Runtime Verification of Scientific Computing: Towards an Extreme Scale

Minh Ngoc Dinh1*, Chao Jin1, David Abramson1, Clinton L. Jeffery2
1 Research Computer Center and School of ITEE, The University of Queensland, Australia
2 Department of Computer Science, University of Idaho, USA
m.dinh1@uq.edu.au

Abstract—Relative debugging helps trace software errors by comparing two concurrent executions of a program - one code being a reference version and the other faulty. By locating data divergence between the runs, relative debugging is effective at finding coding errors when a program is scaled up to solve larger problem sizes or migrated from one platform to another. In this work, we envision potential changes to our current relative debugging scheme in order to address exascale factors such as the increase of faults and the nondeterministic outputs. First, we propose a statistical-based comparison scheme to support verifying results that are stochastic. Second, we leverage a scalable data reduction network to adapt to the complex network hierarchy of an exascale system, and extend our debugger to support the statistical-based comparison in an environment subject to failures.

Keywords—exascale computing; stochastic online verification

I. INTRODUCTION
Parallel computing will be subject to frequent faults and non-deterministic results in the coming exascale era. It is envisaged that exascale supercomputers will require a hybrid architecture with a deep memory hierarchy to expose massive parallelism orders of magnitude more than current petascale systems [1, 2]. In addition, power constraints will result in less reliable hardware components to save energy, such as proposed in NTV (Near Threshold Voltage) Computing [3]. Frequent failures and faults are expected to make results non-deterministic in future extreme-scale computers [4, 5].

This disruptive transition towards exascale requires significant changes in algorithms, software and programming methodologies. In order to exploit fine-grained parallelism efficiently, applications need to explore scalable designs using asynchronous algorithms with less inter-process communication [4, 6]. In addition, traditional check-pointing mechanisms to handle faults incur significant overhead in terms of both time and energy because checkpoint saves are expensive and both faulty and non-faulty processes must be relaunched. Solutions, such as the local-failure-local-recovery (LFLR) model, are practical, but result in non-deterministic execution [6], in which repeated runs do not have the same communication order or data movements. Overall, increased asynchrony and non-deterministic algorithms create challenges for bitwise reproducibility of computing results [5, 6], which need to be addressed by the scientific applications.

Scientific applications that motivate exascale computing, such as combustion and climate modelling, are complex because they model physical processes, through well-posed systems of equations (e.g. Navier-Stokes, Maxwell’s equations). The complex simulation process and its stochastic nature imply the uncertainty of key values at runtime. Dongarra et al. indicate that “requirements for bitwise reproducibility will need to be relaxed and will most likely need to be replaced with statistical concepts” [5]. This implies a key challenge in testing and debugging exascale codes, in which one needs to be able to detect subtle errors in very large data using robust statistical profiles, possibly obtained through data mining algorithms. Inspired by relative debugging [7, 8], we propose a statistical-based online validation method to examine the correctness of stochastic algorithms at runtime. This method derives the statistical features from a reference model, which is typically a small scale of execution that has been verified, and a suspect execution at large scale, and then compares them to detect soft errors.

To motivate our methodology, we study the statistical properties of a molecular dynamics code executing at different scales, and examine their similarity and difference statistically with both correct and incorrect executions. In addition, we extend the design of our relative debugger to support the statistical-based runtime verification in an environment subject to frequent faults. In particular, this paper makes the following contribution:

- A runtime verification methodology for stochastic algorithms-based scientific computing;
- A scalable and fault tolerant data collection and aggregation framework to derive the statistical features for a large scale scientific simulation;
- A proposed extension of a relative debugger which supports the runtime verification of stochastic algorithms.

The remainder of the paper is organized as follows. Section II discusses related works and our motivation. Section III presents the new validation methodology and Section IV describes a case study. Section V presents the extension of our parallel debugger. We discuss the applicability of our proposed technique in Section VI. Finally, Section VII presents our conclusion and future work.

II. RELATED WORK
Stochastic algorithms are expected to play an important role in addressing issues of scalability and non-determinism [5]. However, the computation results cannot be bitwise identical
due to both platform issues and algorithms’ built-in stochastic nature. This issue makes detecting errors difficult at large scales [5, 6]. The de facto development cycle of scientific computing is to start from small scale where results have been verified, and then switch to larger scales gradually. However, this flow needs to take into account non-deterministic issues in porting applications to exascale platforms. This section reviews the techniques for debugging parallel applications at a massive scale.

