A Parallel Hybrid-Testing Tool Architecture for a Dual-Programming Model

Ahmed Mohammed Alghamdi[1] , Fathy Elbouraey Eassa[2]
Faculty of Computing and Information Technology[1,2]
King Abdulaziz University, Jeddah, Saudi Arabia

***Abstract*—High-Performance Computing (HPC) recently has become important in several sectors, including the scientific and manufacturing fields. The continuous growth in building more powerful super machines has become noticeable, and the Exascale supercomputer will be feasible in the next few years. As a result, building massively parallel systems becomes even more important to keep up with the upcoming Exascale-related technologies. For building such systems, a combination of programming models is needed to increase the system's parallelism, especially dual and tri-level programming models to increase parallelism in heterogeneous systems that include CPUs and GPUs. There are several combinations of the dual-programming model; one of them is MPI+ OpenACC. This combination has several features that increase the application's parallelism concerning heterogeneous architecture and support different platforms with more performance, productivity, and programmability. However, building systems with different programming models are error-prone and difficult and are also hard to test. Also, testing parallel applications is already a difficult task because parallel errors are hard to detect due to the non-determined behavior of the parallel application. Integrating more than one programming model inside the same application makes even it more difficult to test because this integration could come with a new type of errors. Our main contribution is to identify and categorize OpenACC run-time errors and determine their causes with a brief explanation for the first time in research. Also, we proposed a solution for detecting run-time errors in application implemented in the dual-programming model. Our solution based on using hybrid testing techniques to discover real and potential run-time errors. Finally, to the best of our knowledge, there is no parallel testing tool built to test applications programmed by using the dual-programming model MPI + OpenACC or any tri-level programming model or even the OpenACC programming model to detect their run-time errors. Also, OpenACC errors have not been classified or identified before.**

Keywords*—Software testing; OpenACC run-time error classifications; hybrid testing tool; dual-programming model; OpenACC

I. INTRODUCTION

In the past few years, building Exascale systems based on CPU/GPU heterogeneous architecture has become a hot research topic. Therefore, creating parallel applications with more ability to work with the upcoming Exascale era has also become increasingly important. However, the current parallel programming languages are not satisfying the increasing need for creating parallel applications. Also, parallelism cannot be supported efficiently by the majority of traditional programming languages. Therefore, programming models, which are a group of directives and operations used to support parallelism, have been used to add parallelism to the traditional programming languages.

There are several programming models with different features and used for various purposes. These programming models include the message passing programming model MPI [1] and the programming model that support the shared-memory parallelism such as OpenMP [2]. In addition, there are programming models that support CPU/GPU heterogeneous systems. The programming models that support heterogeneous parallelism are CUDA [3] and OpenCL [4], as low-level programming models, and OpenACC [5] as a high-level heterogeneous programming model.

Testing parallel applications is a difficult task because they have non-determined behavior, which makes it hard to detect their parallel errors when they occur. Even if these errors have been detected and the source code modified, it is difficult to decide if the errors have been corrected or still but hidden. If the application has integrated different programming models that will make the testing process even harder. Although there are many testing tools dedicated to detecting static and dynamic run-time errors, still not enough especially for detecting errors that occur in applications implemented in high-level programming models and also dual-programming models.

This research aims to identify some of OpenACC's run-time errors and design a suitable hybrid-testing tool architecture for systems implemented in C++ and the dual-programming model MPI+OpenACC. The combination of static and dynamic testing techniques will be used for detecting run-time errors by analyzing the source code before and during run time, which will improve the testing time and cover a wide range of errors.

The rest of this paper is structured as follows. Section 2 briefly gives an overview of the related work. Section 3 will identify OpenACC run-time error classification. The proposed architecture will be discussed in Section 4; the discussion will comprise Section 5, and finally the conclusion in Section 6.

II. RELATED WORKS

In testing parallel applications, there are many studies, which varied for several purposes and different scopes. These variations of testing tools including tools that detect a specific type of errors, the used testing techniques, and the targeted programming models as well as the programming model levels single, dual, and try level. There are many researches have been done in detecting a specific type of errors such as data race and deadlock, using several testing techniques. There are many tools that used static and dynamic testing for detecting deadlock such as UNDEAD [15]. Other tools are designed to detect race condition like the tool introduced in [14]. Finally, in [9] there are some detection techniques, which proposed for detecting livelock.

