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

Ahmed M. Alghamdi[1*] and Fathy E. Eassa[2]

King Abdulaziz University, Jeddah, KSA

Summary
Recently, building massively parallel systems has become increasingly important with coming improvements of Exascale related technologies. For building such systems, a combination of programming models is needed to increase the system's parallelism. One of these combinations is the dual-programming model (MPI+X) that has different structures to increase parallelism in heterogeneous systems that include CPUs and GPUs. (MPI + OpenACC) has many advantages and features that increase parallelism with respect to heterogeneous architecture and support different platforms with more performance, productivity and programmability. However, building parallel systems with different programming models is even harder and is more error-prone, which is not easy to test. Also, 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 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.
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 during run time. Using parallel hybrid techniques will enhance the testing time and cover a wide range of 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 like OpenACC.

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

1. Introduction

Building massively parallel supercomputing systems based on heterogeneous architecture has been one of the top research topics in the past few years. Therefore, creating parallel programs has also 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, various programming models have different features and have been 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 consist of a Graphics Processing Unit (GPU) coupled with a traditional CPU. Heterogeneous parallel programming models are CUDA [3] and OpenCL [4], which are considered low-level programming models 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 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 makes testing even more difficult to test. Despite the available testing tools that detect static and dynamic errors, there is still a shortage of such testing tools that detect run-time errors in systems implemented in the high-level programming models.

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

Manuscript received March 5, 2019
Manuscript revised March 20, 2019

2. 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.

2.1 OpenACC

In November 2011, OpenACC which stands for open accelerators, was released for the first time in the International Conference for High-Performance Computing, Networking, Storage and Analysis [6]. 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 2018. 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], [9]
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. [8]
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].

2.2 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 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.

2.3 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

In our research, the dual-programming model (MPI + OpenACC) will be discussed. As mentioned earlier MPI and OpenACC have various advantages, and by combining them, we will enhance parallelism, performance and reduce programming efforts as well as taking advantage of heterogeneous GPU accelerators. That will be achieved by using OpenACC that can be compiled into multiple device types, including multiple vendors of GPUs and multi-core CPUs as well as different hardware architecture. MPI will be used to exchange the data between different nodes as shown in Figure 1, which display how to use MPI for inter GPU communication with OpenACC.

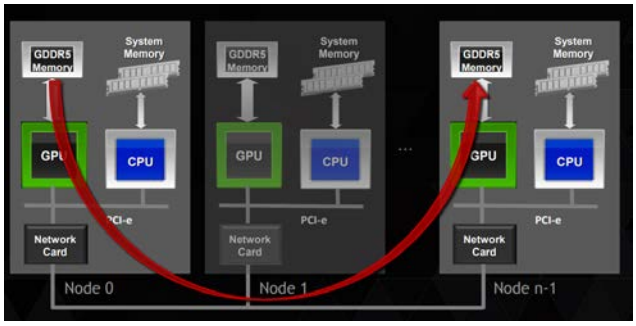

Fig. 1 Multi GPU Programming with MPI and OpenACC [10]

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 [11]. 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.

2.4 Common Run-Time Errors in Parallel Applications

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 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 [12], 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.

2.4.1 Deadlock

A deadlock is a situation in which a program is waiting for an indefinite amount of time. In other words, one or more threads in a group are permanently blocked without consuming CPU cycles. The deadlock has two types: resource and communication deadlock. Resource deadlock is a situation where a thread waits for another thread resource to proceed.
Similarly, communication deadlock occurs when some threads wait for some messages but never receive them. The causes of deadlock are different depending on the programming model used, system nature, and behavior. Once the deadlock occurs, it is not difficult to detect, but in some cases, it is difficult to detect them before they happen, as they occur under certain interleaving [13]. Finally, deadlocks in any system could be either potential or real deadlocks.

2.4.2 Livelock

Livelock is similar to deadlock, except that livelock is a situation in which 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 consume CPU cycles in response to changes in state of the other threads without doing any useful work. As a result, none of the processes will make any progress and will not complete [14]. In a livelock, the thread might not be blocked forever, and it is hard to distinguish between livelock and long-running processes. Also, livelock can lead to performance and power consumption problems because of the useless busy-wait cycles.