A. Scalable debugging techniques and tools

Debugging techniques for large-scale programs range from traditional “step-by-step” debugging, static analysis, through to symbolic analysis and dynamic verification. For example, TotalView [9] and Allinea DDT [10] are the two most popular debuggers that allow users to examine execution per step for highly parallel programs. For symbolic analysis, the Toolkit for Accurate Scientific Software (TASS) is a fine example [11]. Given a set of local state snapshots, TASS uses symbolic execution and explicit state enumeration techniques to evaluate runtime properties of MPI programs. To further explore full-scale debugging, the Stack Trace Analysis Tool (STAT) [12] analyzes an execution trace and partitions the threads into equivalence classes based on their behavior. These behavior classes can be visualized using 2D spatial and 3D spatial-temporal graphs and can help a developer isolate defects. AutomaDeD [13] models the control flow of MPI applications using a Markov model, and can detect abnormal processes in an automated and scalable manner. FlowChecker [14] detects bugs in the MPI library by checking if the delivered messages violate the specification defined by users. Most of these tools verify the behaviors of parallel computation at scales, and they need to address the potential exascale related issues such as frequent faults and non-deterministic computation.

B. Statistics-based debugging

Related works in using statistics to assist debugging tasks for sequential codes includes Zhou et al. [15], Michael et al. [16], and Sudheendra et al. [17]. They demonstrate that statistics-rule-based approaches are very promising in detecting bugs that do not violate any programming rules. DMTTracker [18] leverages a statistics-rule-based technique and provides a solution for parallel applications. The tool can automatically detect the cause of phenomena such as data corruption or deadlocks by observing data movements between parallel processing threads. The Statistical Debugging technique developed by Liblit et al. is another example that uses statistics for debugging [19]. Stochastic failures are reported multiple times and various statistical and modelling techniques can be used to deduce the likely location of the bugs. However, the goal of DMTTracker and the Statistical Debugging technique is only to isolate a certain class of bugs namely program runtime failure. They obviously cannot be used to identify bugs that do not abort the operation of the program but silently corrupt the final results.

C. Relative debugging

Relative debugging [7, 8] helps programmers establish the correctness of parallel applications ported between different platforms. It assists a programmer to locate errors by observing a divergence in relevant data structures between two versions of the same program while they are executing. The effectiveness of relative debugging has been demonstrated for different programming models, including MPI, PGAS, and hybrid models, such as MPI/OpenMP and MPI/OpenACC [20].

However, as we approach exascale, the bitwise reproducibility across different versions of the same program is not likely to hold. Key runtime data structures will not only be stochastic between different runs but also be in different scales because more processing cores are available. Therefore, the traditional bitwise comparison will not be sufficient in the future. In this paper, we propose a software framework to verify the unique features that scientific computing holds as underlying mathematics or physical laws. In addition, tools for the next generation of parallel computing should be fault/failure resilient [6]. Accordingly, we present designs to make our debugger fault tolerant itself.

III. RUNTIME VERIFICATION METHODOLOGY

Scientific programs often simulate physical processes using mathematical models. The computed states must follow domain-specific physical laws or mathematical rules, such as energy conservation and symmetries in physics. Violating these unique features means some undesired behavior occurs. Presently, ad-hoc and application-specific methods are used to validate scientific code during the development cycle [4]. However, these methods are either error-prone or labor-intensive, and difficult to reuse.

A. Approach

In most scientific simulations, key physical measures are represented using multi-dimensional arrays. While these variables are updated iteratively during the simulation, they are expected to uphold certain scientific laws, which can be verified by assertions which test various properties at run time. Importantly, assertions can be tested and verified statistically, regardless of the number of computing processes and the size of the computed data. In our recent work [21], we recognize the importance of extracting statistical information for debugging purposes and develop a ‘statistical assertion’ template to examine the statistical feature for one or multiple targeted variables. With this template, a user can describe a test hypothesis using descriptive statistics and data distribution models. In addition, users are also allowed to construct a complex reference model manually. However, creating a correct statistical model that the target execution can expect to follow is difficult, as it requires strong domain-specific knowledge and statistical skills.

