applied sciences

MDPI

Article

An Architecture for a Tri-Programming Model-Based Parallel Hybrid Testing Tool

Saeed Musaad Altalhi [1,2,*], Fathy Elbouraey Eassa [1], Abdullah Saad Al-Malaise Al-Ghamdi [3], Sanaa Abdullah Sharaf [1], Ahmed Mohammed Alghamdi [4], Khalid Ali Almarhabi [5] and Maher Ali Khemakhem [1]

1 Department of Computer Science, Faculty of Computing and Information Technology, King Abdulaziz University, Jeddah 21589, Saudi Arabia; feassa@kau.edu.sa (F.E.E.); ssharaf@kau.edu.sa (S.A.S.); makhemakhem@kau.edu.sa (M.A.K.)
2 Department of Computer Science, Umm Al-Qura University, Makkah 21955, Saudi Arabia
3 Department of Information Systems, Faculty of Computing and Information Technology, King Abdulaziz University, Jeddah 21589, Saudi Arabia; aalmalaise@kau.edu.sa
4 Department of Software Engineering, College of Computer Science and Engineering, University of Jeddah, Jeddah 21493, Saudi Arabia; amalghamdi@uj.edu.sa
5 Department of Computer Science, College of Computing at Alqunfudah, Umm Al-Qura University, Makkah 21514, Saudi Arabia; kamarhabi@uqu.edu.sa
* Correspondence: ssaltalhi@stu.kau.edu.sa or smtalhi@uqu.edu.sa

Abstract: As the development of high-performance computing (HPC) is growing, exascale computing is on the horizon. Therefore, it is imperative to develop parallel systems, such as graphics processing units (GPUs) and programming models, that can effectively utilise the powerful processing resources of exascale computing. A tri-level programming model comprising message passing interface (MPI), compute unified device architecture (CUDA), and open multi-processing (OpenMP) models may significantly enhance the parallelism, performance, productivity, and programmability of the heterogeneous architecture. However, the use of multiple programming models often leads to unexpected errors and behaviours during run-time. It is also difficult to detect such errors in high-level parallel programming languages. Therefore, this present study proposes a parallel hybrid testing tool that employs both static and dynamic testing techniques to address this issue. The proposed tool was designed to identify the run-time errors of C++ and MPI + OpenMP + CUDA systems by analysing the source code during run-time, thereby optimising the testing process and ensuring comprehensive error detection. The proposed tool was able to identify and categorise the run-time errors of tri-level programming models. This highlights the need for a parallel testing tool that is specifically designed for tri-level MPI + OpenMP + CUDA programming models. As contemporary parallel testing tools cannot, at present, be used to test software applications produced using tri-level MPI + OpenMP + CUDA programming models, this present study proposes the architecture of a parallel testing tool to detect run-time errors in tri-level MPI + OpenMP + CUDA programming models.

Keywords: hybrid analysis; Hybrid MPI/OpenMP; dynamic analysis; race; static analysis; tri-level programming model run-time errors

Citation: Altalhi, S.M.; Eassa, F.E.; Al-Ghamdi, A.S.A.-M.; Sharaf, S.A.; Alghamdi, A.M.; Almarhabi, K.A.; Khemakhem, M.A. An Architecture for a Tri-Programming Model-Based Parallel Hybrid Testing Tool. *Appl. Sci.* **2023**, *13*, 11960. https://doi.org/10.3390/app132111960

Academic Editor: Juan A. Gómez-Pulido

Received: 6 September 2023
Revised: 24 October 2023
Accepted: 30 October 2023
Published: 1 November 2023

1. Introduction

Exascale computing is the next milestone in the domain of high-performance computing (HPC). At present, exascale computing enables systems to compute one EXA, or one quintillion, floating point operations per second (exaFLOPs). However, the objective is to perform more than a quintillion operations per second, which would enable engineers and scientific researchers to address extremely challenging problems, such as climate change, medicine discovery, nuclear fusion, and computerised simulations, to name a few.

An exascale system is unlike conventional HPC due to its abilities, topology, and energy efficiency. Nevertheless, such systems suffer from constraints such as programming frameworks, energy requirements, processor architectures, memory regulation, architectural resilience, and parallel programming. Therefore, these limitations must be addressed to achieve exascale systems.

Exascale systems are primarily plagued by two major challenges, namely energy requirements and its parallelism capabilities. Its energy requirement issues may be overcome using several proposed novel ideas [1–4], while the execution and improvement of its parallelism capabilities may be addressed by combining multiple programming models concurrently. Two or more models, which boast different features, may also be combined for this purpose. However, the architecture of the combined processors must be well understood to successfully and efficiently utilise their resources to build heterogeneous nodes.

Many programming models have been designed for a wide variety of tasks, such as a message passing interface (MPI) [5] for message passing and open multi-processing (OpenMP) [6] for shared memory parallelism. Programming models that contain accelerators, such as graphics processing units (GPUs) and central processing units (CPUs), are also popular options for heterogeneous systems. For instance, programming models such as open accelerators (OpenACC) [7], the Open Computing Language (OpenCL™) [8], and the compute unified device architecture (CUDA) [9]) are commonly used with GPUs.

Multiple studies have proven that it is possible to combine the features of multiple programming models to create hybrid models that can scale between connectivity and processing based on their memory efficiency. These hybrid models can be categorised either as

- Single-level, comprising a standalone model, such as OpenMP, MPI, or CUDA;
- Dual-level (X + Y), comprising two programming models to improve parallelism, such as MPI + OpenMP [10–12] or OpenMP + CUDA [13];
- Tri-level (MPI + X + Y) [14], comprising three distinct programming models to enhance parallelism.

However, the latter two programming models may produce new errors for a plethora of reasons, unlike each of its individual constituent models. Several studies have shown that each programming model boasts different features and serves different purposes. For instance, the purpose of OpenACC is to make the parallel programming process of heterogeneous CPU and GPU systems more straightforward [15], while OpenCL™ is designed to utilise the computing power of GPUs to improve the productivity of GPU applications. However, OpenCL™ may actually decrease productivity as it is complex and difficult to programme [16]. OpenMP, on the other hand, is commonly used in systems and architectures that have shared memory. It provides excellent parallelism at both coarse- and fine-grained levels, as well as seamlessly integrating with a variety of hardware, platforms, systems, and programming languages via its application programming interface (API), and, specifically, its compiler directive streamlines the process of creating multiprocessing shared-memory programmes. Meanwhile, other programming models, such as CUDA [9], allow code to access data from any memory address, which provides developers with some details, such as the kernel boot time mapping of threads, the memory transferred from the host to the device, parallelism, and the temporary storage of data. The compute unified device architecture (CUDA), which is also used in parallel computing, is commonly programmed using the Fortran and C/C++ programming languages. As such, it supports the use of GPUs and offers significant support in this regard.

Parallel models not only require more effort to test but also produce more parallel errors. Furthermore, when parallel models are combined into one application, the causes of these errors change during run-time as the models integrate. As present, there are no compilers capable of detecting these types of run-time errors. Therefore, tri-level programming models currently do not have testing tools that are capable of detecting their run-time errors. As such, the purpose of this present study was to develop a parallel hybrid testing tool using the C++ programming language and a combination of static and

dynamic testing methods to effectively test and detect errors both in real time and during run-time by analysing the source code of systems comprising MPI + OpenMP + CUDA programming models, as parallel hybrid techniques may increase the testing efficiency and detect more errors.

2. Background

This section provides a comprehensive overview of the main aspects of this present study and a more profound analysis of each aspect. The programming models used and the reasons for their selection will be described. Some of the run-time errors and testing techniques will also be explained and discussed.

