Software Testing Techniques for Parallel Systems: A Survey

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

King Abdulaziz University, Jeddah, KSA

Summary
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. However, building systems with different programming models is 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. We are surveying the existing testing tools that test parallel systems to detect run-time errors. We classify the reviewed testing tools in different categories and sub-categories based on used testing techniques, targeted programming models, and detected run-time errors. Despite the effort of building testing tools for parallel systems, much work still needs to be done, especially in testing heterogeneous and multi-level programming models. Hopefully, these efforts will meet the expected improvement in HPC systems and create more error-free systems.

Keywords:
High-Performance Computing; Software Testing; Testing Techniques; Testing Tools Classifications; Exascale Systems; Programming Models;

1. Introduction

High-Performance Computing (HPC) currently is a part of all scientific and manufacturing sectors driven by the improvement of HPC machines, especially with the growing attention to Exascale supercomputers, which has been suggested to be feasible by 2022 by different studies [1]. This continuous improvement poses the challenging task of building massively parallel systems that can be used in these super machines. Parallel applications have to be error-free to satisfy the application's requirements and benefits from the used programming model's abilities and features. It is very difficult to test such applications because of their huge sizes, changeable behavior, and the integration between different programming models in the same application.

This paper tries to investigate different available testing tools and techniques that detect run-time errors in parallel applications that use programming models, including homogeneous and heterogeneous systems. We will review and study different tools and techniques and classify them depending on different factors and characteristics. This review will investigate what still needs to be done in testing parallel systems and directions for future research in this field. Also, we only study deeply some of the reviewed testing tools and techniques based on their relation to our subject.

The rest of this paper is structured as follows. Section 2 briefly gives an overview of some programming models, their levels, and some run-time errors, as well as some testing techniques, which will be discussed in Sections 3 and 4. Our classification for the reviewed testing tools will be discussed in Sections 5 and 6. In Section 7, we will give an overview of Exascale-related concepts and their effects on the software testing. Finally, we will conclude our work in Section 8.

2. Overview of Programming Models

In recent years, building massively parallel supercomputing systems based on heterogeneous architecture has 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 that have different features and are created for different purposes; including message passing such as MPI [2] and shared memory parallelism such as OpenMP [3]. In addition, some programming models support heterogeneous systems, which consisting of a Graphics Processing Unit (GPU) coupled with a traditional CPU. Heterogeneous parallel

Manuscript received April 5, 2019
Manuscript revised April 20, 2019

programming models are CUDA [4] and OpenCL [5], which are low-level programming models, and OpenACC [6] is a high-level heterogeneous programming model. In this section, some of these programming models will be discussed and explained.

2.1 MPI

Message-Passing Interface (MPI) [2] 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, subroutines or methods for programming languages including Fortran, C, and C++. MPI has several implementations, including open source implementations, such as Open MPI [7] and MPICH [8]; and commercial implementations, such as IBM Spectrum MPI [9] and Intel MPI [10]. MPI has several features and advantages, including [11] and [12]:

- **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 in several platforms, hardware, systems, and programming languages. Also, MPI can work perfectly with several programming models and 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, over 430 routines have been defined, including the majority of the previous versions of MPI.

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

2.2 OpenMP

OpenMP (Open Multi-Processing) [13] is a shared memory programming model for Fortran and C/C++ programs. The first OpenMP version was released in October 1997 by the OpenMP Architecture Review Board (ARB) for Fortran. In the following year, OpenMP supported C/C++ programs as well. The latest version, OpenMP 5.0, was released in November 2018 with three main API components that include compiler directives, run-time library routines, and environment variables. OpenMP has several features that include:

- Providing a standard for various shared memory platforms and architectures as well as for several hardware and software vendors.
- Providing the ability to implement both coarse-grain and fine-grain parallelism.
- OpenMP has been implemented in several platforms, hardware, systems, and programming languages.

OpenMP has been implemented in many compilers including various vendors or open-source communities, such as GNU Compiler Collection (GCC), Oracle Developer Studio compilers, Intel Parallel Studio XE, as well as other open-source and commercial compilers and tools that support different versions of OpenMP.

2.3 CUDA

CUDA (Compute Unified Device Architecture) [14] is a parallel computing platform and programming model that supports the use of graphics processing units (GPU) for general-purpose processing, which increases the computing performance dramatically. This technique is called General-Purpose computing on Graphics Processing Units (GPGPU). The first version of CUDA was introduced in November 2006 by NVIDIA Corporation that only targeting NVIDIA GPUs, which can be considered as a lack of portability. However, CUDA is considered as one of the most widely used programming models for GPUs, which are designed to support Fortran and C/C++ programs. In March 2018, the latest CUDA version 9.2 was released. CUDA has several features that include enabling efficient use of the massive parallelism of NVIDIA GPUs. CUDA is a low-level programming model that gives the code the ability to read from an arbitrary address in memory. That gives developers the ability to know some details such as transfer between the host and the memory of the device, temporary data storage, kernel boot time mapping of threads, and parallelism.