In this paper we propose an alternative approach, namely using a small-scale execution of the code as a reference and extracting the statistical feature that is unique to the simulated science as invariant. In particular, at runtime, statistics are effectively drawn from both executions and then are compared to examine their equivalence. This approach is more practical than our previous thinking because the user does not need to specify an expected distribution. Further, the approach is novel because it actually allows a program running at a lower
resolution to serve as a reference code for a higher resolution model. In the past, we have had to use programs of the same scale since data is compared exactly.

The equivalence of two statistics is examined using a set of statistical tests including t-test, χ² goodness of fit test and Kolmogorov-Smirnov test, to name a few. The following specification extracts runtime data from the variable keyval at line 45 of source file code.c in both runs, and compares two histograms constructed using obtained data. If the statistical test result is bigger than the significance level α=0.02, the null hypothesis is accepted which implies that the runtime data from the 2 runs statistically matched.

```
set reduce histogram (8192, 100, 0.0, 1.0)
assert $ref::keyval@code.c:45~$sus::keyval@code.c:45>0.02
```

To support this function, we need a tool that can control the execution of multiple processes/threads and retrieve a massive amount of runtime data. In addition, the tool must be able to formulate, aggregate and compare (or estimate the difference of) data distribution models. The following section, using stochastic molecular dynamics as an empirical study, shows that several statistical properties can be verified across executions with different problem sizes and different numbers of parallel processes. Section V presents the extension of our relative debugger to support verifying statistical features across two programs at runtime in an environment subject to faults.

IV. CASE STUDY

To demonstrate our statistical-based technique, we used a parallel molecular dynamics code [22] written in C using MPI. This code uses the Lennard-Jones (LJ) potential in modelling a fluid, which is popular for investigating various liquid phenomena such as melting, the liquid-vapour surface and nucleation [23], and is a fundamental simulation in molecular dynamics. The simulation consists of a 3D cube that contains randomly positioned particles, with random initial velocities. We exercise 1000 time steps and at each time step, the system computes the new positions for all particles using the interaction force between themselves, and their current velocities. Because all particles have the same mass, their kinetic energy depends only on their speeds, which vary substantially. However, assuming the same initial conditions, consider different executions with different number of particles and/or different number of parallel processes, this scalar value is expected to spread according to a certain distribution (e.g. Maxwell-Boltzmann distribution [24]). In this experiment, we monitor particle speed as we vary two parameters (Table I) in hoping to detect anomalies in a simulation. The experiment was conducted on a Cray system with 268 compute nodes. We reserved 2,000 compute cores for this experiment.

<table>
<thead>
<tr>
<th>Process#</th>
<th>Problem Size</th>
<th>30</th>
<th>60</th>
<th>1,000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Particle#</td>
<td></td>
<td>491,520</td>
<td>983,040</td>
<td>16,384,000</td>
</tr>
</tbody>
</table>

For this statistical approach to work, the tool needs to (1) avoid false positives by ensuring that statistics extracted from bug-free runs, with larger problem sizes or with more processes, remain statistically the same; and (2) flag statistical differences when comparing against erroneous runs. To demonstrate these functions, we verify the correctness of the LJ code as we increase the number of processes from 30 to 60 and 1000 processes with problem sizes specified in Table I. We capture the data distribution of speed values by extracting histograms. Figure 1 shows normalised histograms for correct runs, captured at time steps 500 and 1000 respectively. The x-axis shows bins’ indices while the y-axis displays the ratio of each bin-value against the max bin-value. As shown in Figure 1, the distributions of speed values for all particles are closely similar between these runs.

To illustrate the expressive power and potential of comparing distribution of data values for detecting subtle data errors, we repeat the above runs while we inject a coding bug to slightly corrupt the computation for the force field in runs with 60 and 1000 processes. We use the execution of 30 processes as the reference. The histograms in Figure 2 show a clear shift in distribution of velocity values. While the differences in data distribution can be visually detected in Figure 2, our tool performs Kolmogorov-Smirnov tests to detect differences in these histograms at runtime. p-values from these tests are reported in Figure 3, which show strong similarity between correct runs and divergences against erroneous runs. Furthermore, we also capture and compare mean values (shown in Figure 4) reduced from these key data structures at different time steps during the executions to show that descriptive statistics can also be used to detect collective subtle data errors.

