ALGORITHM-BASED RECOVERY FOR NEWTON'S METHOD WITHOUT CHECKPOINTING

by

Hui Liu

A thesis submitted to the Faculty and the Board of Trustees of the Colorado School of Mines in partial fulfillment of the requirements for the degree of Master of Science (Mathematical and Computer Sciences).

Golden, Colorado

Date 8/29/2011

Signed: Hui Liu

Hui Liu

Approved:

Dr. Zizhong Chen

Assistant Professor of Computer Sciences

Thesis Advisor

Golden, Colorado

Date 08/29/2011

Tracy Camp

Dr. Tracy Camp

Professor and Head,

Department of Mathematical and Computer Sciences

ABSTRACT

Fault tolerance is a necessary capability for today's large parallel systems. Checkpointing is the most popular fault tolerance method used in high-performance computing (HPC). However, increasingly frequent checkpoints are required because of growing failure rates. This fact makes checkpointing more expensive. We present a checkpoint-free fault tolerance technique for Newton's method. It takes advantage of both inherent data dependencies and communication-induced redundancies of parallel applications to tolerate fail-stop failures. Under some conditions, our technique introduces no additional overhead when there is no actual failure in the computation and recovers the lost data with low overhead.

We add fault-tolerance capacity to Newton's method by using our scheme, disk-based checkpointing method and diskless checkpointing method and compare the overhead introduced by these three methods. Experimental results indicate that our scheme introduces much less overhead than disk-based checkpointing and diskless checkpointing.

ACKNOWLEDGMENT

I express my appreciation to my committee members: Dr. Zizhong Chen, Dr. Dinesh Mehta and Dr. Qi Han, for their numerous suggestions and help on my thesis and course work. I am deeply indebted to my advisor, Dr. Zizhong Chen, for his valuable guidance and continuous support during my studies at the Colorado School of Mines.

I appreciate the friendship with friends in the Mathematical and Computer Sciences Department. Teresa Davies, Christer Karlsson and Chong Ding have helped and encouraged me during the past two years. I really appreciate them for many thoughtful discussions.

I also acknowledge the financial support I received from the Department of Mathematical and Computer Sciences and additional support from Dr. Zizhong Chen.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENT

I express my appreciation to my committee members: Dr. Zizhong Chen, Dr. Dinesh Mehta and Dr. Qi Han, for their numerous suggestions and help on my thesis and course work. I am deeply indebted to my advisor, Dr. Zizhong Chen, for his valuable guidance and continuous support during my studies at the Colorado School of Mines.

I appreciate the friendship with friends in the Mathematical and Computer Sciences Department. Teresa Davies, Christer Karlsson and Chong Ding have helped and encouraged me during the past two years. I really appreciate them for many thoughtful discussions.

I also acknowledge the financial support I received from the Department of Mathematical and Computer Sciences and additional support from Dr. Zizhong Chen.

Chapter 1

INTRODUCTION

Parallel system is composed of a large number of hardware and software components. Increasing the number of components simply increases the probability of failures (Schroeder & Gibson, 2006). Parallel system failures are often caused by hardware or software failures. Failures can often be classified into two types:

- fail-stop failures;
- non-fail-stop failures.

Fail-stop failures (Schlichting & Schneider, 1983; Hwang *et al.*, 2005) are such failures where a failed process ceases to operate without sending malicious messages or destroys all associated data. If the failed process can still work but produces incorrect calculations, this failure belongs to the second type.

Today's high performance computing (HPC) applications employ a large number of parallel components to perform the computations and take from a few hours to a few months to complete. If a failure occurs in these lengthy computations, the program needs to be restarted, which is a waste of time. It is necessary to incorporate fault-tolerance in HPC applications in order to tolerate failures. Here, fault-tolerance refers to a technique that allows a system to continue working correctly after a failure.

To handle non-fail-stop failures, researchers (Huang & Abraham, 1984; Anfinson & Luk, 1988) propose algorithm-based fault tolerance (ABFT). These schemes provide error protection in systolic arrays and achieve a very low overhead. ABFT can detect, locate, and correct certain processor miscalculations in matrix computations by using a checksum approach.

To handle fail-stop failures, several fault-tolerance methods have been developed, such as checkpointing and message logging. Here, checkpointing is a kind of method that relies on saving the states of processes in stable storage during failure-free execution. These states of processes are called checkpoints. The checkpoints contain sufficient information for recovery.

We focus on recovering fail-stop failures using the checkpointing-free method in this thesis. The overview of our scheme will be briefly introduced in Section 1.3. Before moving to details, we will briefly review the latest improvements in both checkpointing (Section 1.1) and some other recovery methods (Section 1.2).

1.1 Checkpointing

Checkpointing is a widely used mechanism in modern parallel systems that allows applications to protect themselves against hardware, software and even power failures. HPC applications typically employ checkpointing with rollback recovery to tolerate fail-stop failures. In this case, the application periodically stores checkpoints, including messages received, to stable storage or in memory. If a failure occurs that causes the application to be terminated prematurely, the application can restart from the most recent state by reading saved checkpoints. Checkpointing loses at most an interval computation time in the event of a failure. In the absence of checkpointing, one would need to restart the whole computation from the beginning, thus all previous computations would be wasted.

There are three different levels of checkpointing:

1. system-level checkpointing;

2. compiler-level checkpointing;

3. application-level checkpointing.

A checkpointing scheme performed by the operating system-level (Michele & Regis, 2007) provides automatic recovery without any effort from the programmers. However, it has no knowledge about the application. As a result, it backs up all of an application's states, including some messages that may not be used any further.

The implementation of compiler-based checkpointing schemes (Li *et al.*, 1994) is transparent to the programmer. During the compilation of source code without checkpoints specified, a compiler is adapted to identify checkpoints in the source code and generate checkpoint code to store state information for recovery.

Application-level checkpointing implementations (Moody *et al.*, 2010; Bronevetsky *et al.*, 2003a,b; Elnozahy & Plank, 2004) require the programmer to identify checkpoints in the program and decide which data are chosen to save, increasing the programmer burden and reducing productivity. It has several advantages. First, it runs on different machines and operating systems. Second, in some applications, a few key data structures of the entire computational state need to be saved for recovery. Therefore, the overhead is reduced by decreasing the size of checkpoints.

Currently, there exist several commercial systems and publicly available libraries (Elnozahy & Plank, 2004) supporting various flavors of checkpointing at a low performance overhead. In (Bronevetsky *et al.*, 2003a) and (Bronevetsky *et al.*, 2003b), Bronevetsky *et al.* implement an application-level, coordinated, nonblocking checkpointing system. A C/MPI program can be automatically converted into the equivalent fault-tolerant version by using this system. Lipckpt (Plank *et al.*, 1995) is a portable checkpointing tool for uniprocess programs. Multi-level checkpointing (Gelenbe, 1976; Vaidya, 1994; Moody *et al.*, 2010) allows applications to store lower-overhead, less-resilient checkpoints to stable storage and write the slowest but most resilient checkpoints to the parallel file system. Therefore, it benefits existing systems with better efficiency, reduced load and the ability to tolerate multiple failures.

I/O operation is the main source of overhead in checkpointing. Plank *et al.* develop the diskless checkpointing technique (Plank *et al.*, 1998) that efficiently stores checkpoints in process memories. The technique removes the I/O bottleneck. Compared to standard disk-based checkpointing, it has a lower expected running time in the presence of single process failures.

1.2 Algorithm-Based Recovery for Linear Algebra Operations

Checkpointing can recover fail-stop failures with a relatively satisfiable overhead. However, it is not always the optimal choice to make a computation fault-tolerant. For example, for the application of dense linear algebra computations, since a large amount of memory between checkpoints is often modified, checkpointing introduces considerable overhead in extreme scale systems.

Chen and Dongarra (Chen & Dongarra, 2008) propose a highly scalable checkpoint-free technique to tolerate single fail-stop failure in high performance matrix operations on large scale HPC systems. They prove, that for the outer product version matrix-matrix multiplication algorithm, a checksum relationship in input checksum matrices may be maintained during computation. They also demonstrate that single fail-stop failure in ScaLAPACK (Blackford *et al.*, 1997) matrix multiplication can be tolerated without check-pointing at a decreasing overhead rate of $1/\sqrt{p}$, where p is the number of processes involved for computation.

Chen (Chen, 2011) presents an algorithm-based recovery scheme to tolerate fail-stop failure without checkpointing for many parallel iterative methods, including the

Jacobi method, the Gauss-Seidel method, the successive overrelaxation method, the symmetric successive overrelaxation method, the preconditioned conjugate gradient method, and the preconditioned biconjugate gradient stabilized method. Under certain conditions, inherent redundant information maintained by the iterative methods is enough to recover the lost data on the failed process. With the proposed scheme, one can restart the computations from where the failure occurs.

1.3 Overview of the Proposed Scheme

We present an algorithm-based recovery scheme which tolerates faults without checkpointing. Our scheme uses the parallel application's data dependencies and communication-induced redundancies to recover the lost data on the failed process.

Data Dependencies A data dependency is a relationship between or among attributes such that the value of one attribute depends on the values of the other attributes. Dependency relationships are very common among program variables and can be used to recover lost data. For example, matrix C can be determined by matrices A, B, where $C = A + B$. These three matrices have different values in different iterations. Processes often achieve fault tolerance by saving them in checkpoints periodically during failure-free execution. In fact, if A and B are available, then C can be recovered from the dependency relationship $C = A + B$. Thus A and B are needed to be saved for the recovery, reducing the size of saved data.

Communication-induced Redundancies A distributed memory programming model is often used in parallel implementations for many applications. In distributed memory machines, large data is partitioned among processes. Non-local data is accessed through inter-process communications, which is implemented via

MPI (Snir *et al.*, 1998). These inter-process communications automatically duplicate the same data from sender to receiver. We define the received messages as communication-induced redundancies.

If a failed process's communication-induced redundancies contain its necessary recovery information, its lost data can be accurately recovered by communicating with its neighbors. Otherwise, we modify the message-passing pattern to force the process into sending additional messages to its neighbor processes. Sometimes this modification introduces no additional time overhead.

Whenever a fail-stop failure occurs, our scheme uses the redundant information to recover the lost data on the failed process. Neither checkpoint nor roll-back is necessary. Our technique introduces less overhead compared to checkpointing techniques. Under certain conditions, our technique introduces no overhead when there is no actual failure in the execution.

Nonlinear systems are always inevitable in many science and engineering disciplines such as geophysics, chemistry and physics. Newton's method is the most popular numerical technique for solving nonlinear systems. We add fault-tolerant ability to Newton's method by using our scheme, disk-based and diskless checkpointing (Plank *et al.*, 1998). Comparisons show that our scheme introduces much less overhead than disk-based and diskless checkpointing.