Several testing tools have been dedicated to testing the programming models by using different approaches. Some of them focusing in testing single programming models such as testing tools for MPI [16], [17], OpenMP [18], [19], CUDA [20] and OpenCL [21], but other studies tested the dual-programming models including MPI + OpenMP [11], [22]. Another approach is to focus on the testing techniques such as dynamic testing in [9], [10], static testing in [6]–[8], and the hybrid-testing techniques in [11]–[14].

There are many debuggers used for HPC applications include both open-source and commercial versions. ALLINEA DDT [23], is a commercial debugger that supports C++, MPI, and OpenMP designed to work on Petascale. Another commercial debugger is TotalView [24], which supports MPI, OpenMP and CUDA. These debuggers do not help in testing or detecting run-time errors but used to find the causes of these errors. Also, the thread, process, and kernel are needed to be selected for investigation. In terms of open-source testing tools, there are many tools to detect race condition in OpenMP including ARCHER [25], which used hybrid techniques to identify OpenMP data race. Finally, AutomaDeD [26], MEMCHEKER [16] and MUST [27] are used for detecting errors in MPI.

In terms of OpenACC testing, there is a shortage of testing OpenACC for detecting run-time errors. However, there are some studies that related to evaluating different compilers by creating test cases for OpenACC 2.0 as shown in [28]. Another study has been published in [29], also for evaluating CAPS, PGI, and CRAY compilers. Finally, OpenACC 2.5 was evaluated in [30] for validating and verifying the new feature of OpenACC compilers' implementation

Despite efforts made to create and propose software testing tools for parallel application, there is still a lot to be done primarily for GPU-related programming models and for dual- and tri-level programming models for heterogeneous systems. Heterogeneous systems can be hybrid CPUs/GPUs architectures or different architectures of GPUs. We noted that OpenACC has several advantages and benefits and has been used widely in the past few years, but it has not targeted any testing tools covered in our study. Finally, to the best of our knowledge, there is no parallel testing tool built to test applications programmed by using the dual-programming model MPI + OpenACC or any tri-level programming model.

III. Classification of OpenACC Run-Time Errors

Many studies have been done in detecting and identifying MPI run-time errors, so we will not cover them in our paper. However, OpenACC errors have not been previously investigated, identified, or classified. Therefore, we investigate and analyze OpenACC documents and specifications as well as conducting several experiments to identify and classify run-time errors that cannot be detected by the compilers.

Similar to any programming model that supports parallelism, OpenACC has several run-time errors as a result of their parallel nature. Compilers cannot detect these errors, which occur after compilation and cause several issues without developers' awareness. Usually, run-time errors have similar names but with different behavior and causes. For instance, the race condition in any application implemented by using one type of programming model has different causes and behaves differently from race condition in other applications implemented by another programming model. Also, run-time errors in applications implemented by the dual-programming model are different based on the combination between the hybrid programming models, and some errors occur specifically in a specific programming model.

OpenACC directives can be divided into data management directives and compute directives [31]. Compute directives are responsible for determining the blocks of the source code that can distribute the work to multiple threads in the GPU. Data management directives are responsible for avoiding unnecessary data movement between CPU and GPU. We use only compute directives that lead to moving data from and to the GPU each time we use these directives. The data directives determine data lifetime in the GPU, and in this time the GPU essentially owns the data. The data region in OpenACC is divided into a structured data region and unstructured data region [32]. The structured data region must have an explicit start and end points within a single function, and memory exists within the data region. On the other hand, the unstructured data region can have more than one start and end points and can branch across multiple functions, and memory exists until explicitly deallocated.

In the latest version of OpenACC 2.7 document [33], there is an error that appears several times through this document, which will cause run-time errors if not solved. This error is related to the presence of the variable in the GPU when needed to be used, for several OpenACC directives. Basically, it can happen when the variable is deleted from the GPU by a thread while it is needed by another thread. Furthermore, if any variable is not allocated in the GPU when it is needed, a message that indicates that there is a run-time error occurs in different parts of the source code without considering or explaining the cause or the error type. In addition, discovering these types of errors are difficult, even more complicated discovering them in applications developed by dual-programming model. As a result, we detect, identify and classify some of the OpenACC run-time errors into several categories determined by the way that OpenACC directives interact and behave. This classification includes the common run-time errors that occur because of the nature of parallel systems using the OpenACC programming model. Also, we include other errors related to OpenACC programming model and also classify them into the first two categories. In the following, we show our classifications.

