_Address = Device_Address;
```

Figure 4. Inserted test code for collecting information for each thread in each compute region.

All thread information in each OpenACC compute region from the inserted test code in Figure 5 will be used to test actual parallelism for each OpenACC compute region and detect any differences between the tested and actual parallelism indicating the compute region, which is not parallelized.

```
// Inserted test code for comparing between the actual number of gang and vectors from our dynamic tester and the tested
// number from our static tester
//"ASSERT" if (Data_Structure.Tested_Total_Gangs != Total_Gangs | Data_Structure.Tested_Total_Vectors != Total_Vectors)
//"ASSERT" {
//"ASSERT" cout << "The GANG/VECTOR in Compute Region No --> (" << Compute_region_id << "), are Indecating That
This Compute Region is Not Parallize " << endl;
//"ASSERT" cout << "The Actual Gang/Vector are " << Total_Gangs << " gangs and " << Total_Vectors << " vectors" << endl;
//"ASSERT" cout << "But, they should be " << Data_Structure.Tested_Total_Gangs << " gangs and " <<
Data_Structure.Tested_Total_Vectors << " vectors" << endl;
//"ASSERT" }
```

Figure 5. Inserted test code for comparing between actual gang and vectors and the tested gang and vectors.

In the case of read-write race condition, Algorithm 4 will be used by ACC_TEST static approach to detect different types of related race conditions.

Algorithm 4: Race Detection: Read-Write Check Algorithm

```
    START
1:  SEARCH Equation in ACCComputeRegion
    STORE ACC_EquationID, ACC_EquationComputeRegion, ACC_EquationLoopID,
2:  ACC_EquationArrayVarName, ACC_EquationArrayVarStatus,ACC_EquationArrayVarIndex,
    ACC_EquationArrayVarType
3:  IF (ACC_EquationID_1 = ACC_EquationID_2) && (ACC_EquationArrayVarName_1 =
    ACC_EquationArrayVarName_2)
4:      IF (ACC_EquationArrayVarIndex_1 != ACC_EquationArrayVarIndex_2)
5:        IF ACC_EquationArrayVarStatus_1 = Read && ACC_EquationArrayVarStatus_2 = Write
6:          ERROR REPORTED
7:        ELSE IF ACC_EquationArrayVarStatus_1 = Write && ACC_EquationArrayVarStatus_2 = Read
8:          ERROR REPORTED
9:      ELSE IF ACC_EquationArrayVarStatus_1 = Write && ACC_EquationArrayVarStatus_2 = Write
10:         ERROR REPORTED
11:       END IF
12:   END IF
13: END IF
    END
```

In our dynamic phase, for the case of data race detection for writing and reading to memory and at compiling time, if the addresses cannot be determined or the addresses potentially have conflicts, ACC_TEST static approach will insert codes to be investigated during the dynamic phase. In our dynamic approach, each memory access statement that is marked by our static analysis will be monitored, and their information will be recorded to detect any data race. The information will include variable, thread id, operation (R/W), and memory addresses.

To detect race condition that results from reading and writing to or from the same address in the Host or Device, our dynamic tester will instrument the inserted test statements in the user code to register each address space in Host and Device and then check if there is any writing to the same address at the same time. In this case, the dynamic tester will execute user code and inserted code to discover if there is any reading or writing to the same address. If so, an error message is shown to the user, indicating the race condition along with the reason. The inserted test code is shown in Figure 6, which will use the data structure to store actual information during runtime for each thread, as explained earlier. These test codes will investigate the device addressed in the same compute region as well as across different compute regions.

In terms of detecting OpenACC deadlock, ACC_TEST will use the static analysis to partially detect deadlock by annotating the code parts that need further investigation during runtime. The dynamic part of ACC_TEST will use the instrumentations annotated for testing the threads arrival at the end of each OpenACC compute region. This approach was used because we assumed that each OpenACC parallel compute region could have deadlock. At the end of each OpenACC compute region, our dynamic phase will check the number of threads included in the region and compare them by the number of threads at the end of the OpenACC region. Because of the implicit barrier on OpenACC, which hides information from the developers, our approach will make sure any unexpected behavior is investigated and reported to the developers along with related information. ACC_TEST dynamic approach will also be used to detect deadlock caused by GPU livelock as well as livelock that can occur in the Host and Device interaction.

```
// Inserted test code for detecting and shared memory error by reading and writing from the same address space
// The Following is an Inserted Statements for Testing The Device Addresses In The Same Compute Region-->
//"ASSERT" if(Data_Structure.Device_Addr[Thread_ID_1] == Data_Structure.Device_Addr[Thread_ID_2] &&
Data_Structure.Device_Addr[Thread_ID_1] != 0)
//"ASSERT" {
//"ASSERT" cout << "XX ERROR :: Two Variables Have Been Stored In the Same Address Space At The GPU" << endl;
//"ASSERT" cout << " At The Same Compute Region No.("<< Compute_region_id <<") - With The ID (" <<
Data_Structure.Thread_id[Thread_ID_1] <<") << endl;
//"ASSERT" cout << " and ID (" << Data_Structure.Thread_id[Thread_ID_2] <<") At The Address ("<<
Data_Structure.Device_Addr[Thread_ID_1] <<")" << endl;
//"ASSERT" }

// The Following is an Inserted Statements for Testing The Device Addresses In Different Compute Region-->
//"ASSERT" if(Data_Structure[1].Device_Addr[Thread_ID_1] == Data_Structure[2].Device_Addr[Thread_ID_1] &&
Data_Structure[1].Device_Addr[Thread_ID_1] != 0)
//"ASSERT" {
//"ASSERT" cout << "ERROR :: Two Variables Have Been Stored In the Same Address Space At The GPU " << endl;
//"ASSERT" cout << "At The Compute Region No.("<< Compute_region_id_1 <<")- With The ID (" <<
Data_Structure[1].Thread_id <<")" << endl;
//"ASSERT" cout << "At the Address The Address ("<< Data_Structure[1].Device_Addr[Thread_ID_1] <<")" << endl;
//"ASSERT" cout << "Compute Region No."<< Compute_region_id_2 <<"- With ID " << Data_Structure[2].Thread_id <<" At The
Address" << endl;
//"ASSERT" cout << Data_Structure[1].Device_Addr[Thread_ID_1] << endl;
//"ASSERT" }
//"ASSERT" else if(Data_Structure[1].Device_Addr[Thread_ID_1] == Data_Structure[2].Device_Addr[Thread_ID_2]&&
Data_Structure[2].Device_Addr[Thread_ID_1] != 0)
//"ASSERT" {
//"ASSERT" cout << "ERROR :: Two Variables Have Been Stored In the Same Address Space At The GPU" << endl;
//"ASSERT" cout << "At The Compute Region No.("<< Compute_region_id_1 <<") - With The ID (" << Data_Structure.Thread_id
<<")" << endl;
//"ASSERT" cout << "At the Address the Address ("<< Data_Structure[1].Device_Addr[Thread_ID_1] <<")" << endl
//"ASSERT" cout << " and Compute Region No.("<< Compute_region_id_2 <<") - With The ID (" << Data_Structure[2].Thread_id
<<") At The Address << endl;
//"ASSERT" cout <<("<< Data_Structure[1].Device_Addr[Thread_ID_1] <<")" << endl;
//"ASSERT" }
```

Figure 6. Inserted test code for detecting and shared memory error by reading and writing from same address space.

