Sirius: Probabilistic data assertions for detecting silent data corruption in parallel programs

Tara E Thomas* Anmol J Bhattad* Subrata Mitra Saurabh Bagchi
Department of Electrical and Computer Engineering
Purdue University
{thoma579, bhattad, mitra4, sbagchi}@purdue.edu

***Abstract*—The size and complexity of supercomputing clusters are rapidly increasing to cater to the needs of complex scientific applications. At the same time, the feature size and operating voltage level of the internal components are decreasing. This dual trend makes these machines extremely vulnerable to soft errors or random bit flips. For complex parallel applications, these soft errors can lead to silent data corruption which could lead to large inaccuracies in the final computational results. Hence, it is important to determine the presence and severity of such errors early on, so that proper counter measures can be taken. In this paper, we introduce a tool called Sirius, which can accurately identify silent data corruptions based on the simple insight that there exist spatial and temporal locality within most variables in such programs. Spatial locality means that values of the variable at nodes that are close by in a network sense, are also close numerically. Similarly, temporal locality means that the values change slowly and in a continuous manner with time. Sirius uses neural networks to learn such locality patterns, separately for each critical variable, and produces probabilistic assertions which can be embedded in the code of the parallel program to detect silent data corruptions. We have implemented this technique on parallel benchmark programs - LULESH and CoMD. Our evaluations show that Sirius can detect silent errors in the code with much higher accuracy compared to previously proposed methods.Sirius detected 98% of the silent data corruptions with a false positive rate of less than 0.02 as compared to the false positive rate 0.06 incurred by the state of the art acceleration based prediction (ABP) based technique.**

I. Introduction

Supercomputing clusters are becoming larger and more complex in terms of the numbers and variety of internal components. At the same time, the feature size of transistors is also shrinking to keep up with Moore's Law. To reduce the overall energy consumption of these large systems with sub-components containing billions of transistors, the operating voltage has been significantly lowered.

As a consequence, studies have shown [14] that the frequency of *soft errors* in these kinds of large and complex systems has also increased. Soft error is defined as a transient error event that is typically not predictable, *e.g.,* it may be a flip in a memory element due to the electric charge being changed, say due to a charged particle hitting the memory chip. Other examples are noise on a data bus and a flip-flop latch not being activated due to a delayed clock signal. These errors are caused due to random bit-flips in different hardware components. Such bit-flips are caused because of various reasons *e.g.,* radioactive decay of chip packaging [25] materials resulting in radiation, hardware components hit by extraterrestrial alpha particles at high altitude facilities [5], noise interference with data on the bus especially during low power operations [9] etc.

The most common form of SDC is bit-flips in memory subsystem, cache and registers. Moreover, the probability of such errors increase with increase in the complexity of design of systems, higher clock frequencies, and low operational voltages [24].

Soft errors are a major concern in memory elements as these errors exponentially increased with the size of the memory chips. For example, advanced processors with large embedded SRAMs tend to have failure rates of more than 50000 FIT per chip [1] [5]. However, the frequency of soft errors in hardware combinational logic within the microprocessors is also on the rise([27]. It is predicted that the soft error rate in microprocessors will grow in direct proportion to the number of devices added to it in each succeeding generation [29].Current day's processors with billions of transistors can have 5.7×10^5 to 5.7×10^8 unprotected latches in them prone to SDCs.

Bronevetsky *et al.*. [10] found that on the occurence of a single bit flip in a single run, iterative methods converge with a final error greater than 10% larger than the fault-free error. The majority of these errors remain undetected until a domain expert analyses the generated output. This is an example of one of the serious consequences of soft errors - *silent data corruption*. Silent data corruption occur when the data gets corrupted or altered, but does not lead to a direct, easily perceptible failure such as crash or slowdown. Thus, often, results or data gathered from large scale and long running scientific simulations or mathematical solutions become useless due to silent data corruptions. Evidently, this situation not only leads to monetary loss due to wasted energy and compute time, but also delays the whole process as users have to manually inspect and validate the results to guard against any discrepancies. It is important to note that not just the scientific applications, but also large scale data analytics can equally be affected by silent data corruptions triggered by soft errors. Since, often these data analytics frameworks involve some sort of pipeline architecture, data corruption in

* Authors contributed equally to the work

[1] One FIT is one error in a billion device hours

one component may cause serious data integrity issues due to cascading effect [10]. Hence, it is crucial to have proper detection techniques in place to guard against such silent data corruption.

There are efficient techniques to detect and correct data corruptions in main memory like TMR memory redundancy [23] and several error correcting codes [12], [28]. However, there are no viable solutions to detect silent data corruptions triggered by soft errors in the CPU registers, Arithmetic Logic Unit (ALU) and cache. Few techniques that are used to prevent silent data corruption include providing redundancy to make it likely that transient errors are eliminated [19]. These methods have been used for long and do handle many transient errors. However, they have high time overhead (temporal redundancy) or resource overhead (spatial redundancy). Some other approaches compare the result of the main computation with that computed using a cheaper mechanism [7]. But such approaches have to be carefully tailored to each specific application.

In this paper we introduce a tool called Sirius, which leverages the stencil based computation patterns in parallel programs to detect and notify the user about the possible silent data corruptions. At a high level, the technique used by Sirius is based on the observation that if the computation is parallelized by dividing the overall solution domain into a two or three dimensional grid of sub-domains, then the variables belonging to neighboring sub-domains would have similar range of values. For example, a parallel weather simulation application is very likely to have some kind of grid or mesh structure dividing the overall area in to small regions. The calculations corresponding to these smaller regions are likely to be performed on adjacent cores to minimize the communication overheads. From a physical standpoint, the values of weather related variables for one region is likely to be not drastically different from that of its neighboring regions. Putting these two points together, the intuition is that the value of such "spatial variables", representing the weather pattern in this example, at a particular node is likely to be close to that at its neighboring nodes. More generally, the range of values of the variable at a given node can be inferred from the values of the variable at neighboring nodes. Now, typically such simulation is done over a certain period of time through iteration of a *time-step* loop [21]. In that case, the current values of these weather related variables should also be similar to the values calculated in the recent past i.e., the previous iterations of the time-step loop. Figure 1 represents this idea of a gradation in the values over space as in [1]. The values represent the temperature variation of over the US on 24^{th} April 2016, and it is evident that there is a gradation in the values spatially. *i.e.,* the variable value at a point is related to the variable values at points around it. Similarly, the weather at a place would not vary instantaneously but would rather vary with some kind of gradient over time.

