ASTES

Advances in Science, Technology and Engineering Systems Journal Vol. 4, No. 2, 203-210 (2019)

www.astesj.com

ASTESJ

ISSN: 2415-6698

Parallel Hybrid Testing Tool for Applications Developed by Using MPI + OpenACC Dual-Programming Model

Ahmed Mohammed Alghamdi*, Fathy Elbouraey Eassa

Department of Computer Science, King Abdul-Aziz University, Jeddah, Saudi Arabia

ARTICLE INFO

Article history:
Received: 24 January, 2019
Accepted: 21 March, 2019
Online: 26 March, 2019

Keywords:
Software Testing
Hybrid Testing Tool
OpenACC
MPI
Dual-programming Model

ABSTRACT

Building massively parallel applications has become increasingly important with coming Exascale related technologies. For building these applications, a combination of programming models is needed to increase the system's parallelism. One of these combinations is the dual-programming model (MPI+X) which has many structures that increase parallelism in heterogeneous systems that include CPUs and GPUs. MPI + OpenACC programming model has many advantages and features that increase parallelism with respect heterogeneous architecture and support different platform with more performance, productivity, and programmability.

The main problem in building systems with different programming models that it is a hard job for programmers and it is more error-prone, which is not easy to test. Also, testing parallel applications is a difficult task, because of the non-determined behavior of the parallel application. Even after detecting the errors and modifying the source code, it is not easy to determine whether the errors have been corrected or remain hidden. Furthermore, integrating two different programming models inside the same application makes it even more difficult to test. Also, the misusage of OpenACC can lead to several run-time errors that compilers cannot detect, and the programmers will not know about them.

To solve this problem, we proposed a parallel hybrid testing tool for detecting run-time errors for systems implemented in C++ and MPI + OpenACC. The hybrid techniques combine static and dynamic testing techniques for detecting real and potential run-time errors by analyzing the source code and during run time. Using parallel hybrid techniques will enhance the testing time and cover a wide range of errors. Also, we propose a new assertion language for helping in detecting potential run-time errors. Finally, to the best of our knowledge, identifying and classifying OpenACC errors has not been done before, and there is no parallel testing tool designed to test applications programmed by using the dual-programming model MPI + OpenACC or the single-programming models OpenACC.

1. Introduction

In recent years, building massively-parallel supercomputing systems based on heterogeneous architecture have been one of the top research topics. Therefore, creating parallel programs becomes increasingly important, but there is a lack of parallel programming languages, and the majority of traditional programming languages cannot support parallelism efficiently. As a result, programming models have been created to add parallelism to the programming languages. Programming models are sets of instructions, operations, and constructs used to support parallelism.

Today, there are various programming models which have different features and created for different purposes; including message passing, such as MPI [1] and shared memory parallelism, such as OpenMP [2]. Also, some programming models support heterogeneous systems, which consisting of a Graphics Processing Unit (GPU) coupled with a traditional CPU. Heterogeneous parallel programming models are CUDA [3] and OpenCL [4], which are low-level programming model and OpenACC [5] as a high-level heterogeneous programming model.

Testing parallel applications is a difficult task because parallel errors are hard to detect due to the non-determined behavior of the parallel application. Even after detecting the errors and modifying

*Ahmed Mohammed Alghamdi, Email: amalghamdi@uj.edu.sa

the source code, it is not easy to determine whether the errors have been corrected or hidden. Integrating two different programming models inside the same application even make it more difficult to test. Despite the available testing tools that detect static and dynamic errors, still, there is a shortage in such a testing tool that detects run-time errors in systems implemented in the high-level programming model.

The rest of this paper is structured as follows. Section 2 describes the research objectives, while Section 3 briefly gives an overview of some programming models and some run-time errors. The related work will be discussed in Section 4, the proposed architecture in Section 5, a discussion will be in Section 6 and finally the conclusion with future work in Section 7.

2. Research Objectives

This research aims to develop a parallel hybrid testing tool for systems implemented in MPI + OpenACC dual programming model with C++ programming language. The hybrid techniques combine static and dynamic testing techniques for detecting real and potential run-time errors by analyzing the source code and during run-time. Using parallel hybrid techniques will enhance the testing time and cover a wide range of errors. The following are the primary objectives of our research:

2.1. Provide new static testing techniques for detecting real and potential run-time errors for systems implemented in dual programming model (OpenACC and MPI) and C++ programming language.