In the case of deadlock detection, the source code execution will be continuously working, causing the execution to hang without knowing the problem. Therefore, our dynamic tester will add an asynchronous directive for each compute region that has potential deadlock and number these asynchronous directives with the same number as the compute region. However, if our static tester does not detect any potential deadlock in the compute regions, our dynamic tester will not add the asynchronous directives. Our static tester will be responsible for marking each compute region that has potential deadlock and providing our dynamic tester with the appropriate information needed to proceed with the dynamic testing. Our insertion mechanism will insert test codes for testing the threads' arrival at the end of each OpenACC compute region as well as for testing the threads' arrival at the end of all regions. As shown in Figure 7, at the end of all compute regions, our tester will insert a timer and test the arrival of all threads at the end of each OpenACC compute region. If all threads arrived at the end of all OpenACC compute regions, it indicates that the source code is deadlock-free, but if the timer has ended and not all threads arrived at the end of all regions, it indicates deadlock.

During runtime and after executing the inserted test code, ACC_TEST will investigate the OpenACC compute region that caused the deadlock and provided the developers error message, which indicates the error type and its related compute region. The timer takes into consideration the sending, receiving, and execution time. This can detect the CPU deadlock that resulted from the GPU livelock. Our static phase will also analyze the source code to detect any potential livelock in the GPU from the user source code and without execution. Using the hybrid testing technique will ensure the user code correctness and detect any potential deadlock.

```
// Insert test code for testing the deadlock situation by testing the arrival of threads at the end of each compute region
// and test the arrival of all threads from all compute regions
//"ASSERT" set a timer
// test if all threads have arrived at the end of all compute regions
//"ASSERT" if(acc_async_test_all())
//"ASSERT" {
//"ASSERT" cout << " All threads have been arrived at the end of the compute region" << endl;
//"ASSERT" break;
//"ASSERT" }
// The Following is an Inserted Code To Test The Completion Of Each Compute Region -->
//"ASSERT" if (!acc_async_test(0))
//"ASSERT" {
//"ASSERT" cout << "XX ERROR :: deadlock because not all threads have arrived at the end of compute region No. ->  0" << endl;
//"ASSERT" }
//"ASSERT" if (!acc_async_test(1))
//"ASSERT" {
//"ASSERT"  cout << "ERROR deadlock because not all threads have arrived at the end of compute region No. ->  1" << endl;
//"ASSERT" }
//"ASSERT" if (!acc_async_test(2))
//"ASSERT" {
//"ASSERT"  cout << "ERROR deadlock because not all threads have arrived at the end of compute region No. ->  2" << endl;
//"ASSERT" }
//"ASSERT" if(!acc_async_test_all())
//"ASSERT" {
//"ASSERT"  cout << "ERROR deadlock Because Not all threads have arraived to the end of the compute region" << endl;
//"ASSERT" }
 //"ASSERT" cout << "XXXXX  ---> ERROR: System Timeout :: Not all threads have Arrived<---- XXXXXXX" << endl;
//"ASSERT" abort(); // To force the program to exit in the case of livelock
//"ASSERT" } // End if
```

Figure 7. Inserted test code for testing deadlock situation by testing threads arrival at the end of each compute region.

In terms of MPI error detection, our static approach starts by checking each MPI communication, whether it is point-to-point, collectives, blocking, or non-blocking. This process will determine the MPI calls' locations on the source code, rank, and communication type. In addition, our static approach has the ability to retrieve all attributes related to the MPI sending and receiving calls, including data, counter, data type, source, destination, tag, and MPI communicator as well as other information related to each type of MPI calls. Algorithm 5 shows the process of storing all related MPI calls and checking MPI data type and size mismatching as well as the MPI send/receive pair for detecting any differences between the numbers of sends and receives. This pairing will also examine the message tag to detect any unmatched message pairing. This process will avoid any potential race condition and deadlocks.

In addition, in the case of the MPI sending calls more than receiving calls, this will be reported as there will be messages sent without being received, which can cause potential errors. On the other hand, if the MPI is receiving calls more than sending calls, it will also cause potential errors, including deadlock and race condition, as there will be processes waiting for receiving messages forever. In addition, some cases of deadlock will be detected when there is any MPI_Recv call without sender, and potential deadlock will be detected when using the wild card receive. In the case of having MPI_Send without receive, it will cause lake of resource errors. In addition, when there is a data exchange between two different processes where the same process sends and receives, it can cause potential deadlock, which needs further detection during ACC_TEST dynamic phase. Finally, the receive/receive deadlock will be detected by the static phase, but the send/send deadlock needs further investigation during the dynamic phase.

ACC_TEST dynamic phase is responsible for detecting deadlocks and race conditions by using the annotation from the static phase and using the insertion mechanism for executing testing during runtime. Connections that have potential errors will be investigated during our dynamic phase, which enhances the testing time and system performance by only testing the required parts of the code and minimizing overheads. In terms of detecting deadlock in point-to-point blocking communication, the following code in Figure 8 shows the inserted codes that test MPI_Send and MPI_Recv calls.

Algorithm 5: MPI Communication Detection

START
SEARCH mpi_send calls in user code
STORE mpi_send_comm_type, mpi_send_no, mpi_send_line_no, mpi_send_data, mpi_send_count, mpi_send_data_type, mpi_send_dest, mpi_send_tag,mpi_send_comm, mpi_send_rank, mpi_send_match, recver_line_no, mpi_isend_request
SEARCH mpi_recv calls in user code
STORE mpi_recv_comm_type, mpi_recv_no, mpi_recv_line_no, mpi_recv_data, mpi_recv_count, mpi_recv_data_type, mpi_recv_source, mpi_recv_tag,mpi_recv_comm, mpi_recv_status, mpi_recv_rank, mpi_recv_match, mpi_recv_potential, mpi_irecv_request
SEARCH mpi_sendrecv calls in user code
STORE mpi_sendrecv_no, mpi_sendrecv_line_no, mpi_sendrecv_send_data, mpi_sendrecv_send_count, mpi_sendrecv_send_data_type, mpi_sendrecv_dest,mpi_sendrecv_send_tag, mpi_sendrecv_recv_data mpi_sendrecv_recv_count, mpi_sendrecv_recv_type, mpi_sendrecv_source,mpi_sendrecv_recv_tag, mpi_sendrecv_comm, mpi_sendrecv_status
SEARCH mpi_collective_comm calls in user code
STORE mpi_collective_no, mpi_collective_line_no, mpi_collective_data, mpi_collective_count, mpi_collective_data_type, mpi_collective_root, mpi_collective_comm, mpi_collective_rank
CHECK No of mpi_send_call with No of mpi_recv_call
IF No. of mpi_send_call > mpi_recv_call
    ERROR REPORTED
ELSE IF mpi_send_call < mpi_recv_call
    ERROR REPORTED
END IF
FOR each send and recv pair
    CHECK details of each send and recv pair
    IF there is a difference between data types for the same pair
        ERROR REPORTED
      ELSE IF there is a difference between data size for the same pair
        ERROR REPORTED
      ELSE IF there is a send without recv
        ERROR REPORTED
      ELSE IF there is a recv without send
        ERROR REPORTED
      ELSE IF mpi_recv_source = "MPI_ANY_SOURCE"
        ANNOTATION FOR FURTHER DYNAMIC PHASE
      ELSE IF mpi_recv_tag = "MPI_ANY_TAG"
        ANNOTATION FOR FURTHER ACC_TEST DYNAMIC PHASE
    END IF
      IF mpi_send_rank_1 = mpi_send_dest_2 && mpi_send_rank_2 = mpi_send_dest_1
        ANNOTATION FOR FURTHER ACC_TEST DYNAMIC PHASE
      ELSE IF recver_line_no_1 < mpi_send_line_no_2 && recver_line_no_2 < mpi_send_line_no_1
        ERROR REPORTED
      ELSE IF recver_line_no_1 > mpi_send_line_no_2 && recver_line_no_2 > mpi_send_line_no_1
        ERROR REPORTED
      ELSE IF mpi_send_rank_1 = mpi_send_rank_2 && mpi_send_dest_1 > mpi_send_dest_2 && mpi_send_tag_1 = mpi_send_tag_2
        ERROR REPORTED
END FOR
END

```
// Insert test code for testing point-to-point blocking communication (MPI_Send/MPI_Recv)
// THE FOLLOWING LINE IS HAVE THE INSERTED TESTING CODE AFTER THE MPI_RECV ****
//"ASSERT" MPI_Request request_var_name;
//"ASSERT" MPI_Status status_var_name;
//"ASSERT" int flag_var_name;
//"ASSERT" MPI_Irecv(Buffer, Count, Data_Type, Source, Tag, MPI_Communicator, request_var_name);
//"ASSERT" Set a timer
//"ASSERT" MPI_Test(&request_var_name, &flag_var_name, &status_var_name);
//"ASSERT" if (!flag_var_name)
//"ASSERT" {
//"ASSERT" cout << "ERROR :: ==>> THERE IS A DEADLOCK IN THE MPI_RECV No: " <<
MPI_Recv_ID << "IN RANK NO:" << Rank << "IN LINE NO:" << Line_no << endl;
//"ASSERT" }
//"ASSERT" else
//"ASSERT" {
//"ASSERT" cout << "The Tested MPI_RECV NO:" << MPI_RECV_ID << " is Completed" << endl;
//"ASSERT" cout << "The Flag Value: " << flag_var_name << endl;
//"ASSERT" cout << "Received From Process No: " << status_var_name.MPI_SOURCE << endl;
//"ASSERT" cout << "With TAG No: " << status_var_name.MPI_TAG << endl;
//"ASSERT" }
```

Figure 8. Test code for point-to-point blocking deadlock.

