Runtime verification of scientific codes using statistics

Minh Ngoc Dinh$^1$, David Abramson$^1$ and Chao Jin$^1$

$^1$ University of Queensland, Brisbane St Lucia, Queensland, Australia.
m.dinh1@uq.edu.au, david.abramson@uq.edu.au, chao.jin@uq.edu.au

Abstract
Runtime verification of large-scale scientific codes is difficult because they often involve thousands of processes, and generate very large data structures. Further, the programs often embody complex algorithms making them difficult for non-experts to follow. Notably, typical scientific codes implement mathematical models that often possess predictable statistical features. Therefore, incorporating statistical analysis techniques in the verification process allows using program’s state to reveal unusual details of the computation at runtime. In our earlier work, we proposed a statistical framework for debugging large-scale applications. In this paper, we argue that such framework can be useful in the runtime verification process of scientific codes. We demonstrate how two production simulation programs are verified using statistics. The system is evaluated on a 20,000-core Cray XE6.

Keywords: Runtime verification, debugging, scientific simulation, parallel computing, distributed memory

1 Introduction

Runtime verification refers to the process of monitoring and analyzing information extracted from a running system in order to detect and possibly respond to behaviors that violate expected properties. It is used in testing scientific simulations to examine whether the expected scientific models are correctly developed. As the complexity of parallel programming increases, however, it is challenging to guarantee the correctness of large-scale scientific applications using traditional verification methods.

Large-scale scientific programs often generate enormous multi-dimensional floating-point structures and these complex structures are distributed across thousands of parallel processes. These characteristics lead to several challenging issues with mining the runtime data for verification purposes. First, because the data is so large, employing traditional visualization techniques become impractical at runtime. While techniques such as graphical display or animation can help visualizing very large datasets, it is not always possible to “see” the errors in the massive amount of data. Second, the complexity of present parallel programs increases significantly due to parallelization techniques such as message passing, halo-cells/ghost-band, process synchronization and load balancing. Applying these parallelization techniques may generate additional concurrency-related errors that exacerbate the difficulty of enforcing the correctness of scientific computing. Although many tools have been developed...
to mitigate the difficulty of testing and debugging parallel programs, typically these tools focus on the
behaviours of concurrent execution instead of guaranteeing the correct features of scientific models.

We argue that examining the statistical features extracted from the runtime data of large-scale scientific applications is an effective way to verify that statistical properties of the scientific laws hold. In particular, the massive amount of floating-point data computed by scientific applications normally follows the statistical features enforced by the scientific models (more details are discussed in Section 2). In fact, other than generating raw data, most high-performance software also produces patterning information in the form of histograms, probability distributions or data models. These statistics not only give the users insights to the observed phenomena, but also sometimes display unusual details of the computation. However, instrumenting the source code in order to capture statistical attributes delays the verification process because scientists often need to separate the recording of data from its analysis. Hence, providing a runtime verification tool that enables programmers to compute and compare ad-hoc statistics against user-defined statistical models without requiring them to recompile their source code addresses this issue. Furthermore, a tool that supports tracing of statistical discrepancies at runtime will allow close monitoring of the application even though the state could be very big. However, to the best of our knowledge, no existing tools efficiently support observing the statistical features for a large amount of online data.

Our previous work provides a parallel assertion template, called a statistical assertion and presents the preliminary results of the framework on using statistics for debugging purposes [1]. Statistical assertions enable programmers to devise statistical hypotheses and to verify them against the (possibly distributed) runtime data. While proven to be useful for debugging large scientific codes, the major difficulty in using statistical assertions (or assertions in general) is the need for users to know where and how an assertion will be triggered. In this work, we introduce a new statistical template called a statistical watchpoint to verify statistical properties across numerical time steps. We also further enhance the semantics of the statistical assertion template. The new features are evaluated with real production codes. Specifically, this paper presents the following contributions:

- A study on scientific applications to highlight the benefits of using statistics in the runtime verification process.
- Several generic templates such as the history variable and the statistical watchpoint that enable a user to specify practical testing hypotheses.
- Case studies that demonstrate the detection of known bugs in two production scientific codes, using the proposed statistical method.
- Evaluation of the data-reduction framework on a moderate-size supercomputer to show the scalability of the runtime verification process.

