Handling Soft and Hard Errors for Scientific Applications

DISSERTATION

Presented in Partial Fulfillment of the Requirements for the Degree Doctor of Philosophy in the Graduate School of The Ohio State University

By

Jiaqi Liu, B.S., M.S.
Graduate Program in Computer Science and Engineering

The Ohio State University

2017

Dissertation Committee:
Dr. Gagan Agrawal, Advisor
Dr. Mircea-Radu Teodorescu
Dr. P. Sadayappan
© Copyright by

Jiaqi Liu

2017
Abstract

Due to the rapid decrease in Mean Time Between Failure (MTBF) in High Performance Computing, fault tolerance emerged as a critical topic to improve overall performance in the HPC community. In recent decades, along with the decrease in size of hardware, and the extensively used near-threshold computation for energy saving, the community is now facing more frequent soft errors than ever. Particularly, due to the difficulty in detecting soft errors, we are in urgent need for a general solution for these errors. Our work includes providing efficient and effective solution to handle soft and hard errors for parallel system.

We start from solving the write bottleneck of the traditional checkpoint and restart. We exploit the communication structure to find locally finalized data, as well as each process’s contribution to globally finalized data. We allow each node to take independent checkpoint using this information and therefore achieve uncoordinated checkpointing. We checkpoint asynchronously by overlapping the workload of checkpoint with computation, so that the system avoids write congestion.

We discovered that the soft error impact in convergent iterative applications’ output follows a pattern. We developed a signature analysis based detection with checkpointing based recovery, which is driven by the observation that high order bit flips can very negatively impact execution, but can also be easily detected. Specifically, we have developed signatures for this class of applications.
For non-monotonically convergent applications, we observed that the signature of silent data corruption is specific to an application but independent of the input dataset size for the application. Based on this observation, we explored an approach that involves machine learning technique to detect soft errors. We use off-line training framework of machine learning, construct classifiers with representative inputs and periodically invoke the classifiers during execution to verify the status.

Our work not only focuses on optimizing the existing fault tolerance solution to handle general case of faults, but also includes exploring new algorithms that detects and recovers from soft errors. We proposed an algorithm level fault tolerance solution for molecular dynamic applications to detect soft errors and recover from the error. We also developed an algorithm level recovery strategy, so that the applications do not need traditional checkpoint to back up the computation state.

Finally, we supported in-situ analysis paradigm with fault resilience. We explored a Map-Reduce like platform for in-situ analysis and discovered the possibility of achieving runtime execution state by utilizing the redundant properties of reduction objects during computation. With the state stored in the shared locations among the nodes, we could maintain a checkpoint-restart like mechanism and the system could restart from any previous backup if any node fails. We were able to apply the approach both time-wise and space-wise for the Smart with reasonable extra overhead.
This is dedicated to the ones I love: my parents, my sister and my fiancée.
Acknowledgments

This dissertation is completed not only by my work, it is the result of all the help I got during my days at the Ohio State. There are so many people that I want to express my deepest gratitude to. Without their help and support, it would not have been possible for me to finish this dissertation.

Foremost, I wish to thank my advisor, Prof. Agrawal. He has supported me since I began working in the lab as a master student. It was him who brought me into the high performance computing. His knowledge in the research area and insightful guidance did not only broaden my domain knowledge, but also helped me to establish my research interest and develop my problem solving skills. I also want to thank for his kindness and patience that helped me out through the hard times I had. For some days, I lost my direction for both my research and life, he gave me moral support and freedom that I needed to move on.

Besides my advisor, I would like to thank the rest of my thesis committee: Prof. Mircea-Radu Teodorescu, Prof. P. Sadayappan and Prof. Birgit Alber. Their insightful comments and incisive analysis helped my thesis become stronger and more solid.

I would also like to thank Mr. Zhao Wang and Mr. Zhaowei Hou, who helped me to significantly improve my programming skills and deepen my industry level knowledge during my internship at Facebook.
It is so grateful that I have met wonderful colleagues in our group - Data Intensive and High Performance Computing Research Group. The friendship from them have made my Ph.D. life much more enjoyable. I thank my fellow labmates: Bin Ren, Xin Huo, Tekin Bicer, Yu Su, Yi Wang, Mehmet Can Kurt, Mucahid Kutlu, Sameh Shohdy, Roee Ebenstein, Gangyi Zhu, Peng Jiang, Jiankai Sun, Piyali Das, Jia Guo, Shuangsheng Luo, Omid Asudeh and Emin Ozturk for the stimulating discussions and for all the fun we have had in the past years.

I’m also thankful to my friends Chen Li, Dongzhu Shen, Zheyu Jin as well as those who are further in China for helping me when I needed most.

Finally, I want to extend my special thanks to my parents, my sister and my fiancée Chang Liu, who has always been supportive. No words can express their help to me for my life and study. At last, small thanks to my lovely two cats: Ashball and Flowerball.
Vita

December 28th, 1988 . . . . . . . . . . . . . . . . . . . . . . . . Born - Yanji, China

2010 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . B.S. Software Engineering,
Beihang University, Beijing, China

2015 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . M.S. Computer Science & Engineering,
Ohio State University, Columbus, OH

2015 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . Software Engineering Intern, Facebook,
Menlo Park, CA

Publications

Research Publications


Fields of Study

Major Field: Computer Science and Engineering
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>ii</td>
</tr>
<tr>
<td>Dedication</td>
<td>iv</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>v</td>
</tr>
<tr>
<td>Vita</td>
<td>vii</td>
</tr>
<tr>
<td>List of Tables</td>
<td>xiii</td>
</tr>
<tr>
<td>List of Figures</td>
<td>xv</td>
</tr>
<tr>
<td>List of Algorithms</td>
<td>xix</td>
</tr>
<tr>
<td>List of Code Listings</td>
<td>xx</td>
</tr>
<tr>
<td>List of Abbreviations</td>
<td>xxi</td>
</tr>
<tr>
<td>1. Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Overview of Fault Tolerance</td>
<td>2</td>
</tr>
<tr>
<td>1.1.1 Categories of Fault Tolerance Approaches</td>
<td>3</td>
</tr>
<tr>
<td>1.1.2 Discussion on Fault Tolerance Solutions</td>
<td>5</td>
</tr>
<tr>
<td>1.2 Soft Errors and Hard Errors</td>
<td>7</td>
</tr>
<tr>
<td>1.2.1 Hard Errors</td>
<td>7</td>
</tr>
<tr>
<td>1.2.2 Soft Errors</td>
<td>8</td>
</tr>
<tr>
<td>1.3 Dissertation Contributions</td>
<td>9</td>
</tr>
<tr>
<td>1.3.1 Asynchronous Uncoordinated Application-level Checkpoint</td>
<td>9</td>
</tr>
<tr>
<td>1.3.2 Signature based method and Partial Replication for Iterative</td>
<td>10</td>
</tr>
<tr>
<td>Applications</td>
<td></td>
</tr>
</tbody>
</table>
1.3.3 Algorithm based Fault Tolerance for Molecular Dynamic Applications ................................ 11
1.3.4 Machine Learning based Fault Tolerance for Interactive Convergent Applications .......................... 12
1.3.5 Fault Tolerance Methodologies on Map-Reduce based In-Situ Analysis ................................. 12
1.4 Outline ................................................................................................................................. 13

2. A Methodology for Application-Level Asynchronous Checkpointing for MPI Applications .................................................. 15
2.1 Communication Patterns ........................................................................................................... 15
  2.1.1 Irregular Reductions ........................................................................................................... 15
  2.1.2 Stencil Computations ......................................................................................................... 16
2.2 Main Approach ......................................................................................................................... 16
  2.2.1 Asynchronous Checkpointing ............................................................................................ 19
2.3 Implementing Asynchronous Uncoordinated Checkpointing on Applications ............................................. 22
  2.3.1 Dense Grid Computations ................................................................................................... 23
  2.3.2 Indirection Array-Based Computations ............................................................................... 25
  2.3.3 MiniMD .............................................................................................................................. 29
  2.3.4 Discussion ......................................................................................................................... 30
2.4 Experimental Results .................................................................................................................. 31
  2.4.1 Regular Applications .......................................................................................................... 31
  2.4.2 Irregular Applications ........................................................................................................ 33
  2.4.3 Comparison with System Level Checkpointing .................................................................. 35
2.5 Summary ..................................................................................................................................... 37

3. A Practical Approach for Handling Soft Errors in Iterative Applications ................................................. 38
3.1 A Motivating Study .................................................................................................................... 38
  3.1.1 Experiment Design .............................................................................................................. 38
3.2 Methodology ............................................................................................................................. 46
  3.2.1 Bit Flip Signature for Applications with a Convergence Criteria ......................................... 48
  3.2.2 Signature for Molecular Interaction Applications .............................................................. 50
  3.2.3 Partial Replication ............................................................................................................. 54
3.3 Experimental Results .................................................................................................................. 57
  3.3.1 Effectiveness of Signature Analysis Method ......................................................................... 57
  3.3.2 Overheads of Signature Analysis Method ........................................................................... 63
  3.3.3 Accuracy and Overheads with Partial Replication .............................................................. 64
3.4 Summary ..................................................................................................................................... 65

4.1 Overview of the Problem ................................................................. 66
4.2 Framework Design .......................................................................... 70
4.3 Experimental Results ...................................................................... 75
  4.3.1 Accuracy .................................................................................... 76
  4.3.2 Latency ..................................................................................... 81
  4.3.3 Overheads ................................................................................ 82
  4.3.4 Generalized Fault Injection ........................................................ 83
  4.3.5 Evaluations on Sparse Matrices .................................................. 85
4.4 Summary ....................................................................................... 87

5. Algorithm Level Fault Tolerance for Molecular Dynamic Applications .... 89

5.1 Background Algorithm ..................................................................... 89
  5.1.1 Molecular Dynamics .................................................................. 89
  5.1.2 An ABFT Example .................................................................. 90
5.2 Fault Tolerant Molecular Dynamics ................................................ 92
  5.2.1 Soft Error Detection ................................................................ 94
  5.2.2 Optimization ......................................................................... 97
  5.2.3 Recovery ................................................................................. 98
5.3 Experimental Results ..................................................................... 99
  5.3.1 Fault Model ........................................................................... 99
  5.3.2 Effectiveness of Algorithm Level Fault Tolerance ....................... 101
  5.3.3 Overheads ............................................................................... 107
5.4 Summary ....................................................................................... 109

6. Supporting In-Situ analysis with Fault Tolerance ................................. 110

6.1 In-Situ Analysis ............................................................................ 110
6.2 Overview of Smart .......................................................................... 111
  6.2.1 System Overview .................................................................. 111
  6.2.2 User Program ....................................................................... 114
6.3 Framework Design ......................................................................... 115
  6.3.1 Possible Solutions .................................................................. 115
  6.3.2 Reduction Object Based Solution .............................................. 116
6.4 Implementation of Smart-FR .......................................................... 119
  6.4.1 Smart-FR Overview ............................................................... 119
  6.4.2 Failure Handling ..................................................................... 120
  6.4.3 Mode Specific Processing ....................................................... 122
6.5 Experimental Results .............................................. 123
  6.5.1 Environment and Applications ............................... 124
  6.5.2 Checkpointing Overheads ................................... 125
  6.5.3 Recovery Overheads .......................................... 129
6.6 Conclusion ......................................................... 130

7. Related Work ..................................................... 131
  7.1 Checkpoint and Restart ......................................... 131
  7.2 Replication ....................................................... 133
  7.3 Handling Soft Errors ........................................... 133
  7.4 In-Situ Analysis Platforms and Approach for Resilience .... 136

8. Conclusions ......................................................... 138
  8.1 Contributions .................................................... 138
  8.2 Future Work ...................................................... 140
    8.2.1 In-Memory Checkpointing for Soft and Hard Errors in Parallel Systems ........................................ 140
    8.2.2 Automating Soft Error Detection in Simulative Applications .......................... 142

Bibliography ........................................................ 143
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 Applications and datasets used in experiments</td>
<td>40</td>
</tr>
<tr>
<td>3.2 Average normalized difference between the outputs of normal execution and execution with a single bit flip, depending on the location (bit range) and time (iteration) of the bit flip (Sobel)</td>
<td>41</td>
</tr>
<tr>
<td>3.3 Frequency of executions leading to an infinite loop because of a single bit flip: Sobel</td>
<td>43</td>
</tr>
<tr>
<td>3.4 Average normalized difference between normal execution and execution with single bit flip: Jacobi and CG</td>
<td>44</td>
</tr>
<tr>
<td>3.5 Frequency of executions leading to an infinite loop because of a single Bit flip: Jacobi and CG</td>
<td>45</td>
</tr>
<tr>
<td>3.6 Average Normalized difference between output normal execution and output with single bit flip: MolDyn and MiniMD</td>
<td>46</td>
</tr>
<tr>
<td>3.7 Percentage of Undetected Bit Flips: Jacobi, Sobel and CG</td>
<td>58</td>
</tr>
<tr>
<td>3.8 Average Normalized Difference between Normal Execution and Execution with Single Bit Flip + Signature Analysis (Sobel)</td>
<td>58</td>
</tr>
<tr>
<td>3.9 Average Normalized Difference between Normal Execution and Execution with Single Bit Flip + Signature Analysis (Jacobi)</td>
<td>59</td>
</tr>
<tr>
<td>3.10 Average Normalized Difference between Normal Execution and Execution With a Single Bit Flip + Signature Analysis (CG)</td>
<td>59</td>
</tr>
<tr>
<td>3.11 Percentage of Undetected Bit Flips: MolDyn and MiniMD</td>
<td>60</td>
</tr>
</tbody>
</table>
3.12 Average Normalized Difference between Output with Normal Execution and Output Single Bit Flip + Signature Analysis (MolDyn and MiniMD) . . 61

4.1 Datasets used for Training for Each Application . . . . . . . . . . . . . . . 77

4.2 Applications and Input Set Configuration for Experiment . . . . . . . . . . 77

5.1 F-Scores Details under different Tolerance. Shaded parts are the bit ranges that does not result into noticeable errors in the output. Application: miniMD(4k,[-100,-100]) and coMD(3.2k,[-100,-100]). . . . . . . . . . . . . . . 102

5.2 F-Scores Details under different Tolerance. Shaded parts are the bit ranges that does not result into noticeable errors in the output. Application: miniMD(2k,[-100,-100] and coMD(3.2,[-1,1]). . . . . . . . . . . . . . . 104
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1 A (Simplified) Overview of HPC Fault-Tolerance Approaches</td>
<td>3</td>
</tr>
<tr>
<td>1.2 Fault Models and their Effects</td>
<td>7</td>
</tr>
<tr>
<td>2.1 Checkpointing A Dense Grid Application</td>
<td>23</td>
</tr>
<tr>
<td>2.2 Object Checkpointing</td>
<td>29</td>
</tr>
<tr>
<td>2.3 Comparing Execution Without Checkpointing, With Synchronous Checkpointing and Asynchronous Checkpointing: Jacobi</td>
<td>32</td>
</tr>
<tr>
<td>2.4 Comparing Execution Without Checkpointing, With Synchronous Checkpointing and Asynchronous Checkpointing: Sobel</td>
<td>32</td>
</tr>
<tr>
<td>2.5 Comparing Execution Without Checkpointing, With Synchronous Checkpointing and Asynchronous Checkpointing: Euler</td>
<td>33</td>
</tr>
<tr>
<td>2.6 Comparing Execution Without Checkpointing, With Synchronous Checkpointing and Asynchronous Checkpointing: Moldyn</td>
<td>34</td>
</tr>
<tr>
<td>2.7 Comparing Execution Without Checkpointing, With Synchronous Checkpointing and Asynchronous Checkpointing: MiniMD</td>
<td>35</td>
</tr>
<tr>
<td>2.8 Comparing Asynchronous Uncoordinated Checkpointing with System-level Checkpointing (Measured as Slowdown Percentage)</td>
<td>36</td>
</tr>
<tr>
<td>3.1 Frequency of executions leading to crashes because of a single bit flip (reported for crashes happened in all bit ranges): Moldyn and MiniMD</td>
<td>45</td>
</tr>
<tr>
<td>3.2 Control flow of an application incorporating signature detection, checkpointing and recovery steps.</td>
<td>47</td>
</tr>
</tbody>
</table>
3.3 The change in the convergence variable (diffnorm) of Sobel as the execution advances. A single bit flip is introduced when the computation has completed 50% of the execution. 49

3.4 Partial replication with three replicas. Replica 3 is suffering from SDC. 55

3.5 Reduction in Error with Signature Analysis (100% if all errors are removed, and 0% if there is no change) 62

3.6 Execution Times with Signature Analysis and Checkpointing on 64 nodes. 63

3.7 Experiment Result of Partial Replication Method and Partial Replication + Signature Analysis (including Checkpointing and Restart) on 32 nodes: Sobel and CG. Sobel replicates the last 40% of the execution while CG replicates the first 40%. 64

4.1 Relative Normalized Difference when execution is injected with single bit flip during different execution stage: CG 68

4.2 The value of residual of miniFE in the runtime with bit flipped in different bit ranges. The bit flip is inserted when the algorithm finished around 50% of the iteration. The residual value continues to reduce when no bit flip is introduced. In this case, no decrease in residual value can be seen as a signature. 69

4.3 The value of residual of CG in the runtime with bit flipped in different bit ranges. The residual value gradually reduces and converges relatively fast when no bit flip is introduced. The runtime takes longer to complete or even cannot complete under the impact of soft error. (Bitflip within range of 0-32 did not lead to any change. Presented bit flips are injected in around 20%, 40% and 60% of runtime, respectively). In this case, residual value does not decrease / takes long time to decrease is a signature. 70

4.4 Workflow of detection with off-line training method. Life cycle includes sampling stage, profiling stage, execution stage and completion. Application should recover from latest state if any of the model reports a potential soft error. 71
4.5 Results for on-line prediction (AID) on different problem sizes. The figure shows bit flips occurs in 5% interval (20-45, 55-80 is omitted due to the similarity to 50%). .......................................................... 76

4.6 Results for detection with off-line training on different problem sizes. The figure shows bit flips occurs in 5% interval (20-45, 55-80 is omitted due to the similarity to 50%). .......................................................... 78

4.7 Detection rate with different models in different execution stage. Each model shows its detection rate for bit injection in its covered execution range. Input sizes are: MiniFE 150*150, CG 6.2k * 6.2k, HPCCG 6.6k * 6.6k .......................................................... 80

4.8 Overhead shown as slowdown percentages (averaged across all input sizes) 81

4.9 Overhead of Model60 on different input sizes for CG. ......................... 82

4.10 Detection rate with different models in different execution stage on double flips. Each model shows its detection rate for bit injection in its covered execution range. Input sizes are: MiniFE 150*150, CG 6.2k * 6.2k, HPCCG 6.6k * 6.6k .......................................................... 84

4.11 Detection rate with double flips occurring in different 5% intervals (20%-45% and 55%-80% are omitted due to the similarity to the 50% case) . . . 84

4.12 Overall detection rates for Conjugated Gradient with bit flip injected in different execution stage with/without initial guess of the solution for sparse matrix. Training: JGD_SPG/EX3 to JGD_SPG/EX6, Input: JGD_SPG/EX1, JGD_SPG/EX2 .......................................................... 85

4.13 Overall detection rates for Conjugated Gradient with bit flip injected in different execution stage with/without initial guess of the solution for sparse matrix. Training: JGD_SPG/EX1 to JGD_SPG/EX4, Input: JGD_SPG/EX5, JGD_SPG/EX6 .......................................................... 86

4.14 Overall detection rates for Conjugated Gradient with bit flip injected in different execution stage with/without initial guess of the solution for sparse matrix. Training: JGD_SPG/EX Matrices, Input: Koutsovasilis/F2, Schenk/nlp-kkt80 .......................................................... 87

5.1 Example of Converting Force Kernel to MVM ................................. 95
5.2 Average Normalized Difference when injected with single bit flip during
different execution stage: miniMD (4k atoms, input range [-1.0, 1.0]) . . . . 101

5.3 Distribution of TP, FP and FN when bit flip is injected in different bit
ranges. Application: miniMD(4k,[-100,100]) and coMD(3.2k,[-100,100]). 103

5.4 Distribution of TP, FP and FN when bit flip is injected in different bit
ranges. Application: miniMD(2k,[-100,100]) and coMD(3.2k,[-1,1]). . . . 105

5.5 F-Score distribution when neighboring double bit flips are injected. Appli-
cation: miniMD(4k,[-100][100],Tolerance:1e-1), coMD(3.2k,[-100][100],Tolerance:1e-
2) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 105

5.6 F-Score distribution when random double bit flips are injected. Applica-
tion: miniMD(4k,[-100][100],Tolerance:1e-1), coMD(3.2k,[-100][100],Tolerance:1e-
2) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 106

5.7 Overhead of fault detection and recovery: miniMD . . . . . . . . . . . . . 107

5.8 Overhead of fault detection and recovery: coMD . . . . . . . . . . . . . 108

6.1 System Overview of Smart . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 112

6.2 Overview of Time and Space Sharing Mode . . . . . . . . . . . . . . . . . 113

6.3 Workflow of Smart with Fault Tolerance Supported . . . . . . . . . . . . . 119

6.4 Evalution of Overhead in Time Sharing Mode with Heat3D . . . . . . . . 126

6.5 Evalution of Overhead in Time Sharing Mode with Lulesh . . . . . . . . 127

6.6 Evalution of Overhead in Space Sharing Mode . . . . . . . . . . . . . . . . 128

6.7 Overhead shown as slowdown percentages (averaged across all input sizes) 129
List of Algorithms

1 Sample Indirection Array Code ............................................. 17
2 Parallelization of an Indirection Array-based Code ..................... 18
3 Asynchronous Uncoordinated Checkpointing for Indirection Array Based
   Code ................................................................. 22
4 Asynchronous Checkpointing Implementation for Jacobi and Sobel ...... 24
5 Asynchronous Checkpoint for MolDyn ..................................... 27
6 Signature Detection Method for Applications With Convergence ....... 51
7 Signature Detection Method for Molecular Interaction Applications ..... 53
8 Sample Main Loop of Sobel ............................................... 55
9 Sample Main Loop with Triple Replication ................................ 56
10 Example of training a 20% classifier model. ............................. 73
11 Example of runtime detection ............................................. 74
12 Main Kernel of MD application ........................................... 93
13 Force Kernel of MD application .......................................... 94
14 Smart Data Processing .................................................... 116
15 Smart-FR Processing with Fault Tolerance .............................. 121
List of Code Listings

2.1 Example of Irregular Reduction .................................................. 15
6.1 Histogram Processing in Smart ................................................... 114
List of Abbreviations

ABFT . . . . . . . . Algorithm Based Fault Tolerance

C/R . . . . . . . . Checkpoint and Restart

HPC . . . . . . . . High Performance Computing

I/O . . . . . . . . Input and Output

MD . . . . . . . . Molecular Dynamics

MPI . . . . . . . . Message Passing Interface

MTBF . . . . . . . Mean Time Between Failure

MTTF . . . . . . . Mean Time To Failure

MVM . . . . . . . . Matrix Vector Multiplication

Smart . . . . . . In-Situ Mapreduce Lite

SDC . . . . . . . . Silent Data Corruption
Chapter 1: Introduction

Our overall research work has been motivated by three trends in the recent years. First, in the recent decades, as High Performance Computing (HPC) continuously growing in its scale and MTBF consequently decreases, fault tolerance is drawing more attention from HPC communities. Looking forward to the larger scale machines in the near future, it is believed that machines are to experience more failures. Second, C/R is considered as the main approach to provide fault tolerance against both fail-stop and non fail-stop faults [35, 36, 40, 72]. Although C/R approach has shown its promising performance on fault resistance, the overhead of both checkpointing and restart could be non-trivial for many applications as the application scale grows. Third, soft errors quickly became one of the most difficult problems to solve in current days, because of its unpredictability and transiency. These errors often cause abnormal behaviour of the application (including application halts, infinite loops and etc.), or in worse case, they produce erroneous output without affecting the behavior of the application. Soft errors are believed to be more often as the system scale grows and the size of hardware decreases [10, 57].