A. Device-Based Data Transmission Errors

In this classification, run-time errors could happen as a result of mishandling of data using OpenACC directives and data clauses. Developers easily make these errors if they do not pay attention to the usage of OpenACC clauses, which leads to wrong results or non-deterministic behavior. These errors happen when developers mistakenly use the following data clauses in structured or unstructured data clauses. The data clauses include copy, copyin, copyout, create, delete, and present. Several cases can be included in this classification.

1) Errors that lead to run-time error messages can occur in unstructured or structured data regions. These errors will issue a run-time error message that indicates an invalid value. In the unstructured data region, there is a variable in a copyout clause in the exit data region without having the same variable in the enter data region, which the compiler cannot detect. In this case, there are several scenarios to demonstrate this error; one of them is the following code in Fig. 1. We noticed that the variable in the copyout clause is the array "a", but the error will happen because the variable "a" is deleted before the copyout clause, so there is nothing to copy back from GPU to the CPU, as in Fig. 1b. Also, in Fig. 1a, the copyin variable is the array "b", but a different array "a" in the copyout clause, which causes the same run-time error. Finally, if the developers forget to write "acc" in the enter or exit data or forget to enter the data directive, this error will occur, and the compiler will not detect it.

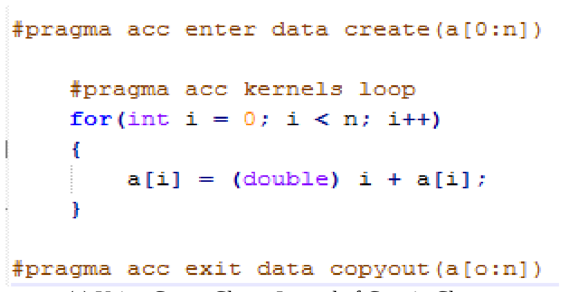
In the structured data region, when the developer used the copy, copyout, create, and present clauses and the variable is not present in the current GPU, the run-time error message will be issued. This error cannot be detected by a compiler and will happen only after compilation at run-time.

2) The other class is errors that could lead to wrong results without the developer's awareness as well as that of the compiler, who therefore cannot detect them. There are several scenarios that could lead to these errors, including the structured and unstructured data regions. In the unstructured data region, if the developer uses a create clause instead of a copyin clause when a copy of the variable needs to be copied from the CPU to the GPU, this leads the program to yield wrong results, as is shown in Fig. 2a. Also, the variable is deleted at the exit region when this variable needs to be copied back to the CPU, as shown in Fig. 2b. Finally, this error can also happen when forgetting to write "acc" at the exit data region or not using the exit data region directive.

In the structured data region, some cases can cause wrong results when using a data clause, including using copyin instead of copy when developers want to copy data to the GPU and do some operations and copy back the results to the CPU. This will cause wrong results that cannot be detected by the compiler. This can also happen if a developer forgets to add "acc" to the data region directive.

B. Memory Errors

These errors also can cause wrong results or run-time errors, like the previous classification, but can also affect the GPU memory by keeping variables and matrixes in the GPU memory without using them. The main idea of this error classification is that it is based on sending variables to the GPU without getting back any variables, or keeping some of them without using or deleting them. Also, this can occur when creating variables in the GPU without deleting or copyout them at the exit of the region. This can cause several issues, including affecting the GPU performance by consuming the GPU memory unnecessarily and can cause further errors in the case of using the same memory location with another variable in another part of the code. This type of error happens in the unstructured data region because the developer must determine the enter data and exit data by himself, while in the structured data region the developer only determines the data clause to use, and the compiler will deal with the internal operations needed at the enter and exit data.

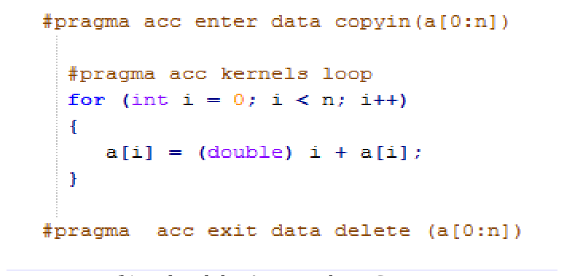


```
#pragma acc enter data create(a[0:n])

    #pragma acc kernels loop
    for(int i = 0; i < n; i++)
    {
        a[i] = (double) i + a[i];
    }

#pragma acc exit data copyout(a[o:n])
```

(a) Using Create Clause Instead of Copyin Clause.