For detecting race condition, ACC_TEST will compare all received calls' actual information with our static phase information to detect any potential race conditions, as displayed in Figure 9, where some values will be inserted and compared with the result from actual values. Similarly, to test the MPI_sendrecv, ACC_TEST will split this connection into MPI_Send and MPI_Recv and test them individually, as shown in Figure 10.

```
// Insert test code for testing race condition in point-to-point blocking communication (MPI_Send/MPI_Recv)
//"ASSERT" if (status_var_name.MPI_TAG != Static_Tag_Value | status_var_name.MPI_SOURCE !=
Static_Source_Value)
//"ASSERT" {
//"ASSERT" cout << " XXX MPI BLOCING COMMUNICATION ERROR RACE CONDITION XXX" << endl;
//"ASSERT" cout << "XX ERROR : In Line " << Line_no << " THERE IS A RACE CONDITION IN THE
MPI_RECV No: " << MPI_RECV_ID  << endl;
//"ASSERT" cout << "IN RANK NO:" << rank << " ECAUSE THE MPI_RECV No." << MPI_RECV_ID <<
"RECEIVED FROM RANK: "<< rank <<" WITH TAG: " << Tag << endl;
//"ASSERT" cout << " BUT IT SHOULD BE RECEIVED FROM RANK:" << Static_Source_Value << "WITH
TAG:" << Static_Tag_Value << endl;
//"ASSERT" }
```

Figure 9. Test code for point-to-point blocking race condition.

In the case of collective communication, MPI_Bcast be will examined during runtime to detect any deadlock by inserting some testing codes as displayed in Figure 11 for testing the data exchange between different MPI broadcasts. The annotation has been used for replacing each MPI blocking broadcast (MPI_Bcast) with MPI non-blocking broadcast (MPI_Ibcast) for avoiding any runtime blocking behavior and testing the arrival of all broadcasts calls and to compare actual information with the tested broadcast calls.

In terms of the used instrumentation method, we add the testing codes to the source code, because by adding the testing code to the user code, the testing will be distributed, increasing the reliability of our testing tool and avoiding the single point of failure, which happens when using the second method of having centralized control and call function for testing. In our case, reliability is more important than a smaller size. In addition, the testing code will be used in the testing house. In the operation

phase, the user has the choice to use the uninstrumented source code, which will allow the compiler to ignore the test code, and the code size will not affect the operational user code. In addition, by using the chosen method, we will enhance performance by having distributed testing codes rather than centralized control.

```
// Insert test code for testing point-to-point blocking communication (MPI_Sendrecv)
// THE FOLLOWING LINE IS HAVE THE INSERTED TESTING CODE AFTER THE MPI_SENFRECV ****
//"ASSERT" MPI_Request request_sendrecv_var_name;
//"ASSERT" MPI_Status status_sendrecv_var_name;
//"ASSERT" int flag_sendrecv_var_name;
// THE FOLLOWING LINE IS HAVE THE MPI_IRECV INSERTED CODE ****
//"ASSERT" MPI_Irecv(Buffer, Count, Data_Type, Source, Tag, MPI_Communicator, request_sendrecv_var_name);
// THE FOLLOWING LINE IS HAVE THE MPI_SEND INSERTED CODE ****
//"ASSERT" MPI_Send(Buffer, Count, Data_Type, Destination, Tag, MPI_Communicator);
//"ASSERT" Set a Timer
//"ASSERT" MPI_Test(&request_sendrecv_var_name, &flag_sendrecv_var_name, &status_sendrecv_var_name);
//"ASSERT" if (!flag_sendrecv_var_name)
//"ASSERT" {
//"ASSERT" cout << "ERROR :: ==>> THERE IS A DEADLOCK IN THE MPI_SENDRECV No: " <<
MPI_Recv_ID << "IN RANK NO:" << Rank << "IN LINE NO:" << Line_no << endl;
//"ASSERT" }
//"ASSERT" else
//"ASSERT" {
//"ASSERT" cout << "The Tested MPI_SENDRECV NO: << MPI_SENDRECV_ID << "is Completed" << endl;
//"ASSERT" cout << "The Flag Value: " << flag_sendrecv_var_name << endl;
//"ASSERT" cout << "Received From Process No: " << status_sendrecv_var_name.MPI_SOURCE << endl;
//"ASSERT" cout << "With TAG No: " << status_sendrecv_var_name.MPI_TAG << endl;
//"ASSERT" }
```

Figure 10. Test code for point-to-point blocking deadlock.

```
// Insert test code for testing collective communication (MPI_Bcast)
// THE FOLLOWING LINE IS HAVE THE INSERTED TESTING CODE AFTER THE MPI_Bcast ****
//"ASSERT" MPI_Request request_Bcast_var_name;
//"ASSERT" MPI_Status status_Bcast_var_name;
//"ASSERT" int flag_bcast_var_name;
//"ASSERT" MPI_Ibcast(&data, 1, MPI_INT, 0, MPI_COMM_WORLD, &Bc_request_2);
//"ASSERT" Set a Timer
//"ASSERT" MPI_Test(&Bc_request_2, &Bc_flag_2, &Bc_status_2);
//"ASSERT" if (!Bc_flag_2)
//"ASSERT" {
//"ASSERT" cout << "ERROR :: ==>> THERE IS A DEADLOCK IN THE MPI_Bcast No:2 IN RANK NO: 2 IN
LINE NO: 32 " << endl;
//"ASSERT" }
//"ASSERT" else
//"ASSERT" {
//"ASSERT" cout << "The Tested MPI_Bcsat NO: 2 is Completed" << endl;
//"ASSERT" cout << "The Flag Value: " << Bc_flag_2 << endl;
//"ASSERT" cout << "Received From The Root No: " << Bc_status_2.MPI_SOURCE << endl;
//"ASSERT" }
```

Figure 11. Test code for collective communication.