Even though there are studies made on both C/R approach and soft errors to provide more robust fault tolerance to specific scientific applications for specific system configurations, compared to the current system scale and the scale we will obtain in the near future, existing solutions achieve will achieve lower performance when the system scale grows, or
when the configuration changes. Even more, there is no generalized solutions that support fault tolerance to most scientific applications.

Based on these observations, we believe that improving the fault resiliency or resistance for scientific applications is a crucial issue to resolve. Our purpose is to offer generalized fault tolerant solution to popular scientific applications to help the application stay resilient to both fail-stop and non fail-stop faults, with acceptable overhead to the execution time and system resource usage. We approach this by investing applications based on the error signatures and recover using asynchronous checkpoint and recovery. We approached an asynchronous checkpoint scheme to achieve better performance for checkpointing. We also studied the error signature to determine possible appearance of soft errors to perform a recovery process to avoid the error.

In this chapter, we would like to introduce the background knowledge about the fault tolerance methods, soft and hard errors, and the categories of applications that we are working with, including dense grid computation and irregular reductions. Then we will take an overview of our current work, which includes methodology of asynchronous uncoordinated application level checkpoint, signature detection based approach for soft errors/partial replications for iterative applications and algorithm based fault tolerance approach for molecular dynamics.

1.1 Overview of Fault Tolerance

This section introduces common fault tolerance solutions. We will discuss the overall fault tolerance solutions in high level, and then discuss them in detail in different categories.
1.1.1 Categories of Fault Tolerance Approaches

In Figure 1.1, we give a (simplified) overview of the space of solutions for HPC fault-tolerance. Besides checkpointing, another approach is algorithm-level fault-tolerance [8, 20,24,72], where an algorithm’s properties are exploited (typically to build-in redundancy). While this approach can overcome many of the overheads of general checkpointing, it has two key limitations: 1) as the name suggests, the solution is very specific to a particular algorithm, and 2) the fault-tolerant algorithm needs to be implemented by the programmer manually while developing the application.
Checkpointing itself can be system-level or application-level. In system-level checkpointing, the entire system state from the operating system’s view-point is saved, and moreover, checkpointing is initiated in an application-oblivious fashion. For message-passing distributed environments, there are two main families of system-level checkpointing protocols, which are the coordinated and uncoordinated checkpointing protocols [35, 40]. Coordinated checkpointing, which is the most popular approach in HPC systems today, involves processes coordinating to produce a consistent global system state, which consists of states of all processes and channels among the processes. It has the advantage of supporting simple recovery and efficient garbage collection, but the synchronization time for creating a consistent global system state and writing their checkpoints to the persistent storage could be high, and involves bursty writes on the I/O system, adversely impacting the application’s performance. Uncoordinated checkpointing is asynchronous, i.e. each process takes a checkpoint independently, but has to be combined with message logging to limit the number of processes to rollback in the event of a failure. Various message logging protocols have been proposed with different associated overheads of logging application messages [7].

Manual or programmer-controlled application-level checkpointing has been frequently used in HPC, especially when the programmers find the automated (system-level) checkpointing overheads unacceptable. The approach usually results in very low overheads, particularly because the programmers can determine: 1) the required state that needs to be checkpointed, at the point in the application where the checkpoint is taken, and 2) the point in the program to take the checkpoint (e.g. at the end of the time-step iteration), so that less (or no) coordination between the processes is needed, and the amount of state that needs to be stored is small. The disadvantage of this approach is that it puts significant additional
burden on the programmer (especially when one includes the restart logic). Moreover, the I/O involved in checkpointing code adds to the portability problem, and some of the I/O-related optimizations that are highly desirable for checkpointing might not be performed by the user.

1.1.2 Discussion on Fault Tolerance Solutions

C/R has become a dominant approach for providing fault tolerance for HPC in the past decades due to its robust performance and reliability. This approach periodically saves application data and system status in reliable storage. If a fault occurs, the system can recover its status from the latest checkpoint and continue the execution, instead of restating from the beginning. This approach is rather reliable as long as the storage is reliable. However, C/R is not suitable for all applications because of following three reasons: A) if the application suffers from relatively high error rate. In extreme situation, e.g. if MTBF is shorter then the C/R interval, then application will simply not execute but only restart. B) Involving large amount of I/O operations is rather expensive. Cost of reading and writing checkpoint is usually greater than people usually believe, which makes it more expensive to shorten the C/R intervals. This makes it become a tradeoff between the fault detection latency and C/R frequency. A system can checkpoint more frequently to achieve better fault detection latency and restart overhead but spending more resources in taking checkpoint, which is considered to be overhead for application execution. On the other hand, if the checkpoint is taken less frequently, when the system detects a failure, it has to recover from the state far away from current, which increase the computation cost. C) C/R can only handle fail-stop failures where a node completely halts and the failure can be easily detected. For recently being increasingly concerned transient errors (e.g. soft errors), applications
need other approaches to cover these situations. Due to the above observations, C/R ap-
proach will become a limitation for the future applications as the system and application
scale grows rapidly. Therefore, in the recent years, studies have been made to seek for a
possible optimization or even alternative solution to support fault tolerance for large-scale
applications.

One alternative is to enhance checkpoint is application level C/R, which is commonly
believed to be a faster checkpoint approach as it reduces the C/R overhead by reducing the
checkpoint size. It is reported that checkpoint size can be reduced up to 80% using the
application level checkpoint compared to system level.

However, these approaches still have limitations and heavily depend on application
and system status. Application level checkpoint relies on application level knowledge and
therefore needs the effort from application developer. Asynchronous checkpoint introduces
potential threat of cascade roll back effect.

As C/R approach brings many concerns, the community moved their attention towards
possible ABFT approach solutions, which can potentially solve problem during the runtime
with lower overhead but more complicated implementation. Huang et al. [47] proposed a
classical algorithm based fault tolerance approach for matrix multiplication in 1984, which
is now a fundamental theory for many studies. In this approach, an extra row and column
(which holds sum of rows and columns respectively) is added to matrices before comput-
ation so that the multiplication also generates a matrix with extra row and column. In
order to verify if the computation is mathematically correct, they only needed to check if
the last element of the matrix holds the sum of the elements in the matrix. Variations of
this checksum approach are quickly proposed for other linear solvers such as matrix vector
multiplications (MVM).
1.2 Soft Errors and Hard Errors

Figure 1.2 shows the fault models and their effects. Errors can be categorized into hard errors and soft errors based on how the error occurs.

1.2.1 Hard Errors

Hard errors refer to permanent and unrecoverable errors in the system. These errors can include hardware errors like disk errors or memory failures. Hard errors are usually caused by bad product offered by the manufacturer, and usually result in fail-stop situation. Due to the nature of the hard errors, these errors are relatively easy to detect, e.g. a node that does not respond to the system for a long time (i.e. the node is offline) might be suffering from hard errors. Therefore, it is relatively easier to perform the consecutive recovery process once such an error is detected.
1.2.2 Soft Errors

On the other hand, soft errors are commonly believed to be a worse error to handle with.

Soft errors are unpredictable events that cause bit flips in memory which produce errors in logic, and possibly affect the computation status [76]. Soft error is considered to be one of the most difficult problems to solve today. The reasons of soft error can be various, including packaging material, cosmetic radiation, voltage fluctuation and even the temperatures [10, 57]. As energy usage becoming more concern to the HPC systems, aggressive power management is applied to systems and consequently chips are expected to perform near-threshold operations more often today and in the future. Meanwhile, chips keep decreasing in size and are organized in even more compressed way in newer systems, which potentially causes more radiation noise. For such reasons, soft errors are occurring more frequently than commonly believed. It is also proved that potential impact from soft errors are certainly not negligible for larger applications that run in multi-chips and multi-cores [10]. Therefore, it’s now in urgent need to approach to a effective solution to the soft errors.

Looking closer into soft errors, the impacts of them can be categorized as follows: a) application executes and finishes normally (Successful execution). This may happen when soft errors occur in resources that is no longer used by the application, or is later overwritten by the application. In this case, the corrupted data is not consumed by the application hence performs no impact. b) application aborts because of erroneous computations, segmentation faults in memory and etc., caused by unexpected value change due to the soft error. c) application executes with undetectable errors. This is usually the most complicated situation to study. Soft errors may result in value change in application’s data, and
the corrupted data may happened to be consumed by the application. If this occurs, application will execute based on the wrong data and further causes more errors in results compared with those generated by applications with no soft errors. These errors are commonly notably huge and not acceptable in practice. At the same time, the corrupted data may influence the execution of the applications, making applications taking longer/shorter time to finish or even unable to finish in reasonable time. For example, a change of value in the condition of a loop may change execution status of the application. In all of these circumstances, data corruptions are not exposed to applications so that it is not detectable and normally consumed by the applications themselves. So that these data corruptions are usually called silent data corruption (SDC).

1.3 Dissertation Contributions

In this section, we give a brief introduction to the current progress on our research work, including three components.

1.3.1 Asynchronous Uncoordinated Application-level Checkpoint

Fault-tolerance has become an important topic in HPC today. In the last two decades, MPI has been the dominant programming model in computational sciences and there has been a significant amount of work on checkpointing for MPI. Most of the current approaches are based on system-level checkpointing, but the overheads of these approaches are becoming unacceptable.

In this work, we present an approach for significantly improving the efficiency of checkpointing and restart. We refer to our approach as the *asynchronous, uncoordinated, application-level* checkpointing approach. The idea is to exploit the communication structure to find locally finalized data, as well as each process’s contribution to globally finalized
data. Once such data has been identified, each process can proceed to checkpoint this data independently, i.e., make checkpointing uncoordinated. Furthermore, this data can now be checkpointed asynchronously, i.e., the computations and the I/O can be overlapped. We have applied this idea to five different applications, including two regular applications, two irregular applications involving indirection arrays, and one particle simulation application. We have compared our approach against synchronous application-level and system-level checkpointing, and have shown that we can reduce the slowdowns due to checkpointing by up to 90% by the asynchronous approach.

1.3.2 Signature based method and Partial Replication for Iterative Applications

With reducing feature sizes, there is a growing need for soft errors to be handled at the software level. This paper examines a different approach for handling soft errors at the software level. We focus on iterative scientific applications, including the applications involving convergence conditions, and molecular simulations. After studying the impact of bit flips on convergence and correctness of these applications, we propose the following method for improving accuracy of these applications in the presence of silent data corruptions. We develop signatures that detect the soft errors that can have the most negative impact on the applications. For iterative scientific applications with a convergence condition, the convergence variable itself can serve as a signature, since bit flips in high order bits can cause a sharp jump (as opposed to a reduction) in the values. Similarly, for molecular interaction applications, we develop a technique based on monitoring the displacement of positions and velocities. Finally, we identify that for certain applications, bit flips in certain part of the computation is a lot more likely to impact final results. We propose partial replication to help improve accuracy. In applying our approach for five scientific applications,
we find that our signature based method removes all infinite loops because of bit flips, reduces the error in the final results by up to 99%, and has less than 6% overhead (with an additional 24% overhead for checkpointing and restart). The reduction in error can be as high as 99.9% while using partial replication together with our signature analysis for two of the applications.

1.3.3 Algorithm based Fault Tolerance for Molecular Dynamic Applications

Previous works have stated the solutions to the applications that have specific patterns to converge, which can be used to detect the occurrence of soft errors. However, irregular applications such as molecular dynamics usually don't have such convergence criteria. Hence, alternative solution is needed for such applications to support fault tolerance. An alternative approach is ABFT, which detects and possibly recovers the errors within the algorithm level. In this way, the application achieves fault tolerance without expensive I/O cost.

We present an algorithm level solution to detect the soft errors in molecular dynamic applications. We construct a matrix to hold the intermediate data, and convert the main logic of force computation to a matrix vector multiplication. Then we could apply well-known ABFT algorithm on matrix vector product to detect potential soft errors in the computation. Once the soft error is detected, we propose two ways for recovery. 1) Recover in algorithm level. This process consumes larger memory compared to the next approach, while it avoids I/O from checkpoint and restart. 2) Recover from latest checkpoints. Since molecular dynamics are not memory intensive, the cost of I/O is relatively acceptable depends on the users need. Hence the application could recover from the latest checkpoint upon a failure.
1.3.4 Machine Learning based Fault Tolerance for Interative Convergent Applications

A variety of soft error handling methods have been proposed in the HPC community, including replicating computation, feature based runtime detection, algorithm-level protection and others. However, these methods are either relatively expensive or not sufficiently accurate or designed specific for certain applications.

For convergent iterative applications, we observe that their progression of values of the residual leaves a signature of SDC, which is specific to an application but independent of the input dataset size. Based on this observation, we explore a different approach to soft error detection, which involves machine learning technique for off-line training of an application with representative inputs, and on-line detection using the model, applied even to a different dataset. Our experimental evaluation shows that our method is low-cost and effective, and outperforms online detection.

1.3.5 Fault Tolerance Methodologies on Map-Reduce based In-Situ Analaysis

Recent studies on in-situ analysis can be broadly classified into application level algorithms and system level resource scheduling platforms. Smart [90], however, is proposed with a unique point of view to the in-situ problems, promisingly adopts Map-Reduce framework to in-situ analysis which places analytic implementations in the high level while hiding the scheduling complexities from the users. Due to its unique implementation, the error handling becomes complicated as both computation and analysis are placed in the same stage. Traditional fault handling methods are difficult to be employed as the simulation and consumption of the data is on-the-fly and data is not likely to be stored in a stable storage like off-line analysis. This indicates a potential information loss in the presence
of a node failure, which results in difficulties in retrieving data to continue the analysis. Performing analysis on simultaneously with computation also complicates traditional C/R since the checkpoint needs not only protect the simulation, but also the analytical procedure, if a better efficiency is desired. Therefore, if no further optimization is taken, in the worst case, both simulation and analysis procedure has to restart from the beginning when hard failures occur. These problems do not only exist on the Smart, but also applies to other Map-Reduce like in-situ analysis implementations.

Considering the threats stated above, it is clearly beneficial to have a fault tolerant framework with high-level interface, automatically collects critical data to checkpoint and supports restarting from any checkpoints to continue simulation/analysis, while transparentizing complicated checkpoint synchronization in simulation and analysis procedures. We employ Smart as the carrier of this framework, and successfully made Smart capable of surviving node failures, without adding much complications to existing APIs. We achieve this by tracking the reduction object in Smart, ensuring that every reduction objects are processed by the analysis codes.

We have evaluated both the functionality and efficiency of Smart-FR with massive data by using two scientific simulations and analytical tasks on both multi-core and many-core clusters. We evaluated the overhead that fault resiliency brings to the Smart as well as the detailed distribution of the overheads.

1.4 Outline

The rest of this dissertation is organized as follows: Chapter 2 introduces a methodology of asynchronous uncoordinated application level checkpoint. Chapter 3 presents signature based method and partial replication to handle soft errors in iterative applications.
Chapter 4 introduces algorithm level fault tolerance for molecular dynamic applications. Chapter 5 discusses automated soft error detection using machine learning mechanism for convergent iterative applications. Chapter 6 introduces a fault tolerant framework for in-situ analysis, especially for Map-Reduce based analysis. Chapter 7 lists the related work from other researchers and compares ours with them, and Chapter 8 concludes this dissertation.
Chapter 2: A Methodology for Application-Level Asynchronous
Checkpointing for MPI Applications

This chapter introduces an uncoordinated asynchronous checkpointing strategy for iterative applications. We first illustrate the idea to take checkpoint by partition, then show how the details of implementation to different class of applications, and finally reports the experiment results.

2.1 Communication Patterns

Before we start, let us introduce the communication patterns that we have studied in this work. The communication patterns we will introduce include *Irregular Reductions* and *Stencil Computations*.

2.1.1 Irregular Reductions

**Listing 2.1 Example of Irregular Reduction**

```c
Real X(numNodes), Y (numEdges); ! Node&Edge Data arrays
Integer IA(numEdges,2); ! Indirection array
Real RA(numNodes); ! Reduction array

for (i=1; i<numEdges; i++) {
    RA(IA(i,1))=RA(IA(i,1))op(Y(i)op X(IA(i,1))op X(IA(i,2)));
    RA(IA(i,2))=RA(IA(i,2))op(Y(i)op X(IA(i,1))op X(IA(i,2)));
}
```
Irregular reductions can also be referred to as unstructured grids in Berkeley dwarfs. Unlike structured grids, nodes in unstructured grids are connected by edges explicitly, since the connectivity of nodes cannot be determined by node positions (coordinates). The code shown above (Listing 2.1) shows a typical irregular reduction loop. Each loop updates two elements in reduction array \( RA \) using commutative and associative operations \( op \), through an indirection array (edges) \( IA \). Because each node may be connected by multiple edges, the update of the elements in \( RA \) involves reductions. The related data in an irregular reduction loop could be categorized as read only input data and reduction result. In the above example, the input data are arrays \( X, Y \) and \( IA \), while the reduction result is \( RA \). We use molecular dynamic applications as the example of the irregular reduction applications.

2.1.2 Stencil Computations

Stencil applications involve computations of updating each element in the input based on the values of its neighboring elements and itself. The input for stencil applications is usually a structured grid or matrix with two or higher dimensions. Stencil applications are usually quite simple and easy to implement in serial code, however, optimizations to the parallel code are complicated for programmers. Therefore, maintaining fault tolerance regardless of the implementation is of importance. Jacobi, Sobel are typical stencil computation applications.

2.2 Main Approach

In this section, we will describe the main underlying ideas of our approach, which is the asynchronous uncoordinated checkpointing at the application-level.

The motivation behind this scheme is as follows. As we had stated earlier, coordinated checkpointing requires the system to communicate across all the nodes to ensure
that the checkpoints are consistent. All the nodes start checkpointing at the same time and finish the procedure at the same time. Before nodes can start writing the checkpoint, all communications and computations should be finished. Similarly, only when all nodes have finished writing checkpoints, any of the nodes can start any computation or communication. Coordinated checkpointing is used not only for system-level checkpointing, but also for application-level checkpointing [49], since it simplifies recovery. However, coordinated checkpointing, at both system and application-level, has several disadvantages. First, additional synchronization can add a slowdown to the application. Second, all I/O for checkpointing is performed within a specific phase, and such bursty I/O can degrade performance.

As we had explained above, an alternative to coordinated checkpointing is *uncoordinated checkpointing* [35], though it has the cost of message logging and potential need for rollback while recovery. Because of the complexity of logic of message logging and rollback during recovery, to the best of our knowledge, uncoordinated approach has not been used for application-level checkpointing.

We have developed an approach that exploits properties of the application to perform uncoordinated checkpointing without any need for message logging. To explain the idea, we use the following code segment, where computations are performed on an unstructured grid.

**Algorithm 1: Sample Indirection Array Code**

1. for $i = 0 \rightarrow \text{num}_{-}\text{edges}$ do
2. \hspace{1em} $X[IA[i,1]] = X[IA[i, 1]] + Y[i]$;
3. \hspace{1em} $X[IA[i,2]] = X[IA[i, 2]] + Y[i]$;
4. end for
Suppose the above loop is executed multiple times within a time-step loop. Note that the array \(IA\) is based on edges, whereas the array \(X\) is based on the nodes. Every iteration of the loop is updating the two end points of an edge (the nodes).

Now, suppose the code is parallelized by diving the edges among the processes. With such partitioning, values at nodes that are end-points of edges belonging to multiple partitions will be updated by computations on multiple processes. In practice, this is managed by allocating \textit{ghost cells}, which are then communicated and reduced at the end of the computation. The outline of the parallel code will be as follows:

\begin{algorithm}
\caption{Parallelization of an Indirection Array-based Code}
\begin{algorithmic}[1]
\State \textbf{for} \(i = 0 \rightarrow \text{num\_edges}/\text{num\_procs}\) \textbf{do}
\State \(X[IA[i,1]] = X[IA[i,1]] + Y[i]\);
\State \(X[IA[i,2]] = X[IA[i,2]] + Y[i]\);
\State \textbf{end for}
\State \textbf{Scatter}(X);
\end{algorithmic}
\end{algorithm}

The \texttt{Scatter(X)} step involves communication of values of ghost cells, followed by reduction (addition) to finalize the contents of the array \(X\) for the next iteration. Now, the state of the application at the end of the iteration is the content of the array \(X\), which in turn is the collection of the values of the portion of \(X\) that is owned by each process. After all communications finish, all processes can output their component of \(X\) onto stable storage. This will be the typical coordinated checkpointing approach. However, as we stated earlier, this can be extremely expensive, because of the need for coordination and bursty I/O.

The approach we are using is different, and is based on recognition of the reduction property of the application. The output of an iteration is divided into two components. The first part represents the part of the output space that could be accessed/updated by more
than one process, and is thus referred to as the *global reduction space*. The second part represents the remaining part of the output space that only needs to be accessed/updated by its host process, referred to as the *local reduction space*. Now, the output space for the entire computation is composed of the each process’s contribution to the global reduction object and its local reduction object. Based on this information, $X$ can be reconstructed during recovery, and the next iteration of the computation can be started.

The advantage of this approach is that processes do not need to wait for the communication step to perform checkpointing. Instead, checkpointing can be performed as a set of values have been finalized. Furthermore, it has the advantage of facilitating asynchronous I/O for checkpointing (without large memory overheads), which we will discuss next.

2.2.1 Asynchronous Checkpointing

Most checkpointing is performed using synchronous I/O operations, which has the disadvantage that we have to wait for a checkpoint to complete and then only we can resume computations. This wastes CPU cycles unnecessarily. Suppose we want to use asynchronous checkpointing with the coordinated approach, in which the values of the array $X$ are output at the end of each iteration. Because elements of $X$ are to be updated at the start of the next iteration, we need to copy elements before asynchronous operations are used. This will add to the memory requirements as well as introduce the cost of memory copy operations.

In comparison, use of the uncoordinated approach can allow asynchronous operations without any or large additional memory requirements. Particularly, during one iteration of the time-step loop, we write the checkpoint with multiple asynchronous operations. Ideally, the data can be written into checkpoints whenever it is prepared (computed), though
making system calls too frequently also results in poor performance. Thus, we can perform asynchronous I/O writes whenever a portion of the checkpoint (we call the portion a trunk) is ready. The implementation of this approach depends upon the nature of the computations in the application. Thus, we discuss it separately for applications involving regular and irregular accesses.

**Implementation for Regular Accesses**

For simplicity, let us suppose the checkpoint is performed on a linear memory space (e.g. checkpointing a one-dimensional array). It is relatively straightforward to implement asynchronous uncoordinated checkpointing in this case. We simply write checkpoints whenever a portion of the memory is ready. We logically partition the memory into several trunks, and each process computes data and writes it into the memory linearly. When all values within a trunk has been finalized, we perform an asynchronous I/O operation. The contents being written do not need to be copied into another buffer, since subsequent computations will be updating a different portion of the array. Essentially, one trunk will be written while the next trunk is being computed. When the process finishes computing next trunk, one of these two situations can arise: 1) the previous trunk has finished writing, in which case the process can directly trigger the asynchronous write operation on the next trunk, or alternatively, 2) the previous trunk has not finished writing - in this case, we can either wait for the previous trunk to finish, or we can just simply write this trunk to another checkpoint file (to avoid corruption of buffers). If we choose the latter option, then we have to carefully manage checkpointing files. If we choose the former, since the write operation is already processed during the computation, it is unlikely that we will have to wait for a
long time. Thus, we can block on the previous operation, and perform the next write opera-
tion and resume computations only after the former finishes. One exception is with respect
to the last trunk, which must be written using a blocking operation.