The material in this thesis is, in part, a reproduction of published material (Liu *et al.*, 2011).

1.4 Thesis Organization

The rest of the thesis is organized as follows: **Chapter 2** introduces our communication-induced checkpoint-free fault tolerance technique. **Chapter 3** introduces the basic

Newton's method and parallel Newton's method for nonlinear systems. We develop fault tolerant versions of parallel Newton's method so that fail-stop failures in these computations can be tolerated without checkpointing. This approach can be efficiently implemented by using MPI (Snir *et al.*, 1998) and ScaLAPACK (Blackford *et al.*, 1997). In **Chapter 4**, we analyze the fault tolerance overhead and scalability of the proposed method. In **Chapter 5**, we experimentally compare our method with disk-based and diskless checkpointing on ra.mines.edu. **Chapter 6** concludes the thesis and discusses future work.

Chapter 2

THE PROPOSED FAULT TOLERANCE METHOD

Many parallel programs in a message-passing distributed system employ a number of processes to perform a computation and use MPI to exchange messages. A failure in one process results in a failure of the entire computation. The computation cannot continue until the failed process is recovered and all processes involved in the computation are in the same state.

To allow a successful recovery, the fault tolerance scheme must contain enough redundancy. Our fault tolerance implementations require the programmer to choose sent messages within the execution of the application such that the collections of messages sent by all processes will yield the necessary recovery information. This method depends on data dependencies and communication-induced redundancies of the application. Furthermore, the programmer needs to modify the message passing pattern and implement recovery code, including the decision of which data structures and message passing patterns need to be constructed in order to improve the efficiency of communication and recovery. The recovery code reconstructs the connections among the processes and recovers the lost data in the failed process by using the stored redundant information on the other processes.

2.1 Failure Model

Part of the distributed system can fail while the rest continues to work. Our scheme can take advantage of the redundancy offered by distributed systems to make applications more robust. The fail-stop failure model (Schlichting & Schneider, 1983; Hwang *et al.*, 2005) appears frequently in distributed systems. We can create tests only for the modeled faults and use experiments to reflect the effectiveness of fault-tolerance techniques.

The fail-stop model makes some assumptions about the behavior of processes: the failed processes stop working and cannot communicate with the others; the surviving processes can continue working when some processes die in the message passing system, and it is possible to replace the failed processes in the message passing system and continue the communication after the replacement.

Generally, the following three steps handle fault-tolerance (Chen & Dongarra, 2008) :

1. fault detection
2. fault location
3. fault recovery

Many programming environments such as FT-MPI (Fagg *et al.*, 2004) and Open MPI (Gabriel *et al.*, 2004) can help the program to detect and locate fail-stop process failures.

2.2 The Proposed Method

In the absence of failures, processes communicate with one another by sending and receiving messages to access non-local data and perform their own computations, as shown in Figure 2.1. In practice, we do our fault tolerance implementations by writing code to (i) save the values of key received variables into memory during failure-free execution, and (ii) restore the entire computational state from relatively small

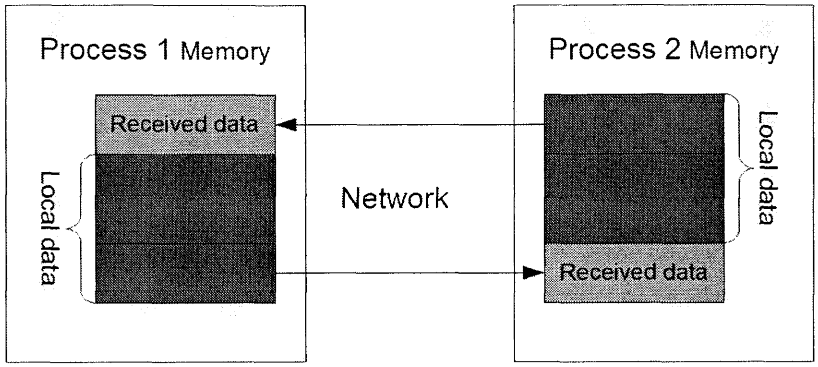


Figure 2.1. Parallel Programming Models

amounts of saved data during recovery. Using our scheme, it is possible to restart the application from where the failure occurs without checkpointing.

2.2.1 Data Dependencies

Dependency relationships between different variables are very common and play an important role in many classical numerical algorithms. Data B is dependent upon data A if B can be uniquely determined from A in a relation. By using our scheme, the programmer is required to exploit knowledge about the application to figure out the inherent dependencies among the program's variables in order to reduce the size of saved data.

For example, Newton's method is the most popular numerical technique for solving the nonlinear system $F(\vec{x}) = \mathbf{0}$. Details are discussed in Section 3.1. An inherent dependency of Newton's method is that the Jacobian matrix of F depends on $\vec{x}$. After a failure, our scheme can use communication-induced redundancies to recover $\vec{x}$ on the failed process. Then the relative Jacobian matrix can be recovered directly by using the data dependency. Thus, we can recover Jacobian matrix without storing it. In other words, $\vec{x}$ is the data needed to be stored while the Jacobian matrix is recovered by using data dependencies.

Checkpointing can also benefit from inherent data dependencies. The overhead of checkpointing in a failure-free environment is dominated by the cost of saving the checkpoints on stable storage (i.e., disk). This cost depends on the checkpoint size. Therefore, reducing the overhead of checkpointing can be achieved by minimizing the sizes of checkpoints. Based on the inherent data dependencies, we can safely omit certain data from the checkpoint and guarantee that the checkpoint contains enough information for recovery. In some applications, we can significantly reduce checkpoint size by using data dependencies to achieve a low overhead of checkpointing.

2.2.2 Communication-induced Redundancies

A process of the message-passing distributed system receives some messages from the others and then performs some computations. If there is a message on its way, the process should wait for it. The relative computation is blocked until all required messages are received. These received messages are the inherent redundant information for the system during failure-free operation. If the receiver records the inherent redundant information, it is possible to recover the lost data on the failed processes without either checkpoint or roll-back. The computation can be restarted from where the failure occurs. If there is no failed processor in the total computation, the fault tolerate overhead (time) sometimes is zero.

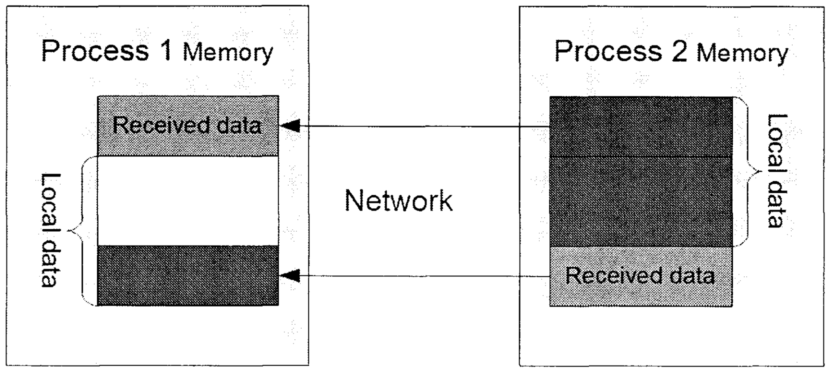
Assume that there is p available processes. Let the process i send the message S_{ij} to the process j, where $i \neq j, 1 \leq i, j \leq p$. R_i denotes the process i's data that needs to be stored on other processes according to the data dependency analysis.

If $R_i \subseteq \bigcup_{i \neq j, j=1}^{p} S_{ij}$, the sent messages are enough for a successful recovery. Our scheme does not introduce overhead during failure-free execution.

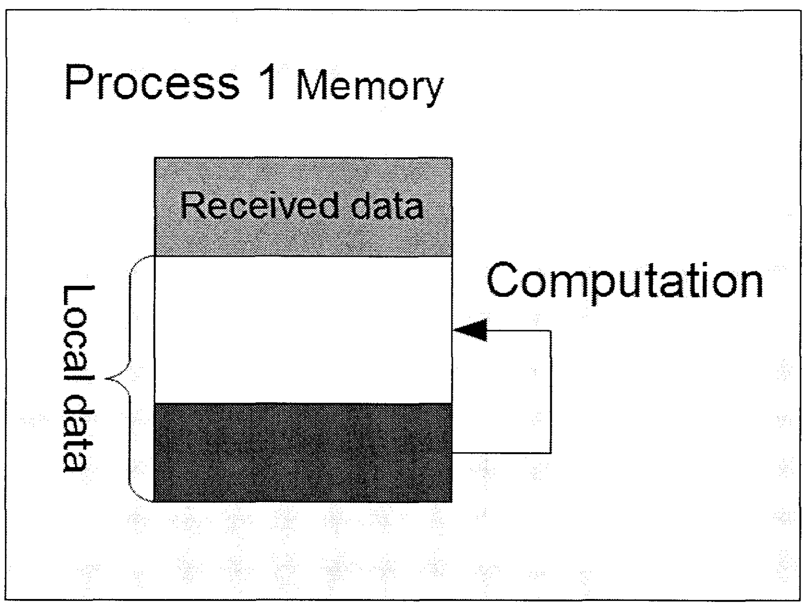
Otherwise, we modify the message-passing pattern in order to increase the amount of redundancy. A potential way is forcing processes to send more messages which are the unsent necessary recovery information in the previous message-passing pattern. The new message-passing pattern introduces the additional communication overhead. The main challenge is minimizing the additional communication overhead.

2.2.3 Single Failure Recovery

If process i loses all associated data, it asks its neighbors to send back its necessary recovery information. Thus, some variables of process i are directly recovered through communication. Our scheme uses inherent dependencies among the variables to recover the remaining variables via calculating. Once the rest of the variables are calculated, all data on process i is recovered and the application continues from the failure point. An example is shown in Figure 2.2.



(a) Phase 1: process 1 asks process 2 to send data back. The middle part is the lost data of process 1.



(b) Phase 2: The middle part is recovered through computation by using data dependencies.

Figure 2.2. Single Failure Recovery by Using Our Scheme

Chapter 3

FAULT TOLERANT PARALLEL NEWTON'S METHOD

In this chapter, we show how to reconstruct the lost data on a failed process in parallel Newton's method to solve nonlinear systems. From now on, we represent fault tolerant parallel Newton's method by FT-Newton.

Fault tolerance can be achieved by using our scheme, which is a communication-induced checkpoint-free fault tolerance technique. Our scheme uses the data dependencies of Newton's method and communication-induced redundancies to recover the lost data on the failed process.

