Algorithm Based Fault Tolerance: A Perspective from Algorithmic and Communication Characteristics of Parallel Algorithms

Upama Kabir

A thesis
In the Department
of
Computer Science And Software Engineering

Presented in Partial Fulfillment of the Requirements
For the Degree of
Doctor of Philosophy (Computer Science) at
Concordia University
Montréal, Québec, Canada

October 2017

© Upama Kabir, 2017

CONCORDIA UNIVERSITY
School of Graduate Studies

This is to certify that the thesis prepared

By: **Upama Kabir**

Entitled: **Algorithm Based Fault Tolerance: A Perspective from Algorithmic and Communication Characteristics of Parallel Algorithms**

and submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy (Computer Science)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final Examining Committee:

______________________ Chair
Dr. Nizar Bouguila

______________________ External Examiner
Dr. Parimala Thulasiraman

______________________ External to Program
Dr. Nawwaf Kharma

______________________ Examiner
Dr. Sudhir Mudur

______________________ Examiner
Drs. Todd Eavis

______________________ Supervisor
Dr. Dhrubojyoti Goswami

Approved by ______________________
Dr. Volker Haarslev, Graduate Program Director

December 11, 2017

______________________ ______________________

Date of Defence

Dr. Amir Asif, Dean
Faculty of Engineering and Computer Science

ABSTRACT

Algorithm Based Fault Tolerance: A Perspective from Algorithmic and Communication Characteristics of Parallel Algorithms

Upama Kabir, Ph.D.
Concordia University, 2017

Checkpoint and recovery cost imposed by checkpoint/restart (CP/R) is a crucial performance issue for high-performance computing (HPC) applications. In comparison, Algorithm-Based Fault Tolerance (ABFT) is a promising fault tolerance method with low recovery overhead, but it suffers from the inadequacy of universal applicability, i.e., tied to a specific application or algorithm. Till date, providing fault tolerance for matrix-based algorithms for linear systems has been the research focus of ABFT schemes. As a consequence, it necessitates a comprehensive exploration of ABFT research to widen its scope to other types of parallel algorithms and applications. In this thesis, we go beyond traditional ABFT and focus on other types of parallel applications not covered by traditional ABFT. In that regard, rather than an emphasis on a single application at a time, we consider the algorithmic and communication characteristics of a class of parallel applications to design efficient fault tolerance and recovery strategies for that class of parallel applications. The communication characteristics determine how to distributively replicate the fault recovery data (we call it the *critical data*) of a process, and the algorithmic characteristics determine what the application-specific data is to be replicated to minimize fault tolerance and recovery cost. Based on communication characteristics, parallel algorithms can be broadly classified as (i) embarrassingly parallel algorithms, where processes have infrequent or rare interactions, and (ii) communication-intensive parallel algorithms, where processes have significant interactions. In this thesis, through different case studies, we design ABFT for these two categories of algorithms by considering their algorithmic and communication characteristics. Analysis of these parallel algorithms reveals that a process contains sufficient information that can help to rebuild a computational state if any failure occurs during the computation. We define this information as critical data, the minimal application-level data required to be saved (securely) so

that a failed process can be fully recovered from a most recent consistent state using this fault recovery data. How the communication dependencies among processes are utilized to replicate fault recovery data is directly related to the system's fault tolerance performance. We propose ABFT for parallel search algorithms, which belong to the class of embarrassingly parallel algorithms. Parallel search algorithms are the well-known solution techniques for discrete optimization problems (DOP). DOP covers a broad class of (parallel) applications from search problems in AI to computer games, e.g., Chess and various games, traveling salesman problem, various AI search problems. As a case study, we choose the parallel iterative deepening A* (PIDA*) algorithm and integrate application-level fault tolerance with the algorithm by replicating critical data periodically to make it resilient. In the category of communication-intensive algorithms, we choose Dynamic programming (DP) which is a widely used algorithm paradigm for optimization problems. We choose parallel DP algorithm as a case study and propose ABFT for such applications. We present a detailed analysis of the characteristics of parallel DP algorithms and show that the algorithmic features reduce the cardinality of critical data into a single data in case of n-data dependent task. We demonstrate the idea with two popular DP class of applications: (i) the traveling salesman problem (TSP), and (ii) the longest common subsequence (LCS) problem. Minimal storage and recovery overhead are the prime concern in FT design. On that regard, we demonstrate that further optimization in critical data is possible for particular DP class of problems, where the degree of dependency for a subproblem is small and fixed at each iteration. We discuss it with the 0/1 knapsack problem as a case study and propose an ABFT scheme where, instead of replicating the critical data, we replicate a bit-vector flag in peer process's memory which is later used to rebuild the lost data of a failed process. Theoretical and experimental results demonstrate that our proposed methods perform significantly better than the conventional CP/R in terms of fault tolerance and recovery overheads, and also in storage overhead in the presence of single and multiple simultaneous failures.

Acknowledgments

I would like to express my profound appreciation to many people who supported me during my Ph.D. and who helped me to complete my thesis. Their generous support made this dissertation possible.

Firstly, I would like to express my sincere gratitude to my supervisor Dr. Dhrubajyoti Goswami for his invaluable guidance and continuous support throughout my Ph.D. study. In particular, he has assisted me to grasp research techniques and to have a solid understanding of how to draw research questions and dig into the research area. Further, my sincere thankfulness goes to him for his insightful directions, necessary revisions, and constructive suggestions on the preparation of our research papers. I am truly indebted to him for his knowledge, thoughts, and friendship.

Besides my supervisors, I would like to thank my committee members, Dr. Sudhir Mudur, Drs. Todd Eavis, and Dr. Nawwaf Kharma, for their insightful comments and encouragement, to widen my research from various perspectives.

I would like to acknowledge Calcul Quebec and Compute Canada to facilitate an extreme supercomputing environment for academic studies. In my research, most of the computations were made on the supercomputer Briarée from Université de Montréal, Colosse from Laval Université and Guillimin from McGill University managed by Calcul Québec and Compute Canada. The operation of this supercomputer is funded by the Canada Foundation for Innovation (CFI), the ministère de l'Économie, de la Science et de l'innovation du Québec (MESI) and the Fonds de recherche du Québec - Nature et technologies (FRQ-NT).

I gratefully acknowledge the funding sources that made my Ph.D. work possible. I was funded by the NSERC Strategic Grant and Concordia University.

Furthermore, I would like to thank my student colleagues and friends for lending their hands and sparing time in needs. Admittedly, I would remember the time spent together and would recall the happiness, sorrows, and cooperation during my journey

at Concordia University.

I would like to thank my husband Dr. Mosaddek Hossain Kamal Tushar and my daughter Anwesha Kamal for their continuous support, understanding, and assistance whenever I needed them throughout my research work. Specially, I am indebted to my husband, who assists me in this journey with his profound knowledge in the field of computer science and engineering.

Last but not the least, I would like to thank my parents Professor Ahmad Kabir and Nilufar Begum, and to my sisters Shaily Kabir and Dr. Mitra Kabir. Their unending love and strong faith always inspired me and bolstered my confidence throughout my journey abroad. I am always grateful to them for their encouragement and support.

Contents

List of Figures x

List of Tables xii

Abbreviation xiii

1 Introduction 1

1.1 Overview and Objectives 1

1.2 Motivation and Problem Statements 5

1.3 Contribution 7

1.4 Thesis Outline 10

2 Literature Review and Preliminaries 11

2.1 Message Passing Parallel Program 12

2.2 Algorithm Based Fault Tolerance 17

2.3 Types of Failure in HPC Systems 21

2.4 Discrete Optimization Problem 22

2.5 Parallel Search Algorithms 23

2.6 Related Work 25

2.6.1 Recent Trends and Failures Scenario in High-performance Computing 25

2.6.2 Causes of Failures 27

2.6.3 Fault Tolerance in Cluster 28

3 ABFT for Embarrassingly Parallel Algorithms 37

3.1 Motivation 38

3.2 Algorithmic Characteristics of PIDA* 38

3.3 Fault Tolerance for Parallel Iterative Deepening A* (PIDA*) Search Algorithm . . . 41
3.3.1 Critical data . . . 42
3.3.2 Fault Tolerant PIDA* . . . 45
3.3.3 Failure Recovery . . . 48
3.3.4 Consistent state . . . 50
3.3.5 Example of FTPIDA* Protocol . . . 52
3.4 Performance analysis . . . 55
3.4.1 Timing Overhead of FTPIDA* . . . 55
3.5 Numerical results . . . 57
3.6 Conclusion . . . 61

4 ABFT for Communication Intensive Parallel Algorithm 62
4.1 Motivation . . . 63
4.2 Characteristics of Parallel Dynamic Programming . . . 64
4.2.1 DP Algorithmic Characteristics . . . 65
4.2.2 Data Dependency on Different DP Formulation . . . 65
4.2.3 Parallelism in DP . . . 70
4.2.4 Parallel DP Communication Characteristics . . . 73
4.2.5 General Computation and Communication Characteristics . . . 76
4.3 Fault Tolerance for Parallel Dynamic Programming . . . 77
4.3.1 Critical Data . . . 78
4.3.2 Consistent State . . . 80
4.3.3 Fault Tolerance Protocol . . . 81
4.3.4 Fault Recovery Protocol . . . 83
4.4 Performance Analysis . . . 87
4.4.1 An analysis of message optimization during fault tolerance . . . 87
4.4.2 An analysis of extra message overhead during fault tolerance . . . 89
4.4.3 An analysis of timing overhead during fault tolerance and recovery 91
4.5 Experimental Evaluation . . . 93
4.5.1 Experimental results of TSP . . . 93
4.5.2 Experimental results of LCS . . . 97
4.6 Conclusion . . . 100

5 Optimization in Fault Tolerance **101**

5.1 Motivation 101

5.2 0/1 Knapsack Problem 102

5.3 Optimization in Fault Tolerance 104

5.3.1 An example of Fault tolerance 108

5.4 Fault Recovery 110

5.5 Theoretical Analysis 111

5.5.1 An analysis of extra message overhead 111

5.6 Experimental Evaluation 113

5.7 Conclusion 117

6 Discussion and Future Work **118**

6.1 Future Work 121

Bibliography **123**

List of Figures

1.1 High performance Computing System Performance over time by Top500 [41] . . . 2
1.2 Major Architectural Categories for HPC Systems by Top500 [41] . . . 3
2.1 Message Passing System . . . 12
2.2 Send/ Receive Operation of message passing Paradigm . . . 14
2.3 Consistent and Inconsistent Global State of Message Passing Distributed System . . . 16
2.4 An Example of Checkpointing . . . 17
2.5 An Example of Failure and Recovery . . . 17
2.6 Basic Working Principle of ABFT . . . 19
2.7 Checksum Matrix Multiplication [63] . . . 20
2.8 Error in Full Checksum Matrix [63] . . . 21
3.1 A sample search tree and local DFS stack of a process P_i at time t_i for the 8-puzzle problem . . . 39
3.2 Search tree and local stack of P_i at time t_j . . . 43
3.3 Critical data ($B \setminus A$ of Lemma 3) of P_i at time t_j (refer to figure 3.1 and figure 3.2) . . . 44
3.4 Example of FTPIDA* algorithm . . . 53
3.5 Fault Tolerance Overhead (without fault): FTPIDA* vs. CP/R . . . 58
3.6 Failure Recovery Overhead: FTPIDA* vs. CP/R . . . 59
3.7 Performance Gain: FTPIDA* vs. CP/R . . . 60
4.1 Computation Dependency in Different DP Problems . . . 70
4.2 Fine-grain and Coarse-grain Parallel Distribution. Cells marked with the same number execute in a parallel manner . . . 72
4.3 Dynamic Programming (DP) Table for LCS of strings $X = BACBAD$ and $Y = ABAZDC$ . . . 73

4.4 DP Solution of a Sample TSP Problem . . . 74
4.5 Communication characteristics of parallel DP . . . 75
4.6 General Algorithmic and Communication Characteristics . . . 77
4.7 General Algorithmic Characteristics of DP with $n:1$ Cardinality . . . 78
4.8 Recovery Overhead of Proposed ABFT with a Single Failure . . . 94
4.9 Fault Tolerance Overhead without fault (ABFT vs. CP/R) . . . 95
4.10 Fault Tolerance and Recovery Overhead With Multiple Simultaneous Failure (ABFT vs. CP/R) . . . 95
4.11 Performance Improvement (Our proposed ABFT vs. CP/R) . . . 96
4.12 Extra Message Overhead for ABFT . . . 97
4.13 Fault Tolerance Overhead (ABFT vs. CP/R) . . . 99
4.14 Recovery Overhead with Multiple Simultaneous Failures . . . 99
5.1 Data dependency and the DP table in a parallel solution to the 0/1 Knapsack problem . . . 103
5.2 Fautlt Tolerance for 0/1 Knapsack Problem . . . 109
5.3 Recovery Overhead with single failure. . . . 114
5.4 Recovery Overhead with simultaneous multiple failures. . . . 115
5.5 Fault Tolerance Overhead (without fault) . . . 116
5.6 Performance Improvement (Our scheme vs. Diskless checkpointing) . . . 116
5.7 Failure Recovery Overhead . . . 117

List of Tables

1.1 Top 5 HPC systems by TOP500 2

3.1 Fault Recovery Information (FRI) of Processes During Execution of FTPIDA* 54

3.2 Backup data size for FTPIDA* and CP/R 60

4.1 Configuration of the Clusters [74] 93

4.2 Gain in Critical data 97

4.3 Gain in Critical Data Size for LCS problem 98

5.1 Data Dependency Among the Processes 108

5.2 Experimental configuration 113

5.3 Extra message overhead 114

Abbreviations

ABFT	Algorithm Based Fault Tolerance
ALC	Application Level Checkpointing
CD	Critical Data
CFDR	Computer Failure Data Repository
CFI	Canada Foundation for Innovation
CP/R	Checkpoint and Restart
DOP	Discrete Optimization Problem
DP	Dynamic Programming
FFT	Fast Fourier Transform
FRI	Fault Recovery Information
FRM	Fault Recovery Message
FRQ˙NT	Fonds de recherche du Québec - Nature et technologies
FT	Fault Tolerance
FTPIDA*	Fault Tolerant PIDA*
HPC	High Performance Computing
ILP	Integer Linear Programming
LANL	Los Alamos National Laboratory
LANL	Los Alamos National Laboratory
LCS	Graphics Processing Unit
LCS	Longest Common Susequence

LLNL — Laurence Livermore National Laboratory

LLNL — Laurence Livermore National Laboratory

MCM — Matrix Chain Multiplication

MPI — Message Passing Interface

MPMD — Multiple Program Multiple Data

MPP — Massive Parallel Processor

MTTF — Mean Time to Failure

NCSA — National Contre for Supercomputing Application

NERSC — National Energy Research Scientific Computing Center

NERSC — National Energy Research Scientific Computing Center

PBFS — Parallel Best First Search

PDFBB — Parallel Depth First Branch and Bound

PDFS — Parallel Depth First Search

PIDA* — Parallel Iterative Deepening A*

PNNL — Pacific Northwest National Laboratory

SLC — System Level Checkpointing

SNL — Sandia National Laboratory

SNL — Sandia National Laboratory

SPMD — Single Program Multiple Data

TSP — Travelling Salesman Problem

Chapter 1

Introduction

1.1 Overview and Objectives

Failure becomes a reality rather than a rarity with the growing scale of high-performance computing (HPC) systems. HPC systems refer to the system that consists of thousands of processors and millions of cores, used to solve scientific, engineering and research problems which are computationally complex and data-intensive. As with time, the HPC generation moves from terascale to petascale and ready to enter into the exascale era (expected by 2021); it incorporates more complexity by the increasing number of cores or sockets per processor. The TOP500 list [41] provides the list of today's top 500 most powerful computing systems used for high-performance scientific applications. Systems are ranked according to their performance of the Linpack benchmark, which solves a dense system of linear equations. Initially, HPC systems were marked by vector processors, but later massive parallel processors (MPP) became the major contributors but from early 2000's clusters built with off-the-shelf components have gained attention as a computing platforms for end-users of HPC computing systems (Figure 1.1: advancement in HPC systems performance since 1993, figure 1.2: architectural trend in HPC systems and table 1.1 presents the list

Rank	Name	Total Cores	Rmax [PFlop/s]
1	Sunway TaihuLight	10649600	93.0
2	Tianhe-2	3120000	33.8
3	Piz Daint	361760	19.6
4	Titan	560640	17.6
5	Sequoia	1572864	17.2

Table 1.1: Top 5 HPC systems by TOP500

of the first five powerful supercomputers, published on June, 2017 by TOP500 [41])

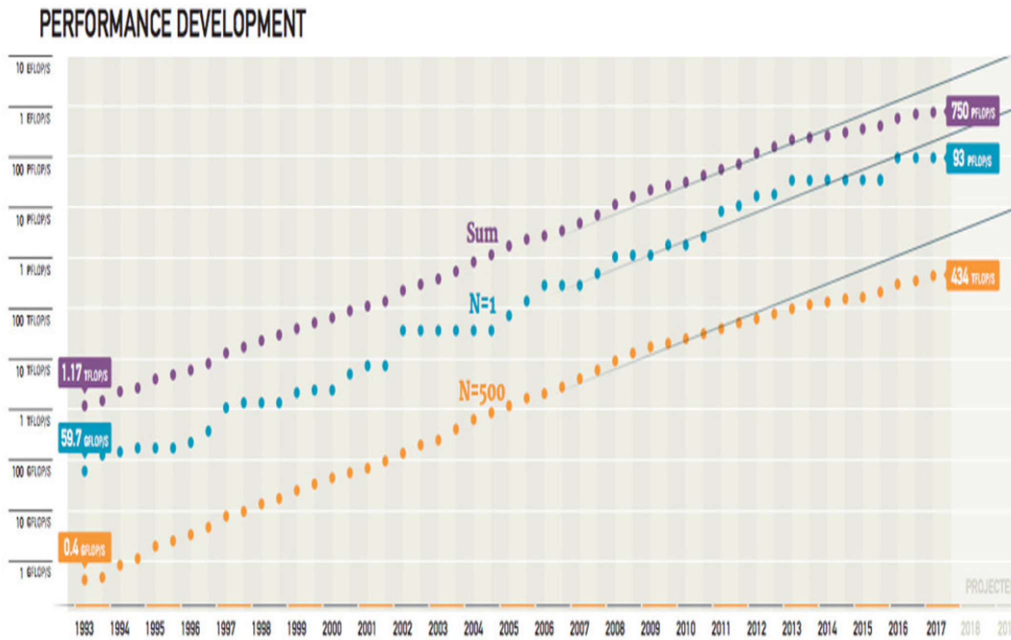


Figure 1.1: High performance Computing System Performance over time by Top500 [41]

Research has shown that the failure rate of HPC systems increases with the increasing number of sockets which according to top500 is doubling every year in the best supercomputers [21, 54]. These systems are highly complex in architecture and are prone to failure because of their complex and dense architecture. While the growth in system scale enhances the performance, meanwhile it also significantly hampers reliability of the system. As the systems move from petascale to exascale, the number of system components will be increasing faster than component reliability. Therefore,

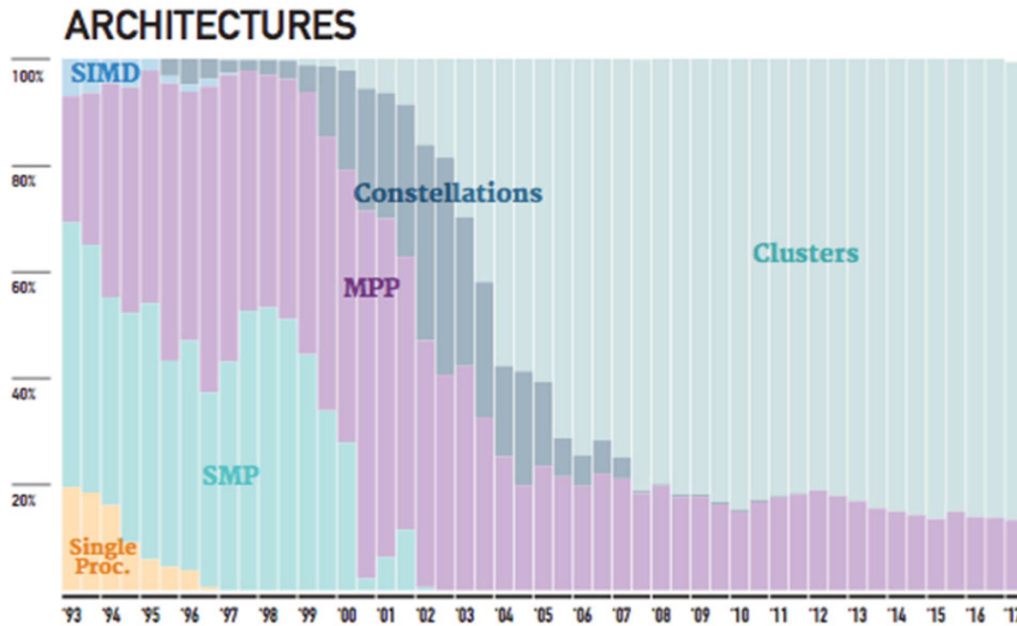


Figure 1.2: Major Architectural Categories for HPC Systems by Top500 [41]

failure is inevitable and frequent. Notably, system reliability decreases proportionally with the increasing scale. Research finds that current petascale system faces multiple failures each day [22]. For example, a detailed study on Blue Waters, a petascale supercomputer (with a total of 724,480 cores and 26 PBytes storage and with 13.1 petaflops speed) at the National Center for Supercomputing Applications (NCSA), at the University of Illinois at Urbana-Champaign for a period of 261 days showed that a failure occurs (across all categories) every 4.2 h, while the system suffered system-wide outages approximately every 160 hours [71]. It is predicted that with the current growth rate in system size, the mean time to failure (MTTF) of such HPC systems might drop to a few hours [21, 69]. As a consequence, long-running HPC applications executing for several days or even months can face frequent interruptions, which in turn decrease the overall performance and increase their overall execution times. Projection from current large petascale systems predicts that an exascale system will face multiple failures per day and the MTTF is between 35 to 39 minutes [10]. Another

study assumed the MTTF for an exascale system become 20 minutes [46]. Adding to these sources of failure, recent trends in HPC systems also represent off-the shelf clusters which are an appealing and cost-effective solution to run computationally intensive scientific applications but far less reliable. The concept of supercomputing is redefined by these off-the shelf clusters because of their excellent price-performance ratios, which accelerates failure probability with increasing growth rate of such commodity HPC systems. It is evident that long running scientific applications execute for several days or even months may face frequent interruptions as MTTF can even be much shorter than the total execution time of such an application, which in turn decreases the overall performance and increase their overall execution times.

The HPC applications typically encounter fail-stop failures where the whole system stops functioning even with a single process or processor failure [80]. Fault tolerance (FT) becomes a necessity for long running applications to avoid computational wastage. Traditional CP/R is the commonly used fault tolerant method for HPC applications to tolerate fail-stop failures. In this case, every process periodically saves its state on a stable storage at a globally coordinated or uncoordinated manner. If a failure occurs, all the processes use the last checkpoints to resume their execution from a globally consistent state, instead of restarting the computation from scratch. Checkpointing provides fault tolerance to HPC applications transparently without modifying the applications. Though conceptually checkpoint is very straightforward and applicable to wide range of applications, it is not scalable to large scale as it introduces enormous overhead. A major source of overhead is the time taken to store massive checkpoint data in the disk by a large number of processes periodically [92, 98]. To resume the execution of such an application after failure, all the processes have to roll back to the previous consistent state, even with the failure of one process.

Variants of conventional CP/R like incremental checkpointing, diskless checkpointing, etc. were introduced, but still, they suffer from the checkpoint overhead. An alternative to CP/R is Algorithm based Fault Tolerance (ABFT) proposed by Huang and Abraham [63], which is found to be an effective solution for failure recovery in large scale systems. Unlike traditional CP/R, which checkpoints system state, ABFT only saves application state and, as a result, it has reduced overhead. A more detailed review of these different approaches is presented in the next chapter.

1.2 Motivation and Problem Statements

Checkpointing provides fault tolerance to HPC applications transparently without modifying the applications. Though conceptually checkpoint is very simple, still it is not scalable to exascale systems because of its several limitations. The main limitation is the enormous checkpoint and restart-time which may exceed the MTTF. Another significant limitation is the time wasted in rolling back all the running processes to a previous (consistent) checkpoint state for the failure of a single process to resume the execution of a failed application. For instance, with a single process failure, restarting the failed application causes to kill 999999 running processes just for 1 process failure in a $1M$ processes application, which ultimately hinders the system's progress and overall performance. Moreover, the checkpoint size of large applications where tens of thousands of processes are involved is in the order of several tens of TeraBytes [7]. It is observable that the average time for the checkpoint is in between 25 to 30 minutes depends on the system scale [21, 23]. Research shows that if the system has to restart with the full memory snapshot of all the processes from stable storage, it may cost 20 to 30 minutes to resume the execution and can cost 40 minutes to 1 hour to accomplish the entire checkpoint recovery methodology [21]. For the exascale system, it is catastrophic: an exascale system which may contain more than 200,000

socket [10] may face system failures several times per hour as the MTTF reduces proportionally with the number of CPU sockets in the system. Additionally, as the checkpoint and restart time of an exascale machine is close to MTTF, the system either spends most of the time in checkpointing or stuck in a state where the system is being constantly restarted with a new fault, with negligible progress in execution.

HPC systems with increasing scale face failure more frequently and traditional CP/R techniques will be inefficient to handle such failures for exascale systems[21, 98, 97, 54, 22]. Fault tolerance method for HPC applications with low recovery overhead is an active research problem. These concerns have persuaded the exploration of new fault tolerance techniques rather than conventional CP/R. The lack of appropriate fault tolerance solutions is expected to be a major problem at exascale. To make efficient utilization of such systems, measures should be taken so that applications can bear partial failures and continue execution rather than stop operation and restart from the last consistent state. This concern motivates to design for alternative and efficient fault tolerant techniques that do not employ checkpoint and rollback recovery as their premise.

Algorithm Based Fault Tolerant (ABFT) is a promising fault tolerance method for a class of HPC applications and can tolerate partial failures with very low overheads. ABFT approaches promise unparalleled scalability and performance in failure-prone environments. This is in sharp contrast with CP/R, which suffers from increasing overhead with system size. ABFT is found to be more scalable [92], and there have been several explorations for large scale systems[21, 69, 7, 51, 98, 97]. With an increasing failure rate with the growing number of processors, ABFT is found to be an appropriate fault-tolerance technique that outperforms conventional CP/R for current petascale and future exascale systems for certain parallel applications[14, 12, 40, 98].

In spite of its promise, ABFT still has limited applicability because of its present

scope: since it was proposed by Huang and Abraham to detect and correct permanent and transient errors in certain matrix operations on systolic arrays, till date, the ABFT schemes have been primarily focused on different matrix based computations for the linear system to handle the bit-flip error in memory, soft-error or hard-error (process crash). Consequently, it requires an extensive research on ABFT to broaden its scope to other types of parallel applications, apart from the linear systems, which are quite widely used and cover a vast majority of parallel applications.

In this thesis, we go beyond traditional ABFT and focus on other types of parallel applications not covered by traditional ABFT. More specifically, we focus on communication and algorithmic characteristics of classes of parallel applications to determine how to replicate fault recovery data and what data to replicate to minimize cost. The approach facilitates designing efficient fault tolerance and recovery strategies for a class of parallel applications, rather than for a single application at a time. Our focus is on tolerating fail-stop failures where the failed process stops working and all data associated with the failed process is lost. This type of failures is common in today's large computing systems such as high-end clusters with thousands of nodes. The research contributions are discussed in the following and are elaborated in the following chapters.

1.3 Contribution

Two of the most crucial parts of a fault tolerance and recovery protocol are: (i) how to checkpoint and store the fault recovery data with minimum cost, and (ii) in the case of a failure, how to recover the lost data of a failed process with minimum possible overhead. With that notion, we utilize the algorithmic and communication characteristics of parallel algorithms to design their fault tolerance and recovery strategies with minimum possible overhead. Parallel applications can be broadly classified as follows

based on their communication characteristics: (i) embarrassingly parallel applications, where processes have infrequent or rare interactions, and (ii) communication-intensive parallel applications, where processes have significant interactions. How the communication dependencies among processes are utilized to replicate fault recovery data is directly related to the system's fault tolerance performance. Furthermore, analysis of these parallel algorithms reveals that the algorithm itself contains some implicit information that can help to rebuild a computational state with the help of others if any failure occurs during the computation. Conventional CP/R cannot address these issues because it is unaware of the characteristics of the application; whereas considering these features in designing fault tolerance and recovery schemes can eliminate the bottlenecks in large scale HPC system.

The main contributions of the thesis are summarized as follows:

- Traditional approaches to ABFT mainly focus on matrix-based algorithms where data lost in some matrix cell(s) due to failures can be recovered with the help of data in the other cell(s). However, these types of applications cover only a limited spectrum of a vast range of parallel applications. In this research, we broaden the scope of ABFT to other types of parallel applications that may not involve matrix based computations. More specifically, we investigate a general and methodical approach to ABFT for parallel applications which can be classified based on their algorithmic and communication characteristics. Instead of designing ABFT for one application at a time, we demonstrate that a standard ABFT scheme can be developed for a class of parallel applications with similar algorithmic and communication characteristics. In our approach, the communication characteristic of an application determines how to replicate the "fault recovery" data (we call it *critical data*), and the algorithmic characteristics of an application determine what the application-specific critical data

is. We perform the theoretical and empirical evaluations for two important classes of parallel applications, which are significantly different from the focus of traditional ABFT. This is further elaborated in the following.

- We show that extraction of algorithmic features and identification of critical data for the parallel algorithm are pivotal in the design of algorithm based fault tolerance. We demonstrate the ideas by developing a fault tolerance and recovery scheme for parallel search algorithms, which belong to the class of embarrassingly parallel algorithms where the subtasks have infrequent interaction with each other. Parallel search algorithms are used to solve discrete optimization problems (DOP). We choose parallel iterative deepening A* (PIDA*) as a case study, which is an exhaustive search algorithm to find the best solution. We augment the algorithm by inserting application level fault tolerance that uses periodic replication of critical data; we call it the fault-tolerant PIDA* (FTPIDA*). Subsequently, we prove the correctness of FTPIDA* and show that FTPIDA* performs better than CP/R.

- Next, we demonstrate that communication dependency among the subtasks plays a significant role in designing algorithm based fault tolerance. With that notion, we propose a fault tolerance and recovery scheme for communication intensive parallel applications which utilize the communication dependencies of processes to replicate the checkpointed application-data of a process to its peer process's memory. The algorithmic characteristics determine which application data to checkpoint on neighbor process's memory. We illustrate our ideas using parallel dynamic programming (DP) algorithm(s) as a case study. We present a detailed analysis of the characteristics of parallel DP algorithms and show that the algorithmic features reduce the cardinality of critical data into a single data in case of a n data dependent task. We demonstrate the idea with two

well-known DP class of problems: (i) traveling salesman problem (TSP), and (ii) longest common subsequence (LCS) problem. We show that for all the cases the proposed fault tolerance method is exceeding conventional CP/R regarding recovery and memory overhead.

- Optimal size of critical data is a prime concern to minimize the overhead in fault tolerance. With that notion, we show that further optimization in critical data is possible when a task has limited and lower degree of dependency cardinality with other subtasks in the case of a DP type of problem. We evaluate the approach by applying it to a popular DP application: the solution to the 0/1 knapsack problem. Experimental results demonstrate that the proposed method performs better than CP/R.

1.4 Thesis Outline

The remainder of the thesis is organized as follows. Chapter 2 presents the related work and background concepts of the techniques used to develop the fault tolerance solutions. An efficient algorithmic fault tolerance strategy for embarrassingly parallel applications is provided in Chapter 3. Chapter 4 presents the fault tolerance solution for the communication intensive parallel applications. Optimization in the critical data in case of communication intensive parallel applications is presented in chapter 5. Chapter 6 concludes the thesis with a discussion of the future research directions.

Chapter 2

Literature Review and Preliminaries

High-performance computing (HPC) systems are highly complex systems consisting of thousands of processors and millions of cores to solve computationally complex applications in science, engineering, and large-scale data analytics. They cover a broad range of systems, from supercomputers to cluster to cloud to the grid. In this chapter, firstly, we will discuss the notion of the message passing paradigm which is considered the de-facto standard for designing parallel applications on HPC systems. Failure probability of HPC systems increases with increasing scale; consequently, long-running HPC applications encounter frequent failures which ultimately decrease overall performance and increase total execution time. Fault tolerance becomes a necessity for these applications to avoid computational wastage. In this context, we will discuss checkpoint/restart and algorithm based fault tolerance which are commonly used fault tolerance techniques for current HPC systems and which we apply to this research work. Finally, we will cite contemporary and relevant research works related to our finding. This chapter summarizes the state of the art in fault tolerance techniques for HPC systems and thus is the foundation for the subsequent chapters.

