To the Graduate Council:

I am submitting herewith a dissertation written by Zizhong Chen entitled "Scalable Techniques for Fault Tolerant High Performance Computing." I have examined the final electronic copy of this dissertation for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Doctor of Philosophy, with a major in Computer Science.

Jack J. Dongarra

Major Professor

We have read this dissertation
and recommend its acceptance:

James S. Plank

Shirley Moore

Ohannes Karakashian

Accepted for the Council:

Anne Mayhew

Vice Chancellor
and Dean of Graduate Studies

(Original signatures are on file with official student records.)

SCALABLE TECHNIQUES FOR FAULT TOLERANT HIGH PERFORMANCE COMPUTING

A Dissertation

Presented for the

Doctor of Philosophy Degree

The University of Tennessee, Knoxville

Zizhong Chen

May 2006

Dedication

This dissertation is dedicated to my dearest parents, Daide Chen and Yongqun Yang, and my loving wife, Shunlan Lu.

Acknowledgments

I would like to express my deepest gratitude to my advisor, Dr. Jack Dongarra, for leading me into the area of high performance computing and for his precious guidance, endless support, and valuable discussions throughout the process of this research. I am grateful to Dr. Dongarra for providing me generous financial support for this research and ample opportunities to share the research ideas in various conferences and workshops.

I would also like to express my appreciation to Dr. James Plank, Dr. Shirley Moore, and Dr. Ohannes Karakashian for serving on my graduate committee and for providing valuable comments and constructive suggestions towards improving the quality of this research.

I am grateful to all people who have helped me. I especially thank Graham Fagg, Edgar Gabriel, George Bosilca, Julien Langou, Piotr Luszczek, Kenneth Roche, Thara Angskun, Jelena Pjesivac-Grbovic, Zhiao Shi, Min Zhou, Yuanlei Zhang, Haihang You, and Fengguang Song for their valuable help and precious friendship.

I am extremely thankful to my wife, Shunlan Lu, for her endless love, patience, and understanding. Without her support, it would be impossible for me to complete this work. I could never thank my parents enough for their love, sacrifices, and encouragement that are crucial to the completion of my studies.

The author acknowledges the support of the research by the Los Alamos National Laboratory under Contract No. 03891-001-99 49 and the Applied Mathematical Sci-

ences Research Program of the Office of Mathematical, Information, and Computational Sciences, U.S. Department of Energy under contract DE-AC05-00OR22725 with UT-Battelle, LLC.

Abstract

As the number of processors in today's parallel systems continues to grow, the mean-time-to-failure of these systems is becoming significantly shorter than the execution time of many parallel applications. It is increasingly important for large parallel applications to be able to continue to execute in spite of the failure of some components in the system. Today's long running scientific applications typically tolerate failures by checkpoint/restart in which all process states of an application are saved into stable storage periodically. However, as the number of processors in a system increases, the amount of data that need to be saved into stable storage increases linearly. Therefore, the classical checkpoint/restart approach has a potential scalability problem for large parallel systems.

In this research, we explore scalable techniques to tolerate a small number of process failures in large scale parallel computing. The goal of this research is to develop scalable fault tolerance techniques to help to make future high performance computing applications self-adaptive and fault survivable. The fundamental challenge in this research is scalability. To approach this challenge, this research (1) extended existing diskless checkpointing techniques to enable them to better scale in large scale high performance computing systems; (2) designed checkpoint-free fault tolerance techniques for linear algebra computations to survive process failures without checkpoint or rollback recovery; (3) developed coding approaches and novel erasure correcting codes to help applications to survive multiple simultaneous process failures. The fault tolerance schemes we intro-

duce in this dissertation are scalable in the sense that the overhead to tolerate a failure of a fixed number of processes does not increase as the number of total processes in a parallel system increases.

Two prototype examples have been developed to demonstrate the effectiveness of our techniques. In the first example, we developed a fault survivable conjugate gradient solver that is able to survive multiple simultaneous process failures with negligible overhead. In the second example, we incorporated our checkpoint-free fault tolerance technique into the ScaLAPACK/PBLAS matrix-matrix multiplication code to evaluate the overhead, survivability, and scalability. Theoretical analysis indicates that, to survive a fixed number of process failures, the fault tolerance overhead (without recovery) for matrix-matrix multiplication decreases to zero as the total number of processes (assuming a fixed amount of data per process) increases to infinity. Experimental results demonstrate that the checkpoint-free fault tolerance technique introduces surprisingly low overhead even when the total number of processes used in the application is small.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

As the unquenchable desire of today's scientists to run ever larger simulations and analyze ever larger data sets drives the size of high performance computers from hundreds, to thousands, and even tens of thousands of processors, the mean-time-to-failure (MTTF) of these computers is becoming significantly shorter than the execution time of many current high performance computing applications.

Even making generous assumptions on the reliability of a single processor or link, it is clear that as the processor count in high end clusters grows into the tens of thousands, the mean-time-to-failure of these clusters will drop from a few years to a few days, or less. The current DOE ASCI computer (IBM Blue Gene L) is designed with 131,000 processors. The mean-time-to-failure of some nodes or links for this system is reported to be only six days on average [1].

In recent years, the trend of the high performance computing [13] has been shift-

ing from the expensive massively parallel computer systems to clusters of commodity off-the-shelf systems[13]. While commodity off-the-shelf cluster systems have excellent price-performance ratio, the low reliability of the off-the-shelf components in these systems leads a growing concern with the fault tolerance issue. The recently emerging computational grid environments [36] with dynamic resources have further exacerbated the problem.

However, driven by the desire of scientists for ever higher levels of detail and accuracy in their simulations, many computational science programs are now being designed to run for days or even months. Therefore, the next generation computational science programs need to be able to tolerate failures.

1.1 Problem Statement

Assume a computing system consists of many nodes connected by network connections. Each node has its own memory and local disk. There is at least one processor on each node and only one application process on each processor. The communication between processes is assumed to be message passing. Assume a process may fail due to the failure of a processor or many other reasons. We assume a *fail-stop* failure model: the failed process stops working and all data associated with the failed process are lost.

Although other types of failures exist, in this work we only consider this type of failure. This type of failure is common in today's large computing systems such as high-end clusters with thousands of nodes and computational grids with dynamic resources.

Today's long running scientific applications typically tolerate failures by checkpoint/restart approaches in which all process states of an application are periodically saved into stable storage. The advantage of this approach is that it is able to tolerate the failure of the whole system. However, in this approach, if one process fails, usually all surviving processes are aborted and the whole application is restarted from the last checkpoint.

The major source of overhead in all stable-storage-based checkpoint systems is the time it takes to write checkpoints to stable storage [53]. The checkpoint of an application on a, say, ten-thousand-processor computer implies that all critical data for the application on all ten thousand processors have to be written into stable storage periodically, which may introduce an unacceptable amount of overhead into the checkpointing system. The restart of such an application implies that all processes have to be recreated and all data for each process have to be re-read from stable storage into memory or re-generated by computation, which often brings a large amount of overhead into restart. It may also be very expensive or unrealistic for many large systems such as grids to provide the large amount of stable storage necessary to hold all process state of an application running on thousands of processes.

Furthermore, as the number of processors in the system increases, the total number of process states that need to be written into the stable storage also increases linearly. Therefore, the fault tolerance overhead increases linearly. Figure 1.1 shows how a typical checkpoint/restart approach works.

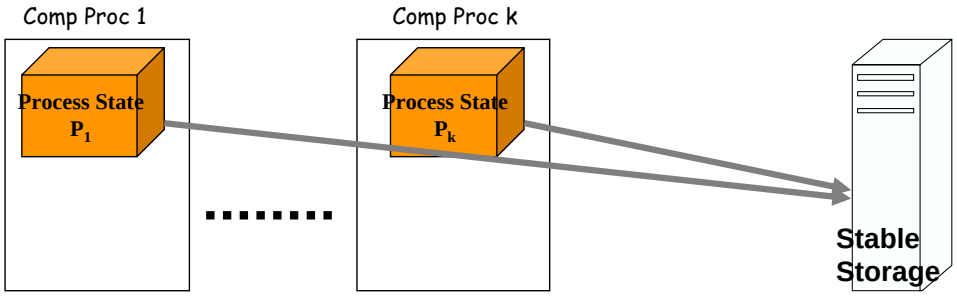


Figure 1.1: Tolerate failures by checkpoint/restart approach

Due to the high frequency of failures and the large number of processors in next generation computing systems, the classical checkpoint/restart fault tolerance approach may become a very inefficient way to handle failures. More scalable fault tolerance techniques need to be investigated.

