Data Assertions for Catching Silent Errors in Parallel Programs

Ming Shi, YiChieh Ho

I. Introduction

Scientific applications often process large amounts of input data and generate output data. There may be errors in the input data or errors in the algorithm that processes the input data. An example of the former is that the disk system had an error and an example of the latter is that there is floating point overflow or underflow. As a result of these kinds of errors, the program may fail in an obvious way, such as, by crashing, hanging, or failing some sanity checks built into the program. Several people have worked on detecting such control flow errors and working back to find what the root cause was [1]–[3]. However, the other kind of possibility is what worries scientists that is of the silent error. This means that the program fails but there is not an obvious manifestation, i.e., a manifestation that is obvious to the non-expert in that particular science domain. For example, the program may determine that there is a match of the genetic material extracted from a scene of crime with that of the genetic material obtained from a suspect, when in reality there is not.

1. Running the application multiple times on the same machine and hoping that the error was a transient one and goes away.

2. Running the application in multiple copies on different machines and hoping that the same error does not affect multiple machines, such as, a lack of memory happens on one machine affecting one copy of the application, but not the others.

3. The most sophisticated technique is to use machine learning to put in data assertions. The idea is that there will be some training runs, say with data set for which the correct answer is known, or at a small scale when the applications output can be hand checked. Through these training runs, the

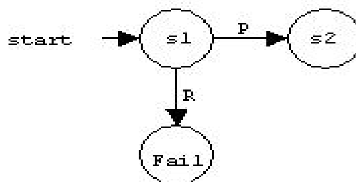


Fig. 1: CoderSurfer Chart

algorithm will learn what are the correct ranges for data values of various variables and then embed these as data assertions (such as, $assert(num_gene_sequences < 10^6)$) in the program.

We utilize the commonly used code browser tool, that is CodeSurfer to help us find the critical variables of the silent errors. CodeSurfer is a code browser that understands pointers, indirect function calls, and whole-program effects. CodeSurfer is the most sophisticated code browser available for C/C++ or x86 machine code; it is the static analysis tool of choice for organizations that manually review software for critical applications. While CodeSonar is an automated static analysis tool that finds bugs and generates a report of defects in the code, CodeSurfer is a program-understanding tool that makes manually analyzing code more efficient. Many program-understanding tools interpret code loosely. In contrast, CodeSurfer does a precise analysis. Program constructs including preprocessor directives, macros, and C++ templates (in the case of source code) and machine instructions (in the case of Intel x86 binaries) are analyzed correctly. CodeSurfer calculates a variety of representations that can be explored through the graphical user interface or accessed through the optional programming API. CodeSurfer Path Inspector is a CodeSurfer extension that helps you understand sequencing properties in programs. The CodeSurfer Path Inspector is an optional extension for CodeSurfer that answers complex questions about the flow of execution, to help you understand a program's behavior. In an example of query Construction and Evaluation, twenty-five query templates are provided. Each template is in the form of a state machine. The user specifies the transitions of the state machine by associating a set of program points with each transition. In the DNS example above, the query template used is called P occurs before R and is illustrated below.

II. Background

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 [5]. 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 don't make the system unstable. Hence, they don't 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 [6] 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.

There has been significant work done on detecting crash and slow downs in parallel programs. The work done by Bronevetsky et al. [1] 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. [4] 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 [2] uses similarity among tasks of a parallel program for runtime error detection.

In this project, we will work with an open source parallel program written in C-MPI. MPI programs

