Probabilistic Data Assertions for Catching Silent Errors in Parallel Programs

Anmol J. Bhattad
School of Electrical and Computer Engineering
Purdue University
West Lafayette, IN 47907
Email: bhattad@purdue.edu

Tara Elizabeth Thomas
School of Electrical and Computer Engineering
Purdue University
West Lafayette, IN 47907
Email:thoma579@purdue.edu

***Abstract*—With the advancement in technology and increase in the amount of data to be analyzed, parallel programs have become widespread. These programs are complex and any silent error in the data could have a significant impact on the results. Hence, it is important to determine the presence of such errors. In this project, we identify a novel idea of using probabilistic data assertions based on spatial and temporal locality of physical variables for C-MPI based parallel programs. We use Neural networks to learn the patterns and produce these assertions which can be embedded in the source code. We have shown the implementation of this technique on a parallel benchmark program - LULESH. The results show that our technique effectively detects silent errors.**

I. Introduction

With the advancement in the areas of internet of things, data storage and analytics, the amount of data generated to be analyzed has increased exponentially. In this scenario, there is a trend towards parallel processing so that huge amounts of data can be processed simultaneously. While dealing with such large amounts of data processed in parallel, even a small error in data values can cause serious data integrity issues because data flow is usually cascaded [1]. This results in corruption of final output. Hence, it is crucial to ensure reliability of parallel programs for accuracy, so that that processor power and time are not spent in vain.

Obvious errors like crashes and system slowdowns in these programs can be detected easily. There also currently exists effective techniques to root cause these errors and to aid in debugging them. However, a lot of data errors are such that they dont make the system unstable. Hence, they dont lead to a crash, hang or failure of any sanity test of the system and might go undetected. These errors are silent data errors. In fact, the complexity in hardware and software increases the chances of silent data errors, and hence it is critical to address these kinds of errors in parallel programs. Silent errors usually remain undetected unless specifically analyzed by an expert in the field. Studies [5] have shown that this could lead to hefty losses and can even be catastrophic when it comes to critical applications.

There are no effective techniques to predict these kinds of errors. Traditional techniques include running a program multiple times, and hence providing temporal redundancy to ensure that transient errors are eliminated. Another technique is running the program on multiple sets of machines assuming that this would eliminate machine hardware specific errors, if any. However, these techniques are not sufficient and it is important to come up with more efficient techniques to detect and notify the user about the possible silent data corruptions.

II. Related work

There has been significant work done on detecting crash and slow downs in parallel programs. The work done by Bronevetsky et al. [2] 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. [3] 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 [4] 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 [6] 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. [7] 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. [8] uses values computed by a simple parallel solver and compare them with the values from the main solver. Berrocal et. al.

[9] 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. The work by Sharma et. al. [10] 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.

III. Approach

Many parallel programs deal with variable values spread out over a region or over time. It is a common practice in such programs to use different cores to handle the values of specific parameters of interest like, say temperature or velocity at different locations [11] [12]. This is prevalent because calculations and manipulations using these variables could be done independently to calculate other parameters for the spatial regions in parallel. These parallel processes, very often follow synchronous communication, i.e, in each iteration after having progressed to some stage, they communicate few of these parameters using some protocol, like MPI.

A. Principle of locality

It is intuitive that in programs that deal with physical quantities like pressure, position etc, there would be some kind of gradient or pattern among these variables along spatial co-ordinates. For example,if nearby nodes are computing the temperature values of geographically nearby regions, the range of values will be small. This is called spatial locality. Also, if a program calculates values of physical quantities over steps of time, then we expect the variation to be gradual. For example, if nearby nodes are computing the temperature values of a regions at consecutive time periods, the range of values will be small. This is called temporal locality.