2.4 OpenCL

OpenCL, which stands for Open Computing Language, is an open standard for general-purpose parallel programming of heterogeneous systems that include CPUs, GPUs, and other processors [15]. Also, OpenCL is a low-level programming model that is similar to CUDA that supports several applications ranging from consumer and

embedded software to HPC solutions. However, OpenCL is more complex and has a very low-level API that requires significantly more code than CUDA [16]. The first OpenCL 1.0 was released in December 2008 by the Khronos™ Group, Inc., and the latest version was released in May 2017. Also, OpenCL is compatible with C/C++ programming languages and has been implemented in several platforms and companies, including NVIDIA Corporation, Intel, IBM Corporation, Apple Inc., and AMD as well as others. OpenCL has the key feature of portability, which makes it possible to run any OpenCL kernel on any conformant implementation.

2.5 OpenACC

In November 2011, OpenACC stands for open accelerators and was released at the first time in the International Conference for High-Performance Computing, Networking, Storage, and Analysis [17]. OpenACC is a directive-based open standard developed by Cray, CAPS, NVIDIA, and PGI. They designed OpenACC to create simple high-level parallel programming model for heterogeneous CPU/GPU systems that are 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." [6]. The latest version of OpenACC was released in November 2018. OpenACC has several features and advantages compared with other heterogeneous parallel programming models, including:

- Portability: Unlike a programming model like CUDA that works only on NVIDIA GPU accelerators, OpenACC is portable across different types of GPU accelerators, hardware, platforms, and operating systems [6], [18].
- OpenACC is compatible with various compilers and gives flexibility to the compiler implementations.
- It is a high-level programming model, which makes targeting accelerators easier by hiding low-level details. For generation of low-level GPU programs, OpenACC relies on the compiler using the programmer codes.
- Better performance with less programming effort, which gives the ability to add GPU codes to existing programs with less effort. That will lead to reducing the programmer workload and improvement in programmer productivity and achieving better performance than OpenCL and CUDA [19].
- 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” [6].

3. Overview of Some Common Run-Time Errors

There are several types of run-time errors that happen 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 or causes. For example, deadlock in MPI has different causes and behaviors compared with OpenACC deadlocks. Also, run-time errors in the dual-programming model are different. Also, some run-time errors happen specifically in a particular programming model. By investigating the documents of the latest version of OpenACC 2.7 [20], 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 a run-time error. This case happened in a non-shared memory device 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, and to detect them in applications developed by dual-programming model is even more complicated. In the following, some popular run-time errors will be displayed and discussed in general with some examples.

3.1 Deadlocks

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: resource and communication deadlock. Resource deadlock is a situation in which a thread waits for another thread resource to proceed. Similarly, communication deadlock occurs when some threads wait for some messages, but they never receive these messages [21], [22]. The reasons that cause deadlock are different depending on the used programming models, system nature, and behavior. Once the deadlock occurs it is not difficult to detect, but in some cases, it is difficult to detect them beforehand, as they

occur under certain interleaving [23]. Finally, deadlocks in any system could be a potential deadlock that may or may not happen based on the execution environment and sequence, or real deadlocks, which definitely will occur.

3.2 Livelocks

Livelock is similar to deadlock, except that a livelock is a situation that arises when two or more processes change their state continuously in response to changes in 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 be completed [24], [25]. In a livelock, the thread might not be blocked forever, and it is hard to distinguish between livelock and a long-running process. Also, livelock can lead to performance and power consumption problems because of the useless busy-wait cycles.

3.3 Race Condition

A race condition is a situation that might occur 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 [26]. Some researchers do not differentiate between data race and race condition, which will be explained in the data race definition.

3.4 Data Race

A data race happened when there are two memory accesses in the program both perform concurrently in two threads or target the same location [27]. For example, at least one read and one write may happen at the same memory location at the same time. The race condition is a data race that causes an error. However, the data race does not always lead to a race condition.

3.5 Mismatching

Mismatching is a situation that happens in arguments with one call, which can be detected locally and are sometimes even detected by the compiler. Mismatching can take several forms, including wrong type or number of argument, arguments involving more than one call, or in collective calls. Developers need to pay special attention when comparing matched pairs of derived data types.

4. Overview of Testing Techniques

There are many techniques used in software testing, which include static and dynamic as well as other techniques. Static testing is the process of analyzing the source code before the compilation phase for detecting static errors. It handles the application source code only without launching it, which gives us the ability to analyze the code in detail and have full coverage. By contrast, the static analysis of parallel application is complicated due to unpredicted program behavior, which is parallel application nature [29], [30]. However, it will be very useful 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 launching programs that are sensitive to the execution environment, and slow down the speed of application execution. It is useful to use dynamic analysis in a parallel application, which affords 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 whether errors are corrected or hidden.