**Implementation for Irregular Accesses**

If the memory is not accessed regularly, there are a distinct set of implementation chal-
lenges. Particularly, since the memory access is not linear, we cannot define a trunk directly
from the memory. Hence, we create a trunk based on a *threshold*. With the application’s
knowledge, during the computation, we are able to know that a particularly element is being
updated for the last time in the current time-step iteration. If so, we can push the data into
the trunk immediately after the data is computed. If the trunk is full (size of data stored
exceeds the threshold), then we can write the trunk asynchronously. To make sure that
the trunk is not overwritten during asynchronous operations, we typically need to allocate
memory for two trunks.

We explain this idea further using the example we were discussing earlier in this sec-
tion. As we can see, the array $X$ is accessed irregularly by the indirection array $IA$. As a
result, we can not partition the array $X$ into trunks directly. However, with the application’s
knowledge, we can know whether there is duplicated value in the array $IA$ or not. If the
array $IA$ holds values that are all different, every update is the final update. Hence, we can
copy the value to the trunk directly, right after the element of the array $X$ is updated. If there
are duplicated values, then some of the updates are not final, and in this case, a check shall
be done before pushing this a value to the trunk. This is shown through the pseudo-code
below.

The push operation copies the value to the current available trunk (i.e. the trunk on
which I/O is not being performed). The trunk size is the current data size that is stored in
Algorithm 3: Asynchronous Uncoordinated Checkpointing for Indirection Array Based Code

1. for $i = 0 \rightarrow num\_edges$ do
2. \hspace{1em} $X[IA[i, 1]] = X[IA[i, 1]] + Y[i]$; //update
3. \hspace{1em} $X[IA[i, 2]] = X[IA[i, 2]] + Y[i]$; //update
4. \hspace{1em} if $X[IA[i, 1]] = final$ then
5. \hspace{2em} trunk.push($X[IA[i,1]]$);
6. \hspace{1em} end if
7. \hspace{1em} if $X[IA[i, 2]] = final$ then
8. \hspace{2em} trunk.push($X[IA[i,2]]$);
9. \hspace{1em} end if
10. \hspace{1em} if trunk.size() $\geq$ trunk.threshold() then
11. \hspace{2em} trunk.write();
12. \hspace{2em} trunk.switch();
13. \hspace{1em} end if
14. \hspace{1em} end for
15. \hspace{1em} trunk.wait();

the trunk. When the number of elements in the trunk meets the threshold, then we can start to write this trunk into the checkpoint file. While this trunk is being written, we will have to switch to a different memory structure to receive newly updated values. One important parameter here is the trunk threshold. A very small size can increase the number of write operations, which can be an overhead. A large size implies larger memory, and moreover, the last write operation, which has to be blocking in nature, takes longer.

2.3 Implementing Asynchronous Uncoordinated Checkpointing on Applications

We now describe how we have applied our approach on a number of applications. We have chosen a set of applications and kernels that involve very distinct communication patterns. This has allowed us to investigate the issues in efficiently and correctly implementing asynchronous uncoordinated checkpointing.
2.3.1 Dense Grid Computations

In dense grid computations, memory accesses are regular with respect to the memory layout. Moreover, same locations are not updated multiple times within a single loop. Thus, we could follow the memory access pattern to partition the memory into trunks. As shown in the Figure 2.1, suppose the data needed for checkpoint is stored in a two-dimensional memory space, and the computation is performed from left to right in each iteration. In these application, no data that has been computed is re-written later in the same iteration. Therefore, it is possible to perform the write immediately after the value is computed. In the checkpoint iteration, we can simply partition (logically) the grid into several trunks, and start computation (Figure 2.1(a)). If the computation of a trunk is finished, then we initiate an asynchronous write on the trunk to write checkpoints (Figure 2.1(b)), and the process
continues the computations. When the computation of the next trunk is finished, then as discussed in the previous section, if the previous trunk write operation is finished, then we can continue to write the current trunk. Otherwise, we can wait, or write to a different checkpoint file. We follow this scheme to compute and checkpoint all the trunks (Figure 2.1(c)). When the process finishes the computation for the last trunk, we will have to wait for writing this trunk before we go to next iteration (Figure 2.1(d)).

In this paper, we have used Jacobi and Sobel as applications with dense grid computations for implementing our approach. They both have similar pattern of memory accesses, which is seen in other dense grid computations. The versions of the applications we used operate on a two-dimensional array, and update a point with the values of its neighbors, repeating the process for every point in the array.

The outline of the code with asynchronous checkpointing looks as follows:

**Algorithm 4: Asynchronous Checkpointing Implementation for Jacobi and Sobel**

1. \( j \leftarrow 0; \)
2. for \( i = 0 \rightarrow n \) do
3. update \( x[i] \);
4. if \( i - j = \text{trunk.size} \) then
5. /*wait for previous write to finish*/
6. wait();
7. /*asynchronously write a trunk*/
8. aiowrite(from \( x[j] \) to \( x[i] \));
9. \( j \leftarrow i; \)
10. end if
11. end for
12. /*wait for last write*/
13. wait();

Because we can directly checkpoint the data from memory for every trunk size each time, therefore, no extra memory space is needed. We perform an asynchronous write
whenever the procedure computes the data for a trunk size (line 4). The `wait()` step in line 6 involves waiting for the asynchronous write operation of the previous trunk to finish, before we trigger a new asynchronous write operation in line 8. When the computation of current iteration finishes, we perform the last asynchronous write for the last trunk and wait for this operation to finish (line 13).

### 2.3.2 Indirection Array-Based Computations

Next, we have experimented with two applications that involve irregular computations based on unstructured arrays. *Euler* [62] is an application that simulates an unstructured grid, and indirection array is used to store the end-points of edges. *Moldyn* is a molecular dynamics application that is parallelized based on partitioning of molecules, and uses indirection arrays to capture interactions between molecules [48]. Both these applications have a similar structure of computation.

We implemented our approach on the MPI version of these applications, where parallelization involves partitioning the space into several partitions, and each node performs computation on multiple partitions. For example, in the Moldyn application, we update every molecule’s information using its neighbors’ information. The computation is performed over different partitions and during the computation of each partition, the process needs the data of its own molecules and the neighboring partitions’ molecules. In these two applications, the memory accesses are very irregular, and it is difficult to partition a certain part of the memory to use for asynchronous writes.

A more straight-forward application of our idea will be to create a trunk with a threshold and push the data with last updated value to the trunk. However, this requires good
knowledge of the application structure to determine which update is the last update for every element. When the data structures and data accesses are complicated, it is extremely hard to perform such an implementation.

We, therefore, use an object-level perspective, instead of the element-level perspective to apply our checkpointing idea. Obviously, deciding when an entity has been last updated is easier when the number of entities is smaller. More specifically, our implementation involves creating a relatively large trunk to hold the data. The trunk now accepts various objects instead of scalar items. During the computation, when an object of a data structure is computed with the final value, then we push the object into the trunk. Similarly, when the data in the trunk size has reached the threshold, we push the trunk into checkpoints. During the writing procedure, processes continue doing their computation for other objects, and push them to currently available trunk(s). And, when the data size in the trunk again reaches the threshold, we write this trunk to the checkpoint. This is repeated until the entire application state has been checkpointed. The idea is shown in Figure 2.2. This figure describes a process checkpointing objects. In the figure, there are two trunks in use. Trunk-2 has the objects A, B, and C stored, which is being asynchronously written into the checkpoint file. At the same time, the Trunk-1 is available and accepting the objects that have been finalized. Whenever the process finishes computing an object, then we push the object into the Trunk-1, until this trunk is also full (hits its threshold). This process is repeated till all objects have been finalized and checkpointed.

The following pseudo-code is an implementation of this idea for the Moldyn application.
Algorithm 5: Asynchronous Checkpoint for MolDyn

1. for $i = 0 \rightarrow max\ iteration$ do
2. /*initialization and some computations*/
3. for each partition do
4. local computation;
5. end for
6. for each partition do
7. send/recv data with neighbor partition;
8. end for
9. for each partition do
10. update partition;
11. /*potential checkpoint here*/
12. trunk.push(partition);
13. if $size \geq threshold$ then
14. trunk.write();
15. end if
16. end for
17. /*other computations*/
18. end for

In this implementation, the data is stored in a data structure $partition$, which includes molecules’ coordinates, velocity, force, and other information. During the 1st loop, computations are done using the local information from within each partition. Then, each partition prepares and sends the data that is needed by its neighbor(s), and similarly, receives the information it needs from others. With the data received, the process finally goes into the third loop to finish the final computation. An element-level implementation will be very hard, since we will need to study the partitions, see which element’s final update is in which iteration, and keep track of the order in which elements are pushed to trunks and checkpoints. Without keeping track of the order, it is impossible to perform recovery correctly.

Working at the object-level can help simplify the implementation. It is easy to observe that the third loop will be the final update for each partition. And, the access mode for the
third loop is relatively linear, in the sense that we are just traversing all the partitions to update the data. Since the third step is also very time consuming, we can focus on the third loop to take our checkpoints. To minimize the cost of checkpointing, we also checkpoint only the data necessary for the recovery, i.e. the information attached to the molecules only. On recovery, user could get the molecule’s initial data, perform partition across the space, and re-allocate partitions to the processors to restart.
2.3.3 MiniMD

MiniMD is the most complex application we have implemented our approach on. MiniMD is one of mini-applications from the Mantevo suite from Sandia National Laboratories. The goal of the Mantevo project is to create representative applications (a few thousand lines of code) of full-scale scientific applications. Particularly, MiniMD is a smaller version of a large-scale software LAMMPS. MiniMD is different from the Moldyn application we considered above, in the sense that it is based on partitioning of the space across the processes.

The challenge with MiniMD was that the data needed for checkpoints is spread in multiple data structures, and they were computed at different places in the program. The core data for this application is stored in the atom object, which includes the coordinates, forces, and velocities information. The atom data structure is not completely updated until

---

1Please see https://software.sandia.gov/mantevo

---
the very last loop (which updates final values of velocities), and the last loop is not the most compute-intensive component. If we partition the computations within the last loop and then perform asynchronous writes, the writing time will greatly exceed the computation time, and the benefit of asynchronous operations will be lost.

Looking over the implementation, we realize that the final update of velocity occurs during the last loop in an iteration of the time-step loop, final update of force occurs after the `force.compute()` function and the final update of the coordinates happens after exchanging a communication step. Thus, we can checkpoint them separately right after they are computed, e.g., we push `atom.x` to the trunk and start writing to checkpoint right after exchanging information with the neighbors.

### 2.3.4 Discussion

The implementation approaches for the three different kind of applications are quite different. However, the main idea remains the same: find when any elements or objects have been finalized, and then seek opportunity to overlap their output with ongoing computation. Two main constraints are applied, however - first, we do not output any value that has not been finalized for the particular iteration of the time-step loop. Second, we make sure, through global synchronization, that all trunks have been finalized before a time-step iteration ends.

An important point needs to be made regarding the approach for irregular applications. We use an intermediate structure, referred to as an `object`, before writing to the trunk. This introduces additional copy operations, but works well because of the relatively low cost (as compared to disk accesses) for memory operations. We also use only one trunk to overlap I/O operations and computation. If checkpointing the trunk is finished before the
computation for the next object is finished, then this trunk could be re-used to store the new object. On the other hand, if the trunk is still being used when the next object is ready, the process could wait.

2.4 Experimental Results

In the previous section, we have described how our approach has been implemented for five different applications. This section presents a detailed experimental study, where we study the overheads of asynchronous checkpointing for these applications, and the benefits as compared to synchronous checkpointing and system-level checkpointing with MPI-BLCR.

All experiments are performed on a cluster where each node has two quad-core 2.53 GHz Intel(R) Xeon(R) processors, with 12 GB RAM, executing RedHat Enterprise Linux Server release 6.1, and Gigabit ethernet as the interconnect. We evaluated the total execution time for each applications running on 4, 8, 16, and 32 nodes.

2.4.1 Regular Applications

Figures 2.3 and 2.4 shows the results for Jacobi and Sobel applications, respectively. For Jacobi, we used an input matrix with $8,000 \times 8,000$ size, and took checkpoints every 40 iterations, while executing the application for 200 iterations in total. The overhead of synchronous checkpointing varies between 17% and 100%, with relative slowdown increasing quickly as the number of nodes increases. This is because of greater I/O contention with increasing number of nodes. The slowdown with asynchronous checkpointing is under 10% for 4, 8, and 16 nodes, and around 18% for 32 nodes. Overall, the reduction in overhead from use of the asynchronous approach is up to 82%.

In checkpointing there is a trade-off between the cost of checkpointing (which increases with increasing frequency of checkpointing) and the repeated computation in case of a
Figure 2.3: Comparing Execution Without Checkpointing, With Synchronous Checkpointing and Asynchronous Checkpointing: Jacobi

Figure 2.4: Comparing Execution Without Checkpointing, With Synchronous Checkpointing and Asynchronous Checkpointing: Sobel
failure (which increases with decreasing frequency of checkpointing). Our experiments with Sobel were designed to see the benefits of asynchronous approach when checkpointing frequency is higher, specifically, every 15 iterations of the time-step loop. Again, we can observe that the checkpointing overhead is increasing as the number of nodes increases, though absolute overheads are somewhat higher than those for Jacobi, because of higher checkpointing frequency (but not at the same ratio as increase in checkpointing frequency, because of more computation in each iteration). Our asynchronous checkpointing approach reduces the overheads by between 35% and most 44%.

2.4.2 Irregular Applications

Figures 2.5 and 2.6 show the results from Euler and Moldyn, two applications that involve indirection arrays. We used 427 MB input size for Euler and 222 MB for Moldyn. For Euler, we took checkpoints every 60 iterations, while executing the application for
Figure 2.6: Comparing Execution Without Checkpointing, With Synchronous Checkpointing and Asynchronous Checkpointing: Moldyn

200 iterations. Similarly to the regular applications, we can observe that the checkpoint overhead is increasing as the number of nodes increases. As we can see, the synchronous checkpointing overhead varies between 50% and 400%. However, our approach overlaps this writing overhead with the computation time, reducing the I/O contention. As is shown in Figure 2.5, our approach could reduce the overhead for checkpointing by almost 50% in all cases.

The results are quite different for Moldyn. In this application, the size of the checkpoints is significantly smaller as compared to the amount of computation. Unlike other applications, the overheads decrease with increasing number of nodes, because the checkpoint size from each node becomes very small as the number of nodes increases. Asynchronous approach can almost completely overlap computation and I/O, except for the last trunk. Overall, the overhead of synchronous checkpointing is reduced by up to 90% by the asynchronous approach.
Figure 2.7: Comparing Execution Without Checkpointing, With Synchronous Checkpointing and Asynchronous Checkpointing: MiniMD

Figure 2.7 shows the results from miniMD. Similarly to other applications, the overall checkpointing overhead increases as the number of nodes increases. We can see that the overhead of synchronous writing checkpoint varies from 12% to 107%, while our approach has overhead between 6% to 74%. Our approach could reduce the checkpointing overhead by at least 30% compared with synchronous checkpointing.

2.4.3 Comparison with System Level Checkpointing

In all of the above experiments, we compared our approach against synchronous application-level checkpointing. However, the most common practice for MPI applications is system-level checkpointing. While it is to be expected that application-level checkpointing will be faster than system-level checkpointing, we conducted a series of experiments to quantify the difference.
Figure 2.8 shows the result of the experiment. We run each application on 8, 16, and 32 nodes, and took checkpointing as frequently as we did in the previous set of experiments. For the first two applications, system-level checkpointing is not much slower than synchronous application-level checkpointing, though our asynchronous approach is still significantly faster. The reason for only a modest difference between synchronous application-level and system-level checkpointing is these are simple kernels, without too many arrays, and the amount of state to be saved by application-level and system-level checkpointing.
is quite similar. For more complex applications, the advantage of application level checkpointing is more clear. For example, in Euler, the only data that we checkpoint is atoms’ and edges’ information, while system-level checkpointing stores all the data structure created with the intermediate values. Similar pattern occurs with Moldyn. In fact, for Moldyn on 32 nodes, slowdown with our approach is negligible, whereas the slowdown with system-level checkpointing is nearly 150%.

2.5 Summary

Driven by the need for drastically improving the efficiency of checkpointing and restart, this paper has presented an asynchronous, uncoordinated, application-level checkpointing approach. The idea is to exploit the communication structure to find locally finalized data, as well as each process’s contribution to globally finalized data. Using this information, each process can proceed to checkpoint this data independently, i.e., make checkpointing uncoordinated. Furthermore, the computations and the I/O can be overlapped, i.e. the required data can now be checkpointed asynchronously. Applying this idea to five different applications, we have shown a large improvement over synchronous coordinated checkpointing approach.

This work motivates the need for developing static analysis techniques to automatically apply our method on existing MPI applications. This will be a topic for future investigation.
Chapter 3: A Practical Approach for Handling Soft Errors in Iterative Applications

This chapter introduces methodology to detect soft errors in iterative applications. We first demonstrate a set of experimental results of impact from soft errors. The result shows indicate that bit flip occurring in higher order bits results into more significant errors, while bit flip in lower order bits results into trivial data changes. In iterative applications, the error effect also depends on when the bit flip occurs. We introduce the approaches to detect and handle the soft errors based on these observation.

3.1 A Motivating Study

In this section, we summarize the results from a study to understand the impact of soft errors on iterative scientific computations.

3.1.1 Experiment Design

We developed a soft error injector based on PIN [76], which is a dynamic instrumentation tool for binary files. Given the features of PIN-tools, our soft error injector traces applications’ heap usage, and injects soft errors. These errors can be injected at any place, and at any time, which can be controlled by us. The fault-injection involves flipping a bit
value from 0 to 1 (or vise versa) in the memory, with the goal of simulating a soft error that can occur in practice.

We evaluated the impact of soft errors on five applications, which are, Jacobi, Sobel, Conjugated Gradient (CG), Molecular Dynamic (MolDyn), and MiniMD. Jacobi is a commonly used kernel that performs 5-point stencil-based computation on a two-dimensional structured mesh. Sobel is an edge detection application which convolves two \( 3 \times 3 \) masks to the input matrix with two-dimensional points and performs a 9-point stencil-based computation. Conjugate Gradient is a popular method that solves the linear algebra problem \( A \times x = b \). Moldyn is a molecular dynamics application that is parallelized based on partitioning of molecules and captures interactions between the molecules. MiniMD is a mini-app developed by the Mantevo project at Sandia National Labs. The goal of the Mantevo project is to create representative mini versions of full-scale scientific applications. Particularly, MiniMD is a smaller version of a large-scale software LAMMPS.

Among the five applications, Jacobi, Sobel and CG have specific convergence conditions. They have a regular accesses pattern and do not involve any use of pointers. MolDyn and MiniMD are different implementations of molecular dynamics, and simply simulate movement of molecules over time, without a convergence condition. Both of them involve more indirection-based accesses, which imply more pointer dereferences. The configuration used for each application is presented in Table 3.1. All experiments are performed on a cluster where each node has two quad-core 2.53 GHz Intel(R) Xeon(R) processors, with 12 GB RAM, executing RedHat Enterprise Linux Server release 6.1, and Gigabit ethernet as the interconnect.

Hardware designers have stated that 1 bit-flip per 24 hours per core can be expected to the high-end of the expected range of bit errors in the conceivable future [64]. Consistent
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Dataset (per core)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jacobi</td>
<td>$500 \times 500$ grid points</td>
</tr>
<tr>
<td>Sobel</td>
<td>$4096 \times 4096$ grid points</td>
</tr>
<tr>
<td>CG</td>
<td>$8192 \times 8192$ Input Matrix</td>
</tr>
<tr>
<td>MolDyn</td>
<td>4000 atoms</td>
</tr>
<tr>
<td>miniMD</td>
<td>10800 atoms</td>
</tr>
</tbody>
</table>

Table 3.1: Applications and datasets used in experiments

with this, we experiment with one bit flip during the execution of a single application (which executes on up to 64 nodes for up to several minutes), noting that the chances of more than 1 bit flip during this period are extremely rare. To study the impact of bit errors, we varied both the time of the bit flip, as well as the bit-range where it occurs. For instance, we inject the single bit flip at different times, i.e., within first 20% of the iterations of a normal execution, next 20%, and so on. Similarly, the bit flip is injected in different bit-ranges of the target floating point number - i.e., the least significant 4 bits, next 4 bits, and so on. Because of the way floating point (and double precision) numbers are represented in memory, clearly the relative change in the value of a number is different for bit flips in different bits. For instance, in a double precision variable, if any of the bits between 56th and 63th bit is flipped, the exponent component of the value gets modified.

Table 3.2 shows the difference between the normal output and the output that is computed when a bit flip occurs on Sobel. The values in the table correspond to the average normalized difference between the two outputs for different bit-range and time combinations. The average normalized difference between two n-sized datasets $A$ and $A'$ is computed by this formula: $\sqrt{\frac{\sum_{i=1}^{n} (A_i - A'_i)^2}{n}}$. Results show that bit flips do not have any noticeable
Table 3.2: Average normalized difference between the outputs of normal execution and execution with a single bit flip, depending on the location (bit range) and time (iteration) of the bit flip (Sobel).

<table>
<thead>
<tr>
<th>Bit range</th>
<th>Iteration where the bit flip occurs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0~20%</td>
</tr>
<tr>
<td>0~3</td>
<td>0</td>
</tr>
<tr>
<td>4~7</td>
<td>0</td>
</tr>
<tr>
<td>8~11</td>
<td>0</td>
</tr>
<tr>
<td>12~15</td>
<td>0</td>
</tr>
<tr>
<td>16~19</td>
<td>0</td>
</tr>
<tr>
<td>20~23</td>
<td>0</td>
</tr>
<tr>
<td>24~27</td>
<td>0</td>
</tr>
<tr>
<td>28~31</td>
<td>0</td>
</tr>
<tr>
<td>32~35</td>
<td>0</td>
</tr>
<tr>
<td>36~39</td>
<td>0</td>
</tr>
<tr>
<td>40~43</td>
<td>0</td>
</tr>
<tr>
<td>44~47</td>
<td>0</td>
</tr>
<tr>
<td>48~51</td>
<td>0</td>
</tr>
<tr>
<td>52~55</td>
<td>0</td>
</tr>
<tr>
<td>56~59</td>
<td>0.02412725</td>
</tr>
<tr>
<td>60~63</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 3.2: Average normalized difference between the outputs of normal execution and execution with a single bit flip, depending on the location (bit range) and time (iteration) of the bit flip (Sobel).
impact on the output of the application when they occur in lower bit ranges. Specifically, we see 0 difference when the bit flip occurs in 0-40th bit (in practice, it is less than $1.0e^{-15}$ or lower, but a user analyzing the output is extremely unlikely to notice such a difference). The iterative steps of these computations, where values of neighboring elements are averaged, seem to minimize the impact of a single bit flip over time. However, starting from the bit flips in 40th and higher order bits, the magnitude of the difference increases when the bit flip location is changed to higher order bits. When the bit flip occurs in the bit range in 60-63th, there is a decrease in the output differences. This is mainly due to the fact that the highest bit stands for the sign, so that the flip on this bit simply reverses the sign of the value, and this change seems to get minimized over the iterations.

Another pattern we can observe from this table is that even for the bit flips in the same bit range, the later a bit flip occurs, the worse its impact on the final output is. This is likely because with fewer remaining iterations, it is harder for the change to be smoothened over time. In comparison, if the value changes in the earlier phases of the execution, its impact can be averaged out by the underlying algorithm.

The results reported in Table 3.2 are for cases when the execution terminates normally, which is actually always the case when the bit flip is in 0-55th bit. However, higher order bit flips can introduce abnormal situations like infinite loops. This is because higher bits stand for the exponent, and making the changes on these bits can cause instability and lack of convergence. Table 3.3 shows the frequency of infinite loops in Sobel caused by single bit flips (in practice, we terminate the execution if it takes much longer than the normal number of iterations to execute).