2.1 Message Passing Parallel Program

The message passing programming paradigm is the most widely used platform for the parallel computer with a distributed address space where each processor has a local memory to which it has exclusive access. There exists no concept of global memory. Figure 2.1 illustrates the logical view of the system that supports the message passing paradigm. This paradigm presumes the system with distributed memory architecture i.e. each process has its address space and process to process communication occurs through message passing. Processes perform a task in collaboration with each other by exchanging messages. The message passing approach is naturally well adapted for the distributed memory cluster. However, it can be used on shared memory multi-processors, networks of workstations or even uni-processor systems. Conceptually, in

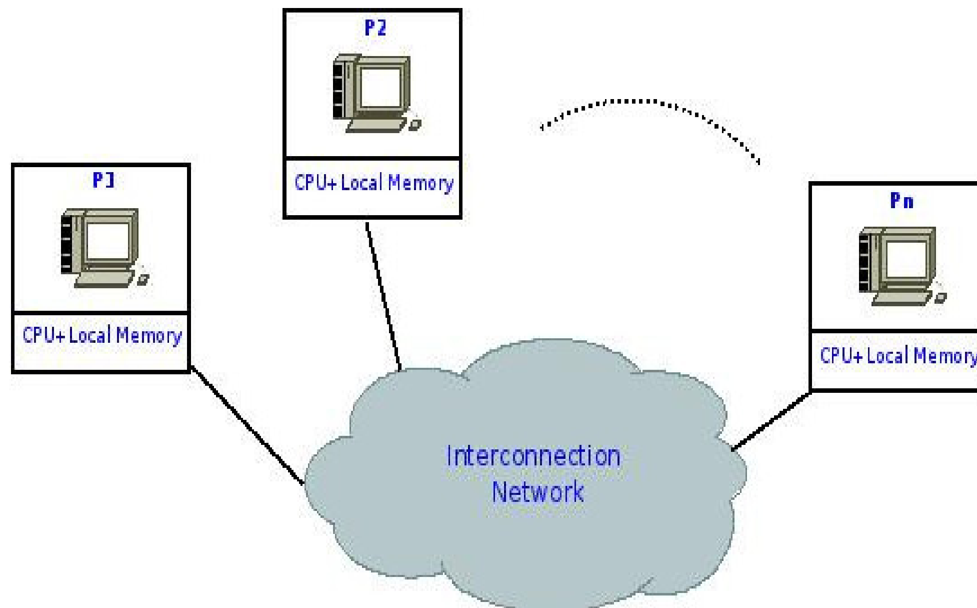


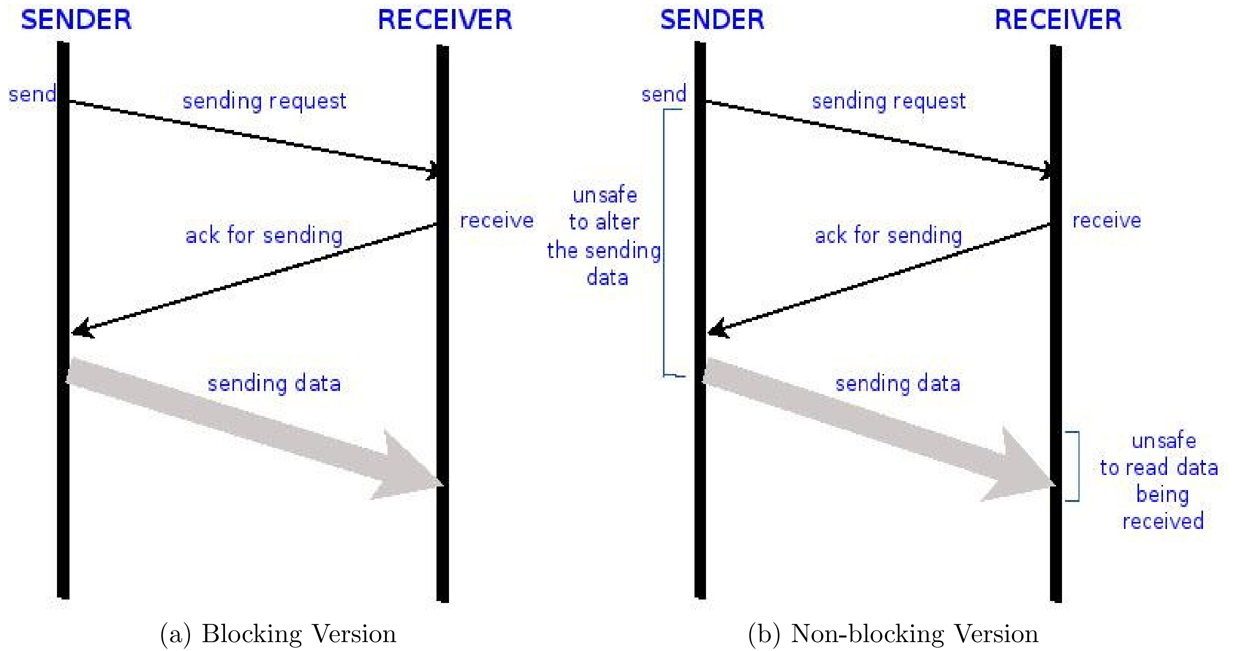
Figure 2.1: Message Passing System

message passing paradigm, each of the processes could execute a different program (MPMD, multiple program multiple data), but for the convenience of the program design, it is usually written with the notion of a single program, multiple data (SPMD) approach. In SPMD approach, most processes deal with the same code. This code

can be designed in a completely asynchronous or loosely synchronous manner. Interestingly, the programmer is solely responsible for incorporating parallelism inside the code by explicitly distributing the data and computational load among the processes. The programmer accomplishes this by analyzing the serial algorithm and consecutively by finding the concurrency point inside it.

Send (send()) and receive (receive()) are the primitives for interaction among the processes in message passing. Send/Receive operations can be blocking and non-blocking. In the blocking scenario, send() cannot complete until it gets an acknowledgment of the corresponding receive() from the receiver end. On the other hand, in non-blocking send() returns the control to the sender process, and the sender can perform any other computation without waiting for any acknowledgement from the receiver. Later the sender can check the status of the non-blocking operation. Moreover, the user must take precautions not to modify the message/data until the exchange is fully completed. Both the protocols have buffered and non-buffered versions. In the case of the buffered version, a designated buffer is used between sender and receiver for data exchange. In the non-buffered approach, the communication happens through a 3-way handshake between the peer processors. Figure 2.2 displays the working principle of both blocking and non-blocking send/receive operations with a non-buffered approach.

The communication mode among the processes belongs to two broad categories: (i) point-to-point communication, (ii) collective communication. Examples of point-to-point are blocking and non-blocking send and receive operations. Collective communication includes broadcast, gather, and all-to-all operations. This communication is accomplished through a standard message passing communication library. Communication libraries often provide a broad set of communication functions to support different point-to-point transfers and global communication operations. The



(a) Blocking Version

(b) Non-blocking Version

Figure 2.2: Send/ Receive Operation of message passing Paradigm

most popular portable communication library for message passing paradigm is MPI (Message-Passing Interface) [57]. We are using MPI to implement all the proposed message passing applications of this thesis work.

Checkpoint/Restart (CP/R) is the most common fault tolerance technology for message-passing high-performance parallel and distributed computing systems. A message passing system is as a system that is composed of a set of application processes which communicate through messages. Each process has a set of totally ordered local events including message sending and receiving. Moreover, the messages among different processes also maintain a partial ordering, and they are represented by Lamport's happened-before relation (usually denoted as $\rightarrow$) [68]. The processes achieve fault tolerance by saving recovery information periodically during failure-free execution in stable storage. The recovery information is the state of the process, which is called the checkpoint of the process. Processes are prone to failures, in which case they lose all of their state information. Recovery can happen when the process's

computational state can be restored to a fault-free state by rolling back all processes to the most recent checkpoint state.

A system state of a message passing system is the global state of the system which is a collection of local states of all the participating processes and states of the message channels within the system. The local checkpoint is the local state of a process which is necessary for that process to resume its operation after the failure. Similarly, the global checkpoint is considered as the global state of the system. A system's state is said to be consistent if there exists no such message in the set of checkpoints, which reflects a receiving but no corresponding sending of that message [68]. This message is known as an "orphan message." Figure 2.3 demonstrates a sample execution of a message-passing distributed system consisting of 4 processes with (i) the consistent global state and (ii) the inconsistent global state. Figure 2.3(a) represents the consistent global state as message m_{23} has been sent by process P_2 but not yet received. On the other hand, figure 2.3(b) displays the inconsistent state, as the local state of process P_4 shows a message reception m_{34} from process P_3 but the local state of P_3 reflects m_{34} is not sent yet. Here, m_{34} is an orphan message. In the global state of a system, a situation may arise when it shows messages that have been sent but not yet received, known as the in-transit message. In such cases, these messages should be recorded for the purpose of rebuilding the message channel after failure. In figure 2.3(a), m_{23} is an example of an in-transit message. A correct recovery protocol is required to eventually recover the system to a consistent global state with all the recorded information. Checkpoint-based recovery system has three broad categories: (i) uncoordinated checkpointing, (ii) coordinated checkpointing and (iii) communication-induced checkpointing. Here, we describe the basic idea of checkpointing with the help of a suitable example of coordinated checkpointing using the concept of distributed snapshot [68]. In figure 2.4, initially the initiator (in

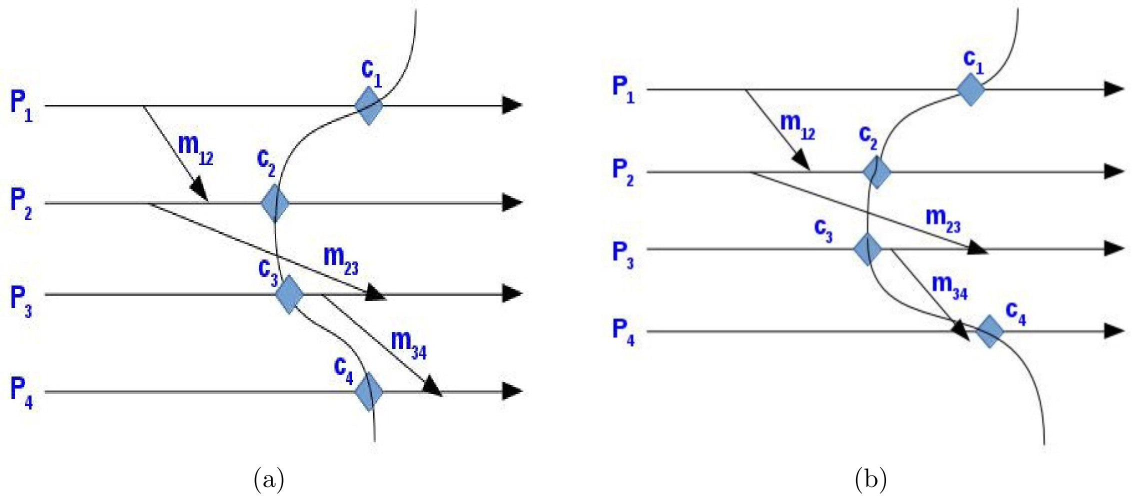


Figure 2.3: Consistent and Inconsistent Global State of Message Passing Distributed System

this case, process P_3) takes a checkpoint and broadcasts a marker message (checkpoint request message) to all other active processes P_1, P_2 and P_4 . Each process then takes its checkpoint after receiving the first marker message and rebroadcasts the marker message to all other processes before sending any application messages. Usually we denote a checkpoint as $c_{i,n}$, where it represents the checkpoint of process P_i with sequence number n. Moreover, this index can serve to trigger a process to take a checkpoint if the receiver's local checkpoint index is lower than the piggybacked checkpoint index from the marker message. Failure recovery ensures that upon failure, the system will be recovered in a former consistent global state, not necessarily the one that has occurred just before the failure. Figure 2.5 illustrates the failure recovery for a system with 4 processes, where P_1 has failed, and it rollbacks itself to its most recent checkpoint $c_{1,2}$ and all other processes rollback themselves to the checkpoint that gives the most recent recoverable global consistent state. Here, the recovered global consistent state becomes $\{c_{1,2}, c_{2,2}, c_{3,1}, c_{4,1}\}$.

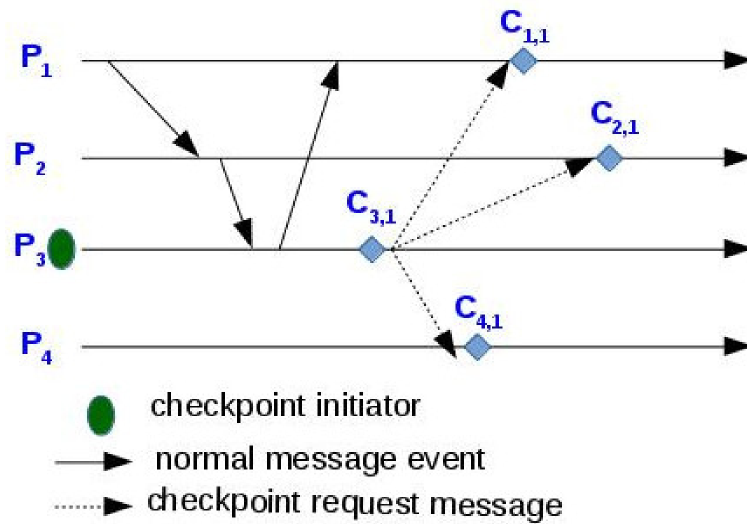


Figure 2.4: An Example of Checkpointing

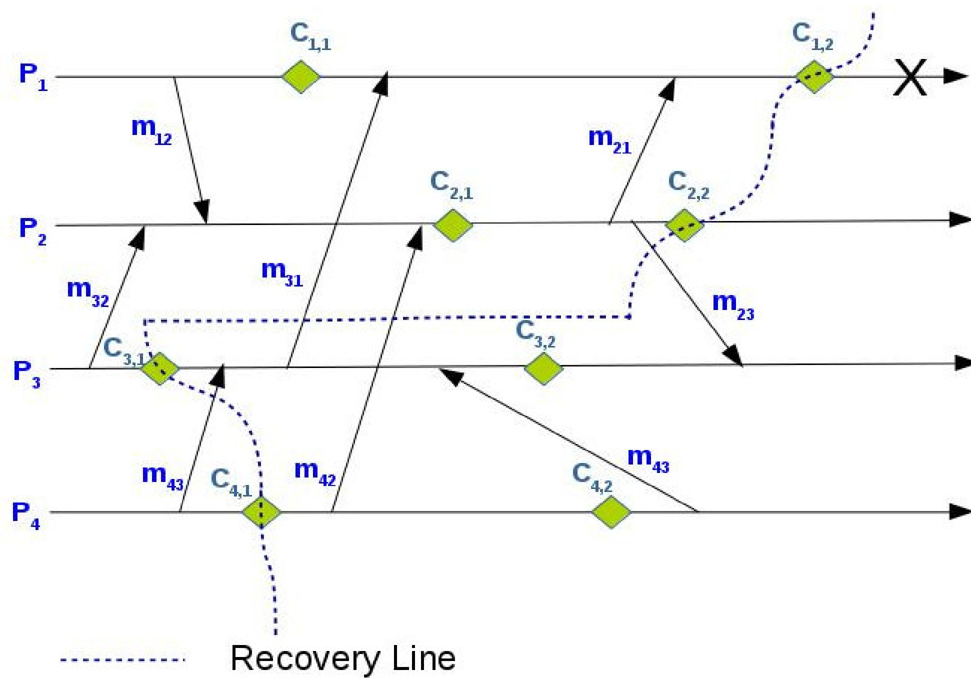


Figure 2.5: An Example of Failure and Recovery

2.2 Algorithm Based Fault Tolerance

Algorithm-based fault tolerance (ABFT) was first proposed by Huang and Abraham for detecting and correcting permanent or transient hardware failures on systolic array [63]. Here, fault tolerance is attained by integrating the fault tolerance scheme

with the existing algorithm. The detection and correction of errors which occur during computation with multiple processors is accomplished after the completion of the computation with little overhead. The general idea of ABFT is to introduce information redundancy in the data and maintain this redundancy during the calculation. The operation applied to the input matrix can then be extended to apply over the input matrix and its extended columns at the same time, maintaining the checksum relation between data in a row and the corresponding checksum column(s). If a failure hits processes during the computation, the data hosted by these processes is lost. However, in theory, the checksum relation being preserved, if enough information survived the failure between the initial data held by the surviving processes and the checksum columns, a simple inversion of the checksum function is sufficient to reconstruct the missing data and pursue the operation.

The basic steps of ABFT to compute the matrix multiplication of two matrix, $C = X * Y$ are: (1) encode the original matrix before the computation which is known as checksum matrix (2) design the algorithms in such a way that it can work with encoded matrix and (3) distribute the computation steps of the redesigned algorithm among multiple processors in such a way that failure of any processor will affect as little data as possible. ABFT algorithm takes the checksum matrix as an input operand and produces an encoded output matrix. The encoded output matrix helps to detect, locate and correct the error after the completion of the computation if it occurs during computation, illustrated in figure 2.6. Encoding of the matrix is composed of two checksum vectors: one for the rows of the matrix and one for the columns. A checksum matrix is constructed from the original matrix with an additional row or column that contains a checksum vector [63].

- Column checksum matrix X_c is a $(n+1)$-by-m matrix containing n rows of matrix X in its first n rows and a column summation vector in $(n+1)$st row.

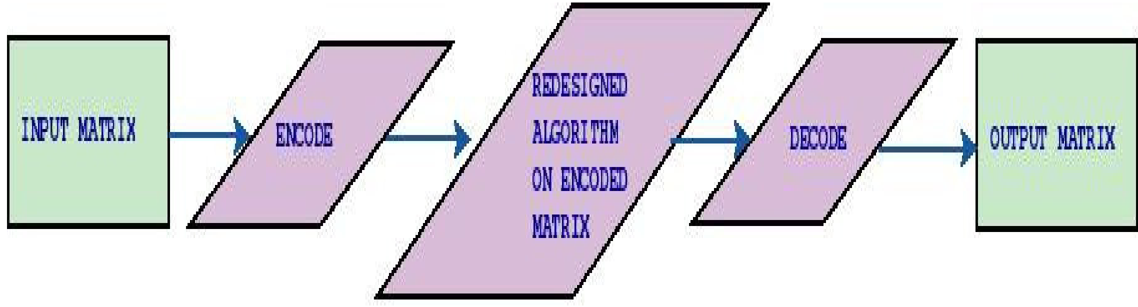


Figure 2.6: Basic Working Principle of ABFT

Each element of the column summation vector is represented as:

$$x_{n+1,j} = \sum_{i=1}^{n} x_{i,j} \quad for \;\; 1 \leq j \leq m \tag{2.1}$$

which gives the encoded matrix

$$X_c = \begin{bmatrix} x_{11} & x_{12} & \dots & x_{1m} \\ x_{21} & x_{22} & \dots & x_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n1} & x_{n2} & \dots & x_{nm} \\ \sum_{i=1}^{n} x_{i1} & \sum_{i=1}^{n} x_{i2} & \dots & \sum_{i=1}^{n} x_{im} \end{bmatrix}$$

- Row checksum matrix Y_r is a n-by-$(m+1)$ matrix containing m columns of matrix Y in its first m rows and a row summation vector in $(m+1)$st row. Each element of the row summation vector is represented as:

$$y_{i,m+1} = \sum_{j=1}^{m} y_{i,j} \quad for \;\; 1 \leq i \leq n \tag{2.2}$$

$$Y_r = \begin{bmatrix} y_{11} & y_{12} & \dots & y_{1n} & \sum_{j=1}^{m} y_{1j} \\ y_{21} & y_{22} & \dots & y_{2n} & \sum_{j=1}^{m} y_{2j} \\ \vdots & \vdots & \vdots & \vdots & \\ y_{n1} & y_{n2} & \dots & y_{nn} & \sum_{j=1}^{m} y_{nj} \end{bmatrix}$$

- Full checksum matrix M_fc is a $(n+1)$-by-$(m+1)$ matrix which is the row checksum matrix of the column checksum matrix X_c or the column checksum matrix of the row checksum matrix Y_r .

$$M_{fc} = \begin{bmatrix} m_{11} & m_{12} & \dots & m_{1n} & \sum_{j=1}^{n} m_{1j} \\ m_{21} & m_{22} & \dots & m_{2n} & \sum_{j=1}^{n} m_{2j} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ m_{m1} & m_{m2} & \dots & m_{mn} & \sum_{j=1}^{n} m_{mj} \\ \sum_{i=1}^{m} m_{i1} & \sum_{i=1}^{m} m_{m2} & \dots & \sum_{i=1}^{m} m_{in} & \sum_{i=1}^{m} \sum_{j=1}^{n} m_{ij} \end{bmatrix}$$

Figure 2.7 presents the organization of checksum matrices for the matrix product operation of two matrices X and Y. An error is detected by computing the sum

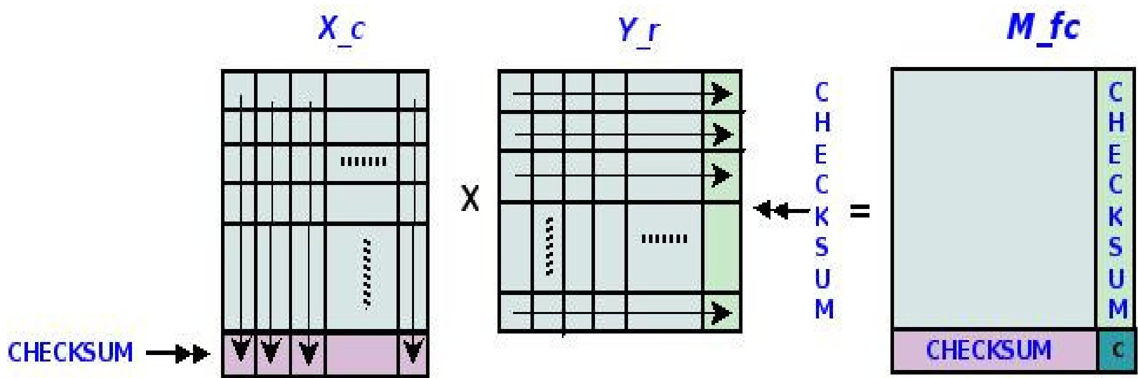


Figure 2.7: Checksum Matrix Multiplication [63]

(s_{rc}) of information in each row or column and comparing it with the corresponding checksum (m_{rc}) in the summation vector. An inequality in the comparison is the identifier of an inconsistent row or column. The location of the error is determined

from the intersection of the inconsistent row and column of M_{fc}, shown in figure 2.8. The identified error is fixed by adding the difference between s_{rc} and m_{rc} with the erroneous element of the information.

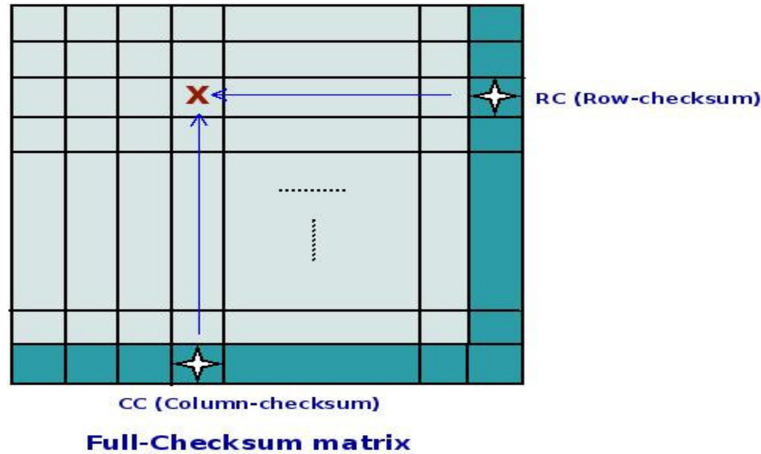


Figure 2.8: Error in Full Checksum Matrix [63]

2.3 Types of Failure in HPC Systems

There exist many types of errors, faults, and failures for HPC systems, which vary in nature and action. Some of the faults cause a fatal interruption in execution; others are corrupting the data in a silent way which is unrecognized by the system. Based on the impact of the failures, faults or errors in HPC system is mainly categorized as the hard fault and soft fault. The hard error causes the program to abort. On the other hand, soft errors corrupt the state of a computing system but not its overall functionality. Soft errors pose a serious issue when they lead to Silent Data Corruption (SDC) in user applications. If undetected by the application, a single SDC can corrupt data causing applications to output incorrect results, malfunction or hang. Both types of faults errors introduce different failure modes in HPC systems. Mainly there are two failure modes exist in literature: (i) fail-stop, and (ii) fail-continue. In

the fail-stop mode, the failed process completely stops working, and all data associated with the failed process are lost. On the contrary, in fail-continue failures, the program execution continues, but the computation results from the computing cannot be trusted anymore.

2.4 Discrete Optimization Problem

Discrete optimization problem (DOP) is an optimization problem over discrete values [65]. It has enormous research and practical impact as several application domains belong to DOP. Scheduling problems, network design problems, inventory management, transportation problems, optimal layout of VLSI chips, robot motion planning, game playing, and computer vision are common application areas which fall into DOP [49, 3]. A discrete optimization problem consists of a set of candidate solutions S and the cost function f. The objective of DOP is to find the feasible or optimal solution among a set of candidate solutions considering all the constraints of the problem. Formally a DOP can be denoted as (S, f, Σ), where $S = \{S_1, S_2, ..., S_n\}$ is the set of all candidate solutions and $f(S_i)$ is the cost function for each candidate solution and Σ is the set of constraints that must be satisfied by the candidate solutions of the problem. The goal is to find an optimal solution such that

$$f(S_{opt}) \leq f(S_i) \qquad s.t \quad \Sigma \tag{2.3}$$

for all $S_i \, \epsilon \, S$.

Integer linear programming (ILP), branch-and-bound, heuristics, dynamic programming (DP), and approximation algorithms are the methods used to solve DOP. As DOP problems belong to NP-hard problems, parallel processing is a way to obtain reasonable performances for those problems where there is a need for real-time

solutions or optimal solutions are required. For a more detailed explanation, see [3].

2.5 Parallel Search Algorithms

Many of the discrete optimization problems have a natural characterization in terms of a graph. The set of all solutions $S = \{S_1, S_2, ..., S_n\}$ represents vertices of the graph, and each edge contains the cost to reach a designated vertex. As it can be portrayed as a graph search or tree search problem, solution of this problem is to find a path from a node to a goal node or to find a minimum cost path from a node to one of the multiple goal nodes. Different branch and bound search algorithms are common means to solve DOP [3, 56].

Technically, DOP belongs to the class of NP-hard problems [3, 53].It is not possible to get a polynomial time solution for such problems except using an exponential number of processors, but for several problems there exists some heuristic search algorithms that solve subproblems in polynomial time. Parallel search techniques can be employed to solve discrete optimization problems illustrated as graph search problems in polynomial time using parallel machines. Followings are different parallel search algorithms.

- Parallel Depth First Search (PDFS)

  Parallel depth first search is accomplished by dividing the search space among a multiple number of processors. Each processor executes the sequential depth-first search (DFS) on its disjoint part of the search space in parallel. When a processor has completed its search, it requests for unexplored search space from other processors. All the processors quit when one of them found the goal node or all the processors finish searching the search space without finding any target node [75, 67, 78].

- Parallel Iterative Deepening A* (PIDA*)

  Iterative deepening A* (IDA*) executes incrementally deepening cost bound depth first search with respect to heuristic function. IDA* carries out cost bound DFS on every iteration until the cost to expand a node n, $[f(n) = g(n)+h(n)] > t(n)$ where $t(n)$ is a threshold value. In the beginning, the heuristic value of the root is the initial cost bound, and it increments to the minimum value that excels the previous one till the solution of the problem is found [34].

  In parallel IDA*, all the processors use the same cost bound, and each processor executes parallel DFS on its search space using the same cost bound in each iteration. After completion of each iteration, one designated processor defines cost bound for the next iteration. All the processors resume parallel DFS with this new cost bound, and this process continues until the goal node is found [3, 75, 76, 31].

- Parallel Depth First Branch and Bound (PDFBB)

  Parallel DFBB works in the same manner of PDFS but with some extra information. Here, all processors have to keep information about current best solution along with the information needed for PDFS. The current best solution information can be held either in shared memory or in distributed memory which helps to prune the search space by identifying the worst solution path as compared to the current best [3, 75, 67].

- Parallel Best First Search (PBFS)

  In best first search algorithm, heuristic value gives direction to which part of the search space likely contains the solution of the problem. The well-known BFS technique is A* algorithm. A* defines a heuristic function for any node n which express the optimal cost path through node n.

The heuristic function $f(n) = g(n) + h(n)$ where $g(n)$ represents the cost from an initial node to node n and $h(n)$ represents the heuristic estimation to reach from node n to goal node [3, 56, 66].

It keeps two lists named "open" and "closed." "Open list" contains the unexpanded node, sorted according to the lower heuristic value and "closed list" contains the nodes those are expanded and the successors of the expanded node are kept in the open list. The node with lower heuristic value is picked from the open list, expanded, successors are stored in the open list with respect to heuristic value and the expanded node is kept in closed list. This process continues until goal node is found.

In parallel BFS, the simplest method is to allow different processors to expand different current best nodes on the open list, stored in a global space [3, 66]. This idea is known as the centralized strategy where global open list introduces contention. The alternative is the distributed approach where each processor has its local open list. At first, the search space (by expanding some initial nodes) is statically divided and distributed to different nodes. Now each node works on its local list with the similar fashion that described above [3, 56, 66].

2.6 Related Work

2.6.1 Recent Trends and Failures Scenario in High-performance Computing

Today's high-performance computing (HPC) systems consist of thousands of processors and millions of cores. These systems are highly complex in architecture and are highly parallel with millions of processing units, tera or petabytes of memory

and networking elements. The TOP500 provides statistics on the 500 most powerful commercially available computer systems known to us today[41]. As with time, the HPC system moves from terascale to petascale to exascale (expected by 2021); it incorporates more complexity by the increasing number of cores or sockets per processor. Recent trends in HPC systems also represent off-the-shelf clusters which are an appealing and cost-effective solution to run computationally intensive scientific applications. Clusters of general purpose, inexpensive machines interconnected by high-speed communication networks are currently widely used for parallel computation and as backend processing servers for a growing number of commercial applications. The concept of supercomputing is redefined by these off-the-shelf clusters because of their excellent price-performance ratios.

More incorporation increases failure rate, and a corresponding decrease in mean time to failure (MTTF). MTTF is decreasing further than disk checkpoint time and also than recovery time with this increasing scale[85]. More incorporation increases failure rate, which can be between 20 to 1100 per year, introducing more fault tolerance overhead to the system [83, 81]. Research has shown that the failure rate of HPC increases with the increasing number of sockets which according to top500 [54] is doubling every year in the best supercomputers [21]. The introduction of multicore machines also accelerates the failure frequency [21]. Bluewaters came to market in 2011 with 300,000 CPU cores with eight cores per CPU, so almost 40,000 sockets in total. Sequoia, which came out in 2012, has 1,000,000 CPU where each CPU has 8 cores, giving a total of 125,000 sockets [21]. There is speculation that if the number of sockets increases in this fashion, then the MTTF of the machine can become minutes for top parallel machines [21]. Resilience is the biggest challenge with these large-scale computers as the systems grow more complex day by day and the failure becomes a frequent event instead of an exceptional event [95, 83, 81, 55, 21, 82, 54].

Researchers have analyzed the data collected from LANL(from 1996 to 2005) composed of 22 different systems and have presented statistics for the causes of failures, the failure rate of the system and individual node, the time between failures and also the time that was taken to repair them [81]. These statistics show that the failure rate varies from system to system: from 17 to 1159 failures per year per system. The data shows that the average failure rate of an individual node has a maximum value of 3 per year and irrespective of any failure type, the average repair time of a system is nearly 6 hours. The prediction is that if a system has expanded at a rate of doubling the number of cores/chips in every 30 months, then the overall system utilization drops drastically almost to zero by 2013 as the system is spending more time for a checkpoint or recovering from failures [83, 81]. Research data has shown that less than 4% of the total nodes were responsible for 70% of the overall failures of the system [79, 59].

2.6.2 Causes of Failures

The computer failure data repository (CFDR) aims at accelerating research on system reliability by filling the nearly empty collection of public data with detailed failure data from a variety of large production systems.