The paper is structured as follows. Section 2 presents a study of different types of scientific simulations to demonstrate the necessity of statistical verification. In sections 3 and 4, we describe the design and the implementation of our statistical-based verification framework. Section 5 delivers case studies showing two production programs, LAMMPS [2] and FLASH [3] are verified using the proposed technique. The analysis and evaluation of the performance including the observed overheads and the speedup obtained on a Cray XE6 system is provided in Section 6. Further, we openly discuss the practice of using statistical assertions and watchpoints in Section 7. Related work is presented in Section 8, and conclude the paper in Section 9 with some discussion about future enhancement.

2 Statistical verification for scientific code

A scientific model is an abstract representation of a physical system in which scientific laws such as conservation of energy and mass govern the design, and a scientific simulation is the implementation of such model. Because the aggregated attributes found in a simulation evolve in predictable ways
during its course, unexpected values can be used to detect potential errors in the model or in the code that implements the model. Here, we describe several trends in modelling including computational fluid dynamics, molecular dynamics, and statistical physics. We highlight the statistical properties that govern scientific simulations; thus can be used to verify the code as part of the testing process.

2.1 Computational fluid dynamics (CFD)

Computational fluid dynamics (CFD) uses algorithms and numerical techniques to describe and analyses the interaction between liquids and gases in a finite volume [4]. Applications of CFD include weather forecasting, aerospace design, and biomedicine, to name a few [4]. A CFD simulation can be implemented using different methods such as the finite volume method, the vortex method, probability density function (PDF) methods, etc. Because the fundamental basis of most CFD problems are the turbulence equations, the mass and energy conservation equations, and the Navier–Stokes equations, a CFD simulation must ensure conservation of quantities such as mass, total energy and momentum. Also, in simulating the fluid flows through an inlet, the total pressure recovery always stays constant or decreases. Such constraints can be asserted and provide verification of the code since the consistency relations are usually a statement of some analytic result. Further, consider the PDF method which gives the probability of the velocity at point \( x \) being in a certain time interval. The general distribution of velocity values at time \( t_i \) must follow a certain density function (for instance Gaussian distribution function). Such attributes can also be tested to ensure the correct computation after every time step.

2.2 Molecular dynamics (MD)

A molecular dynamics (MD) simulation describes the physical movement and the interaction of atoms and molecules for a period of time [5]. Attributes such as position, and velocity are computed by solving the equations of motion, while interaction between the particles and potential energy are determined by molecular mechanics potential functions. MD is useful for studying the structure, dynamics/thermodynamics of biological molecules and their complexes.

Similar to CFD, MD also enforces an energy conservation law, which states that the total amount of energy in an isolated system remains constant. Therefore, a drift of this quantity may signal coding errors. In addition, in a typical MD simulation, particles interact with their neighbors, and their speeds are updated accordingly. While the speed varies a great deal, from very slow particles to very fast ones, this scalar value spreads according to the Maxwell-Boltzmann distribution [5]. Therefore, monitoring particles’ speeds can help detecting anomalies in a simulation. Such observation can also be specified as an assertion.

2.3 Statistical physics

Statistical physics uses statistical methods, such as probability theory, to describe the behavior of an ensemble of particles in terms of physical laws governing particle motions [6]. While presenting a stochastic process, the state of the system can be deterministically represented by a set of values with a probability distribution. Such determinism can be verified at runtime using statistical invariants, and make testing stochastic simulations more practical, whilst reducing the complexity of processing raw data. For instance, the most widely studied in statistical physics field is the Ising model [6] where each site (i.e. a particle) interacts with all others in the system to update its magnetic state. Even though the system is said to be stochastic, the energy decreases as the simulation evolves and the total energy of the system can be described by the “Hamiltonian” \( H \). A simple way to calculate \( H \) for a site \( i \) is to sum over all pairs of sites which are nearest neighbours if the Euclidean distance between them is 1. In addition, the probability model for states is given by the Boltzmann or the Gibb distributions. These attributes can be expressed as statistical assertions.
3 A statistics-based verification framework

Verifying the statistical correctness of scientific codes is realized by either testing statistical hypotheses, or analyzing runtime data to identify abnormal statistical values. First, users can formulate a statistical hypothesis by defining a distribution model using a random number generator and comparing it against statistical features derived from a target data structure. Such a comparison can highlight undesired details and allow a user to reason about a particular computation. Second, because adjacent time steps in a simulation show high data correlation, computing statistics from runtime data and keeping track of those values between time steps can help identifying potential errors and outliers.