We repeated the same evaluation on Jacobi and CG. The results, presented in Table 3.4, are similar to Sobel in the sense that a single bit flip in a lower-order bit does not cause
Table 3.3: Frequency of executions leading to an infinite loop because of a single bit flip: Sobel

<table>
<thead>
<tr>
<th>BitRange</th>
<th>0~20%</th>
<th>20~40%</th>
<th>40~60%</th>
<th>60~80%</th>
<th>80~100%</th>
</tr>
</thead>
<tbody>
<tr>
<td>0~55</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>56~59</td>
<td>20%</td>
<td>60%</td>
<td>50%</td>
<td>30%</td>
<td>80%</td>
</tr>
<tr>
<td>60~63</td>
<td>20%</td>
<td>50%</td>
<td>45%</td>
<td>50%</td>
<td>50%</td>
</tr>
</tbody>
</table>

any difference in the final output, whereas the differences are noticable or even significant with a bit flip in the higher order bit. However, in contrast to the scenarios where we vary the bit flip location, these applications react differently when the time that the bit flip occurs is changed. For Jacobi, there is no strong relationship between timing of bit flip and the difference in the results, while for CG the earlier the bit flip, the more significant the difference in the final result is. We attribute this to the nature of the computations, i.e., there is more averaging of values over time in some applications, and a greater compounding impact for others. As in Sobel, infinite loops also appear both in Jacobi and CG. The frequency of infinite loops (caused by single bit flip) on these applications is shown in Table 3.5.

Finally, we have experimented both with MolDyn and MiniMD, which involve pointer operations and indirect accesses. The first observation is that unlike the previous three applications, bit flips can cause a crash (a segmentation fault or a similar error) for these two applications. These are primarily caused by pointers getting corrupted. Figure 3.1 shows the frequency of executions leading to crashes due to a single bit flip. As we can see, the likelihood of crashes is largely independent of the iteration where the bit flip occurs.
<table>
<thead>
<tr>
<th>BitRange</th>
<th>Jacobi</th>
<th>Conjugated Gradient</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Iteration where Bit Flip Occurs</td>
<td>0~20%</td>
</tr>
<tr>
<td>0~27</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>28~31</td>
<td>0.00001</td>
<td>0.00001</td>
</tr>
<tr>
<td>32~35</td>
<td>0.000830762</td>
<td>0.0000782842</td>
</tr>
<tr>
<td>36~39</td>
<td>0.00311893</td>
<td>0.000290578</td>
</tr>
<tr>
<td>40~41</td>
<td>0.00126861</td>
<td>0.00106568</td>
</tr>
<tr>
<td>44~47</td>
<td>0.00600346</td>
<td>0.0034079</td>
</tr>
<tr>
<td>48~51</td>
<td>0.0278655</td>
<td>0.0396418</td>
</tr>
<tr>
<td>56~59</td>
<td>4816.29</td>
<td>10036.6</td>
</tr>
<tr>
<td>60~63</td>
<td>0.332153</td>
<td>0.571175</td>
</tr>
</tbody>
</table>

Table 3.4: Average normalized difference between normal execution and execution with single bit flip: Jacobi and CG
Table 3.5: Frequency of executions leading to an infinite loop because of a single Bit flip: Jacobi and CG

Figure 3.1: Frequency of executions leading to crashes because of a single bit flip (reported for crashes happened in all bit ranges): Moldyn and MiniMD

Also, though not shown in the figure, it is also largely independent of the location of the bit which flips.

For bit flips where the application does not crash, the differences in the final results are shown in Table 3.6. In the table, we omit the bit ranges over which the difference is either
3.6 Methodology

This section describes the techniques that we propose to mitigate the major effects caused by soft errors on execution. More general approaches for detection of SDC involve either replicating the execution twice and verify the correctness by comparing the results
Figure 3.2: Control flow of an application incorporating signature detection, checkpointing and recovery steps.

generated by the two replicas, or create three replicas and use voting. These techniques obviously consume a large amount of memory. We are proposing techniques with much lower overheads, though they are not as general. Our approach involves a signature analysis based detection scheme with recovery using periodic checkpoints, possibly coupled together with partial replication.
Specifically, our approach is to detect the application level *signature* left by a soft error and start a recovery procedure upon its detection. The approach is obviously dependent upon feasibility of identifying such a signature. Figure 3.2 shows the proposed control flow of an application using signature analysis to detect and recover soft errors. First, at each or every few iterations, a signature detection mechanism is run on the convergence variable to detect the soft error occurrence. If no anomaly is detected in this phase, the application is considered as executing normally, so the current error free state can be checkpointed to the stable storage and the next iteration can be started. Otherwise, detection of a soft error signature suggests that a possible SDC has occurred, so the application needs to be restarted from the latest checkpoint.

### 3.2.1 Bit Flip Signature for Applications with a Convergence Criteria

Since high order bit flips have a much more significant impact on execution compared to low order bit flips, we prioritize the protection of high order bits. It turns out that the convergence criteria associated with iterative solvers can be used to detect high order bit flips. In iterative scientific applications with a convergence criteria, values associated with grid point values undergo large changes for the first few iterations and the rate of change slowly reduces as the execution advances. The speed of change in grid point values between two consecutive iterations is captured by a single or a set of convergence variables, e.g. \( \text{diffnorm} \) in Jacobi and Sobel, \( \text{residual} \) of solution matrix in CG.

For these applications, soft error detection can be efficiently performed by monitoring the convergence variable only, since an abnormal change in the convergence variable can be considered as a soft error signature. Under normal execution, the convergence variable keeps decreasing – though the pace of the decrease would be slower as the data gets into
Figure 3.3: The change in the convergence variable (diffnorm) of Sobel as the execution advances. A single bit flip is introduced when the computation has completed 50% of the execution.

more stable status – until it finally satisfies the convergence criteria. On the contrary, a high order bit flip on a grid point value transitively results in a significant change in the previous and current value of the convergence variable and consequently breaks its pattern of change. As long as the change is significant enough, there will be a detectable signature introduced. Furthermore, it turns out that if the bit flip results in an unexpected difference in the output, there must be a noticeable difference in the convergence variable as well. Similarly, note that if the bit flip results in an infinite loop, it must be because the convergence variable does not decrease.

Figure 3.3 shows the impact of a single bit flip on the convergence variable (diffnorm) of Sobel. For this experiment, we created separate scenarios in each of which the bit flip is inserted into a different bit range (0-51, 52-55, 56-59 or 60-63), when the computation has completed 50% of the execution (50%th iteration). The figure shows that the average
normalized difference (diffnorm) decreases stably until the execution reaches the iteration that the failure is injected. When the bit flip occurs, diffnorm makes an unexpected “jump”. Such an anomaly can be easily detected as long as a copy of the previous diffnorm is recorded. Note that the bit flip in bit range 0-51 is not detected, since bit flips in this range do not create any major (noticeable) impact on the execution as Table 3.2 suggested.

Algorithm 6 illustrates our signature detection and recovery scheme for iterative convergent applications. At the beginning of each iteration of the main loop, the current value of the convergence variable (diffnorm) is compared to its previous value (previous_diffnorm). An observed increase is considered as a signature of the bit flip and the execution is restarted from the last checkpoint. Before moving to the next iteration, the value of diffnorm is updated by applying the normalized difference formula on the current \( A' \) and previous application data \( A \).

### 3.2.2 Signature for Molecular Interaction Applications

In contrast to convergent applications that we considered so far, determining a soft error signature can be difficult for applications that do not have such a convergence property. To demonstrate the feasibility of designing signatures of bit flips for applications that do not have a convergence condition, we studied two molecular simulation applications. These applications simulate the movements of atoms in a three-dimensional space for a predefined number of iterations. At each iteration, a new position and velocity vector is calculated for an atom with respect to its interactions with the closest neighbor atoms. To establish a soft error signature, we make use of the application specific knowledge that the change in the position and velocity of an atom should not be dramatic since the set of neighbors for an atom generally remains identical between two consecutive iterations. Consequently,
Algorithm 6: Signature Detection Method for Applications With Convergence

1. Computations before the main Loop
2. ...
3. while \( \text{diffnorm} \leq \text{minimum threshold} \) do
4.  /* check for signature */
5.  if \( \text{diffnorm} \geq \text{previous_diffnorm} \) then
6.   Perform recovery process;
7.  else
8.  /* record diffnorm */
9.  previous_diffnorm ← diffnorm;
10. end if
11. ...
12. Perform (Periodically) Checkpoint;
13. Update \( A[][] \rightarrow A'[][] \);
14. ...
15. /* update diffnorm */
16. diffnorm = 0
17. for \( i = 0 \rightarrow N \) do
18.   for \( j = 0 \rightarrow M \) do
19.     diffnorm += \((A'[i][j]-A[i][j])*(A'[i][j]-A[i][j])\);
20.   end for
21. end for
22. \( A[][] \leftarrow A'[][] \);
23. diffnorm = sqrt(diffnorm);
24. end while
25. ...
26. Computations after the main loop
27. ...

51
throughout execution, we trace the changes in atoms’ coordinates and velocities and allow reasonable updates that do not change these properties by more than a predefined threshold. This way, the high order bit flips can be efficiently detected, since they will lead to a tremendous change, exceeding the threshold value, on any of the atom properties. The threshold value can be designed to dynamically adjusting as the computation progresses.

As a more concrete example, suppose that all atoms’ velocities are changing at most by \( + \pm 100 \) units during a couple of consecutive iterations. This observation suggests that the initial threshold value could be set to 100, and any following velocity change within 100 should be considered as a regular effect of normal execution. However, if a high order bit flip occurs, the effected value will likely show a very large difference, exceeding 100 threshold, compared to its previous value. Note that, although the pace of change is preserved for the majority of execution, atom properties might indeed go through a large change \( (+ - 1000) \) sometimes as part of the normal execution. To prevent such false positive scenarios, an application can be monitored for several initial iterations and the threshold value can be increased further, if such an exponential change is detected even under normal execution.

Algorithm 7 shows the signature detection scheme for MiniMD. In the algorithm, we show how the bit flips on atoms’ force property can be detected. In practice, we apply this approach to other properties like velocity and position for better accuracy. In each iteration, the maximum amount of difference and maximum amount of factor of increment are recorded in \( \textit{diffmax} \) and \( \textit{incremax} \), respectively. In next iteration, if the change on a force value is within \( \textit{diffmax} \) and \( \textit{incremax} \), the update is considered error free. As shown, during detection, a more flexible allowance scheme can be used by increasing \( \textit{diffmax} \) and \( \textit{incremax} \) variables with an offset value. However, if the change exceeds
Algorithm 7: Signature Detection Method for Molecular Interaction Applications

1. Computations before the main Loop
2. ...
3. for iter = 0 → N do
4.   ...
5.   Perform (periodically) Checkpoint;
6.   ...
7.   /* check for signature of Force */
8.   for i = 0 → num_atoms do
9.     Update atom[i].force → atom[i].force'
10.    atomdiff← abs(atom[i].force'-atom[i].force);
11.    if atomdiff≥diffmax+offset then
12.       Sig1← true
13.     end if
14.    if atomdiff/atom[i].force≥incremax+offset then
15.       Sig2← true
16.     end if
17.    if Sig1 or Sig2 then
18.       Perform recovery process;
19.     end if
20.   end for /*end of signature detection
21.   /* update diffmax and incremax */
22.   for i = 0 → num_atoms do
23.     atomdiff← abs(atom[i].force'-atom[i].force);
24.     diffmax←Max(atomdiff, diffmax);
25.     incremax←Max(atomdiff/atom[i].force, incremax);
26.     atom[i].force←atom[i].force'
27.   end for
28. end for
29. ...
30. Computations after the main loop
31. ...
the allowed limit, it is considered as a soft error signature and the execution is rolled back to the last known error free state.

**Recovery:** To facilitate recovery, we save application state periodically to stable storage and restart the computation from the most recent checkpoint once a soft error is detected. Clearly, the cost of these checkpoints is a significant factor for the overall costs of the applications. We use application level checkpointing in our implementation and give the programmer more control on the application data that is to be checkpointed.

### 3.2.3 Partial Replication

In applications with convergence property, there is a close relation between the impact of a soft error and the time that it occurs during execution. For example, as the experiment on Sobel in Section 3.1 has shown, high order bit flips are relatively tolerable, if they happen at the first 60% of the execution. Therefore, instead of an expensive full replication scheme that protects the entire execution from the beginning to termination, a partial replication approach can be efficiently employed for a small portion of execution that is more sensitive to soft errors.

Our approach for a replication factor of two can be summarized as follows. When the compute routine finishes more than certain amount of iterations and enters the critical stages, the partial replication is triggered. Then, all data and defined operations on it are replicated to a replica compute task. Each replica performs computation using its own data and generates its own result. The system performs periodic cross checking to monitor and compare the execution results from each replica. If the results from two replicas are different, a soft error occurrence is detected and application has to restart to produce the
correct results. Note that, here, using one more replica (a total of three) and a simple voting scheme make it even possible to find the correct result without the need for a restart.

Figure 3.4 shows a sample control flow of partial replication scheme using three replicas. Once more critical stages of the application are reached, we keep three copies of the computation and perform them separately. Periodically, we perform communication among three replicas and find if there is any difference on produced results. In this figure, replica 3 is detected as having a different result from the other two. Hence, replica 3 is considered as infected by a SDC and the generated result is treated as wrong.

Algorithm 8: Sample Main Loop of Sobel
1. for \( i = 0 \rightarrow n \) do
2.   update data[];
3. end for
Algorithm 9: Sample Main Loop with Triple Replication

1. \( \text{START} \leftarrow \text{false} \);
2. \( \text{for } i = 0 \rightarrow n \text{ do} \)
3. \( \text{if } i \geq \text{Replication}_\text{Start} \&\& i \leq \text{Replication}_\text{End} \text{ then} \)
4. \( \text{if } \neg \text{START} \text{ then} \)
5. \( \text{replicate data[]} \text{ to data}_\text{copy1}[]; \)
6. \( \text{replicate data[]} \text{ to data}_\text{copy2}[]; \)
7. \( \text{START} \leftarrow \text{true}; \)
8. \( \text{end if} \)
9. \( \text{if } \text{START} \text{ then} \)
10. \( \text{update data}_\text{copy1}[]; \)
11. \( \text{update data}_\text{copy2}[]; \)
12. \( \text{update data[]} ; \)
13. \( \text{compare data[]}, \text{data}_\text{copy1}[], \text{data}_\text{copy2}[]; \)
14. \( \text{if difference found then} \)
15. \( \text{copy back majority result} \)
16. \( \text{continue;} \)
17. \( \text{end if} \)
18. \( \text{end if} \)
19. \( \text{else} \)
20. \( \text{update data[]} ; \)
21. \( \text{end if} \)
22. \( \text{end for} \)

For better illustration, consider the loop in Algorithm 8 as the main loop that is to be protected after a certain amount of execution. The algorithm updates \( \text{data[]} \) for \( n \) many iterations. With user defined threshold of starting iteration for replication, the loop can be re-implemented as shown in Algorithm 9.

As we can see from the implementation, if the execution reaches the threshold, it is considered to be entering soft error critical part of the execution and the algorithm triggers the replication. We replicate entire \( \text{data}[] \) array and computation (update progress). Therefore along with the original \( \text{data}[] \), we maintain triple computation towards \( \text{data}[] \). At the end of each iteration, a comparison between \( \text{data}[] \) and all of its replicas (\( \text{data}_\text{copy1}[] \) and
are performed, so as to determine if there was a soft error occurred. If soft error has occurred, one of \texttt{data}[] and its replicas should differ from the others. In that case, the process keeps the majority results, copies the result to the replica that is different and continues the execution.

3.3 Experimental Results

We designed a series of experiments with three goals. First, we evaluate the effectiveness of signature analysis based method for detection of errors (together with checkpointing based recovery). Second, we wanted to examine further improvements from partial replicas. Finally, we evaluate the overheads of checkpointing and signature analysis. The set of applications and the datasets used are the same as used in the motivating study and previously summarized in Table 3.1. In all our experiments, a single bit flip was introduced during the execution, and we conducted experiments varying the bit range and the iteration range of the injection. Each application was checkpointed after every 20% of its execution.

3.3.1 Effectiveness of Signature Analysis Method

Applications with a Convergence Criteria

We first applied our signature detection method to Sobel, Jacobi, and CG. Based on the earlier results, we know that only bit flips in 48-63 bits for Sobel and Jacobi, and bit flips in 40-63 bits for CG consistently cause noticeable difference in the final results. Therefore, we applied our signature analysis method for bit flips in these ranges. Application of signature analysis on CG is somewhat different from Sobel and Jacobi. This is because the convergence variable does not strictly decrease in a monotonic way during the normal execution. In this case, instead of detecting immediate \textit{jump} as the bit error signature, we set a window for convergence variable change and keep studying the averaged change within
<table>
<thead>
<tr>
<th>Jacobi</th>
<th>BitRange</th>
<th>40~43</th>
<th>44~47</th>
<th>48~51</th>
<th>52~55</th>
<th>56~59</th>
<th>60~63</th>
</tr>
</thead>
<tbody>
<tr>
<td>Percentage of Undetected Bit Flips</td>
<td>-</td>
<td>-</td>
<td>95.2%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td></td>
</tr>
<tr>
<td>Sobel</td>
<td>BitRange</td>
<td>40~43</td>
<td>44~47</td>
<td>48~51</td>
<td>52~55</td>
<td>56~59</td>
<td>60~63</td>
</tr>
<tr>
<td>Percentage of Undetected Bit Flips</td>
<td>-</td>
<td>-</td>
<td>73.6%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td></td>
</tr>
<tr>
<td>Conjugate Gradient</td>
<td>BitRange</td>
<td>40~43</td>
<td>44~47</td>
<td>48~51</td>
<td>52~55</td>
<td>56~59</td>
<td>60~63</td>
</tr>
<tr>
<td>Percentage of Undetected Bit Flips</td>
<td>78%</td>
<td>67%</td>
<td>44%</td>
<td>14.9%</td>
<td>10.4%</td>
<td>2%</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.7: Percentage of Undetected Bit Flips: Jacobi, Sobel and CG

<table>
<thead>
<tr>
<th>BitRange</th>
<th>Iteration where Bit Flip Occurs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0~20%</td>
</tr>
<tr>
<td>0~39</td>
<td>0</td>
</tr>
<tr>
<td>40~43</td>
<td>0</td>
</tr>
<tr>
<td>44~47</td>
<td>0</td>
</tr>
<tr>
<td>48~51</td>
<td>0</td>
</tr>
<tr>
<td>52~55</td>
<td>0</td>
</tr>
<tr>
<td>56~59</td>
<td>0</td>
</tr>
<tr>
<td>60~63</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 3.8: Average Normalized Difference between Normal Execution and Execution with Single Bit Flip + Signature Analysis (Sobel)

the window. Since the average value of the convergence variable should be decreasing over time in a normal execution, an increase in the average value over the window signifies a data corruption.

Tabel 3.7 shows the percentage of bit flips that are not detected using our approach. The results from Sobel and Jacobi show that when the bit flip occurs in bit range from 48 to 51, we missed a large fraction of them. However, every single bit flip that occurred 52nd bit and onwards was detected by our method for both Sobel and Jacobi. The results from CG
<table>
<thead>
<tr>
<th>BitRange</th>
<th>0~20%</th>
<th>20~40%</th>
<th>40~60%</th>
<th>60~80%</th>
<th>80~100%</th>
</tr>
</thead>
<tbody>
<tr>
<td>0~27</td>
<td>0</td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>28~31</td>
<td>0.00001</td>
<td>0.0001</td>
<td>0.00001</td>
<td>0</td>
<td>0.00001</td>
</tr>
<tr>
<td>32~35</td>
<td>0.000830762</td>
<td>0.000782842</td>
<td>0.0000897468</td>
<td>0.0000641421</td>
<td>0.00004</td>
</tr>
<tr>
<td>36~39</td>
<td>0.00311893</td>
<td>0.000290578</td>
<td>0.000358156</td>
<td>0.000377787</td>
<td>0.000349726</td>
</tr>
<tr>
<td>40~41</td>
<td>0.00126861</td>
<td>0.00106568</td>
<td>0.00114831</td>
<td>0.00137716</td>
<td>0.00158508</td>
</tr>
<tr>
<td>44~47</td>
<td>0.00600346</td>
<td>0.0034079</td>
<td>0.00535458</td>
<td>0.00623356</td>
<td>0.00571476</td>
</tr>
<tr>
<td>48~51</td>
<td>0.0278655</td>
<td>0.0396418</td>
<td>0.0565583</td>
<td>0.0714002</td>
<td>0.0418322</td>
</tr>
<tr>
<td>52~55</td>
<td>0</td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>56~59</td>
<td>0</td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>60~63</td>
<td>0</td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 3.9: Average Normalized Difference between Normal Execution and Execution with Single Bit Flip + Signature Analysis (Jacobi)

<table>
<thead>
<tr>
<th>BitRange</th>
<th>0~20%</th>
<th>20~40%</th>
<th>40~60%</th>
<th>60~80%</th>
<th>80~100%</th>
</tr>
</thead>
<tbody>
<tr>
<td>0~23</td>
<td>0</td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>24~27</td>
<td>1.41E-07</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>28~31</td>
<td>2.00E-07</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>32~35</td>
<td>2.00E-07</td>
<td>1.00E-07</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>36~39</td>
<td>0.038274</td>
<td>1.00E-07</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>40~43</td>
<td>0.00047584</td>
<td>1.29E-06</td>
<td>3.46E-07</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>44~47</td>
<td>1.03E-07</td>
<td>0.0018748</td>
<td>8.61E-07</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>48~51</td>
<td>1.19E-06</td>
<td>0.00022345</td>
<td>0.000582422</td>
<td>2.00E-07</td>
<td>0</td>
</tr>
<tr>
<td>52~55</td>
<td>0.061175</td>
<td>0.0010013</td>
<td>0.000041234</td>
<td>0.00207243</td>
<td>0</td>
</tr>
<tr>
<td>56~59</td>
<td>0</td>
<td>0</td>
<td>1.00E-06</td>
<td>0.000382426</td>
<td>0</td>
</tr>
<tr>
<td>60~63</td>
<td>0</td>
<td>0</td>
<td>0.000311375</td>
<td>0.00017884</td>
<td>1.00E-06</td>
</tr>
</tbody>
</table>

Table 3.10: Average Normalized Difference between Normal Execution and Execution With a Single Bit Flip + Signature Analysis (CG)
Table 3.11: Percentage of Undetected Bit Flips: MolDyn and MiniMD

<table>
<thead>
<tr>
<th>BitRange</th>
<th>0~47</th>
<th>48~51</th>
<th>52~55</th>
<th>56~59</th>
<th>60~63</th>
</tr>
</thead>
<tbody>
<tr>
<td>MolDyn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Percentage of Undetected Bit Flips</td>
<td>-</td>
<td>10.5%</td>
<td>3%</td>
<td>5%</td>
<td>8%</td>
</tr>
<tr>
<td>MiniMD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Percentage of Undetected Bit Flips</td>
<td>-</td>
<td>68%</td>
<td>43%</td>
<td>2%</td>
<td>1%</td>
</tr>
</tbody>
</table>

have a larger variance – even some of the bit flips from highest bits could not be detected, though there is still a trend of improving efficacy of the signature method with increasing order of the bits.

Next, we focus on the overall accuracy of the final results with our method – i.e., introduction of a single bit flip, followed by signature analysis, and recovery from the last checkpoint if our signature analysis method can detect the bit flip, and comparing them against a normal execution. These results are reported in Tables 3.8, 3.9, and 3.10. For the bit ranges where all bit flips were detected, clearly the execution is identical to the normal execution. In other cases, the final outputs are not the same as in normal execution, though the errors are almost always small. In comparing the errors against the errors shown in Section 3.1, we see a large decrease in errors for all applications. This shows that even for a given bit range, the bit flips that can cause a more significant difference in the final results are detected by our method.