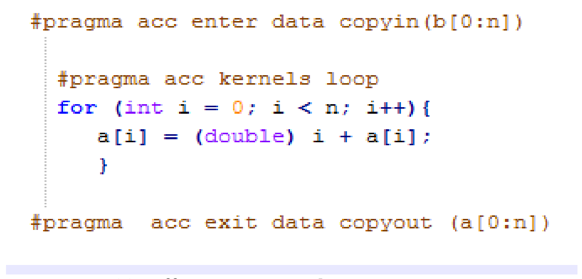
```
#pragma acc enter data copyin(a[0:n])

  #pragma acc kernels loop
  for (int i = 0; i < n; i++)
  {
     a[i] = (double) i + a[i];
  }

#pragma  acc exit data delete (a[0:n])
```

(b) Deleted the Array without Copyout.

Fig. 1. Unstructured Data Region has Errors Leading to Wrong Results.

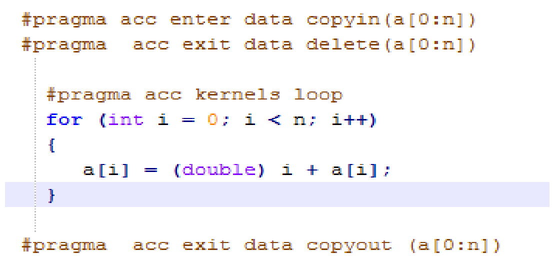


```
#pragma acc enter data copyin(b[0:n])

  #pragma acc kernels loop
  for (int i = 0; i < n; i++){
     a[i] = (double) i + a[i];
     }

#pragma  acc exit data copyout (a[0:n])
```

(a) Different Copyin and Copyout Arrays.



```
#pragma acc enter data copyin(a[0:n])
#pragma  acc exit data delete(a[0:n])

  #pragma acc kernels loop
  for (int i = 0; i < n; i++)
  {
     a[i] = (double) i + a[i];
  }

#pragma  acc exit data copyout (a[0:n])
```

(b) Deleted the Array before the Copyout.

Fig. 2. Unstructured Data Region has Errors Leading to a Run-Time Error Message.

C. Race Condition

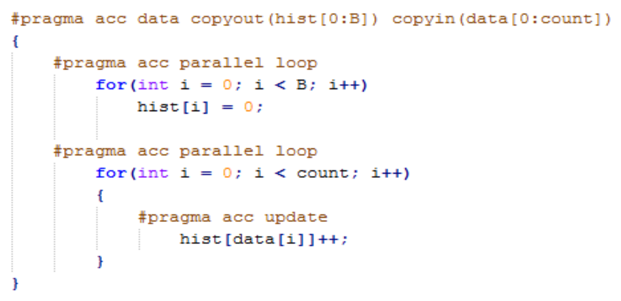
OpenACC race condition has different causes and behaves slightly different from other programming models, because of the nature of OpenACC and how it works in the heterogeneous architecture. Race condition can occur because of multiple threads will execute several processes concurrently and the thread execution sequence makes a difference in the parallel execution results. In addition, by using OpenACC, developers cannot guarantee the threads' execution order [32]. Because of the developers' responsibility to make sure there is no data dependency, OpenACC is more likely to have race conditions, which can happen as a result of several causes and situations, including:

1) The synchronization between host and device or vice versa to keep the data coherence between them. The programmer should be careful when dealing with updating data between host and device and should know when and how to do this updating because sometimes this can cause a race condition. The following example in Fig. 3 shows data race occurring because multiple threads may be updating the same element in the "hist" array, which will cause a race condition. Also, these parallel regions are in the same data region.

2) Paralyzing for loops: We paralyze for the loop so the code in the body of the loop can run in parallel using concurrent hardware execution thread. The iteration variable appears to be incremented sequentially, but threads are actually using different values of iteration variable in this, for the loop variable may be running in parallel at the same time. OpenACC makes no guarantee about the execution order of the threads. Moreover, it is possible that the last iteration of the loop may be completed before the zero loop iteration. This will lead to a potential race condition when the order of execution is important and affects the operations executed in the loop body, or when there is a dependency between some variables in the loop block that need to be updated before completing the next statements.

3) Shared data read and write: This situation involves creating a shared array or any dataset in a global kernel, and trying to read, write, and update from different threads concurrently—for example, writing a value by a thread in a location and reading the same location by another thread. This may cause a potential race condition.