These techniques are analyzing the source code before compilation for detecting static errors. Some run-time errors can also be detected from the source code, such as send-send deadlocks in Figure 2 B. These errors should be sent to developers to solve them because they will occur definitely in run-time. Also, potential run-time errors are errors that might or might not be occurred after compilation and during run-time. The reasons cause these potential errors can be detected from the source code before compilation by using static testing. However, if these errors have not been detected, it will become run-time errors. As a result, the developers should be warned about these errors and consider them; also our tool will instrument these errors by using assertion language.

The source code will include a combination of the program implemented in C++ and dual programming model source codes, which leads to one big size source code including a considerable number of statements. These static testing techniques will decrease the time of detecting run-time errors after the compilation, which will speed up the system testing time. These techniques also will allow us to correct or inform developers by providing them with a list of potential errors that in some cases in the running time these errors might happen.

The following example in Figure 1 shows a potential run-time error, when process_1 first receive request from any process beside process_0, there is no problem. However, if process_1 receives from process_0 first, the statement REC_FROM (P_0) will never, and the process_1 will be waiting. In that case, from the source code we discover that, somehow, this will cause a run-time error (Deadlock). This situation called a potential deadlock.

Also, Figure 2 shows an example of a real run-time error called (deadlock), which happened because of Process_0 block and waiting for receiving from Process_1, which also block and waiting for receiving from Process_0. Similarly, this also happened between Process_2 and Process_3.

Process_0 (P_0)	Process_1 (P_1)	Process 2 (P_2)	Process 3 (P_3)
SEND_TO (P_1)	REC_FROM (ANY)	SEND_TO (P_1)	SEND_TO (P_1)
	REC_FROM (P_0)	REC_FROM (P_3)	

Figure 1: Potential Deadlock caused by Wildcard Receive

Process_0 (P_0)	Process_1 (P_1)
REC_FROM (P_1)	REC_FROM (P_0)
SEND_TO (P_1)	SEND_TO (P_0)

(A) Receive - Receive Deadlock

Process_2 (P_2)	Process_3 (P_3)
SEND_TO (P_3)	SEND_TO (P_2)
REC_FROM (P_3)	REC_FROM (P_2)

(B) Send - Send Deadlock

Figure 2: Real Deadlocks caused by two different reasons

2.2. Providing a new assertion language for helping in detecting potential run-time errors.

This assertion language will be used to specify the properties of the programs under test and to verify that the developers' assumptions of the program remain valid during the program run-time. During testing, assertion statements help for the recording of some information, testing the correctness of statements, and monitor the values of variables. To do this, the dynamic tester will automatically insert assertion statements into the code, then provides a method for capturing, organizing, and analyzing assertions output. This will help to increase the error detection capability of a test by using the instrumentation technique. The instrumentation approach based on the idea that the tested part of a program can be specified regarding assertion or values that must be assumed by variables at specific critical points in the program, which can cause run-time errors [6].

Usually, assertion statements start with comment symbol of the programming language, such as "//" in C++, before each assert statements. The main reason behind this is reducing the compiled code that will be delivered to the customers because any statement starts with the comment symbol will be ignored during the compilation. In other words, the assert statements are in the source code but not in the compiled code, which will be delivered to the customers.

2.3. Provide new parallel dynamic testing techniques for detecting run-time errors for systems implemented in dual programming model (OpenACC and MPI) and C++ programming language.

These techniques will use the provided assertion language for detecting errors that happened during run-time, by instrumenting and analyzing the system during run-time. This is challenging because different factors and complicated scenarios can cause these errors. Also, testing parallel programs is a difficult task because of the nature of such programs and their behavior. This will add more work on the testing tool for covering every possible scenario of the test cases and data. As a result, detecting parallel run-time errors is more difficult. Furthermore, these dynamic techniques are sensitive to the execution environment and can affect the system execution time.

2.4. Integrated the provided techniques for developing a parallel hybrid testing tool for systems implemented in dual programming model (OpenACC and MPI) and C++ programming language.

Our proposed architecture will integrate static and dynamic testing techniques for creating a new hybrid testing tool for parallel systems. This allows us to take advantages of both previously mentioned techniques for detecting some of the dynamic errors from the source code by using the static testing techniques, which will enhance the system execution time. Also, our system will work in parallel to detect run-time errors, by creating testing threads depending on the number of the application threads. Intra-process and Inter-process run-time detections will be included in our tool. The inter-process detector will be responsible for detecting run-time errors that happened within the process, and the Intra-process detector for detecting errors happened between processes each other.