Noting that today's high performance computing architectures are usually robust enough to survive partial node failures without suffering complete system failure, it is natural to ask: *can we tolerate partial node failures with lower overhead and better scalability ?*

The focus of this research is to investigate techniques to tolerate partial process failures in large high performance computing applications executing on high-end clusters and grids. The goal is to develop scalable fault tolerance techniques to help these applications to tolerate partial process failures in a scalable way. The fundamental challenge in this research is scalability.

1.2 Contributions

This dissertation develops several scalable fault tolerance techniques to tolerate partial process failures in large-scale parallel and distributed computing. The specific contributions this research makes can be summarized as follows

- **Scalable Checkpointing for Large Parallel Systems:** We introduce several new encoding strategies into the existing diskless checkpointing idea and reduce the overhead to tolerate k failures in p processes from $k(\beta+\gamma)m \ . \ \log p$ to $k(\beta+\gamma)m \ . \ (1+O(\frac{1}{\sqrt{m}})$, where $\frac{1}{\gamma}$ is the rate to perform summation, $\frac{1}{\beta}$ is the network bandwidth between processors, and m is the size of local checkpoint per processor. The introduced checkpoint schemes are scalable in the sense that the overhead to tolerate k failures in p processes does not increase as the number of processes p increases. We evaluate the performance overhead of our fault tolerance approach by using a preconditioned conjugate gradient equation solver as an example. Experimental results demonstrate that our fault tolerance approach can survive a small number of simultaneous processor failures with low performance overhead and little numerical impact.

- **Algorithm-Based Checkpoint-Free Fault Tolerance:** we explore an algorithm-based checkpoint-free fault tolerance approach in which, instead of taking checkpoint periodically, a coded global consistent state of the critical application data is maintained in memory by modifying applications to operate on encoded data. We show the practicality of this technique by applying it to the ScaLAPACK/PBLAS

matrix-matrix multiplication kernel which is one of the most important kernels for ScaLAPACK to achieve high performance and scalability.

- **Numerically Stable Real Number Codes:** We present a class of numerically stable real number erasure codes based on random matrices which can be used in the algorithm-based checkpoint-free fault tolerance technique to tolerate multiple simultaneous process failures. Experiment results demonstrate our codes are numerically much more stable than existing codes in the literature.

- **Condition Numbers of Gaussian Random Matrices:** Let $G_{m\times n}$ be an $m\times n$ real random matrix whose elements are independent and identically distributed standard normal random variables, and let $\kappa_2(G_{m\times n})$ be the 2-norm condition number of $G_{m\times n}$. We prove that, for any $m \geq 2$, $n \geq 2$ and $x \geq |n-m|+1$, $\kappa_2(G_{m\times n})$ satisfies $\frac{1}{\sqrt{2\pi}}\,(c/x)^{|n-m|+1} < P\left(\frac{\kappa_2(G_{m\times n})}{n/(|n-m|+1)} > x\right) < \frac{1}{\sqrt{2\pi}}\,(C/x)^{|n-m|+1}$, where $0.245 \leq c \leq 2.000$ and $5.013 \leq C \leq 6.414$ are universal positive constants independent of m, n and x. Moreover, for any $m \geq 2$ and $n \geq 2$, $E(\log \kappa_2(G_{m\times n})) < \log \frac{n}{|n-m|+1} + 2.258$. A similar pair of results for complex Gaussian random matrices is also established. These theoretical results demonstrate that the coding schemes in our algorithm-based checkpoint-free fault tolerance are numerically highly reliable.

1.3 Limitations of the Research

The size of the checkpoint affects the performance of any checkpointing scheme. The larger the checkpoint size is, the higher the checkpoint overhead will be. The overhead for checkpointing in this research increases linearly as the size of the checkpoint per process increases. Furthermore, the checkpointing in this research could not survive a failure of all processes. Also, to survive a failure occurring during checkpoint or recovery, the storage overhead would double. If an application needs to tolerate these types of failures, a two level recovery scheme [62] which uses both diskless checkpointing and stable-storage-based checkpointing is a good choice.

Compared with the typical checkpoint/restart approaches, the algorithm-based checkpoint-free fault tolerance in this dissertation can only tolerate partial process failures and only works for matrix computations. It needs support from programming environments to detect and locate failures. It requires the programming environment to be robust enough to survive node failures without suffering complete system failure. Both the overhead of and the additional effort to maintain a coded global consistent state of the critical application data in algorithm-based checkpoint-free fault tolerance is usually highly dependent on the specific characteristic of the application.

The real-number and complex-number codes proposed in the research are not perfect. Due to the probability approach we used, the drawback of our codes is that, no matter how small the probability is, there is a probability that a erasure pattern may not be able to be recovered accurately. An interesting open problem is how to construct the

numerically best codes over real-number and complex-number fields.

1.4 Dissertation Organization

The rest of this dissertation is organized as follows. Chapter 2 gives a brief review of the background as well as related work. Chapter 3 develops some scalable checkpointing strategies for large parallel systems. Chapter 4 explores some algorithm-based checkpoint-free fault tolerance techniques for high performance matrix computations. In Chapter 5, we address the practical numerical issue in algorithm-based checkpoint-free fault tolerance by proposing a class of numerically good real number erasure codes. In Chapter 6, we evaluate the condition numbers of Gaussian random matrices used in Chapters 4 and 5. Chapter 7 concludes the dissertation and discusses future work.

Chapter 2

Background

In this chapter, we briefly review related work in fault tolerant high performance computing and introduce the background of our research.

2.1 Related Work

Fault tolerance techniques can usually be divided into three big branches and some hybrid techniques. The first branch is *messaging logging*. In this branch, there are three sub-branches: *pessimistic messaging logging*, *optimistic messaging logging*, and *casual messaging logging*. The second branch is *checkpointing and rollback recovery*. There are also three sub-branches in this branch: *network disk based checkpointing and rollback recovery*, *diskless checkpointing and rollback recovery* , and *local disk based checkpointing and rollback recovery*. The third branch is *algorithm-based fault tolerance*.

There has been much work on fault tolerant techniques for high performance com-

puting. These efforts come in basically four categories

1. **System level checkpoint/message-logging** [18, 2, 47, 38, 40]: Most fault tolerance schemes in the literature belong to this category. The idea of this approach is to incorporate fault tolerance into the system level so that the application can be recovered automatically without any efforts from the application programmer. The most important advantage of this approach is its transparency. However, due to lack of knowledge about the semantics of the application, the system typically backs up all the processes and logs all messages, thus often introducing a huge amount of fault tolerance overhead.

2. **Compiler-based fault tolerance approach** [56, 9, 50]: The idea of this approach is to exploit the knowledge of the complier to insert the checkpoint at the best place and to exclude irrelevant memory areas to reduce the size of the checkpoint. This approach is also transparent. However, due to the inability of the compiler to determine the state of the communication channels at the time of the checkpoint, this approach is difficult to use in parallel/distributed applications that communicate through message passing.

3. **User-level Checkpoint Libraries** [61, 28, 5]: The idea of this approach is to provide some checkpoint libraries to the programmer and let the programmer decide where, when, and what to checkpoint. The disadvantage of this approach is its non-transparency. However, due to the involvement of the programmer in the checkpoint, the size of the checkpoint can be reduced considerably, and hence

the fault tolerance overhead can also be reduced considerably.

4. **Algorithmic fault tolerance approach** [35, 46, 34, 8]: The idea of this approach is to leverage the knowledge of algorithms to reduce the fault tolerance overhead to the minimum. In this approach, the programmer has to decide not only where, when, and what to checkpoint but also how to do the checkpoint, and hence the programmer must have deep knowledge about the application. However, if this approach can be incorporated into widely used application libraries such as ScaLAPACK and PETSc, then it is possible to reduce both the involvement of the application programmer and the overhead of the fault tolerance to a minimum.

Our research in this dissertation is mainly concentrated on incorporating fault tolerance into tightly coupled large scale high performance computation intensive applications. Because these applications are often communication intensive, checkpoint/rollback-recovery and algorithm-based fault tolerance approaches generally work better than message logging approaches.

In the rest of this section, we will confine our literature review mainly to checkpointing techniques and algorithm-based fault tolerance instead of general fault tolerance schemes.

2.1.1 Checkpointing and Rollback Recovery

Most traditional distributed multiprocessor recovery schemes are designed to tolerate an arbitrary number of failures. Hence they store their checkpoint data in a central

stable storage. The central stable storage usually has its own fault tolerance techniques to protect it from failures. But the bandwidth between the processors and the central stable storage is usually very low. Several experimental studies presented in [57] have shown that the main performance overhead of checkpointing is the time spent writing the checkpoint data to the central stable storage.