ACC_TEST collected some historical information during the dynamic and static phases and stored it in a log file for use in debugging and tracking changes in user code. The runtime errors detected by our dynamic phase will be stored in a report separate from the report for the static analysis detection. After the execution of our testing for MPI and OpenACC-related programs, ACC_TEST will create a historical log file including several useful types of information to be used in debugging. This file will include the following information related to MPI + OpenACC programs:

(1) Summary of OpenACC regions: total number of compute, structured, and unstructured data regions, as well as the starting and ending points for each region;
(2) Compute region variables information;
(3) Loops information;
(4) Equations information;

(5) Equation data race analytics information;
(6) Equations read/write race condition analytics information;
(7) Thread information gang/vector;
(8) MPI-blocking communication information, including information about MPI_Send, MPI_Recv, and MPI_Sendrecv;
(9) MPI non-blocking communication information, which has information about MPI_Isend and MPI_Irecv;
(10) MPI collective communication information;
(11) MPI rank information.

Finally, our testing tool will produce several files reporting errors detected in our static and dynamic phases. In summary, our testing tool outputs will be:

(1) Inserted source code, including user codes and uninstrumented test codes;
(2) Static errors report, including information of errors detected during our static phase;
(3) Dynamic errors report, including information related to errors detected during our dynamic phase;
(4) Historical log file, including information from our static analysis;
(5) Historical log file for our dynamic analysis.

ACC_TEST has the ability to detect MPI + OpenACC hybrid-based programs and to detect OpenACC-based programs and MPI-based programs individually.

4. Discussion and Evaluation

Our tool has been implemented and tested for verifying and validating the ACC_TEST. Several experiments have been conducted, covering several scenarios and test suites for testing our proposed solution and ensuring ACC_TEST's capability to detect different types of errors in MPI, OpenACC, and MPI + OpenACC dual-programming models. Because of the lack of the dual-programming models MPI + OpenACC benchmarks, we created our own hybrid programming models' test suites for evaluating our testing techniques as well as the error coverage. We built several test cases for testing our proposed techniques and our testing tool, as shown in Table 1. Because of the lack of MPI + OpenACC benchmarks, we created our own hybrid programming models test suites for evaluating our testing techniques as well as the error coverage. These test suites include both OpenACC and MPI directives for building parallel programs using the dual-programming models MPI + OpenACC. We built these test suites for evaluating our hybrid-testing tool and examining its ability to cover the runtime errors that we targeted and to measure different overheads, including size, compilation, and execution overheads.

Table 1. Our test cases for evaluating ACC_TEST.

Test Case	Description
MPI_ACC_BC	OpenACC + MPI Blocking Communication Error-Free Code
MPI_ACC_BPD	OpenACC + MPI Blocking Communication Potential Deadlock
MPI_ACC_NPR	OpenACC + MPI Non-Blocking Communication Potential Race
MPI_ACC_NPD	OpenACC + MPI Non-Blocking Communication Potential Deadlock
MPI_ACC_BRD	OpenACC + MPI Blocking Communication Real Deadlock
MPI_ACC_NTM	OpenACC + MPI Non-Blocking Communication Tag Mismatching
MPI_ACC_BRL	OpenACC + MPI Blocking Communication Resource Leak
MPI_ACC_BPR	OpenACC + MPI Blocking Communication Potential Race
MPI_ACC_BCD	OpenACC + MPI Blocking Collective Communication Deadlock
MPI_ACC_BCR	OpenACC + MPI Blocking Collective Communication Race Condition

Table 2 shows our hybrid testing tool's ability to detect errors, which occur in MPI + OpenACC dual-programming models. We collected all errors that can be identified by our OpenACC and MPI testers and examine them on our hybrid testing tool for the dual-programming model. We found that our integrated tool could detect all errors targeted.

Table 2. ACC_TEST error coverage for MPI + OpenACC dual-programming model.

Errors	Detected by Our MPI Tester *	Detected by Our OpenACC Tester *	Detected by Our Integrated Tester *
OpenACC Data Clause Related Errors			
Create Clause	X	√	√
Copy Clause	X	√	√
Copyin Clause	X	√	√
Copyout Clause	X	√	√
Delete Clause	X	√	√
Present Clause	X	√	√
Memory Errors	X	√	√
Data Transmission Errors	X	√	√
OpenACC Race Conditions			
Host/Device Synchronization	X	√	√
Loop Parallelization Race	X	√	√
Data Dependency Race	X	√	√
Data Read and Write Race	X	√	√
Asynchronous Directives Race	X	√	√
Reduction Clause Race	X	√	√
Independent Clause Race	X	√	√
OpenACC Deadlock			
Device Deadlock	X	√	√
Host Deadlock	X	√	√
Livelock	X	√	√
MPI Point-to-point Blocking/Non-blocking Communication			
Illegal MPI Calls	√	X	√
Data Type Mismatching	√	X	√
Data Size Mismatching	√	X	√
Resource Lacks	√	X	√
Request Lost	√	X	√
Inconsistence Send/Recv Pairs	√	X	√
Wildcard Receive	√	X	√
Race Condition	√	X	√
Deadlock	√	X	√
MPI Collective Blocking/Non-blocking Communication			
Illegal MPI Calls	√	X	√
Data Type Mismatching	√	X	√
Data Size Mismatching	√	X	√
Incorrect Ordered Collective Operation	√	X	√
Race Condition	√	X	√
Deadlock	√	X	√

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

In the following, Figure 12 displays the number of detected errors detected by our tool, including the number of errors detected by our static and dynamic approaches.

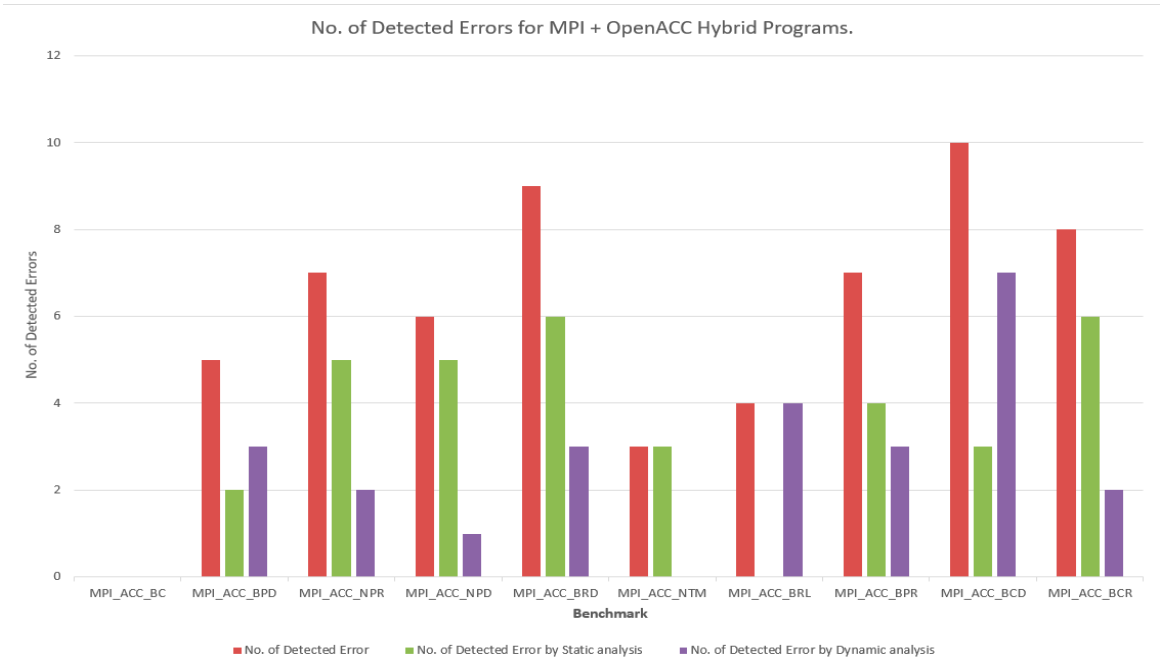

Figure 12. No. of detected errors for MPI + OpenACC hybrid programs.