3.1 Newton's Method

Consider the nonlinear system

$$F(\vec{x}) = \mathbf{0}, \quad \vec{x} = (x_1, x_2, \cdots, x_n) \tag{3.1}$$

where $\vec{x} \in \Re^n$ are the variables and $F(\vec{x}) = (f_1(\vec{x}), f_2(\vec{x}), \cdots, f_n(\vec{x}))$. We denote the Jacobian matrix of F at $\vec{x}$ by $J(\vec{x})$.

$$J(\vec{x}) = \begin{pmatrix} \frac{\partial f_1(\vec{x})}{\partial x_1} & \frac{\partial f_1(\vec{x})}{\partial x_2} & \cdots & \frac{\partial f_1(\vec{x})}{\partial x_n} \\ \frac{\partial f_2(\vec{x})}{\partial x_1} & \frac{\partial f_2(\vec{x})}{\partial x_2} & \cdots & \frac{\partial f_2(\vec{x})}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_n(\vec{x})}{\partial x_1} & \frac{\partial f_2(\vec{x})}{\partial x_2} & \cdots & \frac{\partial f_n(\vec{x})}{\partial x_n} \end{pmatrix} \tag{3.2}$$

$$\vec{g} = \begin{pmatrix} f_1(\vec{x}) \\ f_2(\vec{x}) \\ \vdots \\ f_n(\vec{x}) \end{pmatrix} \tag{3.3}$$

Assume that there is $\vec{x}^* \in \Re^n$ such that $F(\vec{x}^*) = \mathbf{0}$ with $J(\vec{x}^*)$ nonsingular.

Nonlinear systems are very hard to solve explicitly. In some instances, direct methods are computationally prohibitive. Generally, Newton's method (Dennis J. E. & Schnabel, 1996) linearizes the nonlinear system about the current approximation. This will result in a linear system and thus can be solved for the next approximation. The procedure is iteratively executed until some certain stop criteria is met. The main advantages of Newton's method are the fast rate of convergence and easy implementation. Due to those advantages, it is one of the most powerful techniques for nonlinear systems. It is well known that Newtons method has local quadratic convergence if the Jacobian matrix is Lipschitz continuous and nonsingular at a solution $\vec{x}^*$ of the system (Dennis J. E. & Schnabel, 1996). Details are shown in Algorithm 1. Many applied problems can be reduced to nonlinear systems. However, the dimension of modern problems (millions of variables and hundreds of thousands of nonlinear equations) may be too large to be solved by sequential Newton's method. Therefore, a parallel version of Newton's method is needed for solving such large-scale scientific computing problems on parallel computer systems.

Large-scale scientific simulation has driven the size of high performance computers from thousands, to tens of thousands, and even to hundreds of thousands of processors. Linear system solver is the key computational kernel of the parallel Newton's method. Very efficient implementation of this operation for parallel computers

Algorithm 1 Basic Newton's method

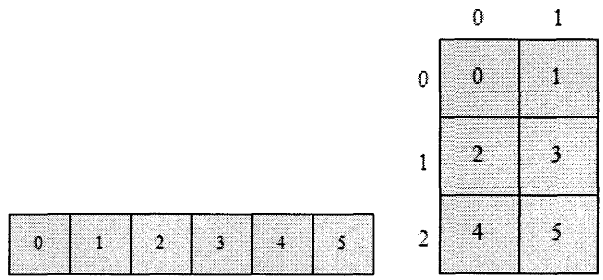
Require: $n > 0, \vec{x_0}, \varepsilon$;
$\vec{x} = \vec{x_0}$;
$\vec{g} = F(\vec{x})$;
$J = J(\vec{x})$;
while $\|\vec{g}\| > \varepsilon$ **do**
Solve $J\vec{v} = -\vec{g}$;
$\vec{x} = \vec{x} + \vec{v}$;
$\vec{g} = F(\vec{x})$;
$J = J(\vec{x})$;
end while

with distributed architectures exists within ScaLAPACK (Blackford *et al.*, 1997). Thus, we avoid the parallelization difficulties in Newton's method. TOUGH2-MP (http://tough2.com/index2.html) is a general-purpose numerical parallel simulation program for multi-phase fluid and heat flow in porous and fractured media, which employs Newton's method to solve nonlinear algebraic equations.

Data decomposition is the core operation in constructing parallel algorithms. By using the two-dimensional block-cyclic data distribution (Blackford *et al.*, 1997), ScaLAPACK makes dense matrix computations as efficient as possible. The array descriptor $DESC\_$ is used for the ScaLAPACK routines solving dense linear systems and eigenvalue problems, which contain the details on block cyclic data distribution.

ScaLAPACK organizes this one-dimensional array of processes into a two-dimensional process grid. Therefore, we can identify a process by using its row and column coordinates. An example of such a map is shown in Figure 3.1.

The block-cyclic distribution scheme maps the global matrix onto the rectangular process grid. In this distribution, the global matrix is first divided into several blocks, and each process owns a collection of blocks. Given an element a_{ij} of the global matrix



(a) One-dimensional process array (b) Two-dimensional process grid

Figure 3.1. Process Grid in ScaLAPACK

A, the process coordinate (p_i, q_j) that a_{ij} resides can be calculated by

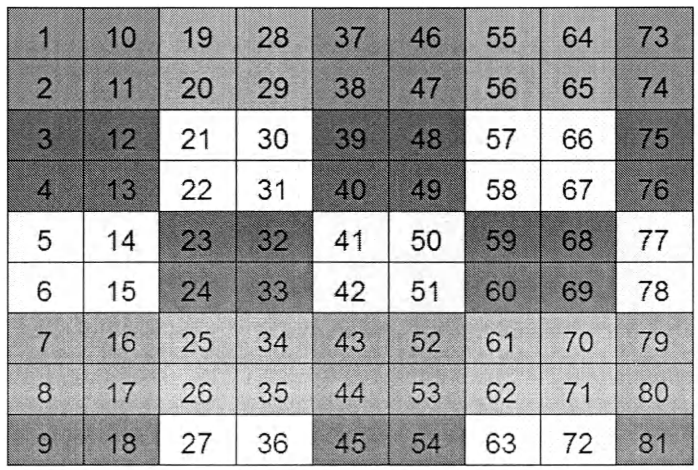
$$
\begin{aligned}
p_i &= \lfloor \frac{i}{mb} \rfloor \% P \\
q_j &= \lfloor \frac{j}{nb} \rfloor \% Q
\end{aligned}
\tag{3.4}
$$

where mb is the row block size, nb is the column block size, P is the number of process rows in the process grid and Q is the number of process columns in the process grid. A vector is distributed, considering the vector as a column of the matrix.

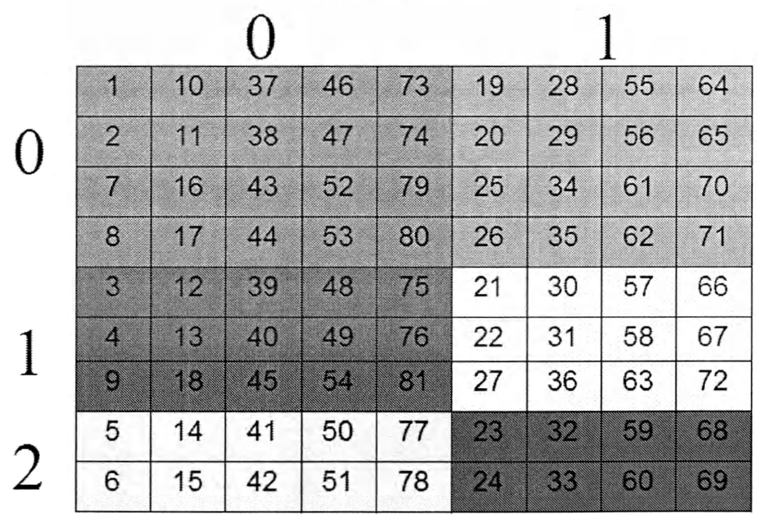
Figure 3.2 is an example of mapping a 9×9 matrix A onto a 3×2 process grid according two-dimensional block-cyclic data distribution with $mb = nb = 2$.

3.2 Parellel Newton's Method

The parallel implementation of Newton's method requires that the operation for $J\vec{v} = -\vec{g}$ be parallelized. In this thesis, we use ScaLAPACK to solve $J\vec{v} = -\vec{g}$. Thus J is distributed onto the process grid by using the two-dimensional block-cyclic data distribution. In this case, vectors $\vec{x}$, $\vec{v}$ and $\vec{g}$ are distributed onto one column of process grid.



(a) Global view



(b) Local (distributed) view: mapping a 9×9 matrix onto a 3×2 process grid.

Figure 3.2. The Two-Dimensional Block-Cyclic Data Distribution

In order to construct Jacobian matrix J, the parallel method requires data exchange for $\vec{x}$. Assume all elements in matrices and vectors are 8-byte double precision floating-point numbers. Assume every process has the same speed and disjoint pairs of processes can communicate without interfering with each other. Assume it takes $\alpha + \beta k$ seconds to transfer a message of k bytes regardless of which processes are involved, where α is the latency of the communication and $\frac{1}{\beta}$ is the bandwidth of the communication. Assume a process can concurrently send a message to one partner and receive a message from a possibly different partner.

For example, each process has a copy of global $\vec{x}$ by employing MPI's all-gather reduction. In this implementation, the all-gather reduction requires each process to send m messages. The total number of $\vec{x}$ elements transmitted during the all-gather is $n(2^m - 1)/2^m$. Hence the expected execution time for the all-gather step is $\alpha m + 8n\beta(2^m - 1)/2^m$. To broadcast the global $\vec{x}$ using a simple binary tree broadcast algorithm, it takes $2(\alpha + 8n\beta)log_2Q$. Therefore, the time T_{vector_comm} for the communication in the Newton's step is:

$$\begin{aligned} T_{vector\_comm} &= \alpha m + 8n\beta(2^m - 1)/2^m + 2(\alpha + 8n\beta)log_2Q \\ &= \alpha(\frac{n}{P} + 2log_2Q) + 8n\beta[(2^{\frac{n}{P}} - 1)/2^{\frac{n}{P}} + 2log_2Q] \end{aligned}$$

3.3 FT-Newton with Our Scheme

In parallel Newton's method, the dense vector $\vec{x}$, g and the symmetric positive definite matrix J are often partitioned into p sub-vectors and sub-matrixes, x_i, g_i and J_i, respectively, where p is the number of processes used for the computation, x_i, g_i and J_i are assigned to the i^{th} process, and i = 1, 2, $\cdots$, p.