2.1. Programming Models

2.1.1. Message Passing Interface (MPI)

The MPI standard [5] defines the interface for message passing libraries. Made publicly available in May of 1994, the MPI method exchanges messages between processes to enable parallel programming and facilitate the transportation of data. The purpose of MPI is to achieve portability, efficiency, and flexibility in message passing applications. It is noteworthy that MPI is a specification, not an implementation or programming language, and its operations are expressed using Fortran, C, or C++ functions, subroutines, or methods. The MPI offers numerous options, including Open MPI [17] and message passing interface chameleon (MPICH) [18], which are open-source and free, as well as commercial options, such as IBM® Spectrum MPI [19] and Intel® MPI [20]. As the MPI provides exceptional features [19,20], it is highly advantageous for the following.

- Standardisation, as it is an exclusive message passing library that can, satisfactorily, be considered a standard. It works on almost any high-performance computing system and, in most cases, can replace any existing message passing library.
- Portability, as it requires minimal or no source code to port an application to a supported platform.
- Vendors, as they can to use its native capabilities to examine performance opportunities and achieve optimal performance, wherever possible. Optimal algorithms may also be developed for any implementation.
- Functionality, as it has over 430 functions, with MPI-2 and MPI-1 included in MPI-3. However, a dozen or so routines are required to create a typical MPI application.
- Availability, as it has a wide selection of vendor and open-source options available.

2.1.2. Open Multi-Processing (OpenMP)

The OpenMP model [21] is a programme-shared memory that contains an API comprising compiler directives, variables, and a library of run-time routines. This structure has set a standard that is suitable for several shared-memory platforms, architectures, hardware, programming languages, and software vendors. It also facilitates the implementation of coarse- and fine-grained parallelisms. The OpenMP model can also be executed on many open-source or commercial compilers, such as GNU's Not Unix! (GNU) Compiler Collection, Intel® (Santa Clara, CA, USA), and Oracle® (Austin, TX, USA) Developer Studio compilers [22].

2.1.3. Compute Unified Device Architecture (CUDA)

The CUDA [23] programming model and parallel platform commonly uses GPUs to increase computing performance. As CUDA 2006 was first introduced by the NVIDIA® Corporation (Santa Clara, CA, USA), it was only compatible with NVIDIA® GPUs for a long time, thereby rendering it unportable. However, Fortran has since developed a CUDA programming model using the C/C++ programming languages for GPUs. Nevertheless, CUDA's latest 12.0 version, which was released on 8 December 2022, only enables NVIDIA® GPUs to take full advantage of its extensive parallelism. When programming in CUDA, the low-level nature of the language facilitates reading from any memory address. This

provides programmers with insights into the process of transferring memory from the host to the device, the storage of temporary data, thread mapping during kernel start-up, and the implementation of parallelism.

2.2. Tri-Level Programming Model (OPENMP + CUDA + MPI)

Combining several programming frameworks can enhance the abilities and efficacy of heterogeneous platforms. Nevertheless, such models necessitate extensive and robust programming to benefit from extensively parallel supercomputing systems. Integrating OpenMP, MPI, and CUDA may improve the performance and dependability of a model. Such frameworks can then be used to estimate the effects of climate change, locate oil wells, perform real-time imaging, conduct protein folding research, and train deep-learning systems. It can also be applied in numerous other domains, such as machine learning, Big Data systems, cloud computing, cybersecurity, simulation, software development, computer networking, and graphics. Therefore, integrating OpenMP, CUDA, and MPI can offer different advantages, such as better performance and parallelism and lower programming requirements, and benefit from GPU acceleration.

This present study discusses a tri-level programming model (MPI + OpenMP + CUDA) as the benefits of combining these three frameworks are numerous and will not only enhance parallelism and performance but decrease programming effort and capitalise on GPU acceleration.

A tri-level programming architecture comprising MPI, OpenMP, and CUDA would facilitate parallelism for different hardware systems. It would also facilitate parallelism at numerous stages that comprise intra-node (OpenMP), inter-node (MPI), and graphics processing units (CUDA). Its benefits would include scalability, robust performance due to MPI, error management using OpenMP, portability, atomics, tasking extensions, and accelerated computing. Moreover, CUDA offers robust programming, superior capabilities, optimisation, and thread synchronisation. Therefore, combining these frameworks to create a tri-level programming architecture would yield better performance and energy requirements than similar architectures. More specifically, as it is based on several parallelism levels, the software can regulate and scale resources based on the hardware's dynamics to yield better efficiency, performance, and robustness.

Nevertheless, this combination may produce run-time errors that occur due to unique reasons and that behave differently. For instance, a tri-level programming model could become over-synchronised, move redundant data, and be inefficient and complex.

2.3. An Overview of Common Run-Time Errors

2.3.1. Deadlocks

A deadlock occurs when multiple processes are trapped in a state of waiting for each other to release resources or complete execution, during which neither process progresses at all. Deadlocks are a serious issue in parallel systems as they can bring the entire system to a halt. Although deadlocks are easy to detect once they have occurred, in some cases, they can be difficult to detect early on due to specific interleaving [24]. Furthermore, the execution environment and sequence of any system may or may not create actual or potential deadlocks.

For example, if an MPI process is dependent on another process for data, a deadlock may occur if the sender is held up in a parallel OpenMP area. Deadlocks can also occur due to workload misbalance or the failure of every thread to approach a barrier. In such cases, the wait time for an MPI process may be infinite.

2.3.2. Livelocks

Livelocks are similar to deadlocks; however, in a livelock, the state of the affected tasks continues to change but never makes any real progress. In some cases, all the tasks stall and remain unfinished [25,26]. Nevertheless, in a livelock, a thread will never be caught in a permanent blockage. As livelocks are plagued by busy–wait cycles, they are not only

power-consuming but may perform poorly. Furthermore, both deadlocks and livelocks can cause parallel systems to become unresponsive and may require manual intervention to resolve.

An example of a livelock would be when two computing processes, A and B, use an MPI to transfer messages; A relays the message to B and then awaits a response, but B relays its own different message to A, rather than sending an instantaneous response for the first message. Therefore, both A and B end up stuck in an endless period of waiting for a response from the other, creating a livelock.

2.3.3. Race Conditions

Race conditions occur when multiple tasks try to access and modify a shared resource simultaneously. This can result in unexpected behaviours or errors [27]. An example of a race condition is when OpenMP threads change data while an MPI is moving data, thereby creating inconsistencies at the receiver's end.

2.3.4. Data Races

A data race is a critical race condition that occurs when several tasks attempt to simultaneously access and modify the same memory location [28]. This can cause the data to become corrupt or inconsistent, leading to errors or incorrect results.

An example of a data race is when OpenMP uses at least two threads to simultaneously alter one variable located in a parallel area sans synchronisation, leading to unanticipated outcomes. Another example is when several threads that are not synchronised or possess operational atomicity write to one memory location in a CUDA system.

2.3.5. Mismatches

Mismatching is another common problem that can occur under many circumstances, such as when the wrong type or number of arguments is made, when more than one call is made simultaneously, when collective calls are made, or when tasks are assigned to the wrong processors or are not properly synchronised, thereby causing delays, decreasing performance, and yielding incorrect outputs.

There are multiple mismatch issues in MPI that warrant understanding to fully comprehend. For instance, a type mismatch occurs when the type of data sent by the sender does not match the type of data that the receiver anticipates. Meanwhile, a tag mismatch occurs when the tag that the sender tags the data with does not match the tag that the receiver anticipates.