2.4.3 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 [15]. Also, it can happen when there are two memory accesses in the program both are performed concurrently by two threads or targeting the same location. For example, at least one read and one write may happen at the same memory location, at the same time.

2.5 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 [16]–[18]. 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.

3. Related Works

There are many studies have been done in software testing for HPC and parallel software. These studies are varied for different purposes and scopes. These variations include testing tools or detection of a specific type of error or for a different type of error. Some studies focus on using static testing techniques [19]–[21] to detect errors by analyzing the source code to find real as well as potential run-time errors [22], [23], dynamic testing techniques [13], [24] to detect errors after execution and at run time, or hybrid-testing techniques [25]–[27]. In addition, detecting errors in programming models also vary from the testing tool for single-level programming models to the tri-level programming model. Even in the same classification of programming model, there is variation in testing the programming models themselves because each programming model has a different type of error to detect as discussed earlier in Section 2.4.

For detecting a specific type of error, there are many research works on detecting deadlock, livelock, and race condition by using different techniques. In deadlock detection, many tools and studies use static or dynamic testing techniques to detect deadlocks including resource and communication deadlocks. UNDEAD [28] is intended for deadlock detection and prevention, which helps to defeat deadlocks in production software with enhancing run-time performance and memory overheads. Regarding detecting data race, the hybrid test-driven approach has been introduced in [26] to detect data race in task-parallel programs. Finally, some livelock detection techniques have been proposed in [14], [29].

Regarding testing the programming model, many approaches have been introduced to test and detect errors in parallel software. Many studies have been done with single-level programming models such as MPI [24], [30], OpenMP [35]–[37], and CUDA [35], [36], 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 [29], [41] .

Many existing HPC debuggers include both commercial and open-source versions. One commercial debugger is ALLINEA DDT [34], which supports C++, MPI, OpenMP, and Pthreads, and has been designed to work on all scales, including Petascale. The other is TotalView [33], which supports MPI, Pthreads, OpenMP and CUDA. However, these debuggers do not help to test or detect errors, but are used to find the reasons behind those errors. Also, the developer needs to select the thread, process, and kernel to be investigated.

In terms of open-source testing tools, ARCHER [27] is a data race detector for an OpenMP program that combines static and dynamic techniques to identify data race in large OpenMP applications. In addition, AutomaDeD [41] (Automata-based Debugging for Dissimilar Parallel Tasks) is a tool that detects MPI errors by comparing similarities and dissimilarities between tasks. MEMCHEKER [30] allows finding hard-to-catch memory errors in MPI applications such as overwriting of memory regions used

in non-blocking communication and one-sided communication. Furthermore, MUST [24] detects run-time errors in MPI and reports them to the developers, including MPI deadlock detection, data type matching, and detection of communication buffer overlaps.

There are limited studies of OpenACC in testing and detecting static and dynamic errors. There are some studies regarding related OpenACC testing. In [42], they evaluate three commercial OpenACC compilers by creating a validation suite that contains 140 test cases for OpenACC 2.0. They also check conformance, correctness, and completeness of certain 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 [43], in which three commercial compilers were evaluated, including CAPS, PGI, and CRAY. Similarly, this OpenACC test suite was published in [44] 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 [45], which considered detecting numerical differences that can occur due to computational differences on different OpenACC directives. They proposed a solution for that by generating code from the compiler to run each computer 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 determine what data to compare.

Despite the efforts made in creating and proposing software-testing tools for parallel application, there is still much to be done, especially for OpenACC and dual-programming models for heterogeneous systems. Finally, to the best of our knowledge, there is no parallel testing tool designed to test applications programmed by using OpenACC or the dual-programming model (MPI + OpenACC) to detect their run-time errors.