Calculating J is the most important computation in Newton's method. In order to calculate $J_{ij} = \frac{\partial f_i(\vec{x})}{\partial x_j}$ on the i^{th} process, the sub-vector x_j , where j $\neq$ i, need to be sent from the j^{th} process to the i^{th} process. Hence, both the j^{th} and i^{th} process hold a copy of x_j. Therefore, in parallel Newton's method, it is possible to maintain multiple copies of the same subvector of $\vec{x}$ in different processes without additional time overhead. If a sub-vector of $\vec{x}$ on the j^{th} process is lost, it is possible to recover x_j by getting it from the i^{th} process. When x_j is available, the relative J can be recovered through calculating.

Our scheme is a communication-induced checkpoint-free fault tolerance technique. The main procedure of FT-Newton with our scheme can be summarized in Algorithm 2.

A message-passing pattern which describes how different processes communicate with each other and how much data can be sent though a communication is important for applications. In many applications, Jacobian matrix is a sparse matrix; for example, J is a block diagonal matrix. Under this case, some processors' local data is enough for calculating their J. Thus communication-induced redundancies can not maintain enough amount of redundancy for the successful recovery. We should modify the message-passing pattern to force processes to send more information out, introducing additional overhead. In some special cases, additional overhead may be zero.

For example, assume that S_i denotes the local data on the process P_i, where $S_1 = \{x_1, x_2, x_3\}$, $S_2 = \{x_4, x_5, x_6\}$, $S_3 = \{x_7, x_8, x_9\}$. Each process needs to exchange some elements with the other two processes. In the original communication, we need 4 units time to send all needed elements, but one element of each process (i.e., x_3) does not be saved on other processes. In the fault tolerant communication, we also need 4 units time. However, we ensure that each element is saved on other processes. Therefore, all elements can be accurately reconstructed. Obviously, it is possible to

Algorithm 2 FT-Newton with our scheme

Require: $n > 0, \vec{x}, \varepsilon$;
  Construct process grid;
  Distribute J, $\vec{x}$, $\vec{v}$ and $\vec{g}$ onto the processes;
  if recover then
    Detect and locate fail-stop process failures with the aid of programming environment;
    The replacement of the failed process ask its neighbors to send i, $\vec{x}^{(i-1)}$, J_descrip, g_descrip and x_descrip;
    Calculate $\vec{g}$ and J;
    goto *while*;
  end if
  $\vec{x} = \vec{x_0}$;
  With the need of computation, each process exchanges data $\vec{x}$ with neighbors.
  Each process records the received messages which are the necessary recovery information for its neighbors.
  $\vec{g} = F(\vec{x})$ and $J = J(\vec{x})$;
  while $\|\vec{g}\| > \varepsilon$ **do**
    Solve $J\vec{v} = -\vec{g}$;
    With the need of computation, each process exchanges data $\vec{v}$ with neighbors.
    Each process records the received messages which are the necessary recovery information for its neighbors.
    $\vec{x} = \vec{x} + \vec{v}$;
    $\vec{g} = F(\vec{x})$;
    $J = J(\vec{x})$;
  end while

maintain the copies for each sub-vector of $\vec{x}$ in different processes without additional time overhead. As shown in Figure 3.3.

3.4 Failure Recovery

The main benefit of our scheme is the ability to tolerate fail-stop failures without either checkpointing or rollback recovery. This lowers the expected running time of the application in the presence of failures. Now we discuss the case where there is only one process failure. When a process failure occurs during the execution of the i^{th} iteration, the following five values and two expressions can be recovered: i, $\vec{x}^{(i-1)}$, J_descrip, g_descrip and x_descrip and $\vec{g}$, J, then the computation can be restarted from the i^{th} iteration.

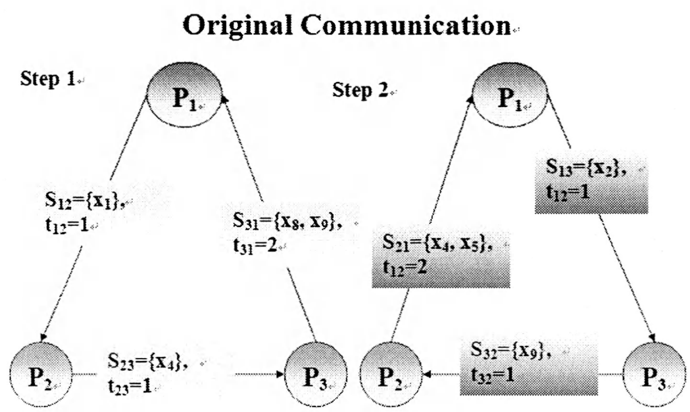
In the i^{th} iteration, if processes do not discard the messages they received in the last iteration, then all five values and two expressions that are necessary for restarting the i^{th} iteration can be recovered from the following steps:

1. The iteration loop variable i can be recovered by getting it from any surviving neighbor process.

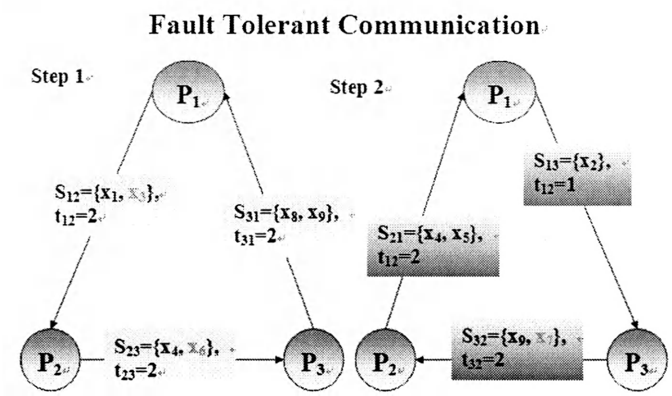
2. J_descrip, g_descrip and x_descrip are the elements of the array descriptor $DESC\_$ and can be recovered by getting them from any surviving neighbor process. Then we can rebuild the missing process grid and data distribution information, because of the characteristic of ScaLAPACK.

3. $\vec{x}^{(i-1)}$ can be recovered by communicating with any surviving neighbor process.

4. $\vec{g}$ and J can be reconstructed by using data dependencies.



(a) Original Communication: S_{ij} denotes the messages from the i^{th} process to j^{th} process, t_{ij} is the communication time between the i^{th} process and j^{th} process.



(b) Fault Tolerant Communication

Figure 3.3. Original Message-Passing Pattern v.s. Fault-Tolerance Message-Passing Pattern

Chapter 4

PERFORMANCE ANALYSIS

In this chapter, we analyze the expected parallel program execution time in a failure-prone environment and compare our scheme with checkpointing.

Assume a program takes T to finish in a failure-free environment. Further, assume that the system fails with a constant failure rate λ and the occurrence of failures follows a Poisson process. Then the probability density function (Young, 1974; Daly, 2006) is

$$f(t) = \lambda e^{-\lambda t} \tag{4.1}$$

4.1 Expected Program Execution Time without Fault-tolerant Technique

Let E_{non-ft} denote the expected program execution time without fault-tolerant techniques. When no fault-tolerant technique is used in a failure-prone environment, the probability that a failure will occur at the time τ in the execution is given by $\lambda e^{-\lambda\tau} d\tau$.

Without a fault-tolerant technique support, the application must roll back to the beginning when a failure occurs. If a failure occurs at τ, we would need to take τ to redo the lost calculations. The expected time under failure is

$$E_{non-ft} = e^{-\lambda T} \times T + \int_0^T (\tau + E_{non-ft}) \times (\lambda e^{-\lambda\tau} d\tau) \tag{4.2}$$

Therefore,

$$E_{non-ft} = \frac{e^{\lambda T} - 1}{\lambda} \tag{4.3}$$

The expression 4.3 indicates that the expected program execution time will increase almost exponentially as the failure rate increases. As the complexity of a computer system increases, its failure rate is drastically increased. Therefore, the system does not produce useful results in a reasonable time with a large number of failures.

4.2 Expected Program Execution Time for Checkpointing

Due to recovery, we need to save the current states of the processes into storage at pre-determined intervals. Each of the pre-determined intervals is called a checkpoint interval, denoted by I. Let E denote the expected execution time with checkpointing and $E_{interval}$ denote the expected execution time for an interval. Suppose the time (i.e., overhead) for one checkpoint is t_c. It takes t_r to load the the checkpoints from storages after a failure occurs. τ is the time required for redoing the work completed prior to the failure. Then

$$I = \frac{T}{N} + t_c \tag{4.4}$$

where N is the number of checkpoints.

Extreme scale systems need more frequent checkpointing for fault tolerance. Since writing checkpoints into storage is time consuming, checkpointing too frequently introduces unnecessary overhead. How frequently checkpointing should be taken is a very important question. Many researchers (Young, 1974; Daly, 2006; Bouguerra *et al.*, 2010) have proposed several models to approximate the optimal checkpoint interval to minimize the total application execution time.

Let I_{opt} denote the optimal checkpoint interval and E_{opt} denote the optimal expected program execution time with checkpoint. By "optimal", we mean the necessary number of checkpoints to obtain the theoretically smallest overhead (time of checkpointing plus time of recovery). We will discuss how to calculate I_{opt} in the following paragraph.