Symbolic testing [31], [32] is a technique that allows the automatic exploration of paths in a program. It works by deriving a general representation of the program behavior from the program code. The concrete inputs of the program unit will be replaced with symbolic values, and the execution of the program will be simulated so that instead of values, all variables hold symbolic expressions. Also, symbolic execution can analyze programs to determine what inputs cause each part of the program to be executed. This technique has been used to detect run-time errors and creating testing tools. However, this technique has several limitations such as path explosion, which makes it impossible to scale with large programs.

Hybrid testing is combining more than one of the mentioned testing techniques, which gives the ability to cover a wider range of run-time errors. This combination takes the advantages of two testing techniques, reduces disadvantages, and reduces the testing time. The combination of static and dynamic testing is the most used hybrid testing techniques to detect run-time errors in parallel systems using programming models. Finally, it is the run-time error type and behavior that determine which techniques will be used because some errors cannot be detected by static analysis, and others cannot be detected by dynamic techniques.

5. Testing Techniques Classifications

In our study, 58 different testing tools and techniques have been reviewed, varying from open-source to commercial tools. Including different types of testing techniques, targeted programming models discovered run-time errors for different purposes when discovering the errors or tracking the cause of these errors (debuggers). Only the parallel systems-related testing has been included in our study, where we are surveying the parallel systems testing techniques. Also, we focus on the testing techniques used to detect run-time errors that occur on parallel systems, which include the explained run-time errors in Section 3. We chose the tools and techniques in our study from more than 100 testing tools and techniques. We eliminate any tool or technique that does not meet our objectives. We aim to survey testing tools and techniques that detect run-time errors in parallel systems that use programming models.

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**, which includes an individual programming model such as MPI, OpenMP, or CUDA.
- **Dual-Level Programming Model**, which is a combination of two programming models working together to enhance parallelism such as MPI + OpenMP and MPI + CUDA. Many types of research refer to this level of parallelism by MPI + X.
- **Tri-Level Programming Model**, which combines three different programming models to work together as a hybrid programming model. Some studies refer to this level with MPI + X + Y.

As a result, we classify the used testing techniques into four categories, which include Static, Dynamic, Hybrid, and Symbolic testing techniques. Also, we classify these techniques into two subcategories to determine the targeted programming model level, where they are single- or dual-level programming models. The following subsections will discuss our classifications.

5.1 Static Testing Techniques

Six testing tools have been classified as testing tools that use a static testing technique. These tools have used this technique to detect run-time errors in the parallel program that use programming models.

1. Single-Level Programming Model Testing Techniques

The testing tools [33], [34], and GPUVerify [35] have used the static technique to detect data race in CUDA, OpenCL, and OpenMP programming models individually. For OpenMP, the testing tools [36], [37], ompVerify [38] have been used to detect data race. Finally, MPI-Checker [39] used static techniques to detect MPI mismatching errors.

2. Dual-Level Programming Model Testing Techniques

In the reviewed testing tools, there are no testing tools that used static testing techniques for detecting run-time errors in the dual-level programming model.

5.2 Dynamic Testing Techniques

In our survey, there are 34 testing tools that use dynamic testing techniques for detecting run-time errors in parallel programs. These tools will be classified based on the targeted programming models as follows:

1. Single-Level Programming Model Testing Techniques

Regarding detecting errors in the MPI programming model, 14 testing tools use a dynamic technique that targets MPI. The testing tools MEMCHECKER [7], MUST [40], [41], STAT [42], Nasty-MPI [43], and Intel Message Checker [44] have been used to detect MPI run-time errors including deadlocks, data race, and mismatching. For detecting deadlocks and mismatching, the following tools are used, including MPI-CHECK [45], GEM [46], and Umpire [24]. The tools PDT [47], MAD [48], and [49] are used to detect deadlocks and race conditions in MPI. For deadlocks, MOPPER [50] and ISP [51] are used. Finally, MPIRace-Check [52] has been used to detect race condition in MPI.

For OpenMP run-time error detection, there are several tools such as Intel Thread Checker [53] and Sun Thread Analyzer [53] that detect deadlock and data races. Also, VORD [54] and [55] are used to detect the data race in OpenMP. RTED [56] is using dynamic testing to detect deadlocks and race conditions in MPI and OpenMP individually. Also, NR [57], RaceStand [58], eNR Labeling [59], and [60] are for OpenMP data race detection.

For detecting data race in CUDA using dynamic techniques, the testing tool in [61] is used. Regarding data race detection, there are several testing tools for different programming models including; GUARD [62], RaceTM [63] and KUDA [64] for CUDA. For detecting errors in heterogeneous programming model by using dynamic

testing, WEFT [65] is used to detect deadlocks and race conditions.

Finally, deadlocks detection in parallel programs using UNDEAD [22], Sherlock [66], and ConLock [23], and livelock detection can be seen in CBuster.

2. Dual-Level Programming Model Testing Techniques

For testing the hybrid MPI/OpenMP programming model using dynamic testing, the testing tools MARMOT [67] and [68] have been used for detecting deadlocks, race conditions, and mismatching.