2.4. Testing Techniques

Various techniques, such as static and dynamic testing, to name a few, can be used to develop testing software. Static testing identifies static errors pre-compilation by analysing the source code. This facilitates the analysis of the code in detail and fully covering the application source code without launching the application itself. Unfortunately, due to the nature of parallel applications, it is more difficult to analyse statically as programmes behave unpredictably when they are actually running [29,30]. Nevertheless, static analysis facilitates the identification of run-time defects within the source code, as well as both potential and existing issues, such as race conditions and deadlocks. Dynamic testing involves analysing a system while it is running to detect errors that may arise during run-time. However, such programmes can be affected by the execution environment, which can cause the application to slow down. Dynamic analysis facilitates the flexible monitoring and detection of each thread in parallel applications. However, it is difficult to test parallel code in its entirety as it is not possible to verify whether the errors have actually been corrected after they were corrected the first time around.

The type and behaviour of an error determines how and which techniques to use as both static and dynamic techniques cannot detect all errors. As such, this present study

proposes a hybrid technique that operates parallelly to identify both run-time errors and examine threads and was tested on a variety of situations and applications.

3. Literature Review

Multiple studies have tested HPC and parallel software for various purposes and scopes. Testing techniques, namely static, dynamic, and hybrids, are used to detect various types of errors. Static testing involves examining the source code prior to system execution [31], while dynamic testing examines a source code while the system is running [24]. Although dynamic techniques have benefits, they also have significant drawbacks, such as the need for multiple test cases and the possibility of undetected errors when the system starts up. Hybrid techniques combine both static and dynamic testing [31,32].

Many tools, such as Intel® Trace Analyzer and Collector [33] and MPI-Checker [34], underutilise static testing to detect run-time errors in MPI. Dynamic testing, however, is widely used by other tools, such as MEMCHECKER [35], Umpire [36], the Marmot Umpire Scalable Tool (MUST) [37–39], MAD [40], MOPPER [41], and run-time error detection (RTED) [42], to detect run-time errors in MPI. Meanwhile, other tools test different run-time errors. These tools use hybrid techniques [43] to detect deadlocks. The ACC_TEST [44] was specifically designed to swiftly identify and precisely locate any run-time errors, including, but not limited to, deadlocks, race conditions, or mismatches.

Various OpenMP testing tools use multiple techniques to detect run-time errors. For instance, GPUVerify [45], OmpVerify [46], and DRACO [47] use static techniques, while Helgrind [48], RTED [42], Valgrind [49], ROMP [50], Intel® Thread Checker [51,52], and Sun Studio® Thread Analyzer [51,53] use dynamic techniques. Meanwhile, AddressSanitizer (ASan) and ThreadSanitizer (TSan) [54,55] can be used to detect run-time errors in systems with OpenMP directives. ARCHER [56], on the other hand, uses a hybrid technique to detect data races in large OpenMP applications. Dynamic techniques are also used to test dual systems that combine MPI + OpenMP, such as Marmot [57,58].

GPUVerify [45], and PolyOMP [59] use static techniques to detect data race errors in systems that contain CUDA. GUARD [60], RaceTM [61], and KUDA [62] are examples of dynamic testing tools, while GMRace [63], GRace [64], Grace [65,66], and SESA [67] are examples of hybrid testing tools that test run-time errors in systems that contain CUDA. The literature review reveals a significant gap in the market in terms of testing tools for tri-level programming models. As such, this present study creates a highly specialised testing tool that incorporates MPI, OpenMP, and CUDA specifically for this unique programming model. Although debugging tools are a vital part of testing, their testing techniques often require clarification. Some debugging tools are commercial, while others, such as such as AutomaDeD [68], ALLINEA DDT [69,70], TotalView [71,72], MPVisualizer [73], Intel®Inspector [74], PDT [75], and Arm DDT [76,77], are not. However, as these non-commercial debugging tools pinpoint the causes of errors instead of testing or detecting errors, they cannot be categorised according to their testing techniques.

The literature review also reveals that there is no testing tool with which to detect run-time errors in MPI + OpenMP + CUDA tri-level programming models. As such, this present study develops a hybrid technique to identify many actual and potential errors in the C++ source code beforehand.

Although multiple extant studies have developed and proposed software testing tools for parallel applications, the detection of static and dynamic errors in tri-programming models warrants further examination. Furthermore, in the case of heterogeneous systems, tri-programming models still require significant improvement. Therefore, it is imperative to address the lack of testing tools with which to detect run-time errors in applications that have been developed using MPI + OpenMP + CUDA tri-level programming models, and it must be resolved immediately to ensure the efficiency and effectiveness of these applications. Table 1 lists the techniques, targeted programming models, errors targeted, and limitations of the tools that previous studies have developed, while Table 2 compares them to the architecture of the proposed tool, which primarily targets programming models

and more errors and categorises them according to the types of errors that they address, i.e., real, potential, or both.

Table 1. A list of comparable extant studies.

References	Technique	Error(s) Detected	Programme Model	Limitation(s)
[34]	Static	Mismatches	MPI	Works with single-level programming models. Only identifies mismatches in MPI.
[45]	Static	Data Races	CUDA, and OpenCL™	Works with single-level programming models.
[59]	Static	Data Races	OpenMP	Works with single-level programming models.
[36]	Dynamic	Deadlocks Mismatched Collective Operations Resource Exhaustion	MPI	Works with single-level programming models.
[37–39]	Dynamic	Deadlocks Data Races Mismatches	MPI	Works with single-level programming models and MPI only.
[66]	Dynamic	Deadlocks, Data Races	CUDA	Works with single-level programming models.
[43]	Hybrid	Deadlocks	MPI	Works with single-level programming models. Only identifies deadlocks.
[44]	Hybrid	Deadlocks Mismatches Livelocks Data Races/Race Conditions	MPI	Work with single-level programming models.
[57]	Hybrid	Data Races	OpenMP	Work with single-level programming models.

Table 2. A comparison of the proposed tool and extant tools.

Programming Model	Tool Name	Run-Time Errors	Error Type
CUDA only	GPUVerify [45]	Data Races	Real
MPI only	MPI-Checker [34]	Mismatches	Potential
MPI only	MUST [37–39]	Deadlocks, Data Races, and Mismatches	Real
OpenMP only	• Intel® Thread Checker [51,52] • Sun Studio® Thread Analyzer [51,53]	Deadlocks and Data Races	Real
CUDA only	GUARD [60]	Data Races	Real
Dual: MPI + OpenMP	Marmot [57,58]	Deadlocks, Race Conditions, and Mismatches	Real
Tri: MPI + OpenMP + CUDA	Proposed Tool	Deadlocks, Race Conditions, Data Races, Mismatches, and Livelocks	Both

The literature review provides a detailed analysis of the different testing tools currently available (Tables 1 and 2). It is noteworthy that, at present, a parallel testing tool that can detect run-time errors in applications created using a tri-level MPI + OpenMP + CUDA programming model does not exist.

4. Architecture of the Proposed Tri-Level Programming Model

Figures 1 and 2 illustrate the construction of a hybrid testing tool for MPI + OpenMP + CUDA and C++. The design comprises two parts. The first part identifies any run-time issues and notifies the developer during the static phase, while the second part uses

assertion statements to automatically detect errors in a dynamic manner. Therefore, the proposed hybrid architecture combines static and dynamic testing.