When a typical periodical checkpointing scheme is used in the application, we have the following equation to calculate $E_{interval}$.

$$\begin{aligned} E_{interval} &= e^{-\lambda I} \times I + \int_0^I (\tau + E_{interval} + t_r) \times (\lambda e^{-\lambda\tau} d\tau) \\ &= (e^{\lambda I} - 1)(\frac{1}{\lambda} + t_r) \\ &= (e^{\lambda\frac{T}{N} + \lambda t_c} - 1)(\frac{1}{\lambda} + t_r) \end{aligned} \tag{4.5}$$

Then we can calculate E by the expression 4.6.

$$E = N \times E_{interval} \tag{4.6}$$

As the expressions 4.5 and 4.6 stated, the change of N results in changes of $E_{interval}$ and E. In order to find the optimal number of checkpoints N, we minimize the expression 4.6.

$$\begin{aligned} \min_{\mathrm{N}}\{E\} &= \min_{\mathrm{N}}\{N \times E_{interval}\} \\ &= \min_{\mathrm{N}}\{N \times (e^{\lambda\frac{T}{N} + \lambda t_c} - 1)(\frac{1}{\lambda} + t_r)\} \end{aligned} \tag{4.7}$$

Let $\frac{\partial(E)}{\partial(N)} = (t_r + \frac{1}{\lambda}) \times [e^{\lambda(\frac{T}{N} + t_c)}(1 - \frac{\lambda T}{N}) - 1] = 0$, then we can find the minimum N, denoted by N_{opt}.

$$\begin{aligned}
e^{-\lambda t_c} &= e^{\lambda \frac{T}{N}}(1 - \frac{\lambda T}{N}) \\
e^{-\lambda t_c} &= (1 + \frac{1}{1!}\frac{\lambda T}{N} + \frac{1}{2!}(\frac{\lambda T}{N})^2 + \frac{1}{3!}(\frac{\lambda T}{N})^3 + \cdots)(1 - \frac{\lambda T}{N}) \\
e^{-\lambda t_c} &\approx 1 - (\frac{\lambda T}{N})^2 \frac{1}{2} \\
N_{opt} &\approx \frac{\lambda T}{\sqrt{2(1 - e^{-\lambda t_c})}}
\end{aligned} \tag{4.8}$$

So we can determine the value of I_{opt} by using the expression 4.4.

$$\begin{aligned}
I_{opt} &= \frac{T}{N_{opt}} + t_c \\
&= \frac{\sqrt{2(1 - e^{-\lambda t_c})}}{\lambda} + t_c
\end{aligned} \tag{4.9}$$

Therefore, the optimal expected program execution time becomes:

$$E_{opt} = \frac{\lambda T}{\sqrt{2(1 - e^{-\lambda t_c})}}(\frac{1}{\lambda} + t_r)(e^{\sqrt{2(1-e^{-\lambda t_c})} + \lambda t_c} - 1) \tag{4.10}$$

If λt_c is small, then

$$e^{\lambda \frac{T}{N} + \lambda t_c} = \frac{1}{1 - \frac{\lambda T}{N}} = \frac{N}{N - \lambda T} \tag{4.11}$$

In this case,

$$E_{opt} = \frac{\lambda T}{1 - \sqrt{2(1 - e^{-\lambda t_c})}}(\frac{1}{\lambda} + t_r) \tag{4.12}$$

When $E_{opt} < E_{non-ft}$, an application can benefit from checkpointing. Equation 4.12 and Equation 4.3 indicate that as T grows, E_{opt} increases more slowly than E_{non-ft}. In other words, checkpointing improves the applications expected running time when failures occur.

4.3 Expected Program Execution Time for Our Scheme

In this thesis, the proposed fault tolerance scheme is a checkpoint-free technique. When a process failure occurs, the overhead to recover from the failure is t_{rr}. Let $E_{checkpoint-free}$ denote the expected program execution time by using the proposed fault tolerance scheme. We assume no failure occurs during the recovery. The expected number of failures during the whole program execution is consequently λT. The total recovering time is $\lambda T t_{rr}$. Therefore, the total expected program execution time $E_{checkpoint-free}$ can be estimated by

$$\begin{aligned} E_{checkpoint-free} &= T + \lambda T t_{rr} \\ &= T(1 + \lambda t_{rr}) \end{aligned} \tag{4.13}$$

To improve the efficiency of parallel computing, our scheme should achieve fast self recovery through the computation period. From Equation 4.13, if $\lambda t_{rr} \ll 1$, $E_{checkpoint-free} \approx T$, which means that our scheme may introduce no additional overheard when no failures occur.

4.4 Our Scheme vs. Checkpointing

Traditional checkpointing and our scheme are used in parallel applications. As stated in Figure 4.1, with checkpointing support, the expected execution time linearly increases as λ increases, whereas it is almost flat as λ increases with our scheme.

Let $P_{overhead}$ denote the additional percentage of overhead introduced by checkpointing, then

$$\begin{aligned} P_{overhead} &= \frac{E_{opt} - E_{checkpoint-free}}{E_{checkpoint-free}} \\ &= \frac{\lambda}{1 - \sqrt{2(1 - e^{-\lambda t_c})}} \frac{1 + \lambda t_r}{1 + \lambda t_{rr}} - 1 \end{aligned} \tag{4.14}$$

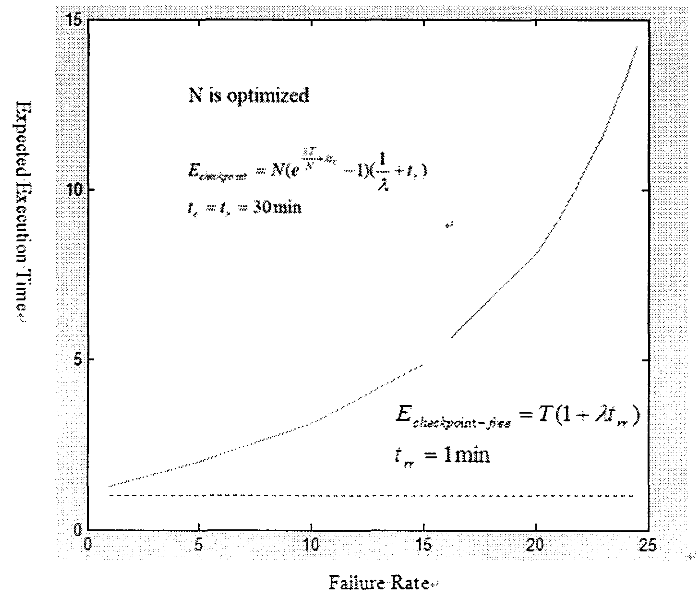


Figure 4.1. Expected Program Execution Time: Our Scheme v.s. Checkpointing

When $t_r \approx t_{rr}$,

$$P_{overhead} \approx \frac{\lambda}{1 - \sqrt{2(1 - e^{-\lambda t_c})}} - 1 \tag{4.15}$$

If $\lambda t_c \ll 1$, we obtain $1 - e^{-\lambda t_c} \approx \lambda t_c$. Therefore,

$$P_{overhead} \approx \frac{1}{1 - \sqrt{2\lambda t_c}} - 1 \tag{4.16}$$

Equation 4.16 indicates that as t_c increases, the fault tolerance overhead reduced by our scheme increases when λ is fixed.

Suppose that the system suffers two failures each day, a parallel application takes several days to complete and the time overhead for one checkpoint t_c is 5 minutes. Under this assumption, $\lambda t_c = 0.00694$ and $P_{overhead} \approx 13.35\%$. When $t_c = 10$ minutes, $\lambda t_c = 0.01389$ and $P_{overhead} \approx 20\%$.

Chapter 5

EXPERIMENTAL EVALUATION

In this chapter, we incorporate fault tolerance into Newton's method, experimentally evaluate the proposed algorithm-based recovery scheme and compare it with disk-based checkpointing and diskless checkpointing. We run all implementations, which are shown in Table 5.1, on the supercomputing cluster ra.mines.edu at Colorado School of Mines. Ra.mines.edu is a supercomputing cluster dedicated to energy sciences at Colorado School of Mines. The peak performance of the cluster is 23 teraflops. The MPI implementation on the cluster is Open MPI version 1.3.4.

Table 5.1. Implementations in Our Experiments

Abbreviation	Description
BASE	No fault-tolerant technique
Disk-based	Checkpointing to disk using MPI-I/O
Diskless	Simple diskless checkpointing
Our Scheme	An algorithm-based recovery scheme without checkpointing

The problem ARGTRIG from the CUTEr test set (Gould *et al.*, 2003) is a nonlinear problem involving coupled trigonometric equations. For demonstration purpose, consider solving the following problem ARGTRIG using parallel Newton's method:

$$F(\vec{x}) = \vec{0}, \quad \vec{x} = (x_1, x_2, \cdots, x_n)$$

where $F(\vec{x}) = (f_1(\vec{x}), f_2(\vec{x}), \cdots, f_n(\vec{x}))$ with

$$f_i(\vec{x}) = n - \sum_{j=1}^{n} cos(x_j) + i(1 - cos(x_i)) - sin(x_i) \tag{5.1}$$

The initial guess is $\vec{x_0} = (1/n, \cdots, 1/n)$. Note that this system has a full Jacobian.

5.1 Disk-based Checkpointing

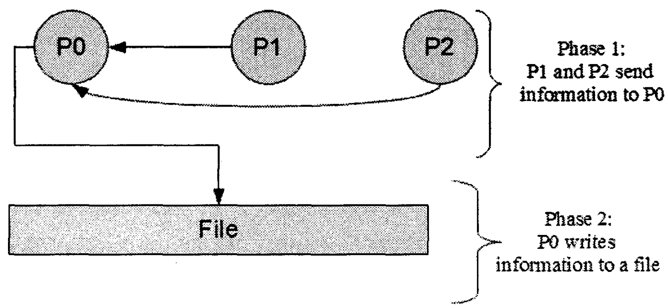
Many programming environments store their checkpoints on disks for fault-tolerance, such as Lipckpt (Plank *et al.*, 1995). I/O is an important part of disk-based checkpointing and the main source of overhead in checkpointing. There are three common ways of doing I/O in parallel programs:

1. Sequential I/O: all processes send their states to the master process (i.e., rank 0) and the master writes them to a unique file. Writing large amounts of data by a single process is cumbersome and slow. Sequential I/O becomes a bottleneck for disk-based checkpointing.

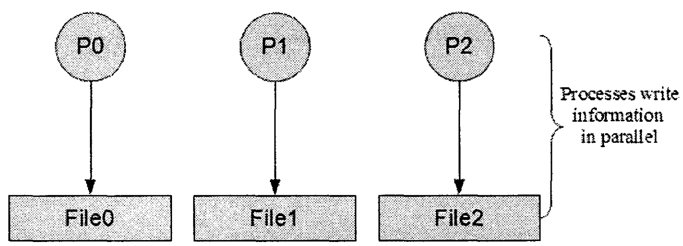
2. Simple parallel way: each process performs I/O on a separate file. There are lots of small files to manage. It is difficult to merge and split the global file.

3. Parallel I/O: multiple processes read data from or write output to a common file. Since some number or all of the processes simultaneously participate in an I/O operation, large amounts of data can be moved to/from disk quickly.

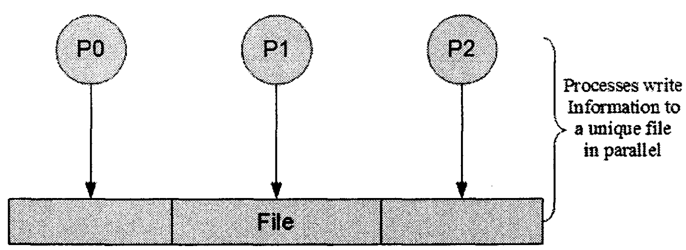
MPI-I/O (Gropp *et al.*, 1999) is a good implementation for parallel I/O and has over 55 calls related to file input and output, including collective I/O calls and non-blocking I/O calls. Processes can access a file independently. MPI-I/O can read/write noncontiguous data from a file to contiguous memory location. In order to achieve



(a) Sequential I/O: : $P0, P1$ and $P2$ are three processes.