4) Asynchronous Directive: OpenACC supports the asynchronous and wait directives. Programmers are responsible for ensuring their applications synchronization when they use asynchronous and wait directives to avoid a race condition between the host and device. In OpenACC, there is a hidden barrier at the end of each compute region, and the CPU thread execution will not proceed until all GPU threads have arrived at the end of the OpenACC compute region [33]. By default, all OpenACC directives are synchronous, which means that the CPU thread sent the required data and instructions to the GPU; after that, the CPU thread will wait for the GPU to complete its work before continuing execution [31].



```
#pragma acc data copyout(hist[0:B]) copyin(data[0:count])
{
    #pragma acc parallel loop
        for(int i = 0; i < B; i++)
            hist[i] = 0;

    #pragma acc parallel loop
        for(int i = 0; i < count; i++)
        {
            #pragma acc update
                hist[data[i]]++;
        }
}
```

Fig. 3. Race Condition because of Synchronization between Host and Device.

Using asynchronous and wait directives allows the CPU to continue working while the GPU works at the same time, which allows the pipelining execution of the system and enhances performance. However, when developers use these directives without considering their system requirements and miss using them, this can lead to a race condition. The following code in Fig. 4 shows the code that has a race condition because of the misuse of the asynchronous directive. The array "A" will go to asynchronous queue number 1, and "B" will go to queue number 2, and the CPU will continue working without waiting for the previous two queues to be completed and without considering that these arrays are needed before computing array "C". Therefore, array "C" will have a race condition that leads to wrong results.

5) Reduction clause: The reduction clause variable copies are generated for each loop iteration, similar to the private clause, reducing all of these private copies into one final result that will be returned from the GPU to the CPU [33]. In OpenACC, a reduction clause can specify the operator on the scalar variables, including summation, multiplication, and maximum and minimum operators. Some compilers will detect reduction on the reduction variable and implicitly insert the reduction clause, but for others, the programmer should always indicate reductions in the code.

Although some data dependency can be avoided by using a reduction clause, misusing the reduction clause in some cases will lead to OpenACC race condition. When there is no reduction clause that will lead also to race condition. Because reduction clause combines the result of each copy of the reduction variable with the original variable at the end of the OpenACC compute region, the variable should be initialized to some value based on the used operator before using the reduction clause. Otherwise, undefined behavior will result.

6) Independent clause: When developers use the independent clause, that tells the compiler that this loop is data-independent, which can cause problems if there is a dependency. It is the developers' responsibility to ensure not using independent clause if there is a data dependency because it allows programmers to tell the compiler that the loop iterations are data-independent These independent loop clauses can be used to tell the compiler that all loop iterations are independent, which means that there is no dependency or relation between any two loop iterations.

D. Deadlock

In OpenACC, one cause of deadlock in the CPU is having livelock in the GPU. This happens because of the nature of the implicit barrier at the end of each compute region. That means the GPU will be busy with the livelock while the CPU is waiting for the GPU to finish its operation. In the usage of the asynchronous directive, the GPU livelock also causes CPU deadlock, but by different behavior in terms of the usage of the wait directive; this will cause the CPU to have deadlock in that statement. In this case, the deadlock behaviors are based on the asynchronous and wait directive interactions. The following Fig. 5 shows deadlock situations that occur in the OpenACC application.

```
#pragma acc data copy(A[:N], B[:N], C[:N])
{
    #pragma acc parallel loop copy(A[:N])
        for(int i = 0; i < N; i++)
        {
            A[i] = i;
        }
    #pragma acc parallel loop copy(B[:N])
        for(int i = 0; i < N; i++)
        {
                B[i] = i;
                if ( i ==3 )
                {
                    while (i == 3)
                    {
                        B[i] = 3;
                    }
                }
        }
    #pragma acc parallel loop
        for (int i = 0; i < N; i++)
        {
            C[i] = A[i] + B[i];
        }
}
```

(a) Implicit Barrier Deadlock.

```
#pragma acc data copy(A[:N], B[:N], C[:N])
{
    #pragma acc parallel loop copy(A[:N])
        for(int i = 0; i < N; i++)
        {
            A[i] = i;
        }
    #pragma acc parallel loop copy(B[:N]) async
        for(int i = 0; i < N; i++)
        {
                B[i] = i;
                if ( i ==3 )
                {
                    while (i == 3)
                    {
                        B[i] = 3;
                    }
                }
        }
    #pragma acc parallel loop
        for (int i = 0; i < N; i++)
        {
            C[i] = A[i] + B[i];
        }
}
#pragma acc wait
```

(b) Wait Directive Deadlock.