In [52, 53], Plank proposed to use diskless checkpointing technique as an approach to tolerant single failures with low performance overhead when stable storage is not available. Diskless checkpointing is a technique where processor redundancy, memory redundancy and failure coverage are traded off so that a checkpointing system can operate in the absence of stable storage. Experimental studies presented in [57] have shown that diskless checkpointing has much better performance than traditional disk based checkpoint techniques.

In [39], parity based diskless checkpointing is incorporated into several matrix operations with low performance overhead. In [39], the author also proposed to use checksum and reverse computation methods to tolerate single failures for some matrix operations to reduce the memory usage of the diskless checkpointing.

There are also several papers which compare the performance of different diskless checkpointing schemes. In [12], Chiueh compares the performance of different diskless checkpointing schemes on a massively parallel SIMD machine. Their experiments were performed on a DECmpp 12000 machine. The DECmpp 12000 machine has 8192 processors with each processor owning 64 Kbytes of RAM, but without any local disk for

each processor. They implemented three chechpointing schemes (checkpoint mirroring, parity checkpointing and partial party checkpointing) for a matrix-matrix multiplication application. The checkpoint procedure itself is fault tolerant in their implementation. The XOR operation was performed following an O(logN) binary tree fashion. Their experiment result shows that the checkpoint mirroring scheme is an order of magnitude faster than the parity checkpointing scheme, but introduces twice as much memory overhead as the parity checkpointing scheme. In [57], Silva also did some experimental studies about diskless checkpointing The experiments were done on an Xplorer Parsytec machine with 8 transputers (T805). Their experimental results show that the checkpoint mirroring has much better performance than the n+1 parity scheme. But the checkpoint mirroring scheme always presents more memory overhead than the n+1 parity scheme. In [53], Plank also report that the checkpoint mirroring scheme has lower performance overhead than the parity scheme if the checkpoint data is stored on local disk instead of the memory of a processor.

Local disk can also be used to store the checkpoint data. In [48], Plank applies RAID strategies to deal with local disk checkpoint data so that his checkpoint strategies can yield better performance for a smaller amount of fault coverage than traditional disk based checkpointing. In his paper, coordinated checkpoints are first taken to the local disk of each processor and then checkpointing mirroring, $n+1$ parity, or Reed-Solomon Coding are used to encode the local checkpoint data to the local disk of other processors. This strategy uses the local disk to replace the memory to tolerate small process failures,

thus achieving low checkpoint overhead when there is not enough memory to do diskless checkpoint.

To tolerate an arbitrary number of failures with low performance overhead, in [62], Vaidya proposed a two-level distributed recovery approach. A two-level recovery scheme tolerates the more probable failures with low performance overhead, while less probable failures maybe tolerated with a higher performance overhead. In his example, the more probable single failures are tolerated with diskless checkpointing (checkpoint mirroring), while the less probable multiple failures are tolerated with traditional disk based checkpointing. In that example, he demonstrated that to minimize the average overhead, it is often necessary to take both diskless checkpoints and disk based checkpoints.

Checkpoint can be done either at the system-level or at the application level. In [57], Silva compared the performance overhead of system-level checkpointing and user defined checkpointing. Their experiments were done on an Xplorer Parsytec machine with 8 transputers (T805). The experiments showed that user defined checkpointing schemes have much lower performance overhead than system-level checkpointing schemes. But the degree of the performance improvement is dependent on specific applications.

2.1.2 Algorithm-Based Fault Tolerance

Algorithm-based fault tolerance (ABFT), which was originally developed by Huang and Abraham [35], is a low-cost fault tolerance scheme to detect and correct permanent and transient errors in certain matrix operations on systolic arrays. The key idea of the ABFT technique is to encode the data at a higher level using checksum schemes and to

redesign algorithms to operate on the encoded data. One of the most important characteristics of algorithm-based fault tolerance is that it assumes a fail-continue model in which failed processors continue to work but produce incorrect calculations. Therefore, ABFT has to address the issue of error detection, location and correction.

Various checksum codes [45] have been suggested for fault tolerant matrix computation on processor arrays. However, due to the potential round-off, overflow, and underflow errors, the use of these codes has been limited. In floating point arithmetic, where no computation is exact, it is difficult to distinguish errors resulting from faulty hardware from errors resulting from round-off errors. Previous checksum codes in the literature are also quite suspect in their numerical stability when correcting multiple failures.

2.1.3 Naturally Fault Tolerant Algorithms

This class of approaches considers the specific characteristic of an application and designs fault tolerance schemes according to the specific characteristic of an application.

In [27], Geist et. el. investigated the natural fault tolerance concept in which the application can finish the computation task even if a small amount of application data are lost, therefore, checkpoint can be avoided for this type of applications. The authors try to establish a theoretical foundation for a new class of algorithms called super-scalable algorithms that have the properties of scale invariance and natural fault tolerance. Scale invariance means that the individual tasks in the larger parallel job have a fixed number of other tasks with which they communicate independent of the number

of tasks in the application. Finite difference algorithm is one such example. What scale invariance does is to isolate the failure and not make it a property of the total number of tasks. Fault tolerance can then be handled locally by self healing or natural fault tolerance. A parallel algorithm has natural fault tolerance if it is able to get the correct answer despite the failure of some tasks during the calculation. For example, an iterative algorithm may still converge despite lost information. It is not that the calculation are taken over by other tasks, but rather that the nature of the algorithm is that there is a natural compensation for the lost information. If an algorithm is naturally fault tolerant, then failure recovery (meaning data recovery) can be avoided. However, failure detection and notification are still needed to inform the algorithm to adapt. In [8], a checkpoint-free scheme is given for iterative methods. In [34], a checkpoint-free scheme is incorporated into a parallel direct search application.

2.2 Failure Model

To define the problem we are targeting and clarify the differences with traditional fault tolerance approaches, in this section we specify the type of failures we are focusing on.

We assume our target computing systems have many nodes which are connected by network connections. Each node has its own memory and local disk. There is at least one processor on each node and only one application process on each processor. Assume the target application is optimized to run on a fixed number of processes. Unlike in traditional algorithm-based fault tolerance which assumes a failed process continues to

work but produce incorrect results, in this work we assume a *fail-stop* failure model. In a fail-stop failure model, the failed process is assumed to stop working and all data associated with the failed process are assumed to be lost. The surviving processes can neither send nor receive any message from the failed processes.

2.3 FT-MPI

Current parallel programming paradigms for high-performance distributed computing systems are typically based on the Message-Passing Interface (MPI) specification [44]. However, the current MPI specification does not specify the behavior of an MPI implementation when one or more process failures occur during runtime. MPI gives the user the choice between two possibilities of how to handle failures. The first one, which is the default mode of MPI, is to immediately abort all the processes of the application. The second possibility is just slightly more flexible, handing control back to the user application without guaranteeing that any further communication can occur.

2.3.1 FT-MPI Overview

FT-MPI [20] is a fault tolerant version of MPI that is able to provide basic system services to support fault survivable applications. FT-MPI implements the complete MPI-1.2 specification, some parts of the MPI-2 document and extends some of the semantics of MPI for allowing the application the possibility to survive process failures. FT-MPI can survive the failure of n-1 processes in a n-process job, and, if required, can

re-spawn the failed processes. However, the application is still responsible for recovering the data structures and the data of the failed processes.

Although FT-MPI provides basic system services to support fault survivable applications, prevailing benchmarks show that the performance of FT-MPI is comparable [21] to the current state-of-the-art MPI implementations.

2.3.2 FT-MPI Semantics

FT-MPI provides semantics that answer the following questions:

1. what is the status of an MPI communicator after recovery?
2. what is the status of ongoing communication and messages during and after recovery?

When running an FT-MPI application, there are two parameters used to specify which modes the application is running.

The first parameter, the 'communicator mode', indicates the status of an MPI object after recovery. FT-MPI provides four different communicator modes, which can be specified when starting the application:

- ABORT: like any other MPI implementation, FT-MPI can abort on an error.
- BLANK: failed processes are not replaced, all surviving processes have the same rank as before the crash and MPI_COMM_WORLD has the same size as before.

- SHRINK: failed processes are not replaced, however the new communicator after the crash has no 'holes' in its list of processes. Thus, processes might have a new rank after recovery and the size of MPI_COMM_WORLD will change.

- REBUILD: failed processes are re-spawned, surviving processes have the same rank as before. The REBUILD mode is the default, and the most used mode of FT-MPI.

The second parameter, the 'communication mode', indicates how messages, which are on the 'fly' while an error occurs, are treated. FT-MPI provides two different communication modes, which can be specified while starting the application:

- CONT/CONTINUE: all operations which returned the error code MPI_SUCCESS will finish properly, even if a process failure occurs during the operation (unless the communication partner has failed).