The computer failure data repository (CFDR) provided by Usenix shows the failure scenario of some well-known existing HPC systems from 1996 to 2009 [5]. Los Alamos National Laboratory (LANL), National Energy Research Scientific Computing Center (NERSC), Pacific Northwest National Laboratory, Sandia National Laboratory (SNL) and Laurence Livermore National Laboratory (LLNL) in the USA are the contributors of this data. All of the systems as mentioned earlier are massively parallel.

Analysis of the statistical data released by Usenix [5] has identified that the causes

of failure in supercomputers are mainly hardware failures, software failures, network failure, human, environmental and some unknown reasons [83, 81, 55, 41]. Research on data from LLNL and SNL is comprised of 5 supercomputers' data: Blue Gene/L (with 131072 processors), Red Storm (with 10880 processors), Thunderbird (with 9024 processors), Spirit (with 1028 processors), and Liberty (with 512 processors). This research has concluded that most of the failure is due to software (64%), hardware(19%) [72]. The wider job is the job which needs more nodes may notably influence the failure rate [100].Among all the causes identified, hardware failure is the most significant one with more than 50% of the overall failures, and the next one is software failure with 20% of the overall failures [81, 21, 55]. As the analysis took place on a different set of systems, different result were obtainedd, regarding the main reason for failures [22].

2.6.3 Fault Tolerance in Cluster

Fault tolerance is an important design objective in the HPC community. The primary purpose of fault-tolerance is to produce correct results in the presence of errors. The right measure of fault tolerance will help highly extensive computation, and long-running applications to continue their execution. A full review on fault tolerance in HPC systems is out of the scope of this thesis. In this thesis, we will mainly discuss the research contribution in fault tolerance for parallel and distributed systems in two major classes: (i) checkpoint and restart (CP/R) and (ii) algorithm-based fault tolerance (ABFT).

Despite its over-all benefits, as CP/R introduces huge recovery overhead, research has shown interest in an alternative to CP/R for high-performance computing(HPC). One of the promising alternatives is the algorithm-based fault tolerance (ABFT) technique which is still being explored. ABFT was first introduced by Huang and

Abraham [63], and the aim was to detect and correct the failures (which appear during execution) after the completion of the computation. Some research has been going on this arena.

2.6.3.1 Checkpoint and Restart

The most commonly used fault tolerant technique for the high-performance parallel computer is checkpoint/restart (CP/R), which periodically stores the system state or process's state in a stable storage in a failure-free execution, and the application can then recover itself after failure by using this saved information. A detailed survey on checkpoint rollback-recovery protocol for distributed systems can be found in [45, 44], which also highlights the three subcategories of checkpointing: (i) coordinated, (ii) uncoordinated and (ii) communication-induced checkpointing. Checkpointing has two fundamental approaches: (i) system-level checkpointing (SLC) and (ii) application-level checkpointing (ALC). System level checkpointing is done in the operating system level; like BLCR [61, 62], it provides an entirely transparent checkpoint of the whole process. System-level checkpointing requires saving the entire memory, and thus the cost of checkpointing is directly proportional to the memory footprint of the process. The most significant advantage of this scheme is its transparency. Still, it has some limitations: it is system-dependent, not portable and there is a huge checkpoint and recovery overhead. For instance, complete system-level checkpointing of a parallel system with thousands of cores can be impractical and expensive as the total checkpoint size of the thousands of nodes is in the order of terabytes [91].

In application-level checkpointing(ALC), an application developer has full freedom as to where to take a checkpoint. The application programmer inserts the checkpoint directive and calls in a position in the program where he wants to take the checkpoint and also selects what data needs to be saved to recover the process

state. ALC helps to reduce the checkpoint size overhead but it puts the full responsibility on the programmer, and moreover, it complicates the application program coding [60, 89]. A non-blocking, coordinated application level checkpointing for an MPI program is described in [18], where application states of the MPI processes are saved by inserting a checkpoint directive inside the MPI program, and this shows that the overhead is minimal. As checkpoint and recovery cost is a major concern in system-level checkpointing for large scale systems, different solutions to reduce the size of the checkpoints at the application level can be found in [86, 32, 33]. In spite of its advantages, the fact that ALC suffers from the lack of user transparency, is a large drawback.

Diskless-checkpointing is a desirable alternative to the traditional checkpointing [73]. In this model, checkpointing happens in two steps: firstly, the application processor saves its state (address space, registers, etc.) on its memory rather than on the stable disk and, secondly, the checkpoint processor encodes the in-memory checkpoint and stores this encoding checkpoint in the memory, which is later used to recover the failed processors. When a failure occurs, the nonfailed application processor rollback with its stored in-memory checkpoint and the checkpoint processors with the encoding checkpoint help to recover the failed application processors. Though this addresses the overhead performance of disk-based checkpointing, it still suffers from memory overhead which makes it an inefficient technique with the increasing scale of modern large-scale parallel and distributed systems. The group-based checkpointing idea introduced in [52, 93, 90] has widespread synchronous and asynchronous parallel applications. Fault tolerance support for MPI programs by incorporating fault tolerance semantics with the existing standard MPI library has been found in several research papers [47, 35, 87, 16].

2.6.3.2 Algorithm based Fault Tolerance

Algorithm-based fault tolerance (ABFT) was first proposed by Huang and Abraham for detecting and correcting permanent or transient hardware failures on systolic array [63]. The detection and correction of errors which occur during computation with multiple processors is accomplished with a low overhead after the completion of the computation. The ABFT algorithm takes checksum vector or matrix as an input operand and produces an encoded output matrix. The encoded output matrix helps to detect, locate and correct errors that can occur during a computation after the completion of the calculation. Researchers broadened ABFT research considering the above checksum concept with the same fail continue model [4, 6, 11, 70]. An algorithm based error detection mechanism to detect a faulty processor in the hypercube is proposed in [6]. Lanczos methods are used to optimize the total amount of work in error detection and correction levels in ABFT [11]. To identify and to locate transient errors in systolic array computation, a unified checksum method for LU decomposition, Gaussian elimination and the QR decomposition is proposed in [70], and also some error analysis helps to find out the effects of round-off errors on the checksum.

Research on ABFT extended for the fail-stop environment where the system will stop the execution of the failed component while the other remaining units continue their work [27, 28, 30, 38, 98, 42, 43, 92, 69, 39, 26, 14, 1, 2]. An encoded global checksum relationship is defined with the help of application data and is maintained throughout the operation in case of parallel linear algebra [28].

The ABFT method proposed by Huang and Abraham [63] cannot correct the effect of failures during execution; rather, it can detect, locate and recover from an error after the completion of the operation. ABFT needs to be familiarized with the recovery

of the error during execution and maintain the global checksum relationship throughout the computation, known as the "online" approach [29, 26]. Researchers enhanced the ABFT method of Huang and Abraham [63] by introducing the failure free environment just after the failure during the execution, instead of detecting, locating and recovering after the completion of the computation for linear algebra. Researchers also claimed that the computational cost is very low with an increasing number of processors with a constant problem size [14] for a single failure. To be competitive for high-performance computation in large systems, ABFT needs to preserve the " online" feature. Research shows that the Cannon algorithm and Fox algorithm for matrix multiplication do not conserve the checksum relationship during the computation. Only the outer product strategy of matrix multiplication preserves the "online" features for a single failure in the fail-stop model [26]. In his research paper, Chen has shown that certain iterative methods (especially parallel sparse matrix-vector multiplication) with parallel data partition satisfying certain conditions contain essential redundant information. With the help of this redundant information, the recovery of lost data can be possible, and as well the recovery cost is much less than the existing checkpoint with a single process failure [27].

ABFT for High-Performance Linpack Benchmark has advised that it can recover from the fail-stop without checkpoint, with the help of right look LU factorization algorithm and proves that it maintains checksum relationship throughout the computation [39, 38].

Considering the complexity and scaling of future exascale systems, an ABFT for HPC applications is proposed. Here, the application can continue its execution with redundant data in the presence of multiple failures through an accelerated recovery method which helps to reconstruct redundancy in the presence of multiple failures [30].

Data dependency and communication dependency are a general phenomenon in message passing parallel applications. Considering these features, an ABFT is developed for future exascale computing. Data of a failed processor can be recovered by keeping the redundant copy of dependent data and messages in other processors. This has been demonstrated with Newton's method, a well-known technique to solve a nonlinear system [69].

A hybrid of ABFT checksum and checkpoint: checksum for right factor and checkpoint algorithm for the left factor of dense matrix factorization like LU, Cholesky and QR has been proposed to survive single fail-stop failure. The research also claims false tolerance overhead decreases with an increasing amount of computing entity and problem size [42, 43].

An enhanced ABFT to existing ABFT scheme has been proposed for future exascale systems considering multiple failures in succession. It is proved theoretically that the proposed scheme incorporates little overhead as compared to the existing ABFT for exascale [98].

A new approach to distributing the redundant parity data (fault recovery data) of the original matrix to different processors to tolerate multiple simultaneous correlated failures in future exascale machines has been presented in [1, 2]. Moreover, methods to keep fault recovery data (parity data) and computation data together in one processor instead of keeping them separate have also been demonstrated here. Recently, [85] an ABFT technique for two-sided matrix factorization in the presence of soft errors, has been developed.

A resilient framework for local dependency dynamic programming in unreliable memory like DRAM is proposed in [19], where the research mainly concentrated with memory fault. The researcher further address the issue that unreliability of DRAM

memory causes bit-error which introduces corrupted data in the memory which ultimately changes the logical state of the memory bit. A combination of majority techniques and Karp-Rabin fingerprints were used to identify the bit errors and later on semi-resilient data from unreliable memory were used to prove the correctness of the DP table. Researchers further extended the idea by proposing a general framework with other types of dynamic programming, and their results theoretically showed that with a limited number of memory faults at any level of memory hierarchy, there was low overhead in runtime and a small number of cache misses [20].

ABFT and checkpointing have been combined to provide a generic hybrid ABFT for one-sided dense matrix factorization to survive multiple fail-stop failures, where, the upper right triangle of the matrix is protected with ABFT, and the lower left triangle is preserved with Q-parallel checkpoint [17]. A recent publication discussed ABFT schemes for two-sided matrix factorizations (Hessenberg, tridiagonalization, bidiagonalization) in the presence of soft-errors is presented [96]. An assessment of a unified approach of ABFT and checkpointing has been shown in [15, 13]. In recent works, a unified approach of ABFT and checkpointing has been employed for conventional iterative HPC applications to recover from failures where the computation of the application alternates between ABFT protected, and checkpoint protected phases. ABFT is used to protect the library calls and traditional periodic checkpointing is used when the application enters the library calls and also, if needed, between these calls [15, 13]. The research believes that fault tolerance at exascale needs the combination of checkpointing and algorithm-based recovery techniques, as is most appropriate for different phases within a single application.

2.6.3.3 Other Fault Tolerance and Recovery Methods

- Replication

  Replication is well-exercised fault tolerance technique. This technique replicates the data on different other systems. In the replication techniques, a request can be sent to one replica system in the midst of the other replica system. In this way, if a particular or more than one node fails to function, it will not cause the whole system to stop functioning. Though replication adds redundancy in a system, it can alleviate the scalability issue of CP/R in large-scale systems. A Replication approach to provide resilience in exascale systems is proposed with two replication techniques : (i) group replication: entire application instances are replicated and (ii) process replication: a single instance of each application process is replicated [24]. State machine replication has been proposed as an alternative to CP/R for upcoming exascale systems [50].

- Task Resubmission

  It is one of the widely used fault tolerance technique in the current large-scale system, especially in cloud computing for MapReduce algorithms. Here, during run-time, a manager splits the work into tasks, and the tasks are executed by worker nodes. Whenever a failed task is detected, it is resubmitted either to the same or a different resource at runtime. Hadoop an Apache open source framework based on Java, which is commonly used for MapReduce [94]. Hadoop has two main components: (i) MapReduce and (ii) HDFS (Hadoop Distributed File System). Both of these components provide fault tolerance. HDFS provides fault tolerance through replication of data and MapReduce provides fault tolerance by resubmitting the task to other nodes in case of task failures and by rescheduling the tasks to other nodes in case of the node failures [84]. However, this method can be vulnerable to the failure of the manager node responsible

for scheduling. Moreover, failure of a single machine incorporates 50% increase in total execution time in case of a Hadoop job. A a new model, named "Spark" has been introduced to address the scalability issue of Hadoop in case of iterative machine learning algorithms [99].

Chapter 3

ABFT for Embarrassingly Parallel Algorithms

In this chapter, we present an efficient algorithm based fault tolerance strategy for a group of applications that exhibit similar structural and behavioral features. Parallel search algorithms are viewed as a collection of independent tasks which can execute concurrently and which barely communicate with each other during the lifetime of the algorithm. Depth-first search (DFS), depth-first branch and bound (DFBB), iterative deepening A* (IDA*), and best first search (BFS) are some of the standard search algorithms. Parallelization strategies of these search algorithms have been well researched [56]. These algorithms are widely used to solve discrete optimization problems (DOP) in parallel systems. Fundamentally, DOP problems belong to the class of NP-hard problem, which are computation-intensive applications. Parallel search algorithms are major solution strategy to give a real-time solution of many DOP problems with acceptable performance. In that regard, here, we develop a fault tolerance scheme for parallel search algorithms. We demonstrate our idea with parallel iterative deepening A* (PIDA*) which is a variant of parallel DFS to find the best solution. This similar strategy is also applicable to other parallel depth-first

search techniques. We show that our fault tolerance method has notable performance compare to the existing CP/R solutions.

3.1 Motivation

In section 2.6.3, we present two major methods of fault tolerance in HPC systems. The most common technique is checkpointing. Although conceptually checkpoint is very simple, it is not scalable to large systems as it introduces enormous overheads. Checkpoint and recovery cost imposed by checkpoint/restart (CP/R) is a crucial performance issue for HPC applications. In comparison, Algorithm-based fault tolerance (ABFT) is a promising alternative with low overheads, but it suffers from the inadequacy of universal applicability. Till today, ABFT implementation is confined to a particular class of parallel applications which mainly involve matrix-based computations. We widen the span of ABFT to other types of parallel applications rather than those that engage matrix based computation. Regarding applicability, here we demonstrate ABFT with parallel search algorithms, which belong to the class of embarrassingly parallel algorithms. The proposed solution is a general fault tolerance strategy for a group of parallel algorithms that exhibit similar algorithmic characteristics.

3.2 Algorithmic Characteristics of PIDA*

PIDA* is depth-first search (DFS) algorithm used to solve discrete optimization problems (DOP) where the search space unfolds as a tree at runtime[77]. The unexplored nodes in the tree-based algorithm can be represented as a stack (we call it a DFS stack to distinguish it from a process' system-level stack), and the parallel algorithm

involves distributing the local DFS stacks among processes at run-time and then dynamically balance loads among processes based on a work-request protocol, i.e., a process that runs out of work requests work from other processes. Figure 3.1 illustrates the snapshot of the search tree and the local DFS stack of a process while solving an 8-puzzle problem.

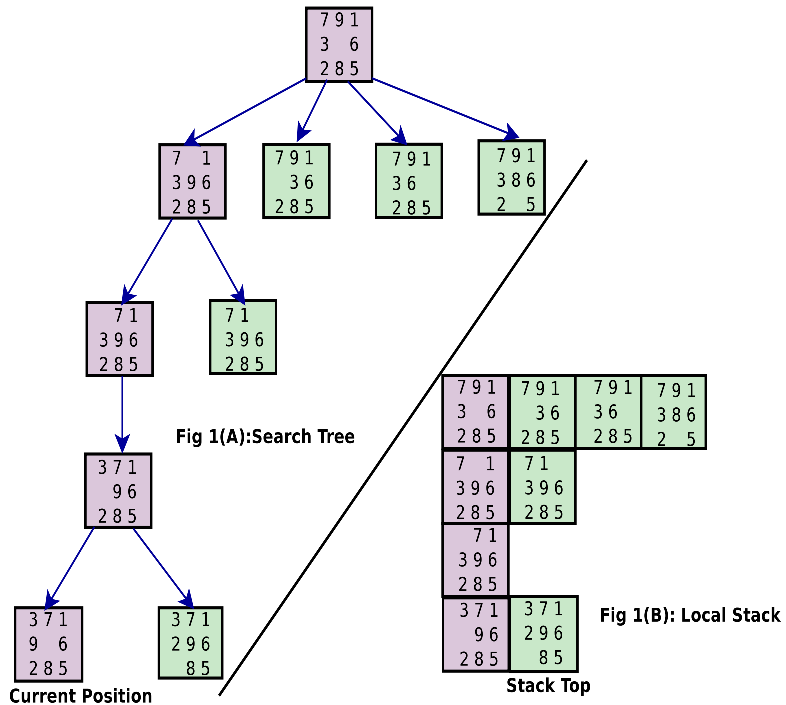


Figure 3.1: A sample search tree and local DFS stack of a process P_i at time t_i for the 8-puzzle problem

Algorithm 3.1 presents the parallel iterative deepening A* algorithm. It is evident from algorithm 3.1 that (i) B_DFS and (ii) GETWORK are the basic functions in PIDA*. Following is a description of these two functions:

- **B_DFS** does a cost bound depth first search (DFS) starting at the top node of the DFS stack. $VISIT$ expands an unexplored child "nextson" at the top of a process's local DFS stack and signals termination to other processes if a goal node is found; otherwise, $ADDCHILDREN$ adds the children of nextson to

Algorithm 3.1 PIDA* based on [77]

```
1: function PIDA*()                                        ▷ Parallel IDA* on Processor P_i
2:     while (not solutionfound) do
3:         if (stack[i]=empty) then                                    ▷ There is no work
4:             GETWORK();
5:             continue;
6:         end if
7:         if ( stack[i] ≠ empty) then
8:             B_DFS(stack[i]);
9:         end if
10:        if (TERMINATION_TEST()=TRUE) then
11:            cb = MIN{nextcb[k], 1 ≤ k ≤ N};      ▷ Cost bound, cb, for next iteration;
12:            Initialize stack_depth, cb and nextcb for the next iteration;
13:        end if
14:    end while
15: end function
16: function B_DFS(Stack[i])                                            ▷ Bounded DFS
17:    while ((not solutionfound) and (depth > 0)) do
18:        if (stack[i]=empty) then
19:            depth[i] = depth[i] − 1;                                     ▷ Backtrack
20:            Continue;
21:        end if
22:        VISIT(Stack[i]);                 ▷ Visit best child "nextson" from top of stack
23:        if (not solutionfound) then
24:            ADDCHILDREN(nextson);
25:            depth[i] = depth[i] + 1;                     ▷ Advance to the next depth
26:        end if
27:        nextcb[i] = MIN(nextcb[i], nextson.cost);
28:    end while
29: end function
```

the top of the DFS stack and the computation repeats.

- **GETWORK** is for balancing loads among processes. It contains two functions: $WORK_REQUEST$ and $WORK_SEND$. Using $WORK_REQUEST$, whenever a process runs out of work, it requests work from another process; the other process can be selected at random or in a round-robin fashion based on the application's criterion. With $WORK_SEND$, a requested process sends part of its work (local DFS stack) to a requesting process.

One important characteristic of PIDA* and other depth first search algorithms is that each $VISIT$ is independent of visits to other nodes in the search tree and can be performed in parallel without any inter-dependency. This characteristic defines the *critical data*, i.e., the minimal application data to be saved to withstand n-process failure for some n. It also defines the fault-tolerance strategy for this class of applications based on the following observations: (i) fault of one process will not hinder other processes from proceeding with their searches because each search is independent; (ii) if the faulty process' search would have led to a solution then, upon its recovery from a previous consistent state, the solution will be eventually reached; hence only the performance and not the correctness of the search is affected by the failure; (iii) if the faulty process' search would not have led to a solution, then the failure would have little, if any, impact on the overall search outcome and performance.

3.3 Fault Tolerance for Parallel Iterative Deepening A* (PIDA*) Search Algorithm

Here, we present a generic fault tolerance strategy for parallel search algorithms. We illustrate it with parallel iterative deepening A* (PIDA*) algorithm, which is a variant of parallel DFS.

3.3.1 Critical data

We define *critical data* as the minimal application data required to be saved (securely) so that a failed process can be fully recovered from a most recent *consistent state* using this and any previously saved data. We discuss *consistent state* in subsection 3.3.4. Obviously, the size of critical data and the fault-tolerance strategy determine the overhead in ABFT.

As an example, in sequential DFS the unexplored children of the already expanded nodes, together with the currently visited node, constitute the critical data. If this critical data is (securely) saved at a snapshot i then the critical data of the process at a subsequent snapshot $i+1$ will constitute any additional data that is to be saved, i.e., critical data is progressive. These unexplored children are usually represented as a DFS stack, with the top of the stack containing the next child to be visited (figure 3.1). So the critical data at first snapshot constitute the current DFS stack together with the currently visited node (if any), i.e., the DFS stack right before the visit of the current node. In the case of parallel DFS, the DFS stack is distributed among processes, and each process independently operates on its local DFS stack. So the critical data at snapshot 0 to withstand a process P_i's failure is the process' local stack right prior to its visit to the current node (if any).

Figure 3.1 and fig.3.2 show the stack contents of a process at two different snapshots. Figure 3.3 shows the critical data as the difference in the stack contents between the two snapshots. These are further elaborated in the next subsection.

Lemma 3.1. *The size of critical data in DFS is (i) linear in depth and (ii) non-monotonic in time.*

Proof The critical data in DFS constitutes the unexplored nodes in the current path of the dynamically generated DFS tree. This data is usually represented as a DFS stack, where each level of the stack contains the parent and its unexplored children

Figure 3.2: Search tree and local stack of P_i at time t_j

(figure 3.1) on the current path being explored. The already explored paths, which did not lead to a solution, are pruned. The average number of children, also known as the *branching factor,* b is a constant for a particular application. If d is the current depth, then the size of the stack is O(db) which is O(d) as b is constant, i.e., it is linear in depth. Moreover, a path can change due to backtracking and so is the depth, which either increases on progress or decreases on backtracking, and hence the size of the stack is non-monotonic in time.

Lemma 3.2. *The critical data of a process in a parallel DFS at the instant of visiting a node v is a subset of the critical data in sequential DFS at the instant of visiting the same node.*

Figure 3.3: Critical data ($B \setminus A$ of Lemma 3) of P_i at time t_j (refer to figure 3.1 and figure 3.2)

Proof In a parallel DFS, there is no guarantee that a node v visited by the sequential search will ever be visited in the parallel search (because the solution might have already been reached by another process in another path), and vice verse. The following applies if the node is visited by both sequential and parallel search. For each node in the DFS tree, there is a unique path from the root based on the basic assumption that visit of a node is deterministic, i.e., it will always produce the same result deterministically. The DFS stack contains all the explored nodes in the path (i.e., the parents) and their unexplored children. So, in a parallel search, for the process P which has visited the node v, part of the path can be with another process from which P borrowed work, and part(s) of the path can be with another process(es) which borrowed work from P directly or indirectly. So the local stack contents of P, i.e., its critical data, is a subset of the stack contents of the sequential search at the instant of visiting v.

3.3.2 Fault Tolerant PIDA*

We present the fault tolerant PIDA* (FTPIDA*), which embeds fault-tolerance support to PIDA* without altering the original algorithm. As discussed in the previous subsection, the crucial information that is needed to re-execute a process from a previous consistent state is its critical data. The basic idea is to replicate the processes' critical data distributively at periodic intervals (e.g., determined by the Mean Time To Failures (MTTF) for the particular system) so that certain number of simultaneous failures can be tolerated, and the failed process(es) can be restarted from a previous consistent state(s) using the saved critical data. We discuss about consistent state in subsection 3.3.4.

Following are the basic assumptions in the design of FTPIDA*:

- There is a one-to-one mapping between the set of processes and the set of processors. A process is comprised of multiple threads running PIDA*, FTPIDA*, and the recovery kernel respectively
- crash-stop (aka fail-stop) failure of the processor is considered. It should be noted that process failure, with the underlying processor still healthy, is a softer failure situation and a subset of the following techniques can be applied for failure recovery
- failures are independent, i.e., a failure in one processor does not induce failure in another processor
- a processor failure does not cause the application to abort automatically
- no failure occurs during recovery
- processors that share the same system resources will be the least choices to keep the backup of each other

- communication is reliable

Empirical data has shown that a process could face at most one failure in its lifetime [51]. In that regard, even though two simultaneous failures are rare, we consider this possibility. However, the algorithm is flexible enough and can easily be extended to keep more redundant copies to sustain more simultaneous failures.

Algorithm 3.2 FTPIDA*

```
 1: function INITIALIZE()
 2:     Arrange all N processes in a logical ring.
 3:     Each process executes PIDA* (algorithm 1)
 4:     set A ← ∅                                              ▷ Initialize set A to empty
 5:     Execute FTPIDA* at each c interval or after sending work to   another process      ▷
    FTPIDA* is event driven      ▷ Let us call FTPIDA* executed on process P_i as FTPIDA*(P_i)
 6: end function
 7: function FTPIDA*(P_i)
 8:     Lock stack                              ▷ Mutex lock to read DFS stack consistently
 9:     set B ← {contents_of_currentstack} ∪ {visited_node}                 ▷ Critical data
10:     Unlock stack
11:     Left_neighbour = P_{(i+N-1)%N}
12:     Right_neighbour = P_{(i+1)%N}
13:     C ← B \ A                               ▷ Content of Fault recovery message (FRM)
14:     send C to both Left_neighbour and Right_neighbour
15:     A = C
16:     if (work_received) then                         ▷ receive work from a sender process
17:         set working stack
18:         send FRM to Left_neighbour and Right_neighbour                  ▷ Backup point
19:         send workACK to the sender process after backupACK received
20:     end if
21:     if (workACK_received) then            ▷ receive acknowledgement from a receiver process
22:         commit DFS stack update                              ▷ Stack update is confirmed
23:         send update FRM to Left_neighbour and Right_neighbour           ▷ Backup point
24:     end if
25:     if (backupACK_received) then                ▷ receive acknowledgement from neighbours
26:         commit backup point      ▷ Update FRI of the process for any future recovery from this
    point
27:     end if
28:     if (FRM_received) then      ▷ receive FRM, which contains critical data, from a neighbour
    process
29:         update Fault Recovery Information (FRI)
30:         send backupACK to the sender process
31:     end if
32: end function
```

Algorithm 3.2 represents the fault tolerance algorithm FTPIDA*. The following describes the working principles of the algorithm.

- At the beginning, all the N process involved in PIDA* (and FTPIDA*) initialize a logical ring among themselves (line 2 in Algorithm 3.2), using either a centralized or a distributed consensus scheme, by defining process $P_{(i+1)\%N}$ as the right_neighbour and process $P_{(N+i-1)\%N}$ as the left_neighbour for an arbitrary process P_i.

- Each process periodically exchanges Fault Recovery Message (FRM), which contains incremental critical data, with its neighbours at every periodic interval c, where $c < MTTF$, or after each work distribution (refer to lines 9-15 and 18-24 in Algorithm 3.2, and figure 3.3). The status of a process can be either "active" or "passive." A process is active when it has left_neighbour and right_neighbour, and it is executing. "Passive" means the process does not have its stack for execution instead it just keeps backup data for other processes, but it is ready to become "active."

Following is a formal description of the contents of FRI and FRM:

- FRI (Fault Recovery Information): $FRI = (S, Left_{ID}, Left_S, Right_{ID}, Right_S, LLeft_{ID}, RRight_{ID})$

- FRM (Fault Recovery Message): $FRM = (S, Left_{ID}, Right_{ID})$

where, S stands for the local DFS stack of a process; $Left_{ID}$ and $Left_S$ are respectively the id and the critical data of the left_neighbour; $Right_{ID}$ and $Right_S$ are respectively the id and the critical data of the right_neighbour; $LLeft_{ID}$ is the left_neighbour id of its left_neighbour and $RRight_{ID}$ is the right_neighbour id of its right_neighbour. Note that $FRM \subset FRI$.

For failure detection, we consider the existing distributed fault detection technique in a parallel and distributed system that uses the push model by exchanging "heartbeat" messages among the neighboring processes [48]. Note that a false positive does

not alter the correctness of the algorithm; it only increases the unnecessary overhead of replicating the work of a healthy process which is falsely identified as failed.

Lemma 3.3. *The size of critical data in FTPIDA* at each interval is $O(f)$, where f is the corresponding cost bound.*

Proof IDA* uses the cost function $f(n) = g(n) + h(n)$ where $g(n)$ is the sum of the edge costs from the initial node, i.e., root, to the current node n, and $h(n)$ is the heuristic estimate for reaching the goal node from the current node n [77]. Assuming that each edge cost is 1, then $g(n) = d(n)$, the depth of the node n in the DFS tree. This gives $d(n) = f(n) - h(n)$. Consequently, for a given cost bound f, the maximum attainable depth $d_{Max} = f$, when $h = 0$.

According to Lemma 1, the maximum size of critical data at depth d_{Max} is $O(d_{Max})$ which is $O(f)$. Now, according to algorithm 2, the following two cases arise: (i) at first interval, process P sends its entire critical data, i.e., contents of the local stack and currently visited node, to the two neighbours. This is set B in line 10, Algorithm 2. Accordingly, size of this message is $O(f)$ where f is the current cost bound. (ii) if at interval i process P sends set A to its adjacent neighbors and if at interval $(i + 1)$ B is its critical data set, then process P sends $C = B \setminus A$ to its adjacent processes at interval (i+1) (line 14, Algorithm 2). Now, size of the set C is $|B \setminus A| = |B| - |B \cap A|$ and $|B \setminus A|_{Max} = |B|$ which is $O(f)$. So, in either case, the critical data size at each interval is $O(f)$ (Figures 3.1, 3.2 and 3.3).

3.3.3 Failure Recovery

When a process (i.e., the recovery kernel thread of the process) detects its neighbor's failure, the detecting process is responsible for restarting the failed process and bringing it back to the previous consistent state. During recovery, the fault recovery kernel executes one of the following steps:

- If a free processor is available then it sends the fault recovery information (FRI) of the failed process to the newly created process on the available processor and rebuilds the distribution ring by making this new process as its left_neighbour /right_neighbour

- If no free processor is available then it adds the work (DFS stack) of the failed process to its own and rebuilds the logical ring by defining the left_neighbour /right_neighbour of the failed process' left_neighbour /right_neighbour as its own left_neighbour /right_neighbour, modifies its own FRI, and sends FRM to its new left_neighbour /right_neighbour (lines 9-14, Algorithm 3.3)

Algorithm 3.3 represents the fault recovery algorithm for PIDA*.

Algorithm 3.3 FTPIDA* Recovery module

```
1: if (can be recovered) then                                   ▷ Refer to Lemma 4 below
2:     RECOVER()                                                ▷ Recover() is defined below
3: else
4:     INITIALIZE()                                             ▷ Restart Algorithm 2 from beginning
5: end if
6: function RECOVER()
7:     if (free processor is available) then
8:         Send FRI of the failed process to the free processor
9:         Reconstruct the logical ring
10:        Start the failed process from the last committed backup point     ▷ Recover using its
   critical data
11:    else
12:        Merge the critical data of the failed process with its DFS stack
13:        Reconstruct the logical ring
14:        send FRM to Left_neighbour and Right_neighbour                    ▷ Backup point
15:    end if
16: end function
```

Lemma 3.4. *FTPIDA* can tolerate simultaneous failures of at the most* $\lfloor \frac{N}{3} \rfloor \times 2 + \lfloor \frac{N \bmod 3}{2} \rfloor$ *processes in a* N *process ring.*

Proof This follows from the fact that a process' critical data is replicated with two of its neighbours in the logical ring. In the worst case scenario, a process P's both neighbours have failed, and so all its saved critical data is lost. Whenever either of

P's two neighbours is recovered by the recovery kernel, the recovered process restarts from a previous consistent state based on the fault recovery information (FRI) saved in process P (refer to Algorithm 3), which also contains the critical data of P (note that $FRM \subset FRI$). The system is unrecoverable if and only if P also fails, i.e., when 3 consecutive processes in the logical ring fail. Consequently, no consecutive 3 or more processes in the ring can fail simultaneously. Thus, we can divide the N processes into $G = \lfloor \frac{N}{3} \rfloor$ distinct consecutive groups of 3 processes each. In each of these G groups, maximum 2 processes can fail simultaneously; accordingly at the most $G \times 2$ processes can fail simultaneously out of the G groups. Of the remaining $Rem = (N \bmod 3)$ processes, if $Rem = 2$ then at the most 1 of these 2 processes can fail simultaneously with the others in the G groups so that there are no 3 consecutive failures in the logical ring. This gives the second term $\lfloor \frac{Rem}{2} \rfloor$ of the above formula.