The statistical assertion template as described in [1] allows a user to devise a statistical hypothesis, which might be a distribution model, and performs comparison in two modes: 1) interactive mode in which statistical attributes are verified at a breakpoint; and 2) assertion mode which supports automatic comparison of ad-hoc statistics. We briefly review statistical assertion and describe new developments in this template. Further, we describe a novel statistical watchpoint template.

3.1 Statistical assertion

Statistical assertions allow users to compare statistics extracted from two data structures, instead of comparing the exact values. For example, it is possible to assert that the number of elements in an array needs to be in a specific range. Statistics can be used to reflect scientific knowledge behind a computation, thus by using statistical assertions; a user can integrate such knowledge into the verification process and transform it into runtime invariants that ensure the correct execution.

In our previous work, we described some core components such as the data reduction engine and the templates to enable user-defined statistics and data models [1]. First, we provide both built-in functions and templates for users to define their own statistics. Notably, given the underlying parallel platform, statistics can be computed in parallel using the split-phase operation [1] which involves the computation of the primitive statistics for each piece of partitioned data in parallel, and merging all of the primitive statistics to form a completed required statistic. Second, we develop the semantics to support the creation of random variate using random number generators. A random variate is defined using a probability distribution function such as Gaussian, Cauchy etc. Users can also set sample size and other required parameters including mean, standard deviation, and scale parameter values.

In this work, we introduce history variables that allow a statistical hypothesis to reference a limited amount of history (i.e. an observation). Observation variables record the computational results occurring at time $t$. In monitoring a simulation, such observation variables could be asserted against historical records in order to verify that the simulation has not drifted from its expected course. A history variable is a user-defined variable and can be created and assigned a value using the history command, given three parameters. The first parameter is a program variable while the second and third parameters are the min/max number of historical records. For example, the following commands create a history variable called $etot$, which captures the historical total energy value after every simulation time-step. We enable the reduction function stdev; thus the following compare command uses the standard-deviation value reduced from $etot$ to compare against $0.1$. Note that no comparison is conducted until there are 10 or more records in the history variable $etot$.

```
history etot $a::dvalue@"thermo.cpp":1521 10 100
set reduce stdev; compare etot < 0.1
```

3.2 Statistical watchpoint

By definition, a watchpoint [7] is assigned to a particular variable and is used to stop execution whenever the variable is updated, without knowing the location where this may happen. We define a statistical watchpoint as a watchpoint which is triggered when (1) the value of a variable changes and
(2) the retrieved statistics indicate that the runtime data changes significantly. Several primitive statistics are required in each statistical watchpoint. Examples are mean, median, standard deviation and histogram. When a watched variable is updated, relevant statistics are computed to perform required statistical test to determine whether statistically significant difference occurs in the watched variable. The syntax below specifies a statistical watchpoint which invokes a Student’s t-test.

```plaintext
statwatch $a::dvalue#t_test(df, α)
```

### 3.3 Statistical tests

Assume that we are ‘watching’ an array-based variable in a scientific simulation program. In our runtime verification context, a scientist is often more interested in the statistical difference between two sets of numbers rather than in the raw values themselves. A simple statistical test to indicate whether or not the difference between two sets’ means is significant is the *t*-test. Required statistics such as mean, variance and standard deviation are computed and collected from the target variable. They are then used to compute the *p*-value and significant level $α$ providing the degree of freedom $df$.

Besides comparing datasets, verifying statistical attributes also requires the comparison of two abstract data models, for instance the comparison of two histograms. This requires statistical tests such as the $χ^2$ goodness of fit test. Consider the below specification. The *randset* command defines *tmodel* as a random variable consisting of 100,000 samples chosen from the Gaussian distribution with standard deviation of 0.05. The *assert* command describes the comparison of the histogram constructed from the program variable *dt_vals*, and the histogram generated using the local variable *tmodel*. The use of the *estimate operator* “~” indicates a statistical test is required for a particular comparison query. If the $χ^2$ test result is smaller than the significance level $α=0.02$, the hypothesis is accepted and the runtime data at that breakpoint follows the Gaussian distribution model.

```plaintext
randset tmodel “gaussian” 100000 0.05
set reduce histogram(1000,0.0,1.0)
assert $a::dt_vals@code.c:10 ~ tmodel < 0.02
```

### 4 Design and implementation

Our verification framework requires a basic set of functions that support process control and data inspection. Therefore, our verification tool integrates the functions of a conventional parallel debugger, called Guard. In our previous work, we developed a method that parallelizes the reduction of large runtime data into statistical values [1]. In this paper, we only focus on the parts that are used for runtime verification and for evaluating statistical templates such as assertion and watchpoint.