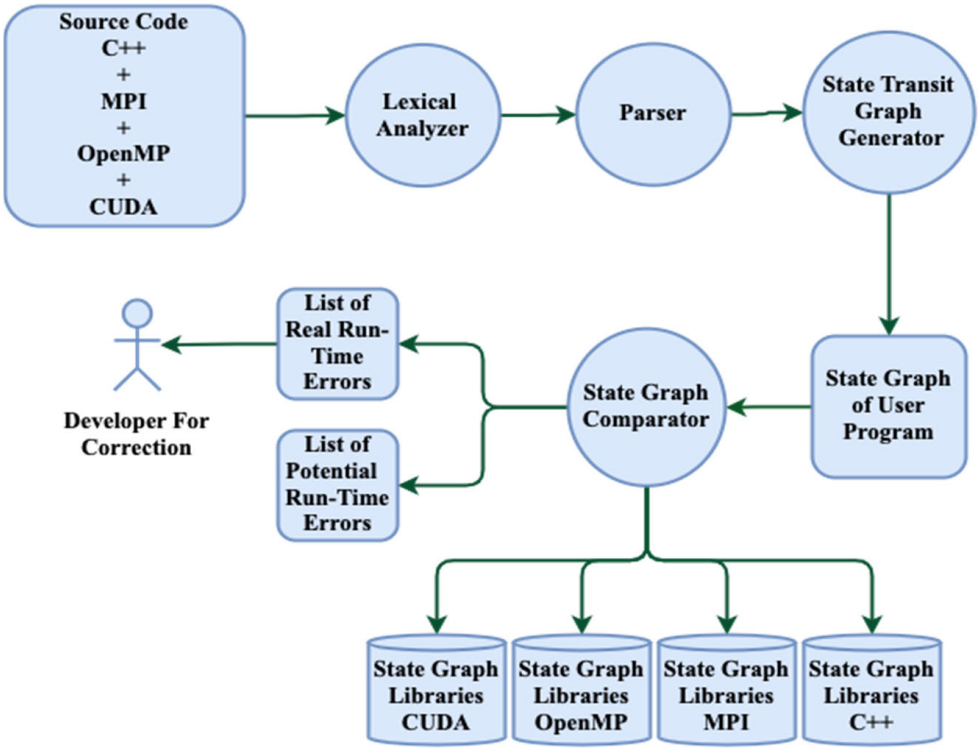

Figure 1. Static portion of the proposed architecture.

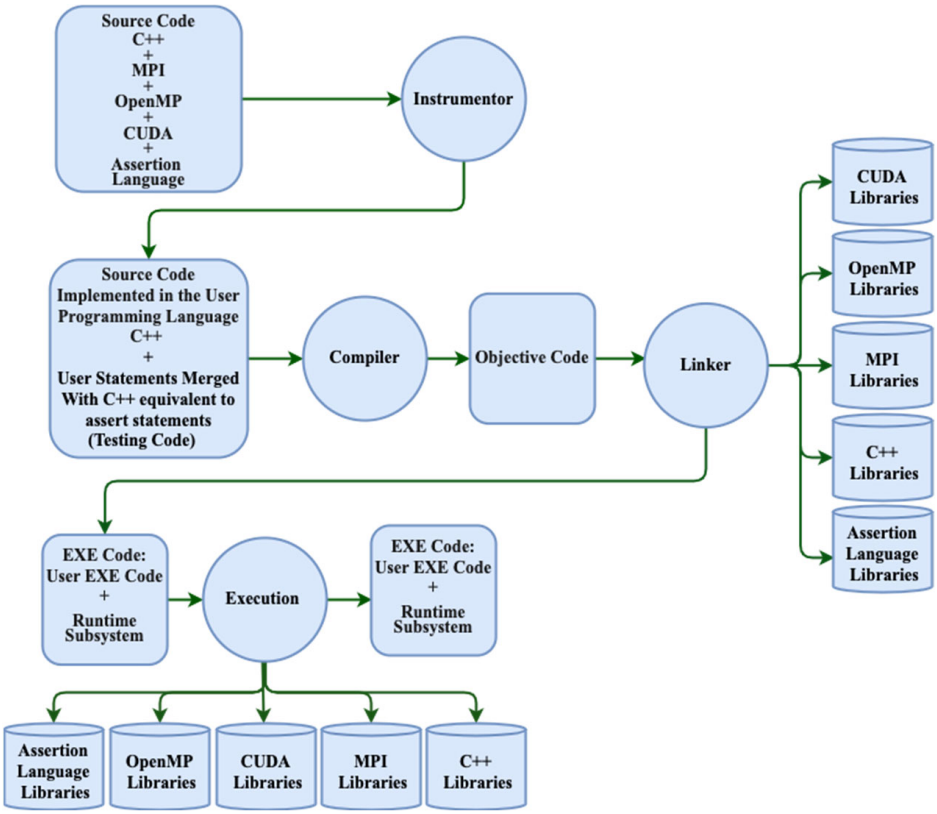

Figure 2. Dynamic portion of the proposed architecture.

After the first static testing phase, the developer is provided a list of actual and potential run-time errors (Figure 1), which can then be input into an assertion process to automatically detect and avoid these potential errors during the second dynamic testing phase. Furthermore, if any errors occur while the programme is operating, the developers are sent an immediate notification to efficiently resolve the issues. Apart from this, identifying these run-time errors and fixing them prior to the second dynamic testing phase decreases the testing time and improves performance. The first static testing phase of the proposed architecture includes the following.

- A lexical analyser that reads the source code, which includes C++, MPI, OpenMP, and CUDA, line by line and then understands the source code before generating a token table containing at least two columns: the token name and token type.
- A parser or syntax analyser that checks the syntax of each statement and detects syntax errors. More specifically, it analyses the syntactical structure of the inputs and determines whether they are in the correct syntax for the programming language used.
- A generator that produces a dependable state transit graph for a code that comprises MPI, C++, OpenMP, and CUDA. An appropriate data structure is employed to build a suitable state graph.
- A state graph comparator that compares the user programme graph with the state graphs of all the programming languages and models. The grammar of each programming language is included in the state graph library, which is available via this comparator. As the static architecture may identify both potential and actual run-time errors, the outcomes of the comparisons are presented in a list. The actual run-time faults are sent to the developer to correct them as these errors will definitely occur if they are not corrected. In the second dynamic testing phase of the proposed architecture, assertions are injected into the sources and then instrumented to examine potential run-time errors.

As seen in Figure 2, in the second dynamic testing phase, the input for the instrumentation process is a combination of the source code and assertion statements. The output of the instrumentor consists of user codes and testing codes written in the C++ programming language. However, the following must be built to produce a dynamic testing tool that detects run-time errors.

1. Assertion language: the testing language that helps to detect and monitor the variables and behaviour of a system during run-time. It is combined with the user code to create a new code as part of the dynamic test.
2. Instrumentor: this is responsible for converting the assertion statement into its equivalent C++ code.
3. Run-time analyser subsystem: this includes a detecting and debugging module.

As seen in Figures 1 and 2, the proposed architecture is a hybrid testing tool that consists of dynamic and static testing to detect both actual and potential run-time errors. The proposed architecture was implemented in C++ to test applications that were built in a tri-level OpenMP + CUDA + MPI programming model. The first static testing phase will detect and display errors from the beginning at the compilation stage, as well as detecting common parallel errors during run-time as part of its second dynamic testing phase by adding assertion statements to the source code. The primary purpose of using an instrumentor is to convert the assertion statements to their C++ code equivalents and analyse the system during run-time. The instrumented codes become EXE codes upon compilation and linking, and they include user executables and run-time subsystems. Lastly, a list of run-time errors is displayed after the EXE codes are executed.

The proposed architecture is implemented in the first static testing phase using Algorithm 1 and in the second dynamic testing phase using Algorithm 2, as seen below.