4. Proposed Architecture

We propose a parallel hybrid-testing tool for the dual-programming model (MPI + OpenACC) and C++ programming language as shown in Figure 2 and 3. This architecture has the flexibility to detect potential run-time errors and report them to the developer, detect them automatically by using assertion statements, 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 source code includes C++ programming language and (MPI + OpenACC) as dual-programming models. The part that displayed in Figure 2 is responsible for detecting real and potential run-time errors by using static testing, producing 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 dynamic testing. In addition, any real run-time errors also will be given to the developer with a warning message, as these errors must be corrected because they will definitely occur during run time. Also, these real run-time errors that have been discovered in the source code can be automatically corrected before the process move to the dynamic testing part, which reduces testing time and enhances testing performance. The static part of the architecture includes:

- **Lexical analyzer:** which will take the source code that includes C++, MPI, and OpenACC as inputs. The analyzer will understand the source code because it has all the information related to the programming language and the determined programming models. Then, it will convert the application source code into tokens and further arrange them into tables of tokens. The output of this analyzer will be a token table, which includes token names and their respective types.
- **Parser:** which is responsible for analyzing the syntax of the input source code and confirming the rules of formal grammar. This process will produce a structural representation of the input (Parser Tree) that shows the syntax relationship to each other, checking for correct syntax in the process.
- **State transit graph generator:** which 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.
- **State graph comparator:** takes the graph for the user program as an input and compares it with the state graphs of each programming language and model. This comparator has access to state graph libraries, which include the respective programming language and model with the correct grammar for each of them. As a result, any differences in these comparisons will be provided in a list of potential run-time errors, as well as some real run-time errors that can be detected by the static part of the architecture. The real run-time errors will be delivered to the developer to correct them because they undoubtedly occur if they do not be corrected. The potential run-time errors will be a move to the assertion language inserting and then instrumented to be considered in the dynamic part of the proposed architecture.

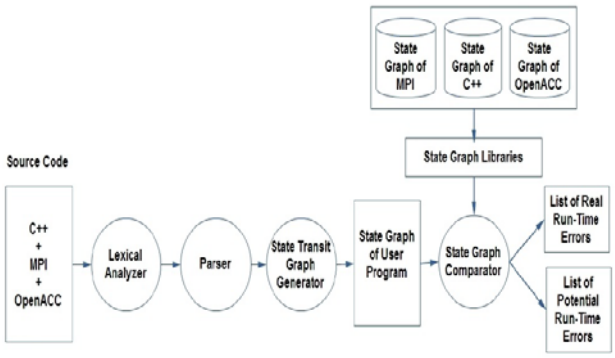

Fig. 2 Static Part of the Proposed Architecture

The dynamic testing part of the proposed architecture is shown in Figure 3, which takes the source code and the assertion statements as an input and move them to the instrumental level. 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 which are written 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 greater code size as it will have both the user code and testing code. The second method is adding the assert statements calling 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 in which any testing that needs 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 would be repeated many times, while with this method it will be written only once and called multiple times.

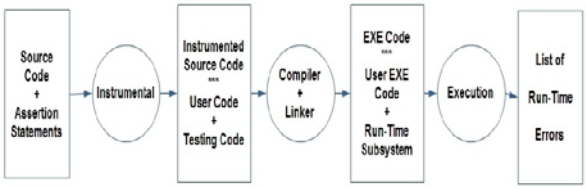

Fig. 3 Dynamic Part of the Proposed Architecture

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.

5. Discussion

There are many available tools, and studies have been done to detect run-time errors that occur in parallel systems that use MPI, CUDA, and OpenMP programming models. However, even though OpenACC can work in heterogeneous architecture, hardware, and platforms, as well as being used by non-computer science specialists, it can easily have several errors. There is no research or testing tool that detects OpenACC run-time errors. Also, OpenACC is increasingly being used in different research fields as one of the main programming models that target Exascale systems. Recently, OpenACC has been used in five of 13 applications to accelerate top supercomputer performance at the World Summit. Three of the top five HPC applications are using OpenACC as well. Therefore, this increase in using OpenACC will generate more errors that need to be detected.

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

The second part of the hybrid technique is a dynamic testing technique for errors that happen 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 in terms of covering every possible scenario of the test cases and data. Furthermore, these dynamic techniques are sensitive to the execution environment and can thus affect the system execution time. Finally, the run-time error type and behavior determines what techniques will be used because static analysis and other methods cannot detect dynamic technique errors.