3. Background

In this section, the main components involved in our research will be displayed and discussed. This will include the programming models that will be used in our research and describing why they have been chosen. Also, some run-time errors and testing techniques will also be described and discussed in this section.

3.1. OpenACC

In November 2011, OpenACC stands for open accelerators, was released for the first time in the International Conference for High-Performance Computing, Networking, Storage and Analysis [7]. OpenACC is a directive-based open standard developed by Cray, CAPS, NVIDIA and PGI. They design OpenACC to create simple high-level parallel programming model for heterogeneous CPU/GPU systems, that compatible with FORTRAN, C, and C++ programming languages. Also, OpenACC Standard Organization defines OpenACC as "a user-driven directive-based performance-portable parallel programming model designed for scientists and engineers interested in porting their codes to a wide variety of heterogeneous HPC hardware platforms and architectures with significantly less programming effort than required with a low-level model." [5]. The latest version of OpenACC was released in November 2017. OpenACC has several features and advantages comparing with other heterogeneous parallel programming models including:

- Portability: Unlike programming model like CUDA works only on NVIDIA GPU accelerators, OpenACC is portable across different type of GPU accelerators, hardware, platforms, and operating systems.[8]
- OpenACC is compatible with various compilers and gives flexibility to the compiler implementations.
- High-level programming model, which makes targeting accelerators easier, by hiding low-level details. For generation low-level GPU programs, OpenACC relies on the compiler using the programmer codes. [9]
- Better performance with less programming effort, which gives the ability to add GPU codes to existing programs with less effort. This will lead to reduce the programmer workload and improve programmer productivity and achieving better performance than OpenCL and CUDA. [10]
- OpenACC allows users to specify three levels of parallelism by using three clauses:
    - Gangs: Coarse-Grained Parallelism
    - Workers: Medium-grained Parallelism
    - Vector: Fine-Grained Parallelism

OpenACC has both a strong and significant impact on the HPC society as well as other scientific communities. Jeffrey Vetter (HPC luminary and Joint Professor Georgia Institute of Technology) wrote: “OpenACC represents a major development for the scientific community. Programming models for open science by definition need to be flexible, open and portable across multiple platforms. OpenACC is well-designed to fill this need.” [5].

3.2. Message Passing Interface (MPI)

Message Passing Interface (MPI) [1] is a message-passing library interface specification. In May 1994, the first official version of MPI was released. MPI is a message-passing parallel programming model that moves data from a process address space to another process by using cooperative operations on each process. The MPI aims to establish a standard for writing message-passing programs to be portable, efficient, and flexible. Also, MPI is a specification, not a language or implementation, and all MPI operations are expressed as functions, subroutine or methods for programming languages including FORTRAN, C, and C++. MPI has several implementations including open source implementations, such as Open MPI [11] and MPICH [12]; and commercial implementations, such as IBM Spectrum MPI [13] and Intel MPI [14]. MPI has several features and advantages including:

- **Standard**: MPI is the only message passing library that can be considered a standard. It has been supported on virtually all HPC platforms. Also, all previous message passing libraries have been replaced by MPI.
- **Portability:** MPI can be implemented on several platforms, hardware, systems, and programming languages. Also, MPI can work correctly with several programming models and work with heterogeneous networks.
- **Availability:** Various versions of MPI implementations from different vendors and organization are available as open source and commercial implementations.
- **Functionality:** On MPI version 3.1 there are over 430 routines has been defined including the majority of the previous versions of MPI.

The new MPI standardization version 4.0 [1] is in progress, which aims to add new techniques, approaches, or concepts to the MPI standard that will help MPI address the need of current and next-generation applications and architectures. The new version will extend to better support hybrid programming models including hybrid MPI+X concerns and support for fault tolerance in MPI applications.

3.3. Dual-Level Programming Model: MPI + OpenACC

Integrating more than one programming model can enhance parallelism, performance, and the ability to work with

heterogeneous platforms. Also, this combination will help in moving to Exascale systems, which need more powerful programming models that support massively-parallel supercomputing systems. Hybrid programming models can be classified as:

- Single-Level Programming Model: MPI
- Dual-Level Programming Model: MPI + X
- Tri-Level Programming Model: MPI + X + Y