In terms of size overheads, we used Equation (1) for measuring the size overhead, as shown in Figure 13. We noted that the size overheads range between 79% and 135%, based on the nature of the source code and its behavior. However, these size overheads will not affect the user code because all the inserted statements will be considered by the compiler as a comment, as they all start with the comment character. These inserted statements will affect the user source code only on the testing house and when these codes pass our instrumenter. We believe that these size overheads are needed because of the nature of runtime errors, which cannot be detected during our static phase analysis and need to be tested during runtime:

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

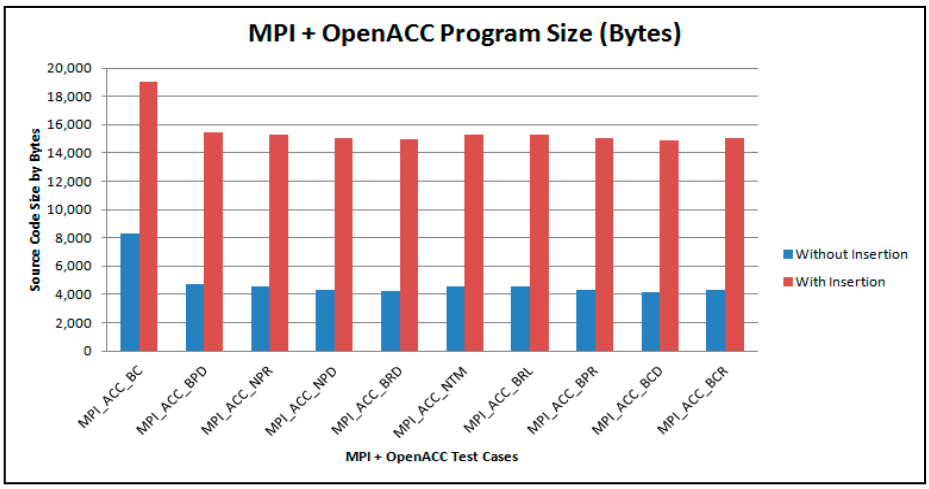

Figure 13. MPI + OpenACC hybrid program size overhead (by bytes).

In terms of compilation and execution times, we measure our test cases and compare the compilation and execution times before and after our insertion process. Figure 14 shows the average compilation time in milliseconds, which is 198 milliseconds before insertion and 230 milliseconds after insertion.

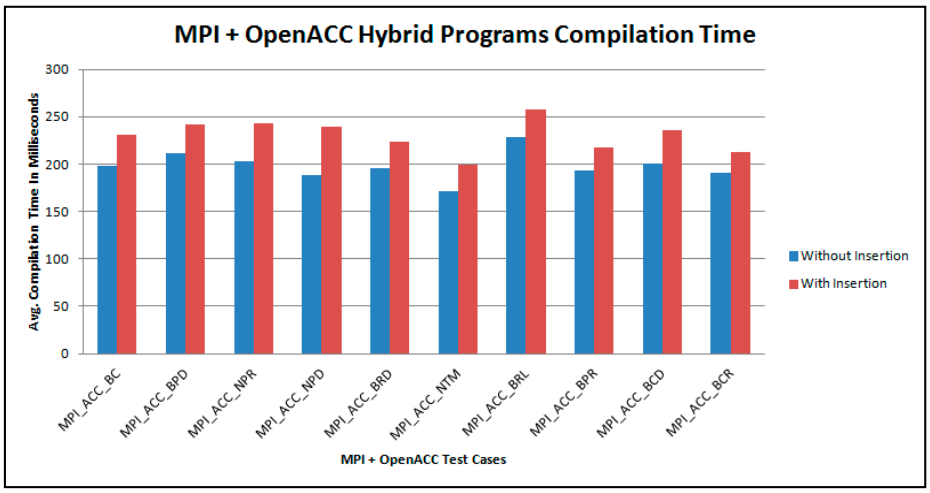

Figure 14. MPI + OpenACC hybrid program compilation time.

In terms of the execution time, there are differences in the execution time based on the number of processes. The average time before insertion ranges between 230 milliseconds in the 4 processes and 1578 milliseconds in the 128 processes. After the insertion, the execution time ranges between 241 and 1765 milliseconds for the 4 and 128 processes, respectively.

Compilation Time (CT) or Execution Time (ET) overheads will be calculated using the following Equation (2):

$$CT/ET\ Overheads = \frac{T\ with\ inserted\ test\ code - T\ without\ inserted\ test\ code}{T\ without\ inserted\ test\ code}. \quad (2)$$

Figure 15 shows the compilation time overhead resulting from ACC_TEST. The compilation overheads range from 12% to 27%, which is considered acceptable and can vary based on the used compiler and system behavior. In terms of measuring the overheads and their relation to the number of MPI processes, Figure 16 shows the execution time overhead for MPI + OpenACC-related test cases.

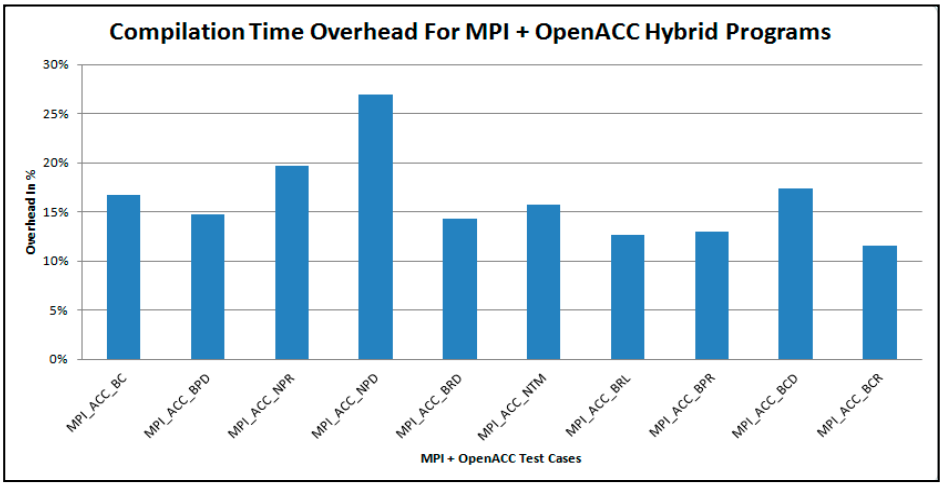

Figure 15. Compilation time overheads for MPI + OpenACC hybrid programs.