3.3.4 Consistent state

Abstractly, each process in PIDA* performs the following steps in the main thread of computation:

- Step 1: Lock stack {Mutex lock for consistency with FTPIDA* thread}
- Step 2: Pick up next node from stack. If node is goal node then go to step 7 else go to step 3
- Step 3: Expand node
- Step 4: Add children to stack (ordered by heuristic estimates)
- Step 5: Unlock stack
- Step 6: Repeat from Step 1
- Step 7: unlock stack, notify others, and end

In FTPIDA*, contents of the local DFS stack prior to each Step 1 constitutes the critical data of a process. This is backed up periodically with the two neighbours in every c interval, where $c < MTTF$, and also during work distribution to a requesting process (Algorithm 3.2). Let us call it a *backup point* at the instant the critical data of a process is backed up with the neighbours.

Upon failure, a process can be restarted from a previous Step 1, as long as the DFS stack contents can be restored to the backup point just prior to the restarted Step 1. From the perspective of the FTPIDA*, we call the process state at a backup point provided the backup is committed (i.e., acknowledgment is received from both neighbours, and local Fault Recovery Information (FRI) is updated) as a *local consistent state*. In other words, a failed process is recovered by restarting it from the most recent local consistent state using the backed up critical data.

Two types of application messages are involved: (i) in PIDA*, the sending and receiving of works (i.e. DFS stack contents) among processes (via GETWORK() in Algorithm 3.1) and (ii) in FTPIDA*, periodic exchange of fault recovery message (FRM) among the adjacent neighbours in the logical ring (Algorithm 3.2). A failed process can be independently recovered from its most recent local consistent state as long as any message dependency is resolved without affecting the other processes. This is elaborated in the following proof.

Lemma 3.5. *FTPIDA* is correct.*

Proof (a) Any message dependency of a process recovered from its most recent local consistent state is resolved without affecting the other processes. This is achieved as follows: (i) any work received from another process is first backed up (committed) and then an acknowledgment is sent to the sender (lines 18-20, Algorithm 3.2). So this committed backup point becomes the new local consistent state; (ii) if the process fails before receiving the work, then the message containing work can be discarded

without affecting the correctness because the sender does not update its local DFS stack until receiving an acknowledgement from the receiver (lines 23-24, Algorithm 3.2); (iii) if the process receives an FRM from neighbour, it first updates its local FRI and then only sends acknowledgement to neighbour. Note that neighbour does not commit its backup point until receiving acknowledgements from both the neighbours (lines 27, 30-31 in Algorithm 3.2); (iv) if the process fails before receiving FRM from neighbour, then upon recovery that FRM is resent by the recovery kernel (which is a subset of FRI. Refer to line 9 of Algorithm 3.3). **(b)** Execution of a process between any two local consistent states is deterministic based on the following: (i) by assumption, visit of a node is deterministic, i.e., each visit of the node produces the same result deterministically; (ii) the only nondeterminism arises in execution is when the DFS stack changes due to random work request and subsequent work transfer. This is handled by the sender by taking a backup with neighbours after the receipt of work is confirmed, which becomes a new local consistent state of the sender (lines 23-24, Algorithm 3.2). Based on (i) and (ii), execution of a process between any two local consistent states is deterministic. As consequences of **(a)** and **(b)**, a failed process can always be restarted from its most recent local consistent state independent of other processes, and the restarted process will ultimately reach the point where it failed and continue execution as if the failure has never occurred. So FTPIDA* is correct, i.e., it will produce the same result as a failure-free execution.

3.3.5 Example of FTPIDA* Protocol

Here we illustrate an example to describe the working principle of FTPIDA* protocol. We assume that the system is a homogeneous cluster of 6 processors, i.e.,$\{P_1,P_2,P_3,P_4,$ $P_5,P_6\}$. Also, each processor keeps two redundant copies of fault recovery data with two of its neighbours. Table 3.1 and figure 3.4 display the FRI and the distribution

ring with every following scenario respectively.

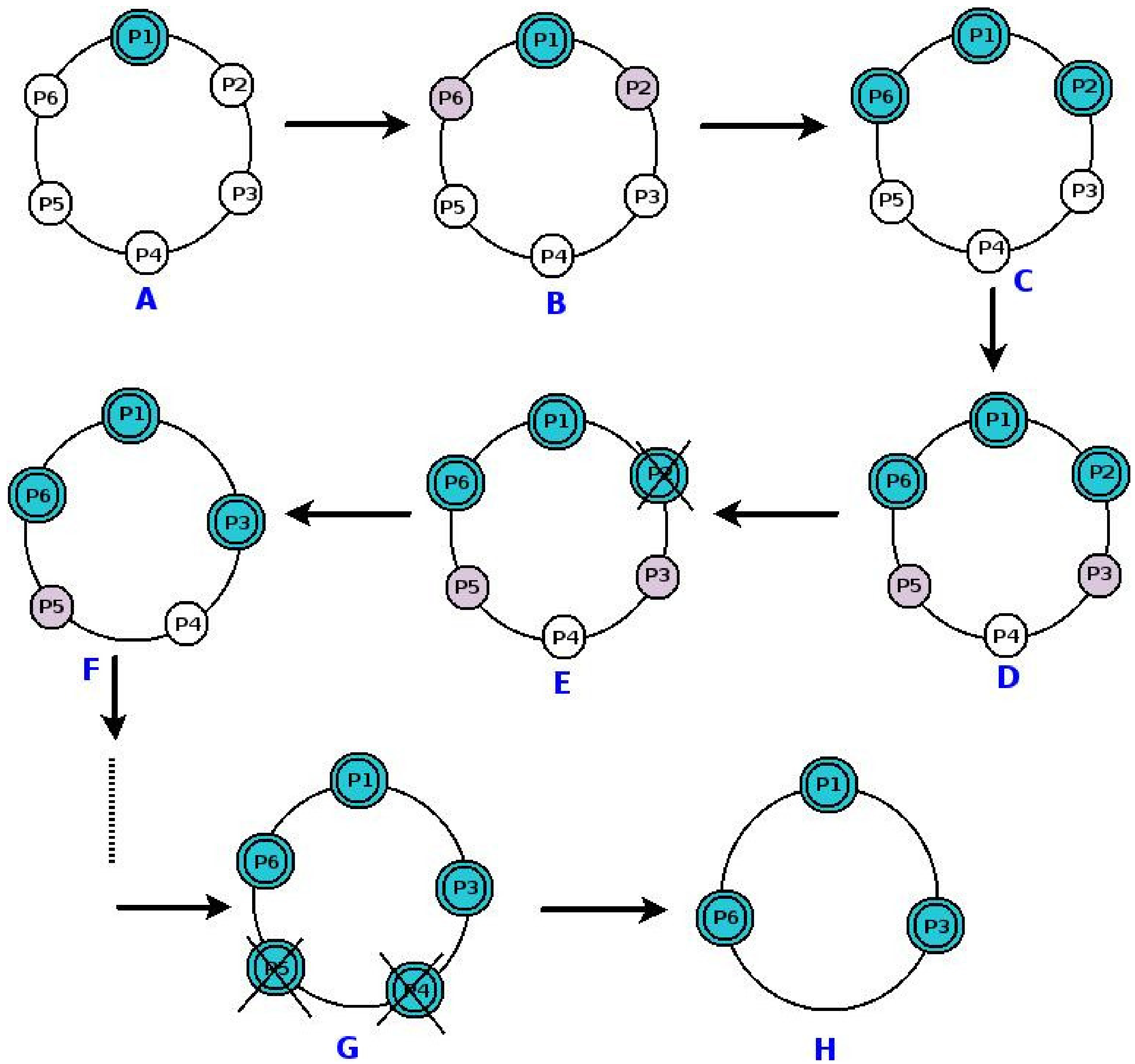


Figure 3.4: Example of FTPIDA* algorithm

- **A**: All the processes initialize a logical distribution ring among themselves. Initially, P_1 has the whole search space. The current stack of P_1 is S. Status of P_1 is "active".
- **B**: Before MTTF, P_1 exchanges FRM with P_6 and P_2, who are the left_neighbour

and right_neighbour of P_1 respectively. The status of P_6 and P_2 become "passive".

- **C**: P_1 receives work request from P_2 and P_6 and it splits it working stack and distributes work accordingly. The status of P_2 and P_6 is now "active". P_1 updates the FRM and exchanges with P_2 and P_6.

- **D**: Before MTTF, P_2 and P_6 exchange their FRM with their left_neighbour and right_neighbour respectively.

- **E**: Failure scenario: P_1 detects its right_neighbour P_2 has failed.

- **F**: P_1 finds the right_neighbour P_3 of the failed P_2 is free and in "passive" state. P_1 define P_3 as its right_neighbour and assign the work of P_2 by sending FRI.

- **G**: P_3 detects its right_neighbour P_4 and P_6 detects it's left_neighbour P_5 have failed.

- **H**: No free processes are available at that moment, so both the processes add the work of their failed neighbour processes to itself respectively.

FRI of all the processes are updated accordingly, shown in table 3.1.

Table 3.1: Fault Recovery Information (FRI) of Processes During Execution of FT-PIDA*

Scenario	FRI Tuple	FRI Tuple	FRI Tuple
(A)	P_1: $(S_1, [_ : _], [_ : _], _, _)$		
(B)	P_1: $(S_1, [P_6 : _], [P_2 : _], _, _)$	P_6 : $(_, [_ : _], [P_1 : S_1], _, P_2)$	P_2: $(_, [P_1 : S_1], [_ : _], P_6, _)$
(C)	P_1: $(S_1, [P_6 : S_6], [P_2 : S_2], _, _)$	P_6: $(S_6, [_ : _], [P_1 : S_1], _, P_2)$	P_2: $(S_2, [P_1 : S_1], [_ : _], P_6, _)$
(D)	P_1: $(S_1, [P_6 : S_6], [P_2 : S_2], P_5, P_3)$	P_6: $(S_6, [P_5 : _], [P_1 : S_1], _, P_2)$	P_2: $(S_2, [P_1 : S_1], [P_3 : _], P_6, _)$
	P_5: $(_, [_ : _], [P_6 : S_6], _, P_1)$	P_3: $(_, [P_2 : S_2], [_ : _], P_1, _)$	
(E)	P_1: $(S_1, [P_6 : S_6], [P_2 : S_2], P_5, P_3)$	P_6: $(S_6, [P_5 : _], [P_1 : S_1], _, P_2)$	
	P_5: $(_, [_ : _], [P_6 : S_6], _, P_1)$	P_3: $(_, [P_2 : S_2], [_ : _], P_1, _)$	
(F)	P_1: $(S_1, [P_6 : S_6], [P_3 : S_3], P_5, _)$	P_6: $(S_6, [P_5 : _], [P_1 : S_1], _, P_3)$	P_3: $(S_3, [P_1 : S_1], [_ : _], P_6, _)$
	P_5: $(_, [_ : _], [P_6 : S_6], _, P_1)$		
(G)	P_1: $(S_1, [P_6 : S_6], [P_3 : S_3], P_5, P_4)$	P_6: $(S_6, [P_5 : S_5], [P_1 : S_1], P_4, P_3)$	P_3: $(S_3, [P_1 : S_1], [P_4 : S_4], P_6, P_5)$
(H)	P_1: $(S_1, [P_6 : S_6], [P_3 : S_3], P_3, P_6)$	P_6: $(S_6, [P_3 : S_3], [P_1 : S_1], P_1, P_3)$	P_3: $(S_3, [P_1 : S_1], [P_6 : S_6], P_6, P_1)$

3.4 Performance analysis

We present an approximate theoretical analysis of FTPIDA*. We borrow some of the arguments in the analysis from [25, 64]. We compare the completion times of a PIDA* application running the traditional blocking coordinated checkpoint/restart (CP/R) protocol versus FTPIDA*. In the following analysis, we assume that there are maximum N number of parallel processes executing at a particular time.

3.4.1 Timing Overhead of FTPIDA*

Let the mean time to failure (MTTF) be m time units. Also, assume that the coordination occurs every c time units and d is the checkpoint time of a process. Let the total execution time of the application without any fault tolerance support be T_o.

In the case of blocking CP/R, the coordination time is proportional to the number of explicit coordination messages, which is in turn proportional to the number of coordinated processes. Thus the maximum coordination time $= \alpha N$, where α is a constant. The worst case execution time of the application running the protocol without encountering any faults is $T_c = T_o + (T_o/c)(\alpha N + d)$. If there occur n failures altogether, then the worst case execution time in the presence of faults becomes $T_{cf} = T_c + n(c + \beta N)$. Here βN is the overhead associated with restarting all the N processes, which is the requirement in CP/R, and β is a constant. By noting that $n = T_{cf}/m$, after substitutions and simple algebraic manipulations we get $T_{cf} = T_o \frac{1+(\alpha N+d)/c}{1-(c+\beta N)/m}$.

In the case of FTPIDA*, for the convenience of comparison with CP/R, let assume that the critical data is saved with the two neighbours in every c time units. The worst-case execution time of FTPIDA* without encountering any failure is $T_p = T_o + (T_o/c)\gamma$, where γ is the overhead to construct the fault recovery message (FRM) from local DFS stack and send the message; details are in Algorithm 3.2. According to Lemma 3, γ is $O(f)$ where f is the corresponding cost bound, assuming that

point-to-point communication cost is proportional to message size. If n be the total number of failures altogether, each of which is a single process failure, then there are two extreme possibilities on the execution time of the application in the presence of failures:

1. The best case scenario is when each of the faulty processes' DFS stack contents does not contribute to a goal node. In that case, unlike CP/R where all processes irrespective of healthy or not have to recover together, the failure of the process has no effect on the overall execution time of the application, and hence total execution time with failures is $T_{pf} \approx T_p = T_o + (T_o/c)\gamma$. It can be noted here that γ can be ignored considering that the overhead to construct FRM from the local stack is insignificant, and message send/receive are asynchronous, i.e., sender and receiver are not blocked. Hence we get: $T_{pf} \approx T_p \approx T_o$.

2. The worst case scenario occurs when each of the faulty processes' DFS stack contents contributes to the goal node in that specific run. In such case the execution time in the presence of n failures is $T_{pf} = T_p + n(c + \delta)$, where δ is the overhead of restarting the failed process. By noting that $n = T_{pf}/m$, after substitutions and simple algebraic manipulations, we get $T_{pf} = T_o \frac{1+\gamma/c}{1-(c+\delta)/m}$. Similar to the previous case, γ can be ignored, and we get: $T_{pf} \approx T_o \frac{1}{1-(c+\delta)/m}$. Apparently, when c gets much smaller than m, the denominator gets much closer to 1 and hence T_{pf} approaches T_o.

The following is a comparison of performance between FTPIDA* and blocking CP/R: in the case of blocking CP/R, as MTTF $m \to (c+\beta N)$ then $T_{cf} \to \infty$. In comparison, in FTPIDA*, as $m \to (c+\delta)$ then $T_{pf} \to \infty$, which may only happen under the worst case scenario of FTPIDA* when there occur n single process failures, each of which contributes to the goal node. Based on empirical data [7], the restarting overhead β in the case of CP/R can be huge; in fact, this overhead is multiplied to restart all

the N processes (i.e., βN). On the contrary, in FTPIDA*, the restarting overhead δ for the faulty process is proportional to critical data size of that process, which is O(f) (Lemma 3), and is a fraction of βN considering that β might involve the entire memory snapshot of a process. Moreover, in the best case scenario, fault has little impact on the total execution time of FTPIDA*. So, in conclusion, FTPIDA* can support much smaller m and hence can tolerant more frequent failures.

3.5 Numerical results

In this section, we evaluate our proposed fault tolerance FTPIDA* and compare its performance with the CP/R technique. All the experiments were performed on the supercomputer Guillimin from McGill University managed by Calcul Québec and Compute Canada[1]. Guillimin has 1200 nodes, 7200 cores and connected by 4TB/s InfiniBand. All nodes are running CentOS 6.3 operating systems. Both FTPIDA* and CP/R implement with OpenMpi 1.8.6.

For our experiment, we implement FTPIDA* and CP/R and use 24 puzzle problem as the case to compare the performance of both the fault tolerance methods. Here, we acknowledge the simultaneous failure of at most 50% of the processes for each test case. For the multiple process failures to occur, we randomly choose the process to be failed with the condition that at most two consecutive neighbours can fail simultaneously. In our implementation, we consider $MTTF = 400$ seconds, and with CP/R, the checkpoint is taken just before every 400 second.

Figure 3.5 shows the overhead of fault tolerance in a failure-free execution for both FTPIDA* and CP/R with the existing application. The result indicates that backup

[1]The operation of this supercomputer is funded by the Canada Foundation for Innovation (CFI), NanoQuébec, RMGA and the Fonds de recherche du Québec - Nature et technologies (FRQ-NT)

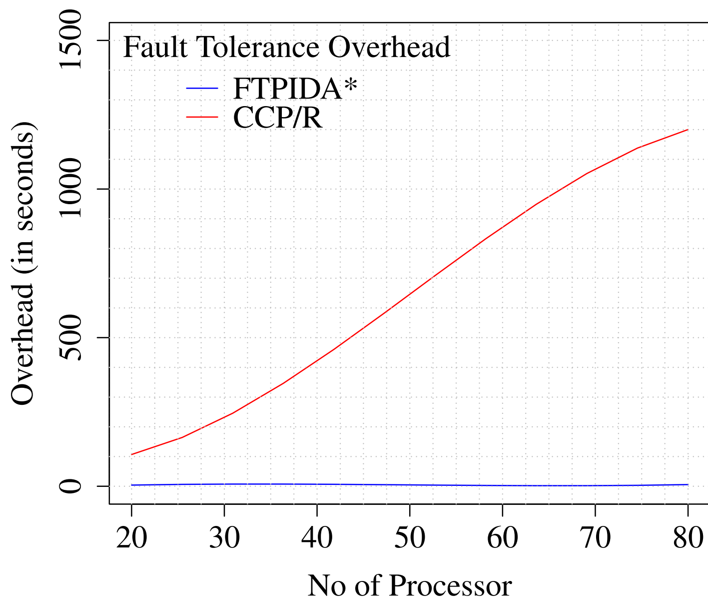


Figure 3.5: Fault Tolerance Overhead (without fault): FTPIDA* vs. CP/R

time for FTPIDA* is subtle and apparently constant with increasing number of processes. On the contrary, for CP/R, checkpoint overhead increases almost linearly at a much higher rate with the increasing scale as compared to our scheme. The reason behind is more checkpoints have to perform with more processes which include more coordination and checkpoint message time with the execution time.

Figure 3.6 represents the comparison of fault recovery overhead between FTPIDA* and CP/R with the different number of the processes. CP/R always spends more time to recover the system from faults, and this recovery time increases with increasing scale. On the other hand, the recovery time of FTPIDA* is almost constant and always smaller than that of CP/R, irrespective of the number of the processes. This is because CP/R resumes all the processes to a globally consistent state even with a single process failure. Whereas, FTPIDA* only resumes the execution of the failed process to its last consistent state (independently) with the critical data from one of its two neighbors. FTPIDA* takes a nearly constant time to recover from one or more processes failure as all the recovery happens independently in parallel. The

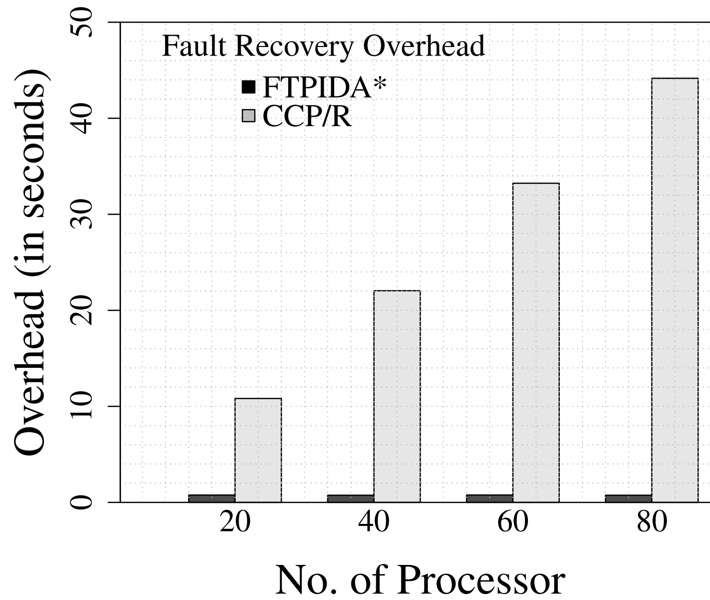


Figure 3.6: Failure Recovery Overhead: FTPIDA* vs. CP/R

result shows that our proposed method has a distinct advantage over CP/R. It can be seen that FTPIDA* method reduces the overhead of fault recovery by 90% at most, as compared to CP/R.

Figure 3.7 shows the performance improvement of FTPIDA* over CP/R with multiple process failures. It is found that the performance of FTPIDA* over CP/R increases linearly with the increasing number of processes. The results show that in a failure scenario, our proposed fault tolerant method has performed well and decreased (improve performance) the overhead nearly 30% to 67% for the number of processes from 20 to 80. The reason behind, the more the number of processes incorporates, the more the time required for CP/R for checkpointing and recovery with more failures.

Table 3.2 shows the total backup data size produced by FTPIDA* and CP/R . The sizes are given in megabytes and column labeled "Reduction" represents the total amount by which the backup data size is lessened by FTPIDA* as compared to CP/R. In all the cases, the results show that the checkpoint data produced by FTPIDA* are much lower than that of CP/R. This is primarily because the FTPIDA* system only

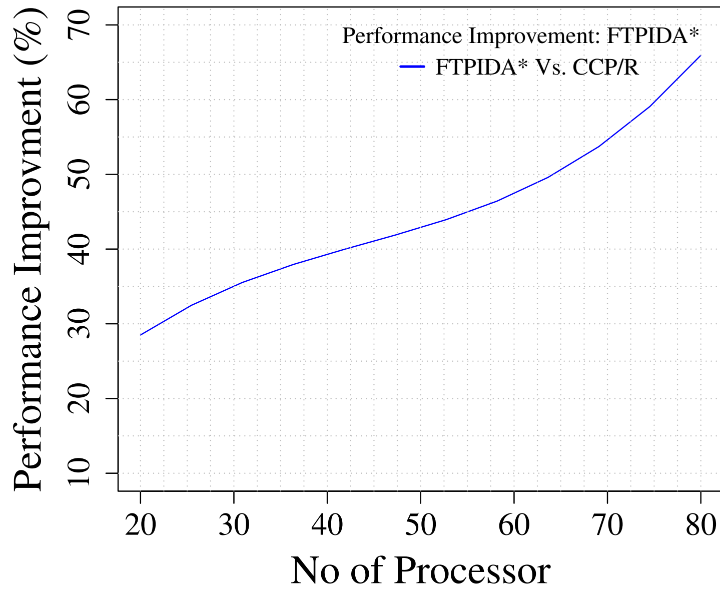


Figure 3.7: Performance Gain: FTPIDA* vs. CP/R

replicates the critical data of its own to two of its neighbor processes. On the other hand, CP/R method freezes the whole systems for a substantial amount of time to collect the backup data from all the processes and create a consistent global backup to recover the system from failure. Therefore, the size of the critical or backup data of FTPIDA* which is needed to recover a system from failure is always smaller than the size of the recovery data of CP/R method.

Table 3.2: Backup data size for FTPIDA* and CP/R

No. of Processes	**Checkpoint Data**	**Backup Data in FTPIDA*(per process)**	**Total Backup Data in FTPIDA***	**Reduction**
20	238.04 MByte	0.86 MByte	17.2 MByte	92.77%
40	358.51 MByte	0.88 MByte	35.2 MByte	90.18%
60	514 MByte	0.90 MByte	54 MByte	89.49%
80	895.12 MByte	0.84 MByte	67.2 MByte	92.49%

3.6 Conclusion

In this chapter, we have presented an algorithm based fault tolerance for parallel search algorithms. It introduces the notion of critical data and also using algorithmic features to design fault-tolerance solution strategy for a class of problems with similar characteristics. Fault-tolerant PIDA* (FTPIDA*) is theoretically analyzed for its performance over CP/R. The simulation results have shown that the proposed fault tolerance strategy has better performance over CP/R regarding fault tolerance and recovery overhead. Also, we have observed from the simulation results that the total memory requirements for backup are significantly less for the proposed FT method as compared to CP/R.

Chapter 4

ABFT for Communication Intensive Parallel Algorithm

In this chapter, we present an algorithm based fault tolerance scheme for parallel dynamic programming algorithms. Dynamic programming algorithms are seen as a collection of interdependent tasks with significant interaction among themselves during the lifetime of the algorithm to solve a given problem. DP can be considered as a multistage problem composed of many sub-problems where subproblems at one level have communication dependency with the subproblems of the previous levels for the computation of tasks. Moreover, all the subproblems belonging to the same level can execute concurrently. The cardinality of data-dependence among subproblems may vary from 1 to n depending on the type of DP problem. We present a detailed analysis of algorithmic and communication characteristics of different parallel DP algorithms: (i) serial monadic, (ii) serial polyadic, (iii) nonserial monadic, and (iv) nonserial polyadic. Further, we show that the algorithmic characteristics of the application determine application data for the checkpoint, and the communication characteristics determine where to replicate those data. With that notion, we propose a generic fault tolerance and recovery protocol for parallel DP which utilizes the

communication dependencies of processes to replicate the checkpointed application-data of a process, in a diskless manner with minimum extra message overhead. We validate our approach with two popular classes of DP problems (i) Longest Common Subsequence (nonserial monadic) and (ii) Traveling Salesman problem (serial monadic) and demonstrate that our proposed method performs better than CP/R for both the cases. Experimental results confirm low fault tolerance overhead over a non-fault tolerant application in a failure-free execution and low recovery cost in the case of single and multiple process failures.

4.1 Motivation

In previous chapters, we have already mentioned that ABFT seems to be an alternative to the existing CP/R for next-generation exascale systems. However, ABFT still has limited applicability in parallel applications because of its non-universality, i.e., it is tied to a particular application or algorithm. In that regard, in this chapter, we present an ABFT scheme for a large class of applications, known as the dynamic programming (DP) class of problems. These problems belong to the class of optimization problems or min/max type of problems where the goal is to find the best solution among all the feasible solutions by maximizing or minimizing an objective function concerning some given constraints. These applications are by nature communication-intensive applications. The communication aspects of an application determine how to distributively save the fault recovery data (we call it the *critical data*) of a process so as to minimize any extra message overhead, and the algorithmic characteristics of an application determine which data is to be saved in order to reduce fault tolerance, and recovery cost as minimizing the FT overhead is a major concern in fault tolerance for HPC systems.

Parallel DP algorithms are used to solve these applications in HPC systems.

Though different parallel DP formulations exist in literature, in a nutshell, all of them exhibit similar characteristics: (i) the computation progresses in successive iterations or stages, and (ii) in each stage, the algorithm computes an optimization function with a set of n different parameters, where each parameter corresponds to the solution of a subproblem, calculated in a previous stage. The optimization function is a pure (mathematical) function with n passing parameters, which does not modify any global state of the application, and updates the local state only when the function completes and commits its update. This optimization function (min/max function) ultimately presents a reduction in information where only one out of n parameters is solely responsible for returning an optimal result for that function. This is the only information that is needed to replicate with peer processes so that the failed process can resume its operation at the failure point with this stored information or data after the failure occurs. This significant feature has a major impact in designing fault tolerance (FT) as minimizing checkpoint data is a major challenge in FT schemes.

4.2 Characteristics of Parallel Dynamic Programming

Dynamic programming (DP) is a well-known algorithmic paradigm to solve optimization problems, which aims to find an optimal solution among several potential ones. DP algorithm solves a complicated problem by breaking it down into simpler subproblems in a recursive manner. DP algorithms proceed by recursively solving a series of subproblems, usually represented as cells in a table. The solution to a subproblem is constructed from solutions to an appropriate set of subproblems.

4.2.1 DP Algorithmic Characteristics

A DP problem is an optimization problem where the solution of a problem is determined based on the solutions of its subproblems. The DP problem can naturally be represented as a multistage graph where each node denotes a subproblem, and a directed edge between two nodes presents the precedence of data dependence among the subproblems. When it depicts as a multistage graph problem, then the nodes can be organized into levels, such that sub-problems at a particular level depend only on sub-problems at previous levels. The solutions to subproblems are maintained in a table, and so DP is a table-driven approach. In the parallel solution to a DP problem, each process solves a set of subproblems and maintains a part of the DP table.
Let $f(x)$ be the solution of any dynamic programming problem x. Here $f(x)$ is the optimal cost or profit associated with the solution, and it can be written as:

$$f(x) = \Phi(r_1, r_2, \cdots, r_n) \tag{4.1}$$

Here Φ is an optimization function (e.g. min or max) and each of the r_i is a solution composed of solutions to sub-problems $\{x_1, x_2, \cdots, x_k\}$, which can be written as:

$$r_i = g(f(x_1), f(x_2), \cdots, f(x_k)) \tag{4.2}$$

In the above, g is called the composition function. Equations 4.1 and 4.2 are part of the generic algorithmic characteristics of all DP problems.

4.2.2 Data Dependency on Different DP Formulation

DP algorithms represent a hierarchy of interdependent sub-problems, where the algorithm advances by recursively solving a series of sub-problems, stored as cells in a

DP table. This recurrence realizes dependence among the sub-problems. Moreover, the dependency defines the stage or level as a set of subproblems whose solutions are independent. Stages introduce a successor-predecessor relationship amongst the sub-problems where the solution of a successor depends on the solution of its predecessors.

Equation 4.1 represents the DP formulation which is a recursive function, where the left-hand side is a function name, and the right-hand side involves optimization of values of a specified cost function. DP formulation is classified into four different categories based on the form of the functional equation and the nature of the recursion: (i) *serial monadic* (TSP problem, 0/1 knapsack problem), (ii) *serial polyadic* (Floyds all pair shortest path), (iii) *nonserial monadic* (LCS problem) and (iv) *nonserial polyadic* (Matrix chain multiplication) [88]. A DP formulation is called monadic if its cost function implicates only one recursive term; otherwise, it is called polyadic. Furthermore, a DP formulation is known as serial if any subproblem in a level can be solved with the subproblems of the immediate preceding levels; otherwise, it is nonserial.

In the following, we show the DP recurrence expression for well-known serial and nonserial DP problems which illustrate their dependency pattern successively.

- Travelling Salesman Problem (TSP): TSP is an example of serial monadic dynamic programming. Formally, TSP is defined as: given a set of n cities $\{1, 2, \cdots, n\}$; and the distances between the cities $d_{i,j} : i, j \in \mathbb{Z}^+$, the goal is to find the shortest tour which originates and terminates at the same city, subject to the constraint that each of the remaining cities must be visited once, and only once. It is an immense computational complex problem which is at the core of many routing and scheduling problems.

DP formulation for the travelling salesman problem (TSP) becomes:

$$c[S,j] = \begin{cases} d_{1,j} & S = 2 \\ \min\{c[S-\{j\},i] + d_{i,j})\} & \text{j}\in \text{S} \wedge \text{j} \neq \text{i} \wedge \text{j} \neq 1 \end{cases} \tag{4.3}$$

Here, $c[S,j]$ be the cost of an optimal tour starting at city 1, visiting each of n cities of $S = \{1, 2, \cdots, n\}$ exactly once and ending up in city j.

- Longest Common Subsequence Problem (LCS): LCS is an example of nonserial monadic dynamic programming problem. LCS is defined as: given two strings $X =< x_1, x_2, \cdots, x_n >$ and $Y =< y_1, y_2, \cdots, y_m >$ over alphabet Σ, the goal is to find the common subsequence Z of maximum length, which means Z is the common subsequence of X and Y and there is no other common subsequence whose length is larger than Z. The objective is to determine $t[n,m]$. LCS is a classic computer science problem which is used often in bioinformatics.
DP formulation of the LCS problem is:

$$t[i,j] = \begin{cases} 0 & i = 0 \vee j = 0 \\ t[i-1, j-1] + 1 & i, j > 0 \wedge x_i = y_j \\ \max\{t[i, j-1], & \\ (t[i-1, j\} & i, j > 0 \wedge x_i = y_j \end{cases} \tag{4.4}$$

$t[i,j]$ represents the length of the longest common subsequence of the first i elements of X and the first j elements of Y. Ultimately, the LCS length of the two strings can be found at $t[n,m]$.

- 0/1 Knapsack Problem: This classical optimisation problem belongs to the serial monadic dynamic programming class of problems. The 0/1 knapsack problem is described as follows: Given n distinct elements, each ith element associated

with unique weight w_i and profit v_i respectively and the knapsack with capacity C, where $w_i, v_i, C \in \mathbb{Z}^+$; and the goal is to find the set of elements to maximize the total profit of the knapsack within the capacity limit of the knapsack. The 0-1 Knapsack Problem restricts the number of elements each item can be selected to zero or one. More formally, let $X = [X_1, X_2, \cdots, X_n]$ be a solution set of the elements in which $X_i = 1$ if the ith element is included in the knapsack, otherwise it becomes 0.