Another important observation from our experiments (not reported in any of the tables) is that there were no infinite loops, for either of the three applications, after our applying our signature detection method.
### Table 3.12: Average Normalized Difference between Output with Normal Execution and Output Single Bit Flip + Signature Analysis (MolDyn and MiniMD)

<table>
<thead>
<tr>
<th>BitRange</th>
<th>Iteration where Bit Flip Occurs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0~20%</td>
</tr>
<tr>
<td>MolDyn</td>
<td></td>
</tr>
<tr>
<td>0~40</td>
<td>0</td>
</tr>
<tr>
<td>41~43</td>
<td>0.0000830762</td>
</tr>
<tr>
<td>44~47</td>
<td>0.031893</td>
</tr>
<tr>
<td>48~51</td>
<td>1.0e-6</td>
</tr>
<tr>
<td>52~55</td>
<td>0</td>
</tr>
<tr>
<td>56~59</td>
<td>0</td>
</tr>
<tr>
<td>60~63</td>
<td>1.0e-8</td>
</tr>
<tr>
<td>MiniMD</td>
<td></td>
</tr>
<tr>
<td>0~40</td>
<td>0</td>
</tr>
<tr>
<td>41~43</td>
<td>0.00012248</td>
</tr>
<tr>
<td>44~47</td>
<td>0.0173003</td>
</tr>
<tr>
<td>48~51</td>
<td>0.03682</td>
</tr>
<tr>
<td>52~55</td>
<td>0.00532</td>
</tr>
<tr>
<td>56~59</td>
<td>1e-9</td>
</tr>
<tr>
<td>60~63</td>
<td>0</td>
</tr>
</tbody>
</table>

Application with Specific Signatures

MiniMD and Moldyn do not involve a specific convergence condition – as a result, the signature analysis methods used are different. Table 3.11 shows what fraction of bit flips can be detected by our methods. As we can see, our approach cannot detect bit flips in bit range from 0 to 48 for both applications. From bits 49 and higher, almost of the bit flips are detected for MolDyn, but the approach is effective for MiniMD from 56th bit onwards only. Table 3.12 shows the accuracy after performing signature analysis based detection and recovery based on checkpointing. By comparing against the Table 3.6, we can see that bit flips that have the largest impact on the results are being detected.
One important point to note is that our signature analysis cannot avoid application crashes, which tend to be quite frequent for these two applications (because of pointer dereferences). However, since we are performing checkpointing to deal with bit flips that are detected by our signature analysis method, a similar recovery can be applied if there is a crash because of a segmentation fault.

**Overall Reduction in Errors**

The overall reduction in error by using signature analysis method is shown in Figure 3.5. The reduction in error reported here is computed as follows. We first aggregate the average normalized difference across different cases with the introduction of a single bit flip. Next, we do the same calculation when a single bit flip is introduced, but signature detection method and recovery is applied. The percentage reduction in the second aggregated value over the first one is reported as the reduction in error.
Figure 3.6: Execution Times with Signature Analysis and Checkpointing on 64 nodes.

Our results show that except for MiniMD, all other applications have more than 99% reduction in error. This is because all bit flips causing a numerically large difference are detected and corrected. MiniMD gets only a 95% improvement because it has more bit flips in lower order bits that are not detected and still cause a noticeable difference in the final results. Overall, we can see that our method is quite effective in achieving final results that are either identical or very close to normal execution results, despite an undetected bit flip being introduced.

3.3.2 Overheads of Signature Analysis Method

We now focus on the overall costs of our method. The overhead of signature analysis is from checkpointing, analyzing/checking signatures, and possible recovery from the last checkpoint when a bit flip occurs and is detected by the signature method. In all experiments in this section, we used 64 nodes in the cluster.

Figure 3.6 shows the execution times with our method. We can observe that the overhead of our method is relatively small if the application involves small checkpoints, which
is the case for Jacobi, CG, and Moldyn. The slowdown with our method can get as low as 15% of the execution time, because the cost of taking a checkpoint and recovering from it is small. However, for applications like Sobel and MiniMD, the computation is relatively fast compared to the checkpoints, so the overheads are relatively higher. Still, compared to other methods replicating the execution by a factor of 2 or 3, our overheads are still very low.

### 3.3.3 Accuracy and Overheads with Partial Replication

We applied partial replication method on Sobel and CG. These two applications were chosen because our experiments in Section 3.1 demonstrated that bit flips in the late and early stages, respectively, have a larger impact on the final results. As a result, we replicated last 40% of the execution for Sobel and first 40% of the execution for CG (both applications keep three replicas).

Figure 3.7 (a) shows the reduction in error with the partial replication method, first used independently, and next with signature analysis being applied to the remaining 60% of the
iterations. We see at least 99.5% reduction in the errors for both applications just from the use of partial replication. When combined with signature analysis, we achieve even higher accuracy. In case of Sobel, there were no errors at all after the two methods are combined (i.e, 100% improvement in accuracy), whereas CG is able to achieve 99.9% reduction in error over an execution where a bit flip occurs and is not corrected.

Figure 3.7 (b) shows the overhead of our methods compared to normal execution, again for partial replication alone and then partial replication working in conjunction with signature analysis. As we can see that partial (triple) replication for 40% of execution creates around 80% of overhead to both applications, as one would expect. When combined with signature analysis, Sobel’s overhead increases noticeably due to the overhead of checkpointing, while inexpensive checkpointing for CG adds only a modest additional overhead.

### 3.4 Summary

This chapter introduced application level methodologies for improving resilience against soft errors. Our signature analysis based detection with checkpointing based recovery has been driven by the observation that high order bit flips can very negatively impact execution, but can also be easily detected. Specifically, we have developed signatures for iterative applications with a convergence criteria, and for molecular simulations. We have also proposed partial replication as a candidate approach for certain applications.

Detailed experimentation with five applications has shown the effectiveness of our methods. On one hand, we can reduce the errors by up to 99% using just signature based detection and checkpointing based recovery, with an overhead that is less than 30% on the average. Combining partial replication with signature analysis can also further improve accuracy, though with more substantial overheads.
Chapter 4: Automated Soft Error Detection for Iterative Convergent Applications Using Offline Training

We introduced general method of signature detection in Chapter 3, which has best performance on monotonically convergent applications. In this chapter, we discuss a machine learning based soft error detecting approach for iterative non-monotonically convergent applications. To this class of applications, we observe that their progression of values of the residual leaves a signature of SDC, which is specific to an application but independent of the input dataset size. Based on this observation, we explore a different approach to soft error detection, which involves machine learning technique for off-line training of an application with representative inputs, and on-line detection using the model, applied even to a different dataset.

4.1 Overview of the Problem

Before showing our approach utilizing the machine learning techniques, we give a quick overview of the class of soft errors we are dealing with, and the signature detection method in the previous work [61] that we will improve on.

To establish the scope of our work in this chapter, we observe that the studies conducted by others [9, 57, 60] as well as similar experiments we have conducted and show that soft errors can result into the applications to 1) finishes normally; or 2) aborts due to error
memory errors; or 3) continue and finish with undetected error and output incorrect results, which is also known as SDC.

In this chapter, we focus on the last type of soft errors. This is because the first type does not produce erroneous results and the second type can be simply recovered from traditional fault tolerance approaches like C/R. To put the results evaluating our method in context, we introduce Figure 4.1 to show how single bit flips affects the correctness for Conjugate Gradient (CG). The reported errors averages of the normalized difference between the correct output and the outputs with a single bit flip injection. We randomly flipped in one of the bits for each bit range and for each execution stage (first 20% of execution, the next 20%, and so on). We repeated the process for 10 times, and reports the averaged value of all successful executions as the results in the figure.

It is observed the overall impact is greater, when the bit flip occurs in a higher order bit. The overall difference nearly reaches order of 3 when the bit flip occurs in bit range of 32th and higher, and may reach order of 9 when it goes to bit range 56th and higher (specifically, in 56th to 63th bits and 0% to 20% of the execution). Thus, even a single bit flip could bring a huge negative impact to the algorithm correctness.

In several previous efforts [11, 61], silent data corruption has been related to the value of residuals in iterative applications. In many direct solvers, the residuals have a monotonic behavior, which gets altered when a bit-flip occurs. This allows for a simple detection of soft-errors. However, many other convergent and iterative applications, the residual do not have a strict monotonic behavior. Yet, it turns out that soft errors leave a signature on the progression of values of residuals.
Figure 4.1: Relative Normalized Difference when execution is injected with single bit flip during different execution stage: CG

For the motivating study, we experimented with a DOE mini-app called MiniFE\(^2\). In our experiments, bit-flips were randomly injected on a variable in the heap, and were randomly applied to any of the 64 bits, and we monitored the value of residual. Figure 4.2 shows that the application still shows an obvious signature for a bit flip injection - the residual value no longer decreases and stays the same when a bit flip is injected, which suggests that the application stopped converging. Similarly, Figure 4.3 shows the behavior when the bit-flip is injected into the CG application, which is another iterative solver where the residual is not strictly monotonic. We can see that if a bit flip is introduced, the application either takes longer time to converge (for bit range 0 - 31), or it does not converge anymore (for bit range 32 - 63).

\(^2\)Please see http://www.nersc.gov/users/computational-systems/cori/nersc-8-procurement/trinity-nersc-8-rfp/nersc-8-trinity-benchmarks/minife/
Figure 4.2: The value of residual of miniFE in the runtime with bit flipped in different bit ranges. The bit flip is inserted when the algorithm finished around 50% of the iteration. The residual value continues to reduce when no bit flip is introduced. In this case, no decrease in residual value can be seen as a signature.

It is also observed that applications reacts differently to the bit range that the bit flip occurs. In double precision, higher order bits may include exponent bits or high order fraction bits that significantly affect the value, and lower order bits are less likely to affect the value much. This is shown in Figure 4.3, when the bit injection happens in bit range 0-31, although with some extra iterations, the application still converges. However, when the bit flip is in the higher order bits, the residual value grows more rapidly into exceptional range, making the algorithm not being able to converge.

MiniFE, on the other hand, is more vulnerable to soft errors. As shown in Figure 4.2, even low order bit flips (bit range 0-32) will break the algorithm and stop the application from further convergence. This may due to the different tolerance ability of the algorithm’s mathematical property. We know that some minor single point errors could be averaged in
the future iterations, and finally be self-corrected by some algorithms (e.g., CG, bit range 0-32), but other algorithms may not be able to average the error, or not able to survive the error at the same rate. In this experiment, it is shown that miniFE has lower tolerance to soft error than CG.

4.2 Framework Design

Given our observations on applications like MiniFE and CG, our goal is to create a signature, possibly specific to an application, but independent of the input dataset, which can help detect an SDC.
Our main idea is as follows: Given an application, we run it with a set of *sample* inputs and collect the values of the residual, with and without random bit-flips. We refer to it as the *sampling stage*. As the next step, this data is utilized in a classifier algorithm for training a machine learning model – this is the *training stage*. Now, the application can be made available to its users with a built-in classifier that can detect SDCs, even if the application is running with different input dataset from the one it was trained on. Thus, during *execution stage*, the application regularly calls the model to verify if SDC has occurred. If the model indicates a potential soft error, a recovery method is triggered. Figure 4.4 illustrates the idea, whereas the details of each of the phases are given below.

**Sampling Stage:** The goal in the sampling stage is to generate sufficient records of values of residuals to enable training of a model. The application is executed with a set of different...
input sizes multiple times. Bit-flips are performed non-deterministically to the critical data structures, such that the outputs reflects both CORRECT runtime (soft error free, or SDC that does not lead to a significant change in the final result) and CORRUPTED runtime. To achieve better models, input size ranges for sampling should cover most (if not completely) commonly used input sizes, and the number of training sizes are recommended to be large enough. This is off-line process, and the data set generated here could be repeatedly used for training in the future.

**Training Stage:** At the training stage, we focus on generating the models that would classify the runtime sequence of residual values into CORRECT or CORRUPTED class. While several of the well-known classifier algorithms would be suitable, we have experimented with a popular algorithm, Support Vector Machine (SVM), in our study. To have a model that can execute across different input datasets for a given application, the following two preprocessing steps are taken. Before training the models, the data sets are scaled from 0 to 1 (or other user specified ranges) as a way of normalization. For each input size, we obtain the maximum number of iterations for the algorithm to complete from the correct executions. We refer this number of iterations as MaxIter. During the scaling, we only focus on the values that are within the MaxIter, by doing so, we avoid the irrelevant values from delays and infinite loops for the incorrect executions.

Models are trained as follows. To train the classifier model that is called, say, at 20% of the runtime, we use the MaxIter information and filter the data, i.e., discard the data from executions beyond 20%. Now, for each execution, we have a scaled vector representing the series of values of the residuals, together with a label that denotes whether the execution
is CORRECT or CORRUPTED. The goal of the model will be to examine another time-series, and label it as CORRECT or CORRUPTED. Algorithm 10 shows the outline of this process.

Algorithm 10: Example of training a 20% classifier model.

1. Sampling data with different input sizes
2. DataSet ← Sampling
3. 
4. for all data ∈ DataSet do
5.    Extract 20% of the execution;
6.    Scale:
7.    InputSet.push(data);
8. end for
9. 
10. SVM20 = new SVM();
11. SVM20.train(InputSet);
12. ModelPool.push(20, SVM20);
13. 
14. // Continue for other training

We can train models for different fractions of executions, choosing as many fractions as we will like. The example from Figure 4.4 shows a model pool with models trained from every 10% of the execution. Using fewer models can reduce both the training and runtime costs, but can also increase the latency of detection of an SDC. Use of more models, on the other hand, increase training and runtime costs, but can reduce the latency of detection.

Execution Stage: The set of models created during the training phase is available for the execution stage. We can now execute the application with any input (problem) size. However, we assume that the MaxIter is known. During the execution, when the application is at a fraction of iterations for which a model is available, we can choose to invoke the model to verify the correctness of the execution. Verification involves taking the residual
value time-series from the current execution, scaling it to the same range as in the training stage, and then passing it to the model. If the runtime produces values that yield to the same pattern as those in *CORRECT* class (which suggests that current runtime is soft error free or little impact is observed), the model would be expected to classify it as *CORRECT* class. If the bit flip corrupts significant amount of the computation, the pattern will be recognized to be *CORRUPTED* class. In such case, a recovery process is needed.

**Algorithm 11: Example of runtime detection**

1. *Profiling* with *DataSet*
2. *ModelPool* ← *Profiling*
3. ...
4. // Enters main loop of the algorithm
5. *CurIter* ← 0;
6. while !Converged do
7.   ...
8.   *CurProgress* ← *CurIter*/MaxIter*100;
9.   if *ModelPool*.contains(*CurProgress*) then
10.      *CurModel* ← *ModelPool*.get(*CurProgress*);
11.      *SVMClass myClass* ← *CurModel*.classify(*CurModel*);
12.     if *myClass* ≡ *CORRUPTED* then
13.        Start Recovery;
14.     end if
15.   end if
16.   ...
17.   *CurIter*++
18. end while
19. // Other codes of the application...

Algorithm 11 shows an example of runtime that calls all of the available models. In practice, users could choose the frequency of verification as needed.

**Recovery:** If any of the models invoked detects potential threat from soft error, the recovery process will be invoked. However, it is also possible to setup a rule to start recovery when
multiple models agree with the soft error. Any form of the recovery would be applicable in our approach, as long as the recovery guarantees that the execution starts before the state was corrupted. This can be challenging because of the potential latency in detecting soft errors. Although significant latency is not observed in this study, it is possible that errors due to SDC could be amplified much later than the fraction of the execution where it happened. Choosing an earlier recovery point can be helpful in this regard, even if it slows down the application.

4.3 Experimental Results

We focused on the applications where the residual value does not monotonically decrease (such a signature will be trivial to learn), but otherwise have a signature. Thus, we evaluated our work with three mini-apps – MiniFE, Conjugated Gradient (CG), and HPCCG. CG is a popular algorithm for the numerical solution of linear question \( A \times x = b \), whose matrix \( A \) is symmetric and positive-definite. HPCCG and MiniFE are mini-app developed by the Mantevo project at Sandia National Labs [3]. Particularly, HPCCG is an approximation to an unstructured implicit finite element or finite volume application and MiniFE is a proxy application for unstructured implicit finite element codes.

We only focus on the transient bit flips (which causes SDC) in the experiments. Our soft error injector is developed using PIN tool, which can dynamically instrument binary files. We track the applications runtime heap usage, and randomly injects bit flips to the heap during execution. The soft errors can be injected at any time, at any place, either randomly or under our control. The fault-injection involves changing a bit value from 1 to 0 (or vice-versa) in the heap. To avoid first two types of errors described in Section 6.2, and better evaluate the performance, we inject bit flips only to the data structures that must be
Figure 4.5: Results for on-line prediction(AID) on different problem sizes. The figure shows bit flips occurs in 5% interval (20-45, 55-80 is omitted due to the similarity to 50%)

consumed (read) by the program in the future computations, so that the bit flips are more likely to impact the application’s final output.

All of our experiments are performed on a cluster whose nodes have two quad-core 2.53 GHz Intel(R) Xeon(R) processors, equipped with 12 GB RAM. The machine is executing RedHat Enterprise Linux Server release 6.1, with Gigabit ethernet interconnection.

The experiment is conducted with the following goals. First, we want to evaluate the accuracy of detection, and compare it against what is obtained by an online method, AID [27]. Second, we focus on the latency of detection, i.e., with an error injection at a certain stage of execution, how soon are we able to detect the error. Third, we consider the overhead of detection. Finally, we generalize our error-injection to consider multiple (correlated or uncorrelated) bit-flips and examine if the models built on a single bit-flip are still effective.

4.3.1 Accuracy

To evaluate the accuracy of our solutions, we apply both our approach and an on-line prediction method (to compare with) to our applications. On-line prediction methods record data values at runtime and try to predict the value for the next few time-steps. If the
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Input Size (Input Matrix)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MiniFE</td>
<td>100 × 100</td>
</tr>
<tr>
<td></td>
<td>120 × 120</td>
</tr>
<tr>
<td></td>
<td>(increased by 20)</td>
</tr>
<tr>
<td></td>
<td>300 × 300</td>
</tr>
<tr>
<td>CG &amp; HPCCG</td>
<td>5k × 5k</td>
</tr>
<tr>
<td></td>
<td>5.5k × 5.5k</td>
</tr>
<tr>
<td></td>
<td>(increased by 0.5k)</td>
</tr>
<tr>
<td></td>
<td>10k × 10k</td>
</tr>
</tbody>
</table>

Table 4.1: Datasets used for Training for Each Application

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Input Sizes (Input Matrix)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>In Range</td>
</tr>
<tr>
<td></td>
<td>Out of Range</td>
</tr>
<tr>
<td>MiniFE</td>
<td>150 × 150</td>
</tr>
<tr>
<td></td>
<td>50 × 50</td>
</tr>
<tr>
<td></td>
<td>250 × 250</td>
</tr>
<tr>
<td></td>
<td>400 × 400</td>
</tr>
<tr>
<td>CG</td>
<td>6.2k × 6.2k</td>
</tr>
<tr>
<td></td>
<td>3k × 3k</td>
</tr>
<tr>
<td></td>
<td>8.4k × 8.4k</td>
</tr>
<tr>
<td></td>
<td>15k × 15k</td>
</tr>
<tr>
<td>HPCCG</td>
<td>6.6k × 6.6k</td>
</tr>
<tr>
<td></td>
<td>3k × 3k</td>
</tr>
<tr>
<td></td>
<td>8.8k × 8.8k</td>
</tr>
<tr>
<td></td>
<td>15k × 15k</td>
</tr>
</tbody>
</table>

Table 4.2: Applications and Input Set Configuration for Experiment
actual value dramatically differs from the prediction (usually measured with a tolerance threshold), the method would consider the situation to be erroneous. There are various detection models, based on the different approach they took to predict and tune the accuracy. In our experiment, we use an impact driven detection method – AID [27], which is a recently developed method. In this experiment, we use the default configuration of the AID library, apply AID to re-usable data only, and compares its detection rate with ours.

From the study in Section 6.2 and Figure 4.1, we understand that bit flips in only higher order of bits brings impact to the final result. Therefore, in our experiment, we only inject bit flips to 32th bit or higher for CG and HPCCG, and 16th and higher for MiniFE, with the idea that lower order bit-flips, even if undetected, will lead to an accurate final result. Also, in all our results, no false positives were ever found. Thus, the evaluation metric we use is accuracy of detection, which is the fraction of cases with soft-errors that are correctly detected by our method. In other words, since the precision is 100%, we do not report F-score or a similar metric, but just report the recall (referring to it as accuracy here).

Figure 4.6: Results for detection with off-line training on different problem sizes. The figure shows bit flips occurs in 5% interval (20-45, 55-80 is omitted due to the similarity to 50%)
Figure 4.5 shows the overall accuracy of AID. We have experimented the applications with 2 different input sizes. In each execution, we injected bit-flips at a randomly selected location within the heap, and discarded the executions where the bit-flip results in an application crash. In each experiment, we inject bit-flip at a randomly selected point within the first 5% of the execution, or within the next 5% of the execution, and so on. The result we report is the averaged detection rate of 50 runs of executions for each time interval.

From the result, we can see that AID prediction method works well with MiniFE, with the detection rate close to 100%. However, it does not perform well on CG and HPCCG. This may because that the pattern of MiniFE under a soft error is more easily distinguishable, whereas the pattern of values with CG and HPCCG are more complex.

Now we discuss the detection accuracy with off-line training. The input dataset sizes for the sampling stage are summarized in Table 4.1. Each application has 20 input sets that can be used for training. Following the standard cross validation practice in machine learning, we select four input sizes that are not used for training, but for testing the model. These four are selected to include two datasets that are not within the range used for training, and another two that are within the range. The configuration of the experiment is shown in Table 4.2. Similar to our experiments for the AID method, we run the test inputs 20 times for each 5% of the execution, and average the detection rate.

The results are shown in Figure 4.6. The detection rate for MiniFE ranges from 90% to 100%, with misses coming when the input sizes are out of range and the bit-flip occurs near the end of the execution. For CG and HPCCG, the detection rate ranges from 80% to 100% if the bit flip occurs in first 80% of the execution. However, the detection rate drops significantly as the bit flip occurs at the very late stage of the execution. When bit flip occurs around 90 to 95% of the execution, the detection rate would drop to as low as 60%
Figure 4.7: Detection rate with different models in different execution stage. Each model shows its detection rate for bit injection in its covered execution range. Input sizes are: MiniFE 150*150, CG 6.2k * 6.2k, HPCCG 6.6k * 6.6k
for the in-range input and 50% for the out-of-range inputs. The reason will be discussed later as we examine the latency trends.

4.3.2 Latency

Now we evaluate the latency, i.e., after a bit-flip occurs, how soon it is detected by our method. Because the time-series of residual values starts diverging only after the bit-flip occurs, we can expect some delays till the model has sufficient data to conclude that SDC has occurred. In our experiment, we trained the models for every 10% of the execution. The results are shown in Figure 4.7. Model \( x \) represents a model trained for detection after \( x \)% execution is over. For each model, we consider bit-flip at different stages, up to the stage for which the model was trained, and see how effective it is in detection. Figure 4.7(a)-4.7(c) shows the result for Models that cover first 30% of the execution. Despite some variations, the performance on MiniFE is very high, because it is easy to distinguish abnormal behavior. For CG and HPCCG, it is difficult for the models to identify the soft errors.

Figure 4.8: Overhead shown as slowdown percentages (averaged across all input sizes)
at the beginning, however, as the computation continues, the detection accuracy gradually increases. For example, bit-flips occurring in first 5% of the execution are detected at the rate of 50% and 45%, for CG and HPCCG, respectively, by Model10 (Figure 4.7(a)). However, when the runtime reaches 30% of the execution (Figure 4.7(c)), the bit flips in first 5% are detected at a rate of 70% and 65%. This continues as the computation goes on, and at 40% of execution (Model40), both reaches 80% of detection. Similar pattern could be observed for bit flips in other ranges as well. To summarize, our approach needs some time to identify an SDC, however, it still detects SDC in runtime so that recovery process could start in a timely manner.