To leverage these two insights, Sirius instruments the application code to collect values of such spatio-temporal variables. It turns out that in a class of parallel applications, the critical variables are such variables. This class of applications includes simulations of physical phenomena like molecular dynamics, weather prediction, population growth prediction, etc. Few examples of such applications are clover leaf [2], CoMD [3], LULESH [20] etc.. Sirius then builds a neural network based model, one for each such spatio-temporal variable. The model takes as features the values of the variable at neighboring nodes (6 neighbors in a 2D mesh) and the values of the variable at that particular node over previous k time instants (*e.g.,* previous k loop iterations).Then, Sirius inserts assertion statements in the application to check if the actual observed value of those variables falls within a bounded range around the predicted value. When these assertions fail, Sirius notifies the user about a possible data corruption.

This paper makes the following contributions:

- We introduce a novel technique for detecting silent data corruptions using the temporal and spatial locality of physical variables in parallel applications.
- We introduce a systematic approach for choosing the variables most effective for detecting silent data corruptions.
- We developed a neural networks-based tool (Sirius) that automatically generates probabilistic assertions for detecting silent data corruptions.
- We show effectiveness of Sirius using case studies with two widely used parallel applications: LULESH and CoMD. We also quantitatively show that Sirius is much more accurate compared to current state of the art techniques like acceleration-based predictor (ABP)[8] using the first and second derivative in time and the previous value of the data.

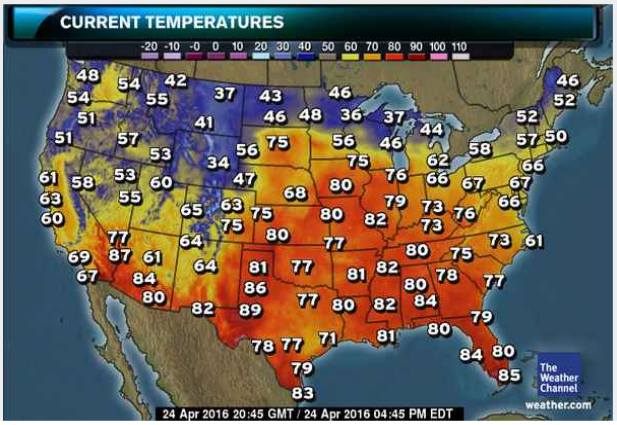


Fig. 1. Gradation in values of physical variable - temperature spatially

Figure 2 gives the overall workflow of Sirius. Step 1 involves instrumenting the application code to obtain the values of selected variables. This is followed by separating out the obtained data into disjoint training, validation and testing data sets and generating feature vectors from this data. The training data set is then fed into a neural network to obtain a model, which is then validated. Based on the error model specified, negative test samples are also generated and this along with

the positive samples are used to test the model and obtain the assertion. The detailed explanation of the workflow will be presented in Section III.

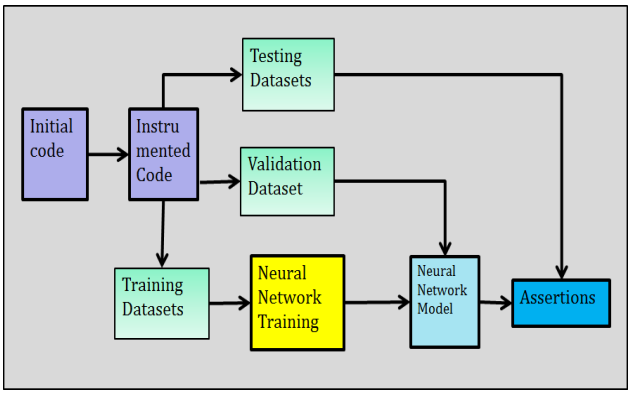


Fig. 2. Workflow of Sirius

The rest of the paper is organized as follows: Section II provides the needed background to effectively understand the paper and Section III describes our solution approach in detail. The implementation of the technique is explained in Section IV, while the evaluation and results are presented in Section V. Section VI has literature review of work done in related areas, and we conclude the paper in Section VII.

II. Background

A. Parallel Programs

Parallel programs consist of multiple threads or processes that execute simultaneously on multiple cores or nodes to speed up the overall execution. The most common modes of interaction between these different cores or nodes are through message passing (usually using MPI paradigm) or through shared memory (using threads). When a large amount of data is to be processed simultaneously, it is a common practice to divide the data into almost equal-sized sub-regions and allocate each such sub-region to different nodes or cores. A large fraction of parallel applications focus on problems related to physical sciences or engineering and frequently solve ordinary differential equations or partial differential equations in multiple variables [13]. These problems are good candidates for parallelism because, in all the sub-regions same physical science equations are used to compute values for variables like pressure, internal energy, velocity, volume etc. Thus, most of these scientific applications use approaches similar to stencil based computation where the data is divided over a 2-D or 3-D mesh before being distributed to multiple nodes for parallel computation using MPI [17]. Frequently the data local to a particular node is updated (using message passing) based on the values of the variables calculated by the nodes handling a physically adjacent sub-region. This pattern of calculation followed by communication to update the data based on results from other nodes continue for several logical time steps either specified through a parameter set by the user (in case of simulation type applications) or until the residual error of the calculation goes below a specified threshold (for iterative solver type applications). Usually to minimize communication overhead, the original data is distributed in such a way such that neighboring nodes operate on the data that are physically adjacent to each other. In the following sections we provide an overview of two representative parallel applications, one performs molecular dynamics simulation another solves a complex equation related to hydrodynamics. In the later sections, we use these applications to show the effectiveness of Sirius in detecting silent errors. It also provides some necessary background on some of the techniques used by Sirius.