5.3 Symbolic Testing Techniques

In our study, there are six testing tools that use symbolic techniques for detecting run-time errors for both single and- dual-level programming models. These tools will be classified by their programming models as the following:

1. Single-Level Programming Model Testing Techniques

KLEE-CL [31] used the symbolic technique for detecting data race in the OpenCL programming model. Also, this technique has been used in OAT [69] for detecting deadlocks and data race in OpenMP. The testing tools GKLEE [70], GKLEEP [71], and PUG [72] was used for testing CUDA programming model to detect deadlocks, data races, and race conditions using symbolic techniques.

2. Dual-Level Programming Model Testing Techniques

In the reviewed testing tools, there are no testing tools that used symbolic techniques for detecting run-time errors in the dual-level programming model.

5.4 Hybrid Testing Techniques

Several reviewed testing tools used hybrid testing techniques, which combine static/dynamic testing or combine static/symbolic testing. In our survey, seven tools used hybrid testing techniques. These tools will be classified into the following subsections by their programming model levels.

1. Single-Level Programming Model Testing Techniques

In this subcategory, five testing tools targeted single-level programming models that include OpenMP, CUDA, and OpenCL. ARCHER [73] and Dragon [74] are testing tools that use hybrid testing techniques to detect data race in OpenMP programming model. GMRace [75], GRace [76], [77], and SESA [78] use hybrid testing techniques to detect data race in the CUDA programming model. Finally, GRace is also used to test the OpenCL programming model for detecting data race. All the previous testing tools used static/dynamic hybrid testing techniques to detect run-time errors except SESA, which used static/symbolic hybrid testing techniques.

2. Dual-Level Programming Model Testing Techniques

Two testing tools used static/dynamic hybrid testing techniques to detect run-time errors in the MPI/OpenMP dual programming model. These tools are PARCOACH [79] and [80], which used the hybrid model to detect deadlocks and other run-time errors resulting from the dual programming model. Even though combining two programming model is beneficial, it creates complex run-time errors that are difficult to detect and determine.

It is noticeable that five testing tools are classified as debugging because we cannot determine the testing techniques that have been used in those tools. These tools are AutomaDeD [81], which is a tool that detects MPI errors by comparing similarities and dissimilarities between tasks. The second tool is ALLINEA DDT [82], which is a commercial debugger that supports MPI, OpenMP, and Pthreads. The third is TotalView [83], which supports MPI, Pthreads, OpenMP, and CUDA. Finally, PDT [47] and MPVisualizer [84] are detecting deadlocks and race conditions in MPI. We note that these five are debuggers that do not help to test or detect errors, but can be used to find out the reasons behind errors. As a result, we could not classify them based on the testing techniques.

The following Table 1 displays the number of testing techniques used for each programming model in the reviewed testing tools. We notice that the dynamic techniques have been mostly used to test MPI and OpenMP for detecting run-time errors. Symbolic testing has been used to detect run-time errors in CUDA. However, OpenACC has not been targeted to be tested by any reviewed testing tools.

Table 1: Relationship between the Used Testing Techniques and the Targeted Programming Models

	Programming Models					
Testing Techniques	MPI	OpenACC	CUDA	OpenCL	OpenMP	Hybrid (MPI/OpenMP)
Static	1	0	2	2	4	0
Dynamic	14	0	4	1	10	2
Symbolic	0	0	4	1	1	0
Hybrid (Static/Dynamic)	0	0	1	0	2	2
Hybrid (Static/Symbolic)	0	0	2	1	0	0
Debugging	5	0	1	0	2	0

To summarize our classifications of the reviewed testing tools, the following three figures will be displayed. Firstly, Figure 1 displays the reviewed testing tools classified by the used testing techniques, showing that the dynamic testing techniques have been used more than other techniques to detect run-time errors.

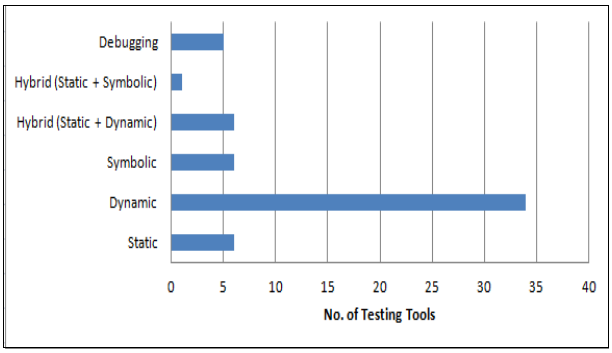

Fig. 1 Testing Tools Classified by Used Testing Techniques

Then, Figure 2 shows the testing tools for testing targeted programming models. It is notable that MPI, OpenMP, and CUDA have been tested in several testing tools, which are considered the most targeted programming models in our survey. However, OpenACC has not been targeted as a tested programming model.