Fig. 4. Deadlock because of the GPU Livelock.

```
#pragma acc parallel loop copy(A[:N]) async(1)
    for(int i = 0; i < N; i++)
    {
        A[i] = i;
    }
for(int j = 0; j < N; j++)
    {
            cout << " A[" << j << "] = " << A[j] << endl;
    }
cout << "*********************************" << endl;
#pragma acc parallel loop copy(B[:N]) async(2)
    for(int i = 0; i < N; i++)
    {
            B[i] = i;
    }
for(int j = 0; j < N; j++)
    {
            cout << " B[" << j << "] = " << B[j] << endl;
    }
cout << "*********************************" << endl;
#pragma acc parallel loop copy(C[:N])
    for (int i = 0; i < N; i++)
    {
        C[i] = A[i] + B[i];
    }
for(int j = 0; j < N; j++)
    {
            cout << " C[" << j << "] = " << C[j] << endl;
    }
```

Fig. 5. Race Condition because of the Miss use of Asynchronous Directive.

IV. PROPOSED ARCHITECTURE

We propose a parallel hybrid-testing tool for applications implemented in the dual-programming model (MPI + OpenACC) and C++ programming language, as shown in Fig. 6. Our design has the ability to detect real and potential run-time errors, with providing the necessary information for programmers to help them fix these errors. Our design has integrated static and dynamic testing techniques for covering a range of errors.

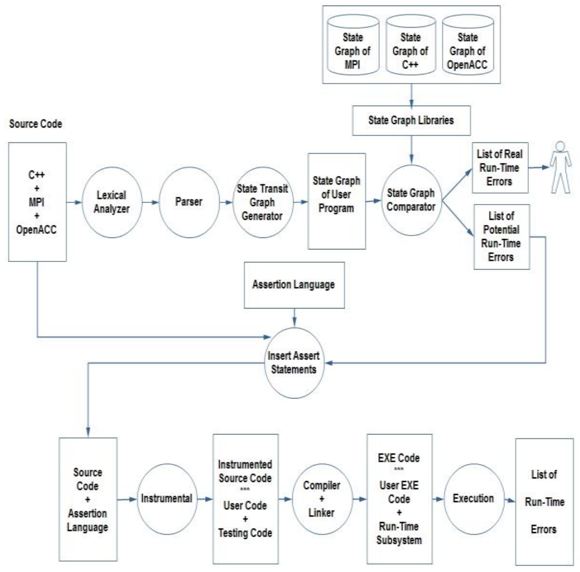


Fig. 6. The Architecture of Parallel Hybrid Testing Tool

The targeted source code will be analyzed in the static phase for discovering both potential and real run-time errors as well as marking the source code for further examination in our dynamic phase of the testing tool. The real errors which will defiantly occur during run time will be sent to the programmers with related information for their action to fix these errors before execution the source code. As a result, static detection helps in enhancing the testing performance by reducing the testing time.

The static part of our architecture includes the following parts:

- Lexical Analyzer; Take the source code as an input and convert it into a table of tokens.
- Parser: Analyze the source code checking for correct syntax in the process and confirming the formal grammar rules.
- State Transit Graph Generator: Generate a state graph for the given source code and represent it in a suitable structure.
- State Graph Comparator: Compare the generated graph with programming language and model graphs.

Any differences in the comparing process will be resulted in a list of run-time errors.

In the dynamic part of our design, the source code and the inserted statements will be taken as an input and pass them to be instrumented. The instrumenter will provide instrumented code that includes both user and testing codes written in C++. The instrumentation has two ways to do include.

1) Add the inserted testing codes to the original source code.

2) Call API functions to test the targeted part of the source code.

The first method generates larger source code size because it includes both user and testing codes. While the second has a smaller size because only the calling statements will be added to the user source code, and the function will test the user code. Also, when we have similar user code to be tested that will lead to repetitive testing code, in the first method, through the instrumented code which makes it even bigger. However, in the second method, we only write the testing code once and call it multiple times without affecting the instrumented source code size.

V. Discussion

Despite efforts made to detect run-time errors in the parallel application, there is still a lot to be done primarily for GPU-related programming models and for dual- and tri-level programming models for heterogeneous systems. Heterogeneous systems can be hybrid CPUs/GPUs architectures or different architectures of GPUs. We noticed that OpenACC has several advantages and benefits and has been used widely in the past few years by non-computer science specialists, but it has not targeted any testing tools covered in our study. Also, OpenACC has been used in the top five high-performance computing applications, and the top supercomputer in the World Summit also use OpenACC in five out of 13 applications [34]. As a result, the increase used of OpenACC will come with more errors that possibly occur and need to be discovered.