B. Procedure

In this project, we try to make use of this inherent spatial and temporal locality of variables in parallel programs to generate probabilistic data assertions, which can serve as indicators of silent data errors by identifying anomalous variable values. First, we instrument the relevant parallel program to dump out the crucial variables which exibit the principle of locality. We run the program with multiple valid inputs to generate a large and wide variety of possible values for these variables. Features are then generated out of this. Negetive samples are obtained from this set by modelling one bit error in the values. A big part of this feature set is used to train and validate a neural network. The remaining part of the feature set is to test the performance of the neural network and decide the threshold. Assertions are generated based on these thresholds and inserted in the code. Figure 1 gives the project overview.

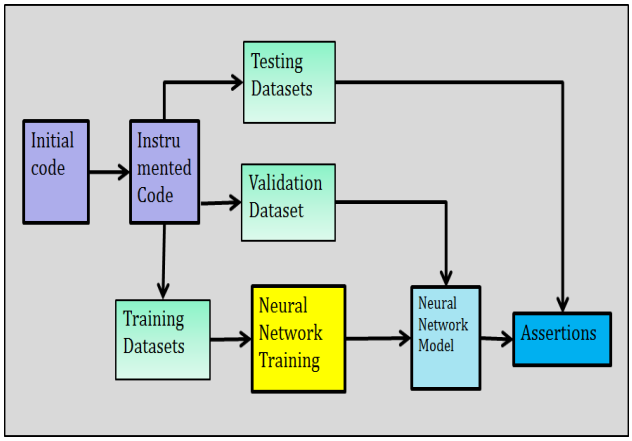


Fig. 1. Overview of the project

C. Selection of Variables

Based on the structure and formulation of the problem that the program solves, we identify few variables that might exhibit the principle of locality. We also ensure that these variables are either output variables or variables that have a significant impact on the outputs. Hence, any data corruptions in their values, will result in the deterioration of the reliability of the program output. These variables are dependent on the values of several other variables from the present and/or previous iterations of the program. Thus, any data corruptions in such variables will peturb the values of our chosen variables. This indirectly identifies silent data corruptions in other variables too.

The values taken by these chosen variables are dumped during runtime. This is achieved by instrumenting the source code. We ensure that we store both the spatial coordinates and the iteration numbers while dumping these variable values. This helps in generating features based on locality.

D. Feature Generation

For each variable calculated at each point in space, our feature set comprises of the values of the same variable at all the adjacent points. For instance, in a three-dimensional space, each point (x, y, z) has 6 neighbours 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 2. 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 neighbouring processor's space and needs inter processor communication to exchange such values and produce the complete feature set for each of the chosen variables.

For exploiting temporal locality in the variable, at time step t we take the value of the choosen variable in the previous iteration or time-step $t-1$ also as a feature.

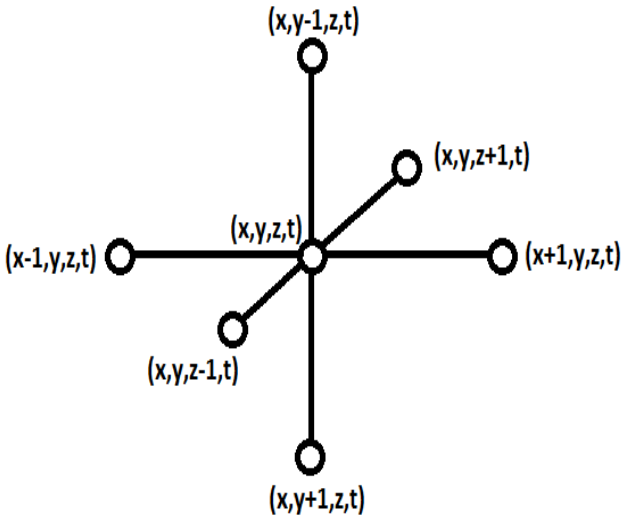


Fig. 2. The neighbours of an element at a unit distance

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 intutive 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.

With the insights from the discussion above, we use 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.

E. Neural Nets

We use neural networks to generate curve fitting models for the chosen variables. When we are unaware of the model of the system, neural networks provide a good approximation of the relationships between the data and corresponding features. Hence we use this algorithm for our project.