DP formulation for this problem is :

$$f[i,m] = \begin{cases} 0 & m \geq 0, i = 0 \\ \infty & m < 0, i = 0 \\ \max\{f[i-1,m], & \\ (f[i-1, m-w_i] + p_i)\} & 1 \leq \text{i} \leq \text{n} \end{cases} \tag{4.5}$$

where, $f[i, m]$ is the maximum profit for a knapsack with a capacity of m using only items $\{1, 2, 3, \cdots, i\}$. Then, let $f[n, C]$ be the maximum profit for a knapsack with a capacity of C using only items $\{1, 2, 3, \cdots, n\}$,which represents the solution.

- Matrix Chain Multiplication Problem (MCM): MCM is a typical example of nonserial polyadic dynamic programming problem. This problem is also known as optimal matrix parenthesization problem. The MCM problem is defined as: given a chain of n matrices, the goal is to determine the optimal sequence of matrix multiplications with the minimum number of scalar multiplications.

  The DP formulation for MCM becomes:

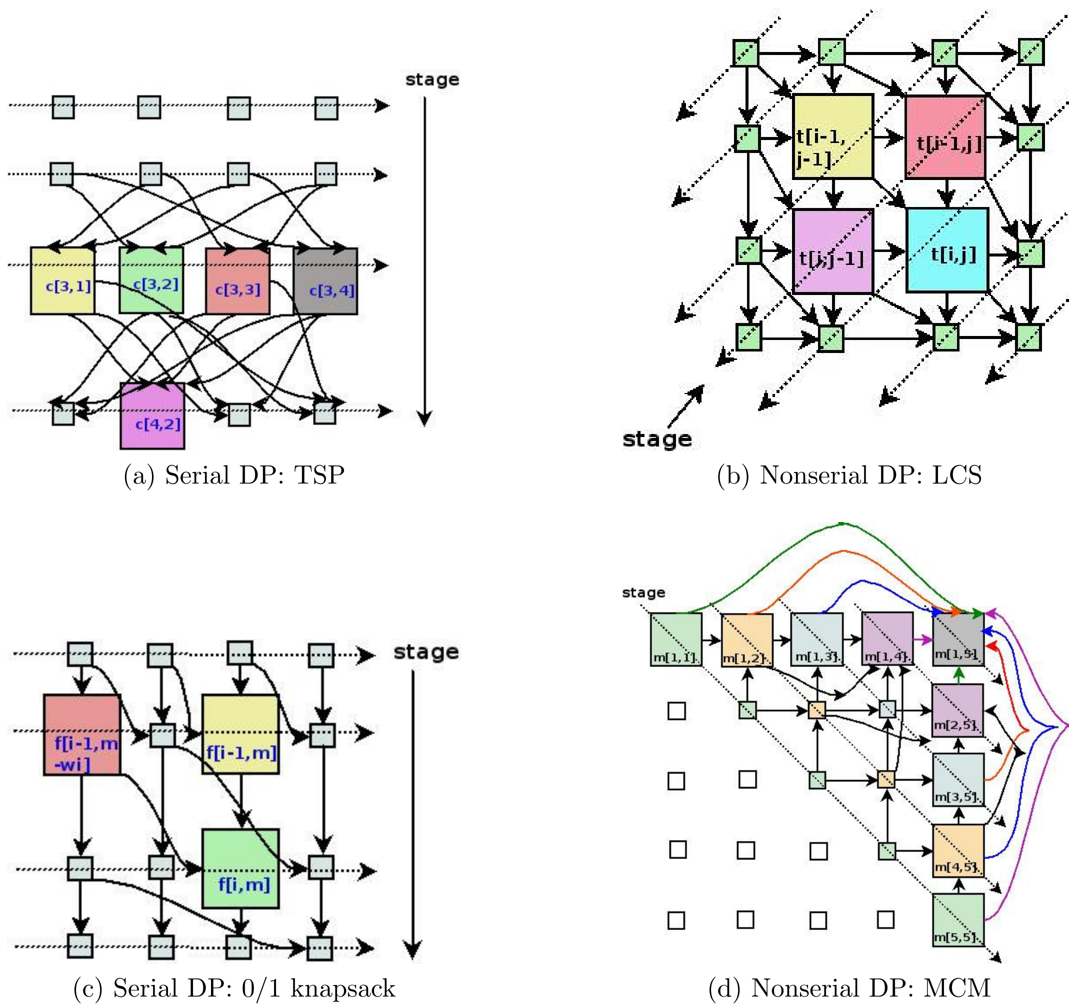
$$m[i,j] = \begin{cases} 0 & j = i, 0 < i \leq n \\ \min\{m[i,k] + m[k+1,j], & \\ r_{i-1}r_k, r_j\} & 1 \leq i < j \leq n, i \leq k < j \end{cases} \tag{4.6}$$

where, m[i,j] represents the optimal multiplication cost of multiplying the matrices $A_i, \cdots, A_j$ means the minimum number of operations for this chain of multiplication. This chain of matrices can be expressed as a product of two smaller chains, $A_i, A_{i+1}, \cdots, A_k$ and $A_{k+1}, \cdots, A_j$. The chain $A_i, A_{i+1}, \cdots, A_k$ results in a matrix of dimensions $r_{i-1} * r_k$, and the chain $A_{k+1}, \cdots, A_j$ results in a matrix of dimensions $r_k * r_j$. The cost of multiplying these two matrices is $r_{i-1} r_k r_j$.

Different DP problems have different dependency patterns. In figure 4.1, we show the dependency and flow of computation for serial and nonserial DP problems respectively. Here, we also portray the cardinality of data-dependence among the subproblems of different stages. In the case of TSP problem, the parallel computation proceeds vertically row-wise, where the amount of computation per subproblem increases monotonically as the computation proceeds to the next level, shown in figure 4.1a. Formally, we can state that TSP is a serial monadic DP class of problem with a monotonically increasing data-dependence with stages and which is $O(n)$ for an n-stage problem.

On the other hand, for LCS, illustrated in figure 4.1b, computation proceeds diagonally and the amount of computation per subproblem is fixed, i.e. the cardinality of the data-dependence remains constant with each stage, and it becomes $\Theta(3)$. Similarly, figure 4.1c shows that the computation advances row-wise using a vertical approach for 0/1 knapsack problem and the cardinality of the data-dependence is fixed with stages, and it becomes $\Theta(2)$.

In MCM, elements are computed in parallel using a diagonal approach, portrayed in figure 4.1d, where computation per subproblem increases with each next stage.



(a) Serial DP: TSP

(b) Nonserial DP: LCS

(c) Serial DP: 0/1 knapsack

(d) Nonserial DP: MCM

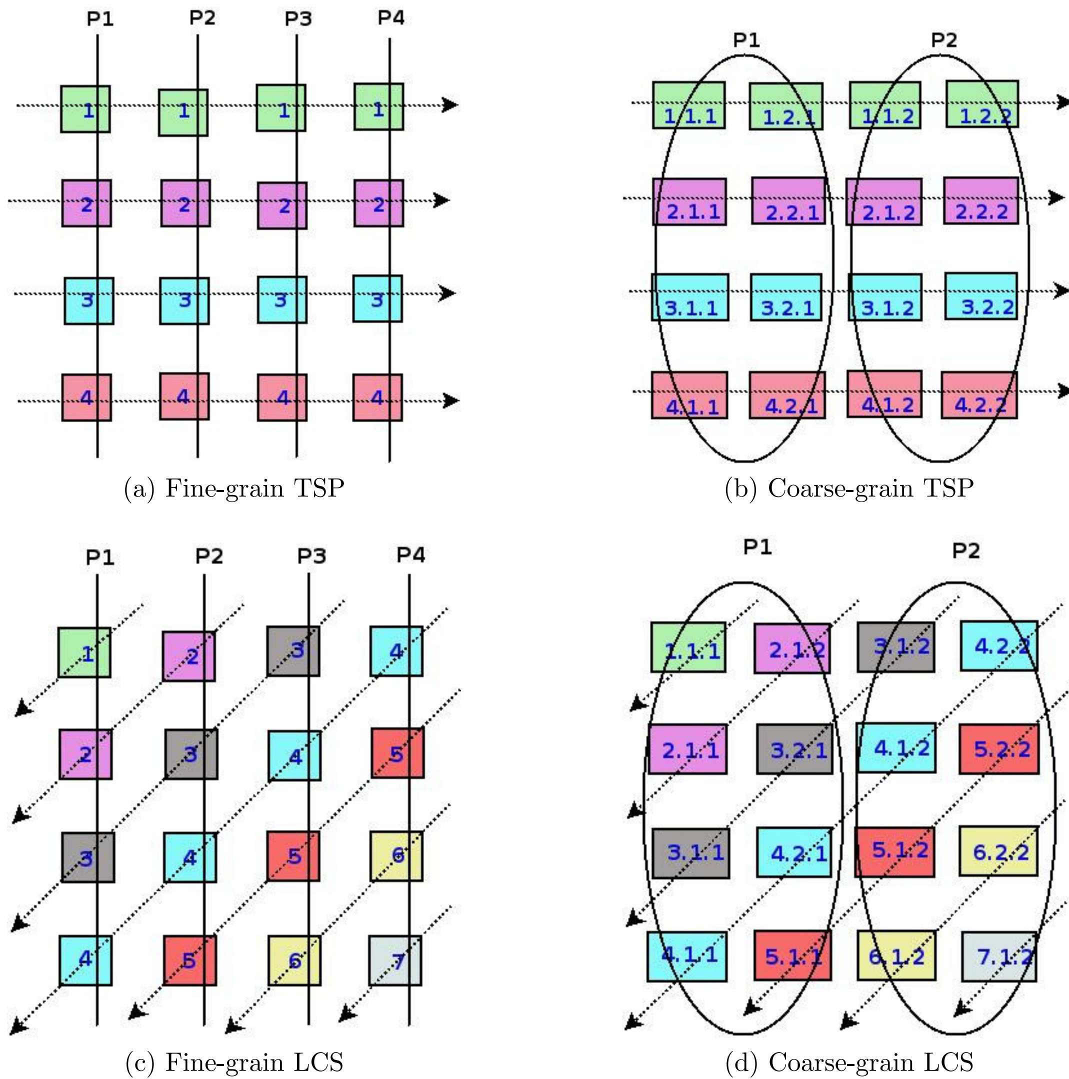
Figure 4.1: Computation Dependency in Different DP Problems

4.2.3 Parallelism in DP

We have already mentioned that in DP algorithms, the computation proceeds by filling the cells of the DP table through iterate over the consecutive series of stages. Once, a stage is computed, it satisfies the dependency for the subsequent stages. For, 0/1 knapsack and TSP problems, cells of each stage are calculated row-wise in the vertical fashion, and for LCS and MCM, this happens in an anti-diagonal manner. Earlier, we mentioned that all the subproblems belonging to a particular stage are independent of each other and can execute concurrently.

For parallel formulation, the DP table is evenly split among the processes such that each process is assigned a single column or multiple columns. All the subproblems belonging to a particular stage are divided and distributed among the processes either in fine-grain or coarse-grain manner, wherein all the processes execute the subproblems, assigned to themselves, in parallel. We illustrate this with two well-known classes of DP problems: (i) TSP, and (ii) LCS.

Figures 4.2a, 4.2b and 4.2c, 4.2d show the coarse-grain and fine-grain distribution for a TSP and LCS problem successively. For a fine-grain distribution, cells marked with the same number compute at the same time by different processes concurrently. On the other hand, for coarse-grain allocation, each cell is represented as a format of $x.y.z$ where x serves the level or stage, y presents the process number, and finally, z expresses the subproblem number, respectively. The subproblems with the same level number x and same subproblem number z can be calculated in parallel.
We have seen from figure 4.1 that the computation of each cell or subproblem needs help from its neighbors from previous stages. So, neighboring processes assist each other in computing for the current iteration with data from previous iterations. Figure 4.3 shows the DP table for the LCS problem for the strings $X = BACBAD$ and $Y = ABAZDC$ for both sequential and parallel formulation. Figure 4.1b and figure 4.3b illustrate that computing a cell $t[i, j]$ indexed by row i and column j process P_j depends on (i) $t[i-1, j]$ handled by P_j itself and (ii) $t[i-1, j-1]$ and $t[i, j-1]$ handled by processor P_{j-1} respectively, from the previous stages.
Figure.4.4 represents a DP solution for a TSP problem for a set of four cities. A distance matrix for the graph is given in fig. 4.4b, wherein each entry expresses the direct communication cost between cities, which is global problem-specific information, available to all the participating processes of the system. Figure 4.4d depicts the computation of each cell in the DP table. Figure 4.4e and figure 4.1a portray that for



(a) Fine-grain TSP

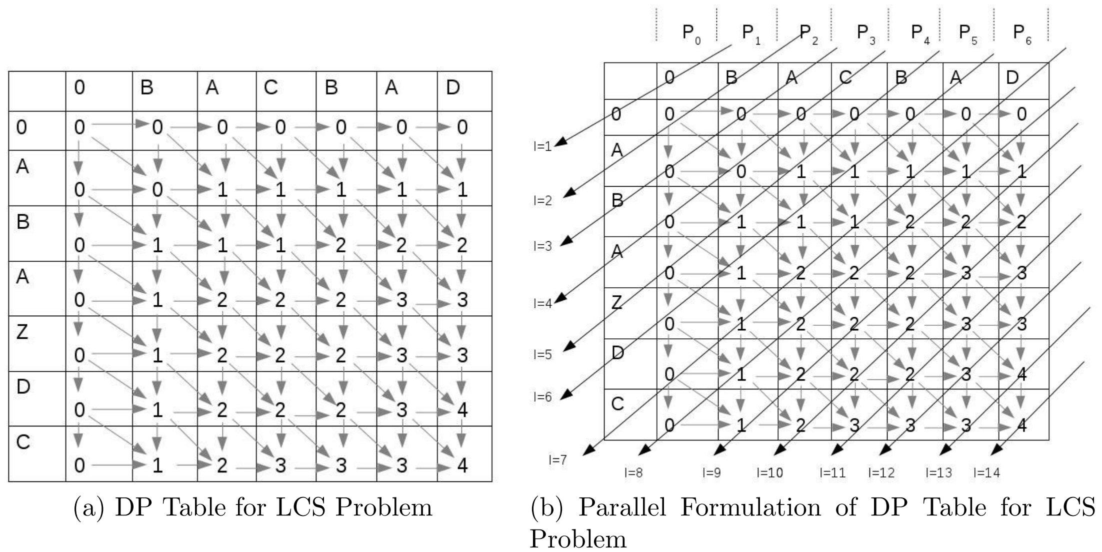
(b) Coarse-grain TSP

(c) Fine-grain LCS

(d) Coarse-grain LCS

Figure 4.2: Fine-grain and Coarse-grain Parallel Distribution. Cells marked with the same number execute in a parallel manner

the solution of the TSP problem, process P_j during iteration i for the computation of $c[S, j]$ depends on at most a total of n number of data from the previous iteration $i-1$ from all other P_x processes, where $j \in S \wedge j \neq x$ and $S = \{1, 2, ..., n\}$ represents the total number of cities.



(a) DP Table for LCS Problem

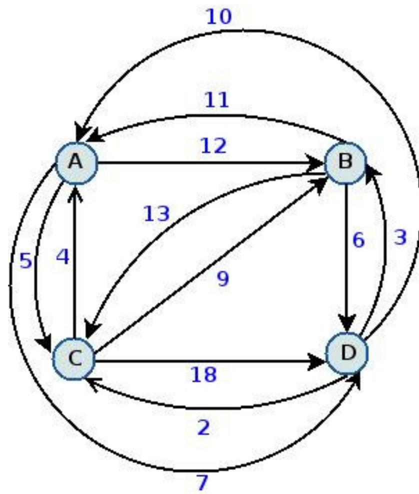
(b) Parallel Formulation of DP Table for LCS Problem

Figure 4.3: Dynamic Programming (DP) Table for LCS of strings $X = BACBAD$ and $Y = ABAZDC$

4.2.4 Parallel DP Communication Characteristics

The communication pattern among the processes during each iteration for the parallel DP solution to the LCS problem is shown in figure 4.5a, based on the assumption that each column of the DP table is mapped to one process. It is assumed that the communication model for this particular solution is based on the request-reply protocol. It is evident from the figure that for process P_j, it becomes process P_{j-1} which remains its dependent peer for the computation throughout the lifetime of the application.

In comparison, figure 4.5b shows the communication characteristics of a fine-grained parallel DP solution to the 0/1 knapsack problem, where process P_m depends on process P_{m-w_i} on iteration i to compute $f[i, m]$. But, for the next iteration $i + 1$, it becomes a different process $P_{m-w_{i+1}}$. Unlike, LCS, the identity of the dependent peer varies with iteration. Though these two are entirely distinct parallel DP formulations, it can be seen that their communication characteristics are quite similar.



(a) Graph of Cities

	A	B	C	D
A	—	12	5	7
B	11	—	13	6
C	4	9	—	18
D	10	3	2	—

(b) Distance Matrix Table

Algorithm 1 DP algorithm for TSP problem

1: **function** DP_TSP()
2: **for** all $j \in n$ **do**
3: c(1,j)= d_{1j}
4: **end for**
5: **for** $s = 2 \longrightarrow n$ **do**
6: **for** all subsets S = {1,2,..,n} of s $\wedge$ contains 1 **do**
7: **for** all $j \in S \wedge j \neq 1 \wedge j \neq i$ **do**
8: c[S,j]=min{c[S-{j},i]+$d_{i,j}$}
9: **end for**
10: **end for**
11: **end for**
12: **end function**

(c) DP Algorithm for TSP [37]

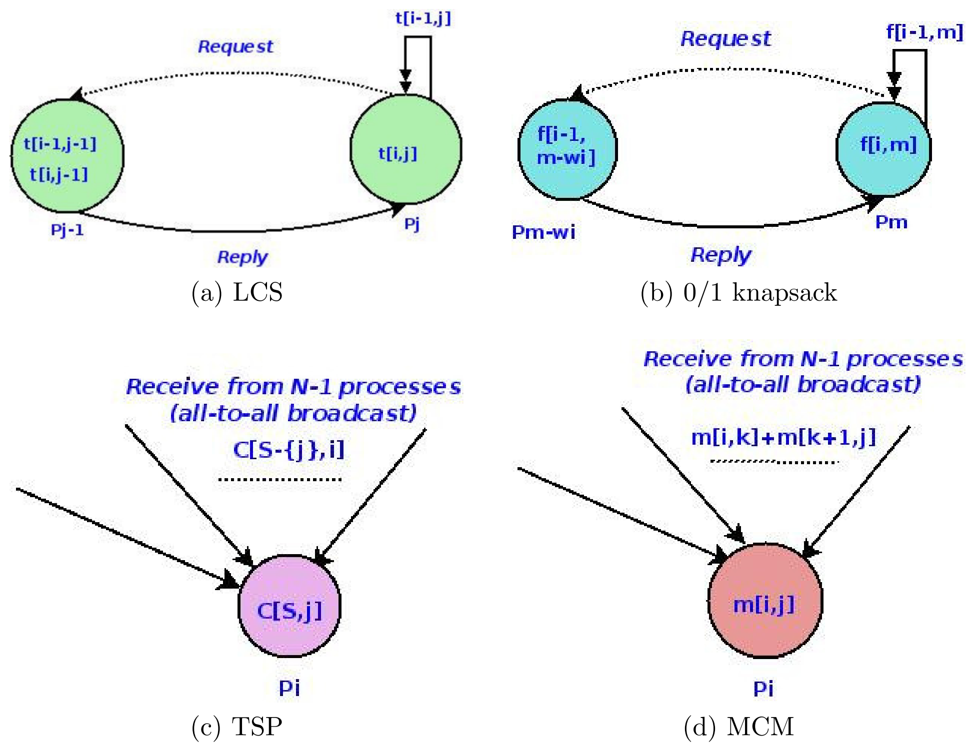
		A	B	C	D
s=0	A	—	11	4	10
s=1	{B}	23	—	20	14
	{C}	9	17	—	6
	{D}	17	16	28	—
s=2	{B,C}	25	—	—	20
	{C,D}	13	12	—	—
	{B,D}	21	—	25	—
s=3	{B,C,D}	24	—	—	—

(d) DP Table

		P1	P2	P3	P4
		A	B	C	D
s=0	A	—	11	4	10
s=1	{B}	23	—	20	14
	{C}	9	17	—	6
	{D}	17	16	28	—
s=2	{B,C}	25	—	—	20
	{C,D}	13	12	—	—
	{B,D}	21	—	25	—
s=3	{B,C,D}	24	—	—	—

(e) Parallelism in DP Table Computation

Figure 4.4: DP Solution of a Sample TSP Problem



(a) LCS

(b) 0/1 knapsack

(c) TSP

(d) MCM

Figure 4.5: Communication characteristics of parallel DP

For the parallel DP solution to the TSP problem, process P_i can have a maximum communication dependency with $n-1$ processes at iteration n, where $2 \leq n \leq N$ and, N is the total number of cities, illustrated in figure 4.5c. In comparison to figures 4.5b and 4.5a, the communication model here is synchronous, involving all-to-all broadcast at each iteration. Figure 4.5d shows the communication characteristics of a fine-grained parallel DP solution to the MCM problem, which has a similar communication dependency pattern to the TSP problem.

4.2.5 General Computation and Communication Characteristics

In general, all serial/nonserial monadic/polyadic DP formulations exhibit similar computational and communication characteristics, as illustrated in figure 4.6. The following are the common characteristics: (1) the computation progresses in successive iterations; (2) in each iteration, a process computes an optimization function Φ with m parameters $par_1, ..., par_m$. Each parameter par_i corresponds to the solution of a subproblem, and is computed either locally in a previous iteration or is received from a different process in the same or a previous iteration. Thus, at the start of each iteration i before the computation of Φ, every process receives solutions to subproblems of iteration $(i-1)$ as parameters to Φ from a set of processes $P_R(i)$ and also sends its locally computed solution(s) to subproblem(s) of iteration $(i-1)$ to a set of processes $P_S(i)$. We name $P_R(i)$ and $P_S(i)$ as *Receiver_Set* and *Sender_Set* respectively for process P. The function Φ is a pure (mathematical) function, i.e., it is side-effect free in a sense that it does not modify any global state of the application, and updates the local state only when the function completes and commits its update. Moreover, function Φ is by nature deterministic; i.e., it always returns the same output with the same set of input parameters. Or in other ways, it always returns the same output state with the same input state at any time of the execution. (3) As well, it is assumed that the *Sender_Set* and *Receiver_Set* of each process P are known globally to every other process. This is an assumption and not a mandatory requirement; the assumption holds for the dynamic programming class of applications mentioned here and will be used in their fault recovery scheme discussed in section 4.3.4.

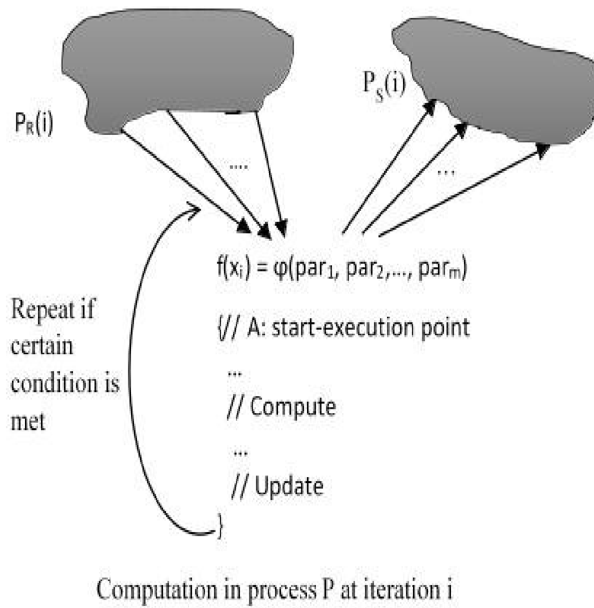


Figure 4.6: General Algorithmic and Communication Characteristics

4.3 Fault Tolerance for Parallel Dynamic Programming

We have developed a general fault-tolerance principle for the parallel dynamic programming class of applications for recovering from fail-stop failures with a goal of minimum fault tolerance and recovery overhead. The goal is accomplished by replicating the minimum application-level fault-recovery data with peers, using the natural dependency pattern of the original algorithm. In that regard, we have shown that among a maximum of n-dependency, only one single entity is solely responsible for generating an optimized result for a particular problem, and this is the only information that is needed to replicate with peer processes, so that the failed process can resume its operation at the failure point with this stored information or data, after failure occurs. We define this information as "critical data."

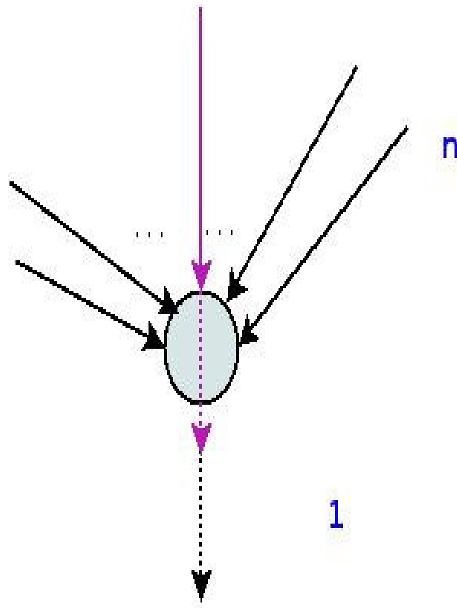


Figure 4.7: General Algorithmic Characteristics of DP with $n : 1$ Cardinality

4.3.1 Critical Data

The functional equation for DP is based on Bellman's principle of optimality [58]. This states that in an optimal policy, whatever the initial state and initial decisions are, the remaining must constitute an optimal policy concerning the state resulting from the first decision [9, 8]. This asserts that for an n-stage process, to find the optimal policy, for each state, we must find the optimal decision that optimizes the cost function. Additionally, it characterizes that the next state at any stage depends only on the current state and the current decision, no other information from the previous states are needed. Furthermore, the optimal cost of any state is only possible when we consider the optimal cost of the previous state. Thus, we can formally express that among the n number of different options, a particular decision led to the optimal value of $f(x)$ in a stage.

Figure 4.7 shows a directed graph with indegree of n and outdegree of 1 that describes the graphical representation of the functional characteristics of any DP problem where ***one out of*** n different inputs is solely responsible for generating the optimized

output.

We define *critical data* (CD) as the minimal application data that is required to recover a failed process from a previous *consistent state*. Referring to equation 4.1, the critical data for a process executing the optimization function Φ is a particular value of r_i among its parameter set $\{r_1, r_2, \cdots, r_n\}$ in order to recover the failed process from the start-execution point of Φ. This is based on the assumption that Φ is a side-effect free function (section 4.2.5).

In general, referring to figure 4.6, the critical data of a process P in order to recover it from the start-execution point of Φ at iteration i is one of the members of its parameter set $\{par_1, \cdots, par_m\}$. Some of these parameters may be computed locally in P during previous iteration(s) while the others are received in messages from the processors that belong to $Receiver_Set$, $P_R = \bigcup_i P_R(i)$. We call these two parts of the critical data : (i) the locally computed part as $CD_{Local}(P, i)$ and (ii) the remotely received part from P_R as $CD_{Remote}(P, i)$. Let us additionally define, $P_S = \bigcup_i P_S(i)$, $CD_{Local}(P) = \bigcup_i CD_{Local}(P, i)$, and $CD_{Remote}(P) = \bigcup_i CD_{Remote}(P, i)$.

It can be seen from the previous discussion that $f(x_{i-1}) \subseteq CD_{Local}(P, i)$, where $f(x_{i-1})$ is a solution to a subproblem at iteration $i - 1$. Moreover, $CD(P, i) = \{CD_{Local}(P, i) \;\; or \;\; CD_{Remote}(P, i)\}$. As an example, for the TSP problem with n cities (equation (4.3) and figure 4.5c), the critical data for any failed process P_i to recover it at $c_s[S, j]$ during iteration s is one particular value $c_{s-1}[S - j, i]$ among n alternatives from the previous stage $s - 1$.

Similarly, for the LCS problem (eqn. (4.4) and figure 4.5a), the critical data for the failed process P_j to recover it at $t[i, j]$ at a stage s becomes one of the following three dependent entities:

$$CD(P_j, t[i, j]) = \{(t[i, j-1]) \;\; or \;\; (t[i-1, j]) \;\; or \;\; (t[i-1, j-1] + 1)\} \tag{4.7}$$

Axiom 1. *For every process* $P' \in P_S(i)$ *: (i)*$P \in P'_R(i)$ *and (ii)* $f(x_{i-1}) \subseteq CD_{Remote}(P')$*.*

Since $CD(P,i) = \{CD_{Local}(P,i)\ or\ CD_{Remote}(P,i)\}$, based on Axiom 1 it can be seen that the critical data of process P during iteration i is replicated among the processes in $P_S(i)$ alongside the inherent communication of the original non-FT algorithm. This property will be useful in designing the FT protocol discussed in the next subsection.

4.3.2 Consistent State

We define a locally consistent state for a process P as the start-execution point of Φ in an iteration i. Referring to figure 4.6, this is point A. A globally consistent state (or consistent state) for the parallel program is the state of a global cut comprising of one locally consistent state from each process so that the cut is consistent [45]. In the fault-tolerance strategy discussed in the following subsection, local critical data, $CD_{Local}(P,i)$ of each process P is replicated with the other processes in the *Sender_Set* and *Receiver_Set* (Figure 4.6).

Upon failure, a failed process is recovered independently from a previous locally consistent state without rolling back the other processes and is discussed in section 4.3.4. This is somewhat analogous to the recovery of a failed process in the case of independent checkpointing with logging where any inter-process dependencies are resolved via message logging. In our case, there is no explicit logging of messages. However, the required data for the recovery of a process is available in the *Sender_Set* and *Receiver_Set* of a process and is recovered upon the failure of a process without needing to roll back any other healthy processes.

4.3.3 Fault Tolerance Protocol

The basic idea of the generic fault-tolerance (FT) protocol is to replicate the locally computed critical data, $CD_{Local}(P)$, of a process P among the processes in P_R and P_S, which P is naturally dependent on as part of the original non-FT algorithm. The goal is to implement an FT strategy based on the original application's inherent communication dependencies to minimize any extra message overhead.

The protocol dictates "how to replicate" the critical data for the class of applications characterized in section 4.2.5 and illustrated in figure 4.6. The specifics of "what to replicate," i.e., what the content of the critical data is, depends on the algorithmic characteristics of the particular application, which is already mentioned in section 4.3.2 with a serial-monadic DP problem TSP and a nonserial-monadic DP problem LCS.

Protocol 1: Fault Tolerance Protocol

Assumptions: (1) We assume that the communication model is asynchronous, based on the request-reply protocol. This assumption facilitates distributed fault detection used in our approach. Thus, referring to figure 4.6, every message from P_R to process P is preceded by a request message from P. Similarly, every message from process P to P_S is preceded by a request message from P_S. The request messages are shown in the figure 4.5. (2) All communications are reliable. (3) Let $\mathcal{U}$ = the set of all processes, where $|\mathcal{U}| = N$. Let $|P_R(i)| = m$ and $|P_S(i)| = n$, where $1 \leq m, n \leq N$. Moreover, let $|P_R(i) \cap P_S(i)| = c \geq 0$, which implies that the *Sender_Set* and *Receiver_Set* may not be disjoint.

The protocol: At the beginning of each iteration i, prior to computing Φ, each process P makes k-way replication of its local solution to the subproblems or CD_{Local} at iteration $(i-1)$ among the processes in the set $P_k(i)$, where $k \leq N-1$. Since process P sends $f(x_{i-1})$ to each process in $P_S(i)$ as part of the original non-FT

algorithm (section 4.2.5 and axiom 1), this implies that $P_S(i) \subseteq P_k(i)$.

One of the following three cases arises:

Case 1: $P_k(i) = P_S(i)$: implies process P already sends $CD(P,i)$ to each process in P_S as part of the original non-FT algorithm (section 4.2.5 and axiom 1), and hence no additional fault-tolerance specific message needs to be sent.

Case 2: $P_k(i) \supset P_S(i) \wedge |P_k(i) - P_S(i)| \leq |P_R(i) - P_S(i)|$: implies (1) $P_k(i) - P_S(i) \subseteq P_R(i) - P_S(i)$, and (2) at the start of an iteration i, process P sends $CD(P,i)$ to each of the $k-n$ processes in $P_R(i) - P_S(i)$ by piggybacking it with the request message, which is part of the original non-FT algorithm.