Algorithm 1 Static Testing Phase

```
1:  Inputting the source code
source_code = input ("Enter the source code containing the (C++) + tri-level
programming (MPI + OpenMP + CUDA):")
2:  Lexical analysis
tokens = perform_lexical_analysis(source_code)
3:  Parsing
parsed_output = perform_parsing(tokens)
4:  Generating the state transition graph for the user code
user_stg = generate_state_transition_graph(parsed_output)
5:  Generating the state transition graphs for the MPI, OpenMP, CUDA, and C++
libraries
mpi_stg = generate_mpi_state_transition_graph ()
openmp_stg = generate_openmp_state_transition_graph ()
cuda_stg = generate_cuda_state_transition_graph ()
cpp_stg = generate_cpp_state_transition_graph ()
6:  Comparing the state transition graphs
Actual_errors = []
if not compare_state_transition_graphs (user_stg, mpi_stg):
Actual _errors. append ("MPI run-time error")
if not compare_state_transition_graphs (user_stg, openmp_stg):
Actual _errors. append ("OpenMP run-time error")
if not compare_state_transition_graphs (user_stg, cuda_stg):
Actual _errors. append ("CUDA run-time error")
if not compare_state_transition_graphs (user_stg, cpp_stg):
Actual _errors. append ("C++ run-time error")
7:  Listing the actual run-time errors
if (Actual _errors) > 0:
print ("Actual run-time errors found: ")
if else:
print ("No Actual run-time errors found.")
else:
print ("Potential run-time errors found: ")
```

A hybrid testing programme, which has been developed precisely for parallel programming systems that are based on a tri-level programming architecture comprising MPI, OpenMP, and CUDA within C++, would be a detailed evaluation system that integrates static (pre-execution) and dynamic (during execution) testing to provide comprehensive information. It would be able to pinpoint and process the integrated OpenMP, MPI, and CUDA interactions, thereby elucidating the synergy of the framework. During the static testing phase, the programme completes a comprehensive evaluation of the code to ascertain specific concerns or pitfalls that might be triggered during run-time, specifically related to using several models. Meanwhile, in the dynamic testing phase, the programme evaluates real-time programme execution as well as recording critical data concerning threads, memory use, communications, and other aspects.

A hybrid tool is a comprehensive evaluation approach that integrates the benefits of dynamic and static testing. As such, it offers detailed verification and application-based evaluations that are specific to the sophisticated tri-level programming framework, thereby improving dependability and performance.

Building a testing tool that was suitable for a sophisticated hybrid architecture comprising MPI + OpenMP + CUDA was challenging. The combination of the dynamic and static evaluation techniques was thoroughly planned to yield adequate performance. Systematic proposals were also developed to better understand how the two approaches might converge. Nevertheless, despite our best efforts, combining CUDA, OpenMP, and MPI still created unforeseen issues. These issues were adequately addressed by simulating different scenarios and building issue categories. Apart from this, the language dynamics

and complexity increased the challenges in terms of the compatibility of the different C++ codebases. It is presumed that continuous integration, beta tests, and real-time feedback systems may be used to address these challenges. Meanwhile, errors were rectified in a timely manner using continuous monitoring and real-time analytics. It was also critical to keep the hybrid tool updated due to the dynamically evolving nature of OpenMP, MPI, and CUDA. A modular approach was used to update the individual modules without affecting others. These updates increased the adaptability and significance of the hybrid tool.

This present study was able to use pioneering approaches and new technologies, as well as addressing the issues that it faced, to successfully build a flexible and productive hybrid testing tool.

Algorithm 2 Dynamic Testing Phase

```
1:  Reading the source code containing the (C++) + tri-level programming MPI +
OpenMP + CUDA and the assert statement as inputted, the potential run-time errors.
2:  Applying instrumentation to the source code, the potential run-time errors, to insert
run-time checks for the run-time errors.
if (MPI is used in the code)
        insert MPI run-time checks
  if (OpenMP is used in the code)
        insert OpenMP run-time checks
  if (CUDA is used in the code)
        insert CUDA run-time checks
  if (assert statements are used in the code)
        insert assert statement checks
  Saving the instrumented source code.
3:  Compiling the instrumented source code and linking it with the relevant libraries.
  if (MPI is used in the code)
        link with MPI library
  if (OpenMP is used in the code)
        link with OpenMP library
  if (CUDA is used in the code)
        link with CUDA library
  Compiling and linking the code.
4:  Executing the instrumented executable code on the target system.
  if (MPI is used in the code)
        execute with MPI run-time
  if (OpenMP is used in the code)
        execute with OpenMP run-time
  if (CUDA is used in the code)
        execute with CUDA run-time
5:  Monitoring the execution of the instrumented code for any run-time errors.
  while (the code is executing)
        if (a run-time error occurs)
            log the error)
6:  Sending a list of the run-time errors to the developer for further analysis and
correction.
7:  Displaying a list of run-time errors.
```

5. Discussion

Multiple studies have examined the use of MPI, CUDA, and OpenMP in parallel systems, as well as developing many tools with which to identify run-time errors. Different programming models can be integrated to obtain features that facilitate the creation of highly parallel systems with heterogeneous architectures that function in exascale systems. OpenMP is the standardised solution for shared-memory platforms and architectures. It can implement coarse- and fine-grained parallelisms in various hardware, systems, platforms, and programming languages. Meanwhile, MPI supports all HPC platforms and implements

them entirely with multiple programming models and diverse networks. Compute unified device architecture (CUDA) programming models represent an extensively used type of programming model and parallel platform that utilises GPUs. Furthermore, as CUDA enables codes to access memory addresses at a low level, it provides developers with some details, such as the kernel boot time mapping of threads, the memory transferred from the host to the device, parallelisms, and the temporary storage of data. Nevertheless, no testing techniques or tools have examined the detection of run-time errors in tri-level programming models, nor have their potential errors been identified or categorised, which presents a challenge in developing such techniques or tools. The present study entailed an in-depth analysis of tri-level MPI + OpenMP + CUDA programming models, performing a series of experiments and simulating various scenarios to gain insights into the diverse run-time error patterns of these models.

The proposed hybrid technique comprises dynamic and static testing. C++ was used to develop this technique for programming models developed using a tri-level OpenMP + CUDA + MPI architecture. It evaluates the source code to identify static errors prior to compilation. Coders should pinpoint likely run-time issues related to the code. It is critical to rectify these issues promptly while executing the code to deliver adequate performance and its intended functionality. Therefore, developers should proactively pinpoint and address run-time issues to ensure that the applications work correctly and sans errors. As run-time errors can arise post-compilation, during execution, or at run-time, static tests were conducted to evaluate the code pre-compilation to identify potential sources of errors. Hence, it is critical to inform coders about such errors so that undetected issues can be addressed at run-time.

As parallel programmes are complex and several scenarios can lead to errors, a second dynamic testing phase was also included to identify errors by instrumenting and assessing the environment at run-time. However, including every feasible test case and data combination can overload a testing tool and raise its workload. The execution time of such dynamic approaches may also be impacted by the underlying environment. Furthermore, as static and other similar tests may not pinpoint particular run-time errors, a suitable approach should be selected only after categorising the run-time issues that have manifested.

The parallel hybrid evaluation approach is a detailed method that leverages sophisticated C++ programmes that are based on the OpenMP, MPI, and CUDA frameworks to pinpoint, diagnose, and rectify run-time issues. More specifically, it integrates aspects of dynamic and static evaluation to offer real-time feedback based on the data from the two testing phases. This provides developers with information that they can use to identify the causes of errors with greater precision compared to using only one approach.

The proposed tool was devised to process and address the complexities associated with heterogeneous systems. As it comprises a wide variety of hardware, specifically CPUs for OpenMP and MPI and GPUs for CUDA, it facilitates precise evaluation and code assessment for several computing elements.