To conclude, without using statistics, such data divergence is hard to detect if (1) a code simulates a stochastic process (e.g. data values are randomly generated or initialized) and (2) a code is scaled to work with larger problem sizes and/or more processors.

![Figure 1. The similarity of histograms with correct executions](image1)

![Figure 2. Histogram difference between the reference and incorrect executions](image2)
The topology of the communication tree is balanced, and the statistical features are statistically tested for verification by the debug client. Finally, at the front-end (the root node of the communication tree), the global statistics are constructed for each application process, and then sends them to the debug client using MRNet. As the primary statistics are transferred upstream along the communication tree, MRNet filters located in the internal nodes gradually aggregate them. The client and servers are connected using the MRNet communication library [25].

Figure 5 illustrates the design that supports the control of multiple programs and comparison of runtime data. It can be used for online verification of stochastic algorithms. In particular, a reference program, which is normally at a small scale, and a suspect code, which is typically at a large scale, are launched together. To compare statistical features for the two applications, each debug server extracts the primary statistics for each parallel process, and then sends them to the debug client using MRNet. As the primary statistics are transferred along the communication tree, MRNet filters located in the internal nodes gradually aggregate them. Finally, at the front-end (the root node of the communication tree), the global statistics are constructed for each application process and are statistically tested for verification by the debug client. The topology of the communication tree is balanced, and the fan-out degree of each internal node is $m$. Given $n$ parallel processes, the depth of the tree is $d = \log_m n$. The multi-stage aggregation of primary statistics is an extension of the split-phase algorithm discussed in [21], which supports deriving statistical features, such as mean, standard deviation, and histogram.

Consider deriving a global histogram for data collected from $n$ parallel processes. Equation (1) illustrates the procedure of aggregation that occurs in each filter of the communication tree, where $h_0^i$ is the global histogram, $h_{f_j}^i$ is the primary histogram extracted for each parallel process, and $\oplus$ is the aggregation operation. The overhead of aggregating histogram is proportional to $O(\log_m n)$, which is scalable as the number of parallel processes increases, as shown in Figure 5.

$$h_{f_j}^i = h_{f_j}^{i+1} \oplus h_{f_j}^{i+2} \oplus \ldots \oplus h_{f_j}^{i+m-1}$$

$$i \in [0, d), j^i = m \times (i - 1), j^0 = 0 \quad (1)$$

Figure 5. The verification framework of Guard

### V. EXTENSION OF RELATIVE DEBUGGER

Our existing relative debugger Guard [8], executes two versions of the same program and compares key data structures between them synchronously at runtime. The most recent release, CCDB (Cray Comparative Debugger) [20], runs on heterogeneous platforms equipped with GPUs and collects data to be compared from a large number of back-end processes in a scalable manner. Guard’s architecture consists of one client running at the front-end and multiple servers running at the back-end, and each server controls the execution of one MPI process (with potentially many computing threads if using OpenMP or pthread models). The client and servers are connected using the MRNet communication library [25].

Figure 5 illustrates the design that supports the control of multiple programs and comparison of runtime data. It can be used for online verification of stochastic algorithms. In particular, a reference program, which is typically at a small scale, and a suspect code, which is normally at a large scale, are launched together. To compare statistical features for the two applications, each debug server extracts the primary statistics for each parallel process, and then sends them to the debug client using MRNet. As the primary statistics are transferred along the communication tree, MRNet filters located in the internal nodes gradually aggregate them. Finally, at the front-end (the root node of the communication tree), the global statistics are constructed for each application process and are statistically tested for verification by the debug client. The topology of the communication tree is balanced, and the fan-out degree of each internal node is $m$. Given $n$ parallel processes, the depth of the tree is $d = \log_m n$. The multi-stage aggregation of primary statistics is an extension of the split-phase algorithm discussed in [21], which supports deriving statistical features, such as mean, standard deviation, and histogram.