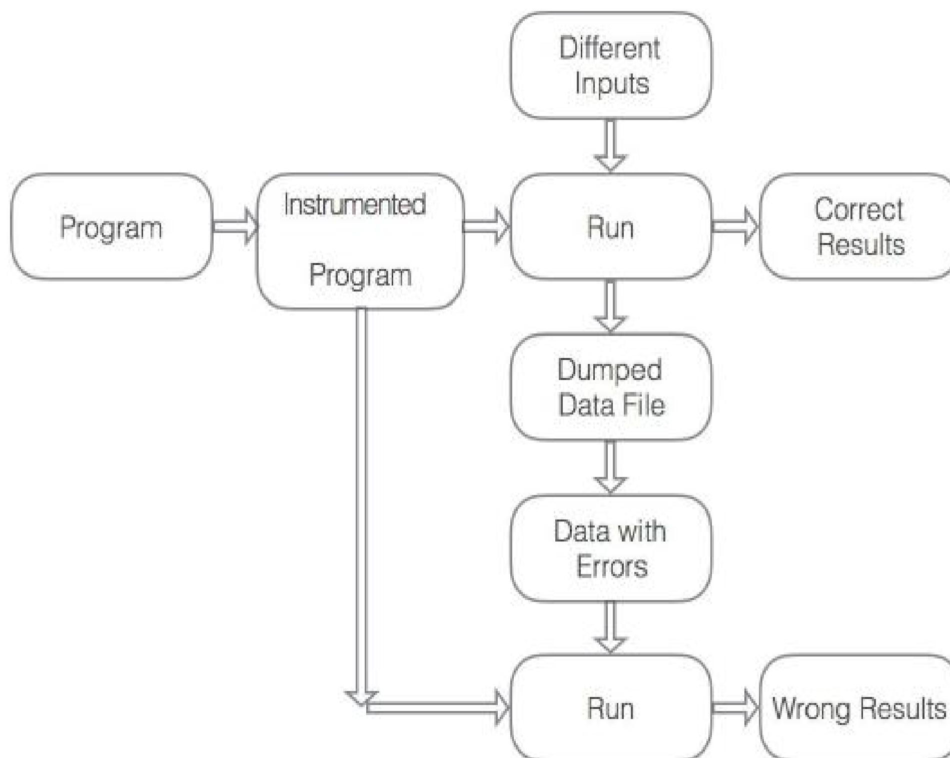


Fig. 2: Architecture of Our Solution Approach

run on multiple machines and sometime operate on massive data sets. The goal for this year will be to develop a technique to identify what are the critical variables that need to be monitored and on which assertions will be created. This will involve using static analysis and possibly dynamic analysis and machine learning to identify which variables have the greatest impact on the end result of a program. As a stretch goal, we will want to identify the original variable that was in error that caused the invariant to be violated.

A project done in this class two years back created invariants that were learned through some training runs using a Deep Neural Network. These invariants were then inserted into the program for detecting the fail silent error. It leveraged the insight that the values of the same variable at nearby nodes are likely to be similar, or the pattern may be easily discernible. For example, if nearby nodes are computing the temperature values of geographically nearby regions, you would expect the range of values will be small.

III. Solution Approach

Our high-level idea of changing input error which influence the final result:

```
1  //Lulesh Workflow
2  class Domain:
3    double x , y, z ;  // coordinates
4    double dx , dy , dz ; // velocities
5    double ddx , ddy , ddz ; //accelerations
6    double fx , fy , fz ; // forces
7    double mass ; // mass
8
9  int main() {
10   Domain domain ;
11
12   while( time constraint ){
13     Calculation( domain ) ;
14   }
15 }
```

CodeSurfer is used to identify and nevigate the deep structure of a program.

We have two feasible approaches. The first is Backward slicing up to the code line that we receive the final result.

Forward Slicing has the Pros that it is efficient. Since it Directly calculates the number of times of the chosen variable that really affects the final results. However it is not very accurate.

The second is to Combine the forward slicing with the backward slicing. Then we can evaluate the criticality of the variables by measuring its position and the checking.

The second idea is much more elegant. We do Backward Slicing + Forward Slicing.

Pros: Check the exact meaning of the selecting variables after doing forward slicing from the code line it is being use for the final result.

Cons: Complex (Not very efficient).