In summary, the proposed parallel hybrid testing tool is an effective instrument with which to identify and address run-time issues that may arise in complex C++ programmes that are based on the OpenMP, MPI, and CUDA frameworks. It accomplishes this by using the pre-run-time code assessment and real-time information of a programme's behaviour when executed to properly identify, analyse, and fix issues.

6. Environment Required to Implement the Proposed Architecture

Two devices will be used to conduct the experiments. The first is the Aziz supercomputer Aziz Supercomputer is located at King Abdulaziz University in Jeddah, Saudi Arabia. It was launched in June 2015 and is operated by the High-Performance Computing Center, which comprises 496 computing nodes and approximately 12,000 Intel® CPU cores. It also has two nodes with NVIDIA Tesla K20® GPUs and two more nodes with Intel® Xeon-Phi accelerators. The second device is a laptop equipped with an 11th Generation Intel® Core i7-10750H @ 2.660 GHz × 12 CPU, 16 GB of random-access memory (RAM),

and an NVIDIA® GeForce GTX 1650 GPU. Both devices will run Ubuntu 20.04.4 LTS for development purposes.

7. Conclusions and Recommendations for Future Studies

Exascale supercomputers have become more feasible as the need for powerful computing has increased. Therefore, it has become crucial to construct significant parallel supercomputing systems that possess diverse architectures. Although hybrid programming models offer many benefits when used to construct parallel systems, they also make the resulting code more complex as they merge multiple parallel models into a single application. As this makes it more difficult to test, new methods are needed to detect run-time errors in these sophisticated programmes.

As such, this present study proposed a hybrid tool that can identify run-time errors in C++ and MPI + OpenMP + CUDA systems in parallel to address this issue. Static and dynamic testing methods were combined to develop a hybrid parallel system testing tool and yield a thorough testing process with reliable results. This combination also enhanced the system execution time, detected dynamic errors from the source code, and improved the performance of the application's system. Apart from this, the proposed tool is also flexible and can be integrated with systems that have been built using various programming models at different levels.

Although software testing and error detection in parallel programming models are important, most studies have focused on other aspects, such as power consumption, the processor architecture, system resiliency, and memory management, in the exascale environment. Therefore, the objective of this present study was to use code execution and other relevant assessment methods to ascertain the efficacy of the proposed method and the suggested hybrid framework, as well as to compare it to other non-hybrid approaches. The outcomes of these comparisons and tests will be reported in an upcoming paper that will also include specific information and potential contributions. The proposed method will also be used in a practical scenario, as well as requiring guidelines, time, and government approval to use the necessary hardware, such as the Aziz supercomputer.

Author Contributions: Conceptualization, S.M.A., F.E.E., A.S.A.-M.A.-G., S.A.S., A.M.A. and K.A.A.; Methodology, S.M.A., F.E.E., S.A.S. and M.A.K.; Software, S.M.A. and A.M.A.; Writing—original draft, S.M.A. and K.A.A.; Writing—review & editing, S.M.A., F.E.E., A.S.A.-M.A.-G., S.A.S. and M.A.K.; Supervision, F.E.E., A.S.A.-M.A.-G. and S.A.S.; Funding acquisition, F.E.E. and A.S.A.-M.A.-G. All authors have read and agreed to the published version of the manuscript.

Funding: This project was funded by the Deanship of Scientific Research (DSR) at King Abdulaziz University, Jeddah, under grant no. (KEP-PHD-20-611-42). The authors, therefore, acknowledge with thanks DSR for technical and financial support.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

Acknowledgments: The authors would like to thank the editor and the anonymous reviewers, whose insightful comments and constructive suggestions helped us to significantly improve the quality of this paper.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Ahmadpour, S.-S.; Heidari, A.; Navimipour, N.; Asadi, M.-A.; Yalcin, S. An Efficient Design of Multiplier for Using in Nano-Scale IoT Systems Using Atomic Silicon. *IEEE Internet Things J.* **2023**, *10*, 14908–14909. [CrossRef]
2. Ahmadpour, S.-S.; Navimipour, N.J.; Mosleh, M.; Bahar, A.N.; Yalcin, S. A Nano-Scale n-Bit Ripple Carry Adder Using an Optimized XOR Gate and Quantum-Dots Technology with Diminished Cells and Power Dissipation. *Nano Commun. Netw.* **2023**, *36*, 100442. [CrossRef]

3. Pramanik, A.K.; Mahalat, M.H.; Pal, J.; Ahmadpour, S.-S.; Sen, B. Cost-Effective Synthesis of QCA Logic Circuit Using Genetic Algorithm. *J. Supercomput.* **2023**, *79*, 3850–3877. [CrossRef]
4. Ahmadpour, S.-S.; Jafari Navimipour, N.; Bahar, A.N.; Mosleh, M.; Yalcin, S. An Energy-Aware Nanoscale Design of Reversible Atomic Silicon Based on Miller Algorithm. *IEEE Des. Test* **2023**, *40*, 62–69. [CrossRef]
5. MPI Forum MPI Documents. Available online: https://www.mpi-forum.org/docs/ (accessed on 6 February 2023).
6. OpenMP ARB About Us—OpenMP. Available online: https://www.openmp.org/about/about-us/ (accessed on 6 February 2023).
7. About OpenACC | OpenACC. Available online: https://www.openacc.org/about (accessed on 6 February 2023).
8. The Khronos Group Inc OpenCL Overview—The Khronos Group Inc. Available online: https://www.khronos.org/opencl/ (accessed on 6 February 2023).
9. NVIDIA about CUDA | NVIDIA Developer 2021. Available online: https://developer.nvidia.com/about-cuda (accessed on 5 September 2023).
10. Thiffault, C.; Voss, M.; Healey, S.T.; Kim, S.W. Dynamic Instrumentation of Large-Scale MPI and OpenMP Applications. In Proceedings of the International Parallel and Distributed Processing Symposium, Nice, France, 22–26 April 2003. [CrossRef]
11. Vargas-Perez, S.; Saeed, F. A Hybrid MPI-OpenMP Strategy to Speedup the Compression of Big Next-Generation Sequencing Datasets. *IEEE Trans. Parallel Distrib. Syst.* **2017**, *28*, 2760–2769. [CrossRef]
12. Wu, X.; Taylor, V. Performance Characteristics of Hybrid MPI/OpenMP Scientific Applications on a Large-Scale Multithreaded BlueGene/Q Supercomputer. In Proceedings of the 14th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing, Honolulu, HI, USA, 1–3 July 2013; Volume 5, pp. 303–309. [CrossRef]
13. Guan, J.; Yan, S.; Jin, J.M. An OpenMP-CUDA Implementation of Multilevel Fast Multipole Algorithm for Electromagnetic Simulation on Multi-GPU Computing Systems. *IEEE Trans. Antennas Propag.* **2013**, *61*, 3607–3616. [CrossRef]
14. Jacobsen, D.A.; Senocak, I. Multi-Level Parallelism for Incompressible Flow Computations on GPU Clusters. *Parallel Comput.* **2013**, *39*, 1–20. [CrossRef]
15. Agueny, H. Porting OpenACC to OpenMP on Heterogeneous Systems. *arXiv* **2022**, arXiv:2201.11811.
16. Herdman, J.A.; Gaudin, W.P.; Perks, O.; Beckingsale, D.A.; Mallinson, A.C.; Jarvis, S.A. Achieving Portability and Performance through OpenACC. In Proceedings of the 2014 First Workshop on Accelerator Programming Using Directives, New Orleans, LA, USA, 17 November 2015; pp. 19–26. [CrossRef]
17. OpenMPI Open MPI: Open Source High Performance Computing. Available online: https://www.open-mpi.org/ (accessed on 6 February 2023).
18. MPICH Overview | MPICH. Available online: https://www.mpich.org/about/overview/ (accessed on 6 February 2023).
19. IBM Spectrum MPI—Overview | IBM. Available online: https://www.ibm.com/products/spectrum-mpi (accessed on 6 February 2023).
20. Introducing Intel®MPI Library. Available online: https://www.intel.com/content/www/us/en/develop/documentation/mpi-developer-reference-linux/top/introduction/introducing-intel-mpi-library.html (accessed on 6 February 2023).
21. Barney, B. OpenMP | LLNL HPC Tutorials. Available online: https://hpc-tutorials.llnl.gov/openmp/ (accessed on 6 February 2023).
22. Oracle Developer Studio 12.5: OpenMP API®. Available online: https://www.oracle.com/application-development/technologies/developerstudio-documentation.html (accessed on 6 February 2023).
23. Harakal, M. Compute Unified Device Architecture (CUDA) GPU Programming Model and Possible Integration to the Parallel Environment. *Sci. Mil. J.* **2008**, *3*, 64–68.
24. Cai, Y.; Lu, Q. Dynamic Testing for Deadlocks via Constraints. *IEEE Trans. Softw. Eng.* **2016**, *42*, 825–842. [CrossRef]
25. Ganai, M.K. Dynamic Livelock Analysis of Multi-Threaded Programs. In *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*; Springer: Berlin/Heidelberg, Germany, 2013; Volume 7687, pp. 3–18. ISBN 9783642356315.
26. Lin, Y.; Kulkarni, S.S. Automatic Repair for Multi-Threaded Programs with Deadlock/Livelock Using Maximum Satisfiability. In Proceedings of the International Symposium on Software Testing and Analysis, San Jose, CA, USA, 21–25 July 2014; pp. 237–247.
27. Münchhalfen, J.F.; Hilbrich, T.; Protze, J.; Terboven, C.; Müller, M.S. Classification of Common Errors in OpenMP Applications. In *Proceedings of the Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), Salvador, Brazil, 28–30 September 2014*; DeRose, L., de Supinski, B.R., Olivier, S.L., Chapman, B.M., Müller, M.S., Eds.; Springer International Publishing: Cham, Switzerland, 2014; Volume 8766, pp. 58–72.
28. Cao, M. Efficient, Practical Dynamic Program Analyses for Concurrency Correctness. Ph.D. Thesis, The Ohio State University, Columbus, OH, USA, 2017.
29. Huchant, P. Static Analysis and Dynamic Adaptation of Parallelism. Ph.D. Thesis, Université de Bordeaux, Bordeaux, France, 2019.
30. Sawant, A.A.; Bari, P.H.; Chawan, P. Software Testing Techniques and Strategies. *J. Eng. Res. Appl.* **2012**, *2*, 980–986.
31. Saillard, E. Static/Dynamic Analyses for Validation and Improvements of Multi-Model HPC Applications. Ph.D. Thesis, Universit'e de Bordeaux, Bordeaux, France, 2015.
32. Saillard, E.; Carribault, P.; Barthou, D. Static/Dynamic Validation of MPI Collective Communications in Multi-Threaded Context. In Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP, Chicago, IL, USA, 15–17 June 2015; ACM: New York, NY, USA, 2015; Volume 2015, pp. 279–280.
33. Correctness Checking of MPI Applications. Available online: https://www.intel.com/content/www/us/en/docs/trace-analyzer-collector/user-guide-reference/2023-1/correctness-checking-of-mpi-applications.html (accessed on 18 June 2023).