Consider deriving a global histogram for data collected from $n$ parallel processes. Equation (1) illustrates the procedure of aggregation that occurs in each filter of the communication tree, where $h_0^i$ is the global histogram, $h_{f_j}^i$ is the primary histogram extracted for each parallel process, and $\oplus$ is the aggregation operation. The overhead of aggregating histogram is proportional to $O(\log_m n)$, which is scalable as the number of parallel processes increases, as shown in Figure 5.

$$h_{f_j}^i = h_{f_j}^{i+1} \oplus h_{f_j}^{i+2} \oplus \ldots \oplus h_{f_j}^{i+m-1}$$

$$i \in [0, d), j^i = m \times (i - 1), j^0 = 0 \quad (1)$$

### A. Scalable debug servers

In a typical HPC system, one compute node consists of multiple cores and each core is assigned one parallel process. Guard’s existing design spawns one debug server per parallel process and a debug server co-locates with the corresponding application process on the same core. Therefore, one compute node runs multiple debug servers and this results in resource competition between the debugging daemons and the application processes located on the same compute node. This approach is unlikely to scale for exascale machines, in which one compute node potentially contains an even larger number of cores.

To improve Guard’s scalability in handling debug servers and decrease unnecessary resource competition at the back-ends, our proposed design spawns one debug server per compute node. Therefore, a debug server controls all of the parallel processes running on the same compute node, using...
GDB (GNU Debugger)’s multiple inferiors. Nevertheless, resource competition between debugging daemons and the application’s processes is inevitable at the back-end, unless spare cores are allocated for debug servers. However, in comparison to the existing design, a single debug server per compute node significantly reduces the resource usage, which consequently lessens the resource competition issue between debugging daemons and the application’s processes. Further, this enhancement potentially reduces the number of communication connections required by the back-end servers.

Each debug server is assigned a unique identifier, called a Server ID. We use the bitvector data structure to represent a group of servers where each bit identifies a server using its index value. Regarding sparse bitvector, to save significant storage space, we introduce range_set [20]. For the rest of this paper, we use bitvector in our enhanced design.

B. A fault tolerant architecture

At runtime, faults can occur either in the debugger and/or the parallel application. Accordingly, handling failures consists of two steps: restoring the debugger, and recovering the affected application. Tolerating faults requires the system to first detect them, and then recover any lost resource, including communication processes, debugger servers, and application processes. Accordingly, Guard’s enhanced design aims to handle a single point failure in the back-end debug servers. Guard’s front-end client is responsible for detecting failures and recovering faults.

1) Failure detection: Guard relies on MRNet’s built-in fault tolerant mechanisms to detect failures at the back-ends. MRNet can report the failures in any tree nodes and it can also recover the loss of internal nodes [25]. By registering a notification function to the MRNet front-end using the callback interface, register_EventCallback, the front-end client receives a failure event notification for any failures that occurred in the debug servers (leaf nodes), and the communication nodes at the back-end. In addition, each back-end server monitors the live state of the parallel processes under its control and notifies the debug client when one or more parallel processes crash.

To track the crashed servers, the debug client maintains a set of servers using a bitvector, called live_set. By default, every bit in live_set is 1. If a debug server crashes, its bit is set to 0. Every time a failure event is notified or a crashed debug server is recovered, live_set is updated accordingly.

2) Failure handling: A typical fault-resilient application handles faults in two ways: 1) using its resilient design to work normally even if failures occurred to some of its processes, often used in Monte Carlo simulations, and 2) depending on the system’s fault tolerant support, such as check-pointing, to bring it to a previous correct state and continue the execution. To support the second approach, our debugger needs to adhere to appropriate procedures to recover crashed debug servers and affected application processes.

The second approach requires the debugger to use the system’s fault tolerant mechanism to restore the debugging session. When a compute node crashes, both the debug server and the parallel processes running on it are affected. We assume that recovery mechanisms, such as check-point/restart (CR), carry over the application’s correct state collected before the failure occurs and restore lost application processes. Therefore, the debugger needs to find out in which compute node CR re-generates the lost parallel processes; re-launch the lost debug server on that compute node and attach the newly launched server to the MRNet tree. The restored server also needs to attach to the newly generated processes on that compute node, and finally notifies the debugger to continue the debugging session without the user’s interference. The detailed implementation of this workflow is outside the scope of this paper.