Case 3: $P_k(i) \supset P_S(i) \wedge |P_k(i) - P_S(i)| > |P_R(i) - P_S(i)|$: implies (1) $P_k(i) - P_S(i) = (P_R(i) - P_S(i)) \cup P_E(i)$ where $P_E(i) \subseteq U - (P_R(i) \cup P_S(i))$, and (2) at the start of an iteration i, process P sends $CD(P,i)$ to each of the processes in $P_R(i) - P_S(i)$ by piggybacking it with the request message, and to each of the processes in $P_E(i)$ in an explicit message.

End of protocol 1

The following two lemmas are the consequences of the above protocol.

Lemma 4.2. *Let* $P_k = \bigcup_i P_k(i)$ *. Then, when using protocol 1, the processes in* $P_k \cup P_R$ *collectively have all the information required to recover* P *from an iteration* $\leq i$.

Proof This immediately follows from the following: (i) by definition, processes in P_R collectively contain $CD_{Remote}(P)$. (ii) Following axiom 1, each member of $CD_{Local}(P)$ is replicated with processes in P_S as part of the original non-FT algorithm. (iii) As a result of protocol 1, each member of $CD_{Local}(P)$ is replicated with processes in $P_k - P_S$, either in a piggyback or in an explicit message. Thus, at start-execution point A (figure 3) in iteration i, processes in $P_k \cup P_R$ collectively have $CD(P) =$

$CD_{Local}(P) \cup CD_{Remote}(P)$, which is all the information required to restart process P from point A in the same iteration or an earlier iteration. This concludes the proof.

Lemma 4.3. *Protocol 1 results in minimum extra message overhead while satisfying Lemma 4.2.*

Proof Explicit protocol messages are required for replicating $CD(P,i) = \{CD_{Local}(P)$ *or* $CD_{Remote}(P)\}$ only in case 3 of protocol 1. This concludes the proof.

As an example of protocol 4.3.3, algorithm 4.2 describes the fault tolerant algorithm for parallel DP solution of LCS problem. Moreover, the parallel dynamic programming algorithm for the LCS problem is given in algorithm 4.1.

Algorithm 4.1 Parallel Dynamic Programming Algorithm for LCS Problem

1: Initialize the DP table t by $t[i,0] = t[0,j] = 0$ where $i = 0,1,...,N$ and $j = 0,1,...,(C-1)$ ▷ Initialization Phase
2: Partition t into C columns and assign to p processors where $p = C$ ▷ Partition Phase
3: For each processor p_j, in each iteration l, during computation of $t[i,j]$
4: **if** entries of both the strings are not identical **then**different
5: Send data request for $t[i-1,j]$ to P_{j-1}
6: Receive data from P_{j-1}
7: Compute $t[i,j] = max\{t[i,j-1], (t[i-1,j]\}$
8: **else**
9: Send data request for $t[i-1,j-1]$ to P_{j-1}
10: Compute $t[i,j] = t[i-1,j-1] + 1$
11: **end if**

4.3.4 Fault Recovery Protocol

Failure of a process causes the loss of all data that belongs to the failed process. The recovery protocol dictates how the critical data of the failed process that is distributed amongst the other processes (in other processors) can be used to reconstruct the lost data of the failed process and bring back the failed process to its most recent start-execution point of Φ. For the dynamic programming (DP) class of problems, failure recovery is composed of two components: (i) recover the failed process from the most

Algorithm 4.2 Fault Tolerant Parallel Dynamic Programming Algorithm for LCS Problem

1: Initialize the DP table t by $t[i, 0] = t[0, j] = 0$ where $i = 0, 1, ..., N$ and $j = 0, 1, ..., (C - 1)$
2: Partition t into C columns and assign to p processors where $p = C$
3: For any processor p_j, in an iteration i, during computation of $t[i, j]$
4: **if** entries of both the strings are not identical **then**
5: Send data request for $t[i, j - 1]$ and critical data to immediate neighbor processor, to P_{j-1}
6: Send reply of the request to the neighbor processor P_{j+1} with (i) requesting data , (ii) critical data
7: **else**
8: Send data request for $t[i - 1, j - 1]$ and critical data to immediate neighbour processor, to P_{j-1}
9: Send reply of the request that it gets from neighbour processor P_{j+1} with (i) requesting data , (ii) critical data
10: **end if**
11: Receive data request and critical data from neighbour processor, P_{j+1} and keep backup of critical data of neighbour processor P_{j+1}
12: Receive reply from the neighbour P_{j-1} with (i) requesting data , (ii) critical data and keep backup of critical data of neighbour processor P_{j-1}

recent recoverable point and (ii) recover the DP table and any other global data that belonged to the failed process.

Referring to the traveling salesman problem, failure recovery becomes the recovery of the computation of the failed process, the column(s) of the DP table that belongs to the failed process (figure 4.2 and figure 4.4e) and global data that is the given distance matrix, which contains the distances between the cities. Similarly, for the nonserial DP problem LCS, failure recovery becomes the column(s) of the DP table of the failed process (Figure 4.2 and figure 4.3b) and the global information that is the given strings.

We assume distributed failure detection: when a process fails, its failure is detected by its alive peer(s). More specifically, in this case, failure of a process P is identified by a peer (or peers) in its $Sender_Set$, P_S while it sends a data request to P. Distributed failure detection [58] is a well-known technique for failure detection in asynchronous message-passing systems.

The protocol:

At the beginning of iteration i, process P_k detects failure of its neighbour P_x,

where $P_k \in P_{xS}(i)$. P_k takes the help from the *Receiver_Set*, $P_{xR}(i-1)$, which is known to the processes in $P_{xS}(i)$ (section 4.2.5), to recover the lost critical data of P_x from iteration $i-1$ and lost data from previous iterations (e.g. portions of the DP table and any global data). Let $\Theta(P_x, i)$ be the lost data of P_x at the start of iteration i and let it be represented by the following formula:

$$\Theta(P_x, i) = \bigcup_{P_z} \Theta(P_z, i-1) + \psi \tag{4.8}$$

In the above, referring to Protocol 1, $\Theta(P_z, i-1) \in CD(P_x, i-1)$, $P_z \subseteq P_{xk}(i-1)$, and ψ represents any problem-specific global data from iterations prior to $i-1$ that is lost due the failure of P_x.

1. *The case with the travelling salesman problem:* For a fine grained DP solution to the TSP problem, wherein each process is assigned the computation of a single column of the DP table, the recovery equation from equation 4.3 becomes:

$$c_s[S, j] = c_{s-1}[S - \{j\}, i] + d_{i,j} \tag{4.9}$$

   here, $c_s[S, j]$ becomes the lost data of the failed process at iteration s, that is recovered with the help of the critical data $c_{s-1}[S - \{j\}, i]$ from the previous iteration $s-1$ and resumes the execution of the failed process at iteration s.

2. *The case with the longest common subsequence problem:* Refereeing to parallel DP formulation (figure 4.3) and communication characteristics (figure 4.5a) of the LCS problem, it is observed that for every cell computation of the DP table, process P_j always has a dependency with process P_{j-1} for all the iterations. From equation 4.7, it can be seen that the critical data for a failed process P_j to resume its computation $t[i, j]$ at iteration l after failure is one of the

following three values :(i) the left element $t[i, j-1]$ from process P_{j-1}, (ii) the upper element $t[i-1, j]$ from itself, and (iii) the upper-left element plus one $(t[i-1, j-1]+1)$ from process P_{j-1}. So, the recovery equation becomes:

$$t_l[i, j] = CD(P_j, t[i, j]) \tag{4.10}$$

where, $t_l[i, j]$ becomes the lost data of the failed process P_j at iteration l. It is evident from equation 4.10 that the P_j's computation is resumed with the help of critical data from its peer processes (process P_{j-1} and process P_{j+1}) with which the critical data was replicated as a part of fault tolerance protocol.

DP Table Recovery: Recovery of the DP table that belongs to the failed process can happen after resuming the operation of that process. Suppose the process fails at iteration i, then the lost entries of the DP table are recovered in the following way: suppose recovered process resumes operation from iteration $i-j$ onward. Then the lost entries of the DP table from iterations $(i-j-1)$ till iteration 1 can be recovered in the same way as in Case 1.1 or Case 1.2 above using the recovery equation (4.8). Since the process can resume its normal operation even if its DP table is not completely recovered, this operation can be overlapped with the normal execution of the resumed process.

Thus it can be seen that the recovery of the failed data and resumption of the failed process are the hot spots of the recovery operation. However, the recovery of the DP table can be carried out in the background and thus the cost of DP table recovery can be masked by the normal execution of the application. How much of the lost DP table needs to be recovered depends on how much backtracking is to be allowed for the recovery of another failed process, since the DP table holds the remote critical data of other processes; and this determines the pruning decision.

4.4 Performance Analysis

In this section, we present an analysis of the algorithmic fault tolerance for parallel dynamic programming algorithms. We describe it using examples of serial and nonserial DP problems, like the TSP and the LCS problems.

4.4.1 An analysis of message optimization during fault tolerance

We have already explained the critical data for any DP problem in section 4.3.1. In the case of the TSP problem, dependency with the number of subproblems increases monotonically as the computation proceeds to the next stage or iteration. On the other hand, for the LCS problem, it remains constant with the stage. For such application, storing only one among n dependant data will save a significant amount of storage which in turn reduce memory and recovery overhead of fault tolerance and recovery scheme. Here, we show how much gain is achieved in the checkpoint data when we save the critical data.

- TSP problem: Suppose there are N cities, where the tour starting at city 1, visiting each of n ($n = N - 1 \wedge n \neq 1$) cities exactly once, and finally ends up in city 1. The algorithm iterates through the increasing value of s, where $1 < s \leq n$ and also s represents the subset size of cities (see figure 4.4c).

  Let,

  1. $X_s = n_{c_s}, :$ be the total number of subsets generated with each s. It also represents the total number of rows with each s in the DP table (see figure 4.4e).

  2. $P = n - s, :$ be the total number of cells in each row generated during s. Each cell represents the solution of a subproblem.

3. $X_p = P * X_s$, : be the total number of subproblems generated with each s.

During iteration s, each subproblem of a row depends on an exactly s number of subproblems from previous rows generated in iteration $(s-1)$, meaning each subproblem has a dependency cardinality of $O(s)$. As only one out of s subproblems is responsible for the optimal return of the current subproblem, instead of replicating all the dependent s entities, replication of a single entity provides the following gain:

Let R_{OB}, be the total optimized backup size, while we replicate a single subproblem solution, becomes:

$$R_{OB} = X_s * P = n_{c_s} * (n-s)$$

R_{UB}, the total un-optimized backup size, while we replicate all the dependent subproblem solutions:

$$R_{UB} = X_s * P * s = n_{c_s} * (n-s) * s$$

So, overall gain $G = \frac{R_{UB}-R_{OB}}{R_{UB}} * 100\% = \frac{n_{c_s}(n-s)(s-1)}{n_{c_s}(n-s)*s} * 100\% = \frac{s-1}{s} * 100\%$.

As an example, to solve a TSP problem with a set of $N = 20$ cities, start from the city 1, traverse all other 19 cities and end in the city 1. For a certain value of $s = 5$, the overall gain becomes: $G = \frac{4}{5} * 100\% \approx 80\%$. For another scenario, while $s = 13$, the overall gain becomes: $G = \frac{12}{13} * 100\% \approx 92.30\%$. This shows that when $s << N$, the total number of subproblems generated with s is more, and more backup has to be done. On the contrary, as $s \approx N$, fewer subproblems are generated, and we have less backup. This shows that the higher the value of s, the more the gain G is.

- LCS problem: For the LCS problem, where the dependency of a subproblem always remains same and which becomes $\Theta(3)$ in every iteration. Among, these three subproblems, only one is responsible for the optimal return of the current subproblem. Therefore, in such a case, the gain becomes: $G = \frac{3-1}{3} * 100\% \approx$

66.6%.

4.4.2 An analysis of extra message overhead during fault tolerance

As a specific case of Lemma 4.3, in the following, we present an analysis of the extra message overhead for protocol 1 with the help of [37]. Suppose there are N processes $P_1, P_2, ..., P_N$ in the system. Moreover, we consider here fine grain distribution of DP table, where each column is assigned to each process.

- TSP problem: First, we calculate the total number of messages in the original (non-FT) parallel solution. At iteration s, each process sends requests to other s processes and receive replies from them. So, a total number of such request-reply messages at iteration $s = 2 * X_P * s = 2 * n_{c_s} * s$. Considering that s could be any value between 1 and n, the total number of messages for all iteration s becomes

$$\sum_{s=1}^{n} 2 * s * n_{c_s} = (n-1)2^n \tag{4.11}$$

  Now, at each s, every subproblem comminicates with s number of subproblems, which clearly indicates a mesage exchange of $O(n)$. So, all together the total number of mesages is $O(n^2 2^n)$. For the fault tolerance, we use the existing messaging of the non-FT algorithm to exchange FT information among processes, so in such case no other extra messages are required for fault tolernce.

- LCS problem: Referring to figure 4.5a, at every iteration s the dependency distance for a process P_i is fixed and it is always with the immediate previous neighbour P_{i-1}, where $1 \leq i \leq N$. It is shown in the figure that, at every iteration s, the leftmost border process does not send any request message to

any other processes (i.e., for such process, $P_{rcv}(s) = \emptyset$). The other $N-1$ processes send requests and receive replies.

So, a total number of such request-reply messages at iteration $s = 1*0+(N-1)*2 = 2(N-1)$.

Similarly, the rightmost border process does not receive any request message from any other processes (i.e., for such process, $P_{snd}(s) = \emptyset$). On the other hand, the other $N-1$ processes receive requests and send replies. So, a total number of such request-reply messages at iteration $s = 2(N-1)$.

Therefore, the total number of message becomes:

$$\{2(N-1)+2(N-1)\}/2 = 2(N-1) \tag{4.12}$$

A process might send an explicit message to replicate its critical data when either of $P_{rcv}(s)$ or $P_{snd}(s)$ is empty (Steps 1 and 2 in protocol 1). In such case, the total number of explicit messages in the FT application = 2 and the total number of messages in the FT application becomes

$$2(N-1)+2 = 2N \tag{4.13}$$

Comparing equations (4.12) and (4.13), it can be seen that the FT protocol results in extra 2 messages over the original non-FT application to withstand 2 consecutive process failures.

4.4.3 An analysis of timing overhead during fault tolerance and recovery

Here, we present a theoretical comparison of the completion time of our proposed ABFT over CP/R for parallel DP . We borrow some of the arguments in the analysis from [25, 64]. In the following analysis, we assume that there are a maximum N number of parallel processes executing at a particular time. Let the mean time to failure (MTTF) be m time units. Also, assume that coordination occurs every c time units and t_c is the checkpoint time of a process. Let the total execution time of the application without any fault tolerance support be T_o.

In the case of CP/R, the coordination time is proportional to the number of explicit coordination messages, which is, in turn, proportional to the number of co-ordinated processes. Thus the maximum coordination time $T_{cord} = \alpha N$, where α is a constant. The worst case execution time of the application running the protocol without encountering any faults is:

$$T_c = T_o + (T_o/c)(\alpha N + t_c) \tag{4.14}$$

Assume there occur n failures altogether, then the worst case execution time in the presence of faults becomes:

$$T_{cf} = T_c + n(c + \beta N) \tag{4.15}$$

where, βN is the overhead associated with restarting all the N processes, which is the requirement in CP/R, and β is a constant. As, $n = T_{cf}/m$, we have

$$T_{cf} = T_o \frac{1 + (\alpha N + t_c)/c}{1 - (c + \beta N)/m} \tag{4.16}$$

In the case of fault tolerant DP, during each iteration the critical data is saved with the neighbours. Let us assume that t_i time units represents one iteration time. The execution time without encountering any failure is

$$T_p = T_o + (T_o/t_i) \tag{4.17}$$

where, T_o/t_i is the overhead for the critical data which basically is in parallel with the execution time of the original algorithm, giving $T_p \approx T_o$.

If n be the total number of failures altogether, each of which is a single process failure, then the total execution time becomes:

$$T_{pf} = T_p + n(t_i + \delta) \tag{4.18}$$

as $n = \frac{T_{pf}}{m}$ and $T_p \approx T_o$, it becomes

$$\begin{aligned} T_{pf} &= T_o + \frac{T_{pf}}{m}(t_i + \delta) \\ T_{pf}[1 - \frac{1}{m}(t_i + \delta)] &= T_o \\ T_{pf} = T_o[1 - &\frac{m}{m - (t_i + \delta)}] \\ &= T_o \frac{1}{1 - \frac{t_i + \delta}{m}} \end{aligned} \tag{4.19}$$

as $t_i << m$ and δ, the overhead for critical data, is apparently insignificant, we have $T_{pf} \approx T_o$.

Based on empirical data [7], the restarting overhead β in the case of diskless CP/R is significant; in fact, this overhead is multiplied to restart all the N processes (i.e., βN). On the contrary, in a fault tolerant DP, the restarting overhead δ for the faulty process is insignificant and is proportional to the critical data size of that process, which is in the order of few bytes and is a fraction of βN. So, we have $T_{pf} << T_{cf}$.

4.5 Experimental Evaluation

In this section, we evaluate our proposed FT technique and compare the result with diskless CP/R for two well-known DP types of problem: TSP and LCS. All experiments were conducted on HPC clusters, Colosse, Briarée and Guillimin of Calcul Québec and Compute Canada [1]. Following are the properties of the systems used here:

Table 4.1: Configuration of the Clusters [74]

Name of the Cluster	Number of Nodes	Core per Node	Memory per Node	Network	Operat-ing System	MPI version
Colosse	960	8	24/48 GB	InfiniBand QDR	CentOS 6.6	OpenMPI 1.8.4
Briarée	672	12	24/48/96 GB	InfiniBand QDR	Scientific Linux 6.3	OpenMPI 1.8.3-gcc
Guillimin	1200	12	24/36/72 GB	InfiniBand QDR	CentOS 6.6	OpenMPI 1.8.3-gcc

For our experiment, first we consider a 29 nodes symmetric TSP problem (bays29.tsp, provided by TSPLIB [101]). Next, for LCS problem, we use the publicly available data of gene index database form the computational biology and functional genomics laboratory at the DANA-Faber Cancer Institute and Harvard School of Public Health [36]. In performance comparison with CP/R, checkpoint is taken after execution of every 50 rows of the DP table.

4.5.1 Experimental results of TSP

Figure 4.8 shows the recovery overhead (%) for our scheme to recover from a single process failure. The recovery overhead is calculated as the ratio of the total recovery time to the total execution time of the recovered application that uses our scheme. It is found that the recovery overhead varies from .005% to .018% with increasing number of processes. The result displays that the recovery cost of a single process

[1]The operation of this supercomputer is funded by the Canada Foundation for Innovation (CFI), NanoQuébec, RMGA and the Fonds de recherche du Québec - Nature et technologies (FQRNT)

failure is meager as the lost data on the failed process is recovered with the help of the peer process where the backup data of the failed process is replicated in peer process's memory. This cost accounts for a small portion of the total execution time of the TSP problem.

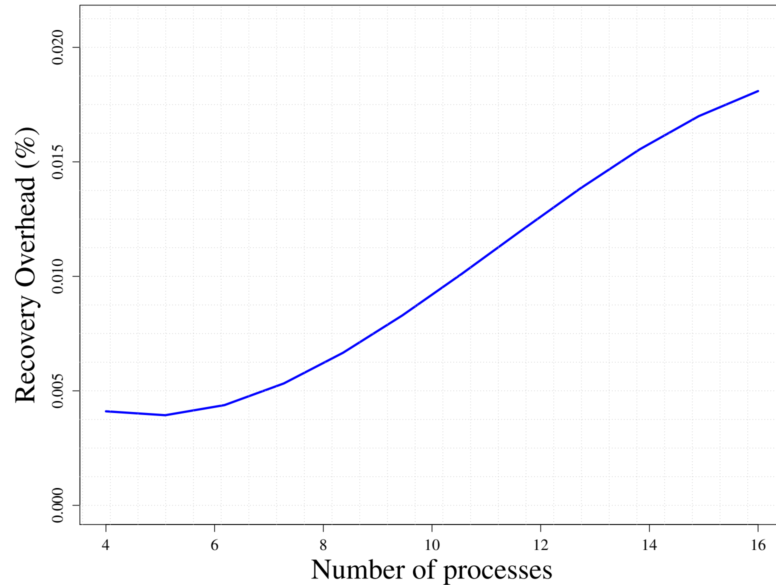


Figure 4.8: Recovery Overhead of Proposed ABFT with a Single Failure

Figure 4.9 reports the overhead of fault tolerance for both ABFT (our proposed scheme) and checkpointing when no failure happens during the execution of TSP algorithm. The result demonstrates that our design introduces very insignificant fault tolerance overhead. The reason is that our proposed ABFT uses the existing messaging of the non-FT implementation to exchange fault recovery data with peer processes, which, in turn, do not add extra fault tolerance time over the execution time of the algorithm. On the other hand, with checkpointing, the overhead increases at a much higher rate with increasing scale as compared to our scheme. This can be attributed to the fact that checkpointing involve coordination cost and explicit messages, and these overheads do not scale well with increasing number of processes.

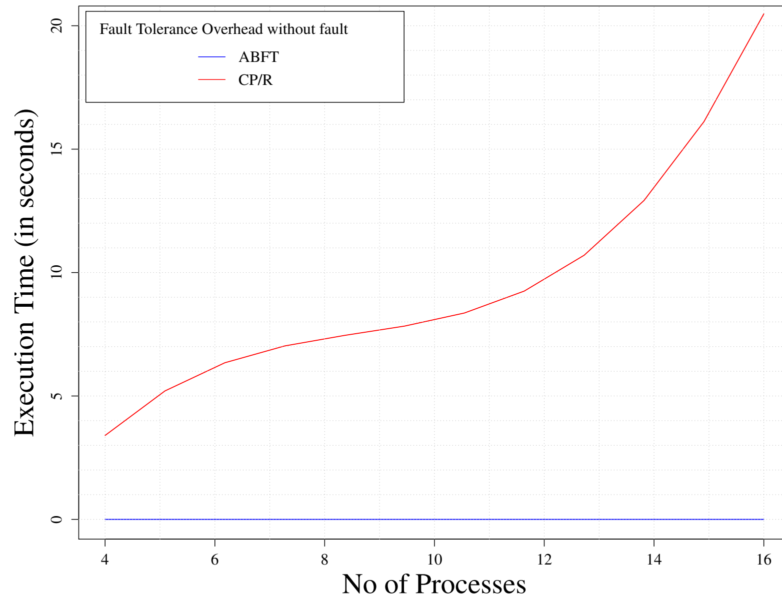


Figure 4.9: Fault Tolerance Overhead without fault (ABFT vs. CP/R)

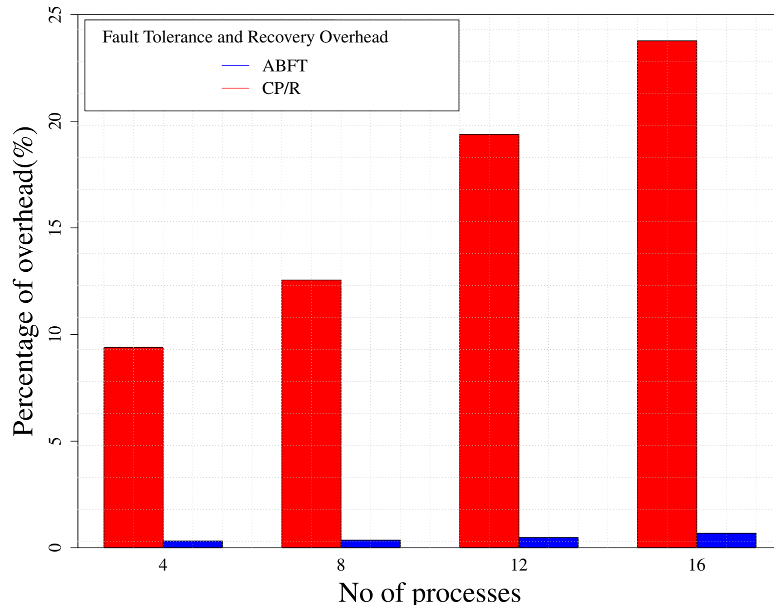


Figure 4.10: Fault Tolerance and Recovery Overhead With Multiple Simultaneous Failure (ABFT vs. CP/R)

Figure 4.10 shows that our scheme has less fault tolerance and recovery overhead

as compared to CP/R in the presence of 60% simultaneous process failure. The overhead in our scheme is almost uniform irrespective of the number of processes and is less than 1% for each case. On the other hand, checkpointing introduces significant overhead which increases with increasing scale and it is from 9% to 23% for a range of 4 to 12 processes. The reason behind is the recovery procedure of CP/R forces all the active and failed processes to roll back and restart from a previous consistent state with failures. Whereas, in our scheme, recovery of all the failed processes from a previous recoverable point happens concurrently without rolling back the other active processes.

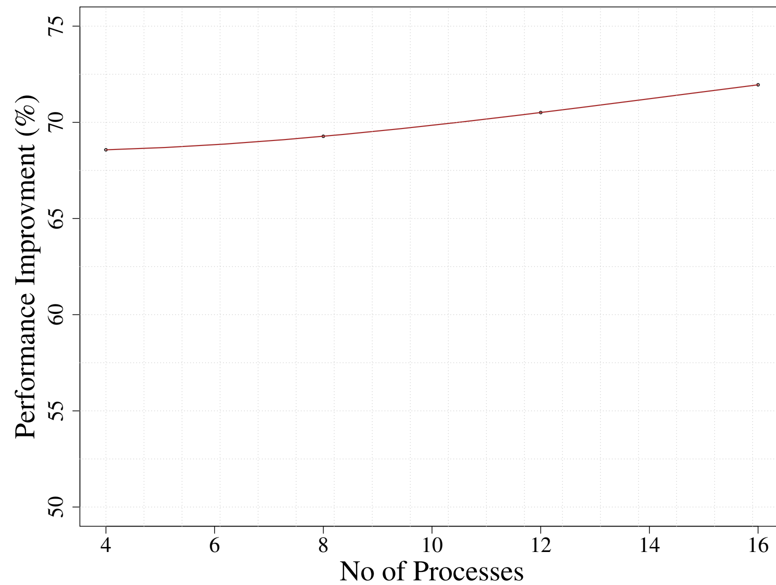


Figure 4.11: Performance Improvement (Our proposed ABFT vs. CP/R)

Figure 4.11 shows the performance improvement of our scheme over diskless checkpointing in the case of single process failure. It is found that the performance gain of our scheme over CP/R is approximate 70%. The reason is our proposed ABFT method is free from coordination and explicit message cost that is central design point of CP/R.

The total gain in critical data is given in table 4.2. It indicates a significant gain in

critical data, which is estimated 47% irrespective of the number of processes, which supports our theoretical finding. Replication of critical data rather than all the dependent data for a subproblem justifies the result.

Table 4.2: Gain in Critical data

No. of Processes	Total Backup Size (nonCritical)	Total critical data size	Gain(%)
4	227595	119598	47.45%
8	196557	104514	46.82%
12	243114	125472	48.39%
16	171028	93431	45.37%

4.5.2 Experimental results of LCS

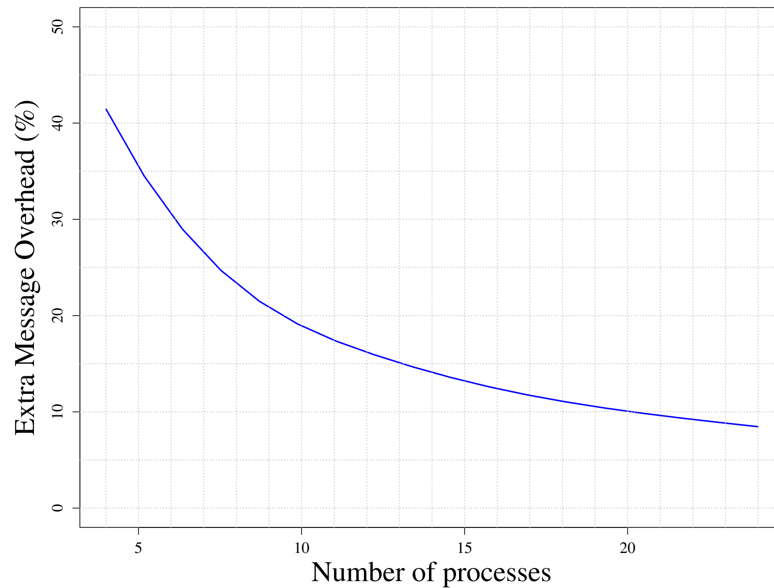


Figure 4.12: Extra Message Overhead for ABFT

One might think it reasonable to speculate that the total number of backup messages exchanged among the process during fault tolerance increases with the number of processes, but practically it remains unchanged. We present this in figure 4.12, where it shows that the extra backup message overhead decreases with increasing

number of processes. The reason behind: the more the processes involved in the computation, the more the messages exchanged among themselves in a non-FT implementation. The overhead declines from 42% to 18% for a total number of processes from 4 to 24.

Table 4.3: Gain in Critical Data Size for LCS problem

No. of Processes	Checkpoint Data	Critical Data in ABFT	Gain(%)
4	16248576	2902800	82.13%
8	5923584	1161120	80.39%
12	8885376	1915848	78.44%
16	11847168	2322240	80.39%
12	14808960	2801050	81.09%
16	17770752	3309192	81.38%

Table 4.3 evaluates the total gain in size of the critical data as compared to checkpoint data. We have already explained in 4.3.1, what will be the critical data for an LCS problem and in our ABFT implementation, we replicate this data with neighbor processes. In the case of CP/R implementation, it backups all the data of the DP table blindly without considering any algorithmic feature of the algorithm, whereas, in ABFT implementation, algorithmic characteristics are acknowledged to select critical data, as describes in 4.3. Here to mention, in our ABFT implementation, each process backup only those data which are modified from the previous iteration. It is evident from table 4.3 that a substantial amount of gain is achieved in critical data size.

Figure 4.13 presents the fault tolerance overhead for both CP/R and our proposed ABFT for LCS problem. We explain it with three different scenarios: 30%, 50%, and 60% process failures. We experience an enormous amount of checkpoint overhead with increasing number of processes in case of CP/R implementation, whereas our proposed ABFT introduces an insignificant amount of overhead as compared to CP/R. The fact is the coordination cost, and the explicit message cost of CP/R causes this huge overhead and which in turn supports the claim that CP/R does not scale

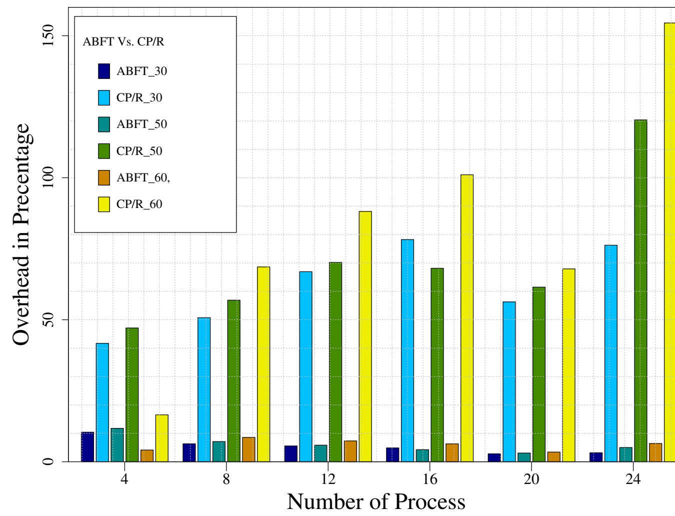


Figure 4.13: Fault Tolerance Overhead (ABFT vs. CP/R)

well with growing number of processes.