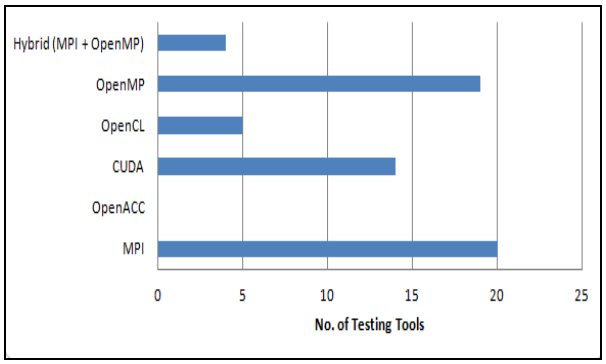

Fig. 2 Testing Tools Classified by Targeted Programming Models

Finally, several run-time errors have been detected in the reviewed testing tools. Figure 3 shows a summary of these errors. Deadlocks and data races are the most detected run-time errors in the reviewed testing tools.

6. Discussion

As we reviewed and presented in the previous section, there are several aspects that need to be discussed and taken into consideration. Firstly, we only reviewed testing tools that target parallel systems, especially systems that use programming models. Our study tries to comprehensively review tools that test parallel systems to detect run-time errors because of its unpredictable behavior and the causes behind them to occur. Usually, compile-time errors can be detected by compilers and reported to developers to be corrected. Furthermore, detecting run-time errors in parallel systems is even more complicated because of the different behaviors of the programming models and their interaction with different programming models.

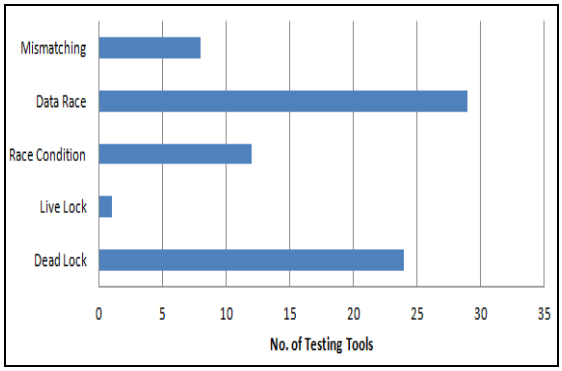

Fig. 3 Testing Tools Classified by Detected Run-Time Errors

Secondly, we classify the reviewed testing tools by the testing techniques used to detect run-time errors. This classification makes it easy to compare different techniques and discovers their features in finding run-time errors and their limitations as well. In this classification, we also classify them into subcategories based on the level of programming models used in the tested parallel systems. We focus on the single- and dual-level programming models because they are widely used in developing parallel systems for different purposes compared to tri-level programming models. Also, the lack of testing tools that target tri-level programming models makes it hard to review and classify the reviewed testing tools into three subcategories. As a result, we classified our reviewed testing tools into what we did in Section 3.

Finally, we noticed that dynamic testing techniques had been used mainly for detecting run-time errors for different programming models. Single-level programming models have been targeted to be tested, especially MPI and OpenMP because of their wide use and their history in programming models. Regarding heterogeneous programming models, CUDA is the most targeted programming model in the reviewed testing tools, while OpenACC has not been targeted in any testing tools, despite their benefits and trending use. We believe that a lot of work needs to be done in creating and developing testing tools for massively parallel systems, especially

heterogeneous parallel systems, which will be needed when the Exascale systems are applied in different fields.

7. Towards Exascale Testing

Over the next few years, an extreme level of computing systems will be feasible by 2022, which leads to building more powerful supercomputers [85]. Exascale systems are expected to make a revolution in computer science and engineering by providing at least one exaFLOPS, which are 1018 floating-point operations per second [86]. In the recent top 500 supercomputers list, SUMMIT is the top supercomputer in the world by 200 pteaFOLPS [1], [87]. SUMMIT is an IBM-built supercomputer running at the Department of Energy's (DOE) Oak Ridge National Laboratory (ORNL), USA. SUMMIT has 4,608 nodes, each one equipped with two 22-core IBM POWER9™ CPUs, and six NVIDIA Tesla V100 GPUs. The nodes are linked together with a dual-rail Mellanox EDR 100G InfiniBand interconnect. Also, SUMMIT processes more than ten petabytes of memory, paired with fast high-bandwidth pathways for more efficiency in data movement [87]. They claim that SUMMIT will take one step to achieving Exascale computing by 2021[1].

As we noticed, high-performance computing has become increasingly important, and Exascale supercomputers will be feasible soon. Therefore, building massively parallel supercomputing systems based on heterogeneous architecture is even more important to increase parallelism. The majority of the top 500 supercomputers in the world use GPUs to enhance performance and parallelism. For example, in the recent list, more than 78% of the top supercomputers used NVIDIA accelerators, and 48% of them used NVIDIA Tesla P100 [88].