- NOOP/RESET: all pending messages are dropped. The assumption behind this mode is, that on error the application returns to its last consistent state, and all currently pending operations are not of any further interest.

2.3.3 FT-MPI Usage

Handling fault-tolerance typically consists of three steps: 1) failure detection, 2) notification, and 3) recovery. The only assumption the FT-MPI specification makes about the first two points is that the run-time environment discovers failures and all remaining processes in the parallel job are notified about these events. The recovery procedure

is considered to consist of two steps: recovering the MPI library and the run-time environment, and recovering the application. The latter one is considered to be the responsibility of the application. In the FT-MPI specification, the communicator-mode discovers the status of MPI objects after recovery, and the message-mode ascertains the status of ongoing messages during and after recovery. FT-MPI offers for each of these modes several possibilities. This allows application developers to take the specific characteristics of their application into account and use the best-suited method to handle fault-tolerance.

Chapter 3

Scalable Checkpointing for Large Parallel Systems

Today's parallel architectures are usually robust enough to survive node failures without suffering complete system failure. However, most of today's high performance computing applications can not survive node failures and, therefore, whenever there is a node failure, have to abort themselves and restart from the beginning or a stable-storage-based checkpoint.

In this chapter, we explore how to build fault survivable high performance computing applications with FT-MPI using diskless checkpointing so that these applications can tolerate partial process failures with lower overhead and better scalability than using the traditional checkpoint/restart approach. We analyze existing diskless checkpointing techniques and introduce several new encoding strategies into diskless checkpointing to

improve the scalability of the techniques. We give a detailed presentation on how to write a fault survivable application with FT-MPI using diskless checkpointing and evaluate the performance overhead of our fault tolerance approach by using a preconditioned conjugate gradient equation solver as an example. Experimental results demonstrate that our fault tolerance approach can survive a small number of simultaneous processor failures with low performance overhead.

3.1 Diskless Checkpointing

Diskless checkpointing [53] is a technique to save the state of a long running computation on a distributed system without relying on stable storage. With diskless checkpointing, each processor involved in the computation stores a copy of its state locally, either in memory or on local disk. Additionally, encodings of these checkpoints are stored in local memory or on local disk of some processors which may or may not be involved in the computation. When a failure occurs, each live processor may roll its state back to its last local checkpoint, and the failed processor's state may be calculated from the local checkpoints of the surviving processors and the checkpoint encodings. By eliminating stable storage from checkpointing and replacing it with memory and processor redundancy, diskless checkpointing removes the main source of overhead in checkpointing on distributed systems [53]. Figure 3.1 is an example of how diskless checkpoint works.

To make diskless checkpointing as efficient as possible, it can be implemented at the application level rather than at the system level [51]. There are several advantages to

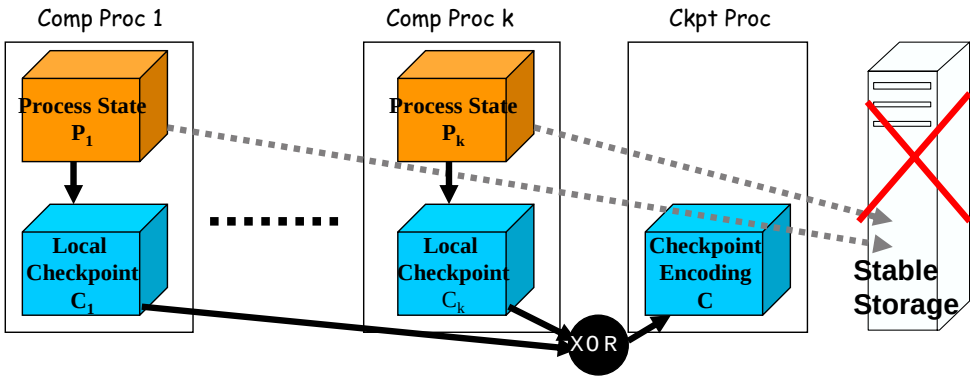


Figure 3.1: Tolerate failures by diskless checkpointing

implement checkpointing at the application level. Firstly, the application level checkpointing can be placed at synchronization points in the program, which achieves checkpoint consistency automatically. Secondly, with the application level checkpointing, the size of the checkpoint can be minimized because the application developers can restrict the checkpoint to the required data. This is opposed to a transparent checkpointing system which has to save the whole process state. Thirdly, the transparent system level checkpointing typically writes binary memory dumps, which rules out a heterogeneous recovery. On the other hand, application level checkpointing can be implemented such that the recovery operation can be performed in a heterogeneous environment as well.

In typical long running scientific applications, when diskless checkpointing is taken from application level, what needs to be checkpointed is often some numerical data [39]. These numerical data can either be treated as bit-streams or as floating-point numbers. If the data are treated as bit-streams, then bit-stream operations such as parity can be used to encode the checkpoint. Otherwise, floating-point arithmetic such as addition

can be used to encode the data.

However, compared with treating checkpoint data as numerical numbers, treating them as bit-streams usually has the following disadvantages:

1. To survive general multiple process failures, treating checkpoint data as bit-streams often involves the introduction of Galois Field arithmetic in the calculation of checkpoint encoding and recovery decoding [49]. If the checkpoint data are treated as numerical numbers, then only floating-point arithmetic is needed to calculate the checkpoint encoding and recovery decoding. Floating-point arithmetic is usually simpler to implement and more efficient than Galois Field arithmetic.

2. Treating checkpoint data as bit-streams rules out a heterogeneous recovery. The checkpoint data may have different bit-stream representation on different platforms and even have different bit-stream length on different architectures. The introduction of a unified representation of the checkpoint data on different platforms within an application for checkpoint purposes scarifices too much performance and is unrealistic in practice.

3. In some cases, treating checkpoint data as bit-streams does not work. For example, in [39], in order to reduce memory overhead in fault tolerant dense matrix computation, no local checkpoints are maintained on computation processors, only the checksums of the local checkpoints are maintained on the checkpoint processors. Whenever a failure occurs, the local checkpoints on surviving computation processors are re-constructed by reversing the computation. Lost data on failed

processors are then re-constructed through the checksum and the local checkpoints obtained from the reverse computation. However, due to round-off errors, the local checkpoints obtained from reverse computation are not the same bit-streams as the original local checkpoints. Therefore, in order to be able to re-construct the lost data on failed processors, the checkpoint data have to be treated as numerical numbers and floating point arithmetic has to be used to encode the checkpoint data.

The main disadvantage of treating the checkpoint data as floating-point numbers is the introduction of round-off errors into the checkpoint and recovery operations. Round-off error is a limitation of any floating-point number calculation. Even without checkpoint and recovery, scientific computing applications are still affected by round-off errors. In practice, the increased possibility of overflows, underflows, and cancellations due to round-off errors in numerically stable checkpoint and recovery algorithms is often negligible.

In this dissertation, we explore the possibility of treating the checkpoint data as floating-point numbers rather than bit-streams. However, the corresponding bit-stream version schemes could also be used if the the application programmer thinks they are more appropriate. In the following subsection, we discuss how the local checkpoint can be encoded so that applications can survive single process failure.

3.1.1 Checksum-Based Checkpointing

The checksum-based checkpointing is a floating-point version of the parity-based checkpointing scheme proposed in [52]. In the checksum-based checkpointing, instead of using parity, floating-point number addition is used to encode the local checkpoint data. By encoding the local checkpoint data of the computation processors and sending the encoding to some dedicated checkpoint processors, the checksum- based checkpointing introduces a much lower memory overhead into the checkpoint system than neighbor-based checkpoint. However, due to the calculating and sending of the encoding, the performance overhead of the checksum-based checkpointing is usually higher than neighbor-based checkpoint schemes. There are two versions of the checksum-based checkpointing schemes.

The basic checksum scheme works as follow. If the program is executing on N processors, then there is an $N+1$-st processor called the checksum processor. At all points in time a consistent checkpoint is held in the N processors in memory. Moreover a checksum of the N local checkpoints is held in the checksum processor. Assume P_i is the local checkpoint data in the memory of the i-th computation processor. C is the checksum of the local checkpoints in the checkpoint processor. If we look at the checkpoint data as an array of real numbers, then the checkpoint encoding actually establishes an identity (3.1) between the checkpoint data P_i on computation processors and the checksum data C on the checksum processor. If any processor fails, then the identity (3.1) becomes an equation with one unknown. Therefore, the data in the failed

processor can be reconstructed through solving this equation.

$$P_1 + \ldots + P_n = C \tag{3.1}$$

Due to the floating-point arithmetic used in the checkpoint and recovery, there will be round-off errors in the checkpoint and recovery. However, the checkpoint involves only additions and the recovery involves additions and only one subtraction. In practice, the increased possibility of overflows, underflows, and cancellations due to round-off errors in the checkpoint and recovery algorithm is negligible.