Furthermore, OpenACC errors have not been identified or classified, which makes it more difficult for us to build our tool. We studied OpenACC and conduct several experiments and build different scenarios to understand the run-time errors behavior in OpenACC as well as their effect when interacting with MPI. Based on these errors we proposed our design and determined the techniques to be used for testing the targeted applications.

In terms of our tool, the hybrid-testing techniques have been considered in our design. The combination of static and dynamic techniques takes advantage of both and reduces testing time by discovering the real errors during static analysis. The first phase is the static analysis approach which analyzes the source code before compilation and sending the resulted report to the programmers with the respective related information that help them to correct these errors. Also, mark the needed part for further dynamic investigation because some errors may or may not occur during run time based on the execution environment and behavior. The second part is the dynamic analysis approach which takes the marked parts from the static analysis, insert suitable testing statements and run them to test the user program.

Dealing with a parallel application is complicated and challenging because of their nature and how they behave, which need more effort to build our tool to cover a wide range of errors and predict scenarios of how the run-time errors will behave in each case. Finally, the used techniques will be determined based on the run-time error type and behavior.

VI. Conclusion and Future Work

As the Exascale supercomputers will be feasible in a few years, there is an increasing importance of building parallel systems. However, there is a shortfall in testing those systems, especially parallel systems that use heterogeneous programming models including high- and low-level programming models. Creating parallel applications by combining more than one programming model has benefits but also with more complex codes which are difficult to test and debug. That will lead to creating new approaches and techniques to detect run-time errors in such complex parallel applications.

Despite efforts made to create and propose software testing tools for parallel application, there is still a lot to be done primarily for GPU-related programming models and for dual- and tri-level programming models for heterogeneous systems. Heterogeneous systems can be hybrid CPUs/GPUs architectures or different architectures of GPUs. We noted that OpenACC has several advantages and benefits and has been used widely in the past few years, but it has not targeted any testing tools covered in our study.

In this paper, the main contribution that we identify and classify OpenACC run-time errors and their causes has been determined, with a brief explanation. Also, we proposed a solution for detecting run-time errors in application implemented in the dual-programming model. Based on the number of the application threads, our system will work in parallel creating testing threads for each needed application thread to be tested.

Our design will be implemented, and its ability for detecting run-time errors will be evaluated in our future work. The AZIZ supercomputer, which is one of the top ten supercomputers in Saudi Arabia will be used in our experiments in detecting parallel applications especially with heterogeneous architecture. Finally, to the best of our knowledge, there is no parallel testing tool built to test applications programmed by using the dual-programming model MPI + OpenACC or any tri-level programming model.

References

[1] Message Passing Interface Forum, "MPI Forum," 2017. [Online]. Available: http://mpi-forum.org/docs/.

[2] OpenMP Architecture Review Board, "About OpenMP," OpenMP ARB Corporation, 2018. [Online]. Available: https://www.openmp.org/about/about-us/.

[3] NVIDIA Corporation, "About CUDA," 2015. [Online]. Available: https://developer.nvidia.com/about-cuda.

[4] Khronos Group, "About OpenCL," Khronos Group, 2017. [Online]. Available: https://www.khronos.org/opencl/.

[5] OpenACC-standard.org, "About OpenACC," OpenACC Organization, 2017. [Online]. Available: https://www.openacc.org/about.

[6] E. Saillard, P. Carribault, and D. Barthou, "MPI Thread-Level Checking for MPI+OpenMP Applications," in EuroPar, vol. 9233, 2015, pp. 31–42.

[7] J. Jaeger, E. Saillard, P. Carribault, and D. Barthou, "Correctness Analysis of MPI-3 Non-Blocking Communications in PARCOACH," in Proceedings of the 22nd European MPI Users' Group Meeting on ZZZ - EuroMPI '15, 2015, pp. 1–2.

[8] A. Santhiar and A. Kanade, "Static deadlock detection for asynchronous C# programs," in Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI 2017, 2017, pp. 292–305.

[9] M. K. Ganai, "Dynamic Livelock Analysis of Multi-threaded Programs," in Runtime Verification, 2013, pp. 3–18.

[10] Y. Cai and Q. Lu, "Dynamic Testing for Deadlocks via Constraints," IEEE Trans. Softw. Eng., vol. 42, no. 9, pp. 825–842, 2016.