The movement towards Exascale can be achieved by hardware as well as software, as we noted in the previous paragraph. The integration between more than one programming models, which is a dual-level or tri-level programming model, will help to achieve Exascale computing. Furthermore, this improvement in computing systems comes with cost and difficulties, regarding building massively parallel systems, reducing energy consumption as well as testing these systems. To create massively parallel systems more than one programming models are needed, especially heterogeneous programming models to support the increasing use of GPUs in supercomputers, as well as to benefit from their features. For some systems, hybrid dual- or even tri-level programming models will be benefited and useful. However, testing parallel applications is not an easy task because the natures of errors in parallel systems are hard to detect due to the non-determined and unexpected behavior of the parallel application. Even after detecting the errors and modifying the source code, it is difficult to determine whether the errors have been corrected or hidden. Integrating different programming models inside the same application make it even more difficult to test. Despite the available testing tools that detect static and dynamic errors, there is still a lack of such a testing tool that detects run-time errors in parallel systems implemented in the heterogeneous programming model.

8. Conclusion

Testing parallel systems that use heterogeneous programming models has become increasingly important, and the movement to Exascale systems makes it even more important to avoid errors that could affect the system requirements, not only errors that can be detected by compilers, but also more critical errors that occur after compilation and cannot be detected by the compilers. As a result, testing tools have been built, and different testing techniques have been used to detect static and run-time errors in parallel systems. These tools and techniques are targeting systems built by several types and levels of programming models.

We studied more than 50 testing tools and classified them according to the used testing techniques, the targeted programming models, and the run-time errors. We tried to discover the limitations and open areas for the researchers in testing parallel systems, which can yield the opportunity to focus on those areas. Before that, we gave an overview of the testing techniques, popular programming models and their different levels, and some common run-time errors that affect parallel systems.

There is an increasing importance of parallel systems in the Exascale era; but there is a shortfall in testing those systems, especially parallel systems that use heterogeneous programming models including high- and low-level programming models. 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.

To conclude, there has been good effort in testing parallel systems to detect run-time errors, but still not enough, especially for systems using heterogeneous programming models, as well as dual- and tri-level programming models. The integration of different programming models in the same system will need new testing techniques to detect run-time errors in heterogeneous parallel systems, which will be addressed in our future work. We believe that to achieve good systems that can be used in Exascale supercomputers, we should focus on testing those systems because of their massively parallel natures as well as their huge size, which add more difficulties and issues. Also, these testing tools should integrate more than one testing techniques and work in parallel to detect run-time errors by creating testing threads, depending on the number of application threads. As a result, using parallel hybrid techniques will enhance testing time and cover a wide range of run-time errors.

Acknowledgments

This project was funded by the Deanship of Scientific Research (DSR), King Abdulaziz University, Jeddah, under grant No. (DG-015-611-1440). The authors, therefore, gratefully acknowledge the DSR technical and financial support.

References

[1] 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.

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

[3] OpenMP Architecture Review Board, “About OpenMP,” *OpenMP ARB Corporation*, 2018. [Online]. Available: https://www.openmp.org/about/about-us/.

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

[5] Khronos Group, “About OpenCL,” *Khronos Group*, 2017. [Online]. Available: https://www.khronos.org/opencl/.

[6] OpenACC-standard.org, “About OpenACC,” *OpenACC Organization*, 2017. [Online]. Available: https://www.openacc.org/about.

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

[8] MPICH Organization, “MPICH,” 2018. [Online]. Available: http://www.mpich.org/.

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

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

[11] E. Gabriel *et al.*, “Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation,” pp. 97–104, 2004.

[12] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI:Portable Parallel Programming with the Message-Passing Interface*. 2014.

[13] B. Barney, “OpenMP,” *Lawrence Livermore National Laboratory*, 2018. [Online]. Available: https://computing.llnl.gov/tutorials/openMP/#Introduction.

[14] M. Harris, “CUDA,” *GPGPU.org*, 2018. [Online]. Available: http://gpgpu.org/developer/cuda.

[15] Khronos OpenCL Working Group, “The OpenCL™ Specification,” 2018.

[16] Khronos Group, “OpenCL,” *NVIDIA Corporation*, 2010. [Online]. Available: https://developer.nvidia.com/opencl.

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

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

[19] 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.

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

[21] 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.

[22] 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.

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

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

[25] 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.

[26] 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.

[27] M. Cao, “Efficient , Practical Dynamic Program Analyses for Concurrency Correctness,” The Ohio State University, 2017.

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

[29] 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/.

[30] 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.

[31] 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.

[32] C. Cadar and K. Sen, “Symbolic execution for software testing,” *Commun. ACM*, vol. 56, no. 2, p. 82, Feb. 2013.

[33] 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.

[34] P. Chatarasi, J. Shirako, M. Kong, and V. Sarkar, “An Extended Polyhedral Model for SPMD Programs and Its Use in Static Data Race Detection,” in *23rd International Workshop on Languages and Compilers for Parallel Computing*, 2016, vol. 9519, pp. 106–120.

[35] E. Bardsley *et al.*, "Engineering a Static Verification Tool for GPU Kernels," in *International Conference on Computer Aided Verification*, 2014, pp. 226–242.

[36] R. Nakade, E. Mercer, P. Aldous, and J. McCarthy, "Model-Checking Task Parallel Programs for Data-Race," in *NASA Formal Methods Symposium NFM 2008*, 2018, pp. 367–382.