B. CoMD

CoMD [3] is a proxy application for a broad class of Molecular Dynamics (MD) simulations. MD simulations in general deal with evaluating the forces between atoms and molecules of a substance to predict the locations, and hence to understand the particle-level behavior of the substance better. CoMD in particular looks at substances where the inter-atomic potentials are within a short range, like those in uncharged metallic materials. So, the simulation needs to evaluate all the forces within the cutoff distance *i.e.,* the distance beyond which van der Waals interaction energy between two atoms is zero of an atom.

CoMD has a structured mesh and utilizes a Cartesian spatial decomposition of atoms across nodes. The nodes execute code in parallel and thus can be either a core or a separate machine. Each node computes various physical forces and updates the positions of all atoms within its domain boundaries. As position of atoms change, and when they shift across domain boundaries, they are handed off from one node to the next node. The cutoff distance for the inter-atomic potentials in CoMD is much smaller than the characteristic size of the spatial domain of the node. To ensure that the atoms pairs which are within the cutoff distance are identified efficiently, the atoms on a node are assigned to cubic link cells which are slightly larger than the cutoff distance. So, an atom in a particular link cell has to only test all the atoms in the same link cell and its neighboring link cells to find the possible interacting atoms. The number of nodes, as well as the number of link cells can be specified by the user.

The application uses a *leap-frog* algorithm to advance the time steps through the simulation. The forces, velocity and positions of the atoms in each link cell are updated in each iteration, and physically moved atoms are handed over to other link cells. In the C-MPI based implementation of CoMD, the relevant variables are passed between different nodes using C-MPI messages.

C. LULESH

LULESH [20] is a C-MPI based benchmark code that solves a *Sedov Blast problem*. It represents a typical hydrocode that approximates the hydrodynamics equations by partitioning the equations spatially. Even though LULESH is specific about the problem it solves, it is a representative of the numerical

algorithms, the programming style and the data transfers in typical C-MPI codes.

It is based on an unstructured hexahedral mesh and splits the variables spatially for volumetric elements based on the mesh. A node on a mesh is a point where mesh lines intersect. The whole region of interest is divided symmetrically into multiple domains, totaling the number of processors used. In the default implementation, each domain is subdivided into p^3 volumetric elements, where p is the problem size specified by the user. The algorithm configures the mesh structure, defines the boundaries and sets initial values to the physical variables. It then evolves by integrating the equations in time. Each domain is processed in parallel and required variable values are communicated using C-MPI messages after each iteration.

D. Assertions

Assertions or program invariants are statements containing expressions that are expected to be true during the instant of execution of the statement. Typically, they are used to ensure that a program is functioning as expected. For instance, suppose we expect the value of a particular variable to be in the range (x, y) at some part of a program, we could write an assertion there, to actually verify that it indeed falls in the range, as we want. In case due to some unexpected error it does not, then the assert statement will be false and it will be flagged for in-depth analysis.

There has been significant work done in automatically inferring program invariants dynamically, in sequential programs and using them to detect errors. For example, [16] introduces a tool named Diakon, which uses inductive logic programming on the execution traces to detect invariants and get assertions. DIDUCE[18] deducts invariants obeyed by sequential programs dynamically and use them to detect issues in the code.

Traditionally, assertions are such that they are always true if there are no errors in the program and hence, are used to verify deterministic properties of a system, which always hold true in correct program executions. However, there are many scenarios where a certain property many not be *always* true but *very likely* to be true, during correct executions. This means that in these scenarios, the statement being false does not necessarily imply an error in the program. Since the probability of the property becoming false when there are no errors in the program is small, it is worthwhile to investigate when it becomes false. Assertions written to check such properties are called probabilistic assertions because, instead of giving a deterministic guarantee, it calculates a probability that the assertion failing is actually due to error in the program code or data.

III. Design and Approach

A. Principle of locality in parallel programs

Many parallel programs deal with variable values spread over a region of space and over a duration of time. In these programs, it is a common practice to use different nodes to handle variable for different locations. Benchmark programs like Cloverleaf [2] and LAMPPS in Sequoia benchmark [4] are some examples that follow this kind of coding practice. For *e.g.,* if a program deals with the variation of temperature over a large region, each core would do the computation for the value of our parameter of interest, *i.e.,* the temperature, for different spatial locations in our region of interest. This approach is prevalent because calculations and manipulations using the variables for one location at a given timestep can be done independently in parallel to the calculations of the parameters for other spatial locations at that timestep. These parallel processes, very often follow synchronous communication, *i.e.,*, in each iteration after having progressed to some stage (typically after all the calculations corresponding to a particular timestep), they communicate few of these parameters using some protocol, like MPI. For ease of communication between the different cores, and for ensuring a systematic implementation, most such programs use neighboring cores to compute the variable or parameter values for neighboring spatial locations.

Sirius makes use of this feature in identifying data errors in variable values. In programs that deal with physical quantities like pressure, position, temperature etc, we intuitively expect there to be some kind of gradient or pattern among these variables along spatial co-ordinates. For *e.g.,* if nearby nodes are computing the temperature values of geographically nearby regions, we expect the values computed by the nodes to fall in a small range. We call this spatial locality. Also, if a program calculates values of physical quantities over steps of time, then we expect the variation to be gradual. We call this property temporal locality. Sirius leverages both temporal and spatial locality to identify possible silent data corruptions in parallel programs.

B. Error Model

Soft errors, leading to silent data corruption as explained in Section I involve a change in the value of some bit of a datum used in the program. Hence, Sirius is designed to detect such kind of errors in parallel programs. The variables used to represent physical variable values in most scientific applications are of floating point data type. It is evident that the impact of soft errors occurring in the lower order bits of the variables will be very minimal to the output of the program, since it would result in a negligible change in the value. However, soft errors that cause changes in the higher order bits affect the output significantly. Sirius is designed such that the soft errors in the higher order bits are caught and flagged with a much higher probability than the errors which occur in the lower order bits, as desired.