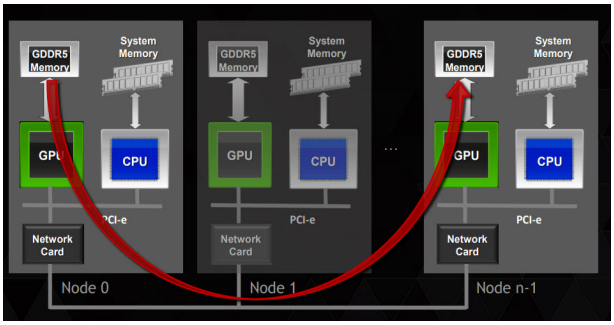

Figure 3: Multi GPU Programming with MPI and OpenACC [15]

In order to write portable and scalable applications for heterogeneous architecture, the dual-programming model MPI + OpenACC can be practical. It inherits the advantages, such as high performance, scalability, and portability from MPI and programmability and portability from OpenACC [16]. However, this dual-programming model might introduce different types of run-time errors, which have different behaviors and causes. Also, some complexities and inefficiencies might happen including redundant data movement and excessive synchronization between the models, which need to be considered and take care of, but it is better than using CUDA or OpenCL, which is more complicated and harder to program, resulting in lower productivity.

3.4. Common Run-Time Errors

There are several types of run-time errors that happened after compilation and cannot be detected by the compilers, which cause the program not to meet the user requirements. These errors even sometimes have similar names, but they are different in the reasons that cause the run-time error or the error behavior. For example, deadlock in MPI has different causes and behaviors comparing with OpenACC deadlocks. Also, run-time errors in the dual-programming model are different. Also, some run-time errors happened specifically in a particular programming model. By investigating the documents of the latest version of OpenACC 2.7 [17], we found that OpenACC has a repetitive run-time error that if a variable is not present on the current device, this will lead to run-time error. This case happened in non-shared memory devices for different OpenACC clauses.

Similarly, if the data is not present, a run-time error is issued in some routines. Furthermore, detecting such errors is not easy to do, and to detect them in applications developed by dual-programming model even more complicated. In the following, some popular run-time errors will be displayed and discussed in general with some examples.

3.4.1. Deadlock

A deadlock is a situation in which a program is in a waiting state for an indefinite amount of time. In other words, one or more threads in a group are blocked forever without consuming CPU cycles. The deadlock has two types including resource and communication deadlock. Resource deadlock is the situation where a thread waits for another thread resource to proceed.

Similarly, the communication deadlock occurs when some threads wait for some messages, but they never receive these messages [18–20]. The reasons that cause deadlock are different depending on the used programming models, systems nature and behavior. Once the deadlock occurs, it is not difficult to detect, but in some cases, it is difficult to detect them before it happened as they occur under specific interleaving. Finally, deadlocks in any system could be potential or real deadlocks.

3.4.2. Livelock

Livelock is similar to deadlock, except that livelock is a situation that happened when two or more processes change their state continuously in response to changes in the other processes. In other words, it occurs when one or more threads continuously change their states (and hence consume CPU cycles) in response to changes in states of the other threads without doing any useful work. As a result, none of the processes will make any progress and will not complete [21,22]. In a livelock, the thread might not be blocked forever, and it is hard to distinguish between livelock and long-running process. Also, livelock can lead to performance and power consumption problems because of the useless busy-wait cycles.

3.4.3. Race Condition

A race condition is a situation that might be occurred due to executing processes by multiple threads and where the sequence of execution for the threads makes a difference in the result of the concurrent execution. The execution timing and order will affect the program's correctness [20,23]. Some researchers do not differentiate between data race and race condition, which will be explained in the data race definition.

3.4.4. Data Race

A data race happened when there are two memory accesses in the program where they both are performed concurrently by two threads or target the same location [23, 24]. For example, at least one read and one write may happen at the same memory location, at the same time. The relation between data race and race condition, the race condition is a data race that causes an error. However, data race does not always lead to a race condition.

3.4.5. Mismatching

Mismatching is a situation that happened in arguments of one call, which can be detected locally and are sometimes even

detected by the compiler. Mismatching can be caused by several forms including wrong type or number of arguments, arguments involving more than one call, or in collective calls. Developers need to make special attention when comparing matched pairs of derived data types. Some examples of mismatching that occurred in MPI as the following [23]:

- To send two (MPI INT, MPI DOUBLE) and to receive one (MPI INT, MPI DOUBLE, MPI INT, MPI DOUBLE
- To send one (MPI INT, MPI DOUBLE) and to receive one (MPI INT, MPI DOUBLE, MPI INT, MPI DOUBLE) (a so-called partial receive).