### 4.3.3 Overheads

Now we show the overhead our method incurs. Like in the accuracy experiment, we apply AID on reusable data only. For our method, we performed the off-line training prior to the experiment. The costs depend on the number of data instances that is used for training purpose. In our experiments, to generate sufficient number of instances, we spent 3500 to 5500 seconds in off-line operations. However, this is a one-time cost and will benefit all
the future executions. Runtime overheads in these experiments are measured for executions
where there is no bit-flip, but the machine learning model is invoked after every 10% of the
iterations. The absolute time taken by the model is independent of the input size, and thus
the relative slowdown is very dependent on the dataset sizes. The results we report in
Figure 6.7 shows the slowdown percentage for both of the approaches by averaged from
all the tested input sizes. We can see that the overall slowdown percentage is under 7%
for all applications. Among the three applications, MiniFE has lower overhead than others,
because it executes for a much larger number of iterations.

We also report the absolute cost of Model60 in CG as an example to better illustrate
how input sizes affect the relative overheads we had shown earlier (Figure 4.9). Data is
averaged from 5 runs in each input size. From Figure 4.9(a), we can see that the absolute
cost of Medel60 call is slightly increasing as the input size increases, however, we get
relative overhead percentage since the overall execution time increases at a faster rate.
This observation suggests that our approach achieves better performance in larger datasets.
Note that the overheads in Figure 4.9(a) are about an order of magnitude lower than those
reported in Figure 4.9, because the former involves invoking just one model, whereas the
latter involves invoking 10 different models during the entire execution.

### 4.3.4 Generalized Fault Injection

All of our experiments so far have focused on detecting single bit-flips on model trained
using a single bit-flip. In practice, multiple bit-flips can occur. Thus, we now focus on evalu-
ating the accuracy and latency of detection with two different models of bit-flip injection
– neighboring (or correlated) bit-flips, and random double bit-flips, where we randomly flip
two bits within a randomly selected single variable, randomly flip two bits anywhere in the
Figure 4.10: Detection rate with different models in different execution stage on double flips. Each model shows its detection rate for bit injection in its covered execution range. Input sizes are: MiniFE 150*150, CG 6.2k * 6.2k, HPCCG 6.6k * 6.6k.

Figure 4.11: Detection rate with double flips occurring in different 5% intervals (20%-45% and 55%-80% are omitted due to the similarity to the 50% case).

entire memory, respectively. In both the cases, the two bit flips are injected at the same time-step. Figure 4.11 shows the accuracy of detection for these two models of bit-flip injection. As we can see from the result, our solution actually works better with double bit-flips compared to single bit-flip evaluated Figure 4.6. This is likely because the error propagates more rapidly, resulting in easier classification (detection). It can also be observed that random flips are slightly more effectively detected than neighboring flips.
The multi-flips impacts the latency, too. Figure 4.10 shows the effect on selected models (other models are not reported, but have the similar effects). As we can see from the figure, double bit flips are generally detected faster than single ones, especially on random flips.

4.3.5 Evaluations on Sparse Matrices

Besides of using typical dense symmetric matrices as inputs, we also evaluated the performance on public sparse matrices. In this experiment, the input sparse matrices include JGD_SPG family, and random selects from Koutsovasilis and Schenk families, as inputs for conjugated gradient application. We train the model with similar setups as above, but with more consideration for initial input for the solutions. We collect the solution sets from the initial inputs, and evaluates if the approach would benefit when the initial guess is close to the final answer.

There are 6 matrices in JGD_SPG family. Figure 4.12 shows the detection rate for CG when the input is JGD_SPG/EX1 and JGD_SPG/EX2, while the models are trained with the remaining matrices in the family. We only report the result when bit flip is introduced.
Figure 4.13: Overall detection rates for Conjugated Gradient with bit flip injected in different execution stage with/without initial guess of the solution for sparse matrix. Training: JGD_SPG/EX1 to JGD_SPG/EX4, Input: JGD_SPG/EX5, JGD_SPG/EX6

in the first 50% of the execution. Flips within 20% to 45% is omitted due to the similarity to others. We have 10 models observing the application behaviors and reports soft error if any of them judges the runtime to be erroneous. The results in Figure 4.12(a) shows that our approach achieves up to 70% of the detection rate when the bit flip occurs. It is similar to the previous results, which is that the detection rate slightly decreases when the bit flip occurs later in the execution. We collected the solutions of CG during the entire executions to evaluate the detection rate with different initial guess. Figure 4.12(b) shows the result when we set the initial guess to half of the final solution. It shows no significant impact even if the solution is pre-set.

Figure 4.13 shows the similar experiment conducted on JGD_SPG/EX5 and JGD_SPG/EX6 while the training was performed with JGD_SPG/EX1 to JGD_SPG/EX4. As we can see, the result is similar to the JGD_SPG/EX1 and JGD_SPG/EX2, which we believe is reasonable.
At last, we trained the model with the entire JGD_SPG family, and evaluated on two matrices, randomly picked from two completely different sparse matrix families: Koutsovasilis/F2 and Schenk/nlpkt80. The result is shown in Figure 4.14. It is observed that the performance is less impressive than evaluating with the same family matrices. This might be caused by the different properties held between matrices families. The decrease pattern of residual from JGD_SPG matrices differs from the testing matrices, such that some impact of soft error may result in the residual behavior to change in a way that is considered correct in the training set. In this case, the soft error will not be detected.

4.4 Summary

This chapter has presented an automated soft error detection approach for iterative convergent scientific applications. We have shown how we use an off-line training framework using machine learning, construct multiple classifier models based on the sampling inputs, and periodically invoke the classifier to verify the status during the executions.
We have evaluated the overhead of our approaches using three applications where the residue is not monotonically decreasing. Our accuracy evaluation shows that our method can achieve an overall accuracy at more than 80%, and almost 100% for one of the applications. The overheads are less than 7% for all three applications, and actually lower for larger datasets. We have compared against a state-of-the-art on-line prediction method, and shown that we have better accuracy, though at the cost of higher latency.
Chapter 5: Algorithm Level Fault Tolerance for Molecular Dynamic Applications

This chapter includes algorithmic solution to provide soft error resiliency for molecular dynamic simulators that are based on Newton’s Laws. We first introduce the fault tolerant matrix vector multiplication, then discuss how this idea could be improved for kernel of molecular dynamic applications. We explore the force computation logic and seek the possibilities of transforming array based computation into the form of matrix vector multiplication.

5.1 Background Algorithm

This section covers background in two important areas. First, we give an overview of molecular dynamics applications. Second, we explain the basics of algorithm-level fault tolerance through a MVM algorithm.

5.1.1 Molecular Dynamics

Classical MD is a computational tool for simulating the properties of liquids, solids, and molecules [74]. A typical computation regards each atom or molecule as a point of mass and applies Newton’s equations to all the points, and updates values such as coordinates, speed, force, temperature and others. For each point $x$, the algorithm collects all of
its neighbors (based on a cut-off distance) and simulate their interactions using Newton’s equations. Thus, MD simulations are radius-based computations \([53, 74]\).

MD simulations are usually not memory intensive since only the atom information is stored. However, the application scale is intense computationally because of the number of atoms and the time-steps the application iterates on. Many millions of atoms or molecules will be simulated simultaneously, and for a large number of iterations, to obtain useful results.

5.1.2 An ABFT Example

Before showing our approach for molecular dynamics, we give an overview of the MVM checker algorithm that is proposed by Sloan et al. \([84]\). Suppose the following computation is performed: \(A \times x = y\), where \(A\) and \(x\) are the input matrix and the vector, respectively, and \(y\) is the output. Let the size of the matrix be \(n \times n\). We introduce \(c\) as a checker vector, which is a vector of length \(n\) with all 1s. The MVM computation is correct if \((c^T \times A) \times x = c^T \times y\), where \(c^T\) is the transposed vector of \(c\).

For better illustration, we take the following example. Consider a \(4 \times 4\) matrix \(A\) and the vector \(x\), then the checker vector \(c\) will be \(\{1, 1, 1, 1\}^T\).

\[
A_{4,4} = \begin{pmatrix} 3 & 5 & 2 & 8 \\ 4 & 7 & 1 & 6 \\ 5 & 8 & 9 & 0 \\ 1 & 1 & 7 & 3 \end{pmatrix}, \quad x = \begin{pmatrix} 3 \\ 3 \\ 1 \\ 9 \end{pmatrix}, \quad c = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}
\]

If there is no error in the computation, the matrix vector product should be the vector \(y\).

\[
y = \begin{pmatrix} 98 \\ 88 \\ 48 \\ 40 \end{pmatrix}
\]

In order to check if the output is correct, we can apply the checker vector to the equation as we discussed earlier. We compute \((c^T \times A) \times x = 274\) and \(c^T \times y = 274\). If the
results are equal, we can see that the original MVM computation is correct. It is also worth noting that such additional check has much lower cost than the original computation.

Now, instead suppose an error occurred during the computation and made the second element of the product to be erroneous. Let us refer this incorrect output to vector $f$:

$$f = \begin{pmatrix} 98 \\ 70 \\ 48 \\ 40 \end{pmatrix}$$

If we multiply vector $c$ to the vector $f$, we get $c^T \times f = 256$, which differs from $(c^T \times A) \times x$. As Sloan et al. indicated, this approach can be extended to locate the errors during the runtime based on the following observation: If the erroneous value is in top half of $f$, then the top half of the checksum vector $(c^T \times A) \times x - c^T \times f$ will be non-zeroes and the bottom half will be zeroes. On the other hand, if the error affected in the bottom half, then the $(c^T \times A) \times x - c^T \times f$ will have zeroes in the top half, but some non-zero values in the bottom half. If the erroneous halves are found, we can recursively apply this approach to locate which elements are faulty. To take an example, suppose the error is in $f_2$ (the second element in output vector $f$), the checker vectors will be as follows.

$$c_{11} = \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \end{pmatrix}, c_{12} = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 1 \end{pmatrix}, c_{21} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}, c_{22} = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix}, c_{23} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}, c_{24} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}$$

We have the followings:

$$c_{11}^T \times f - c_{11}^T \times A \times x = -18 \text{ (Errors found from } f_1 \text{ to } f_2)$$

$$c_{12}^T \times f - c_{12}^T \times A \times x = 0 \text{ (No errors from } f_3 \text{ to } f_4)$$

Since we found errors in range $f_1$ to $f_2$, we apply the checker $c_{21}$ and $c_{22}$ to further determine which element it is.
\[ c_{21}^T \times f_{c_{21}}^T \times A \times x = 0 \ (No \ errors \ in \ f1) \]

\[ c_{22}^T \times f_{c_{22}}^T \times A \times x = -18 \ (Errors \ found \ in \ f2) \]

Now we claim that element \( f_2 \) is erroneous, so that we can apply error correction technique to fix it.

Once the faulty elements are identified, they can be corrected by re-computing corresponding rows in the matrix and the vector. As a specific example, if the erroneous element is \( f_i \) (where \( i \) is the row number in the vector \( f \)), then we can get correct \( f_i \) by computing the dot product of and vector \( x \) and \( i^{th} \) row from the matrix \( A \). A trade-off is that, depending on the matrix size, locating the particular erroneous element may be more expensive than re-computing part of the result vector. Therefore the actual vector size that needs to be re-computed varies based on the particular problem we are dealing with.

ABFT so far shows promising performance in linear algebra solvers. But for irregular applications as MD applications, there are not yet reliable ABFT approaches proposed. We study possible ABFT approach for this class of applications in the next section and evaluate the performance of our approach.

### 5.2 Fault Tolerant Molecular Dynamics

We first study the main kernel of a typical MD application, and then we demonstrate how we can apply ideas similar to those in ABFT MVM checker to the MD kernel.

Algorithm 12 is the main computational kernel of an MD application. In this code, only atom velocities and forces are demonstrated. Here, a spatial partitioning algorithm is used to partition the work space. Each node is assigned with a particular partition of the space, and is responsible for all atoms that stay in that assigned space. In every time-step,
Algorithm 12: Main Kernel of MD application
1. for iter = 0 \rightarrow n do
2. \quad x \leftarrow \text{coordinate}
3. \quad for i = 0 \rightarrow \# of local do
4. \quad \quad \text{//Update } x[i]
5. \quad \quad x[i] \leftarrow dt \ast v[i] + x[i]
6. \quad end for
7. \quad if iter \% re-neighbor then
8. \quad \quad Communicate with neighbors
9. \quad else
10. \quad \quad Rebuild neighbor list
11. \quad end if
12. \quad Update force (run force kernel)
13. \quad v \leftarrow \text{velocity}
14. \quad f \leftarrow \text{force}
15. \quad for i = 0 \rightarrow \# of local do
16. \quad \quad \text{//Update } v[i]
17. \quad \quad v[i] \leftarrow dt \ast force \ast f[i] + v[i]
18. \quad end for
19. end for

Each node iterates all the atoms in its workspace and updates the information (coordinates, velocities, and forces) of the atoms. New coordinates are computed based on the new values of the velocities, and new values of velocities, in turn, are based on the new values of forces at current iteration. To compute these values, each node has its own data structure to hold its atoms coordinates, velocity and forces values. Also, a data structure contains information of neighboring nodes as well as its boundary atoms information, all stored in the form of what is normally referred to as a \textit{neighbor list}. At each iteration, the kernel updates the coordinates of the atoms it is responsible for using the velocities updated in the previous iteration, then computes the force based on the updated coordinates, and updates velocity values based on the force values. In addition, as the atoms move (i.e., their coordinate changes) during the computation, the home node of an atom may change. Therefore, an
update of the neighbor list is performed at periodically. During the update of neighbor lists, we update coordinates of all of the nodes, and potentially re-assign the atoms to the correct nodes based on the latest coordinates of the atom. Algorithm 2 shows the kernel for updating forces. For each atom in the node, the kernel iterates over all of its neighbors using the neighbor list. For all neighbors of the current atom, we compute the difference of coordinate, and then apply Newton’s equation to compute the amount of force.

\begin{algorithm}
\caption{Force Kernel of MD application}
\begin{algorithmic}[1]
\For{$i = 0 \rightarrow \#\text{ of local}$}
\State $x[i] \leftarrow$ coordinate
\EndFor
\State $\text{neigh} \leftarrow$ neighbor – list
\For{$k = 0 \rightarrow \#\text{ of neighbors}$}
\State $\Delta x \leftarrow \text{diff}_x(x[i], \text{neigh}[k])$
\State $\Delta y \leftarrow \text{diff}_y(x[i], \text{neigh}[k])$
\State $\Delta z \leftarrow \text{diff}_z(x[i], \text{neigh}[k])$
\State $rsq \leftarrow \Delta x^2 + \Delta y^2 + \Delta z^2$
\State $sr2 \leftarrow 1/rsq$
\State $sr6 \leftarrow sr2^3$
\State $f \leftarrow sr6 * (sr6 - 0.5) * sr2$
\State $f_{i-x,y,z} \leftarrow \Delta x, y, z * f + f_{i-x,y,z}$
\State Update $f_{\text{neigh}_k}$
\EndFor
\end{algorithmic}
\end{algorithm}

5.2.1 Soft Error Detection

In this study, we focus on the soft error that occurs in the main memory, i.e., anywhere in the main arrays. We also observe that the main challenge is to be able to detect soft errors, since the C/R is a viable method for recovering from an error, once it has been detected. Our main idea is to transform the force computation kernel, so that a checker analogous to the one used in MVM can help detect soft errors.
If we take a closer look at the force computation kernel (Algorithm 13), we can observe that atoms coordinates themselves can be considered as vectors. The neighbors of each atom (in the neighbor list) can be represented as matrices. Viewed this way, the force calculation can be viewed as an MVM calculation. We make this concrete by considering the example in Figure 5.1. The array that stores atoms coordinates is the vector \( V_c \), and each element \( x_i \) in \( V_c \) represents the coordinate of the atom \( x \). We create an \( n \times n \) matrix \( ftM \) where \( n \) is the total number of atoms on the the current node. For each row in \( ftM \), we want row \( i \) to represent all of the atoms that neighbors element \( i \) in \( V_c \). To achieve this, we fill the matrix by the following rule: \( ftM_{i,j} = 1 \) where \( x_j \) is a neighbor of \( x_i \), otherwise \( ftM_{i,j} = 0 \). We refer \( ftM_{i,j} \) to neighboring cell of \( x_i \) in this study.

Once the matrix \( ftM \) is generated, we can perform MVM operation on \( ftM \) and \( V_c \) and get the product \( V_f \). During the MVM procedure, since all non-neighbor cells in \( ftM \) are set to 0, only the neighbors of \( x_i \) will be involved in the computation for \( f_i \), where \( f_i \) is the ith element in output vector \( V_f \). From the example, we can see that since we set neighboring cell to 1, in output vector \( V_f \), each element \( f_i \) is simply the sum of the atom \( x_i \)'s neighbors coordinates, i.e, it is not computing the force variation. However, by simply modifying \( ftM \), we can let \( ftM \times V_c \) produce \( V_f \) which contains the force variation. By

\[
\begin{pmatrix}
0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\
1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 \\
0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\
1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
\end{pmatrix}
\begin{pmatrix}
x_1 \\
x_2 \\
x_3 \\
x_4 \\
x_5 \\
x_6 \\
x_7 \\
x_8 \\
\end{pmatrix}
= 
\begin{pmatrix}
f_1 = x_2 + x_3 + x_6 \\
f_2 = x_1 + x_4 \\
f_3 = x_1 + x_4 + x_5 + x_7 \\
f_4 = x_2 + x_3 + x_6 \\
f_5 = x_3 + x_8 \\
f_6 = x_1 + x_4 \\
f_7 = x_3 \\
f_8 = x_5 \\
\end{pmatrix}
\]

Figure 5.1: Example of Converting Force Kernel to MVM
studying Algorithm 2, we know that for each atom $x_i$ and its neighbor $x_j$, we have the following equations:

$$\delta x = x_i - x_j$$

$$r_{sq} = \delta x^2 + \delta y^2 + \delta z^2$$

$$f = \theta(r_{sq})$$

$$f_i = \delta x \times f$$

To summarize, for each atom $x_i$ and its neighbor $x_j$, the force variation is $f_i = \tau(\delta x)$, which can be denoted as $F(x_i, x_j)$. Given this information, we can construct the $ftM$ by assigning $F(x_i, x_j)/x_j$ instead of 1 to $ftM_{i,j}$. Given the matrix $ftM$ computed like this and $V_c$, the output $V_f$ will hold the expected result. It should be noted that since the coordinates are not updated until all the atoms in the current spaces are iterated through, once a matrix in constructed, it can be used until the iteration terminates.

Now, we have an MVM expression of the MD force computation kernel. The MVM checker described in Section 6.2 can be directly applied to this model, i.e., we can use a checker vector $c$ which has size of $n$ and all elements are 1. At the end of each iteration, we can track the result of $(c^T \times ftM) \times V_c$ and $c^T \times V_f$. If the results are the same, then we can claim that the computation is free of soft errors.

This simple idea, however, has limitations that we address now. First, in MD applications, the values are stored in double precision and thus there is a round-off problem. Therefore, we introduce a threshold $\Psi$, such that as long as the difference in the two expressions is smaller than $\Psi$, we consider the computation to be correct. A trade-off should be considered when determining the threshold $\Psi$. If $\Psi$ is small, then the algorithm will
consider floating point round-off difference as a faulty output, and the result will be a false positive. On the other hand, if the threshold $\Psi$ is large, the algorithm will likely be unable to detect soft errors in some of the least significant bits (though such errors have less impact to the output). We evaluate different threshold values and their impact in the experimental results section.

5.2.2 Optimization

The above solution has a problem, which is that of larger input size. To construct a $ftM$, we would need $n \times n$ space for input size $n$. This is much larger than the size consumed by neighbor lists. Thus, we clearly need to optimize the solution to reduce such cost. This is now achieved as follows. The key idea is that the matrix is never formally generated, or more broadly, no substantial change is made to the data structures. Consider the MVM checker equation

$$(c^T \times ftM) \times V_c - c^T \times V_f \leqslant \Psi$$

Since $(c^T \times ftM)$ produces a vector with 1 row and $n$ columns that stores the sum of columns in $ftM$, we only need to keep a vector of checksum column ($V_{cc}$) in the memory. As soon as the value $ftM_{i,j}$ is computed, we add result into corresponding cell in $V_{cc}$, i.e., perform the operation

$$cc_j += ftM_{i,j}$$

where $cc_j$ denotes the jth element in $V_{cc}$. In this way, we will be able to maintain $V_{cc}$ using $O(n)$ space and $V_{cc}$ eventually becomes the product of $(c^T \times ftM)$. Meanwhile, we reduce $ftM_{i,j}$ and $x_j$ immediately when they are computed, and reduce all neighboring elements of $x_i$ to get $f_i$. Therefore, we can have $V_{cc}$ (the product $c^T \times ftM$), $V_f$ (the
product $ftM \times V_c$, $V_c$, and $c^T$ available, to perform the correctness check. Next, to check the validity of $V_f$, we only need to compare the vector product of $V_{cc} \times V_c$ and $c^T \times V_f$.

Besides of the coordinates and forces, MD application contains several other data structures, which includes those storing properties like velocity, temperature, and others. However, both access and computations on these structures is relatively less intensive compared to the ones storing coordinates and forces. Furthermore, some of these structures (e.g. velocity array) are updated based on the force result from the last (or last few) iteration. Therefore, these data can be protected using more straight-forward algorithms like simple checksums. These data is usually stored in arrays (could be in either one or multi dimension). We collect the checksum value for each dimension before entering the force computation procedure, collect the checksum value again before updating them (i.e. after force computation), and compare this against the previous copy to find if they are the same. As the simplest implementation, we could have all values added up and take the sum as the checksum value. However, this approach also has round-off and potential overflow issue. Therefore, we apply the bit-wise checksum, i.e, instead of adding up the value themselves, we bit by bit add up the data and store them in array in size of $\text{sizeof(double)}$. Additionally, by applying bit checksum, we can potentially omit certain bit ranges to achieve different level of accuracy.

5.2.3 Recovery

Once an error is detected by our approach, we need to recover from the errors, in order to to prevent fault amplification. We propose two separate methods for this purpose.

Checkpoint and Restart: As is discussed above, MD applications are usually not memory intensive, which means that the data size of the application is relatively small. This property
makes the checkpoint and restart to be a natural choice. Either system level or application level checkpoint could be used to recover from an error.

**Algorithm Level Recovery:** We had earlier shown how that the ABFT MVM approach is able to locate the erroneous element, and correct it by re-computing the values. This property can also apply to our approach. Given \( ftM \) and \( V_c \), we can apply partial checksum vector to locate the erroneous values \( f_i \), and re-compute the \( f_i \) using \( i^{th} \) row of \( ftM \) and \( V_c \). Similarly, depending on the size \( n \), we can choose to re-compute part of \( V_f \) instead of locating exact \( f_i \) during the implementation. Locating the \( f_i \) takes \( O(\log n) \) time. By applying algorithm level recovery, we can protect \( V_f \) at each time-step. However, if the error detected happens to be in data structure other than \( V_f \), then the application still needs to restart from latest checkpoint.

Additionally, algorithm level recovery has better performance in terms of time overhead. However, this approach needs intermediate data \( ftM \), which has a high memory cost.

### 5.3 Experimental Results

#### 5.3.1 Fault Model

In our experiments, we only focus on transient bit flips. We developed a soft error injector based on PIN, which is a dynamic instrumentation tool for binary files. With the features of PIN-tools, our soft error injector traces applications heap usage, and injects bit flips randomly to the heap. These errors can be injected at any place, and at any time, either randomly or under our control. The fault-injection involves flipping a bit value from 0 to 1 (or vice-versa) in the memory. To better evaluate the performance of our idea, bit flips are injected to the data structures that must be consumed by the program in the future.
computation, so that the bit flips are more likely to impact the application output. We evaluate our approach in the following two aspects. First, we evaluate the effectiveness of fault detection model, and second, we evaluate the overhead of algorithm-level and the C/R based recovery approaches.