[37] Y. Zhang, E. Duesterwald, and G. R. Gao, "Concurrency Analysis for Shared Memory Programs with Textually Unaligned Barriers," in *Languages and Compilers for Parallel Computing*, Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 95–109.

[38] V. Basupalli *et al.*, "ompVerify: Polyhedral Analysis for the OpenMP Programmer," in *International Workshop on OpenMP IWOMP 2011*, 2011, pp. 37–53.

[39] A. Droste, M. Kuhn, and T. Ludwig, "MPI-checker," in *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC - LLVM '15*, 2015, pp. 1–10.

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

[41] T. Hilbrich, M. Schulz, B. R. de Supinski, and M. S. Müller, "MUST: A Scalable Approach to Runtime Error Detection in MPI Programs," in *Tools for High Performance Computing 2009*, Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 53–66.

[42] D. C. Arnold, D. H. Ahn, B. R. de Supinski, G. L. Lee, B. P. Miller, and M. Schulz, "Stack Trace Analysis for Large Scale Debugging," in *2007 IEEE International Parallel and Distributed Processing Symposium*, 2007, pp. 1–10.

[43] R. Kowalewski and K. Fürlinger, "Nasty-MPI: Debugging Synchronization Errors in MPI-3 One-Sided Applications," in *European Conference on Parallel Processing Euro-Par 2016*, 2016, pp. 51–62.

[44] V. Samofalov, V. Krukov, B. Kuhn, S. Zheltov, A. Konovalov, and J. DeSouza, "Automated Correctness Analysis of MPI Programs with Intel Message Checker," in *Proceedings of the International Conference ParCo 2005*, 2005, pp. 901–908.

[45] G. Luecke, H. Chen, J. Coyle, J. Hoekstra, M. Kraeva, and Y. Zou, "MPI-CHECK: a Tool for Checking Fortran 90 MPI Programs," *Concurr. Comput. Pract. Exp.*, vol. 15, no. 2, pp. 93–100, 2003.

[46] A. Humphrey, C. Derrick, G. Gopalakrishnan, and B. Tibbitts, "GEM: Graphical Explorer of MPI Programs," in *2010 39th International Conference on Parallel Processing Workshops*, 2010, pp. 161–168.

[47] C. Clemencon, J. Fritscher, and R. Ruhl, "Visualization, Execution Control and Replay of Massively Parallel Programs within Annai's Debugging Tool," in *High-Performance Computing Symposium (HPCS'95)*, 1995, pp. 393–404.

[48] D. Kranzlmueller, C. Schaubschlaeger, and J. Volkert, "A Brief Overview of the MAD Debugging Activities," in *The Fourth International Workshop on Automated Debugging (AADEBUG 2000)*, 2000.

[49] 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.

[50] 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.

[51] G. Gopalakrishnan, R. M. Kirby, S. Vakkalanka, A. Vo, and Y. Yang, "ISP (In-situ Partial Order): a dynamic verifier for MPI Programs," *University of Utah, School of Computing*, 2009. [Online]. Available: http://formalverification.cs.utah.edu/ISP-release/.

[52] M.-Y. Park, S. J. Shim, Y.-K. Jun, and H.-R. Park, "MPIRace-Check: Detection of Message Races in MPI Programs," in *International Conference on Grid and Pervasive Computing GPC 2007*, 2007, pp. 322–333.

[53] C. Terboven, "Comparing Intel Thread Checker and Sun Thread Analyzer," in *Parallel Computing: Architectures, Algorithms and Applications*, 2007, vol. 38, pp. 669–676.

[54] Y.-J. Kim, S. Song, and Y.-K. Jun, "VORD: A Versatile On-the-fly Race Detection Tool in OpenMP Programs," *Int. J. Parallel Program.*, vol. 42, no. 6, pp. 900–930, Dec. 2014.

[55] Y.-J. Kim, M.-H. Kang, O.-K. Ha, and Y.-K. Jun, "Efficient Race Verification for Debugging Programs with OpenMP Directives," in *International Conference on Parallel Computing Technologies PaCT 2007*, 2007, pp. 230–239.

[56] G. R. Luecke *et al.*, "The Importance of Run-Time Error Detection," in *Tools for High Performance Computing 2009*, Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 145–155.

[57] M.-H. Kang, O.-K. Ha, S.-W. Jun, and Y.-K. Jun, "A Tool for Detecting First Races in OpenMP Programs," in *International Conference on Parallel Computing Technologies PaCT 2009*, 2009, pp. 299–303.

[58] M. Metzger, X. Tian, and W. Tedeschi, "User-Guided Dynamic Data Race Detection," *Int. J. Parallel Program.*, vol. 43, no. 2, pp. 159–179, Apr. 2015.

[59] O.-K. Ha, S.-S. Kim, and Y.-K. Jun, "Efficient Thread Labeling for Monitoring Programs with Nested Parallelism," in *International Conference on Future Generation Communication and Networking FGCN 2010*, 2010, pp. 227–237.