C. Overall Technique

For any parallel application, Sirius provides an API `InstrumentCode` that can help instrument the code once the user selects the variables to apply the technique for. The API takes the application file path, the function names and the names of the variables the user has selected as the input. It modifies the file by inserting code to dump out the values taken

by the selected variables in the specified functions during run time. `InstrumentCode` also adds code that ensures that the details about the spatial location and time-step iteration numbers corresponding to the variable values are also kept track of, and will be dumped at run time. This helps in generating features based on locality. The output of the API is the instrumented code. Currently the API for instrumenting the code have been developed only for C-MPI based applications, but can be easily extended to other applications. Details on how to select variables for any generic application will be mentioned in III-D. Once the application is instrumented, it is run with multiple valid inputs, so as to dump out the variables of interest. Sirius now makes use of the inherent spatial and temporal locality in the values of the dumped variables to generate probabilistic data assertions, which can serve as indicators of silent data errors by identifying anomalous variable values. Sirius generates locality based features out of this data. A part of this feature set is used to train and validate a neural network. The remaining part of the feature set apart is to test the performance of the neural network and decide the threshold. Detailed descriptions of how these are done are given in III-E and III-F. Assertions are generated based on these thresholds and inserted in the code.

D. Selection of Variables

The user is expected to have a broad idea of what the application does, and hence know which the relevant variables are, and whether they are likely to follow the principle of locality as described in Section III-A. The second criterion is almost always satisfied while dealing with physical variables, but the user's intuitive idea of the extent of locality expected in each variable (*e.g.*,, position variables are in general expected to show more locality than the corresponding velocity variables) can also be used while making the variable selection. It is recommended that the variables selected are either output variables or variables that have a significant impact on the program outputs. Because the variable values from one iteration of the program are usually used in the next iteration, the errors get cascaded. Hence, any data corruptions in their values, will result in the deterioration of the reliability of the program output. Also, if the variables chosen are dependent on the values of several other variables from the present and/or previous iterations of the program, any data corruptions in such variables will perturb the values of our chosen variables. Thus, through proper selection of variables to apply the technique on, it is possible to indirectly identify silent data corruptions in other variables too.

The values taken by these chosen variables are dumped by Sirius during run-time. This is achieved by instrumenting the source code. Sirius stores both the spatial coordinates and time-step iteration numbers while dumping these variable values. This helps in generating features based on locality.

E. Feature Generation

Sirius provides an API `GetFeatures` which takes the dumped data files as the input and generates files with feature sets for the variable values. For each variable calculated at each point in space, the default feature set created by Sirius comprises of the values of the same variable at all the adjacent points as well as the value of the variable at that point, in the previous time instant $t-1$. For instance, in a three-dimensional space, each point (x, y, z) has 6 neighbors i.e, $(x-1, y, z)$, $(x+1, y, z)$, $(x, y-1, z)$, $(x, y+1, z)$, $(x, y, z-1)$ and $(x, y, z+1)$, as shown in figure 3. For a point which lies on the boundary of the processor's space, one or more of the above mentioned adjacent point are present in a neighboring processor's space and uses inter processor communication to exchange such values and produce the complete feature set for each of the chosen variables. By default, Sirius assumes that adjacent cores correspond to neighboring nodes in simulation. However, this might not always be true. To tackle this issue, there is a provision in the API where the user can specify the pattern in which the simulation nodes are distributed on the cores. For exploiting temporal locality in the variable, at time step t we take the value of the chosen variable in the previous iteration or time-step $t-1$ also as a feature.

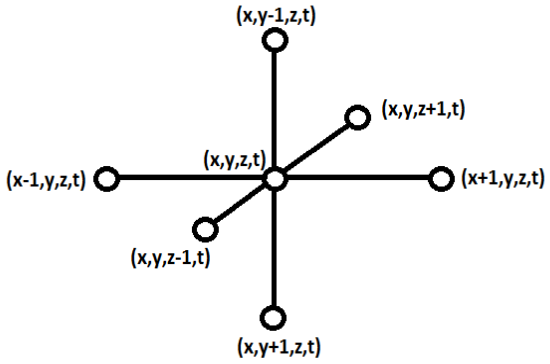


Fig. 3. The neighbors of an element at a unit distance

Other possible features are any program parameters or user inputs that influence the spatial or temporal distribution of the variables. For example, if we have a parameter corresponding to the time-step size, it is intuitive to believe that a run with a larger value of the time-step would result in greater variation in the value of variables in consecutive iterations, when compared to a run of the same program with a smaller time-step value. In case the user is aware of any such feature that is present in the program, Sirius also gives them a provision to add that to the feature list.

With the insights from the discussion above, Sirius uses the dumped values to generate a feature set for each of the chosen variables. We get one sample feature per variable, per iteration, for every point in the space where the value of the variable is calculated.

F. Neural Nets

Once the feature sets for the data points are obtained for the desired variables, Sirius takes these features and

divides the data points into disjoint training, validation and testing data sets. Sirius uses neural networks to generate curve fitting models for the chosen variables by training using the above mentioned training data set. We obtain a model which predicts the variable value, given it's feature set. This model is further validated using the validation data set. Neural networks provide a good approximation of the relationships between the data and corresponding features when the model of the system is a black box. Hence we chose this algorithm while developing Sirius, making Sirius essentially application agnostic. More information on how we used neural networks this will be provided in Section IV-D.

G. Prediction and Assertion Generation

Sirius has an API `InjectErr` for the user to specify the error model he/she wants. It takes the dumped data file with the test data, bit range for error injection, and the number of errors to be injected per data as the input, and provides a file with the error-injected data samples as the output. This is done by randomly choosing one of the bits (in the bit range) of the variables and complementing it. This feature is currently implemented for double precision floating point variables, but can be extended to other data types too. By default, the error model is a single bit error in the 8 higher order mantissa bits and the exponent bits. The user can change this as per his/her discretion. In short, `InjectErr` generates negative samples based on the error model specified, by altering the fraction of data kept apart for testing (test data set) that was dumped while running the instrumented code.