We experimented on two different implementations of the main molecular dynamics idea, which are miniMD and CoMD. miniMD is a mini-app developed by the Mantevo project at Sandia National Labs. The goal of the Mantevo project is to create representative mini versions of full-scale scientific applications. Particularly, MiniMD is a smaller version of a large-scale software LAMMPS. CoMD is a reference implementation of typical classical molecular dynamics algorithms and workloads. It is created and maintained by The Exascale Co-Design Center for Materials in Extreme Environments (ExMatEx). The code is intended to serve as a vehicle for co-design by allowing others to extend and/or re-implement it as needed to test performance of new architectures or programming models.

All experiments are performed on a cluster where each node has two quad-core 2.53 GHz Intel(R) Xeon(R) processors, with 12 GB RAM, executing RedHat Enterprise Linux Server release 6.1, and Gigabit ethernet as the interconnect.

To put the results evaluating our method in context, Figure 5.2 shows how single bit flip affects the correctness for miniMD (original code). The initial input coordinates are real numbers in range of -1 to 1. Errors are computed by averaging the normalized difference between the normal output and the outputs with single bit flip injected. For each bit range and for each execution stage (first 20% of execution, the next 20%, and so on), we randomly injected the bit flip in one of the bits. This process is repeated 10 times, and the results reported in the figure are the averaged value of all successful executions.
Figure 5.2: Average Normalized Difference when injected with single bit flip during different execution stage: miniMD (4k atoms, input range [-1.0, 1.0])

It is observed that when the bit flip occurs in a higher order bit, the overall impact is greater. The overall difference nearly reaches order of 1 when the bit flip occurs in bit range of 48th and higher, and may reach order of 10 when it goes to bit range 56th and higher (specifically, in 56th to 63th bits, 60% to 80% of the execution). Considering the initial input range is [-1, 1], difference in order of 10 suggests that all atoms in the final output might be 10 times far away from where they were suppose to be. Thus, we can see that even a single bit flip can have a very negative impact on correctness of the algorithm.

5.3.2 Effectiveness of Algorithm Level Fault Tolerance

To evaluate the effectiveness of our approach, we use the metric F-score, which is also commonly used to summarize effectiveness of any method [87]. An F-score can be obtained using the following formula:
F-Scores in Different Bit Ranges: miniMD(4k, [-100,100])

<table>
<thead>
<tr>
<th>Bit Range</th>
<th>56~63</th>
<th>48~55</th>
<th>40~47</th>
<th>32~39</th>
<th>24~31</th>
<th>16~23</th>
<th>0~15</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.00E+01</td>
<td>1.00</td>
<td>1.00</td>
<td>0.93</td>
<td>0.77</td>
<td>0.67</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>1.00E+00</td>
<td>1.00</td>
<td>0.93</td>
<td>0.93</td>
<td>0.77</td>
<td>0.55</td>
<td>0.22</td>
<td>0.00</td>
</tr>
<tr>
<td>1.00E-01</td>
<td>1.00</td>
<td>1.00</td>
<td>0.93</td>
<td>0.86</td>
<td>0.67</td>
<td>0.44</td>
<td>0.22</td>
</tr>
<tr>
<td>1.00E-02</td>
<td>0.94</td>
<td>1.00</td>
<td>0.86</td>
<td>0.77</td>
<td>0.73</td>
<td>0.55</td>
<td>0.22</td>
</tr>
<tr>
<td>1.00E-03</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
</tbody>
</table>

F-Scores in Different Bit Ranges: coMD(3.2k, [-100,100])

<table>
<thead>
<tr>
<th>Bit Range</th>
<th>56~63</th>
<th>48~55</th>
<th>40~47</th>
<th>32~39</th>
<th>24~31</th>
<th>16~23</th>
<th>0~15</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.00E+00</td>
<td>1.00</td>
<td>1.00</td>
<td>0.86</td>
<td>0.77</td>
<td>0.67</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>1.00E-01</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>0.86</td>
<td>0.77</td>
<td>0.40</td>
<td>0.00</td>
</tr>
<tr>
<td>1.00E-02</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>0.93</td>
<td>0.77</td>
<td>0.55</td>
<td>0.00</td>
</tr>
<tr>
<td>1.00E-03</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>0.77</td>
<td>0.67</td>
<td>0.67</td>
<td>0.00</td>
</tr>
</tbody>
</table>

Table 5.1: F-Scores Details under different Tolerance. Shaded parts are the bit ranges that does not result into noticeable errors in the output. Application: miniMD(4k,[-100,-100]) and coMD(3.2k,[-100,-100]).

\[
F\text{- score} = \frac{2 \times TP}{2 \times TP + FP + FN} \quad (5.1)
\]

Where, \( TP \), \( FP \), and \( FN \) are defined as follows: \( TP \) is the number of true positives (i.e, in our context, where a fault is injected and is detected), \( FP \) is the number of false positives (where a fault is not injected yet the algorithm reports its detection), and \( FN \) the number of false negatives (where a fault is injected and is not detected)

The fault injection was done as follows. Data for both applications was stored in double precision numbers. We divided the 64 bit double size into 8 bit ranges, with each bit range covering 8 bits. We inject bit flip into the target range and monitor the behavior of the algorithm, and finally compute the F-score. For each bit range, we run the application for 8 times.
Figure 5.3: Distribution of TP, FP and FN when bit flip is injected in different bit ranges. Application: miniMD(4k, [-100, 100]) and coMD(3.2k, [-100, 100]).

Table 5.1 shows the result we have for both applications. A trade-off we have discussed before involves the choice of the tolerance threshold. We experimented with both applications under different values tolerance threshold. The results show that an appropriate choice of the threshold, if the bit flip occurs in higher order bits, our approach is able to detect the fault easily. Figure 5.3 shows the details of the number of True Positives and False Positives/Negatives. We have nearly 100% true positives and almost no false negatives when the bit flip occurs in the 48th 63th bit for both miniMD and coMD. However, when the bit range is lowered to 32th 47th, we begin to observe more false negatives. As the bit range is becoming lower, we observe more false negatives. This is because when the bit flip occurs in the lower order bit, the impact to the value of the variable becomes smaller. This will result in a less significant value change on the data, which may be lower than the tolerance threshold. The F-score for bit range 0th 21th is very low. While higher values of threshold cause false negatives, there is another problem associated with very low values. If the threshold is small, even though no faults are injected, during the computation, the value will be rounded off, which may trigger a false positive. Therefore, the threshold values should be carefully chosen to obtain best performance. In miniMD, with the current
Table 5.2: F-Scores Details under different Tolerance. Shaded parts are the bit ranges that does not result into noticeable errors in the output. Application: miniMD(2k,[-100,100] and coMD(3.2,[-1,1]).

data size, the best threshold based on the result is 0.1 and the best threshold for coMD is 0.01.

However, it also needs to be noted that not detecting bit flips over insignificant bits is not necessarily a problem, as the final output is not impacted by an undetected bit flip. Hence we evaluate the overall F-score for our approach for unsafe ranges only. If we chose threshold 0.1 in miniMD, we can achieve an F-score of 0.9476. We can also get F-score of 0.9405 under threshold 0.01 for coMD. This result shows that our algorithm is effective since any algorithm with F-score greater than 0.9 is commonly believed to be effective.

We have also evaluated our idea using different data sizes. The results shown in Table 5.2 indicates that data size affects the effectiveness of our algorithm but within a small factor. In both miniMD and coMD, the initial data (atoms location, velocity, force and other values) is randomly generated only to simulate the interaction among the atoms. We have
altered the initial random data to study whether the data set affects the performance or not. The result shows that the initial data set impacts the threshold determination but not the effectiveness of the algorithm. As long as a proper threshold is chosen, the algorithm can still achieve relatively high effectiveness. i.e. F-score of 0.9256 in coMD under threshold value of 1e-5. Another observation we can obtain is that, regardless of the data set size or the initial data values, our approach achieves high performance when the bit flip occurs in the higher order bits. Higher order bit flip is always easily detected in all the experiments. The detailed distribution of TP/FP/FN for Table 5.2 is attached in Figure 5.4.
All of the analysis above involved injecting one bit-flip during the execution of the program. However, in practice, there could be either uncorrelated or correlated bit-flips during the execution. We now experiment with both of these possibilities, and evaluate both how they impact accuracy of the application, and how our techniques can apply to their detection and correction.

To simplify our experiments, we injected both bit-flips at the same iteration. For each combination of the different two bit ranges, we executed the applications 10 times and analyze the result as before. In the first experiment, we flip two bits on randomly chosen bits of the same 64 bit variable, whereas in the second experiment, it is applied randomly across the heap. For each application, we choose the optimal tolerance value that is suggested in Table 5.1

Figure 5.5 shows the overall score. As we can see from the figure, for each bit range, our method performs better when there’s another bit flip involved. As an example, the average F-Score of bit range 32th 39th is 0.8, which is higher than it was for single bit flip (0.77). This could be due to the contribution of the second bit flip - making the value more detectable by the current tolerance value. Flipping two random bits may result into two
value changes. However, by the nature of MVM checker, our method can also apply even if there are multiple errors in the output. Figure 5.6 show the results for random double bit flips. We can observe a slight downgrade compared to the result from the neighboring double bit flips. However, the overall pattern is still similar – as long as the bit flip occurs in the higher bit ranges, our method can effectively catch the bit flip.

5.3.3 Overheads

We evaluated the overhead of our detection and recovery procedure, which includes both C/R and algorithm level recovery, for both miniMD and coMD. The experiment is designed to involve weak scaling, so that we can observe the behavior of the nodes when handling the same amount of the job. We applied our implementation on 1 node, 32 nodes and 64 nodes respectively. The results are shown in Figures 5.7 and 5.8.
From the figure, we can see that the overhead of fault detection and recovery ranges from 3.7% to 6.8% in miniMD, and 6.9% to 7.2% for coMD. In miniMD, we have collected detailed cost for different processes. It is easily observed that, when the number of nodes in the system increases, cost for force computation does not increase much while communication cost and neighbors list cost increases dramatically. As a result, the percentage that force computation takes in the total time cost decrease as the number of nodes increases. This explains why we are getting lower fault detection overhead for larger numbers of nodes (6.8% for single node, 3.7% for 64 node).

For the recovery, we took application level checkpoint to store the atoms information, so that we can reduce the overhead from synchronizing the entire system. The costs for recovery are also very small compared to the execution time. The reason is because MD applications are not memory intensive. For example, in miniMD, we take 5000 atoms in a node. For each checkpoint, the application need only to write the information of 5000 atoms.
atoms, which is a relatively low load. However, the algorithm level recovery is even faster. This can be explained if all of the recovery operations are performed in linear memory space.

5.4 Summary

This chapter has developed an algorithm-based fault tolerance solution for molecular dynamics. We show how we are able to map the key computational kernel of molecular dynamic to matrix vector multiplication, in which the matrix holds the intermediate data, the vector comprises the the coordinate of the atoms, and the final force is the matrix vector product. This has allowed us to adapt existing MVM based solutions to this problem, though additional optimizations are required for efficiency. We have also provided two options to recover when an error is detected. The application can recover from latest checkpoint or can repeat corrupted computation.

We have evaluated the overhead of our approaches using two mini-apps that capture the properties for a full-scale molecular dynamics application – MiniMD and CoMD. Our effectiveness evaluation shows that our method can always achieve an F-score of over 0.9, provided an appropriate tolerance threshold is chosen. The overall overhead of detection and recovery is also always less than 10%.
Chapter 6: Supporting In-Situ analysis with Fault Tolerance

This chapter introduces a methodology that provides fault resiliency for in-situ scientific analysis. We maintain a checkpoint and restart mechanism on Map-Reduce like in-situ platform by utilizing the property of reduction objects during the computation. We discovered that the ordering of reduction does not impact the final result if all the objects are eventually reduced. Based on this observation, we obtained fault resiliency by storing and tracking the progress of reduction. If any node fails, based on the pre-stored information, the system could restart by distributing corresponding reduction workload. The idea was evaluated on Smart. We were able to apply the approach both in time-wise and space-wise for the Smart with reasonable extra overhead.

6.1 In-Situ Analysis

Large number of the scientific applications are running on the systems with co-processors and accelerators, including GPUs and Intel MIC, which have a large number of cores with only small amount of memory per core. This architecture brings a related issue - data movement costs becomes a dominant consideration in designs and operations on a parallel machine. And it is not surprising if this becomes a bottleneck in the future because parallel systems will grow faster in scale with co-processors and accelerators rather than memory.
as the main order need of science and engineering simulations is still large-scale computing powers, and will finally face data movement bottlenecks.

As a solution to this upcoming challenge, in-situ analytics [6,14,52,54,56,71,85,88,90] has emerged as a promising data processing paradigm, and is widely accepted the HPC community. In-Situ analytics is able to utilize simulations and analytics concurrently on the same compute node, performs analysis as soon as the simulated data becomes available. Since analytic results are typically orders of magnitude smaller than the original data, efficient in-situ data processing would avoid a very large amount of (if not completely) expensive data movement compared to the traditional analytics that needs the simulated data to be stored in a persistent storage and perform analysis offline.

6.2 Overview of Smart

Smart supports in-situ processing by accessing simulated data directly from memory in each node, using a variant of the MapReduce API. This API avoids outputting key-value pairs to lower the memory requirements for the analytics phase.

6.2.1 System Overview

Smart processes data as follows. At each compute node (or core), the simulation program generates data at each (or every few) time-step(s) and places this data in the memory. The partitions are considered as the input of Smart analytics jobs, and the jobs are launched from the same code region. The Smart runtime scheduler then processes the partitioned data by blocks, while each block is divided into equal splits for threads to process.

Reduction and combination procedures are performed while processing each split, which are carried out using two important data structures – reduction map and the combination map, respectively. These data structures alleviate the need for storing key-value pairs,
which normally consumes a lot of memory in the traditional MapReduce. Reduction object, which is user defined, holds the accumulated value across all key-value pairs that have the same key. During the reduction procedure, the runtime locates reduction objects in the reduction map for the keys generated for elements, and then reduces the element on its reduction object immediately. On the other hand, in combination procedure, the reduction objects are combined into a single combination map locally, which are to be merged on the master node later on.

Smart provides two in-situ processing modes, referred to as *time-sharing* and *space-sharing*, respectively. Time sharing modes are designed for the case where memory is likely limited, and all cores in the node are used by the simulation program. The idea is to pause the simulation program and consume the simulation output, even avoiding unnecessary data.
copying. Space sharing is designed for many-core systems, and dedicates certain cores for analytics only.

Figure 6.2 describes the workflow of time and space sharing modes. In time sharing, a read pointer on memory space is employed to identify the simulated data from current time-step, making it possible to perform the analysis from the same address. However, the data may be overwritten in the next iteration(s) by the simulation program, hence the analysis must complete before simulation continues. This makes the simulation and analytics programs run alternately, sharing the data.

Space sharing mode, on the other hand, involves an internal circular buffer to cache the data from the simulation. At each time-step, if buffer has available spots, simulation
output data is cached. If space is not available, the simulation pauses for buffer availability. Therefore, in space sharing mode, Smart maximize the utilization of available computation power, while distribution of simulation threads and analytic threads may affect the actual performance, depends on the workload of simulation and analysis.

**Listing 6.1 Histogram Processing in Smart**

```cpp
// Derive a Reduction Object
struct Hist : public RedObj {
    size_t count = 0;
}

// Derive a Scheduler:
class HistogramBuilder : public Scheduler<double> {
    int gen_key(const Chunk& chunk) const override {
        return chunk.start;
    }
    void accumulate(const Chunk& chunk, unique_ptr<RedObj>& red_obj) override {
        if (red_obj == NULL) {
            red_obj.reset(new Hist);
        }
        red_obj->count ++;
    }
    void merge(const Chunk& chunk, unique_ptr<RedObj>& com_obj) override {
        com_obj->count += red_obj->count;
    }
}
```

6.2.2 User Program

To use Smart for in-situ processing, user needs to complete the following two steps: 1) define a reduction object from the pre-implemented base class, and 2) implement gen_key, reduction (accumulate), and combination function.

**gen_key:** generates a key based on the input data element from data chunk.

**accumulate:** reduces the reduction objects that has the same key. The data is reduced to a local reduction map.
**combination**: merges all *reduction objects* from the *reduction map* to the *local combination map*, for further merging into the *global combination map*.

After implementing functions, when Smart launches, the runtime invokes the task scheduler and starts data processing based on the user implemented APIs. The Listing 6.1 shows the Histogram setup as an example application.

### 6.3 Framework Design

This section describes our approach and implementations to support fault-tolerant in-situ analytics.

#### 6.3.1 Possible Solutions

Two common approaches for resilience are either not applicable or inefficient in this context. Replicating the jobs, i.e., running each simulation twice with associated analytics is easy implementation-wise, but very inefficient in terms of resource utilization. Another option could be to use ABFT. This approach can at most apply to the simulation code, and solutions are very application-specific.

C/R is normally considered as a popular and promising solution. Checkpointing refers to the state of the runtime being periodically stored in a persistent storage, whereas recovery implies using the most recent state and continuing from that state. As we stated earlier, checkpointing can be system-level or application-level. System-level checkpointing can be applied to the in-situ analytics case directly, as all aspects of a system state, including communication buffers and I/O channels are stored. However, the main drawback is that checkpoint sizes can be very large. Application-level checkpointing solutions can drastically reduce the I/O requirements. However, while application-level checkpointing for a
simulation code is well understood, applying it to in-situ analysis and the combination of simulation and in-situ analysis is an open question.

6.3.2 Reduction Object Based Solution

We now describe a method for automated application-level checkpointing for in-situ analytics being performed with Smart.

Algorithm 14: Smart Data Processing

1. for all iteration do
2. // Codes for preprocessing...
3. Distribute global_combination_map to combination_map
4. Distribute combination_map to reduction_map;
5. for all chunk in input do
6. // Get key from combination_map
7. key = gen_key((chunk, data, combination_map))
8. accumulate((chunk, data, reduction_map[key]))
9. for all pair(key, reduction_obj) in reduction_map do
10. if combination_map.contains(key) then
11. merge(reduction_obj, combination_map[key]) else
12. combination_mapemplace(reduction_obj)
13. end if
14. end for
15. // Remaining codes for clean ups and special case handlings
16. end for
17. // Output results from combination_map

The main logic of Smart’s data processing is summarized in Algorithm 14. Smart processes simulated data using reduction objects, which are maintained in both reduction map and combination map. In each iteration, for each data element in current split, Smart generates a key – either an existing one in the combination map or a new one, and then accumulate the corresponding value using the reduction map. Therefore, the simulated data is consumed and analyzed (reduced) immediately without being passed through any
downstream users. Finally, the content of the reduction map is merged to the combination map and will eventually be merged into a global combination map for output. This logic is executed in parallel across the system.

Now, consider a set of data element $D$ to be processed in the global dataset. This dataset will be distributed into multiple non-overlapping subsets (called splits) to be processed. Let us suppose the subset in the split $i$ is $D_i$, and the reduction objects for $D_i$ be $Red_{Obj}(D_i)$. Let $Reduce$ be global reduction function that reduces/accumulates all reduction objects in the system, and $Reduce_i$ be the local reduction functions in each split that reduces local data elements only. Then after local reduction, we achieve reduction objects for the split $i$, which can be denoted as $Red_{Obj_i} = Reduce_i(Red_{Obj}(D_i))$.

Now, an important observation is that once a partition has been processed, all important information is in the reduction object, and the values in the partition or any other data structure are not needed for computing or recomputing the final result. Another observation is that the global reduction result will be the same for any other possible way of partitioning over $D$. In other words, if $D_1, D_2, \ldots D_n$ and $D'_1, D'_2, \ldots D'_m$ are two different disjoint partitions over $D$, then

$$Reduce(Red_{Obj_1}, Red_{Obj_2}, \ldots, Red_{Obj_n}) \equiv Reduce(Red_{Obj'_1}, Red_{Obj'_2}, \ldots, Red_{Obj'_n})$$

These observations can be exploited to support Smart with fault tolerance with very low overheads. For each node, the local reduction objects are copied after computations on a certain number of data elements. The set of data elements is marked as processed if their reduction object resulting after computation on them has been backed up.

Now, let us consider a failure occurring in the node $i$. Let $p(D_i)$ denote the set of elements that had been processed before the reduction object was last saved. If $D_i$ is the
data set that node $i$ was originally supposed to process, then to continue the processing, only $D_i - p(D_i)$ needs to be processed to get the full copy of $Red_Obj_i$. The remaining dataset could be processed either on the remaining nodes that did not fail, or on the restarted node $i$, if possible. This satisfies the correction of data processing because of the observations above.

We now discuss how this idea can be further extended to restarting both simulation and analytics. Suppose that all nodes take a checkpoint from the simulation program after every few time-steps, and these checkpoints are coordinated, i.e., all nodes can restart the simulation program from the same time-step. Next, assume, on a given node, a simulation outputs a data segment $S_i$ after the time-step $i$. Let $S_j$ be the last output before the most recent checkpoint of the simulation program is taken. Let $S_k$ be the last segment processed before the reduction object is copied and stored. Now, we have three possibilities:

- If $j = k$, the case is simple. The simulation program starts from the last checkpoint and the analytics program starts processing the output segment $S_{j+1}$ using the last saved copy of the reduction object.

- If $j < k$ – we have the information that data segments till $S_k$ have been processed. Thus, outputs $S_{j+1} \ldots S_k$ will be discarded by the analytic program. Starting with the output segment $S_{k+1}$, analytics code will resume processing, also using the last saved copy of the reduction object.

- If $j > k$ – this case can be challenging to handle. However, as we will describe in the next section, we avoid this possibility for the time-sharing mode and can work around this for the space sharing mode.
### 6.4 Implementation of Smart-FR

In this section, we discuss how the ideas presented in the previous section are implemented as part of the Smart framework, creating a fault tolerance in-situ analysis system Smart-FR.

#### 6.4.1 Smart-FR Overview

Figure 6.3 shows the main workflow of Smart-FR. Smart-FR retains the main runtime functionality of Smart, and adds fault tolerance module with the scheduling and reduction phases. In each time-step (or every few time-steps) simulation code outputs data elements that are fed to analytics engine. At a time-step where checkpoint is scheduled, when computation and communication is finished, application-level checkpoint is invoked by the fault tolerance scheduler (FR Scheduler).
The Smart runtime scheduler then takes this data, evenly distributes it into multiple splits, and proceeds to the data processing stage. As described earlier, each data element is reduced to the reduction object. During the reduction stage, FR scheduler periodically marks the progress of the reduction, and stores the information, along with the processed reduction objects, in stable storage. In fact, if the data elements in each splits are processed in a sequential loop locally, the loop variable itself would be the indicator of reduction progress. After reducing all of the data elements in the split, the runtime begins combination step. Similar backup of intermediate steps is performed, i.e., marking the progress and copying the updated combination map. If all splits are processed and the results are merged to the global combination map, the analytical results are passed to the downstream users as the output of in-situ analysis. In the case that the final output is generated without any failure, the backups of reduction objects and combination maps can be discarded.

6.4.2 Failure Handling

Now let us consider the possibilities when the failure might occur. The node failure could occur 1) Smart scheduling stage after the simulated data are generated, 2) reduction stage, 3) combination stage and 4) any other stage.