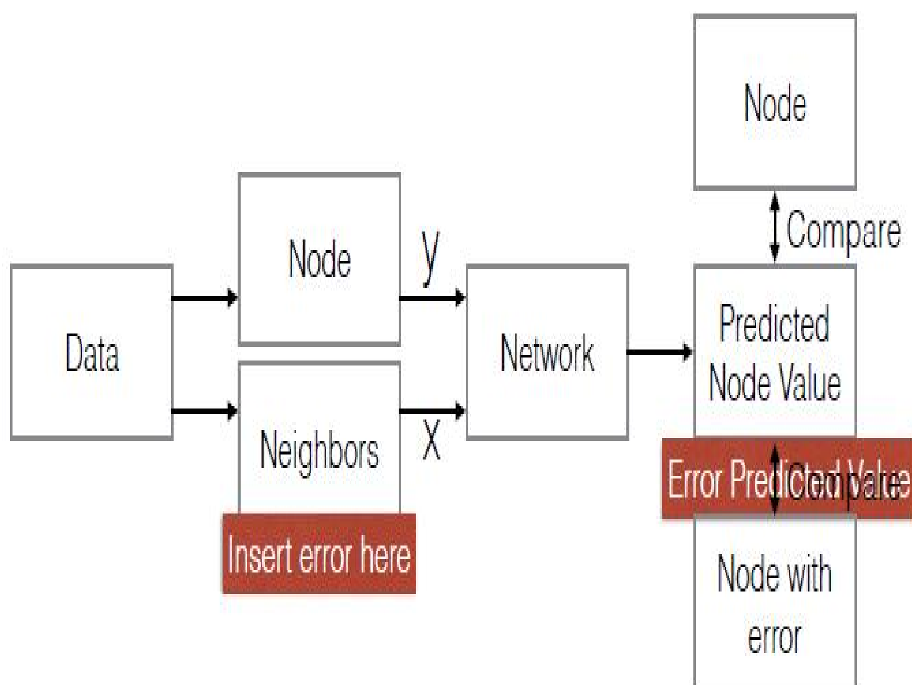
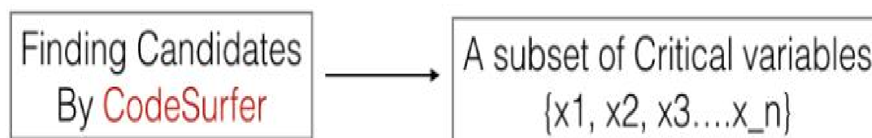


Fig. 3: Analysing Framework



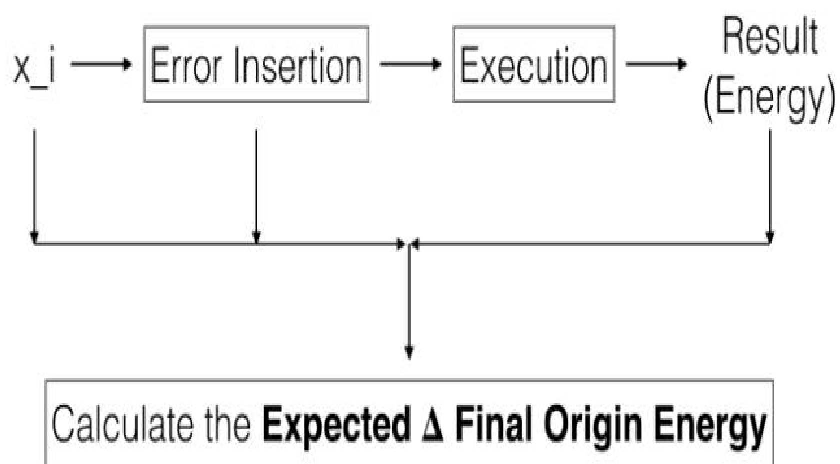


Fig. 4: Updated Solution Approach

IV. Detailed Design

A. Error Model

We first flip one of all 64 bits of a double-precision floating-point. We assume,

1. an uniform probability of each bit flipping.

2. Only one error occurs in the entire program.

3. Uniform probability of each iteration an error occurs.

We define the critical variables as the variables with the largest expected difference between original output and erroneous output.

B. Target Program

We use the Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (Lulesh). Hydrodynamics, which describes the motion of materials relative to each other when subject to forces. LULESH approximates the hydrodynamics equations discretely by partitioning the spatial problem domain into a collection of volumetric elements defined by a mesh.