Sirius runs the positive and negative test samples through the neural net to get the predicted output, x' of the variable. This is compared with the actual value, x by calculating the difference between actual and predicted values, which is the relative error, $e = \frac{|x-x'|}{x+\epsilon}$, where ϵ is a very small number to ensure numerical stability.

The relative error e is then thresholded to different values to get the corresponding true positive rate (TPR) and false positive rate (FPR). This is then used to plot an ROC curve with TPR on the y-axis and FPR on the x-axis. Based on the ROC curve obtained, the user can choose the allowed deviation, or the relative error threshold to be used in the assertion. If the difference between the value obtained while running the application differs from the value predicted by the neural net by an amount greater than this chosen relative error threshold, Sirius considers it significant and flags a probable silent data error. Then Sirius extracts the neural net parameters of the trained models to get the mathematical expressions for generating probabilistic assertions. Then, assertions of the form $abs(observed\_value - predicted\_value) < allowed\_deviation$ are generated from the output of the neural nets. The allowable prediction error threshold $allowed\_deviation$ to be used in the assertion is decided based on the ROC curves we obtain as explained above.
These assertions are inserted in the source code at appropriate places to catch silent errors. An example of such an assertion which we inserted as part of our evaluations will be shown in Section IV. Though the training and thresholding process may take time, this is a one time process per application, and once the assertions are embedded in the application, error detection does not have any significant overhead. This means that computing the output of the neural net for a new run which might have different feature values is not computationally intensive and can be easily accommodated during run-time.

IV. Evaluation

A. Software and Machine Specifications

We run Sirius and the parallel benchmark applications on HP ProLiant DL60 Gen9 systems with 40 cores and 256Gb of RAM. We use MATLAB R2015a to run the Neural Network training for Sirius.

B. Experiment 1: LULESH

We identified LULESH [20] which is essentially a computer simulation that involves solving the hydrodynamics equations as a potential benchmark to evaluate Sirius because it is representative of codes for a wide variety of parallel applications, having both the spatial and temporal locality. The LULESH algorithm computes the forces at each node to calculate the accelerations, and then integrates it to find the velocity and position values at the node. The value of position at nodes along x, y and z directions (variables called as $domain.x$, $domain.y$ and $domain.z$) and the volume of each volumetric element (variable called as $determ$) were identified as important outputs of the code and we chose these variables to generate the assertions.

We then used the default version of the `InstrumentCode` API to instrument as explained in Section III-C the code to dump the above mentioned variables for all processors. The LULESH-MPI default code was run for problem sizes 3, 4, 5, 6 and 7 using 8 parallel processors to dump the values of the chosen variables along with the nodal coordinates and iteration numbers. Then we use the default `GetFeatures` to generate features using the variable values at the nearby nodes and the variable value at the same node during the previous time-step for all the above runs, as discussed in Section III-E.

C. Experiment 2: CoMD

The MPI version implementation of CoMD 1.1 [3] was chosen as the second benchmark to evaluate our technique because it uses structured mesh which is also a common approach to implementing parallel programs. As mentioned in Section II-B, CoMD is a proxy app that simulates molecular dynamics. The code works by iteratively updating the particle positions, along with computing forces and redistributing atoms among ranks when their positions change, as the simulation progresses. We ran the reference problem in CoMD which is is solid Copper starting from a face-centered cubic (FCC) lattice. The initial thermodynamic conditions were set at the default values of $600K$ and $lat = 3.615Angstroms$.The simulation was done on 8 cores, with the total number of atoms

= $nAtoms = 4*nx*ny*nz$, where $nz = ny = nx = 80$. We identified the positions of the atoms along x, y and z directions *i.e.,* $r0$, $r1$ and $r2$ respectively as the relevant variables for evaluating our technique. The code was instrumented and the relevant values were dumped, and features generated.

D. Procedure

Once the features are obtained as explained above, Sirius merges the data from the runs for different problem sizes and build the data set for each of the above mentioned variables. There are 846248 and 1229312 samples for each of the variables in the global data set for LULESH and CoMD respectively. This set is divided into training, validation and testing data. Sirius separates out 70% of the samples for training, 15% for validation and remaining 15% of the samples for testing. Then, Sirius trains a neural network with 2 layers in MATLAB with the training and validation data. Unless specified otherwise, all experiments used a simple 2-layer NN with number of neurons in the hidden layer $nneuron$ as 10. To test the robustness of the neural net, we generate negative samples using `InjectErr` to mimic bit errors in floating point. Negative test data set is generated by inverting one random bit in each sample of the testing data set. The variables considered are of double-precision floating point data type. In all experiments, other than the ones explicitly looking at the performance of Sirius with different error models, we perturb only the most significant 8 bits of the mantissa (significand) and all the bits of the exponent. More information on error injection is given in Section IV-E. The size of the negative test samples is same as the positive test samples.

We use Sirius to find the relative errors and plot the ROC curves as explained in Section III-G. The mathematical expression extracted from the NN model which was trained along with the chosen relative error threshold based on the ROC curve is used to obtain the assertion. These assertions are then inserted to the application code. An example of this is given in Figure 4.

We also do a statistical t-test to test how good the assertion generated is, by evaluating the significance of the difference between predicted value and the actual value of the variables of our concern in the correct as well as the error injected test data. These values help the user in assessing how strong the assertion is, and the confidence level to which the user can identify an error flagged by the assertion as a real silent data corruption.

E. Error Injection

Soft errors can be modeled as one or more bit errors. Since the probability of occurrence of more than one soft error in a data item at a time is considerably less, for evaluating our model, in most experiments, we consider only single bit errors. Because multiple bit errors change the data value more, it is also obvious that our technique will perform better in that case, so to lower bound our performance, we limit the error model in most experiments to only 1 bit error at a time. As mentioned in Section III-B double precision floating point data type is widely used to record physical variable values. This is true in the case of the benchmarks we evaluate the technique on too. Hence our current implementation is for this type of values. Also, as previously explained, to ensure that Sirius considers soft errors that will actually have a reasonable impact on the program outputs, in the 64 bit implementation of float in the program, we inject errors into bits from 44 to 63, out of which 44-51 are the higher mantissa bits, 52-62 are the exponent bits and 63 is the sign bit. We also perform experiments with errors injected only in the exponent and sign bit, and also observe the performance of Sirius with multiple bit errors injected.