F. Error Model

Silent errors are typically caused by circuit noise generated during low power operations in the circuit [14], or radiation from chip packaging [15]. These errors can be modelled as one or more bit errors. For evaluating our model, we consider only single bit errors. The variables that we deal with in such cases are floating points. While modelling single bit errors we consider complementing only the exponent bits or the higher order significand bits, as any change in one bit of the lower order significand bits would not impact the value of the variable much.

G. Assertions

Based on the neural network that we tested using positive and neagative samples, we generate threshholds. We generate assertions by threshholding the output of the neural nets. These assertions are inserted in the source code at appropriate places to catch silent errors. To compute the output of the neural net for a new sample is not compuittionally intensive and can be easily accomodated during runtime.

IV. Implementation

Many of the physical sciences or engineering problems involve solving ordinary differential equations or partial differential equations in multiple variables. One such problem is solving the hydrodynamics equations, which basically describes how materials respond when subject to forces. Since solving these equations is a complex problem involving the same set of variables like pressure, internal energy, velocity, volume etc, at different points using the same set of equations, it is a good candidate for implementation as a parallel program. We have selected a computer simulation that solves such a problem to demonstrate our technique for this project.

A. LULESH

LULESH MPI [13] is a C-MPI based benchmark code that solves the Sedov Blast equations. 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 interesct. 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 after each iteration.

We identified LULESH as a potential benchmark to evaluate our idea for the project 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 instrument the code by altering the source code of LULESH 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 extract 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-D.

We merge the data from the runs for different problem sizes and randomly create disjoint training, validation and testing data sets. We separate out 70% of the samples for training, 15% for validation and remaining 15% of the samples for testing. We train a neural network with 2 layers in Matlab with the training and validation data. To test the robustness of the neural net, we generate negative samples to mimic one bit errors in floating point. This is done by randomly choosing one of the bits of the variables (data type double) and complementing it. The size of the negative test samples is same as the positive test samples. We run 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 relative error, e,
$e = \frac{|x-x'|}{x+\epsilon}$,
where ϵ is a very small number to ensure numerical stability. The ROC curves are obtained by thresholding the error e at different values.

Then we extract the neural net parameters of the trained models to get the mathematical expressions for generating probabilistic assertions. The threshold is decided based on the ROC curves we obtained above.

V. Results

We build the dataset for each of the above mentioned variables over different problem sizes. There are 846248 samples for each of the variables in the global dataset. This set is divided into training, validation and testing. Negetive 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. We peturb only the most significant 8 bits of the significand and all the bits of the exponent.

To study the performance of our scheme we evaluate the performance of our models over the positive and negetive test samples.As mentioned above, 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. To study the impact of the performance of our technique with different number of neuron in the hidden layer, we ran experiments with 3, 5 and 10 neurons ($nNeuron = 3, 5, 10$). Figures 3 shows the performance over the four variables, domain.x 3a, domain.y 3b, domain.z 3c and determ 3d. 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.

Figure 4 shows the comparision 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 $determ$.

VI. Conclusion

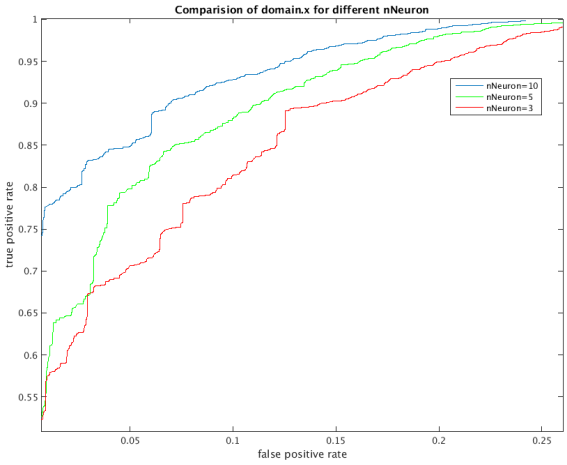
We suggest a novel scheme for generating assertions in parallel programs to catch silent data errors. After identifying and dumping the variables of interest over different runs of a C++-MPI program, we use neural nets to build a model for each variable. These models serve as assertions, for which, the thresholds are decided by evaluating the prediction of neural nets over positive and negative test samples.