34. Droste, A.; Kuhn, M.; Ludwig, T. MPI-Checker. In Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC, Austin, TX, USA, 15 November 2015; ACM: New York, NY, USA, 2015; pp. 1–10.
35. Keller, R.; Fan, S.; Resch, M. Memory Debugging of MPI-Parallel Applications in Open MPI. *Adv. Parallel Comput.* **2008**, *15*, 517–523.
36. Vetter, J.S.; de Supinski, B.R. Dynamic Software Testing of MPI Applications with Umpire. In Proceedings of the ACM/IEEE SC 2000 Conference (SC '00): Proceedings of the 2000 ACM/IEEE Conference on Supercomputing), Dallas, TX, USA, 4–10 November 2000; p. 51.
37. Hilbrich, T.; Schulz, M.; de Supinski, B.R.; Müller, M.S. A Scalable Approach to Runtime Error Detection in MPI Programs. In *Tools for High Performance Computing 2009*; Springer: Berlin/Heidelberg, Germany, 2010; pp. 53–66. ISBN 978-3-64211-260-7.
38. RWTH Aachen University. *MUST: MPI Runtime Error Detection Tool*; RWTH Aachen University: Aachen, Germany, 2018.
39. Hilbrich, T.; Protze, J.; Schulz, M.; de Supinski, B.R.; Muller, M.S. MPI Runtime Error Detection with MUST: Advances in Deadlock Detection. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, Salt Lake City, UT, USA, 10–16 November 2012; pp. 1–10.
40. Kranzlmueller, D.; Schaubschlaeger, C. A Brief Overview of the MUST MAD Debugging Activities. *arXiv* **2000**, arXiv:cs/0012012.
41. Forejt, V.; Joshi, S.; Kroening, D.; Narayanaswamy, G.; Sharma, S. Precise Predictive Analysis for Discovering Communication Deadlocks in MPI Programs. *ACM Trans. Program. Lang. Syst.* **2017**, *39*, 1–27. [CrossRef]
42. Luecke, G.R.; Coyle, J.; Hoekstra, J.; Kraeva, M.; Xu, Y.; Park, M.-Y.; Kleiman, E.; Weiss, O.; Wehe, A.; Yahya, M. The Importance of Run-Time Error Detection. In *Tools for High Performance Computing 2009*; Müller, M.S., Resch, M.M., Schulz, A., Nagel, W.E., Eds.; Springer: Berlin/Heidelberg, Germany, 2010; pp. 145–155. ISBN 978-3-642-11261-4.
43. Saillard, E.; Carribault, P.; Barthou, D. Combining Static and Dynamic Validation of MPI Collective Communications. In Proceedings of the 20th European MPI Users' Group Meeting, Madrid, Spain, 15–18 September 2013; ACM: New York, NY, USA, 2013; pp. 117–122.
44. Alghamdi, A.S.A.; Alghamdi, A.M.; Eassa, F.E.; Khemakhem, M.A. ACC_TEST: Hybrid Testing Techniques for MPI-Based Programs. *IEEE Access* **2020**, *8*, 91488–91500. [CrossRef]
45. Betts, A.; Chong, N.; Donaldson, A.F.; Qadeer, S.; Thomson, P. GPU Verify: A Verifier for GPU Kernels. In Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA, New York, NY, USA, 19–26 October 2012; pp. 113–131.
46. Basupalli, V.; Yuki, T.; Rajopadhye, S.; Morvan, A.; Derrien, S.; Quinton, P.; Wonnacott, D. OmpVerify: Polyhedral Analysis for the OpenMP Programmer. In *OpenMP in the Petascale Era: 7th International Workshop on OpenMP, IWOMP 2011, Chicago, IL, USA, 13–15 June 2011*; Proceedings 7; Springer: Berlin/Heidelberg, Germany, 2011; Volume 6665, pp. 37–53. [CrossRef]
47. Ye, F.; Schordan, M.; Liao, C.; Lin, P.-H.; Karlin, I.; Sarkar, V. Using Polyhedral Analysis to Verify OpenMP Applications Are Data Race Free. In Proceedings of the IEEE/ACM 2nd International Workshop on Software Correctness for HPC Applications (Correctness), Dallas, TX, USA, 12 November 2018; pp. 42–50.
48. Jannesari, A.; Kaibin, B.; Pankratius, V.; Tichy, W.F. Helgrind+: An Efficient Dynamic Race Detector. In Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing, Rome, Italy, 23–29 May 2009; pp. 1–13.
49. Nethercote, N.; Seward, J. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. *ACM SIGPLAN Not.* **2007**, *42*, 89–100. [CrossRef]
50. Gu, Y.; Mellor-Crummey, J. Dynamic Data Race Detection for OpenMP Programs. In Proceedings of the SC18: International Conference for High Performance Computing, Networking, Storage and Analysis, Dallas, TX, USA, 11–16 November 2018; pp. 767–778. [CrossRef]
51. Terboven, C. Comparing Intel Thread Checker and Sun Thread Analyzer. In *Advances in Parallel Computing*; IOS Press: Amsterdam, The Netherlands, 2008; Volume 15, pp. 669–676.
52. Intel(R) Thread Checker 3.1 Release Notes. Available online: https://registrationcenter-download.intel.com/akdlm/irc_nas/1366/ReleaseNotes.htm (accessed on 8 March 2023).
53. Sun Microsystems. Sun Studio 12: Thread Analyzer User' s Guide. Available online: https://docs.oracle.com/cd/E19205-01/820-0619/820-0619.pdf (accessed on 8 March 2023).
54. Serebryany, K.; Bruening, D.; Potapenko, A.; Vyukov, D. AddressSanitizer: A Fast Address Sanity Checker. In Proceedings of the USENIX Annual Technical Conference (USENIX ATC 12), Boston, MA, USA, 13–15 June 2012; pp. 309–318.
55. Serebryany, K.; Potapenko, A.; Iskhodzhanov, T.; Vyukov, D. Dynamic Race Detection with LLVM Compiler: Compile-Time Instrumentation for ThreadSanitizer. In *International Conference on Runtime Verification*; Springer: Berlin/Heidelberg, Germany, 2012; Volume 7186, pp. 110–114. [CrossRef]
56. Atzeni, S.; Gopalakrishnan, G.; Rakamaric, Z.; Ahn, D.H.; Laguna, I.; Schulz, M.; Lee, G.L.; Protze, J.; Muller, M.S. ARCHER: Effectively Spotting Data Races in Large OpenMP Applications. In Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS), Chicago, IL, USA, 23–27 May 2016; pp. 53–62. [CrossRef]
57. Hilbrich, T.; Müller, M.S.; Krammer, B. Detection of Violations to the MPI Standard in Hybrid OpenMP/MPI Applications. In *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*; Springer: Berlin/Heidelberg, Germany, 2008; Volume 3744, pp. 26–35. ISBN 3540294104.
58. Krammer, B.; Bidmon, K.; Müller, M.S.; Resch, M.M. MARMOT: An MPI Analysis and Checking Tool. *Adv. Parallel Comput.* **2004**, *13*, 493–500. [CrossRef]