V. Results

To study the performance of our scheme we evaluate the performance of our models over the positive and negative test samples. As mentioned in Section IV, we calculate the relative error e by comparing the actual values and the values predicted by the neural net model. These errors are then thresholded at different values and we find the true positive and false positive rates and plot the ROC curves.

We use a two layer neural net curve fitting model, with one hidden layer and one output layer. By default, the number of neurons in the hidden layer of the neural network $nNeuron$ is set to 10. Figure 5(a) show the performance over the four chosen variables in LULESH, $domain.x$, $domain.y$, $domain.z$, and $determ$. Figure V shows the comparison of the ROCs for the selected four variables, for $nNeuron = 10$. We see that the true positive rate at 0.15 false positive rate is $determ$ - 81.14%, $domain.x$- 96.82%, $domain.y$- 97.22%, $domain.z$-95.69%. We see that the accuracy is higher for position variables than the volume. This might be because of the intricacies involved in calculation of $determ$.

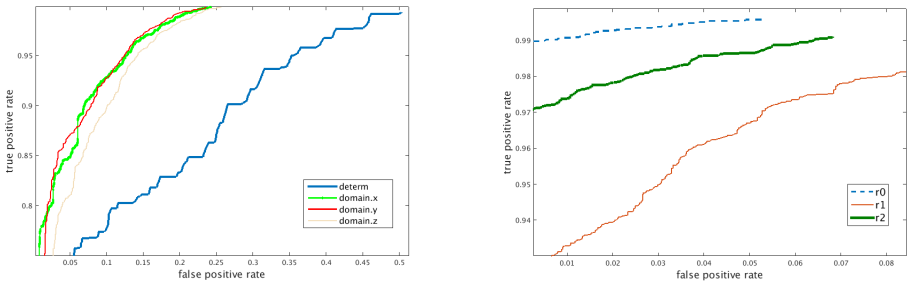
Figure 5(b) shows the performance for the position variables of CoMD explained in section IV-C. At 0.02 false positive rate, the true positive rate for $r0$ variable is 99.27%, $r1$ is 97.82% and $r2$ is 93.95% respectively. Overall, the performance of Sirius is better on CoMD than LULESH, because of more similarity among the neighboring atoms of CoMD and less variation after each time step.

Acceleration-based predictor (ABP) technique works best among the ones mentioned in other state of the art techniques [8]. This technique uses the first and second derivative in time and the previous value of the data to compute the expected value of the data. Figure 7 shows that Sirius outperforms ABP significantly for the variable $domain.y$ from LULESH. This is because ABP fails to exploit spatial locality and the variable we have considered fails to fit this rigid model.

To show the requirement of using a non-linear system like neural network than a simple liner regression, we replace the neural network with logistic regression using normal equations. Figure 6(a) and Figure 6(b) show the performance of this as compared to Sirius for the variable $domain.y$ from LULESH and $r0$ from CoMD. It is clear that using neural networks gives us significant improvements. By restricting the number of hidden layers to 1 and number of neurons as 10, we can minimize overheads.

```
out1[0]=-0.2516;
out1[1]=0.1721;
out1[2]=-1.4786;
out1[0]+=  0.1058*feat[0]   -0.4450*feat[1]   -0.0177*feat[2]   -0.0223*feat[3]   + 0.7221*feat[4]   -0.3293*feat[5]   -0.0466*feat[6];
out1[1]+= -0.6892*feat[0]   +0.4177*feat[1]   +0.0166*feat[2]   + 0.0195*feat[3]   -0.5471*feat[4]  +  0.2623*feat[5]  +  0.0391*feat[6];
out1[2]+= -1.1883*feat[0]   +3.1755*feat[1]   +0.0832*feat[2]   + 0.1358*feat[3]   -1.4057*feat[4]   -1.4057*feat[5]  +  0.2740*feat[6];
out=-0.0020;
out+= 10.7323*out1[0] +  13.0224*out1[1]   -0.4764*out1[2]  ;
if((out-Posx[i])/(Posx[i]+0.0000000001)>0.2 ||  (out-Posx[i])/(Posx[i]+0.0000000001)<-0.2){printf("Observed value differs from predicted value\n");}
```

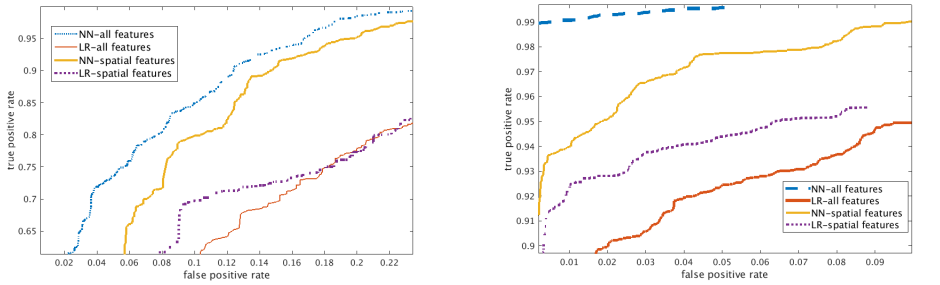
Fig. 4. Example of an assertion generated by Sirius



(a) ROC for the variables from LULESH

(b) ROC for the variables from CoMD

Fig. 5. Performance evaluation of Sirius



(a) Comparison of Sirius with logistic regression for the variable domain.y

(b) Comparison of Sirius with logistic regression for the variable r0

Fig. 6. Comparison of performance of Sirius with alternate techniques

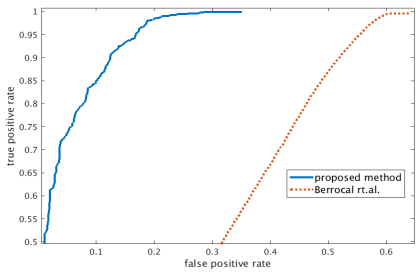


Fig. 7. Comparison of Sirius with technique in [8] for the variable domain.y

Another advantage of using neural networks is that it can learn both, the spatial and temporal features simultaneously. Building a system with both temporal and spatial features using neural network performs better than just spatial features as shown in Figure 6(a) and Figure 6(b). When the same experiment was repeated with logistic regression, using both spatial and temporal features doesn't outperform just spatial features, making it necessary to have a more complex non-linear learner like neural networks.