#### 4.1 Computing statistics in parallel

The proposed verification tool needs to invoke and control multi-process parallel programs; thus we base our implementation on a parallel debugger, Guard. A *Statistic API*, which consists of both built-in statistical functions and functions to support user-defined statistics, is integrated into Guard. The process to verify a statistical attribute for a large dataset that distributes across processes is fully discussed in [1]. In particular, this process handles arbitrary data reduction requests in order to retrieve required statistics. We discuss several parallel templates used to encapsulate such requests below. Note that, a statistical attribute can be verified either against the global state of the program (i.e. involving all processes) or its subset. Processes not involved with the statistical test will continue to run and they cannot modify the under-evaluation data because a backend server makes copy of such data.
4.2 Statistical assertion templates

To efficiently handle user-defined data models (we call this feature a deferred variable [1]), only the definition of the model is stored at the frontend client while the content is populated at the backend servers. This allows using the available computing resources that are otherwise unused at the breakpoints to populate the content of the deferred variable and extracting relevant statistical attributes in parallel. We use the Extract&Evaluate template to evaluate the deferred variables (Figure 1). The special symbols $\oplus$ and $\odot$ represent a synchronization node and a merge node in the graph respectively. Complex operations such as COMPARE or DISPLAY are encapsulated into sub-graphs and presented as boxes in the diagram for simplicity purpose. For a comparison query that involves a deferred variable, this template takes arguments including a program variable $V$ and a deferred variable $X$. The basic Extract template takes three parameters consisting of a variable name, plus file name and line number to define a breakpoint location. This information is used in a number of sub-graphs including SET_BREAKPOINT, GO, WAIT_BREAKPOINT and READ_VAR. For example, an assertion statement such as:

```
assert $p::v1@"code1.c":35 > randomset
```

will result in the generation of the ASSERT description:

```
ASSERT($p, "v1", "code1.c", 35, randomset, ")
```

Upon receiving an Extract&Evaluate request, a backend server returns a structure with two datasets or two set of statistics. Supporting parallel reduction of runtime data requires a template to encapsulate the reduction description so that backend servers can perform reduction individually. This is realized by using the Extract&Reduce template which has an extra parameter that describes the reduction function to apply on the runtime data at the backend (Figure 2).

4.3 Statistical watchpoint template

As mentioned, our verification tool is based on primitive debugging engine GDB [7]. Therefore, a statistical watchpoint is supported via the hardware watchpoint provided by GDB. When a watched variable is updated, GDB triggers the backend debug servers to retrieve a list of attached statistics for this variable. We modify the Extract&Reduce template (as shown in Figure 3) to encapsulate and instruct our backend servers to compute the required statistics. Statistical tests are conducted (as discussed in Figure 3. Statistical watchpoint template
Section 3.3) to determine whether the variable value has statistically changes. Significant changes will prompt the tool to halt at that execution point for the user to examine the state of the program.

5 Case Studies

We describe two case studies that demonstrate the use of statistical techniques in verifying the runtime correctness of large-scale parallel scientific programs. The first case study targets a MD simulator called LAMMPS [2] while the second examines a hydrodynamics code called FLASH [3]. Both programs solve a system of PDE equations with a large number of processes, while manipulating large amounts of data. We use statistical assertions to demonstrate the verification process.

5.1 Case study 1 – LAMMPS

Large-scale Atomic/Molecular Massively Parallel Simulator (LAMMPS) is an open-source classical MD code [2]. It can simulate atomic, polymeric, biological, metallic systems using a range of force fields and boundary conditions. LAMMPS is developed with C++/MPI. Recently in LAMMPS, two real errors have been reported when certain statistical properties were violated [8]. We describe several statistical assertions that can be used to detect these errors in a simulation with 2,048,000 atoms.