VII. Future Work

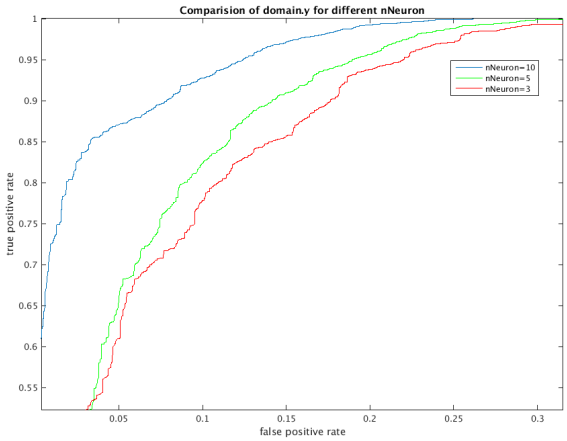
The negative samples are generated and tested offline. To better understand the impact of assertions in detecting silent data errors, we could have a fault injector to malign the parallel code during runtime to model one bit errors. The performance and robustness of the assertions could be better tested by injecting errors in other variables and studying the impact. We will also evaluate our technique on other benchmarks like CoMD [12]. We also look forward to devise a scheme for automatic variable identification and code instrumentation so that the complete process can be automated.

References

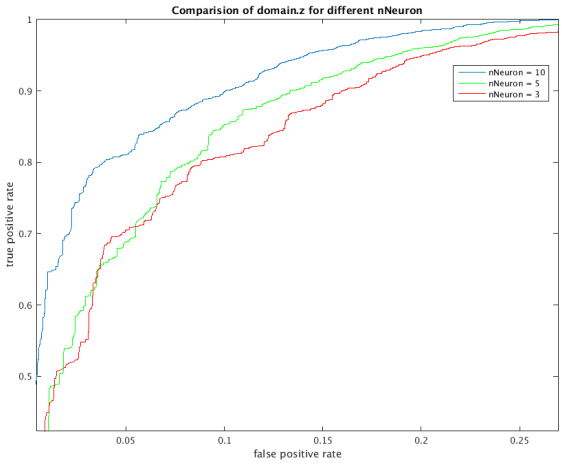
[1] Fiala, David, Frank Mueller, Christian Engelmann, Rolf Riesen, Kurt Ferreira, and Ron Brightwell. "Detection and correction of silent data corruption for large-scale high-performance computing." In Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, p. 78. IEEE Computer Society Press, 2012.



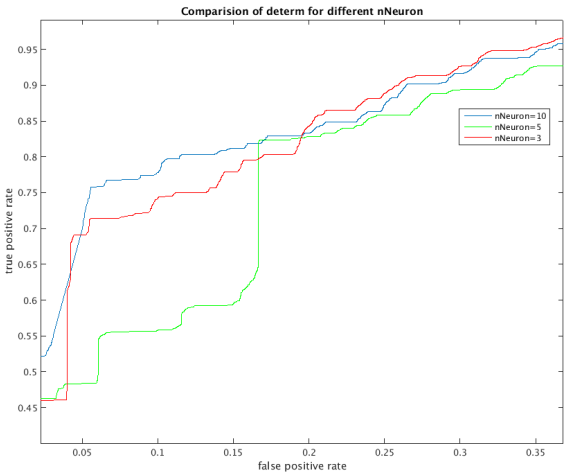
(a) ROC for the variable domain.x



(b) ROC for the variable domain.y



(c) ROC for the variable domain.z



(d) ROC for the variable determ

Fig. 3. Performance of nueral nets for variables with varying number of nuerons in the hidden layer