Supporting fault resilient programs, such as Monte Carlo simulations, could be less complicated because the crash of a few parallel processes does not impact the execution of the whole application. However, several actions are required for the debugger to stay functional, especially the data aggregation framework for online statistical verification, in the event that some debug servers or some parallel processes crash. With the current design, the data aggregation process relies on the fixed number of back-end debug servers; thus it fails to function if any server crashes or becomes nonresponsive. We describe an enhancement below to address this issue.

C. Resilient collection and comparison of statistical features

Each statistical test is assigned a unique verification identifier, called VID. In debugging/verifying a time-step based simulation, a test could potentially be examined multiple times within a session, and each iteration is assigned a unique sequence identifier, called SID. For every statistical test, its SID starts from 0 and is incremented each loop iteration. In addition, every test is associated with a number of parallel processes specified by its definition, called proc_set. The processes in a proc_set are controlled by a set of back-end debug servers, called target_set, which is a bitvector. Unlike live_set, the default value of each bit in target_set is 0. Only if one server is present for the statistical test, its bit in target_set is 1.

The rest of this section uses histogram as an example to illustrate collecting statistical features and comparing them in a resilient manner. The internal data structure representing a histogram, h, consists of 4 parts: 1) VID, 2) SID, 3) the bins of histogram, and 4) the associated target_set, as illustrated in

Figure 6. The aggregation of histogram messages
Figure 6. This data structure is used to represent the initial histogram at back-end servers, partial histograms aggregated in the MRNet filters and the final global histogram constructed in the debug client.

1) Collection: Each debug server collects histograms for the parallel processes that are involved in the statistical test and under its control. These histograms are merged locally and the result histogram is forwarded to MRNet. The initial target_set only contains the debug server itself. Unlike the current design that uses MRNet WaitForAll filter, the aggregation in our new design occurs asynchronously using DoNotWait filter. Whenever the filter receives aggregation requests from its child nodes, it categorizes them into different groups according to VID and SID. For the requests within each group, their histograms are aggregated and the associated target_set are merged before forwarding them upstream, as depicted in Figure 6. With DoNotWait filter, each MRNet internal node forwards received messages upstream without waiting for all of its child nodes. Therefore, even if any leaf node crashes and provides no response, the data collection/aggregation keeps progressing.

2) Comparison: At runtime, suppose some servers of a target_set crash at the back-end. The exact debug servers that a comparison should consider is the intersection of its target_set and the current live_set, which is called curr_set, as defined by (2). The debug client updates curr_set for each ongoing statistical test according to every notification of failure event.

\[ \text{curr_set} = \text{target_set} \cap \text{live_set} \] (2)

Due to the asynchronous aggregation process, partial histograms do not arrive at the front-end at the same time, but gradually. Similar to each filter, the debug client aggregates histograms according to their VID and SID. For each global histogram, when its associated bitvector is identical to its curr_set, it is ready for comparison against its reference counterpart. Therefore, the debug client needs to maintain a dictionary of global histograms. For example, Figure 7 shows 5 continuous histograms for a statistical test, from S1 to S5. The state of each global histogram starts with “not ready”, and updates to “ready” after its global form is constructed. When it is compared, its state changes to “verified”.

3) Resilience: While collecting and aggregating the statistics, if some parallel processes crashed at the back-end, their debug servers can safely ignore them. However, curr_set is updated at the front-end after the failure is detected. Therefore, the aggregation and comparison is not affected by any fault that occurs to the leaf nodes of the communication tree.

If a communication node crashes, the MRNet tree can restore it automatically. But this results in missing aggregation messages, and leaves some histograms in the ‘not ready’ state infinitely. As a result, some statistical tests never trigger because the comparisons of corresponding histograms never occur. For example, in Figure 7, S4 is never ready due to message loss. However, this has no impact on the subsequent examination, such as both S1 and S2. As discussed in Section IV, the divergence of statistical features generated by the undesired behavior is persistent throughout the whole execution. Therefore, errors can still be detected later by the tool and thus the tool is resilient to system failures or faults.