To study the impact of the performance of our technique with different number of neurons in the hidden layer, we ran experiments with 3, 5 and 10 neurons ($nNeuron = 3, 5, 10$). Our results for this experiment for the variable $domain.y$ in

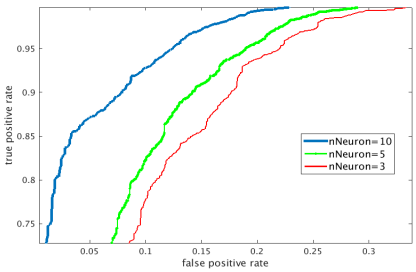


Fig. 8. Performance with different number of neurons for the variable domain.y(lulesh)

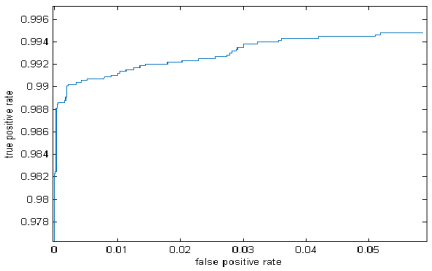
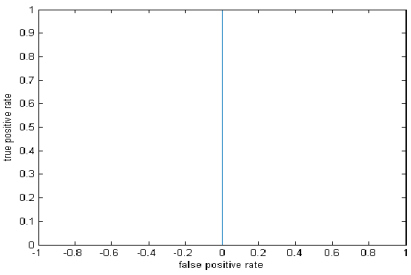
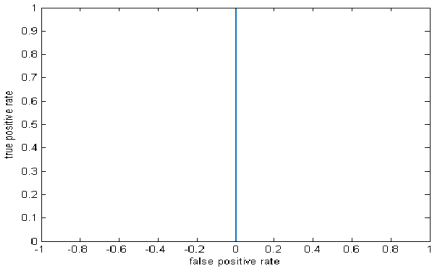


Fig. 10. Performance with 2 errors injected in the value of r0 in CoMD

LULESH is given in Figure 8. We see that the neural net with 10 hidden neurons outperforms the ones with 5 and 3. We observe that the difference between the true positive rates with different $nNeuron$ reduces with the increase in the false positive rate.



(a) Performance of Sirius with errors injected only in the exponent bits for the variable r0



(b) Performance of Sirius with errors injected only in the exponent bits for the variable domain.y

Fig. 9. Performance of Sirius with errors in the exponent and sign bit

To evaluate the performance of Sirius when errors occur at different bit positions, we also conduct experiments by varying the error model. Since the computations are affected much more when there is an error in the exponent or sign bits, we also use a different error model where the soft error is uniformly distributed among these bits only. The ROC curves for these experiments for the variables $r0$ in CoMD and $domain.y$ in lulesh are shown in Figure 9. It is clear that Sirius works very well for errors in this range with a false positive rate of 0 errors.

We also ran experiments with the error injected uniformly in 2 random bits from positions 44 to 63 at the same time to see how Sirius would perform. As expected, it was observed that the performance is better in this case. The ROC curve for the variable $r0$ in CoMD is plotted in Figure 10

We also perform the statistical t-test as explained in Section IV-D. It is seen that for all variables, the difference between the observed value and the predicted value in the true data set (without errors injected) and the error-injected data set was significant, and the P-Value is less than 0.00001, which indicates a very high confidence level in the generated assertions.

VI. Related Work

There has been significant work done on detecting crash and slow downs in parallel programs. The work done by Bronevetsky *et al.* [11] proposes a tool AutomaDeD which addresses these issues by using semi-Markov models to model control flow and timing behavior of applications. It then identifies the task that first manifested a bug, using clustering and then identifies the specific code region and execution. [22] have developed Prodometer, a loop-aware, progress-dependence analysis tool for parallel programs. It creates states in Markov model by intercepting MPI calls. These states represent code executed within and between MPI calls to be used for debugging. Blockwatch [30] uses similarity among tasks of a parallel program for runtime error detection.

Deterministic assertions and invariants have been traditionally used to ensure data integrity in serial programs. Diakon [16] identifies and implements techniques for dynamically inferring invariants. It obtains execution traces by running an updated source code obtained by instrumenting serial programs by source to source translation. It then uses Inductive Logic Programming on the traces to get assertions. Elkarablieh *et al.* [15] repairs a data structure which violates an assertion by performing a systematic search based on symbolic execution to repair the structure. Heuristics are used to minimize mutations and prune searches for better repairs.

To catch silent errors in ODEs and PDEs Benson *et al.* [7] uses values computed by a simple parallel solver and compare them with the values from the main solver. [8] presents another technique to catch silent data corruptions

at application level by training the linear predictors to catch anomalies during execution. The linear predictors are trained during the run-time. We have applied the approach described in this paper, and it is seen that Sirius performs better than this. The corresponding graph has been presented in V. A recent paper [6] uses an idea of bounding the range of the variables and evolving the bounds based on linear interpolation assuming smoothness of physical variables offers yet another approach to detecting silent data corruption in HPC programs. They consider the range of fields, based on variable values. So, though the false positives detected using this technique is low, the number of true silent data errors detected is also less, because it can't detect errors values that let the field values in the range. Implementation based on this approach in the benchmarks we considered seemed to detect only 15%-20% of the silent data corruptions introduced as per our fault model. The work by Sharma *et al.* [26] models silent data corruptions in the CPU operations and registers. They targets stencil computations to train a regression models for error detection. They suggest improvement in training procedure to reduce the amount of training data.