(b) Simple parallel way



(c) Parallel I/O

Figure 5.1. Three Common Ways of Doing I/O in Parallel Programs

high performance, MPI-I/O writes as large of chunks as possible and reads/writes noncontiguous data.

In this thesis, we implement disk-based checkpointing by using MPI-I/O to tolerate fail-stop failures. For Newton's method, $\vec{x}$, $\vec{g}$ and J are updated during each iteration. So all memory is changed and we should save $\vec{x}$, $\vec{g}$ and J into a unique file to protect the application from inevitable failures.

5.2 Diskless Checkpointing

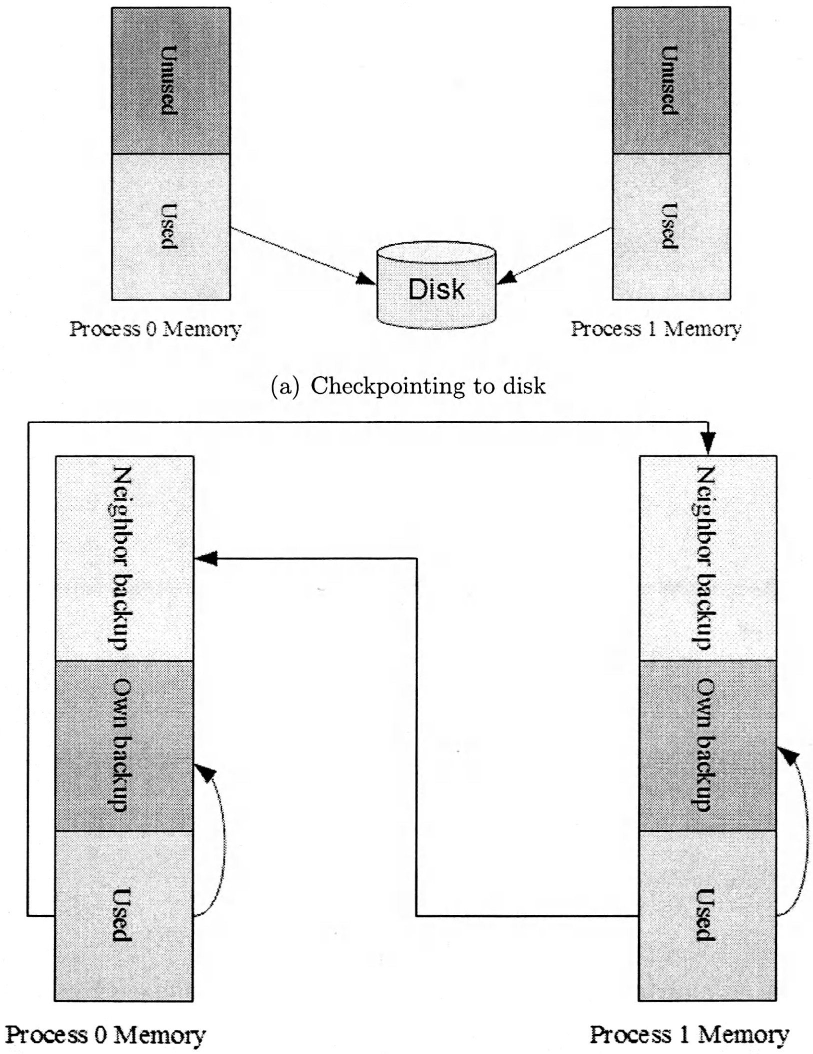
As shown in Figure 5.2 (a), in standard disk-based checkpointing, processes save contents of their address spaces to disk. With diskless checkpointing technique (Plank *et al.*, 1998), processes efficiently store checkpoints in local memories, not on disk. Extra memory usage is required. Since memory is not stable, diskless checkpointing cannot tolerate a power failure.

The simple diskless implementation, which is shown in 5.2 (b), is composed of two parts:

1. Saving the state of each process in its memory and

2. Communicating with its neighbor and storing the neighbor's state in memory. For example, the data of process $2i$ (i.e., $\vec{x}$, g and J of Newton's method) is saved on process $2i+1$. If process $2i$ dies, it can ask process $2i+1$ to send the backup data to recover its own lost data.

For our scheme, the failed process only requires its neighbor process to send $\vec{x}$, and then it can recover the related lost data.



(a) Checkpointing to disk

(b) Simple diskless checkpointing

Figure 5.2. Two Types of Checkpointing

5.3 Disk-based Checkpointing vs. Diskless Checkpointing

In this section, we compare the expected running times of the disk-based and diskless checkpointing.

Table 5.2. Some Parameters of Checkpointing

Checkpointing Method	t_c (sec)	t_r (sec)	t_{rb} (sec)
Disk-based	0.2434	0.0307	9.8674
Diskless	0.0742	0.0626	2.4668

To determine N_{opt} and I_{opt} by Equation 4.8 and Equation 4.9, we obtain the execution time T of an application in a failure-free environment and also the time to save a single checkpoint t_c by running the codes. t_r is the necessary time to load one checkpoint. After loading the checkpoint, the lost computation is redone. t_{rb} is the calculation time between the latest checkpoint and the failure point.

The maximum number of iterations is 500 and the size of variables n is 4000. The fixed number of processes is 4 and $T = 1233.42$ seconds. Some fixed parameters (t_c, t_r, t_{rb}) are shown in Table 5.2. The number of failures is $\lambda \times T$. In this set of experiments, we simulate a fail-stop failure during the execution. In Table 5.3 and Table 5.4, we use different values of λ to compare the performances of disk-based and diskless checkpointing.

T_{ckpt} is the execution time with checkpointing and T_{recv} is the execution time in the presence of failures, checkpointing, and recovery. *R_ckpt* is the checkpointing overhead ratio and *R_recv* is a measure of the performance penalty due to checkpointing, recovery and failures.

$$
\begin{aligned}
R\_ckpt &= \frac{T_{ckpt} - T}{T} \\
R\_recv &= \frac{T_{recv} - T}{T}
\end{aligned}
\tag{5.2}
$$

Table 5.3. Overhead Ratio of Disk-based Checkpointing

λ (sec^{-1})	N_{opt}	I_{opt} (sec)	R_ckpt	R_recv
2.00 $\times 10^{-4}$	25	49.58	0.49%	1.30%
3.67 $\times 10^{-4}$	35	35.48	0.67%	1.49%
7.37 $\times 10^{-4}$	50	24.89	1.01%	1.79%
9.56 $\times 10^{-4}$	55	22.67	1.04%	1.88%

Table 5.4. Overhead Ratio of Diskless Checkpointing

λ (sec^{-1})	N_{opt}	I_{opt} (sec)	R_ckpt	R_recv
2.00 $\times 10^{-4}$	62	19.97	0.37%	0.58%
3.67 $\times 10^{-4}$	83	14.93	0.50%	0.70%
7.37 $\times 10^{-4}$	125	9.94	0.75%	0.96%
9.56 $\times 10^{-4}$	167	7.46	1.04%	1.21%

Compared to the most traditional method — disk-based checkpointing, diskless checkpointing has a much smaller interval between checkpoints and a comparable overhead for taking checkpoints. From Table 5.2, the time for taking one checkpointing of disk-based checkpointing is slower than diskless checkpointing. By using diskless checkpointing, the user can checkpoint more frequently with a small performance penalty. In some cases (the first three rows of Table 5.3 and Table 5.4), diskless checkpointing outperforms disk-based checkpointing.

As stated in Table 5.3 and Table 5.4, the recovery overhead of diskless checkpointing is lower than disk-based checkpointing. Even with the same overhead of taking checkpoints, diskless checkpointing has a much smaller overhead for recovery. This fact is due to the computation in disk-based checkpointing that may return to much earlier states than states that the computation in diskless checkpointing may return to. For example, when $\lambda = 7.37 \times 10^{-4}\ sec^{-1}$, we simulate a fail-stop failure at iteration 245. In diskless checkpointing, the computation returns to the iteration 244. While in disk-based checkpointing, it returns to iteration 240.

5.4 Comparison

In order to assess the performance of our scheme as compared to the original code without fault-tolerance method, disk-based checkpointing and diskless checkpointing, we design some experiments under environments whether the fail-stop failures occur or not.

The number of iterations for all programs is 500. For Newton's method with checkpointing, we checkpoint every 10 iterations. The involved processes are organized into a two-dimensional process grid and the global matrix J, vectors $\vec{x}$, $\vec{g}$ are mapped onto the rectangular process grid by using the block-cyclic distribution scheme (details are shown in section 3.1).

We increase the total number of processes and the size of problem for computing but keep the same size of distributed data on each process in every experiment. Disk-based checkpointing requires extra disk usage to store checkpoints. Extra memory usage is needed for diskless checkpointing and our scheme. Figure 5.3 reports the storage overhead per process for all experiments. The diskless checkpointing has the largest memory overhead, while our scheme has a much smaller overhead, only 7.82 Kb.

5.4.1 Performance Overhead under Failure-Free Environment

The purpose of these experiments is to measure the performance penalty of taking fault tolerance techniques to survive the fail-stop failures. During the execution time,

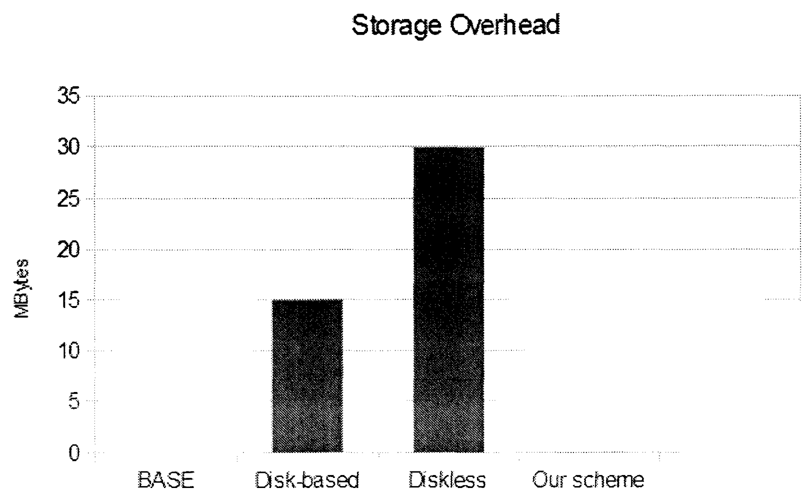


Figure 5.3. Storage Overhead per Process (Mb)

there are no failed processes. Table 5.5 reports the execution time of each test. We measure the checkpointing times separately to minimize the disturbance of the noise of the program execution time to the checkpoint time.

The individual checkpoint time is reported in Table 5.6. Figure 5.4 compares the time overhead (%) of different fault tolerance techniques for different sizes of problems.