The basic checksum scheme can survive only one failure. However, it can be used to construct a one-dimensional checksum scheme to survive certain multiple failures.

3.1.2 Overhead and Scalability Analysis

Assume diskless checkpointing is performed in a parallel system with p processors and the size of checkpoint on each processor is m bytes. It takes $\alpha+\beta x$ to transfer a message of size x bytes between two processors regardless of which two processors are involved and. α is often called latency of the network. $\frac{1}{\beta}$ is called the bandwidth of the network. Assume the rate to calculate the sum of two arrays is γ seconds per byte. We also assume that it takes $\alpha+\beta x$ to write x bytes of data into the stable storage. Our default network model is the duplex model where a processor is able to concurrently send a message to one partner and receive a message from a possibly different partner. The more restrictive simplex model permits only one communication direction per processor.

We also assume that disjoint pairs of processors can communicate each other without interference each other.

By simply organizing all processors as a binary tree and sending local checkpoints along the tree to the checkpoint processor (see Figure 3.2) [53], the time to perform one checkpoint, $T_{diskless-binary}$, can be represented as

$$T_{diskless-binary} = 2\lceil \log p \rceil \ . \ m(\beta + \gamma) + 2\lceil \log p \rceil \ . \ \alpha.$$

Note that, in a typical checkpoint/restart approach (see Figure 1.1), the time to perform one checkpoint, $T_{checkpoint/restart}$, is

$$T_{checkpoint/restart} = p \ . \ m\beta + p \ . \ \alpha.$$

Therefore, by eliminating stable storage from checkpointing and replacing it with memory and processor redundancy, diskless checkpointing improves the scalability of checkpointing greatly on parallel and distributed systems.

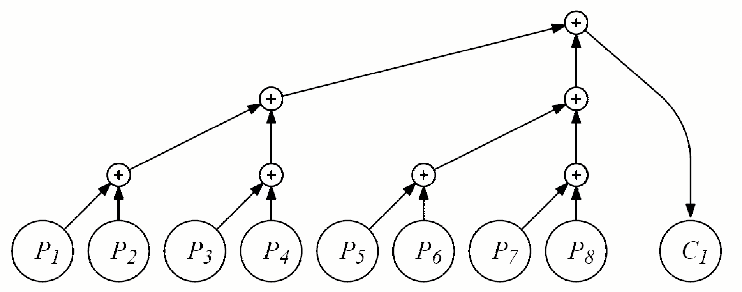


Figure 3.2: Encoding local checkpoints using the binary tree algorithm

3.2 A Scalable Algorithm for Checkpoint Encoding

Although the existing diskless checkpointing technique improves the scalability of checkpointing dramatically on parallel and distributed systems, the overhead to perform one checkpoint still increases quickly ($T_{diskless-binary} = 2\lceil \log p \rceil (\alpha + \beta m + \gamma m)$) as the number of processors increases. In this section, we propose a new style of encoding algorithm which improves the scalability of diskless checkpointing significantly. The new encoding algorithm is based on the pipeline idea.

3.2.1 Pipelining

The key idea of pipelining is (1) the segmenting of messages and (2) the simultaneous non-blocking transmission and receipt of data. By breaking up a large message into smaller segments and sending these smaller messages through the network, pipelining allows the receiver to begin forwarding a segment while receiving another segment.

Data pipelining can produce several significant improvements in the process of checkpoint encoding. First, pipelining masks the processor and network latencies that are known to be an important bottleneck in high-bandwidth local area networks. Second, it allows the simultaneous sending and receiving of data, and hence exploits the full duplex nature of the interconnect links in the parallel system.

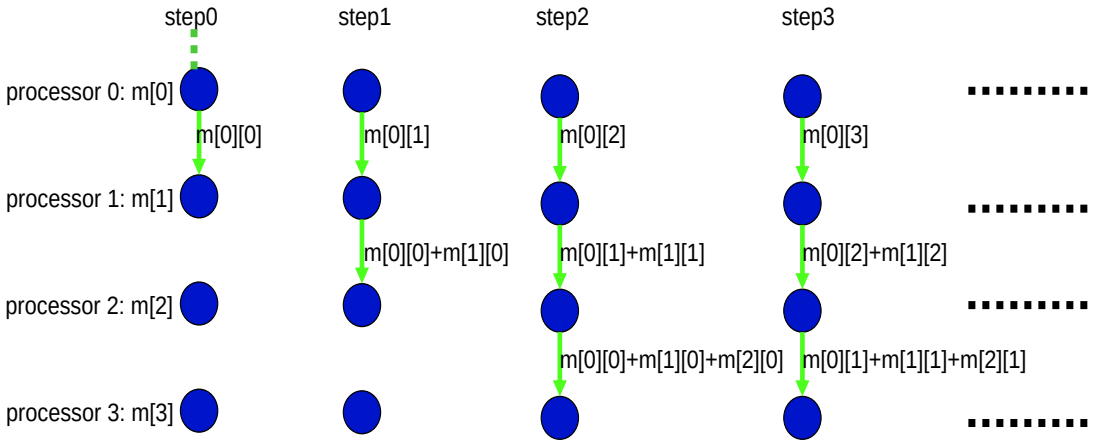


Figure 3.3: Chain-pipelined encoding for diskless checkpointing

3.2.2 Chain-pipelined encoding for diskless checkpointing

Let $m[i]$ denote the data on the i^{th} processor. The task of checkpoint encoding is to calculate the encoding which is $m[0] + m[1] + ... + m[p-1]$ and deliver the encoding to the checkpoint processor.

The chain-pipelined encoding algorithm works as follows. First, organize all computational processors and the checkpoint processor as a chain. Second, segment the data on each processor into small pieces. Assume the data on each processor are segmented into t segment of size s. The j^{th} segment of $m[i]$ is denoted as $m[i][j]$. Third, $m[0]+m[1]+...+m[p-1]$ are calculated by calculating $m[0][j]+m[1][j]+...+m[p-1][j]$ for each $0 \leq j \leq t-1$ in a pipelined way. Fourth, when the j^{th} segment of encoding $m[0][j]+m[1][j]+...+m[p-1][j]$ is available, start to send it to the checkpoint processor.

Figure 3.3 demonstrates an example of calculating a chain-pipelined checkpoint en-

coding for three processors (processor 0, processor 1, and processor 2) and deliver it to the checkpoint processor (processor 3). In step 0, processor 0 sends its $m[0][0]$ to processor 1. Processor 1 receives $m[0][0]$ from processor 0 and calculates $m[0][0] + m[1][0]$. In step1, processor 0 sends its $m[0][1]$ to processor 1. Processor 1 first concurrently receives $m[0][1]$ from processor 0 and sends $m[0][0] + m[1][0]$ to processor 2 and then calculates $m[0][1] + m[1][1]$. Processor 2 first receives $m[0][0] + m[1][0]$ from processor 1 and then calculate $m[0][0] + m[1][0] + m[2][0]$. As the procedure continues, at the end of step2, the checkpoint processor will be able to get its first segment of encoding $m[0][0] + m[1][0] + m[2][0] + m[3][0]$. From now on, the checkpoint processor will be able to receive a segment of the encoding at the end of each step. After the checkpoint processor receives the last checkpoint encoding, the checkpoint is finished.

3.2.3 Overhead and Scalability Analysis

In the chain-pipelined checkpoint encoding, the time for each step is $T_{each-step} = \alpha + \beta s + \gamma s$. The number of steps to encode and deliver t segments in a p processor system is $t + p - 1$. If we assume the size of data on each processor is m $(= ts)$, then the total time for encoding and delivery is

$$T_{total}(s) = (p - 1 + t)(\alpha + \beta s + \gamma s)$$

Note that

$$T''_{total}(s) \geq 0, \tag{3.2}$$

and

$$\lim_{s \longrightarrow \infty} T_{total}(s) = \lim_{s \longrightarrow 0} T_{total}(s) = \infty. \tag{3.3}$$

Therefore, there is a minimum for T_{total} when s changes.