6. Conclusion and Future work

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 implement our architecture and evaluate its ability to detect OpenACC run-time errors, also identifying and detecting run-time errors from the dual-programming model (MPI + OpenACC). Our experiments will be conducted using an AZIZ supercomputer, which is one of the top ten supercomputers in the Kingdom of Saudi Arabia. In June 2016, AZIZ was ranked No. 359 among the top 500 supercomputers in the world.

Acknowledgments

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

[7] A. Fu, D. Lin, and R. Miller, “Introduction to OpenACC,” 2016.

[8] 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.

[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. Kraus and P. Messmer, “Multi GPU programming with MPI,” in GPU Technology Conference, 2014.

[11] 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.

[12] OpenACC Standards, “The OpenACC Application Programming Interface version 2.7,” 2018.

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

[14] 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.

[15] 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.

[16] A. Karpov and E. Ryzhkov, “Adaptation of the technology of the static code analyzer for developing parallel programs,” Program Verification Systems, 2018. [Online]. Available: https://www.viva64.com/en/a/0019/.

[17] A. A. Sawant, P. H. Bari, and P. . Chawan, “Software Testing Techniques and Strategies,” J. Eng. Res. Appl., vol. 2, no. 3, pp. 980–986, 2012.

[18] A. Karpov, “Testing parallel programs,” Program Verification Systems, 2018. [Online]. Available: https://www.viva64.com/en/a/0031/.

[19] 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.

[20] 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.

[21] 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.
[22] E. Saillard, P. Carribault, and D. Barthou, “Static/Dynamic validation of MPI collective communications in multi-threaded context,” ACM SIGPLAN Not., vol. 50, no. 8, pp. 279–280, Jan. 2015.
[23] 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.
[24] RWTH Aachen University, “MUST: MPI Runtime Error Detection Tool,” 2018.
[25] 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.
[26] R. Surendran, “Debugging, Repair, and Synthesis of Task-Parallel Programs,” RICE UNIVERSITY, 2017.
[27] Lawrence Livermore National Laboratory, University of Utah, and RWTH Aachen University, “ARCHER,” GitHub, 2018. [Online]. Available: https://github.com/PRUNERS/archer.
[28] 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.
[29] M. K. Ganai, “Dynamic Livelock Analysis of Multi-threaded Programs,” in Runtime Verification, 2013, pp. 3–18.
[30] The Open MPI Organization, “Open MPI: Open Source High Performance Computing,” 2018. [Online]. Available: https://www.open-mpi.org/.
[31] 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.
[32] E. Saillard, P. Carribault, and D. Barthou, “Static Validation of Barriers and Worksharing Constructs in OpenMP Applications,” in Using and Improving OpenMP for Devices, Tasks, and More, vol. 8766, 2014, pp. 73–86.
[33] R. W. S. Inc., “TotalView for HPC,” 2018. [Online]. Available: https://www.roguewave.com/products-services/totalview.
[34] Allinea Software Ltd, “ALLINEA DDT,” ARM HPC Tools, 2018. [Online]. Available: https://www.arm.com/products/development-tools/hpc-tools/cross-platform/forge/ddt.
[35] 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.
[36] Mai Zheng, V. T. Ravi, Feng Qin, and G. Agrawal, “GMRace: Detecting Data Races in GPU Programs via a Low-Overhead Scheme,” IEEE Trans. Parallel Distrib. Syst., vol. 25, no. 1, pp. 104–115, Jan. 2014.
[37] 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.
[38] E. Saillard, P. Carribault, and D. Barthou, “MPI Thread-Level Checking for MPI+OpenMP Applications,” in EuroPar, vol. 9233, 2015, pp. 31–42.
[39] B. Klemme, “Software Testing of Parallel Programming Frameworks,” University of New Mexico, 2016.
[40] T. Hilbrich, M. S. Müller, and B. Krammer, “MPI Correctness Checking for OpenMP/MPI Applications,” Int. J. Parallel Program., vol. 37, no. 3, pp. 277–291, 2009.
[41] 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.
[42] 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.
[43] 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.
[44] K. Friedline, S. Chandrasekaran, M. G. Lopez, and O. Hernandez, “OpenACC 2.5 Validation Testsuite Targeting Multiple Architectures,” 2017, pp. 557–575.
[45] 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.