Table 5.5. Execution Time (seconds) under Failure-Free Environment

num. of proc	n	BASE	Disk-based	Diskless	Our Scheme
4	4000	1233.42	1320.41	1244.08	1233.13
9	6000	2475.26	2631.14	2554.87	2477.46
16	8000	4703.78	5012.12	4974.17	4784.55
36	12000	13059.4	13673.0	13267.8	13026.9

From Table 5.6, we can see that, as the number of processes grows, the time for checkpointing increases. The increase of the checkpointing time leads to the increase

Table 5.6. One Checkpointing Time (seconds) under Failure-Free Environment

num. of proc	n	BASE	Disk-based	Diskless	Our Scheme
4	4000	0	0.2244	0.0744	0
9	6000	0	0.5597	0.1728	0
16	8000	0	1.0091	0.2847	0
36	12000	0	2.469	0.6379	0

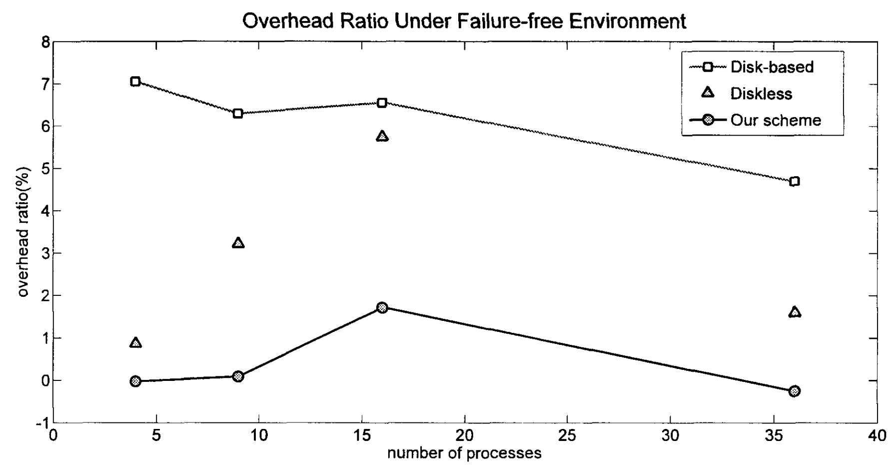


Figure 5.4. Time Overhead Ratio under Failure-Free Environment

of the checkpointing overhead. As stated in Table 5.5 and Figure 5.4, the performance of our scheme falls into the noise margin of the program execution time. In other words, our scheme introduces no overhead when there are no failures.

5.4.2 Performance Overhead of Performing Recovery

This subsection is designed to discuss the performance overhead to perform recovery. A process failure is simulated by deleting all data on its memory. For the disk-based or diskless checkpointing, a process failure recovery is simulated by two steps: 1) reading checkpoints from disk or memory and 2) rolling back to the most recent checkpoints. For our scheme, one recovery is reconstructing all the data necessary to continue the computation on the failed process by using data dependencies and communication-induced redundancies from where a failure occurs. For Newton's method with recovery, we simulate a process failure at iteration 245. During the recovery, the computation rolls back to iteration 240 by using checkpointing.

Table 5.7 reports the execution time of Newton's method with a single recovery. The individual recovery time is the time required for both loading the checkpoints and redoing the lost calculations in the event of a failure, which is reported in Table 5.8. Figure 5.5 plots the time overhead (%) for a recovery of different fault tolerance techniques for different sizes of problems.

Table 5.7. Execution Time (seconds) with Single Recovery

num. of proc	n	BASE	Disk-based	Diskless	Our Scheme
4	4000	1835.15	1335.78	1251.35	1233.99
9	6000	3854.96	2725.07	2696.7	2503.92
16	8000	7788.11	5186.71	5140.27	5022.27
36	12000	20804.9	13824.6	13384.4	13245

Table 5.8. Recover Time(seconds) from One Failure: the value in parenthesis is the time for loading the checkpoint.

num. of proc	n	BASE	Disk-based	Diskless	Our Scheme
4	4000	603.44	12.3460(0.03)	12.3755(0.0602)	0.1113
9	6000	1264.76	26.03(0.0705)	26.69(0.0957)	0.2205
16	8000	2585.96	45.55(0.1143)	52.61(0.1349)	0.3139
36	12000	6872	124.45(0.2638)	116.86(0.2735)	0.7089

Table 5.8 indicates that we need to take more time for a successful recovery when more processes are involved in the computation. Compared to checkpointing, our scheme takes more time to obtain information for further execution. As shown in Figure 5.5, we need at least 50% more time to complete the computation without any

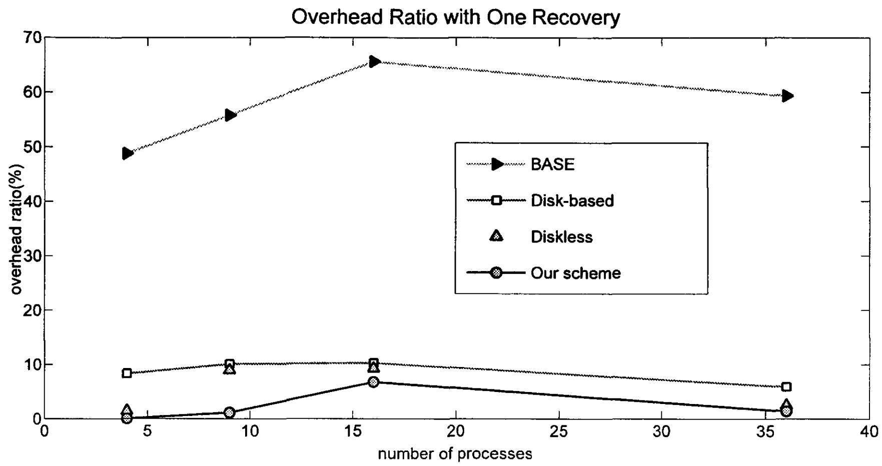


Figure 5.5. Time Overhead Ratio for Single Recovery

fault tolerance techniques. Due to rollback, checkpointing recovery takes much time to redo the computations which are done before the failure occurs. For our scheme, the computation is restarted from where the failure occurs. No rollback is needed. Therefore, overall, it introduces the least time overhead among the four different methods.

Multiple Recoveries The application suffers from multiple fail-stop failures throughout the period of execution time. There is only a single failed process when a fail-stop failure occurs. In this case, our scheme also tolerates single process failures.

In order to measure the impact of the number of failures, we designed a new set of experiments. The number of processes and the size of problem are fixed, which are 4 and 4000. We record the execution times and the recovery times with different numbers of fail-stop failures (1, 2, 4, 6, 8), as shown in Table 5.9 and Table 5.10.

Obviously, as the number of fail-stop failures increases, the performance overhead of performing recovery for the original implementation without any fault-tolerance

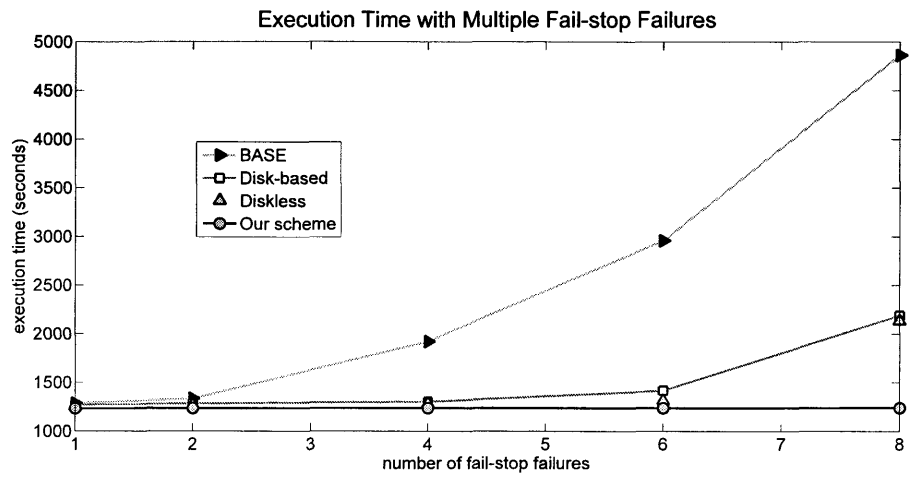
Table 5.9. Execution Time (seconds) with Multiple Recoveries

num. of failures	BASE	Disk-based	Diskless	Our Scheme
1	1285.18	1267.70	1251.35	1233.99
2	1334.73	1284.09	1263.08	1235.07
4	1921.60	1302.73	1283.42	1237.67
6	2958.30	1417.34	1311.48	1237.92
8	4862.94	2188.37	2134.53	1240.13

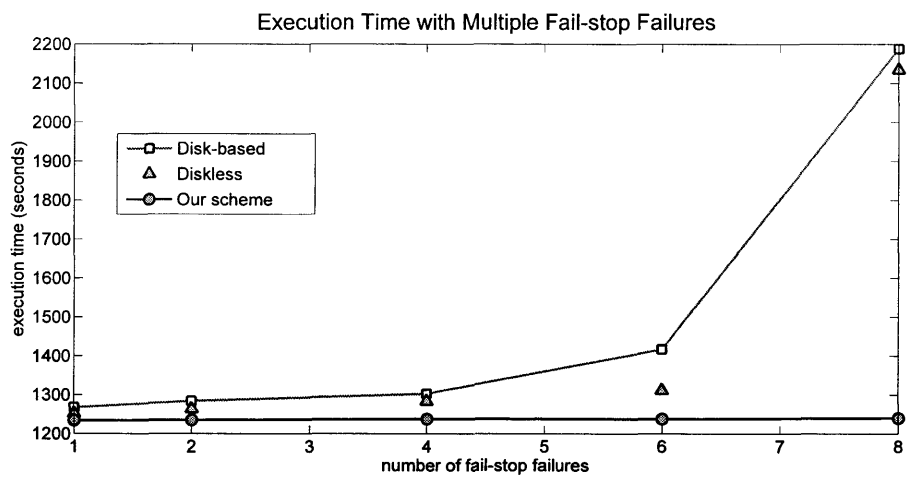
Table 5.10. Recover Time(seconds) from Multiple Failures: the value in parenthesis is the time for loading the multiple checkpoints.

num. of failures	BASE	Disk-based	Diskless	Our Scheme
1	41.903	13.1433(0.0306)	12.3731(0.0602)	0.1113
2	100.92	34.5878(0.0608)	24.7231(0.1158)	0.2170
4	303.18	49.6597(0.1275)	32.2037(0.2511)	0.4231
6	1119.75	91.4172(0.1833)	61.7632(0.3476)	0.6398
8	2625.61	200.5415(0.2735)	144.9899(0.5059)	1.6924

techniques gets worse. From Table 5.10, we can see the recovery time of checkpointing increases approximately linearly as the number of fail-stop failures increases. Compared to the execution time, the additional time for multiple recoveries introduced by our scheme can be ignored. As shown in Figure 5.6 and Table 5.9, our scheme reduces the fault tolerance overhead significantly.