Let

$$T'_{total}(s) = -\frac{m\alpha}{s^2} + (p-1)(\beta+\gamma) = 0.$$

Then, we can get the point that makes $T_{total}(s)$ reach its minimum

$$s_1 = \sqrt{\frac{m\alpha}{(p-1)(\beta+\gamma)}}$$

When $s_1 = \sqrt{\frac{m\alpha}{(p-1)(\beta+\gamma)}}$,

$$\begin{aligned} T_{total} &= (p-1)\alpha + (\beta+\gamma)m + 2\sqrt{(p-1)\alpha(\beta+\gamma)m} \\ &= (\beta+\gamma)m \; . \; \left(1 + 2\sqrt{\frac{(p-1)\alpha}{(\beta+\gamma)m}} + \frac{(p-1)\alpha}{(\beta+\gamma)m}\right) \end{aligned} \tag{3.4}$$

When $(\beta+\gamma)m \gg (p-1)\alpha$, both $2\sqrt{\frac{(p-1)\alpha}{(\beta+\gamma)m}}$ and $\frac{(p-1)\alpha}{(\beta+\gamma)m}$ are small. Therefore, $T_{total} \approx (\beta+\gamma)m$.

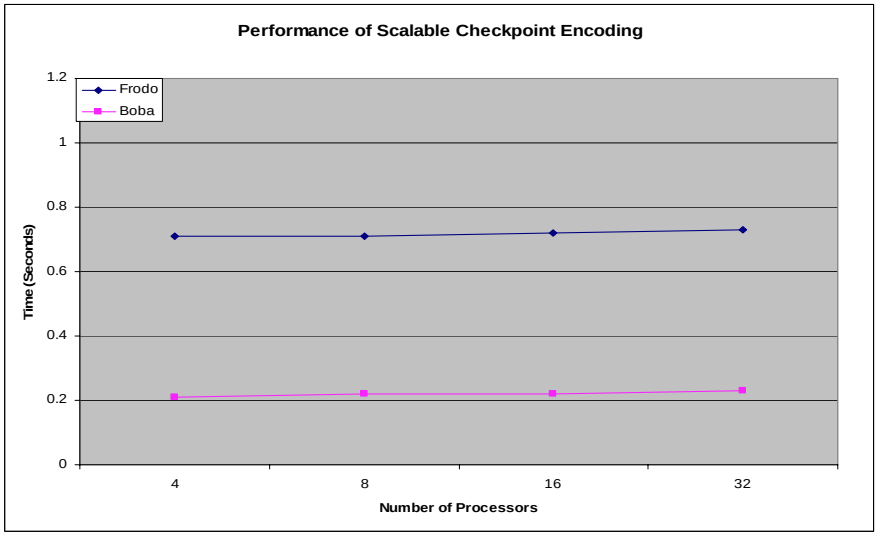


Figure 3.4: Performance of pipeline encoding with 8 Mega-bytes local checkpoint data on each processor

In diskless checkpointing, the size of checkpoint m is often large (Mega-bytes level). The latency α is often a very small number compared with the time to send a large message. If p is not too large, then $(\beta + \gamma)m \gg (p-1)\alpha$. Therefore, in practice, the number of processors will have very little impact on the time to perform one checkpoint.

Figure 3.4 shows the time to perform one checkpoint encoding with 8 Mega-bytes local checkpoint data on each processor.

Note that the overhead for existing checkpoint encoding is $T_{diskless-binary} = 2\lceil \log p \rceil (\alpha + \beta m + \gamma m)$; therefore, the pipeline encoding algorithm improves the scalability of diskless checkpointing.

3.3 Coding to Tolerate Multiple Process Failures

To tolerate multiple simultaneous process failures of arbitrary patterns with minimum process redundancy, a weighted checksum scheme can be used. A weighted checksum scheme can be viewed as a version of the Reed-Solomon erasure coding scheme [49] in the real number field. The basic idea of this scheme works as follow: Each processor takes a local in-memory checkpoint, and m equalities are established by saving weighted checksums of the local checkpoint into m checksum processors. When f failures happen, where $f \leq m$, the m equalities becomes m equations with f unknowns. By appropriately choosing the weights of the weighted checksums, the lost data on the f failed processors can be recovered by solving these m equations.

3.3.1 The Basic Weighted Checksum Scheme

Suppose there are n processors used for computation. Assume the checkpoint data on the i-th computation processor is P_i. In order to be able to reconstruct the lost data on failed processors, another m processors are dedicated to hold m encodings (weighted checksums) of the checkpoint data (see Figure 3.5).

The weighted checksum C_j on the jth checksum processor can be calculated from

$$\begin{cases} a_{11}P_1 + \ldots + a_{1n}P_n & = C_1 \\ & \vdots \\ a_{m1}P_1 + \ldots + a_{mn}P_n & = C_m, \end{cases} \tag{3.5}$$

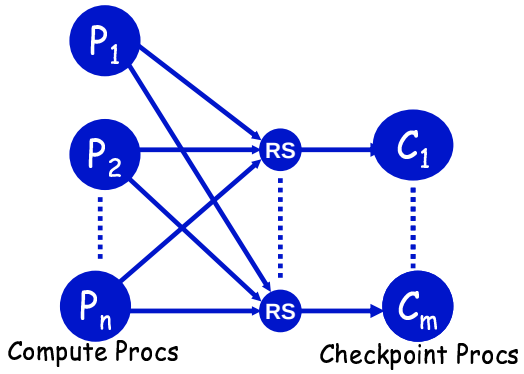


Figure 3.5: Basic weighted checksum scheme for diskless checkpointing

where a_{ij}, $i = 1, 2, ..., m$, $j = 1, 2, ..., n$, is the weight we need to choose. Let $A = (a_{ij})_{mn}$. We call A the checkpoint matrix for the weighted checksum scheme.

Suppose that k computation processors and $m-h$ checkpoint processors have failed. Then there are $n - k$ computation processors and h checkpoint processors that have survived. If we look at the data on the failed processors as unknowns, then (3.5) becomes m equations with $m - (h - k)$ unknowns.

If $k > h$, then there are fewer equations than unknowns. There is no unique solution for (3.5), and the lost data on the failed processors can not be recovered.

However, if $k < h$, then there are more equations than unknowns. By appropriately choosing A, a unique solution for (3.5) can be guaranteed, and the lost data on the failed processors can be recovered by solving (3.5).

Without loss of generality, we assume: (1) the computational processors j_1, $j_2, ..., j_k$ failed and the computational processors j_{k+1}, $j_{k+2}, ..., j_n$ survived; (2) the checkpoint processors i_1, $i_2, ..., i_h$ survived and the checkpoint processors i_{h+1}, $i_{h+2}, ..., i_m$ failed.

Then, in equation (2), $P_{j_1}, ..., P_{j_k}$ and $C_{i_{h+1}}, ..., C_{i_m}$ become unknowns after the failure occurs. If we re-structure (3.5), we can get

$$\begin{cases} a_{i_1 j_1} P_{j_1} + ... + a_{i_1 j_k} P_{j_k} & = C_{i_1} - \sum_{t=k+1}^{n} a_{i_1 j_t} P_{j_t} \\ & \vdots \\ a_{i_h j_1} P_{j_1} + ... + a_{i_h j_k} P_{j_k} & = C_{i_h} - \sum_{t=k+1}^{n} a_{i_h j_t} P_{j_t} \end{cases} \tag{3.6}$$

and

$$\begin{cases} C_{i_{h+1}} & = a_{i_{h+1}1} P_1 + \ldots + a_{i_{h+1}n} P_n \\ & \vdots \\ C_{i_m} & = a_{i_m 1} P_1 + \ldots + a_{i_m n} P_n. \end{cases} \tag{3.7}$$

Let A_r denote the coefficient matrix of the linear system (3.6). If A_r has full column rank, then $P_{j_1}, ..., P_{j_k}$ can be recovered by solving (3.6), and $C_{i_{h+1}}, ..., C_{i_m}$ can be recovered by substituting $P_{j_1}, ..., P_{j_k}$ into (3.7).

Whether we can recover the lost data on the failed processes or not directly depends on whether A_r has full column rank or not. However, A_r in (3.6) can be any sub-matrix (including minor) of A depending on the distribution of the failed processors. If any square sub-matrix (including minor) of A is non-singular and there are no more than m process failed, then A_r can be guaranteed to have full column rank. Therefore, to be able to recover from no more than any m failures, the checkpoint matrix A has to satisfy the condition that *any square sub-matrix (including minor) of A is non-singular.*

How can we find such kind of matrices? It is well known that some structured matrices such as Vandermonde matrix and Cauchy matrix satisfy this condition.