Distribution of velocity values

In a MD system, particles interact with neighbors and their velocity values vary substantially, from very slow particles to very fast ones. However, this scalar value spreads according to the Maxwell-Boltzmann distribution [5]. Users reported an error where this condition is violated in a simple simulation. The second error is a wrong index value that was used to initialize velocity values in source file `velocity.cpp`. This bug prevents from using ramped velocity distribution in the z direction. Accordingly, we developed several assertions to monitor the distribution of velocity values and verify these errors. After the initialization routine completes and after a time-step, velocity values are obtained to construct histograms (called `velocity histograms`), while the expected histograms are created with `randset` commands for the Gaussian and Maxwell-Boltzmann distributions. The reported errors can be detected by comparing the expected histograms with the velocity histograms (Figure 4).

Consider the script below. The `randset` commands build the datasets of 2,048,000 samples from both Gaussian and Maxwell-Boltzmann distributions. For the Maxwell-Boltzmann distribution, two constants are required: the temperature $T$ and the particle’s mass $m$. The `assert` commands request both the velocity data and the random number set to be reduced to histograms with 100 bins and data ranging in $[-7.0,7.0]$ and $[0.0,8.0]$.

```
randset mb maxwell-boltzman 256000 1.44 1.0
randset gauss “gaussian” 256000 0.2
```

Figure 4. Histograms showing distributions of velocity values (a) Gaussian (b) Maxwell-Boltzmann

* Each process generates 256,000 samples. With 8 processes running in parallel, we generate a total of 2,048,000 samples.
† Because the velocity variable is a dynamic 2D array, we need to cast the pointer to a 2D array with fix sizes.
Conservation of energy and momentum

In an isolated MD system, forces are time independent; thus the total energy, which is the sum of kinetic and potential energies, should stay approximately constant [5] compared to the initial provided total energy (via initial temperature). A drift of this quantity may therefore signal errors. Further, one should also monitor the total momentum of all particles. This was initially normalized to zero and should stay very small during the simulation. To verify these properties, the standard-deviation of energy values can be tested after each time step to ensure that total energy values don’t vary significantly. Below, we create a history variable called `etot` to capture the historical total energy values (retrieved from `dvalue`), and compute the standard-deviation value after every simulation time-step. Similarly, in the second assertion, the total momentum value of all particles is reduced using their velocity values and is compared against 0. Small numerical errors is masked with the `set error`.

```plaintext
set reduce histogram 100 -7.0 7.0
assert $a::"(double[256000][3])**v"@velocity.cpp":313 ~ gauss < 0.02
assert $a::"(double[256000][3])**v"@velocity.cpp":118 ~ gauss < 0.02
assert $a::"(double[256000][3])**atom\rightarrow v"@verlet.cpp":287 ~ mb < 0.02
```

5.2 Case study 2 – The FLASH code

FLASH is a high performance hydrodynamics code for studying the nuclear/thermonuclear flashes and also the fully compressible, reactive flows found in astrophysical context [3]. It solves a range of PDE equations and supports simulations from X-ray bursts to Richtmyer-Meshkov instabilities. Integrated with an AMR package called ParaMesh, FLASH supports a block-structured adaptive grid in which resolution elements are placed only where they are needed most [3]. FLASH is MPI based.

Recently, a bug was discovered in the mapping of particles to the mesh, which disrupts the conservation of total mass as a simulation progress [9]. In FLASH, the total mass of a zone can be computed using the density value and the volume of the zone. While the density of the particles in a zone should vary, the total mass value remains constant due to the law of mass conservation. This attribute can be verified using a statistical assertion that monitors the total mass values (over time). The bug was found in FLASH3.3 and fixed in FLASH4.0. We apply the assertion below to both versions to verify that the error is fixed. The script creates a history buffer that holds historical values of variable `gsum` obtained at line 184 in `IO_writeIntegralQuantities.F90`. When more than 10 records are in the history buffer, standard deviation is computed to verify that the total mass stays constant.

```plaintext
history mass $b[0]::"gsum(1)"@ "IO_writeIntegralQuantities.F90":184 10 100
set reduce stdev; assert mass < 0.1
```

6 Performance evaluation

In this section, we are interested in the following measurements at scale. First, a statistical assertion can comprise of user-defined reduction routines and data models, and history variables. We analyze the performance of using history variables separately because it has to incrementally build up the history buffer with runtime data. Second, because the reduction process has a sequential aggregation phase, the overall performance will be bound by a sequential factor, according to Amdahl’s law. We perform a strong scaling experiment to measure this factor and the point where strong scaling no longer takes effect. Finally, we measure the raw overheads of invoking statistical watchpoints. We only
measure watchpoints because both assertion and watchpoint share the same reduction engine, but a watchpoint would possibly be triggered more often than an assertion in production mode. This experiment explores the impacts of overheads in computing the statistics.