(a)



(b)

Figure 5.6. Execution Time for Multiple Recoveries

Chapter 6

CONCLUSION

This thesis provides an algorithm-based recovery scheme for Newton's method to deal with fail-stop failures. Our scheme uses both data dependencies and communication-induced redundancies to tolerate fail-stop failures with low overhead. The parallel Newton's method maintains enough inherent redundant information for the accurate recovery of the lost data on the failed process. Therefore, there is no need to save as much information as checkpointing. The fault tolerant Newton's method with our scheme can restart the computation from where the failure occurs. In contrast, checkpointing can only return to the closest checkpoint when a failure occurs, and there is always an interval between the closest checkpoint and the failure point. This improvement leads to a more time-saving scheme compared to checkpointing. Experimental results show that our scheme introduces much less overhead than disk-based and diskless checkpointing. The results also demonstrate that our scheme is often highly scalable and has good potential to scale to extreme scale computing and beyond.

Our scheme can be used to add the fault-tolerance capability to many classical numerical algorithms. Generally speaking, these algorithms naturally contain relatively strong data dependencies. On the other hand, the parallel implementations often have communication-induced redundancies, which is also useful in our scheme. Both data dependencies and communication-induced redundancies provide room for algorithm improvements.

6.1 Limitations

Programmers are responsible for determining the data dependencies and communication-induced redundancies of a specified application. The performance of our scheme highly depends on the quality of the data dependencies and communication-induced redundancies discovered by programmers. To obtain the best performance, one must go to the deep inside of the original program to unveil the underlying data relationships. This is arduous work for programmers.

In addition, different numerical algorithms have different data dependencies. The experience obtained from former examples may be useless in handling the next problem. For example, the conjugate gradient method and Newton's method are two similar but different methods for solving the linear system of equations $Ax = b$. The conjugate gradient method solves the system by using the residual r^k at the kth iteration, where $r^k = b - Ax^k$. If we already have r^k, we can easily recover the corresponding x^k. Here, to recover r^k, processes should keep the messages they received in the last two iterations, which are the conjugate vectors p^{k-1}, p^{k-2} and the scalar β^{k-2}. r^k is recovered by using Equation 6.1 (Chen, 2011).

$$\begin{aligned} r^{k-2} &= \beta^{k-2} P^{k-2} - P^{k-1} \\ \beta^{k-2} &= \frac{(r^{k-1})^T r_{k-1}}{(r^{k-2})^T r^{k-2}} \\ r^k = r^{k-1} &- \frac{(r^{k-1})^T r^{k-1}}{(P^{k-1})^T A P^{k-1}} A P^{k-1} \end{aligned} \tag{6.1}$$

However, in Newton's method (details can be found in **Chapter 3**), the Jacobian matrix at the kth iteration J^k only relies on the variables x^k. The data dependency for recovering J^k is significantly different than that of the conjugate gradient method.

For some applications, researchers try to send a few messages to reduce the communication overhead so that communication-induced redundancies do not contain enough information for a recovery. In this case, the proposed scheme cannot achieve the best performance.

6.2 Future Work

Fault-tolerant sparse-matrix computation is more complicated and more useful in interdisciplinary research. For example, numerically solving Partial differential equations (PDE) often boils down to solve sparse linear system via discretizations. The matrices generated by discretizations are large and sparse in many common cases. They have relatively few nonzero entries, and these entries are usually located in a regular pattern on a few diagonals. The sparse matrix representation requires much less memory usage than the dense matrix representation. In order to determine the row and column positions of the nonzero matrix elements, the description of the sparsity must be encoded and stored. In the future, we may extend the technique to handle sparse matrix computation and integrate it with TOUGH2-MP to tolerate failures.

A major challenge exists in this extension. Memory and network performance limit the performance of sparse matrix computation. Since only a small constant number of arithmetic operations perform on communicated data, many researchers (Demmel *et al.*, 2008) try to minimize the communication between processes to improve the performance of sparse matrix computation. In this case, communication-induced redundancies may be limited. Thus, it may not contain enough information for the recovery. Our scheme needs to be improved in this aspect.

REFERENCES

Anfinson, C.J., & Luk, F.T. 1988. A Linear Algebraic Model of Algorithm-based Fault Tolerance. *IEEE Transactions on Computers*, **37**, 1599–1604.

Blackford, L.S., Choi, J., Cleary, A., D'Azevedo, E., Demmel, J., Dhillon, I., Dongarra, J., Hammarling, S., Henry, G., Petitet, A., Stanley, K., Walker, D., & Whaley, R. C. 1997. *ScaLAPACK Users' Guide.* SIAM.

Bouguerra, M.S., Trystram, D., & Wagner, F. 2010. *An Optimal Algorithm for Scheduling Checkpoints with Variable Costs.* INRIA, Research report.

Bronevetsky, G., Marques, D., Pingali, K., & Stodghill, P. 2003a. Automated Application-level Checkpointing of MPI Programs. *Pages 84–94 of: the ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.* ACM Press.

Bronevetsky, G., Marques, D., Pingali, K., & Stodghill, P. 2003b. C3: A System for Automating Application-level Checkpointing of MPI Programs. *Pages 357–373 of: the 16th International Workshop on Languages and Compilers for Parallel Computing, LCPC.*

Chen, Z. 2011. Algorithm-Based Recovery for Iterative Methods without Checkpointing. *In: Proceedings of the 20th ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC 2011).* ACM Press.

Chen, Z., & Dongarra, J. 2008. Algorithm-Based Fault Tolerance for Fail-Stop Failures. *IEEE Transactions on Parallel and Distributed Systems*, **19**, 1628–1641.

Daly, J.T. 2006. A Higher Order Estimate of the Optimum Checkpoint Interval for Restart Dumps. *Future Generation Computer Systems*, **22**, 303–312.

Demmel, J., Hoemmen, M., Mohiyuddin, M., & Yelick, K. 2008. Avoiding Communication in Sparse Matrix Computations. *In: IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2008.*

Dennis J. E., JR., & Schnabel, R. B. 1996. *Numerical Methods for Unconstrained Optimization and Nonlinear Equations.* SIAM.

Elnozahy, E.N., & Plank, J.S. 2004. Checkpointing for Peta-Scale Systems: A Look into the Future of Practical Rollback-Recovery. *IEEE Transactions on Dependable and Secure Computing*, **1**, 97–108.

Fagg, G. E., Gabriel, E., Bosilca, G., Angskun, T., Chen, Z., Pjesivac-Grbovic, J., London, K., & Dongarra, J. J. 2004. Extending the MPI Specification for Process Fault Tolerance on High Performance Computing Systems. *In: Proceedings of the International Supercomputer Conference.*

Gabriel, E., Fagg, G.E., Bosilca, G., Angskun, T., Dongarra, J.J., Squyres, J.M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., Castain, R.H., Daniel, D.J., Graham, R.L., & Woodall, T.S. 2004 (September). Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. *Pages 97–104 of: Proceedings, 11th European PVM/MPI Users' Group Meeting.*

Gelenbe, E. 1976. A Model of Roll-back Recovery with Multiple Checkpoints. *Pages 251–255 of: Proceedings of the 2nd International Conference on Software Engineering.*

Gould, N. I. M., Orban, D., & Toint, P. L. 2003. CUTEr and SifDec: A Constrained and Unconstrained Testing Environment, revisited. *ACM Trans. Math. Software*, **29**, 373–394.

Gropp, W., Lusk, E.L., & Thakur, R. 1999. *Using MPI-2: Advanced Features of the Message Passing Interface.* The MIT Press.

Huang, K., & Abraham, J.A. 1984. Algorithm-based Fault Tolerance for Matrix Operations. *IEEE Transactions on Computers*, **33**, 518–528.

Hwang, J.H., Balazinska, M., Rasin, A., Cetintemel, U., Stonebraker, M., & Zdonik, S. B. 2005. High-availability Algorithms for Distributed Stream Processing. *Pages 779–790 of: Proc. ICDE.*

Li, CC.J., Stewart, E.M., & Fuchs, W.K. 1994. Compiler-assisted Full Checkpointing. *Software: Practice and Experience*, **24**, 871–886.

Liu, H., Davies, T., Ding, C., Karlsson, C., & Chen, Z. 2011 (May). Algorithm-Based Recovery for Newtons Method without Checkpointing. *Pages 1536–1543 of: Proceedings of the 25th IEEE International Parallel & Distributed Processing Symposium (IPDPS 2011), DPDNS'11 Workshop.*

Michele, P., & Regis, L. 2007. Effective Checkpoint and Rollback Using Hardware/OS Collaboration. *Pages 370–378 of: 22nd IEEE International Symposium on Defect and Fault-Tolerance in VLSI Systems (DFT 2007).*

Moody, A., Bronevetsky, G., Mohror, K., & de Supinski, B.R. 2010. Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System. *In: Proceesings of the 2010 ACM/IEEE conference on Supercomputing.*

Plank, J. S., Li, K., & Puening, M. A. 1998. Diskless Checkpointing. *IEEE Trans. Parallel Distrib. Syst.*, **9**, 972–986.

Plank, J.S., Beck, M., Kingsley, G., & Li, K. 1995. Lipckpt: Transparent Checkpointing under UNIX. *In: Proc. USENIX Winter 1995 Technical Conf.*

Schlichting, R. D., & Schneider, F. B. 1983. Fail-stop Processors: an Approach to Designing Fault-tolerant Computing Systems. *ACM Transactions on Computer Systems*, **1**, 222–238.

Schroeder, B., & Gibson, G. A. 2006. A Large-Scale Study of Failures in High-Performance Computing Systems. *Pages 249–258 of: Proceedings of the International Conference on Dependable Systems and Networks.*

Snir, M., Otto, S., Lederman, S. Huss, Walker, D., & Dongarra, J. 1998. *MPI-The Complete Reference, Volume 1: The MPI Core, 2nd. (Revised) edition.* MIT Press.

Vaidya, N. H. 1994. *A Case for Multi-Level Distributed Recovery Schemes , Tech. Rep.* Ph.D. thesis, Texas A&M University.

Young, J.W. 1974. A First Order Approximation to the Optimum Checkpoint Interval. *Communications of the ACM*, **17**, 530–531.