However, in computer floating point arithmetic where no computation is exact due to round-off errors, it is well known [2] that, in solving a linear system of equations, a condition number of 10^k for the coefficient matrix leads to a loss of accuracy of about k decimal digits in the solution. Therefore, in order to get a reasonably accurate recovery, the checkpoint matrix A actually has to satisfy *any square sub-matrix (including minor) of A is well-conditioned.*

It is well-known [14] that Gaussian random matrices are well-conditioned. To estimate how well conditioned Gaussian random matrices are, we have proved the following Theorem:

Theorem 1 *Let $G_{m\times n}$ be an $m\times n$ real random matrix whose elements are independent and identically distributed standard normal random variables, and let $\kappa_2(G_{m\times n})$ be the 2-norm condition number of $G_{m\times n}$. Then, for any $m \geq 2$, $n \geq 2$ and $x \geq |n-m|+1$, $\kappa_2(G_{m\times n})$ satisfies*

$$P\left(\frac{\kappa_2(G_{m\times n})}{n/(|n-m|+1)} > x\right) < \frac{1}{\sqrt{2\pi}}\left(\frac{C}{x}\right)^{|n-m|+1},$$

and

$$E(\ln \kappa_2(G_{m\times n})) < \ln \frac{n}{|n-m|+1} + 2.258,$$

where $0.245 \leq c \leq 2.000$ and $5.013 \leq C \leq 6.414$ are universal positive constants

independent of m, n and x.

We omit the proof of the Theorem 1 here and put it in Chapter 6. Note that any sub-matrix of a Gaussian random matrix is still a Gaussian random matrix. Therefore, a Gaussian random matrix would satisfy the condition that any sub-matrix of the matrix is well-conditioned with high probability.

Theorem 1 can be used to estimate the accuracy of recovery in the weighted checksum scheme. For example, if an application uses 100,000 processors to perform computation and 20 processors to perform checkpointing, then the checkpoint matrix is a 20 by 100,000 Gaussian random matrix. If 10 processors fail concurrently, then the coefficient matrix A_r in the recovery algorithm is a 20 by 10 Gaussian random matrix. From Theorem 1, we can get

$$E(\log_{10} \kappa_2(A_r)) < 1.25$$

and

$$P(\kappa_2(A_r) > 100) < 3.1 \times 10^{-11}.$$

Therefore, on average, we will lose about one decimal digit in the recovered data and the probability to lose 2 digits is less than 3.1×10^{-11}.

3.3.2 One Dimensional Weighted Checksum Scheme

The one dimensional weighted checksum scheme works as follows. Assume the program is running on $m \times n$ processors. Partition the $m \times n$ processors into m groups with

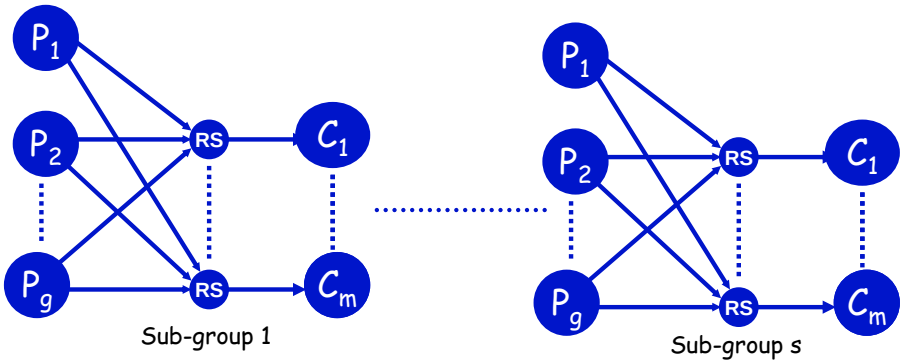


Figure 3.6: One dimensional weighted checksum scheme for diskless checkpointing

n processors in each group. Dedicate another k checksum processors for each group. In each group, the checkpoint are done using the basic weighted checksum scheme (see Figure 3.6). This scheme can survive k processor failures in each group. The advantage of this scheme is that the checkpoints are localized to a subgroup of processors, so the checkpoint encoding in each sub-group can be done in parallel. Therefore, compared with the basic weighted checksum scheme, the performance of the one dimensional weighted checksum scheme is usually better.

By using a pipelined encoding algorithm in each subgroup, the time to perform one checkpoint in the one dimensional weighted checksum scheme is

$$T_{one-dimensional} = (\beta+\gamma)m \cdot \left(1 + 2\sqrt{\frac{(p-1)\alpha}{(\beta+\gamma)m}} + \frac{(p-1)\alpha}{(\beta+\gamma)m}\right) \quad (3.8)$$

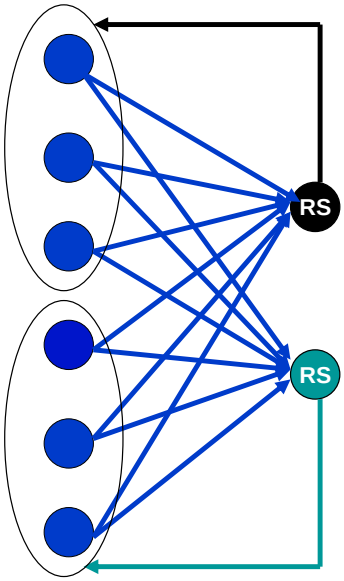


Figure 3.7: Localized weighted checksum scheme for diskless checkpointing

3.3.3 Localized Weighted Checksum Scheme

The localized weighted checksum scheme works as follows. Assume we want to tolerate k simultaneous process failures. Divide all processes onto subgroups of size $k(k+1)$. In each group, the checkpoint encoding is performed like the basic weighted checksum scheme (see Figure 3.7). But each encoding is distributed into $k+1$ processes in the subgroup. Note that there are $k(k+1)$ processes in each subgroup, therefore, all k encodings can be replicated in $k+1$ processes with each process hold only one encoding. This scheme can survive k processor failures in each group. The advantage of this scheme is that the checkpoints are localized to a subgroup of processors, so the checkpoint encoding in each sub-group can be done in parallel. Therefore, compared with the basic

weighted checksum scheme, the performance of the localized weighted checksum scheme is usually better. Another advantage of the localized weighted checksum scheme is that it does not require dedicated processes to hold the checkpoint encoding.

By using a pipelined encoding algorithm in each subgroup, the time to perform one checkpoint in the localized weighted checksum scheme is

$$\begin{aligned} T_{one-dimensional} &= (\beta+\gamma)m\left(1+2\sqrt{\frac{k(k+1)\alpha}{(\beta+\gamma)m}}+\frac{k(k+1)\alpha}{(\beta+\gamma)m}\right) \\ &\quad +\beta m\left(1+2\sqrt{\frac{k\alpha}{\beta m}}+\frac{k\alpha}{\beta m}\right) \end{aligned} \tag{3.9}$$

3.4 A Fault Survivable Iterative Equation Solver

In this section, we give a detailed presentation on how to incorporate fault tolerance into applications by using a preconditioned conjugate gradient equation solver as an example.

3.4.1 Preconditioned Conjugate Gradient Algorithm

The Preconditioned Conjugate Gradient (PCG) method is the most commonly used algorithm to solve the linear system $Ax = b$ when the coefficient matrix A is sparse and symmetric positive definite. The method proceeds by generating vector sequences of iterates (i.e., successive approximations to the solution), residuals corresponding to the iterates, and search directions used in updating the iterates and residuals. Although the length of these sequences can become large, only a small number of vectors needs to be

Compute $r^{(0)} = b - Ax^{(0)}$ for some initial guess $x^{(0)}$
for $i = 1, 2, \ldots$
    solve $Mz^{(i-1)} = r^{(i-1)}$
    $\rho_{i-1} = {r^{(i-1)}}^T z^{(i-1)}$
    if $i = 1$
        $p^{(1)} = z^{(0)}$
    else
        $\beta_{i-1} = \rho_{i-1}/\rho_{i-2}$
        $p^{(i)} = z^{(i-1)} + \beta_{i-1} p^{(i-1)}$
    endif
    $q^{(i)} = Ap^{(i)}$
    $\alpha_i = \rho_{i-1}/{p^{(i)}}^T q^{(i)}$
    $x^{(i)} = x^{(i-1)} + \alpha_i p^{(i)}$
    $r^{(i)} = r^{(i-1)} - \alpha_i q^{(i)}$
    check convergence; continue if necessary
end

Figure 3.8: Preconditioned conjugate gradient algorithm

kept in memory. In every iteration of the method, two inner products are performed in order to compute update scalars that are defined to make the sequences satisfy certain orthogonality conditions. The pseudo-code for the PCG is given in Figure 3.8. For more details of the algorithm, we refer the reader to [4].

3.4.2 Incorporating Fault Tolerance into PCG

We first implemented the parallel non-fault tolerant PCG. The preconditioner M we use is the diagonal part of the coefficient matrix A. The matrix A is stored as sparse row compressed format in memory. The PCG code is implemented such that any symmetric, positive definite matrix using the Harwell Boeing format or the Matrix Market format

can be used as a test problem. One can also choose to generate the test matrices in memory according to testing requirements.