6.1 Using history variables

The execution starts with a backend server process and transfers runtime data to the frontend client. Collecting historical records from a runtime variable for a history variable should take constant time. Only when the minimum number of records retrieved, the frontend client will compute the statistics sequentially, and this time linearly increases as the number of records increases. However, since the maximum number of records is given, this time is also bounded by a certain constant. We measure the times (Table 1) to compute some standard statistics on history buffers, with sizes up to 32MBs (over 8-million integer records). It shows that using history variables is relatively inexpensive.

<table>
<thead>
<tr>
<th>Size (MB)</th>
<th>Max</th>
<th>Min</th>
<th>Mean</th>
<th>Sum</th>
<th>Stdev</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.04</td>
<td>0.03</td>
<td>0.03</td>
<td>0.04</td>
<td>0.04</td>
</tr>
<tr>
<td>16</td>
<td>0.72</td>
<td>0.71</td>
<td>0.71</td>
<td>0.73</td>
<td>0.71</td>
</tr>
<tr>
<td>32</td>
<td>1.44</td>
<td>1.43</td>
<td>1.44</td>
<td>1.47</td>
<td>1.48</td>
</tr>
</tbody>
</table>

Table 1: Computing standard statistics (in seconds)

6.2 Strong scaling experiment

Both watchpoint and assertion modes share the same underlying comparison mechanism. Evaluating a statistical condition that involves reduction of large datasets consists of a few phases. First, the backend servers independently obtain data from GDB and perform the requested statistical operation before transferring the results to the frontend client. We call data-reduction time plus data-transfer time the server time. Second, the frontend sequentially performs both the final reduction and the statistical tests. This is the client time. The entire processing time is labelled as the overall time.

We measure the performance of a statistical procedure that involves creating histograms from a 6GB global data structure. This could be an assertion or a watchpoint where histograms of runtime data are compared. We chose histogram because the procedure involves the $\chi^2$ goodness of fit test. The performance data is collected on a Cray XE6 system with 20,000 cores. The results are also used to compute the speedup. Because the performance data is only collected for 64 processes and more, we assume perfect speedup for the tool when running with less than 64 processes.

Figure 5 presents the elapsed time for executing the statistical procedure on a log scale. The results show that the overall assertion time is dominated by the server time. As the number of cores increases, this measure falls as expected. Note that the client time grows as the number of cores increases, because of the cost of constructing two histograms and comparing them sequentially at the frontend. This is the performance bottleneck for our current implementation. We notice that the client time becomes larger than the server time at around 5,000 cores. This results in the overall time levelling out instead of reducing further, even with more cores added. Consequently, we receive poor speedup after
8000 cores, as shown in Figure 6. The overall histogram-construction procedure can be further tuned to improve the speedup. Nevertheless, the tool achieves a good speedup overall (Figure 6), and more importantly, reduces the statistical reduction time down to an order of seconds, indicating its feasibility as a runtime code-verification aid.

### 6.3 Overhead for statistical watchpoint

We measure the overall overhead of using a statistical watchpoint at runtime by running a simple MPI code that invokes 20K processes and generates a 6GB global-structure. Without watchpoint, the execution takes 103 seconds. With a watchpoint added, the total execution time is 107.9 seconds, equivalence to <5% overhead (4.9 seconds latency). This latency is mainly the cost of waiting for the watchpoint to trigger on all 20K processes.

Note that a watchpoint is triggered when the target variable is updated; thus we expect the overall execution to slow down proportionally to the number of updates. We capture the latency times produced with increasing number of updates (Table 2). Clearly, the use of watchpoint should be avoided when the variable is updated continuously (e.g. in a loop). We advise users to use both watchpoints and assertions to reduce redundant processing time of runtime data.