[60] E.-K. Ryu, K.-S. Ha, and K.-Y. Yoo, "A Practical Method for On-the-Fly Data Race Detection," in *International Workshop on Applied Parallel Computing PARA 2002*, 2002, pp. 264–273.

[61] M. Boyer, K. Skadron, and W. Weimer, "Automated Dynamic Analysis of CUDA Programs," in *Third Workshop on Software Tools for MultiCore Systems (STMCS)*, 2008.

[62] V. Mekkat, A. Holey, and A. Zhai, "Accelerating Data Race Detection Utilizing On-Chip Data-Parallel Cores," in *International Conference on Runtime Verification*, 2013, pp. 201–218.

[63] S. Gupta, F. Sultan, S. Cadambi, F. Ivancic, and M. Rotteler, "Using hardware transactional memory for data race detection," in *2009 IEEE International Symposium on Parallel & Distributed Processing*, 2009, pp. 1–11.

[64] C. Bekar, T. Elmas, S. Okur, and S. Tasiran, "KUDA: GPU accelerated split race checker," in *Workshop on Determinism and Correctness in Parallel Programming (WoDet)*, 2012.

[65] 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.

[66] M. Eslamimehr and J. Palsberg, "Sherlock: scalable deadlock detection for concurrent programs," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2014*, 2014, pp. 353–365.

[67] B. Krammer, T. Hilbrich, V. Himmler, B. Czink, K. Dichev, and M. S. Müller, "MPI Correctness Checking with Marmot," in *Tools for High Performance Computing*, Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 61–78.

[68] 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.

[69] 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.

[70] G. Li, P. Li, G. Sawaya, G. Gopalakrishnan, I. Ghosh, and S. P. Rajan, "GKLEE: Concolic Verification and Test Generation for GPUs Guodong," in *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming - PPoPP '12*, 2012, p. 215.

[71] K. Kojima, A. Imanishi, and A. Igarashi, "Automated Verification of Functional Correctness of Race-Free GPU Programs," *J. Autom. Reason.*, vol. 60, no. 3, pp. 279–298, Mar. 2018.

[72] G. Li and G. Gopalakrishnan, "Scalable SMT-based verification of GPU kernel functions," in *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering - FSE '10*, 2010, p. 187.
[73] Lawrence Livermore National Laboratory, University of Utah, and RWTH Aachen University, "ARCHER," *GitHub*, 2018. [Online]. Available: https://github.com/PRUNERS/archer.
[74] O. Hernandez, C. Liao, and B. Chapman, "Dragon: A Static and Dynamic Tool for OpenMP," in *International Workshop on OpenMP Applications and Tools WOMPAT 2004*, 2005, pp. 53–66.
[75] 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.
[76] M. Zheng, V. T. Ravi, F. Qin, and G. Agrawal, "GRace: a low-overhead mechanism for detecting data races in GPU programs," *ACM SIGPLAN Not.*, vol. 46, no. 8, p. 135, Sep. 2011.
[77] Z. Dai, Z. Zhang, H. Wang, Y. Li, and W. Zhang, "Parallelized Race Detection Based on GPU Architecture," in *Advanced Computer Architecture. Communications in Computer and Information Science*, Springer, Berlin, Heidelberg, 2014, pp. 113–127.
[78] P. Li, G. Li, and G. Gopalakrishnan, "Practical Symbolic Race Checking of GPU Programs," in *SC14: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2014, pp. 179–190.
[79] 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.
[80] 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.
[81] 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.
[82] Allinea Software Ltd, "ALLINEA DDT," *ARM HPC Tools*, 2018. [Online]. Available: https://www.arm.com/products/development-tools/hpc-tools/cross-platform/forge/ddt.
[83] R. W. S. Inc., "TotalView for HPC," 2018. [Online]. Available: https://www.roguewave.com/products-services/totalview.
[84] A. P. Claudio, J. D. Cunha, and M. B. Carmo, "Monitoring and debugging message passing applications with MPVisualizer," in *Proceedings 8th Euromicro Workshop on Parallel and Distributed Processing*, 2000, pp. 376–382.
[85] J. Carretero *et al.*, "Energy-efficient Algorithms for Ultrascale Systems," *Supercomput. Front. Innov.*, vol. 2, no. 2, Apr. 2015.
[86] P. W. Coteus, J. U. Knickerbocker, C. H. Lam, and Y. A. Vlasov, "Technologies for exascale systems," *IBM J. Res. Dev.*, vol. 55, no. 5, p. 14:1-14:12, Sep. 2011.
[87] The U.S. Department of Energy's Oak Ridge National Laboratory, "SUMMIT," 2018. [Online]. Available: https://www.olcf.ornl.gov/olcf-resources/compute-systems/summit/.
[88] E. Strohmaier, J. Dongarra, H. Simon, and M. Meuer, "TOP500 List - June 2018," *TOP 500 Organization*, 2018. [Online]. Available: https://www.top500.org/lists/2018/06/.