We show the details about how we inject errors in Fig. 7.

C. Details about the exact solution framework

Idea I: Baseline - Brute-force:

We can test all possible conditions (each iteration and bit). This is very precise however very time-consuming. For example, for 64 bits, 100 iterations, 40 variables, and 1 sec / execution, we need to run for $64 * 100 * 40 * 1 = 256000sec = 71hours$. Moreover, in reality, there may be more iterations and more possible variables.

Idea II: Sampling Brute-force:

For example, we inject error every 10 iterations, and only selecting 15 most significant bits. Similarly, for the considering case where there exists 15 bits, 10 iterations, 10 variables, 1 sec / execution, we need $15*10*10*1 = 1500seconds = 25minutes$ to get the results. It is neither accuracy nor efficient.

Idea III: Hierarchical Searching:

```
while ( Timing Condition || Number of Iterations ){
    Calculate( structure Domain ) ;
    TimeIncrement();
}
structure Domain {
    double velocity ;
    double accelerations ;
    double energy ;
};
```

Fig. 5: Lulesh Structure

```
while ( Timing Condition || Number of Iterations ){
   if( current_iteration == target_iteration)
      Error_injection ( target_variable , target_bit ) ;
   Calculate( structure Domain ) ;
   TimeIncrement();
}
```

Fig. 6: Lulesh Structure

Given a condition, testing all the variables. Then, select a few variables into the second testing with the second error condition. First Round : Flipping the 63rd bit at the 10th iteration. Selecting 10 variables with the largest expected error. Second Round: Flipping the 62nd bit at 10th iteration.

Hence, for all these ideas, number of times of a variable involving in a computation appears to follow the rules that the more a faulty variable is used, the larger error will be. We consider all the number of Addition or Subtraction and the number of multiplication or division.

V. Implementation

We implement Baseline - Brute-force, Sampling Brute-force, Hierarchical Searching, Backward slicing and the combination of forward slicing with the backward slicing. Especially, for the combination of the forward slicing and backward slicing using CodeSurfer, we first do the backward slicing from where we get the final result to check where the critical variables influence the final results. Then we start from the code line where the variables influence the final results to do the forward slicing. By doing these browsing, we can get the number of times the variable influence the final result and then we can make decisions based on the following two assumptions:

a. The number of times the critical variable influence the final result increase linearly with time.

b. The value of the output error accumulate with the times of the critical variable influence the final result.

Then we change the time of how long the critical variables influence the final result. Then by borrowing ideas from classical convex optimization, we know,

Subgradient: If $f : U \to R$ is a real-valued convex function defined on a convex open set in the Euclidean space R^n, a vector in that space is called a subgradient at a point x_0 in U if for any x in U one has $f(x) - f(x_0) \geq v * (x - x_0)$.

Theorem 1: Fixing Δ, where $\Delta = x - x_0$, the error of output should scale as the subgradient v in the dual space $V*$.

Fig. 7: Expected Difference for Higher Rate Case

VI. Experimental results

A. Experiment Setting

We only consider variables which are double-precision floating point data members in structure Domain. Remove "nan" condition because it is detectable. Only choose 3 most critical variables. 100 iterations, 8 cores, and input size equal to 9. Ideal Output = 285687.

For higher sampling rate Brute-Force, we do every 10 iterations with 40 domain data members. We flip every bit. The result is shown in Fig. 8.

The critical variables are $domina.m_v(Volume)$, $domain.m_e(Energy)$, $domina.m_p(Pressure)$, and the expected difference are 1.99985158371e+11, 1088473, 111210, respectively.

For Lower sampling rate - Brute-Force, we do every 20 iterations with 40 domain data members. We flip first 12 bits. The result is shown in Fig. 9.

The critical variables are $domina.m_v(Volume)$, $domain.m_e(Energy)$, $domina.m_volo(RelativeVolume)$, and the expected difference are 1.74610282929e+12, 7250256, 277532, respectively.