Case 1 is easy to handle since the analysis has not even started, and thus the system only needs to read the simulated data from the simulation checkpoint for analysis, and proceed from the checkpoint. Cases 2 and 3 is more complicated since the node fails in the middle of reduction or combination. Consider case 2, where the failure occurs during reduction. If the system needs to restart (as in Smart), the runtime could start by reading the simulated data from the simulation checkpoint as in case 1, and begin scheduling and splitting as usual. However, when starting reduction, the runtime does not need to process
Algorithm 15: Smart-FR Processing with Fault Tolerance

1. for all iteration do
2. // Codes for preprocessing...
3. // Distribute reduction_map and combination_map
4. for all chunk in input do
5. if FR_Scheduler.restart() then
6. // Let Iterator jumps to checkpoint
7. chunkID ← FR_Scheduler.red_progress
8. reduction_map ← FR_Scheduler.reduction_map
9. continue
10. end if
11. // Get key from combination map
12. key = gen_key((chunk, data, combination_map))
13. accumulate((chunk, data, reduction_map[key]))
14. // Record reduction progress and reduction objects
15. FR_Scheduler.red_progress ← chunkID
16. FR_Scheduler.reduction_map ← reduction_map
17. end for
18. for all pair(key, reduction_obj) in reduction_map do
19. if FR_Scheduler.restart() then
20. // Let Iterator jumps to checkpoint
21. pairId ← FR_Scheduler.comb_progress
22. combination_map ← FR_Scheduler.comb_map
23. continue
24. end if
25. if combination_map.contains(key) then
26. merge(reduction_obj, combination_map[key])
27. else
28. combination_map.emplace(reduction_obj)
29. end if
30. end for
31. // Remaining codes for clean ups and special case handlings
32. end for
33. // Output results from combination_map
every each data element from the beginning, since the reduction progress before failure and its updated reduction objects are stored already. The reduction could continue from wherever it stopped and update the unprocessed data, (Algorithm 15, lines 5 to line 10).

Failure in the combination stage is handled in a similar fashion. If the reduction is finished, the iterator will jump to the end of loop condition and skips the reduction. And then, the similar steps are performed with the combination to continue the analysis (Algorithm 15, lines 19 to line 24). Finally, the simulation can restart and continue from its checkpoint.

In case 4, there is no promising approaches available since the intermediate data is not logged. The system has to restart from the latest checkpoint possible. However, the analytical code does not restart from any backups, since the analysis for the checkpoint time-step is over and output already.

6.4.3 Mode Specific Processing

Recall that the Smart has time-sharing mode and space sharing mode available. Some of the fault tolerance functionality is specific to these modes.

**Time Sharing Mode**

In Smart-FR, time sharing mode works mostly similar to the original Smart, since all nodes work on simulation first and then proceed to the analysis. Analytical logic is naturally placed behind, and the system will not continue to the next iteration unless analysis is finished. This protocol makes the fault tolerance mechanism easier, since every data elements produced by the simulation is always consumed in a single iteration. This, in turn, ensures that simply checkpointing the simulated data before any processing is sufficient.
**Space Sharing Mode**

In space sharing mode, computing cores are divided into two groups – simulation cores and analysis cores. Cores that work on simulation are not concerned about how the simulated data is processed – they continue the simulation as long as the container structure circular buffer is available. On the other hand, the analysis cores take the simulated data from the buffer and process them. The analysis begins as soon as there are data available in the circular buffer, which means that they do not need to wait for the simulation finishes.

This mechanism creates coordination issues for checkpointing. Particularly, there can be some data loss if only the backups of reduction/combination and checkpoint of the simulated data are taken. We now describe how this problem is addressed. We checkpoint the simulated data as usual, *i.e.* application level checkpoint is taken right after the simulation cores finishes their jobs. Backups on the analysis are taken in a similar way as in time sharing mode, the processed data is backed up with their updated reduction object. Besides of that, a global copy of circular buffer is also maintained. This copy should always be updated as the circular buffer is modified (consumed) by the analysis cores.

In this way, we ensure that all the simulated data elements are captured in the analytical runtime, either processed or stored in the circular buffer. During the restart, besides of continuing reduction/combination from the backups, the latest copy of circular buffer is also restored. This would ensure that every simulated data elements are reduced in the analysis procedure with no data loss.

### 6.5 Experimental Results

In this section, we evaluate the efficiency of the techniques we have introduced, by comparing the overhead of Smart-FR over Smart. First, we evaluate the overhead that
Smart-FR runs with checkpointing overheads (including application-level checkpointing of simulation and reduction-object checkpointing within Smart). Then, we evaluate the total overhead when a restart is involved.

6.5.1 Environment and Applications

We experimented five analytics applications for time-sharing mode and four analytics applications for the space sharing mode. Our experimental application covers five different classes of today’s in-situ analysis, which are: 1) statistical analysis: histogram, who represents value distribution using equivalent width buckets, 2) visualization: grid aggregation [91], who collects elements in a grid to a single element for multi-resolution visualization, 3) clustering: k-means [96], who groups the elements based on k cluster centers, 4) feature analysis: logistic regression, who computes relationships between a variable and multiple independent variables, and 5) window-based analysis: moving average, who computes the average value in a sliding window.

These analyses could be applied to various simulations – we choose two commonly available mini-apps, which are Heat3D [1] and Lulesh [2]. The experiment is conducted on two different clusters. Time sharing experiment is performed on a multi-core cluster where each node has two quad-core 2.53 GHz Intel(R) Xeon(R) processors, with 12 GB RAM, executing RedHat Enterprise Linux Server release 6.1, and Gigabit ethernet as the interconnect (40GB/s). The simulation program is scaled to all available nodes, in which we have used up to 32 nodes (256 cores) in our experiment. Space sharing experiment is conducted on Stampede Cluster, whose node has an Intel Xeon Phi SE10P coprocessor with 61 cores and 1.1 GHz clock frequency. Each coprocessor contains 8GB of GDDR5 memory, with 8 dual-channel controllers, with a peak memory performance of 320GB/s.
The interconnection uses a FDR 56GB/s InfiniBand network of Mellanox switches, consisting of a fat tree topology of eight core-switches and over 320 leaf switches with a 5/4 oversubscription.

### 6.5.2 Checkpointing Overheads

We measure checkpointing overheads by processing data first with the original Smart, and then with Smart-FR where we take 5 checkpoints in total during the execution (without restarting).

**Time Sharing Comparison**

We have experimented with all five applications on both Lulesh and Heat3D for the time sharing experiment. 800GB data was processed by both simulation programs and the configuration of applications are as follows: 1) *grid aggregation*: grid size set to 800, 2) *histogram*: the number of buckets set to 1000, 3) *logistic regression*: the number of dimensions set to 15, 4) *k-means*: the number of centroids, dimensions set to 8, 4, respectively, and 5) *moving median*: the window size set to 25.

Figures 6.4 and 6.5 show the results of the experiments. From the result, we can see that for all of the applications, checkpointing only has modest overheads, as low as 5.24% (Moving Median with 24 nodes, Lulesh). Overheads are lower for Lulesh than Heat3D, because the ratio between computation and size of data output in the checkpoint is higher.

**Space Sharing Comparison**

In this set of experiments, we let Lulesh to produce 500GB data on Xeon Phi nodes. We use 60 threads for simulation/analysis in the experiment and the remaining one is reserved for communication and scheduling, i.e., serving as a master core. To see the impact that
Figure 6.4: Evaluation of Overhead in Time Sharing Mode with Heat3D
Figure 6.5: Evaluation of Overhead in Time Sharing Mode with Lulesh
core distribution brings to Smart-FR, we varied the simulation/analytic ratio in 5 different modes from 50/10 to 10/50. Here, “m_n” denotes that m threads are used for simulation and n threads are used for analysis. The applications that are used in the experiments are histogram, k-means, moving median, and logistic regression. All application parameters are as same as in time sharing mode.

The result is shown in Figure 6.6. As we can see, similar to the time sharing mode, the checkpoint overhead stays the same regardless of the number of the nodes. The overhead is as low as 2.8% (Moving Median, 50/10), and the average overhead is 6.5%. Even the worst performance is 8.3% (Histogram, 30/30), which is fairly acceptable for in-situ processing. Generally, the overall processing time either increases or remains the same as the number
of simulation cores decrease, which is because the data element simulation and consumption (by analysis) is roughly at an even speed. In the moving median, the processing time increases by a large number when the available analytical core is small (50/10), this would be due to the lack of consumption power, which keeps circular buffer full and most simulation cores are paused waiting for the analysis. This also explains that when increased the number of analysis threads, the performance also increases.

### 6.5.3 Recovery Overheads

For these experiments, we trigger midway through the execution of an application, and more pertinently, after 10% of the iterations passed after the last checkpoint. We report these overhead as slowdown percentages. Figure 6.7 shows a set of experiment with 100GB data processed using 16 nodes in both Heat3D and Lulesh. An obvious observation is that, Smart-FR obtains lower overhead on Lulesh rather than Heat3D. In Lulesh, if without restart, the overhead percentage would be as low as 5.9%, and is no higher than 9.8%, for
K-means and moving median, respectively. In Heat3D, the lowest and highest are 6.2% and 12.06%, for histogram and moving median, respectively. If we put the restart costs together, the difference is even more obvious, while Lulesh stays in the range from 13.8% to 15%, Heat3D’s overhead ranges widely from 16.3% to 27.2%. Note that a large fraction of the restart cost is repeated execution of 10% of the iterations.

We summarize the observations as follows. In Lulesh, more computation is involved in simulation/analysis, as we had discussed earlier. Besides higher checkpointing overhead, there is a lower overhead of reading the checkpointed data. Overall, we consider the slowdowns for both simulations as quite acceptable, and much lower than the only other possible option, which will be to take system-level checkpoints. Also, we have chosen to take checkpoints after every 20% of the execution. We can have lower (higher) recovery costs by checkpointing more (less) frequently, though it will become with higher (lower) checkpointing costs.

6.6 Conclusion

In this chapter, we introduced, implemented, and evaluated fault tolerance solution for in-situ data processing. Our solution is based on reduction-style processing of analytics and addresses the complications of keeping checkpoints of the simulation and the analytics consistent. Our work is implemented in Smart, an in-situ programming system with a Mapreduce-like API. The experimental results show that our work scales well as the original Smart does, with overheads of fault-tolerance as low as 2.8% in space sharing mode and 5.24% in time sharing mode.
Chapter 7: Related Work

This chapter compares our existing work with related research efforts from other group or institutes. We categorize the related works into three groups: Checkpoint approach, Soft Error Tolerant Solutions, and In-Situ Models and their Fault Tolerance, all are the topics we have focused on.

7.1 Checkpoint and Restart

As a traditional yet promising fault tolerant solution, C/R is investigated intensively for many decades [5, 16, 41, 49, 63, 65, 70, 78]. From which, [16, 41, 49] either proposed alternative solutions to reduce the checkpoint size or investigated the benefits from application-level checkpointing.

Application-level fault-tolerance has long been researched, with a particular emphasis on providing API to a checkpointing system. Haines et al. [41] proposed to combine application-level fault tolerance and system-level fault tolerance. Bronevetsky et al. [16] reported the checkpoint sizes could be reduced by as much as 80% through application-level checkpointing. They also enable asynchronous checkpointing in OpenMP. Schulz et al. [79] have proposed a checkpointing scheme that transfers the programs to a pre-processor, so that it could be transformed to be self-checkpointing and self-restartable. These compiler techniques can be enhanced to further automate our approach in the future.
However, none of the earlier work has shown an asynchronous approach for message-passing applications.

There have been other efforts to optimizing application-level or system-level checkpointing. One idea has been to implement checkpointing at multiple levels to optimize I/O pattern and reduce the overheads. Islam et al. [49] proposed a scalable checkpoint system that compresses the checkpoint data across the nodes before writing to storage. Naksinehaboon et al. [65] proposed incremental checkpoint model that reduces full checkpointing overhead by performing a set of incremental checkpoints between two consecutive full checkpoints. Moody et al. [63] proposed a multi-level checkpointing system that focuses on light-weight checkpoints that can handle common failures, but rely on PFS for less common (but more severe) failures. These ideas are all complementary to our idea of using asynchronous uncoordinated checkpointing, and can be combined with our approach in the future. Besides of that,

Asynchronous I/O is a promising approach that has been applied in other contexts in parallel applications. Patrick et al. [70] showed a detailed study of overlapping parallel I/O. Abbasi et al. [5] proposed to move output data from compute nodes to staging or I/O nodes prior to storage, so as to reduce I/O. Similarly, Sato et al. [78] designed a non-blocking checkpointing system, where storage nodes are used to overlap the I/O with computation in other nodes of the system. However, the work was only applied to system-level checkpointing.
7.2 Replication

Replication of processes been another direction of research [39, 66]. This approach does not require any additional effort from programmers, however, it decreases the efficiency of utilization of resources, and increases the costs and power budget. Replication approach is not only applied to processes – Engelmann et al. came to the conclusion that redundancy can increase compute node availability [37]. Similarly, SWIFT is a compiler-based solution to compute duplicated versions of register values and compare them [77]. Duplicating instructions [46] and monitoring the computations [68] and control flow [67] have also been proposed. Ho et al. have developed a cross-layer fault tolerance model [44]. Recent research on MPI tends to apply redundancy to MPI implementations. Fiala and Elliott et al. [34, 39] constructed a MPI protocol that involves communication between three replicas to detect and correct SDCs, while Ni et al. presented a framework that protects from both soft and hard errors by using two replicas and automatic checkpointing [66].

Another method, Shoestring, protects only the instructions that potentially result in SDCs using SWIFT-like instruction level redundancy approach [38].

7.3 Handling Soft Errors

The high performance computing community has been working on solutions to the problems arising soft errors in recent years. Some recent studies have observed and measured the impact of soft errors or SDCs on specific scientific computations [9, 33, 57, 80], while another group of efforts has focused on supporting specific solvers with resiliency to SDCs [45, 81, 83, 84]. [57] presented a detailed report of soft error effects on scientific applications.
Symptom based Solutions

As a relatively inexpensive and effective approach, Symptom (Signature) based on-line soft error detection has been tremendously studied in recent years [11, 12, 25–27]. Di et al proposed a method to predict next-step values for each data point based on the historical values [25], and further improved the detection ability with error-feedback control model [26]. Later on, impact driven detection is proposed [27] to provide user acceptable execution results. We have extensively compared our work against this work. After-loop detectors is proposed in [43], which detects the validity of variables following loop execution [43]. Symptom based approaches could work at the software level [75], hardware level [89], or at a combination of the two. Our work is also an example of this approach, but is novel in doing offline machine learning based on time-series of residual values.

Additional solutions based on programming models have also been designed. For example, Erez et al. have experimented with the notion of containment domains [22], which can confine errors in certain program segments. These efforts require substantial additional effort from programmers.

Algorithm based Solutions

Algorithm-level fault-tolerance solutions also have been a topic of investigation for almost three decades now. The main idea of this approach is to use the information already exist in the running process to perform recovery. In the case that the algorithm itself does not contain such information, it still is possible to be added with minor change of the algorithm [19, 20, 23, 24, 32, 42, 47, 59, 73, 92].

For example, Huang et al. proposed algorithm-based approaches to protect array operations [47], and later, Chen et al. proposed checksum approaches to cover array-intensive
applications [18]. Hoemmen et al. suggested fault tolerant iterative method that can detect and even tolerate soft errors [45] – however, the approach is limited to GMRES methods.

Ding et al. developed a matrix multiplication method that performs checksum during the computation, so as to be able to detect soft errors on GPUs [28]. Other checksum approaches have also been proposed to detect soft errors. For example, Shirvani et al. developed a method involving checksum over memory [82]. Checksums on data structure like trees are based on data structure specific properties [15]. More recently, Tavarageri et al. proposed checking every read and write operations with compiler support to cover a larger fraction of soft errors [86]. The key difference, and advantage, of our approach is its very low overhead, by using signatures that detect most significant problems. The disadvantage of our method is that it does not detect bit flips in low-order bits – however, as our experiments have shown, significant errors can be corrected. Other checksum based methods may work with data structure specific properties, like checksums on trees [15].

There are efforts specifically for linear algebra solvers. Elliott et al. evaluated the impact of soft errors on GMRES [33]. Casas et al. provided good reports of vulnerability of sparse scientific applications to faults, and proposed a general approach to increase the resilience [17]. Shantharam et al. also reported the impact of soft errors on iterative solvers in scientific applications [80]. Sloan et al. proposed an fault detection method for sparse linear algebra based on the observation that many sparse problems are well structured, so that sampling could be done to determine best checks to apply. They also observed that sparse problems have many reusable resources to reduce the redundant computation [83]. Sloan et al. [84] also proposed an algorithm level matrix vector multiplication checker approach, which we have used in our work. Shantharam et al. came up with fault tolerant
preconditioned Conjugated Gradient for sparse problems [81]. Their approach could be applied when the matrix is strictly diagonally dominant. Algorithm based approaches without disk/checkpoint have been proposed by Plank et al. [72] and Chen et al. [20, 21]. Besides of those, Li et al. approached fault tolerance by combining ABFT with different ECC strategies [57].

7.4 In-Situ Analysis Platforms and Approach for Resilience

The high performance computing community is aware of the challenge that the performance gap between I/O and computing power is increasing, and therefore in-situ analysis [52, 54, 94, 101] has attracted much attention from the community. Platform level and application level have been the main two areas that are mainly studied. Studies of in-situ algorithm level includes indexing [51,54], compression [55,102], visualization [50,93,100], and other analysis including feature extraction [56], fractal dimension analysis [88] and etc. Platform studies, on the other hand, include a time sharing platform - GlodRush [98], who runs simulations and analysis codes together in the same node like Smart. On the other hand, space sharing platforms have also been a popular topic, the efforts in this area could be CoDs [95], Damaris [31], and Functional Partitioning [58].

In-transit processing and hybrid processing are proposed in addition to in-situ, where in-transit leverages extra resources and allows online analysis to be moved to staging nodes who are different from simulation nodes. Such platforms include GLEAN [88], JITStager [4], PreDatA [97] and NESSIE [69]. Many platforms support hybrid processing, some examples can be DataSpace [30], FlexIO [99] and ActiveSpaces [29].

Another topic that has drawn much attention recently is to integrate MapReduce to scientific analytics. However, they either do not support in-situ processing, or requires a
specific data format (for MapReduce). Smart [90], which is the platform we implement our approach on, however, bypasses these challenges.

There is no promising solution that provides fault resilience for in-situ processing, due to the constraint that in-situ particularly possesses. However, Bicer et, al. [13] proposed the reduction object based solution for MapReduce-like system FREERIDE-G. Still, FREERIDE-G was not supporting in-situ analysis.
Chapter 8: Conclusions

This is a summary of the contributions of our current work and a description of our future work.

8.1 Contributions

Our main objective is to investigate fault tolerant solutions based on application behaviors and platform configurations. For different classes of applications, we have designed pattern-specific solutions that provide fault resiliency. At the same time, we optimized existing solutions, and discovered fault resiliency for programming platforms. Our contributions can be summarized as following:

- Our initial efforts optimize application level checkpoint and restart, with efficient utilization of overlapping checkpoint workload with runtime computation. By logically partitioning the checkpoint into multiple splits, we designed a scheme to write checkpoint immediate when a split is updated, such that the checkpoint is taken during the computation of other splits. This asynchronous checkpoint manner is shown faster than traditional application level checkpoints in our experiments, and significantly outperform system level checkpointing.

- We further focused on soft errors on scientific applications that are iterative and convergent. In this work, we studied the impact pattern of bit flips to the applications and
discovered that the bit flip in higher order bit ranges creates more significant errors in the final result. We also observed that iterative convergent applications carry a signature when the bit flip occurs, which is an unexpected jump in the residual values. We developed a methodology to detect the bit flip on-line by monitoring the decrement of the residual values for applications that monotonically converge, or window based average value for applications that eventually converge, but does not in a monotonic pattern. The work is evaluated with a set of typical convergent applications and the result shows a relatively stable detection rate and low overheads.

- We also explored algorithm level solutions for applications that involves Newton’s Law computation. Based on the observation of the fault tolerant matrix vector multiplication, we developed a transformation to convert the molecular dynamic simulation computations to a form of matrix vector multiplication. In such way, the computation would be protected in algorithm level without checkpointing extra data. Furthermore, we also proposed a disk free solution to recover from the algorithm itself by taking extra time to seek the error point. Although requiring extra memory, the fault tolerance overhead is shown to be significantly less than traditional checkpoint and restart in time and size.

- We observed that the signatures of soft error in iterative convergent applications are specific to application but independent to the input data. We challenged an automated soft error detection method by adopting machine learning techniques. We train the models using both correct and incorrect inputs for the applications off-line, such that the model could classify a runtime into either correct state or incorrect state. Then we run applications with the models and evaluate the runtime on-line to identify the
correctness of the computation. This method is evaluated with a set of applications, with different untrained inputs and proved to be correct with little overheads.

- Lastly, we constructed fault tolerant framework for Map-Reduce based in-situ analysis platforms. We utilized the properties of key data structures in the programming model and proposed a fault tolerant solution by storing the distribution and progress of reductions. The reduction objects are expected to be reduced eventually, and if any node fails to finish, the system could re-distribute the workload and restart the task. Our solution does not involve a large size of checkpoint since only the reduction progress is stored. We evaluate our work with applications without fault tolerant supports and observed relatively low overhead.

### 8.2 Future Work

Based on the work we have conducted so far, we can extend our work in a few directions. The first possible direction is to investigate the in-memory checkpointing strategy to handle soft and hard errors in parallel systems. Due to the bandwidth of I/O, checkpoint process is still considered a major drawback of fault tolerance. Accelerating checkpoint process would significantly benefit it. The second direction is to make use of machine learning technique on simulation applications to automate the soft error detection.

#### 8.2.1 In-Memory Checkpointing for Soft and Hard Errors in Parallel Systems

Most of the parallel jobs distribute the workload to multiple nodes that are in charge of one or more partitions, with minor read after write conflicts between partitions, which could be easily handled (by duplicating ghost arrays). Traditionally, the nodes take checkpoints into reliable storage. When soft or hard error occurs, the system tries to reload from
previous checkpoint and re-distribute the workload to nodes. The writing and reloading involves expensive storage access and could be avoided. Assuming that the failure occurs in one or few numbers of nodes in the system, re-loading the entire system is wasteful. We could extend the idea in Chapter 2 to the parallel scale in the following way:

- First, rather than applying soft error detection (in Chapter 3, 4) to the system scale, we apply it to each node instead. The consideration is that, the soft error signature in global scale is contributed by signatures in local scale. We argue that there must be signature of errors that could be detected as easily as in the global scale.

- Second, for each node, we define a shadow node from the system to store an extra copy of data. so that the entire system contains one extra copy of the dataset. We maintain a shadow relationship map in master node. During the computation, each node only computes the node it hosts, but not the shadowing data.

- Third, by the time of checkpointing, each node sends the updated copy of data it hosts to the shadowing nodes. To reduce the network congestion, the data could be sent asynchronously, in a manner described in Chapter 2.

- Finally, when a soft error is detected, the affected node could receive the latest copy of its data from the shadowing nodes and restart the computation. Or, if a hard error is occurred, the shadowing node could continue the failed one’s work from its latest copy of data.

This strategy avoids expensive I/O overhead and system-wise restart by maintaining the checkpoint in local memories. It potentially benefits the system since network costs are much more expensive than storage (like memory). We reduce the network costs using asynchronous data transfer, and avoids I/O network by continuing jobs in shadow nodes.
8.2.2 Automating Soft Error Detection in Simulation Applications

Simulation applications are widely used and signature detection is difficult to apply directly. The applications may not converge and the residual signature may either not obvious or affected by multiple factors. However, the idea of applying machine learning technique may be employed here to learn the typical behavior of the data changes. We list the following points that are needed to be considered:

- Residual signature may not be sufficient for non convergent applications. More dimensions are needed for precise training.

- The main dataset (e.g. coordinates) could be introduced to training since this is the first victim of soft errors and errors may propagate fast there.

- Sampling algorithm could be applied to main data set so that we could achieve smaller set of data for training and testing. Sample data will still carry the signature of errors since it may represent propagated errors.

The problems that have been investigated in this dissertation work are limited to error handling in iterative applications and selected programming platforms, and do not involve the issues related to fault tolerance to non-iterative, non-convergent applications and on other platforms like Hadoop, Spark and etc. Our future work would also include the development of different fault tolerance schemes for these application subsets and platforms.
Bibliography


147


[59] Hui Liu, T. Davies, Chong Ding, C. Karlsson, and Zizhong Chen. Algorithm-based recovery for newton’s method without checkpointing. In *Parallel and Distributed*