3.5. Testing Techniques

There are many techniques used in software testing, which include static, dynamic, as well as other techniques. Static testing is the process of analyzing the source code before compilation phase for detecting static errors. It handles the application source code only without launching it, which give us the ability to analyze the code in details and have full coverage. In contrast, the static analysis of parallel application is complicated due to the unpredicted program behavior, which is parallel application nature. However, it will be beneficial to use static analysis for detecting potential run-time errors and some real run-time errors that are obvious from the source code, such as some types of deadlocks and race condition.

Dynamic testing is the process of analyzing the system during run-time for detecting dynamic (run-time) errors. It demands to launch programs, sensitive to the execution environment, and slow down the speed of application execution. It is useful to use dynamic analysis in the parallel application, which gives the flexibility to monitor and detect each thread of the parallel application. However, it is difficult to cover the whole parallel code with tests, and after correcting the errors, it cannot be confirmed that errors are corrected or hidden.

Finally, it is the error types and behaviors that determine which techniques will be used, because static analysis and others cannot detect dynamic techniques cannot detect some errors. As a result, in our research, a hybrid technique will be used for different purposes and reasons. Furthermore, this hybrid technology will be working in parallel to detect parallel run-time errors and analyzing the application's threads.

4. Related Works

Many studies have been done in software testing for HPC and parallel software. These researches are varied, for different purposes and scopes. These variations include testing tools or detection for a specific type of errors or a different type of errors. Some studies focus on using static testing techniques [25–28] to detect errors by analyzing the source code and find real as well as potential run-time errors [29,30]; dynamic testing techniques [31,32] to detect errors after execution and at run-time; or hybrid testing techniques [33–35]. Also, detecting errors in programming models also varied from the testing tool for single level programming model to the tri-level programming model. Even in the same classification of programming model the variation between testing the programming models themselves, because each programming model has a different error to detect as discussed earlier in Section 3.4.

For detecting a specific type of errors, there are many types of research worked on detecting deadlock, livelock and race condition by using different techniques. In deadlock detection, there are many tools and studies that are using static or dynamic testing techniques to detect deadlocks including resource and communication deadlocks. UNDEAD [19] is a deadlock detection and prevention, which helps to defeats deadlocks in production software with enhancing run-time performance and memory overheads. More deadlock detection can be found in [19,36] . Regarding detecting data race, a hybrid test-driven approach has been introduced in [35] to detect data race in task-parallel programs. Also, many data race detection approaches in [28,37]. Finally, some livelock detection techniques have been proposed in [21,22].

Regarding testing the programming model, many approaches have been introduced to test and detect errors in parallel software. Many studies have been done in a single level programming models such as MPI, OpenMP, CUDA and OpenCL. While some studies focus on dual-level programming models including MPI + X hybrid programming models, which include homogeneous and heterogeneous systems. One popular combination is MPI + OpenMP, which appears in [33,38,39]. Some of these studies focus on dynamic testing, while some of them in regression testing, which is the process of analyzing the system after the maintenance phase.

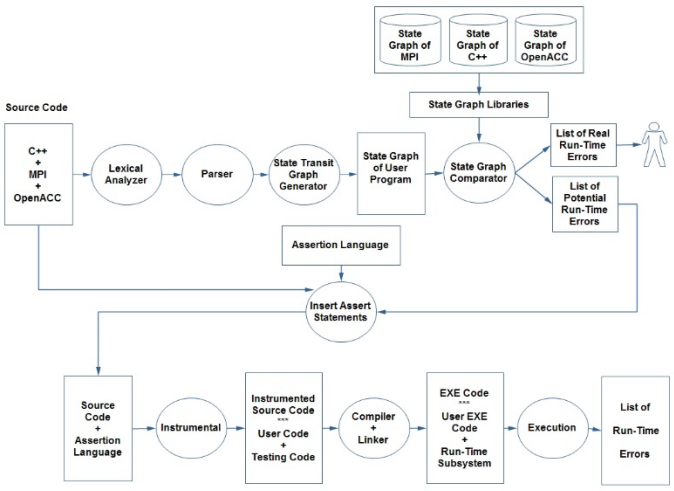

Figure 4: Our Proposed Architecture

Regarding open source testing tools, ARCHER [37] is a data race detector for an OpenMP program that combines static and

dynamic techniques to identify data race in large OpenMP applications. Also, AutomaDeD [42] (Automata-based Debugging for Dissimilar Parallel Tasks) is a tool that detects MPI errors by comparing the similarities and dissimilarities between tasks. MEMCHEKER [11] allows finding hard-to-catch memory errors in MPI application such as overwriting of memory regions used in non-blocking communication and one-sided communication. Furthermore, MUST [32] detects run-time errors in MPI and report them to the developers, including MPI deadlock detection, data type matching, and detection of communication buffer overlaps.