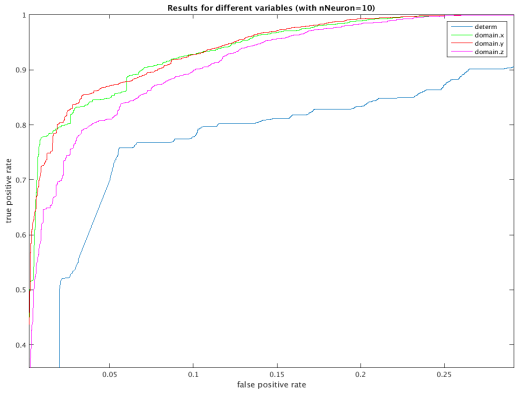


Fig. 4. ROC of different variables with nNuerons=10

[2] Bronevetsky, Greg, Ignacio Laguna, Saurabh Bagchi, Bronis R. De Supinski, Dong H. Ahn, and Martin Schulz. "AutomaDeD: Automata-based debugging for dissimilar parallel tasks." In Dependable Systems and Networks (DSN), 2010 IEEE/IFIP International Conference on, pp. 231-240. IEEE, 2010.

[3] Mitra, Subrata, Ignacio Laguna, Dong H. Ahn, Saurabh Bagchi, Martin Schulz, and Todd Gamblin. "Accurate application progress analysis for large-scale parallel debugging." In ACM SIGPLAN Notices, vol. 49, no. 6, pp. 193-203. ACM, 2014.

[4] Wei, Jiesheng, and Karthik Pattabiraman. "BLOCKWATCH: Leveraging similarity in parallel programs for error detection." In Dependable Systems and Networks (DSN), 2012 42nd Annual IEEE/IFIP International Conference on, pp. 1-12. IEEE, 2012.

[5] Constantinescu, Cristian, Ishwar Parulkar, Rick Harper, and Sarah Michalak. "Silent Data CorruptionMyth or reality?." In Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on, pp. 108-109. IEEE, 2008.

[6] Ernst, Michael D., Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. "The Daikon system for dynamic detection of likely invariants." Science of Computer Programming 69, no. 1 (2007): 35-45.

[7] Elkarablieh, Bassem, Ivan Garcia, Yuk Lai Suen, and Sarfraz Khurshid. "Assertion-based repair of complex data structures." In Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering, pp. 64-73. ACM, 2007.

[8] Benson, Austin R., Sven Schmit, and Robert Schreiber. "Silent error detection in numerical time-stepping schemes." International Journal of High Performance Computing Applications (2014): 1094342014532297.

[9] Berrocal, Eduardo, Leonardo Bautista-Gomez, Sheng Di, Zhiling Lan, and Franck Cappello. "Lightweight silent data corruption detection based on runtime data analysis for hpc applications." (2014).

[10] Sharma, Vishal Chandra, Ganesh Gopalakrishnan, and Greg Bronevetsky. "Detecting Soft Errors in Stencil based Computations." Geophysics 48, no. 11 (1983): 1514-1524.

[11] "Hydrodynamics Challenge Problem", Lawrence Livermore National Laboratory, LLNL-TR-490254, Livermore, CA

[12] http://exmatex.github.io/CoMD/

[13] Karlin, Ian, Jeff Keasler, and Rob Neely. "Lulesh 2.0 updates and changes." Livermore, CAAugust (2013).

[14] Borkar, Shekhar. "Design challenges of technology scaling." Micro, IEEE 19, no. 4 (1999): 23-29.

[15] Sanda, Pia N., Jeffrey W. Kellington, Prabhakar Kudva, Ronald Kalla, Ryan B. McBeth, Jerry Ackaret, Ryan Lockwood, John Schumann, and Christopher R. Jones. "Soft-error resilience of the IBM POWER6 processor." IBM Journal of Research and Development 52, no. 3 (2008): 275-284.