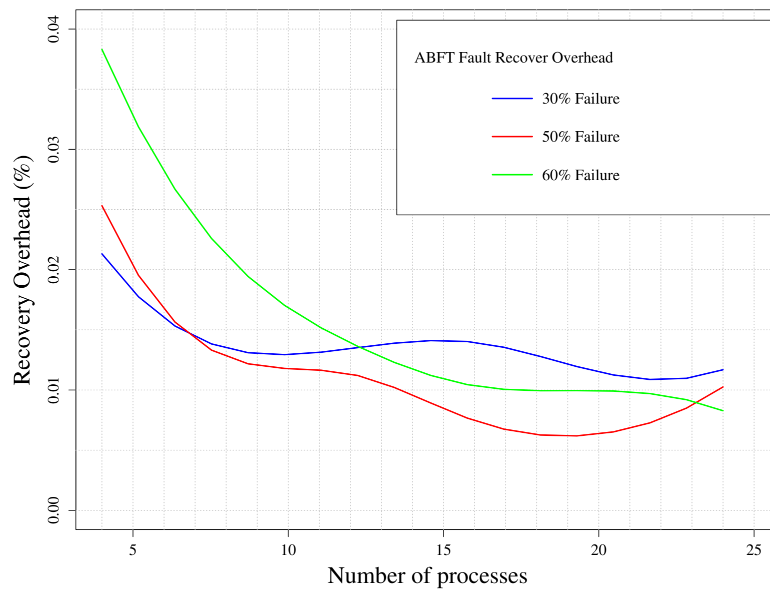


Figure 4.14: Recovery Overhead with Multiple Simultaneous Failures

Figure 4.14 shows the recovery overhead of our proposed ABFT with different numbers of multiple simultaneous process failures. We consider here three failure scenario:

30%, 50%, and 60% simultaneous process failures. The recovery overhead is approximately within a range of 0.021% to 0.012% for 30% process failure, 0.025% to 0.01% for 50% failures and 0.037% to 0.009% for 60% failures. We have found here that for all the failure scenario, the recovery overhead is nominal, which is negligible in practice. The reason behind is all the recovery occurs in parallel. Figure 4.14 shows that, even with the recovery cost included, the total overhead of our proposed ABFT is still very low, and it decreases as the number of processes increases.

4.6 Conclusion

In this chapter, we developed an algorithm based fault tolerance and recovery protocol for parallel dynamic programming algorithms by analyzing the algorithmic and communication characteristics. The algorithmic characteristics determine what application data need to be saved to recover from the fault(s), and the communication characteristics determine how this data should be saved to minimize the cost. Furthermore, we show that for such parallel applications, communication dependency among the processes plays a significant role in the design of fault tolerance strategies with minimum extra message overhead. We evaluated our protocol with two well-known DP application: (i) longest common subsequence (LCS) problem, which is a non-serial DP problem with a fixed data-dependence with every stage and which is $O(3)$ and, (ii) travelling salesman problem (TSP), which is a serial DP problem with a monotonically increasing data-dependence with stage and which is $O(n)$. Experimental results demonstrate that for both the cases the proposed method performs better than checkpointing regarding fault-tolerance and fault-recovery overhead.

Chapter 5

Optimization in Fault Tolerance

In any fault tolerance strategy, optimizations are possible in both time and space regarding how often to replicate, what data to replicate, and when to prune the replicated data. Regarding how often to replicate, it depends on what is the Mean Time to Failure (MTTF). Pruning depends on how much and how long the information is to be saved to tolerate failures; it depends on how many maximum iterations to roll back in a worse case recovery scenario. Optimizations in "what data to replicate" depends on the algorithmic characteristics of an application. In that regard, in this chapter, we discuss it below for the DP solution to the 0/1 knapsack problem. Numerical results demonstrate that our proposed FT scheme is highly efficient.

5.1 Motivation

Minimal storage and recovery cost are the crucial aspect in fault tolerance and recovery design. The optimal size of critical data is one of the main concern to minimize the overhead as mentioned above. In that regard, we demonstrate that for certain DP class of problems, additional optimization in critical data is possible. This is only feasible when each subproblem has a constant lower degree of dependency cardinality

with other subproblems. 0/1 knapsack is a unique example of such problem where each subproblem has a dependency cardinality of 2 which remains unchanged with increasing number of iteration. This feature motivates that instead of replicating the critical data of a process with its neighbour, replication of a bit-vector flag is sufficient for recovery. This flag is used to rebuild the lost data of the failed process with the help of the information available in the other processes.

5.2 0/1 Knapsack Problem

The 0/1 knapsack problem is described as follows: Given a set of n distinct elements and a knapsack with maximum capacity C. Each ith element has some unique weight w_i and profit v_i respectively, where $w_i, v_i, C \in \mathbb{Z}^+$. The goal is to choose a subset of the n elements to fill the knapsack in such a way that the total profit is maximized without having the total weight sum to exceed the capacity C. Let, $X = [X_1, X_2, ..., X_n]$ be a solution set of the elements in which $X_i = 0$ if the element is not in the knapsack and otherwise it is 1. Mathematically, the problem can be formulated as follows:

$$\max \sum_{i=1}^{N} v_i X_i \tag{5.1}$$

subject to

$$\sum_{i=1}^{N} w_i X_i \leq C \tag{5.2}$$

DP formulation for this problem is :

$$f[i,m] = \begin{cases} 0 & m \geq 0, i = 0 \\ \infty & m < 0, i = 0 \\ \max\{f[i-1,m], & \\ (f[i-1, m-w_i] + v_i)\} & 1 \leq \text{i} \leq \text{n} \end{cases} \tag{5.3}$$

where, $f[i, m]$ is the maximum profit for a knapsack with a capacity of m using only items $\{1, 2, 3, \cdots, i\}$. $f[n, C]$ be the maximum profit for a knapsack with a capacity of C using only items $\{1, 2, 3, \cdots, n\}$.

Let us illustrate the optimization and composition functions from the DP formulation of the 0/1 knapsack problem. Here the function f represents the overall profit of including an item i of weight w_i and profit v_i into the knapsack of current capacity m and is given by the formula:

$$f(i, m) = \max(f(i-1, m), f(i-1, m-w_i) + v_i), 1 \leq i \leq N \tag{5.4}$$

As compared with the general formulation of dynamic programming (equations 4.1 and 4.2 from chapter 4), *max* becomes the optimization function Φ and $\{f(i-1, m-wi) + v_i\}$ becomes the composition function in equation (5.4). The goal is to maximize the profit $f(N, C)$ where N is the total number of items and C is the maximum capacity of the knapsack.

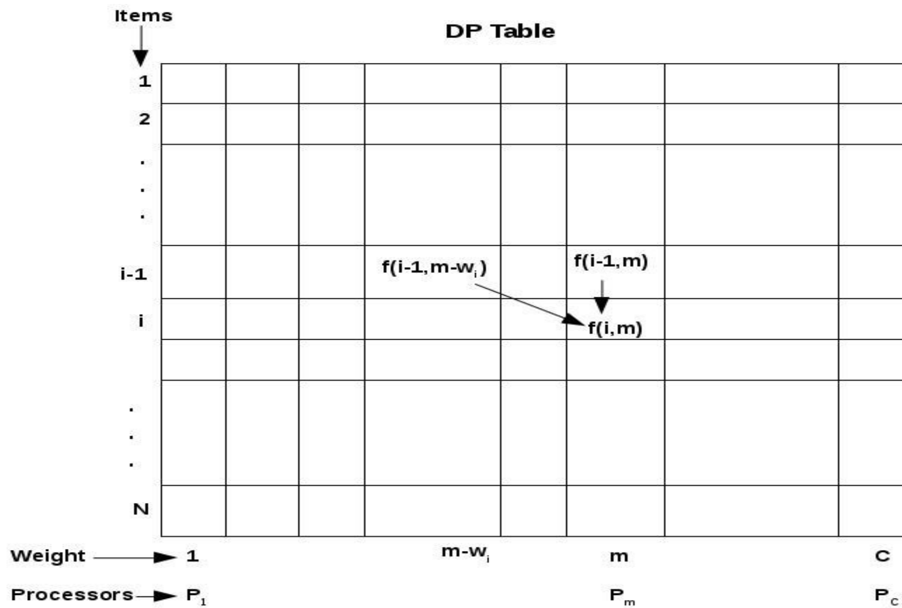


Figure 5.1: Data dependency and the DP table in a parallel solution to the 0/1 Knapsack problem

In a fine-grain parallel solution to the 0/1 knapsack problem, each column is mapped to a single process. So for a knapsack with maximum capacity of C, there are altogether C processes, $P_1, P_2, .., P_C$ in a fine-grain distribution. During each iteration i, to calculate equation (5.4), process P_m depends on a value from itself and also the value from another process P_{m-w_i} provided $m - w_i \geq 1$ from previous iteration $(i-1)$. As shown in figure 5.1, each process owns a column of the DP table. The profit p_i and the weight w_i are parts of the global knowledge, available to all the processes. On the other hand, in a coarse-grained solution, multiple columns are assigned to each process. The fine-grain parallel DP algorithm to solve 0/1 knapsack problem is given in algorithm 5.1.

Algorithm 5.1 Parallel Dynamic Programming Algorithm for 0/1 Knapsack Problem

1: Initialize the DP table f by $f[i,0] = f[0,m] = 0$ where $i = 0, 1, ..., N$ and $m = 0, 1, ..., (C-1)$ ▷ Initialization Phase
2: Partition f into C columns and assign to p processors where $p = C$ ▷ Partition Phase
3: For each processor p_m, at each iteration i, during computation of $f[i,m]$
4: **if** $(m - w_i \geq 1)$ && $(m + w_i \leq C)$ **then**
5: Send data request to P_{m-w_i}
6: Receive data from P_{m-w_i}
7: Receive data request from P_{m+w_i}
8: Send data to P_{m+w_i}
9: Compute $f[i,m] = max\{f[i-1,m], (f[i-1,m-w_i] + v_i)\}$
10: **else**
11: Compute $f[i,m] = f[i-1,m]$
12: **end if**

5.3 Optimization in Fault Tolerance

From the DP example of 0/1 knapsack problem (equation 5.4 and figure 5.1), the critical data for a failed process P_m to recover it from the start-execution point of $f(i,m)$ during iteration i is either (i) $\{f(i-1,m)$ or (ii) $f(i-1,m-w_i)\}$. The first component comes from its own local state and the second component comes from the

local state of process P_{m-w_i} at the end of the $(i-1)$th iteration.

Low storage and recovery overhead are the critical issues in designing fault tolerance. In that regard, instead of replicating CD_{Local} with the processes in P_R and P_S, we can replicate a bit-vector flag for recovery purpose. This flag is used to rebuild the lost data of the failed process with the help of the information available in the other processes. The width of the bit vector depends on the granularity of the solution: if one column of the DP table is mapped per process as in a fine-grained solution then it is a single bit flag; if multiple columns are mapped then the bit vector comprises of one bit per column (figure 5.1).

Let us discuss the fine-grained solution where each process keeps the single-bit flag as part of its local state. Let us call this flag bit for process P_m as f_m. Referring to equation 5.4, at the end of iteration i, if $f(i, m)$ is the same as $f(i-1, m)$, then the flag bit f_m for process P_m is set to 0, otherwise it is 1. For the 0/1 knapsack problem, the flag bit being 0 at the end of iteration i implies that only CD_{Local} contributed to computing $f(i, m)$ and CD_{Remote} had no contribution and vice verse. The single flag bit can be extended to the multiple flag bits (in a bit-vector) in a coarse-grained solution.

The benefits of using the flag bit(s) are not only in reducing message overhead but also in facilitating the recovery of lost data (e.g., a column of the DP table) of a failed process during recovery. This is discussed in the following for the specific case of the 0/1 knapsack problem.

Protocol 1.1: FT protocol for the (fine-grained) parallel DP solution to the 0/1 Knapsack problem

Assumptions: (1) Referring to figure 5.1, the *Sender_Set* and *Receiver_Set* (see chapter 4) for process P_m are denoted as P_{mS} and P_{mR} respectively where $P_{mS}(i) = \{P_{m+w_i}\}$ if $m + w_i \leq C$, otherwise $P_{mS}(i) = \emptyset$; and $P_{mR}(i) = \{P_{m-w_i}\}$ if $m - w_i \geq 1$,

otherwise $P_{mR}(i) = \emptyset$. Note that w_i is the weight of the ith item and C is the maximum knapsack capacity, which is also equal to the total number of processes. (2) The flag-bit for process P_m corresponding to its local solution to subproblem at iteration $(i-1)$ is 2-way replicated among the processes in $P_k(i) = P_{mR}(i) \cup P_{mS} \cup P_E(i)$, where $P_E(i) = \emptyset$ if both $P_{mR}(i) \neq \emptyset$ and $P_{mS}(i) \neq \emptyset$. Wraparound is used to handle the special cases when either one or both of the *Sender_Set* and *Receiver_Set* are empty depending on if the process is a boundary process (with respect to the DP table column mapped to it), and in those cases explicit messages are needed (i.e. $P_E(i) \neq \emptyset$). Considering both boundary and non-boundary cases, the set $P_k(i)$ for a process P_m can be generalized as: $P_k(i) = \{P_{1+(m+w_i-1)\%C}, P_{1+(C+m-w_i-1)\%C}\}$.

The protocol:

At start of iteration i, the flag-bit for process P_m corresponding to its local solution to subproblem at iteration $(i-1)$ is 2-way replicated among the processes in $P_k(i)$ according to the following:

Step 1: if $P_{mR}(i) \neq \emptyset$ then piggyback the flag bit together with the original request message to P_{m-w_i}; otherwise send the flag bit in an explicit message to $P_{1+(C+m-w_i-1)\%C}$.

Step 2: if $P_{mS}(i) \neq \emptyset$ then send the flag bit together with $f(i-1, m)$, as part of the original non-FT algorithm, to P_{m+w_i}; otherwise send the flag bit together with $f(i-1, m)$ in an explicit message to $P_{1+(m+w_i-1)\%C}$.

End of protocol 1.1

Algorithm 5.2 describes fault tolerant algorithm.

Algorithm 5.2 Fault Tolerant Parallel Dynamic Programming Algorithm for 0/1 Knapsack Problem

1: Initialize the DP table f by $f[i,0] = f[0,m] = 0$ where $i = 0,1,...,N$ and $m = 0,1,...,(C-1)$
2: Partition f into C columns and assign to p processes where $p = C$
3: For any process p_m, in an iteration i, during computation of $f[i,m]$
4: **if** $(m - w_i \geq 1)$ **then**
5:     **if** $(m + w_i \leq C)$ **then**
6:         Send data request and flag-bit to process, $P_{(m-w_i)}$
7:         Receive reply with (i) requesting data , (ii) flag-bit from $P_{(m-w_i)}$
8:         Receive data request and flag-bit from P_{m+w_i}
9:         Send reply with (i) requested data , (ii) flag-bit to P_{m+w_i}
10:     **else**
11:         Send data request and flag-bit to process, $P_{(m-w_i)}$
12:         Receive reply with (i) requesting data , (ii) flag-bit from $P_{(m-w_i)}$
13:         Send flag-bit to $P_{\{1+(m+w_1-1)\%C\}}$
14:     **end if**
15: **else**
16:     Send flag-bit to $P_{\{1+(C+m-w_1-1)\%C\}}$
17:     Receive data request and flag-bit from P_{m+w_i}
18:     Send reply with (i) requested data , (ii) flag-bit to P_{m+w_i}
19:     Compute $f[i,m] = \max\{(f[(i-1,m]),(f[i-1,m-w_i]+v_i)\}$
20: **end if**
21: Compute $f[i,m] = f[i-1,m]$

5.3.1 An example of Fault tolerance

Figure 5.2a presents the initial state of an arbitrary dynamic programming table with a set of 6 processes $N = \{P_0, P_1, P_2, P_3, P_4, P_5\}$ to solve 0/1 knapsack problem with a total of 4 elements with weight $w_i = \{1, 2, 3, 4\}$ and profit $v_i = \{3, 7, 2, 9\}$ for a knapsack with capacity $C = 5$. Table 5.1 shows the dependency among the processes to compute each cell during iteration 1 in non-FT implementation.

Table 5.1: Data Dependency Among the Processes

Iteration	**Dependency**
1	$P_2 \to P_0, P_3 \to P_1, P_4 \to P_2, P_5 \to P_3$

Following describes how fault tolerance is accomplished alongside with the execution of the algorithm:

- Processes P_0 and P_1 depends on themselves for the computation. Illustrate in figure 5.2b.

- P_2 piggybacks the flag-bit (FT algorithm) with the data request message (part of original non-FT algorithm) to P_0. Next, P_0 replies with the flag-bit together with the requested data (part of original non-FT algorithm), shown in figure 5.2c.

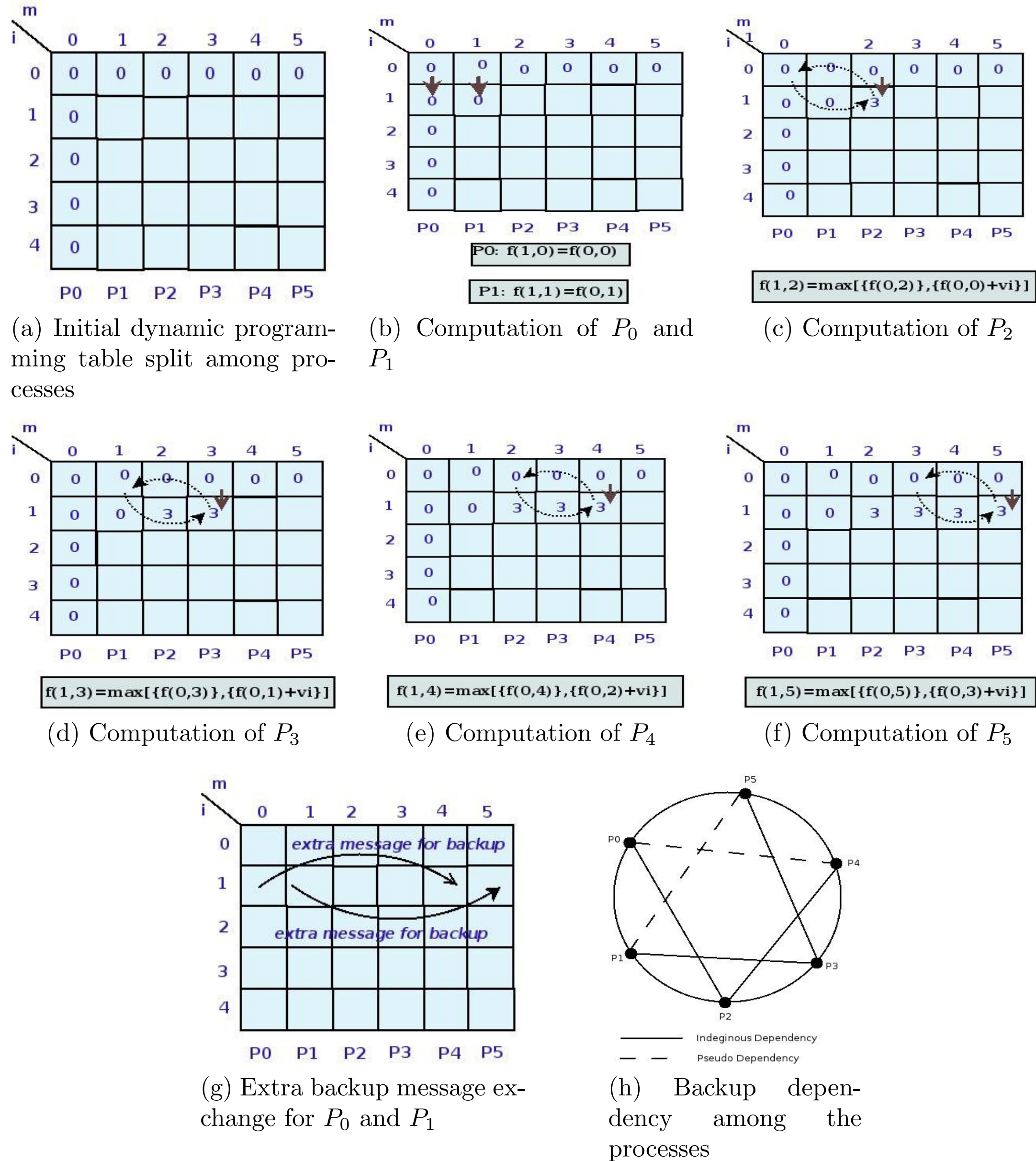
- Similarly, P_3 exchanges with P_1, P_4 with P_2 and, P_5 with P_3 respectively, shown in figure 5.2d- 5.2f.

- P_0 and P_4 will keep backup of P_2 in their local memory. Similarly, P_1 and P_5 will keep the backup of P_3.

- The left border processes P_0 and P_1 will send an extra backup message to the right border processes P_4 and P_5 respectively and vice-verse. Other copy will be with the processes for which they are contributing to their computation or

takes contribution for their computation. For P_0 and P_1, it will be P_2 and P_3 respectively and for P_4 and P_5, it will be P_2 and P_3 respectively, display figure 5.2g.

- The logical backup dependency among the process is shown in figure 5.2h.



(a) Initial dynamic programming table split among processes

(b) Computation of P_0 and P_1

(c) Computation of P_2

(d) Computation of P_3

(e) Computation of P_4

(f) Computation of P_5

(g) Extra backup message exchange for P_0 and P_1

(h) Backup dependency among the processes

Figure 5.2: Fautlt Tolerance for 0/1 Knapsack Problem

5.4 Fault Recovery

We discuss the recovery protocol for a fine grained DP solution to the 0/1 knapsack problem, where each process is assigned a single column of the DP table. We formulate the recovery equation from equation (5.4) by including the flag bit f_m used in protocol 1.1 as follows:

$$f(i,m) = f(i-1,m).\overline{f_m} + \{f(i-1,m-w_i) + p_i\}.f_m \tag{5.5}$$

Here, $f(i,m)$ is the lost data of the failed process P_m from iteration i that needs to be recovered. It can be seen from euqation 5.5 that the flag bit f_m determines whether the local or remote part of the critical data for the process at iteration i contributed to calculating $f(i,m)$.

Referring to Protocol 1.1, suppose process $P \in P_{mS}(i+1)$ detects failure of P_m at start of iteration $i+1$. The recovery protocol resumes the execution of the failed process by recovering the lost data of P_m from iteration i in the following way:

Restart execution of process P_m:

Case 1: The flag bit f_m from iteration i can be recovered (from $P \in P_{mR}(i+1)$).

Case 1.1: If $f_m = 1$ then, referring to Eqn. (5.5), the lost data is recovered using $CD_{Remote}(P_m, i)$ that belongs the *Receiver_Set* $P_mR(i)$ and resume the operation of the failed process from iteration $i+1$.

Case 1.2: If $f_m = 0$ then, referring to Eqn. (5.5), the lost data can be recovered with the help of $CD_{Local}(P_m, i)$. However $CD_{Local}(P_m, i)$ belongs to the failed process P_m and is lost because it could not be received by $P_{mS}(i+1)$ (that is why the failure is detected). So this data needs to be regenerated from a previous iteration where f_m was last set to 1; let this be at iteration $(i-j)$. The following two cases can arise:

Case 1.2.1: If the generation of the lost data at iteration $(i-j)$ is successful with the help of $P_k(i-j)$ (protocol 1.1) then resume the operation of P_m from iteration

$i+1$. Note that the data is unchanged from iteration $(i-j)$ till iteration i because the flag bit had been 0 during this interval.

Case 1.2.2: If not successful (due to failure of some of the processes in $P_k(i-j)$, then backtrack to a previous iteration from where the lost data can be recovered, recalculate the lost data till iteration i, and resume operation from iteration $i+1$.

Case 2: The flag bit f_m from iteration i cannot be recovered. In that case, backtrack to a previous iteration $i-j$, $1 \le j \le (i-1)$, for which the flag bit can be recovered and then follow the same steps as in Case 1.1 or Case 1.2 above to recover the failed data from iteration $i-j$ onward till the current iteration $i+1$.

5.5 Theoretical Analysis

5.5.1 An analysis of extra message overhead

As a specific case of Lemma 4.3, in the following we present an analysis of the extra message overhead for protocol 1.1. Suppose there are N processes $P_1, P_2, ..., P_N$ in the system. Referring to figure 5.1, at the start of iteration i the dependency distance is w_i where $1 \le w_i \le N-1$.

First we calculate the average number of messages in the original (non-FT) parallel solution. At iteration i, processes $P_1, P_2, ..., P_{w_i}$ do not send request messages (i.e., for each of those processes, $P_R(i) = \emptyset$). The other $N - w_i$ processes send requests and receive replies. So, a total number of such request-reply messages at iteration $i = w_i * 0 + (N - w_i) * 2 = 2(N - w_i)$. Similarly, processes $P_N, P_{N-1}, ..., P_{N-w_i+1}$ do not receive request messages (i.e., for each of those processes, $P_S(i) = \emptyset$) but the other $N - w_i$ processes receive requests and send replies. So, total number of such request-reply messages at iteration i becomes $2(N - wi)$. Noting that each message

is counted twice in the previous calculations, so the total number of messages at iteration i:

$$\{2(N - w_i) + 2(N - w_i)\}/2 = 2(N - w_i) \tag{5.6}$$

Considering that w_i could be any value between 1 and $N-1$, the average number of messages at iteration i becomes

$$\frac{1}{N-1}\sum_{w_i=1}^{N-1} 2 * (N - w_i) = \frac{2}{N-1} * \frac{N(N-1)}{2} = N \tag{5.7}$$

Now, referring to protocol 1.1, in the FT application, each process' CD information replicated among the processes in the set $P_k(i)$ and involve explicit messages when either of $P_R(i)$ or $P_S(i)$ is empty (Steps 1 and 2 in protocol 1.1). Based on the previous discussion, total number of such processes for which either $P_R(i)$ or $P_S(i)$ is empty $= 2w_i$. Each of these processes sends an explicit message to save its CD information with a member of $P_k(i)$. Hence, the total number of explicit messages in the FT application $= 2w_i$ and the total number of messages in the FT application becomes

$$2(N - w_i) + 2w_i = 2N \tag{5.8}$$

Comparing equations (5.7) and (5.8), it can be seen that in the average case scenario the FT protocol results in extra N number of messages over the original non-FT application.

The best case scenario in term of extra message overhead arises when $w_i = 1$. From equation (5.6), we can see that the total number of messages at iteration i in the best case scenario $= 2(N-1)$. Compared with the FT implementation, it is evident that the FT protocol results in two extra messages over the original non-FT application in the best case.

The worst case scenario arises when $w_i = N-1$. From equation (5.6), the total

Table 5.2: Experimental configuration

No. of Processes	Knapsack with capacity	Total number of items	Weight Range	Profit Range
12 to 48	100000	1000	10 to 5000	20 to 1000

number of messages at iteration i in the worst case scenario = 2. Comparing with the FT implementation, it can be seen that in the worst case scenario the FT protocol results in 2(N -1) extra messages over the original non-FT application.

5.6 Experimental Evaluation

In this section, we experimentally analyze the proposed ABFT scheme and compare its performance with diskless checkpointing [73]. All the experiments are performed on the HPC cluster Briarée of Calcul Québec and Compute Canada[1]. Briarée has a total of 8064 cores which are distributed among 672 compute nodes and connected via a 4 TB/s Infiniband QDR network. The operating system is Scientific Linux 6.3. and we use OpenMPI 1.8.3 for our implementation.

For our experiments, we consider the DP solution to the 0/1 knapsack problem with a variable configuration as listed in Table 5.2. Profits and weights of items are generated randomly and fault generation is also random. In performance comparison with diskless checkpointing, (diskless) checkpoint is taken after execution of every 50 rows of DP table which is much lower than the MTTF for this specific set of experiments.

Table 5.3 shows the extra message overhead of an FT implementation of the 0/1 knapsack problem using protocol 1.1, based on the configuration in Table 5.2, over a non-FT implementation in a failure-free execution. It can be seen from table 5.3 that the overhead increases linearly with the number of processes and also the number

[1]The operation of this supercomputer is funded by the Canada Foundation for Innovation (CFI), NanoQuébec, RMGA and the Fonds de recherche du Québec - Nature et technologies (FQRNT)

Table 5.3: Extra message overhead

No. of Processes	Total no. of messages (nonFT)	Total number of extra messages	Overhead(%)
12	55303	2390	4.32%
24	110789	5492	4.96%
36	140210	7136	5.09%
48	153110	9364	6.12%

of message exchanges scales up linearly with the number of processes. It should be mentioned here that the total execution time of an FT application in our experiments vary from 204 to 162 seconds as the number of processes varies from 12 to 48.

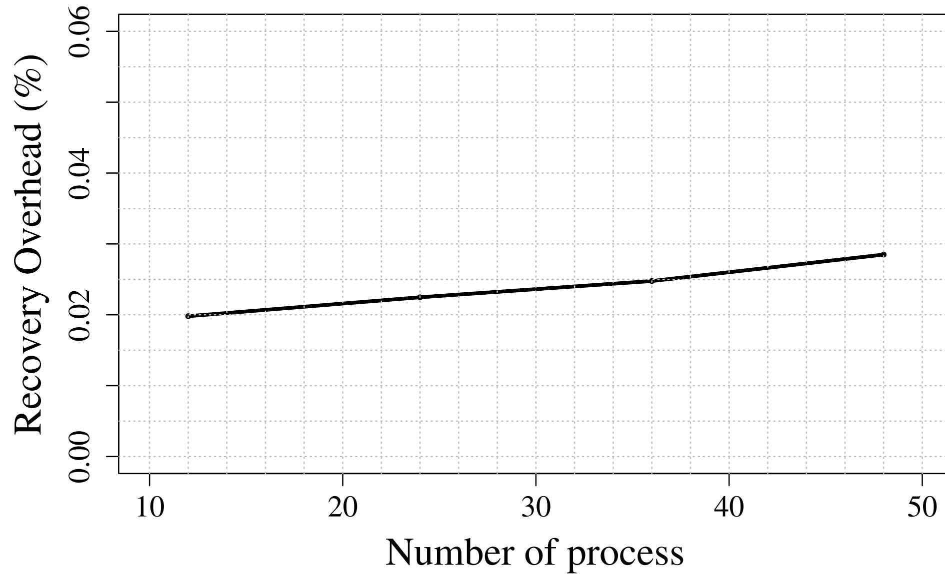


Figure 5.3: Recovery Overhead with single failure.

Figure 5.3 and figure 5.4 show the recovery overhead for our scheme with single and simultaneous multiple failures (up to 50% of the process failures) successively. The recovery overhead is calculated as the ratio of the total recovery time to the total execution time of the recovered application that uses our scheme. The recovery overhead is approximately .025% in both the cases. We found that there is not much of change in the recovery overhead between single and multiple process failures: this can be attributed to the fact that the recovery scheme is distributed, and recovery

of multiple processes can overlap as long as long as they are not dependent on one another.

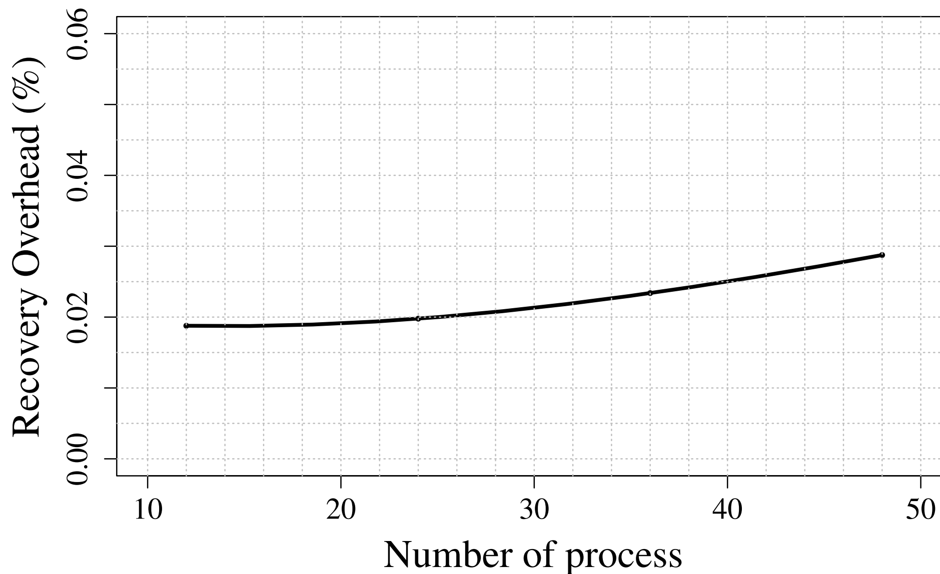


Figure 5.4: Recovery Overhead with simultaneous multiple failures.

Figure 5.5 shows the fault tolerance overhead in a failure-free execution for both diskless checkpointing and our ABFT scheme. Diskless checkpointing is implemented by taking checkpoints after every 50-row operations of the DP table. The result demonstrates that our scheme introduces less fault tolerance overhead as compared to the diskless checkpointing implementation. Also, with diskless checkpointing, the overhead increases at a much higher rate with increasing scale as compared to our scheme. This can be attributed to the fact that diskless checkpointing involve coordination cost and explicit messages, and these overheads do not scale well with increasing number of processes.