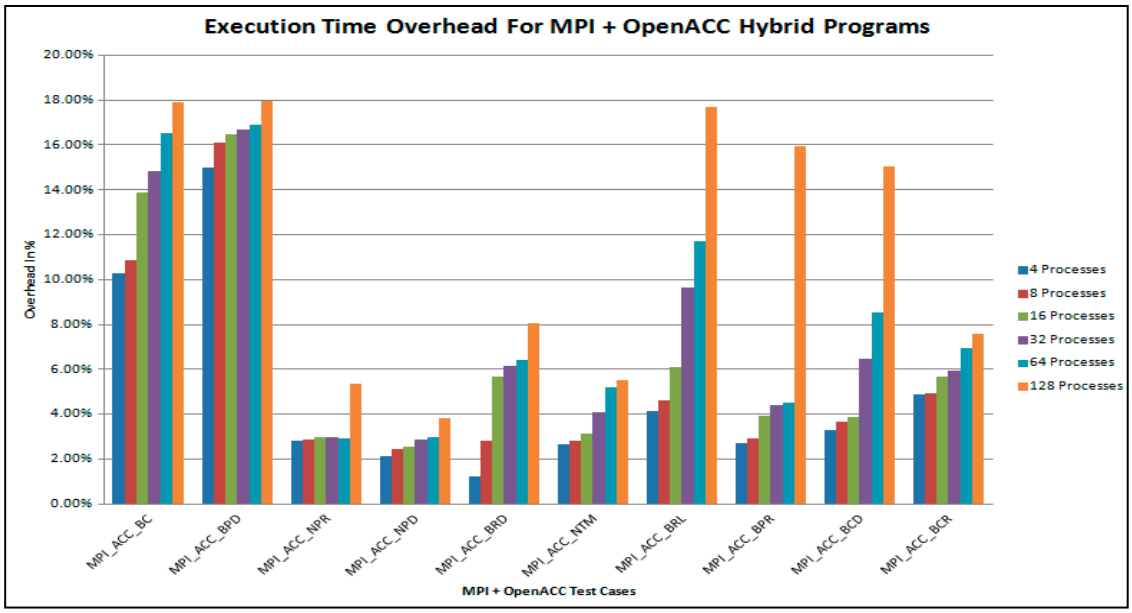

Figure 16. Execution time overheads for MPI + OpenACC hybrid programs.

We successfully minimized the execution overheads to be under 18%; the execution overheads range between 1% and below 18% based on the system behavior, the machine status, and the number of processes.

Finally, Figure 17 shows the execution time for testing programs built using MPI + OpenACC dual-programming models, for which the average testing time is 115 milliseconds.

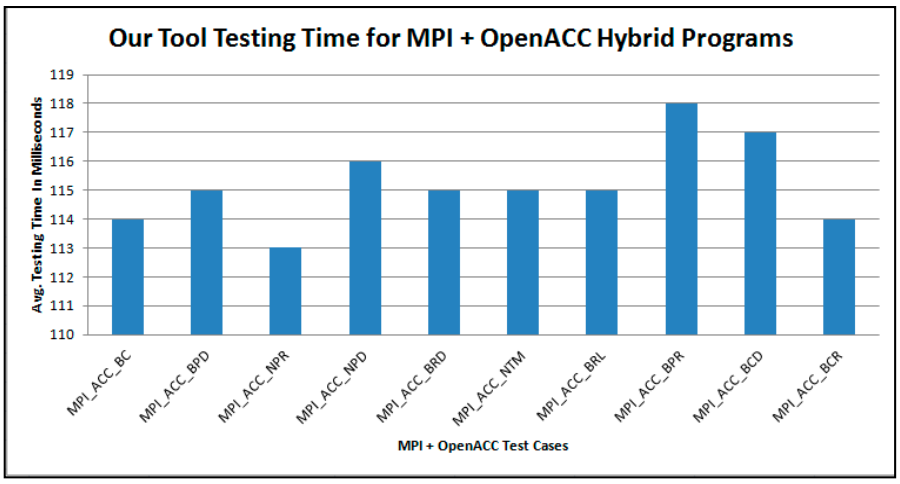

Figure 17. Testing time for MPI + OpenACC hybrid programs.

In comparison to the MPI testing techniques published in [47], our testing tool minimizes the size overhead for testing MPI-related programs because we avoid adding unnecessary messaging (communications) to test the connection between senders and receivers to detect deadlock. Another main advantage of ACC_TEST is that we used distributed testing techniques, unlike some other tools [48,49], which used a centralized manager for detecting MPI-related errors, causing a single point of failure and single point of attack.

OpenACC has been used for building ACC_TEST, which makes it portable and hardware architecture-independent. It therefore works with any type of GPU accelerator, hardware, platform, and operating system. In addition, ACC_TEST is easy to maintain and requires less effort because of the high maintainability of OpenACC. Our insertion techniques help increase the reliability of

ACC_TEST because this technique avoids centralized control and single-point-of-failure problems as well as increase performance by distributing our testing tasks and avoiding centralized controlled testing. ACC_TEST also helps produce more high-quality systems without errors.

In Table 3, we summarize the comparative study conducted in our research. Because there is no published work or existing testing tool that detects OpenACC errors or the dual-programming model MPI + OpenACC, we chose the closest work to compare with our techniques in different attribute.

Table 3. Comparative study summary.

Attributes	MARMOT	MUST	PARCOACH	UMPIRE	GMRace	ACC_TEST
Capability	Deadlock Race Condition Mismatching	Deadlock Mismatching	Deadlock Mismatching Data Race	Deadlock Mismatching	Data Race	Comprehensive
Testing Technique	Dynamic	Dynamic	Hybrid	Dynamic	Hybrid	Hybrid
Distribution	Centralized	Centralized	Centralized	Centralized	Centralized	Distributed
Programming model supported by the tool	MPI + OpenMP	MPI	MPI + OpenMP	MPI	CUDA	MPI + OpenACC
Scalability	Limited	√	√	Limited	Limited	√
Adaptability	Limited	√	Limited	Limited	Limited	√

ACC_TEST has the capability to cover different types of errors in OpenACC, MPI, and dual-programming models. In addition, ACC_TEST used hybrid-testing techniques for covering a wide range of errors while minimizing overheads. ACC_TEST is built based on a distributed mechanism, which avoids single point of failures. Additionally, the dual-programming models MPI + OpenACC have been supported by ACC_TEST for the first time in the research field. ACC_TEST is scalable and adaptable and can run on any platform.

5. Conclusions and Future Work

To conclude, good effort has been made in testing parallel systems to detect runtime errors, but it still insufficient, especially for systems using heterogeneous programming models as well as dual- and tri-level programming models. The integration of different programming models into the same system will need new testing techniques to detect runtime errors in heterogeneous parallel systems. We believe that in order to achieve good systems that can be used in Exascale supercomputers, we should focus on testing those systems because of their massively parallel natures as well as their huge size, which increases difficulties and issues. We enhanced the capabilities of ACC_TEST to detect errors occurring in the hybrid dual-programming models MPI + OpenACC. In addition, ACC_TEST works in parallel by detecting runtime errors with testing threads created based on the targeted application threads. Additionally, our tool can work with any heterogeneous architecture, which will increase the portability of ACC_TEST. We have implemented our solution and evaluated its ability to detect runtime errors. Using our parallel hybrid testing techniques will yield benefits, such as reduction of overhead, enhanced system execution time, and coverage of a wide range of errors.