VII. Conclusion

We introduce Sirius, a novel tool using the concept of spatial and temporal locality in physical variables for generating probabilistic assertions in parallel programs to detect silent data corruption due to soft errors. We also provided suggestions on identifying variables in the parallel application on which the tool works best. We implemented the tool on two representative C-MPI based benchmarks and demonstrated the effectiveness of the technique. The tool is also evaluated by comparing the results to other possible techniques, and it is shown to work better than them.

References

[1] CloverLeaf Benchmark. https://weather.com/maps/ustemperaturemap.

[2] CloverLeaf Benchmark. https://mantevo.org/packages.php.

[3] CoMD Benchmark. http://www.exmatex.org/comd.html.

[4] Sequoia Benchmarks. https://asc.llnl.gov/sequoia/benchmarks/#lammps.

[5] R. C. Baumann. Radiation-induced soft errors in advanced semiconductor technologies. *Device and Materials Reliability, IEEE Transactions on*, 5(3):305–316, 2005.

[6] L. Bautista-Gomez and F. Cappello. Exploiting spatial smoothness in hpc applications to detect silent data corruption. In *High Performance Computing and Communications (HPCC), 2015 IEEE 7th International Symposium on Cyberspace Safety and Security (CSS), 2015 IEEE 12th International Conferen on Embedded Software and Systems (ICESS), 2015 IEEE 17th International Conference on*, pages 128–133. IEEE, 2015.

[7] A. R. Benson, S. Schmit, and R. Schreiber. Silent error detection in numerical time-stepping schemes. *International Journal of High Performance Computing Applications*, 29(4):403–421, 2015.

[8] E. Berrocal, L. Bautista-Gomez, S. Di, Z. Lan, and F. Cappello. Lightweight silent data corruption detection based on runtime data analysis for hpc applications. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, pages 275–278. ACM, 2015.

[9] S. Borkar. Design challenges of technology scaling. *Micro, IEEE*, 19(4):23–29, 1999.

[10] G. Bronevetsky and B. de Supinski. Soft error vulnerability of iterative linear algebra methods. In *Proceedings of the 22nd annual international conference on Supercomputing*, pages 155–164. ACM, 2008.

[11] G. Bronevetsky, I. Laguna, S. Bagchi, B. R. De Supinski, D. H. Ahn, and M. Schulz. Automaded: Automata-based debugging for dissimilar parallel tasks. In *Dependable Systems and Networks (DSN), 2010 IEEE/IFIP International Conference on*, pages 231–240. IEEE, 2010.

[12] C.-L. Chen and M. Hsiao. Error-correcting codes for semiconductor memory applications: A state-of-the-art review. *IBM Journal of Research and Development*, 28(2):124–134, 1984.

[13] I. Chiorean. Parallel methods for solving partial differential equations. *Kragujevac Journal of Mathematics*, 25(25):5–18, 2003.

[14] Y. Crouzet, J. Collet, and J. Arlat. Mitigating soft errors to prevent a hard threat to dependable computing. In *On-Line Testing Symposium, 2005. IOLTS 2005. 11th IEEE International*, pages 295–298. IEEE, 2005.

[15] B. Elkarablieh, I. Garcia, Y. L. Suen, and S. Khurshid. Assertion-based repair of complex data structures. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 64–73. ACM, 2007.

[16] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1):35–45, 2007.

[17] I. Foster. Designing and building parallel programs, 1995.

[18] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the 24th international conference on Software engineering*, pages 291–301. ACM, 2002.

[19] K. Iniewski. *Radiation effects in semiconductors*. CRC Press, 2010.

[20] I. Karlin, J. Keasler, and R. Neely. Lulesh 2.0 updates and changes. Technical Report LLNL-TR-641973, August 2013.

[21] T. Kauranne et al. Introducing massively parallel computers into operational weather forecasting. *Acta Universitatis Lappeenrantaensis*, 2002.

[22] S. Mitra, I. Laguna, D. H. Ahn, S. Bagchi, M. Schulz, and T. Gamblin. Accurate application progress analysis for large-scale parallel debugging. In *ACM SIGPLAN Notices*, volume 49, pages 193–203. ACM, 2014.

[23] S. Mitra, N. Seifert, M. Zhang, Q. Shi, and K. S. Kim. Robust system design with built-in soft-error resilience. *Computer*, (2):43–52, 2005.

[24] G. P. Saggese, N. J. Wang, Z. T. Kalbarczyk, S. J. Patel, and R. K. Iyer. An experimental study of soft errors in microprocessors. *IEEE micro*, (6):30–39, 2005.

[25] P. N. Sanda, J. W. Kellington, P. Kudva, R. Kalla, R. B. McBeth, J. Ackaret, R. Lockwood, J. Schumann, and C. R. Jones. Soft-error resilience of the ibm power6 processor. *IBM Journal of Research and Development*, 52(3):275–284, 2008.

[26] V. C. Sharma, G. Gopalakrishnan, and G. Bronevetsky. Detecting soft errors in stencil based computations. *Geophysics*, 48(11):1514–1524, 1983.

[27] P. Shivakumar, M. Kistler, S. W. Keckler, D. Burger, and L. Alvisi. Modeling the effect of technology trends on the soft error rate of combinational logic. In *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on*, pages 389–398. IEEE, 2002.

[28] C. W. Slayman. Cache and memory error detection, correction, and reduction techniques for terrestrial servers and workstations. *Device and Materials Reliability, IEEE Transactions on*, 5(3):397–404, 2005.

[29] C. Weaver, J. Emer, S. S. Mukherjee, and S. K. Reinhardt. Techniques to reduce the soft error rate of a high-performance microprocessor. In *ACM SIGARCH Computer Architecture News*, volume 32, page 264. IEEE Computer Society, 2004.

[30] J. Wei and K. Pattabiraman. Blockwatch: Leveraging similarity in parallel programs for error detection. In *Dependable Systems and Networks (DSN), 2012 42nd Annual IEEE/IFIP International Conference on*, pages 1–12. IEEE, 2012.