Figure 5.6 shows the performance improvement of our scheme over diskless checkpointing in the case of single process failure. It is found that the performance gain of our scheme over diskless checkpointing is nearly 80%. The reason is again due

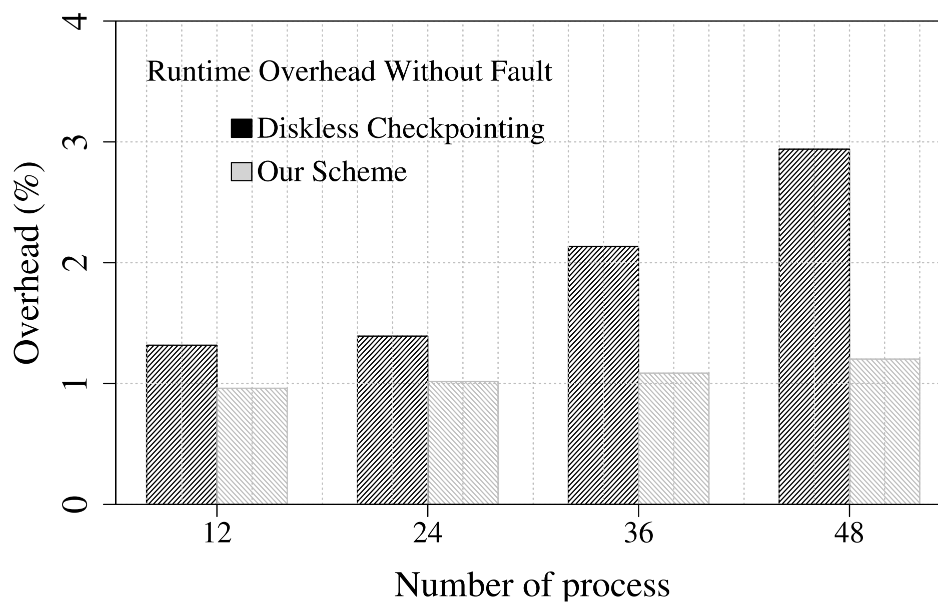


Figure 5.5: Fault Tolerance Overhead (without fault)

to the coordination cost and explicit message overhead, which are either avoided or minimized in our case.

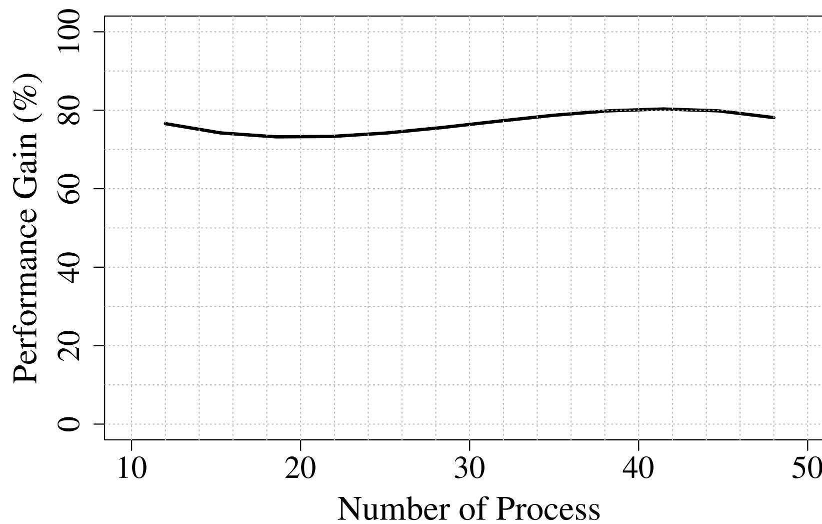


Figure 5.6: Performance Improvement (Our scheme vs. Diskless checkpointing)

Figure 5.7 shows that our scheme has less recovery overhead as compared to diskless checkpointing. The y-axis of the figure is drawn in the logarithmic scale. The

overhead in our scheme is almost uniform irrespective of the number of processes. On the other hand, diskless checkpointing introduces significant overhead which increases with increasing scale. The reason is that, with diskless checkpointing, all the processes are rolled back and recovered from a previous coordinated cut even in the case of a single process failure. Whereas, our scheme only recovers the failed process(es) from a previous recoverable point without rolling back any other processes.

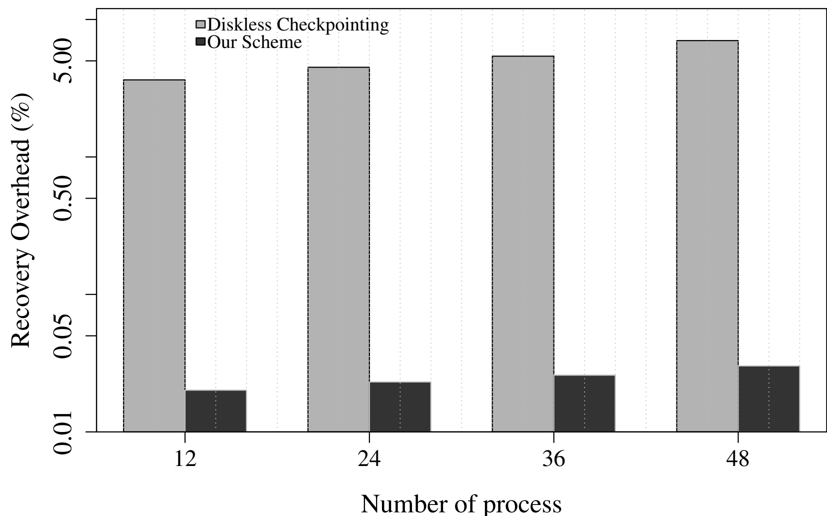


Figure 5.7: Failure Recovery Overhead

5.7 Conclusion

In this chapter, we have presented an ABFT for 0/1 knapsack problem by analyzing it's algorithmic and communication characteristics. The analysis finds more optimization is possible in critical data for such problems where the subproblems have small and fixed dependency cardinality among themselves. Here, we introduce a bit-vector flag as critical data and devise a fault recovery formulation for 0/1 knapsack problem. Experimental results have shown the effectiveness of the proposed FT scheme and indicate that our scheme has introduced much less overhead than CP/R.

Chapter 6

Discussion and Future Work

Fault tolerance is becoming a necessity rather than an accessory for HPC applications. The complex parallel architecture of today's HPC systems introduces more failures with increasing scale. Checkpoint provides an application-transparent fault tolerance to these systems. Minimum checkpoint and recovery overhead is the principal objective of any fault tolerance (FT) design. In that case, the checkpoint and recovery cost imposed by checkpoint/restart (CP/R) is a crucial performance issue for such large-scale systems. In this thesis, we address the several challenging issues of the fault tolerance scheme and develop FT techniques for two broad classes of parallel algorithms with a notion to achieve the objectives of FT design.

In this thesis, we go further than conventional ABFT and concentrate on the different categories of parallel applications not addressed by conventional ABFT. This research studies characteristic of two distinct classes of parallel algorithms: (i) parallel search algorithm and (ii) parallel dynamic programming algorithm and develops ABFT for these two types of algorithms to provide fault tolerance and recovery in case of fail-stop failures. In particular, we bring out the communication and algorithmic characteristics of parallel algorithms to determine what application data needs to be replicated and how this fault recovery data should be replicated to design efficient

fault tolerance and recovery strategies for a class of parallel applications. Theoretical and experimental results have successfully demonstrated that our developed fault tolerance schemes can endure multiple simultaneous process failures with low checkpoint and recovery overhead.

We describe how algorithmic features and identification of fault recovery data (we define it "critical data") are crucial in ABFT design. In that regard, we have developed a fault tolerance and recovery strategy for parallel search algorithms. The aspects of parallel search algorithms associate this to the class of embarrassingly parallel algorithms. We illustrate our idea with PIDA*, a generic parallel search algorithm used to solve a broad range of discrete optimization problems. Fault tolerance is accomplished by integrating our approach with the existing PIDA* algorithm without altering the original algorithm. We have observed that the FTPIDA* (fault tolerant PIDA*) scheme outperforms the CP/R concerning checkpoint and recovery overhead in a failure-free environment and also in the presence of single or multiple process failures. For instance, our simulation results show that the performance improvement of FTPIDA* is almost 30% for 20 processes and 67% for 80 processes as compared to CP/R in the presence of 50% simultaneous process failures. Also, the FTPIDA* outperforms the CP/R in total checkpoint data size. For example, it reduces the overall backup size by 92.77% with 20 processes.

Next, we justify that the presence of strong interdependence among subtasks in a communication exhaustive parallel application performs a vital role in ABFT design. We demonstrate that communication dependency among the processes can be exploited to replicate the fault recovery data (critical data) of one process to its peer process's memory. Additionally, we show that algorithmic characteristics decide what will be the critical data for such applications. We have developed a fault tolerance and recovery strategy for such communication-intensive parallel applications

and demonstrate it with the help of the parallel dynamic programming (DP) class of problems. We further show that for a n-dependent DP problem, where each subtask has a maximum dependency of $O(n)$, algorithm features can reduce the cardinality of critical data from n to 1. Compared with CP/R, our proposed ABFT approach performs significantly better in the case of fault tolerance and recovery cost in the presence of single and multiple process failures. Theoretically, we prove that with our proposed method, replication of a particular data (as critical data) with a peer can achieve an overall gain of $\approx 67\%$ for the LCS problem and for the TSP problem the gain is between 80% to $\approx 90\%$.

The size of fault recovery data is a critical performance bottleneck in FT design. More specifically, minimizing the fault recovery data size for the large-scale system is a major research issue in FT platforms. In that regard, finally, we have demonstrated that for particular DP types of problems, further optimization can be achieved when a subproblem has a fixed small degree of dependency with other subproblems. We evaluate this with a well-known DP problem, the 0/1 knapsack problem and propose an FT scheme for such a case. Here, instead of replicating the critical data, we replicate a bit-vector flag with peer processes for recovery purposes. This flag is used to rebuild the lost data of the failed process with the help of the information available in the other processes. Our simulation results show that the FT implementation has minimal extra message overhead over non-FT implementation. For instance, it is 4.32% with 12 processes and 6.12% with 48 processes. Moreover, for the fault tolerance overhead in a failure-free situation, our proposed FT scheme outperforms CP/R. In the presence of a single process failure, our proposed FT strategy has a constant performance gain of approximately 80% over CP/R with increasing numbers of processes.

6.1 Future Work

The work presented in the thesis goes to provide considerable effort to address the main challenges of existing FT techniques: (i)CP/R, and (ii)ABFT for large-scale HPC systems. Moreover, we present an ABFT solution for two large classes of parallel algorithms to address these issues. The objectives are to minimize fault tolerance and recovery overhead, optimize the size of the fault recovery data, and also legitimize the universal applicability of ABFT. However, there remain several future research directions which may add extra benefits to meet the challenges of the fault tolerance for current petascale and upcoming exascale systems.

This study handles two broad classes of parallel algorithms: (i) parallel search algorithms, and (ii) parallel DP algorithms. This work can be extended to other types of parallel algorithms: divide-and-conquer, graph algorithms, computational geometry, Fast Fourier transform (FFT), nearest-neighbor algorithm, and more to justify the applicability of ABFT.

This work addresses the hard or fail-stop type of faults. This research could be broadened to handle soft faults or errors. The soft error can be hazardous to scientific computation as in most of the cases the faulty components continue to execute the application without reporting any symptoms of the soft fault. The soft error has a catastrophic effect on computation and creates invalid computational results without ever being detected. Research has been done with the traditional ABFT to handle soft errors, but still, there is a lack of exploration. This could be another direction where we can expand our research.

Additionally, we designed and developed the fault tolerance strategy for the cluster, which is one of the platforms of HPC systems. Graphics processing units (GPUs) are another fascinating and inexpensive parallel environment for high-performance applications. But unfortunately, GPUs are very susceptible to soft or transient failures.

However, scientific applications with massive data processing need more reliability even with acceptable overheads. Fault tolerance and its usability for many-core accelerator architectures like GPUs are considered as a critical issue to the efficient use of GPUs for scientific applications. Limited research has been done on developing ABFT for heterogeneous systems with GPUs. This will be one other direction of our future work.

Bibliography

[1] N. Ali, S. Krishnamoorthy, M. Halappanavar, and J. Daily. Tolerating correlated failures for generalized cartesian distributions via bipartite matching. In *Proceedings of the 8th ACM International Conference on Computing Frontiers*, CF '11, pages 36:1–36:10, New York, NY, USA, 2011. ACM.

[2] N. Ali, S. Krishnamoorthy, M. Halappanavar, and J. Daily. Multi-fault tolerance for cartesian data distributions. *International Journal of Parallel Programming*, 41(3):469–493, 2013.

[3] G. Y. Ananth, V. Kumar, and P. Pardalos. Parallel processing of discrete optimization problems. In *IN ENCYCLOPEDIA OF MICROCOMPUTERS*, pages 129–147. Marcel Dekker Inc, 1993.

[4] J. Anfinson and F. T. Luk. A linear algebraic model of algorithm-based fault tolerance. *IEEE Trans. Comput.*, 37(12):1599–1604, Dec. 1988.

[5] A. C. S. Association. The computer failure data repository (cfdr).

[6] P. Banerjee, J. T. Rahmeh, C. Stunkel, V. S. Nair, K. Roy, V. Balasubramanian, and J. A. Abraham. Algorithm-based fault tolerance on a hypercube multiprocessor. *IEEE Trans. Comput.*, 39(9):1132–1145, Sept. 1990.

[7] L. Bautista-Gomez, S. Tsuboi, D. Komatitsch, F. Cappello, N. Maruyama, and S. Matsuoka. Fti: high performance fault tolerance interface for hybrid

systems. In *Proc. International Conference for High Performance Computing, Networking, Storage and Analysis, 2011*, SC '11, pages 32:1–32:32, New York, NY, USA, 2011. ACM.

[8] R. Bellman. On the theory of dynamic programming. *Proceedings of the National Academy of Sciences of the United States of America*, 38(8):716–719, August 1952.

[9] R. Bellman. *Dynamic Programming*. Princeton University Press, Princeton, NJ, USA, 1957.

[10] K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, J. Hiller, S. Karp, S. Keckler, D. Klein, R. Lucas, M. Richards, A. Scarpelli, S. Scott, A. Snavely, T. Sterling, R. S. Williams, K. Yelick, K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, J. Hiller, S. Keckler, D. Klein, P. Kogge, R. S. Williams, and K. Yelick. Exascale computing study: Technology challenges in achieving exascale systems, 2008.

[11] D. L. Boley, R. P. Brent, G. H. Golub, and F. T. Luk. Algorithmic fault tolerance using the lanczos method. *SIAM J. Matrix Anal. Appl.*, 13(1):312–332, Jan. 1992.

[12] G. Bosilca, A. Bouteiller, T. Herault, Y. Robert, and J. Dongarra. Assessing the impact of abft and checkpoint composite strategies. In *2014 IEEE International Parallel Distributed Processing Symposium Workshops*, pages 679–688, May 2014.

[13] G. Bosilca, A. Bouteiller, T. Hérault, Y. Robert, and J. J. Dongarra. Composing

resilience techniques: Abft, periodic and incremental checkpointing. *IJNC*, 5(1):2–25, 2015.

[14] G. Bosilca, R. Delmas, J. Dongarra, and J. Langou. Algorithm-based fault tolerance applied to high performance computing. *J. Parallel Distrib. Comput.*, 69(4):410–416, Apr. 2009.

[15] G. Boslica, A. Bouteiller, T. Hérault, Y. Robert, and J. Dongarra, Jack. Assessing the impact of ABFT and checkpoint composite strategies. In *IEEE International Parallel & Distributed Processing Symposium Workshops*, pages 679–688, 2014.

[16] A. Bouteiller, F. Cappello, T. Herault, G. Krawezik, P. Lemarinier, and F. Magniette. Mpich-v2: A fault tolerant mpi for volatile nodes based on pessimistic sender based message logging. In *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, SC '03, pages 25–, New York, NY, USA, 2003. ACM.

[17] A. Bouteiller, T. Hérault, G. Bosilca, P. Du, and J. Dongarra. Algorithm-based fault tolerance for dense matrix factorizations, multiple failures and accuracy. *ACM Trans. Parallel Comput.*, 1(2):1–28, Feb. 2015.

[18] G. Bronevetsky, D. Marques, K. Pingali, and P. Stodghill. Automated application-level checkpointing of mpi programs. In *Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '03, pages 84–94. ACM, 2003.

[19] S. Caminiti, I. Finocchi, and E. G. Fusco. Local dependency dynamic programming in the presence of memory faults. In *28th International Symposium*

on Theoretical Aspects of Computer Science (STACS 2011), volume 9 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 45–56, Dagstuhl, Germany, 2011. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[20] S. Caminiti, I. Finocchi, E. G. Fusco, and F. Silvestri. Resilient dynamic programming. *Algorithmica*, 77(2):389–425, February 2017.

[21] F. Cappello. Fault tolerance in petascale/ exascale systems: Current knowledge, challenges and research opportunities. *Int. J. High Perform. Comput. Appl.*, 23(3):212–226, Aug. 2009.

[22] F. Cappello, G. Al, W. Gropp, S. Kale, B. Kramer, and M. Snir. Toward exascale resilience: 2014 update. *Supercomput. Front. Innov.: Int. J.*, 1(1):5–28, Apr. 2014.

[23] F. Cappello, H. Casanova, and Y. Robert. Checkpointing vs. migration for post-petascale supercomputers. In *Proc. 39th International Conference on Parallel Processing, 2010*, ICPP '10, pages 168–177, Washington, DC, USA, 2010. IEEE Computer Society.

[24] H. Casanova, F. Vivien, and D. Zaidouni. *Using Replication for Resilience on Exascale Systems*, pages 229–278. Springer International Publishing, Cham, 2015.

[25] S. Chakravorty and L. V. Kale. A fault tolerance protocol with fast fault recovery. In *IEEE International Parallel and Distributed Processing Symposium, IPDPS*. IEEE, 2007.

[26] Z. Chen. Extending algorithm-based fault tolerance to tolerate fail-stop failures

in high performance distributed environments. In *Proceedings of IEEE International Symposium on Parallel and Distributed Processing*, IPDPS, pages 1–8, New York, NY, USA, 2008. ACM.

[27] Z. Chen. Algorithm-based recovery for iterative methods without checkpointing. In *Proceedings of the 20th international symposium on High performance distributed computing*, HPDC '11, pages 73–84. ACM, 2011.

[28] Z. Chen and J. Dongarra. Algorithm-based checkpoint-free fault tolerance for parallel matrix computations on volatile resources. In *Proceedings of the 20th international conference on Parallel and distributed processing*, IPDPS'06. IEEE Computer Society, 2006.

[29] Z. Chen and J. Dongarra. Algorithm-based fault tolerance for fail-stop failures. *IEEE Transactions on Parallel and Distributed Systems*, 19(12):1628–1641, Dec 2008.

[30] Z. Chen, G. E. Fagg, E. Gabriel, J. Langou, T. Angskun, G. Bosilca, and J. Dongarra. Fault tolerant high performance computing by a coding approach. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '05, pages 213–223, New York, NY, USA, 2005. ACM.

[31] D. Cook and R. C. Varnell. Adaptive parallel iterative deepening search. *Journal of Artificial Intelligence Research*, 9:139–166, 1998.

[32] I. Cores, G. Rodríguez, M. J. Martín, P. González, and R. R. Osorio. Improving scalability of application-level checkpoint-recovery by reducing checkpoint sizes. *New Generation Comput.*, 31(3):163–185, 2013.

[33] I. Cores, M. Rodriguez, P. González, and M. J. Martín. Reducing the overhead of an MPI application-level migration approach. *Parallel Computing*, 54:72–82, 2016.

[34] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2006.

[35] C. Coti, T. Herault, P. Lemarinier, L. Pilard, A. Rezmerita, E. Rodriguez, and F. Cappello. Blocking vs. non-blocking coordinated checkpointing for large-scale fault tolerant mpi. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM.

[36] DANA-Faber Cancer Institute and Harvard School of Public Health. The computational biology and functional genomics laboratory. `http://compbio.dfci.harvard.edu/tgi/`, 2014.

[37] S. Dasgupta, C. Papadimitriou, and U. Vazirani. *Algorithms*. McGraw-Hill Higher Education, 2008.

[38] T. Davies, Z. Chen, C. Karlsson, and H. Liu. Algorithm-based recovery for hpl. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP '11, pages 303–304, New York, NY, USA, 2011. ACM.

[39] T. Davies, C. Karlsson, H. Liu, C. Ding, and Z. Chen. High performance linpack benchmark: a fault tolerant implementation without checkpointing. In *Proceedings of the international conference on Supercomputing*, ICS '11, pages 162–171, New York, NY, USA, 2011. ACM.

[40] D. A. G. de Oliveira, L. Pilla, C. Lunardi, L. Carro, P. O. Navaux, and P. Rech. The path to exascale: Code optimizations and hardening solutions reliability.

In *Proceedings of the 5th Workshop on Fault Tolerance for HPC at eXtreme Scale*, FTXS '15, pages 55–62, New York, NY, USA, 2015. ACM.

[41] J. Dongarra, H. Meuer, and E. Strohmaier. Top 500 supercomputing sites, Jun 2017.

[42] P. Du, A. Bouteiller, G. Bosilca, T. Herault, and J. Dongarra. Algorithm-based fault tolerance for dense matrix factorizations. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 225–234. ACM, 2012.

[43] P. Du, A. Bouteiller, G. Bosilca, T. Herault, and J. Dongarra. Algorithm-based fault tolerance for dense matrix factorizations. *SIGPLAN Not.*, 47(8):225–234, Feb. 2012.

[44] I. P. Egwutuoha, D. Levy, B. Selic, and S. Chen. A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems. *J. Supercomput.*, 65(3):1302–1326, Sept. 2013.

[45] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375–408, Sept. 2002.

[46] C. Engelmann and F. Lauer. Facilitating co-design for extreme-scale systems through lightweight simulation. In *IEEE International Conference On Cluster Computing Workshops and Posters (CLUSTER WORKSHOPS)*, 2010.

[47] G. E. Fagg and J. Dongarra. FT-MPI: fault tolerant mpi, supporting dynamic applications in a dynamic world. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 7th European PVM/MPI Users' Group*

Meeting, Balatonfüred, Hungary, September 2000, Proceedings, pages 346–353, 2000.

[48] P. Felber, X. Défago, R. Guerraoui, and P. Oser. Failure detectors as first class objects. In *Proceedings of the International Symposium on Distributed Objects and Applications*, DOA '99, pages 132–, Washington, DC, USA, 1999. IEEE Computer Society.

[49] P. F. Felzenszwalb and R. Zabih. Dynamic programming and graph algorithms in computer vision. *IEEE Trans. Pattern Anal. Mach. Intell.*, 33(4):721–740, Apr. 2011.

[50] K. Ferreira, J. Stearley, J. H. Laros, III, R. Oldfield, K. Pedretti, R. Brightwell, R. Riesen, P. G. Bridges, and D. Arnold. Evaluating the viability of process replication reliability for exascale systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 44:1–44:12, New York, NY, USA, 2011. ACM.

[51] S. Fu. Failure-aware resource management for high-availability computing clusters with distributed virtual machines. *J. Parallel Distrib. Comput.*, 70(4):384–393, Apr. 2010.

[52] Q. Gao, W. Huang, M. J. Koop, and D. K. Panda. Group-based coordinated checkpointing for mpi: A case study on infiniband. In *Proceedings of the 2007 International Conference on Parallel Processing*, ICPP '07, pages 47–, Washington, DC, USA, 2007. IEEE Computer Society.

[53] M. R. Garey and D. S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990.

[54] A. Geist and R. Lucas. Major computer science challenges at exascale. *Int. J. High Perform. Comput. Appl.*, 23(4):427–436, Nov. 2009.

[55] G. Gibson, B. Schroeder, and J. Digney. Failure tolerance in petascale computers. *CTWatch Quarterly*, 3(4):4–10, Nov. 2004.

[56] A. Grama and V. Kumar. State of the art in parallel search techniques for discrete optimization problems. *IEEE Trans. on Knowl. and Data Eng.*, 11(1):28–35, Jan. 1999.

[57] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the mpi message passing interface standard. *Parallel Comput.*, 22(6):789–828, Sept. 1996.

[58] I. Gupta, T. D. Chandra, and G. S. Goldszmidt. On scalable and efficient distributed failure detectors. In *Proceedings of the Twentieth Annual ACM Symposium on Principles of Distributed Computing*, PODC '01, pages 170–179, New York, NY, USA, 2001. ACM.

[59] T. J. Hacker, F. Romero, and C. D. Carothers. An analysis of clustered failures on large supercomputing systems. *J. Parallel Distrib. Comput.*, 69(7):652–665, July 2009.

[60] J. Haines, V. Lakamraju, I. Koren, and C. M. Krishna. Application-level fault tolerance as a complement to system-level fault tolerance. *J. Supercomput.*, 16(1-2):53–68, May 2000.

[61] P. H. Hargrove and J. C. Duell. Berkeley lab checkpoint/restart (BLCR) for LINUX.

[62] P. H. Hargrove and J. C. Duell. Berkeley lab checkpoint/restart (BLCR) for Linux clusters. *Journal of Physics: Conference Series*, 46(1):494, 2006.

[63] K.-H. Huang and J. A. Abraham. Algorithm-based fault tolerance for matrix operations. *IEEE Trans. Comput.*, 33(6):518–528, June 1984.

[64] G. Jakadeesan and D. Goswami. A classification-based approach to fault-tolerance support in parallel programs. *Parallel and Distributed Computing Applications and Technologies, International Conference on*, 0:255–262, 2009.

[65] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to parallel computing: design and analysis of algorithms*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1994.

[66] V. Kumar, K. Ramesh, and V. N. Rao. Parallel best-first search of state-space graphs: A summary of results. In *in Proc. 10th Nat. Conf. AI, AAAI*, pages 122–127. Press, 1988.

[67] V. Kumar and V. N. Rao. Parallel depth first search. part ii. analysis. *International Journal of Parallel Programming*, 16(6):501–519, Dec 1987.

[68] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[69] H. Liu. An algorithm-based recovery scheme for extreme scale computing. In *Proceedings of the 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, IPDPSW '11, pages 2010–2013, Washington, DC, USA, 2011. IEEE Computer Society.

[70] F. T. Luk and H. Park. An analysis of algorithm-based fault tolerance techniques. *J. Parallel Distrib. Comput.*, 5(2):172–184, Apr. 1988.

[71] C. D. Martino, Z. T. Kalbarczyk, R. K. Iyer, F. Baccanico, J. Fullop, and W. Kramer. Lessons learned from the analysis of system failures at petascale:

The case of blue waters. In *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN, 2014, pages 610–621, 2014.

[72] A. Oliner. What supercomputers say: A study of five system logs. In *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, 2007. DSN '07*, pages 575–584, 2007.

[73] J. S. Plank, K. Li, and M. A. Puening. Diskless checkpointing. *IEEE Transactions on Parallel and Distributed Systems*, 9(10):972–986, October 1998.

[74] C. Quebec. Calcul quebec, compute canada. http://www.calculquebec.ca/en/, 2017.

[75] V. N. Rao and V. Kumar. Parallel depth first search. part i. implementation. *Int. J. Parallel Program.*, 16(6):479–499, Dec. 1987.

[76] V. N. Rao, V. Kumar, and K. Ramesh. A parallel implementation of iterative-deepening-a. In *Proceedings of the sixth National conference on Artificial intelligence - Volume 1*, AAAI'87, pages 178–182. AAAI Press, 1987.

[77] V. N. Rao, V. Kumar, and K. Ramesh. A parallel implementation of iterative-deepening-a. In *Proceedings of the Sixth National Conference on Artificial Intelligence - Volume 1*, AAAI'87, pages 178–182. AAAI Press, 1987.

[78] A. Reinefeld and V. Schnecke. Work-load balancing in highly parallel depth-first search. In *In Scalable High Performance Computing Conference*, pages 773–780, 1994.

[79] R. K. Sahoo, A. Sivasubramaniam, M. S. Squillante, and Y. Zhang. Failure data analysis of a large-scale heterogeneous server environment. In *Proceedings of the 2004 International Conference on Dependable Systems and Networks*, DSN '04, pages 772–, Washington, DC, USA, 2004. IEEE Computer Society.

[80] R. D. Schlichting and F. B. Schneider. Fail-stop processors: an approach to designing fault-tolerant computing systems. *ACM Trans. Comput. Syst.*, 1(3):222–238, Aug. 1983.

[81] B. Schroeder and G. A. Gibson. A large-scale study of failures in high-performance computing systems. In *Proceedings of the International Conference on Dependable Systems and Networks*, DSN '06, pages 249–258, Washington, DC, USA, 2006. IEEE Computer Society.

[82] B. Schroeder and G. A. Gibson. Disk failures in the real world: what does an mttf of 1,000,000 hours mean to you? In *Proceedings of the 5th USENIX conference on File and Storage Technologies*, FAST '07, Berkeley, CA, USA, 2007. USENIX Association.

[83] B. Schroeder and G. A. Gibson. Understanding failures in petascale computers, 2007.

[84] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[85] M. Snir, R. W. Wisniewski, J. A. Abraham, S. V. Adve, S. Bagchi, P. Balaji, J. Belak, P. Bose, F. Cappello, B. Carlson, A. A. Chien, P. Coteus, N. A. Debardeleben, P. C. Diniz, C. Engelmann, M. Erez, S. Fazzari, A. Geist, R. Gupta, F. Johnson, S. Krishnamoorthy, S. Leyffer, D. Liberty, S. Mitra, T. Munson, R. Schreiber, J. Stearley, and E. V. Hensbergen. Addressing failures in exascale computing. *Int. J. High Perform. Comput. Appl.*, 28(2):129–173, May 2014.

[86] P. Stodghil, G. Bronevetsky, D. Marques, K. Pingali, and R. Fernandes. Optimizing checkpoint sizes in the c3 system. *Parallel and Distributed Processing Symposium, International*, 11:226a, 2005.

[87] R. Subramaniyan, V. Aggarwal, A. Jacobs, and A. D. George. Fempi: A lightweight fault-tolerant mpi for embedded cluster systems. In *Proc. International Conference on Embedded Systems and Applications (ESA), Las Vegas*, pages 26–29, 2006.

[88] B. W. Wah and G.-j. Li. Systolic processing for dynamic programming problems. *Circuits, Systems and Signal Processing*, 7(2):119–149, Jun 1988.

[89] J. Walters and V. Chaudhary. Application-level checkpointing techniques for parallel programs. In *Distributed Computing and Internet Technology*, pages 221–234. Springer Berlin/Heidelberg, Springer Berlin/Heidelberg, 2006.

[90] C.-L. Wang, F. C. M. Lau, and J. C. Y. Ho. Scalable group-based checkpoint/restart for large-scale message-passing systems. In *IEEE International Parallel & Distributed Processing Symposium*, pages 1–12, Los Alamitos, CA, USA, 2008. IEEE Computer Society.

[91] P. Wang, Y. Du, H. Fu, X. Yang, and H. Zhou. Static analysis for application-level checkpointing of mpi programs. In *2008 10th IEEE International Conference on High Performance Computing and Communications*, pages 548–555, Sept 2008.

[92] R. Wang, E. Yao, M. Chen, G. Tan, P. Balaji, and D. Buntinas. Building algorithmically nonstop fault tolerant mpi programs. In *HiPC*, pages 1–9, 2011.

[93] Z. Wei and D. Goswami. A synchronization-induced checkpoint protocol for

group-synchronous parallel programs. In *Proceedings of 13th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT 2012)*, pages 632–637. IEEE, 2012.

[94] T. White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 1st edition, 2009.

[95] M. Wu, X.-H. Sun, and H. Jin. Performance under failures of high-end computing. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, SC '07, pages 48:1–48:11, New York, NY, USA, 2007. ACM.

[96] P. Wu, N. DeBardeleben, Q. Guan, S. Blanchard, J. Chen, D. Tao, X. Liang, K. Ouyang, and Z. Chen. Silent data corruption resilient two-sided matrix factorizations. In *Proceedings of the 22Nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '17, pages 415–427, New York, NY, USA, 2017. ACM.

[97] E. Yao, M. Chen, W. Zhang, and G. Tan. A new and efficient algorithm-based fault tolerance scheme for a million way parallelism. In *Proceedings of CoRR*, volume abs/1106.4213, 2011.

[98] E. Yao, R. Wang, M. Chen, G. Tan, and N. Sun. A case study of designing efficient algorithm-based fault tolerant application for exascale parallelism. In *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium*, IPDPS '12, pages 438–448, Washington, DC, USA, 2012. IEEE Computer Society.

[99] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark:

Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.

[100] Z. Zheng, L. Yu, W. Tang, Z. Lan, R. Gupta, N. Desai, S. Coghlan, and D. Buettner. Co-analysis of ras log and job log on blue gene/p. In *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, IPDPS '11, pages 840–851, Washington, DC, USA, 2011. IEEE Computer Society.

[101] Zuse Institute Berlin. Mp-testdata-traveling salesman problem instances. http://elib.zib.de/pub/mp-testdata/tsp/tsplib/tsplib.html.