### 7 Discussion

While using assertions for debugging and code verification purposes is not new, this technique has not been widely used for parallel programming due to the following queries. First, which program attributes (e.g. the variables) should be asserted since they are normally decomposed across a number of threads and/or processes? Second, where to place assertions in the code in which synchronization could be an issue? Finally, how assertion violations can be interpreted and the knowledge can be used to identify the implementation error? These queries are even harder to address when dealing with large parallel codes that implement complex systems such as large-scale scientific programs. The proposed statistical framework relieves the programmers from some of these challenges. First, data decomposition can be ignored because the assertions focus on the aggregated, statistical attributes of the programs, not the exact data values themselves. Second, statistical assertions should be placed where key data structures got completely populated; thus contains values that potentially follow expected pattern or data distribution. At these places, threads or processes should have been synchronized. However, if synchronization is neglected, apply statistics on such data structure would quickly reveal the problem. We admit that violation of statistical assertions cannot directly be used to locate the coding error. However, such knowledge could provide much insights into what is wrong with the computation; thus narrow down the scope of the problem. Conventional debugging activities can follow to locate the coding error.

### 8 Related work

Verifying and validating (V&V) a computer simulation is a challenging topic. Many techniques have been proposed to address different aspects of the process. For example, analytical solutions to mathematical equations are often used to approximate the error in a scientific program. However, analytical solutions are often difficult to compute when the model becomes more complex. A well-known area in V&V is about testing assumptions including structural assumptions and data assumptions [10]. In particular, data assumptions techniques use data to build a conceptual model and statistical tests
such as t-tests are used to validate the assumed statistical model. Our statistical technique could be categorised as a data-assumption technique. However, while the models are built using post-mortem data or data collected from many repeated runs, our solution targets the runtime verification process by testing statistical assumptions. We discuss some relevant verification techniques below.

**Face validity** [10] utilizes expert’s knowledge to decide whether a simulation behaves reasonably and makes subjective judgments on whether the outcome of the simulation is sufficiently accurate. Judgements can be given through either the animation generated by the simulation over time or the graphical representation of the output data. To verify stochastic simulations, a technique called **internal validity** [10] that involves comparing the results of several independent runs using different random seeds, can be used. If the random seeds cause inconsistent behaviors between different runs, the model is questioned and the code is further examined. Another technique utilizes different implementations of the same conceptual model and compare their results to verify the accuracy of a new implementation. This is the model-to-model comparison technique (or back-to-back testing) [11]. Differences found help revealing problems with the new implementation of the model. A similar technique, called code-to-code comparison [11], is useful for consistency tests such as nightly build tests.

Recent studies have focused on evaluating assertions in parallel programs. Siegel et al. introduce collective assertion that asserts the global state of the parallel program [12]. Chi-Neng Wen et al. [13] and Daniel Schwartz-Narbonne et al. [14] in separate efforts describe a programming environment in which assertions can be declared and evaluated as a parallel program is executed. In particular, Wen et al. develop a non-intrusive runtime assertion called RunAssert which allows developers to detect potential race condition issues and verify acceptable sequence of instructions. Schwartz-Narbonne et al. introduce the semantics and present a prototype implementation (PAssert) that supports the declaration and execution of program assertions in parallel.

9 Conclusion and future works

Apart from producing enormous multi-dimensional data structures, typical scientific codes also devise models for natural phenomena that are governed by known physical laws. These laws are often realized by solving PDEs, or are estimated through probabilistic methods such as Monte Carlo methods. Often, such processes can be verified using statistics and probabilistic models; thus motivates the use of statistical techniques to improve the code verification process. We have demonstrated a statistical framework that can retrieve and process runtime data to verify expected distribution of qualities as predicted by the physical laws. Especially, the approach can detect several real bugs that were known in two production simulation programs. Moreover, deriving ad-hoc statistics from large runtime datasets can be accelerated using the underlying parallel system, and we demonstrate this on up to 20,000 cores.

We identify several areas for future enhancement. First, the sequential reduction by the client process limits the scalability as indicated in Section 6.2. Future work can reduce the sequential workload required at the frontend client. Second, our current implementation only supports a set of known distributions such as Gaussian, Poisson, Cauchy, and Maxwell-Boltzman etc. We plan to enhance the support for user-defined functions to allow user-define ad-hoc distributions functions. Finally, our technique requires the scientific code developers to be familiar with statistics. We plan to develop a statistics discovery process which allows automatic retrieval and examination of runtime data in order to find abnormal activities during the execution. Enabling statistical watchpoints is the very first step towards this goal.
Reference