MPI + OpenACC applications errors have been successfully detected by using ACC_TEST hybrid-testing techniques. Helping to increase reliability and ensure error-free codes are the main objectives of creating our testing tool. Our tool also achieves covering range of errors with execution overhead in an acceptable level, which is less than 20%. Because of using the testing techniques that provide overheads, the testing processes will be used only in the testing house and will not affect the delivered user applications. Finally, to the best of our knowledge, ACC_TEST is the first parallel testing tool built to test applications programmed by using the dual-programming model MPI + OpenACC. ACC_TEST focuses only on the integration of MPI and OpenACC and the resulting errors that can occur in this integration as well as in MPI and OpenACC individually.

In future works, ACC_TEST will be enhanced to cover errors that occur in tri-programming models MPI + OpenACC + X. In addition, we will enhance our techniques to run on real Exascale systems when they become available. Finally, we will enhance ACC_TEST to be intelligent using AI techniques for generating test cases automatically and using deep learning while testing.

Data Availability: The data used to support the findings of this study are available from the corresponding author upon request.

Author Contributions: Conceptualization, Data curation, Formal analysis, Methodology, Resources, Funding acquisition, Software, Visualization and Writing—original draft: A.M.A.; Methodology, Resources, Supervision, Validation and Formal analysis: F.E.E.; Methodology, Project administration, Validation and Formal analysis: M.A.K.; Methodology, Project administration, Resources, Supervision, and Validation: A.S.A.-M.A.-G.; Software, Supervision, Validation, Writing—review & editing and Formal analysis: A.S.A. (Ahmed S. Alfakeeh); Formal analysis, Visualization, Validation, Writing—review & editing: A.S.A (Abdullah S. Alshahrani); Formal analysis, Visualization, Validation, Writing—review & editing: A.A.A. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by [King Abdulaziz University] grant number [RG-9-611-40] and the APC was funded by [Deanship of Scientific Research].

Acknowledgments: This project was funded by the Deanship of Scientific Research (DSR) at King Abdulaziz University, Jeddah, under Grant no. (RG-9-611-40). The authors, therefore, acknowledge with thanks the DSR's technical and financial support.

Conflicts of Interest: The authors declare that they have no conflicts of interest.

References

1. Alghamdi, A.M.; Elbouraey, F. A Parallel Hybrid-Testing Tool Architecture for a Dual-Programming Model. *Int. J. Adv. Comput. Sci. Appl.* **2019**, *10*, 394–400. [CrossRef]
2. Alghamdi, A.M.; Eassa, F.E. Proposed Architecture for a Parallel Hybrid-Testing Tool for a Dual-Programming Model. *IJCSNS Int. J. Comput. Sci. Netw. Secur.* **2019**, *19*, 54–61.
3. Alghamdi, A.M.; Eassa, F.E. Parallel Hybrid Testing Tool for Applications Developed by Using MPI + OpenACC Dual-Programming Model. *Adv. Sci. Technol. Eng. Syst. J.* **2019**, *4*, 203–210. [CrossRef]
4. Alghamdi, A.M.; Eassa, F.E. OpenACC Errors Classification and Static Detection Techniques. *IEEE Access* **2019**, *7*, 113235–113253. [CrossRef]
5. Eassa, F.E.; Alghamdi, A.M.; Haridi, S.; Khemakhem, M.A.; Al-Ghamdi, A.S.A.-M.; Alsolami, E.A. ACC_TEST: Hybrid Testing Approach for OpenACC-Based Programs. *IEEE Access* **2020**, *8*, 80358–80368. [CrossRef]
6. Alghamdi, A.S.A.; Alghamdi, A.M.; Eassa, F.E.; Khemakhem, M.A. ACC_TEST: Hybrid Testing Techniques for MPI-Based Programs. *IEEE Access* **2020**, *8*, 91488–91500. [CrossRef]
7. Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*; University of Tennessee: Knoxville, Tennesee, 2015.
8. Ashraf, M.U.; Fouz, F.; Eassa, F.A. Empirical Analysis of HPC Using Different Programming Models. *Int. J. Mod. Educ. Comput. Sci.* **2016**, *8*, 27–34. [CrossRef]
9. Ashraf, M.U.; Eassa, F.A.; Albeshri, A.A.; Algarni, A. Performance and Power Efficient Massive Parallel Computational Model for HPC Heterogeneous Exascale Systems. *IEEE Access* **2018**, *6*, 23095–23107. [CrossRef]
10. Kim, J.; Lee, S.; Vetter, J.S. IMPACC: A Tightly Integrated MPI+OpenACC Framework Exploiting Shared Memory Parallelism. In Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing—HPDC'16, Kyoto, Japan, 31 May–4 June 2016; pp. 189–201. [CrossRef]
11. Message Passing Interface Forum. MPI Forum. 2017. Available online: http://mpi-forum.org/docs (accessed on 19 September 2019).
12. OpenACC Standards Org. The OpenACC Application Programming Interface Version 3.0. *Adv. Sci. Technol. Eng. Syst. J.* **2019**, *4.2*, 203–210.
13. Alghamdi, A.M.; Eassa, F.E. Software Testing Techniques for Parallel Systems: A Survey. *IJCSNS Int. J. Comput. Sci. Netw. Secur.* **2019**, *19*, 176–186.
14. Chatarasi, P.; Shirako, J.; Kong, M.; Sarkar, V. An Extended Polyhedral Model for SPMD Programs and Its Use in Static Data Race Detection. In Proceedings of the 23rd International Workshop on Languages and Compilers for Parallel Computing, Houston, TX, USA, 7–9 September 2010; Springer: Cham, Switzerland, 2017; pp. 106–120.
15. Bardsley, E.; Betts, A.; Chong, N.; Collingbourne, P.; Deligiannis, P.; Donaldson, A.F.; Ketema, J.; Liew, D.; Qadeer, S. Engineering a Static Verification Tool for GPU Kernels. In *International Conference on Computer Aided Verification*; Springer: Cham, Switzerland, 2014; pp. 226–242. [CrossRef]

16. Nakade, R.; Mercer, E.; Aldous, P.; McCarthy, J. Model-Checking Task Parallel Programs for Data-Race. *Innov. Syst. Softw. Eng.* **2019**, *15*, 289–306. [CrossRef]
17. Basupalli, V.; Yuki, T.; Rajopadhye, S.; Morvan, A.; Derrien, S.; Quinton, P.; Wonnacott, D. ompVerify: Polyhedral Analysis for the OpenMP Programmer. In *International Workshop on OpenMP*; Springer: Berlin/Heidelberg, Germany, 2011; pp. 37–53. [CrossRef]
18. Droste, A.; Kuhn, M.; Ludwig, T. MPI-Checker. In Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC, Austin, TX, USA, 15 November 2015; pp. 1–10. [CrossRef]
19. The Open MPI Organization. Open MPI: Open Source High Performance Computing. 2018. Available online: https://www.open-mpi.org/ (accessed on 19 September 2019).
20. RWTH Aachen University. MUST: MPI Runtime Error Detection Tool. *JCSNS Int. J. Comput. Sci. Netw. Secur.* **2018**, *19*, 54–61.