D. Asynchronous execution of verification

In the current architecture, Guard’s client uses a dataflow engine to handle breakpoints, retrieving and reducing data, and then comparing them. However, the existing dataflow engine imposes synchronization on the back-end debug servers; thus it incurs a noticeable latency. Examining the statistical features at runtime must have a low overhead. Therefore, asynchronous comparison mode is preferred. We envision changes to the dataflow engine to support asynchronous data handling, as depicted in Figure 8.

After a statistical test is defined and initialized, the debugger sets appropriate breakpoints and each debug server runs its parallel processes, while the debug client waits for receiving responses from servers, including aggregated statistics. After one debug server detects a breakpoint hit, it extracts runtime data from the target variable and then releases the process to continue its execution. The debug server then performs the required reduction task to construct the primary statistics. After the debug server collects the statistical features for all of the involved MPI processes under its control, they are forwarded to MRNet for aggregation. When the debug client receives an aggregation message, it checks if the according global histogram is ready for comparison.

---

**Figure 7.** The dictionary of global histograms maintained for a statistical test

---

**Figure 8.** The asynchronous verification dataflow

---

In case the verification failed (the test failed), the error along with the statistical results are saved and reported to the user. If both executions finish without any assertion violation or any statistical test failed, a test report is also saved, and further verification tests can take place. During the verification process, no synchronization is enforced across back-end
servers for runtime data extraction, reduction or transmission, while the client sets no barrier in communicating with servers. This avoids any synchronization overhead.

VI. DISCUSSION

To ensure that differences observed at different scales are not just due to step functions or inflection points where the problem changes as it changes scale, the statistical comparison framework proposed in this paper assumes that relatively consistent mathematics are involved in the scientific problem. This assumption is a limitation in the applicability of the technique because a user needs to possess firm domain-specific knowledge in picking the right data structures to compare between versions. In addition, we admit that the current statistical-based verification scheme does not know how to handle variables that do not possess known statistical features or physical distributions; thus further work is required. Nevertheless, we argue that the technique is still valuable and widely applicable because most modern scientific applications are developed based on well-known standard frameworks such as computational fluid dynamics (CFD) [26], molecular dynamics (MD) [24], and statistical physics [27]. These frameworks enforce statistical properties that govern scientific simulations; thus can be used to verify the code as part of the testing process. For instance, consider the CFD framework. Because the fundamental basis of most CFD problems are the Navier–Stokes equations, the turbulence equations, and the mass and energy conservation equations, a typical CFD simulation must ensure conservation of quantities such as mass, total energy and momentum. In addition, implementations using probability density function (PDF) methods, for example, also imply that runtime values of applications’ attributes follow a certain density function [26].

The statistical comparison process can be implemented outside of Guard to provide a lightweight plugin for existing applications and deliver a ‘minimally intrusive’ verification process. This will be considered for future work since a more

in-situ data processing tool will be useful not only in identifying computational errors but also in giving the users immediate insights into the observed scientific process. However, Guard is still a valuable tool because it provides conventional parallel debugging supports and functions as well. While failing a statistical test can only notify issues in the runtime data, being able to perform conventional debugging activities might help to locate the coding error.

VII. CONCLUSIONS

As supercomputing and HPC systems evolve towards the next generation, failures and non-determinism creates new challenges in testing and debugging scientific software. This position paper proposes verifying the runtime state of scientific simulations by examining the statistical features in accordance to certain underlying mathematics and physical laws, as dictated by the physics. Using a stochastic molecular dynamics application as a case study, we demonstrate that the extracted statistical features remain consistent across executions with different problem sizes and/or with different number of parallel processes. Therefore, the statistical features derived from a small size run, which is typically validated, can be used as invariants to examine the runtime correctness of the target large-scale execution. This innovative method allows using the simulation at a lower resolution as a model to verify the execution at a higher resolution. It is totally different from our previous comparison methodology that can only examine the exact values of the runtime data, provided that they have the same size. We also envision the necessary extension to our relative debugger to support this new function in an environment subject to failures, including an extension of fault tolerance and a resilient data-aggregation framework.

The current approach is not generic enough to handle arbitrary parallel programs. Future work will focus only generalizing the statistical framework and applying our method to other domains of scientific applications. We will also study and deliver performance study on large-scale machines.

REFERENCES