59. Chatarasi, P.; Shirako, J.; Kong, M.; Sarkar, V. An Extended Polyhedral Model for SPMD Programs and Its Use in Static Data Race Detection. In *Languages and Compilers for Parallel Computing: 29th International Workshop, LCPC 2016, Rochester, NY, USA, 28–30 September 2016*; Revised Papers 29; Springer International Publishing: Cham, Switzerland, 2017; Volume 10136, pp. 106–120. [CrossRef]
60. Mekkat, V.; Holey, A.; Zhai, A. Accelerating Data Race Detection Utilizing On-Chip Data-Parallel Cores. In *Runtime Verification: 4th International Conference, RV 2013, Rennes, France, 24–27 September 2013*; Springer: Berlin/Heidelberg, Germany, 2013; Volume 8174, pp. 201–218. [CrossRef]
61. Gupta, S.; Sultan, F.; Cadambi, S.; Ivančić, F.; Rötteler, M. Using Hardware Transactional Memory for Data Race Detection. In Proceedings of the IEEE International Symposium on Parallel & Distributed Processing, Rome, Italy, 23–29 May 2009. [CrossRef]
62. Bekar, U.C.; Elmas, T.; Okur, S.; Tasiran, S. KUDA: GPU Accelerated Split Race Checker. In *Workshop on Determinism and Correctness in Parallel Programming (WoDet)*; Elsevier: Amsterdam, The Netherlands, 2012.
63. Zheng, M.; Ravi, V.T.; Qin, F.; Agrawal, G. GMRace: Detecting Data Races in GPU Programs via a Low-Overhead Scheme. *IEEE Trans. Parallel Distrib. Syst.* **2014**, *25*, 104–115. [CrossRef]
64. Zheng, M.; Ravi, V.T.; Qin, F.; Agrawal, G. GRace: A Low-Overhead Mechanism for Detecting Data Races in GPU Programs. *ACM SIGPLAN Notices* **2011**, *46*, 135–145. [CrossRef]
65. Dai, Z.; Zhang, Z.; Wang, H.; Li, Y.; Zhang, W. Parallelized Race Detection Based on GPU Architecture. *Commun. Comput. Inf. Sci.* **2014**, *451 CCIS*, 113–127. [CrossRef]
66. Boyer, M.; Skadron, K.; Weimer, W. Automated Dynamic Analysis of CUDA Programs. Available online: https://www.nvidia.com/docs/io/67190/stmcs08.pdf (accessed on 5 September 2023).
67. Li, P.; Li, G.; Gopalakrishnan, G. Practical Symbolic Race Checking of GPU Programs. In Proceedings of the SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, New Orleans, LA, USA, 16–21 November 2014; pp. 179–190. [CrossRef]
68. Bronevetsky, G.; Laguna, I.; Bagchi, S.; De Supinski, B.R.; Ahn, D.H.; Schulz, M. AutomaDeD: Automata-Based Debugging for Dissimilar Parallel Tasks. In Proceedings of the International Conference on Dependable Systems and Networks, Chicago, IL, USA, 28 June–1 July 2010; pp. 231–240.
69. Allinea Software Ltd. ALLINEA DDT. Available online: https://www.linaroforge.com/about (accessed on 5 September 2023).
70. Allinea DDT | HPC @ LLNL. Available online: https://hpc.llnl.gov/software/development-environment-software/allinea-ddt (accessed on 31 October 2023).
71. Totalview Technologies: Totalview—Parallel and Thread Debugger. Available online: http://www.Totalviewtech.Com/Products/Totalview.Html (accessed on 5 September 2023).
72. TotalView Debugger | HPC @ LLNL. Available online: https://hpc.llnl.gov/software/development-environment-software/totalview-debugger (accessed on 5 September 2023).
73. Claudio, A.P.; Cunha, J.D.; Carmo, M.B. Monitoring and Debugging Message Passing Applications with MPVisualizer. In Proceedings of the 8th Euromicro Workshop on Parallel and Distributed Processing, Rhodes, Greece, 19–21 January 2000; pp. 376–382.
74. Intel Inspector | HPC @ LLNL. Available online: https://hpc.llnl.gov/software/development-environment-software/intel-inspector (accessed on 8 March 2023).
75. Clemencon, C.; Fritscher, J.; Rühl, R. Visualization, Execution Control and Replay of Massively Parallel Programs within Annai's Debugging Tool. In Proceedings of the High Performance Computing Symposium, HPCS '95, Montreal, QC, Canada, 10–12 July 1995; pp. 393–404.
76. Arm Forge (Formerly Allinea DDT) | NVIDIA Developer. Available online: https://developer.nvidia.com/allinea-ddt (accessed on 5 September 2023).
77. Documentation—Arm Developer. Available online: https://developer.arm.com/documentation/101136/22-1-3/DDT (accessed on 5 September 2023).