[11] E. Saillard, "Static / Dynamic Analyses for Validation and Improvements of Multi-Model HPC Applications . To cite this version : HAL Id : tel-01228072 DOCTEUR DE L ' UNIVERSITÉ DE BORDEAUX Analyse statique / dynamique pour la validation et l ' amélioration des applicat," University of Bordeaux, 2015.

[12] H. Ma, L. Wang, and K. Krishnamoorthy, "Detecting Thread-Safety Violations in Hybrid OpenMP/MPI Programs," in 2015 IEEE International Conference on Cluster Computing, 2015, pp. 460–463.

[13] Y. Huang, "An Analyzer for Message Passing Programs," Brigham Young University, 2016.

[14] R. Surendran, "Debugging, Repair, and Synthesis of Task-Parallel Programs," RICE UNIVERSITY, 2017.

[15] J. Zhou, S. Silvestro, H. Liu, Y. Cai, and T. Liu, "UNDEAD : Detecting and Preventing Deadlocks in Production Software," in Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, 2017, pp. 729–740.

[16] The Open MPI Organization, "Open MPI: Open Source High Performance Computing," 2018. [Online]. Available: https://www.open-mpi.org/.

[17] E. Saillard, P. Carribault, and D. Barthou, "PARCOACH: Combining static and dynamic validation of MPI collective communications," Int. J. High Perform. Comput. Appl., vol. 28, no. 4, pp. 425–434, 2014.

[18] H. Ma, S. R. Diersen, L. Wang, C. Liao, D. Quinlan, and Z. Yang, "Symbolic Analysis of Concurrency Errors in OpenMP Programs," in 2013 42nd International Conference on Parallel Processing, 2013, pp. 510–516.

[19] P. Chatarasi, J. Shirako, M. Kong, and V. Sarkar, "An Extended Polyhedral Model for SPMD Programs and Its Use in Static Data Race Detection," 2017, pp. 106–120.

[20] R. Sharma, M. Bauer, and A. Aiken, "Verification of producer-consumer synchronization in GPU programs," ACM SIGPLAN Not., vol. 50, no. 6, pp. 88–98, 2015.

[21] P. Collingbourne, C. Cadar, and P. H. J. Kelly, "Symbolic Testing of OpenCL Code," in Hardware and Software: Verification and Testing, 2012, pp. 203–218.

[22] B. Klemme, "Software Testing of Parallel Programming Frameworks," University of New Mexico, 2016.

[23] Allinea Software Ltd, "ALLINEA DDT," ARM HPC Tools, 2018. [Online]. Available: https://www.arm.com/products/development-tools/hpc-tools/cross-platform/forge/ddt.

[24] R. W. S. Inc., "TotalView for HPC," 2018. [Online]. Available: https://www.roguewave.com/products-services/totalview.

[25] Lawrence Livermore National Laboratory, University of Utah, and RWTH Aachen University, "ARCHER," GitHub, 2018. [Online]. Available: https://github.com/PRUNERS/archer.

[26] G. Bronevetsky, I. Laguna, S. Bagchi, B. R. de Supinski, D. H. Ahn, and M. Schulz, "AutomaDeD: Automata-based debugging for dissimilar parallel tasks," in IFIP International Conference on Dependable Systems & Networks (DSN), 2010, pp. 231–240.

[27] RWTH Aachen University, "MUST: MPI Runtime Error Detection Tool," 2018.

[28] J. Yang, "A Validation Suite for High-Level Directive-Based Programming Model for Accelerators a Validation Suite for High-Level Directive-Based Programming Model for," University of Houston, 2015.

[29] C. Wang, R. Xu, S. Chandrasekaran, B. Chapman, and O. Hernandez, "A validation testsuite for OpenACC 1.0," in Proceedings of the International Parallel and Distributed Processing Symposium, IPDPS, 2014, pp. 1407–1416.

[30] K. Friedline, S. Chandrasekaran, M. G. Lopez, and O. Hernandez, "OpenACC 2.5 Validation Testsuite Targeting Multiple Architectures," 2017, pp. 557–575.

[31] S. Chandrasekaran and G. Juckeland, OpenACC for Programmers: Concepts and Strategies, First edit. Addison-Wesley Professional, 2017.

[32] R. Farber, Parallel Programming with OpenACC. 2016.

[33] OpenACC Standards, "The OpenACC Application Programming Interface version 2.7," 2018.

[34] M. McCorkle, "ORNL Launches Summit Supercomputer," The U.S. Department of Energy's Oak Ridge National Laboratory, 2018. [Online]. Available: https://www.ornl.gov/news/ornl-launches-summit-supercomputer.