21. Arnold, D.C.; Ahn, D.H.; de Supinski, B.R.; Lee, G.L.; Miller, B.P.; Schulz, M. Stack Trace Analysis for Large Scale Debugging. In Proceedings of the 2007 IEEE International Parallel and Distributed Processing Symposium, Rome, Italy, 26–30 March 2007; pp. 1–10. [CrossRef]
22. Kowalewski, R.; Fürlinger, K. Nasty-MPI: Debugging Synchronization Errors in MPI-3 One-Sided Applications. In *European Conference on Parallel Processing Euro-Par 2016*; Springer: Cham, Switzerland, 2016; pp. 51–62. [CrossRef]
23. Samofalov, V.; Krukov, V.; Kuhn, B.; Zheltov, S.; Konovalov, A.; DeSouza, J. Automated Correctness Analysis of MPI Programs with Intel Message Checker. In Proceedings of the International Conference ParCo, Malaga, Spain, 13–16 September 2005; pp. 901–908.
24. Luecke, G.; Chen, H.; Coyle, J.; Hoekstra, J.; Kraeva, M.; Zou, Y. MPI-CHECK: A Tool for Checking Fortran 90 MPI Programs. *Concurr. Comput. Pract. Exp.* **2003**, *15*, 93–100. [CrossRef]
25. Humphrey, A.; Derrick, C.; Gopalakrishnan, G.; Tibbitts, B. GEM: Graphical Explorer of MPI Programs. In Proceedings of the 39th International Conference on Parallel Processing Workshops, San Diego, CA, USA, 13–16 September 2010. Available online: http://ieeexplore.ieee.org/document/5599207/ (accessed on 19 September 2019).
26. Ganai, M.K. Dynamic Livelock Analysis of Multi-threaded Programs. In *International Conference on Runtime Verification*; Springer: Berlin/Heidelberg, Germany, 2013; pp. 3–18.
27. Clemencon, C.; Fritscher, J.; Ruhl, R. Visualization, Execution Control and Replay of Massively Parallel Programs within Annai's Debugging Tool. *High-Perform. Comput. Symp. (HPCS'95)* **1995**, 393–404. [CrossRef]
28. Kranzlmueller, D.; Schaubschlaeger, C.; Volkert, J. A Brief Overview of the MAD Debugging Activities. In Proceedings of the Fourth International Workshop on Automated Debugging, Ithaca, NY, USA, 16 December 2000.
29. Do-Mai, A.T.; Diep, T.D.; Thoai, N. Race condition and deadlock detection for large-scale applications. In Proceedings of the 15th International Symposium on Parallel and Distributed Computing, ISPDC, Fuzhou, China, 8–10 July 2016; pp. 319–326. [CrossRef]
30. Forejt, V.; Joshi, S.; Kroening, D.; Narayanaswamy, G.; Sharma, S. Precise Predictive Analysis for Discovering Communication Deadlocks in MPI Programs. *ACM Trans. Program. Lang. Syst.* **2017**, *39*, 1–27. [CrossRef]
31. Gopalakrishnan, G.; Kirby, R.M.; Vakkalanka, S.; Vo, A.; Yang, Y. ISP (In-situ Partial Order): A dynamic verifier for MPI Programs. University of Utah, School of Computing. 2009. Available online: http://formalverification.cs.utah.edu/ISP-release/ (accessed on 19 September 2019).
32. Park, M.-Y.; Shim, S.J.; Jun, Y.-K.; Park, H.-R. MPIRace-Check: Detection of Message Races in MPI Programs. In *International Conference on Grid and Pervasive Computing GPC*; Springer: Berlin/Heidelberg, Germany, 2007; pp. 322–333. [CrossRef]
33. Boyer, M.; Skadron, K.; Weimer, W. Automated Dynamic Analysis of CUDA Programs. In Proceedings of the Third Workshop on Software Tools for MultiCore Systems (STMCS), Boston, MA, USA, 6 April 2008.
34. Mekkat, V.; Holey, A.; Zhai, A. Accelerating Data Race Detection Utilizing On-Chip Data-Parallel Cores. In *International Conference on Runtime Verification*; Springer: Berlin/Heidelberg, Germany, 2013; pp. 201–218. [CrossRef]
35. Gupta, S.; Sultan, F.; Cadambi, S.; Ivancic, F.; Rotteler, M. Using hardware transactional memory for data race detection. In Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing, Rome, Italy, 23–29 May 2009; pp. 1–11. [CrossRef]

36. Bekar, C.; Elmas, T.; Okur, S.; Tasiran, S. KUDA: GPU accelerated split race checker. In Proceedings of the Workshop on Determinism and Correctness in Parallel Programming (WoDet), Istanbul, Turkey, 6 March 2012.
37. Sharma, R.; Bauer, M.; Aiken, A. Verification of producer-consumer synchronization in GPU programs. *ACM Sigplan Not.* **2015**, *50*, 88–98. [CrossRef]
38. Hilbrich, T.; Müller, M.S.; Krammer, B. MPI Correctness Checking for OpenMP/MPI Applications. *Int. J. Parallel Program.* **2009**, *37*, 277–291. [CrossRef]
39. Lawrence Livermore National Laboratory, University of Utah, and RWTH Aachen University. Archer. GitHub 2018. Available online: https://github.com/PRUNERS/archer (accessed on 19 September 2019).
40. Hernandez, O.; Liao, C.; Chapman, B. Dragon: A Static and Dynamic Tool for OpenMP. In *International Workshop on OpenMP Applications and Tools*; Springer: Berlin/Heidelberg, Germany, 2004; pp. 53–66. [CrossRef]
41. Zheng, M.; Ravi, V.T.; Qin, F.; Agrawal, G. GMRace: Detecting Data Races in GPU Programs via a Low-Overhead Scheme. *IEEE Trans. Parallel Distrib. Syst.* **2014**, *25*, 104–115. [CrossRef]
42. Dai, Z.; Zhang, Z.; Wang, H.; Li, Y.; Zhang, W. Parallelized Race Detection Based on GPU Architecture. In *Advanced Computer Architecture. Communications in Computer and Information Science*; Springer: Berlin/Heidelberg, Germany, 2014; pp. 113–127.
43. Saillard, E.; Carribault, P.; Barthou, D. PARCOACH: Combining static and dynamic validation of MPI collective communications. *Int. J. High Perform. Comput. Appl.* **2014**, *28*, 425–434. [CrossRef]
44. Ma, H.; Wang, L.; Krishnamoorthy, K. Detecting Thread-Safety Violations in Hybrid OpenMP/MPI Programs. In Proceedings of the 2015 IEEE International Conference on Cluster Computing, Chicago, IL, USA, 8–11 September 2015; pp. 460–463. [CrossRef]
45. Ahmad, K.; Wolfe, M. Automatic Testing of OpenACC Applications. In *4th International Workshop on Accelerator Programming Using Directives*; Springer: Cham, Switzerland, 2018; pp. 145–159.
46. Krammer, B.; Resch, M.M. Runtime Checking of MPI Applications with MARMOT. *Perform. Comput.* **2006**, *33*, 1–8.
47. Luecke, G.R.; Zou, Y.; Coyle, J.; Hoekstra, J.; Kraeva, M. Deadlock detection in MPI programs. *Concurr. Comput. Pract. Exp.* **2002**, *14*, 911–932. [CrossRef]
48. Krammer, B.; Resch, M.M. Correctness Checking of MPI One-Sided Communication Using Marmot. In Proceedings of the European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting, Bonn, Germany, 17–20 September 2006; pp. 105–114. [CrossRef]
49. Hilbrich, T.; Protze, J.; Schulz, M.; de Supinski, B.R.; Müller, M.S. MPI runtime error detection with MUST: Advances in deadlock detection. *Sci. Program.* **2013**, *21*, 109–121. [CrossRef]