Testing OpenACC has limited studies in testing and detecting static and dynamic errors. There are some researches regarding related OpenACC testing. In [43], they evaluate three commercial OpenACC compilers by creating a validation suite that contains 140 test case for OpenACC 2.0. They also check conformance, correctness, and completeness of specific compilers for the OpenACC 2.0 new features. This test suite has been built on the same concept as the first OpenACC 1.0 validation test suite in [44], which three commercial compilers were evaluated including CAPS, PGI and CRAY. Similarly, this OpenACC test suite was published in [45] for OpenACC version 2.5, which is the past version, to validate and verify compilers' implementations of OpenACC features.

Recently, another testing of the OpenACC application was published in [46], which considered detecting numerical differences that can be occurred due to computational differences in different OpenACC directives. They proposed a solution for that by generating code from the compiler to run each computes region on both the host CPU and the GPU. Then, the values computed on the host and GPU are compared, using OpenACC data directives and clauses to decide what data to compare.

Despite the efforts that have been done in creating and proposing software testing tools for parallel application, still, there is a lot to be done primarily for OpenACC and for dual-programming models for heterogeneous systems. Finally, in our best knowledge, there is not a parallel testing tool built to test applications programmed by using the dual-programming model MPI + OpenACC.

5. Proposed Architecture

We propose a parallel hybrid testing tool for the dual-programming model (MPI + OpenACC) and C++ programming language as shown in Figure 4. This architecture has the flexibility to detect potential run-time errors and report them to the developer, detect them automatically by using assertion language and execute them to get a list of run-time errors, or detecting dynamic errors. This architecture uses hybrid testing techniques including static and dynamic testing. The static testing part is shown in Figure 5 while the dynamic part in Figure 6.

The source code includes C++ programming language and MPI + OpenACC as dual-programming models. The part that displayed in Figure 5 is responsible for detecting real and potential run-time errors by using static testing. This part produces a list of potential run-time errors for the developer.

Also, this list could be an input to the assertion process that these potential errors will be automatically detected and avoided during the dynamic testing part. Also, any real run-time errors also will be addressed to the developed with warning messages, as these errors must be corrected because they will defiantly occur during run-time. Also, these real run-time errors that been discovered from the source code can be automatically corrected before the process move to the dynamic testing part, which reduces the testing time and enhances the testing performance. The static part of the architecture includes:

- **Lexical analyzer:** This will take the source code that includes C++, MPI, and OpenACC as an input. This analyzer will understand the source code because it has all the information related to the programming language and the determined programming models. This information includes keywords, reserved words, operators, variable and constant definitions. Then, it will convert the application source code into tokens and allocate them into tables of tokens. The output of this analyzer will be a token table, which includes token names and their respective type.
- **Parser:** This Part is responsible for analyzing the syntax of the input source code and confirming the rule of a formal grammar. This process will produce a structural representation of the input (Parser Tree) that shows the syntax relation to each other, checking for correct syntax in the process.
- **State transit graph generator:** This part will generate a state graph for the user program, which includes C++, MPI, and OpenACC. This state graph will be represented by any suitable data structure such as a matrix or linked list.

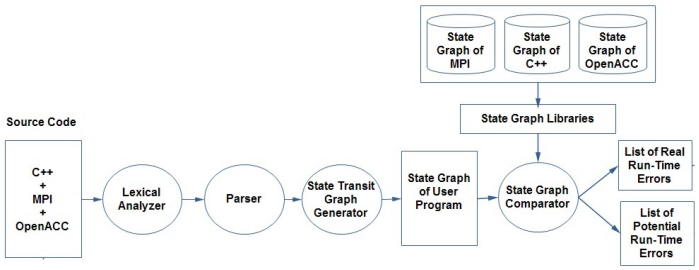

Figure 5: Static Part of the Proposed Architecture

The dynamic testing part of the proposed architecture is shown in Figure 6, which takes the source code and the assertion language as an input and move them to the instrumental. The instrumental depending on the semantics of the assertion language will produce

code in the targeted programming language. The instrumental consist of four modules including; a lexical analyzer, parser, semantic, and code translator. The instrumental will produce an instrumented source code as an output. The instrumented source code includes the user codes and the testing codes both of them wrote in the user code programming language. Two methods can do instrumentation. Firstly by adding the testing codes, assertion statements, to the source code which leads to bigger code size as it will have user code and testing code. The second method is by adding the assert statements as calling of API functions, and these functions will test the part of the code that needs to be tested. This method leads to a smaller code size that any testing needed a call statement will be written, and the function will do the test. It is noticeable when we have the same testing code for several parts of the user code, in the previous method this testing code will be repeated many times, while in this method it will be only written once and called multiple times.