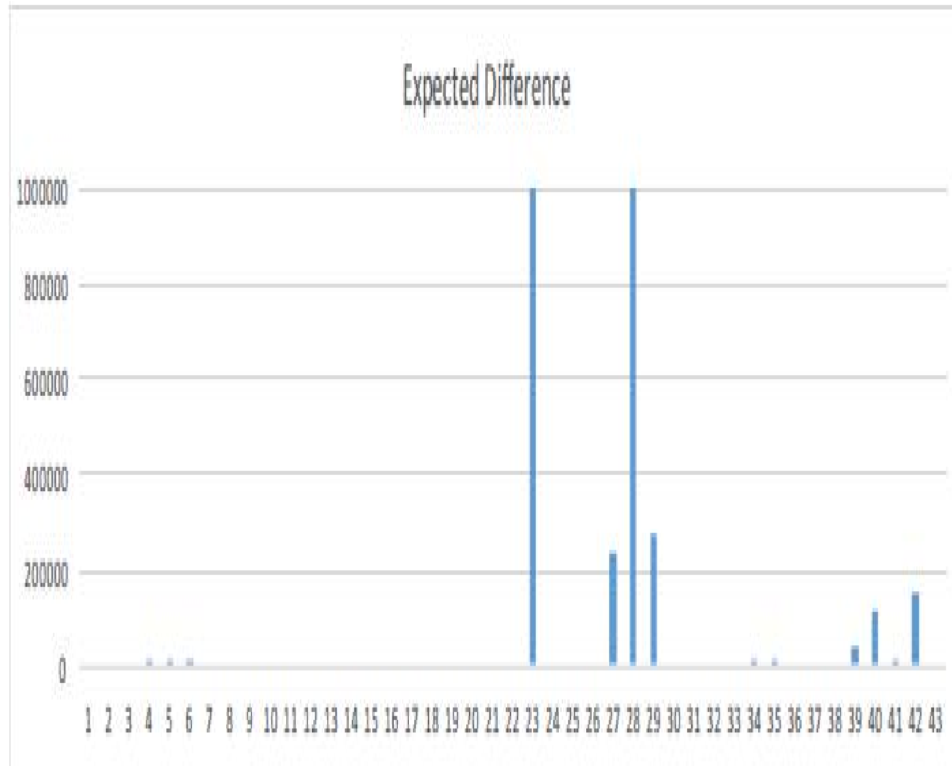


Fig. 8: Expected Difference for Lower Rate Case

For the Hierarchical Searching, we it as in the first round : the 64rd-60th bit, the 50th iteration, for all variables. The result is shown in Fig. 10.

In the second round : the 59th-55nd bit, the 50th iteration, 10 variables. The result is shown in Fig. 11.

The critical variables are $domina.m_v(Volume)$, $domain.m_e(Energy)$, $domain.m_deltatime(Timing)$, and the expected differences for the second round are 37455530, 539999000, 151,630, respectively.

Finally using CodeSurfer, we can compare any pair of critical variables to decide which one is the real critical one, as shown in Fig.12. Here, we take the critical variables $Domain.v$ and $Domain.e$ as an example.

VII. Related work

There has been significant work done on detecting crash and slow downs in parallel programs. The work done by Bronevetsky et al. [1] 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