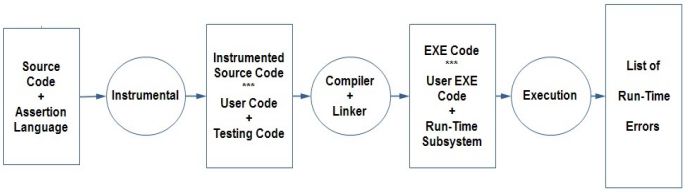

Figure 6: Dynamic Part of the Proposed Architecture

Further investigation of the instrumentation will be considered in our future progress. The resulted instrumented code will be compiled and linked, which results in EXE codes including user executable code and run-time subsystems. Finally, these EXE codes will be executed and provide a list of run-time errors.

6. Discussion

There are many tools, and researches have been done to detect a run-time error that occurs in parallel systems, which used MPI, CUDA, and OpenMP programming models. However, even though OpenACC can work in heterogeneous architecture, hardware, and platforms, as well as used by non-computer science specialist, which easily can have several errors. There is not a research or testing tool that detects OpenACC run-time errors. Also, OpenACC becomes increasingly used in different research fields as well as one of the main programming models targeting Exascale systems. Recently, OpenACC has been used in five of 13 applications to accelerate performance in the top supercomputer in the world Summit. Also, three of the top five HPC applications are using OpenACC as well. Therefore, this increased in using OpenACC will come with more errors that need to be detected.

In our tool, we consider having hybrid testing techniques including static and dynamic testing. This combination takes the advantages of two testing techniques, reduces disadvantages, and reduces the testing time. The first part of the hybrid technique is a static testing technique which analyses the source code before compilation to detect static errors. Some of the run-time errors can also be detected from the source code and should be sent to developers to solve them because they will occur definitely at run-time. In addition, potential run-time errors are errors that might or might not be occurred after compilation and during run-time based on the execution behavior. The reasons that cause these potential errors can be detected from the source code before compilation by using static testing. However, if these errors have not been detected, it will become run-time errors. As a result, the developers should be warned to these errors and consider them.

The second part of the hybrid technique is a dynamic testing technique that is detecting errors that happened during run-time, by instrumenting and analyzing the system during run-time. This is challenging because different factors and complicated scenarios can cause these errors. In addition, testing parallel programs is a difficult task because of the nature of such programs and their behavior. This will add more work to the testing tool for covering every possible scenario of the test cases and data. Furthermore, these dynamic techniques are sensitive to the execution environment and can affect the system execution time. Finally, it is the run-time errors type and behavior that determines which techniques will be used, because static analysis and others cannot detect dynamic techniques cannot detect some errors.

7. Conclusion and Future Works

High-performance computing has become increasingly important, and the Exascale supercomputers will be feasible by 2020; therefore, building massively parallel supercomputing systems based on a heterogeneous architecture has become even more important to increase parallelism. Using hybrid programming models for creating parallel systems has several advantages and benefits, but mixing parallel models within the same application leads to more complex codes. Testing such complex applications is a difficult task and needs new techniques for detecting run-time errors.

We proposed a parallel hybrid testing tool for detecting run-time errors for systems implemented in C++ and MPI + OpenACC. This proposed solution integrates static and dynamic testing techniques for building a new hybrid testing tool for parallel systems. This allows us to take advantages of both previously mentioned techniques for detecting some of the dynamic errors from the source code by using the static testing techniques, which will enhance the system execution time. Also, our system will work in parallel to detect run-time errors, by creating testing threads depending on the number of the application threads.

In our future work, we will identify and classify the OpenACC run-time errors and study their behavior and causes to be our guide in building our testing tool. Also, we will implement our architecture and evaluate its ability to detect OpenACC run-time errors and also we will identify and address the run-time errors that resulted from the dual-programming model MPI + OpenACC. Our experiments will be conducted in AZIZ supercomputer, which is one of the top ten supercomputers in the Kingdom of Saudi Arabia. On June 2016, AZIZ was ranked No. 359 among the Top 500 supercomputers in the world.