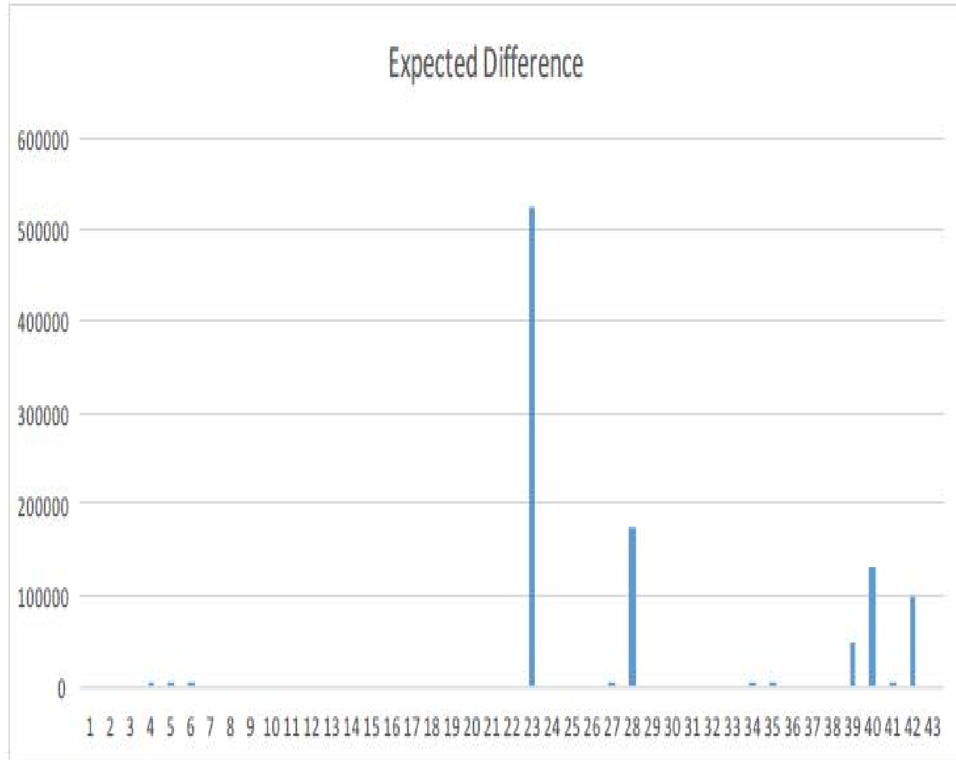


Fig. 9: Expected Difference for Hierarchical Searching in the first round

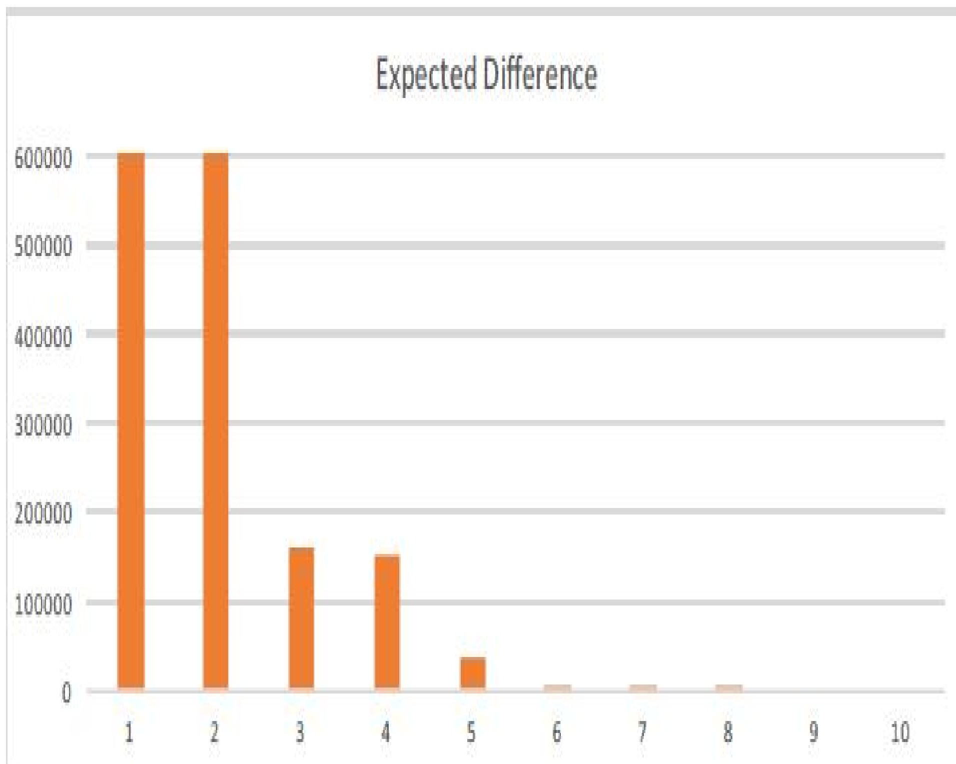


Fig. 10: Expected Difference for Hierarchical Searching in the 2nd round

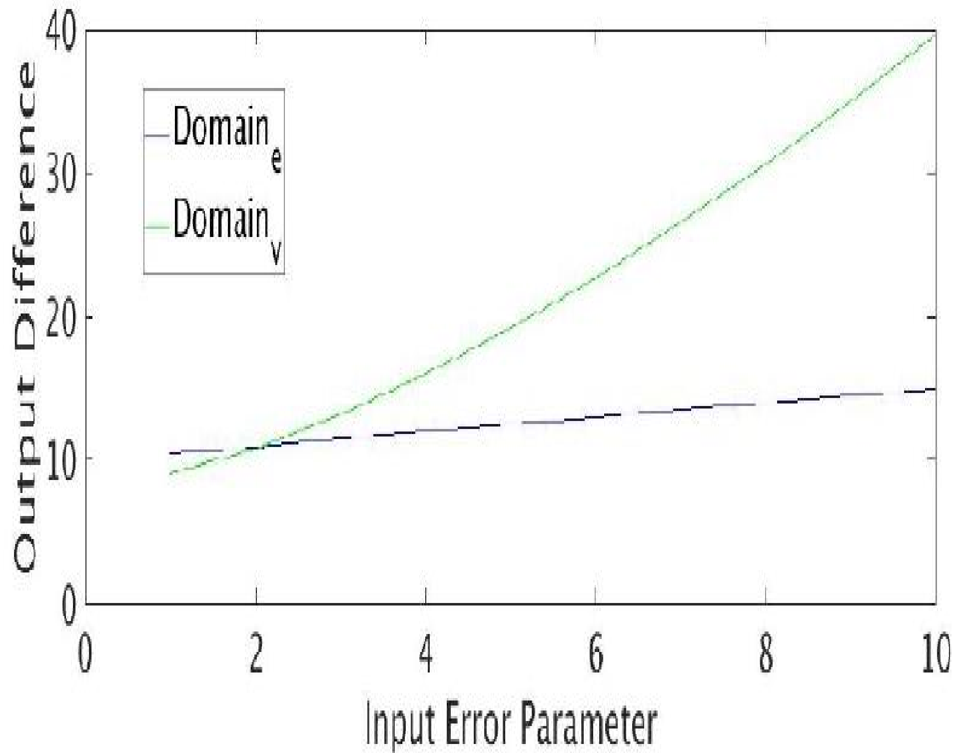


Fig. 11: Comparing critical variables

that first manifested a bug, using clustering and then identifies the specific code region and execution. [4] 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 [2] uses similarity among tasks of a parallel program for runtime error detection.

VIII. Conclusion and Future Work

We proposed Brute-force, Sampling Brute-force, Hierarchical Searching, and the combination approach that using backward slicing and forward slicing jointly. From the result we notice that we can get the critical variables efficiently and with a certain accuracy.

We plan to use different ways to estimate the usage of a variable. Then the checking speed could be even faster. Also, we can use different error models in the future, for example, errors could happen in each computation rather than iteration. Moreover, 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. 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] 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*. IEEE, 2010, pp. 231–240.

[2] 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*. IEEE, 2012, pp. 1–12.

[3] T. E. Thomas, A. J. Bhattad, S. Mitra, and S. Bagchi, “Sirius: Neural network based probabilistic assertions for detecting silent data corruption in parallel programs,” in *Reliable Distributed Systems (SRDS), 2016 IEEE 35th Symposium on*. IEEE, 2016, pp. 41–50.

[4] 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*, vol. 49, no. 6. ACM, 2014, pp. 193–203.

[5] D. Fiala, F. Mueller, C. Engelmann, R. Riesen, K. Ferreira, and R. 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*. IEEE Computer Society Press, 2012, p. 78.

[6] C. Constantinescu, I. Parulkar, R. Harper, and S. Michalak, “Silent data corruptionmyth or reality?” in *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*. IEEE, 2008, pp. 108–109.