Conflict of Interest

The authors declare no conflict of interest.

Acknowledgment

This work was funded by the Deanship of Scientific Research (DSR), King Abdulaziz University, Jeddah, under grant No. (DG1440 - 12 - 611). The authors, therefore, acknowledge with thanks DSR technical and financial support.

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] F. E. Eassa, L. J. Osterweil, and M. Z. Abdel-mageed, "AIDA: a dynamic analyser for Ada programs," *Inf. Softw. Technol.*, vol. 36, no. 2, pp. 107–117, 1994.

[7] SC11, "the International Conference for High Performance Computing, Networking, Storage and Analysis," 2011. [Online]. Available: http://sc11.supercomputing.org/.

[8] A. Fu, D. Lin, and R. Miller, "Introduction to OpenACC," 2016.

[9] M. Daga, Z. S. Tschirhart, and C. Freitag, "Exploring Parallel Programming Models for Heterogeneous Computing Systems," in *2015 IEEE International Symposium on Workload Characterization*, 2015, pp. 98–107.

[10] J. A. Herdman, W. P. Gaudin, O. Perks, D. A. Beckingsale, A. C. Mallinson, and S. A. Jarvis, "Achieving portability and performance through OpenACC," *Proc. WACCPD 2014 1st Work. Accel. Program. Using Dir. - Held Conjunction with SC 2014 Int. Conf. High Perform. Comput. Networking, Storage Anal.*, no. July 2013, pp. 19–26, 2015.

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

[12] MPICH Organization, "MPICH," 2018. [Online]. Available: http://www.mpich.org/.

[13] IBM Systems, "IBM Spectrum MPI," 2018. [Online]. Available: https://www.ibm.com/us-en/marketplace/spectrum-mpi.

[14] Intel Developer Zone, "Intel MPI Library," 2018. [Online]. Available: https://software.intel.com/en-us/intel-mpi-library.

[15] J. Kraus and P. Messmer, "Multi GPU programming with MPI," in *GPU Technology Conference*, 2014.

[16] J. Kim, S. Lee, and J. S. Vetter, "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*, 2016, pp. 189–201.

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

[18] K. Shankari and N. G. B. Amma, "Clasp: Detecting potential deadlocks and its removal by iterative method," in *IC-GET 2015 - Proceedings of 2015 Online International Conference on Green Engineering and Technologies*, 2016.

[19] 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.

[20] J. F. Münchhalfen, T. Hilbrich, J. Protze, C. Terboven, and M. S. Müller, "Classification of common errors in OpenMP applications," *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 8766, pp. 58–72, 2014.

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

[22] Y. Lin and S. S. Kulkarni, "Automatic Repair for Multi-threaded Programs with Deadlock / Livelock using Maximum Satisfiability," *ISSTA Int. Symp. Softw. Test. Anal.*, pp. 237–247, 2014.

[23] B. Krammer and M. M. Resch, "Runtime Checking of MPI Applications with MARMOT," in *Performance Computing*, 2006, vol. 33, pp. 1–8.

[24] M. Cao, "Efficient , Practical Dynamic Program Analyses for Concurrency Correctness," The Ohio State University, 2017.

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

[26] N. Ng and N. Yoshida, "Static deadlock detection for concurrent Go by global session graph synthesis," *CC 2016 Proc. 25th Int. Conf. Compil. Constr.*, vol. 1, no. 212, pp. 174–184, 2016.

[27] 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.

[28] 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.

[29] 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.

[30] A. T. Do-Mai, T. D. Diep, and N. Thoai, "Race condition and deadlock detection for large-scale applications," in *Proceedings - 15th International Symposium on Parallel and Distributed Computing, ISPDC 2016*, 2017, pp. 319–326.

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

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

[33] 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.

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

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

[36] V. Forejt, S. Joshi, D. Kroening, G. Narayanaswamy, and S. Sharma, "Precise Predictive Analysis for Discovering Communication Deadlocks in MPI Programs," *ACM Trans. Program. Lang. Syst.*, vol. 39, no. 4, pp. 1–27, Aug. 2017.

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

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

[39] 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.

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

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

[42] 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.

[43] 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.

[44] 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.

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

[46] K. Ahmad and M. Wolfe, "Automatic Testing of OpenACC Applications," in *4th International Workshop on Accelerator Programming Using Directives*, vol. 10732, 2018, pp. 145–159.