We then incorporate the basic weighted checksum scheme into the PCG code. Assume the PCG code uses n MPI processes to do computation. We dedicate another m MPI processes to hold the weighted checksums of the local checkpoint of the n computation processes. The checkpoint matrix we use is a pseudo random matrix. Note that the sparse matrix does not change during computation; therefore, we only need to checkpoint three vectors (i.e. the iterate, the residual and the search direction) and two scalars (i.e. the iteration index and $\rho^{(i-1)}$ in Figure 3.8).

The communicator mode we use is the REBUILD mode. The communication mode we use is the NOOP/RESET mode. Therefore, when processes failed, FT-MPI will drop all pending messages and re-spawn all failed processes without changing the rank of the surviving processes.

An FT-MPI application can detect and handle failure events using two different methods: either the return code of every MPI function is checked, or the application makes use of MPI error handlers. The second mode gives users the possibility of incorporating fault tolerance into applications that call existing parallel numerical libraries that do not check the return code of their MPI calls. In the PCG code, we detect and handle failure events by checking the return code of every MPI function.

The recovery algorithm in PCG makes use of the *longjmp* function of the C-standard. In case the return code of an MPI function indicates that an error has occurred, all

surviving processes set their state variable to RECOVER and *jump* to the recovery section in the code. The recovery algorithm consists of the following steps:

1. Re-spawn the failed processes and recover the FT-MPI runtime environment by calling a specific, predefined MPI function.

2. Determining how many processes have died and who has died.

3. Recover the lost data from the weighted checksums using the algorithm described in Section 4.3.1.

4. Resume the computation.

Another issue is how a process can determine whether it is a survival process or it is a re-spawned process. FT-MPI offers the user two possibilities to solve this problem:

- In the first method, when a process is a replacement for a failed process, the return value of its MPI_Init call will be set to a specific new FT-MPI constant (MPI_INIT_RESTARTED_PROCS).

- The second possibility is that the application introduces a static variable. By comparing the value of this variable to the value on the other processes, the application can detect whether everybody has been newly started (in which case all processes will have the pre-initialized value), or whether a subset of processes have a different value, since each processes modifies the value of this variable after the initial check. This second approach is somewhat more complex; however, it is fully portable and can also be used with any other non fault-tolerant MPI library.

In PCG, each process checks whether it is a re-spawned process or a surviving process by checking the return code of its MPI_Init call.

The relevant section with respect to fault tolerance is shown in the source code below.

```
/* Determine who is re-spawned */
rc = MPI_Init( &argc, &argv );
if (rc==MPI_INIT_RESTARTED_NODE) {
  /* re-spawned procs initialize */
  ...
} else {
  /* Original procs initialize*/
  ...
}
/*Failed procs jump to here to recover*/
setjmp( env );
/* Execute recovery if necessary */
if ( state == RECOVER ) {
  /*Recover MPI environment*/
  newcomm = FT_MPI_CHECK_RECOVER;
  MPI_Comm_dup(oldcomm, &newcomm);
  /*Recover application data*/
```

```
  recover_data (A, b, r, p, x, ...);
  /*Reset state-variable*/
  state = NORMAL;
}
/*Major computation loop*/
do {
  /*Checkpoint every K iterations*/
  if ( num_iter % K == 0 )
      checkpoint_data(r, p, x, ...);
  /*Check the return of communication
    calls to detect failure. If failure
    occurs, jump to recovery point*/
  rc = MPI_Send ( ...)
  if ( rc == MPI_ERR_OTHER ) {
      state = RECOVER;
      longjmp ( env, state );
  }
} while ( not converge );
```

3.5 Experimental Evaluation

In this section, we evaluate both the performance overhead of our fault tolerance approach and the numerical impact of our floating-point arithmetic encoding using the PCG code implemented in the last section.

We performed four sets of experiments to answer the following four questions:

1. What is the performance of FT-MPI compared with other state-of-the-art MPI implementations?

2. What is the performance overhead of performing checkpointing?

3. What is the performance overhead of performing recovery?

4. What is the numerical impact of round-off errors in recovery?

For each set of experiments, we test PCG with four different problems. The size of the problems and the number of computation processors used (not including checkpoint processors) for each problem are listed in Table 3.1.

All experiments were performed on a cluster of 64 dual-processor 2.4 GHz AMD Opteron nodes. Each node of the cluster has 2 GB of memory and runs the Linux operating system. The nodes are connected with a Gigabit Ethernet. The timer we used in all measurements is MPI_Wtime.

Table 3.1: Experiment configurations for each problem

	Size of the Problem	Num. of Comp. Procs
Prob #1	164,610	15
Prob #2	329,220	30
Prob #3	658,440	60
Prob #4	1,316,880	120

Table 3.2: PCG execution time (in seconds) with different MPI implementations

Time	Prob#1	Prob#2	Prob#3	Prob#4
MPICH-1.2.6	916.2	1985.3	4006.8	10199.8
MPICH2-0.96	510.9	1119.7	2331.4	7155.6
FT-MPI	480.3	1052.2	2241.8	6606.9
FT-MPI ckpt	482.7	1055.1	2247.5	6614.5
FT-MPI rcvr	485.8	1061.3	2256.0	6634.0

3.5.1 Performance of PCG with Different MPI Implementations

The first set of experiments was designed to compare the performance of different MPI implementations and evaluate the overhead of surviving a single failure with FT-MPI. We ran PCG with MPICH-1.2.6 [31], MPICH2-0.96, FT-MPI, FT-MPI with one checkpoint processor and no failure, and FT-MPI with one checkpoint processor and one failure for 2000 iterations. For PCG with FT-MPI with checkpoint, we checkpoint every 100 iterations. For PCG with FT-MPI with recovery, we simulate a processor failure by exiting one process at the 1000-th iteration. The execution times of all tests are reported in Table 3.2.

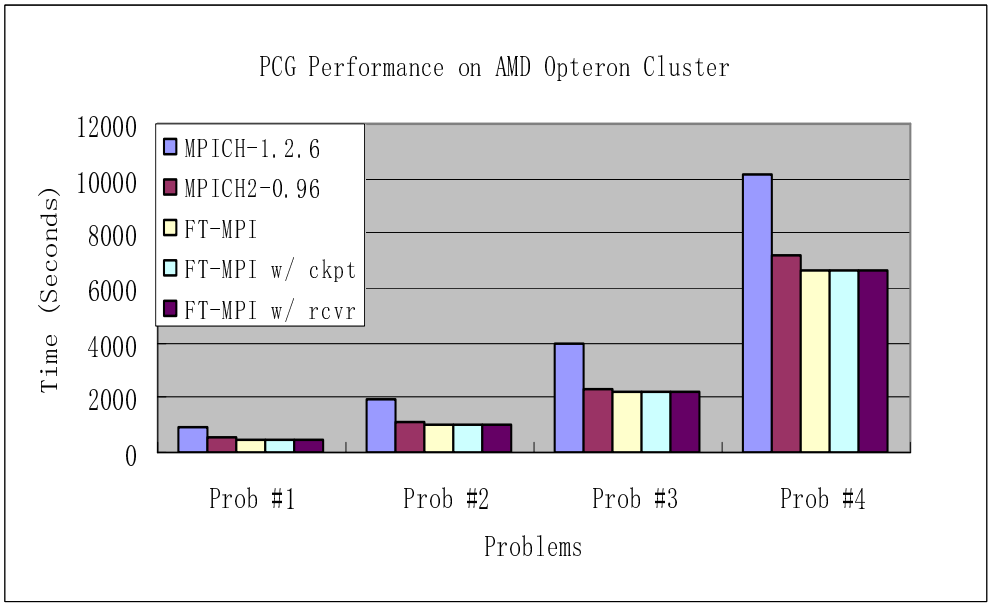


Figure 3.9: PCG performance with different MPI implementations

Figure 3.9 compares the execution time of PCG with MPICH-1.2.6, MPICH2-0.96, FT-MPI, FT-MPI with one checkpoint processor and no failure, and FT-MPI with one checkpoint processor and one failure for different sizes of problems. Figure 5 indicates that the performance of FT-MPI is slightly better than MPICH2-0.96. Both FT-MPI and MPICH2-0.96 are much faster than MPICH-1.2.6. Even if with checkpointing and/or recovery, the performance of PCG with FT-MPI is still at least comparable to MPICH2-0.96.

3.5.2 Performance Overhead of Taking Checkpoint

The purpose of the second set of experiments is to measure the performance penalty of taking checkpoints to survive general multiple simultaneous processor failures. There

are no processor failures involved in this set of experiments. At each run, we divided the processors into two classes. The first class of processors is dedicated to perform PCG computation work. The second class of processors ais dedicated to perform checkpoint. In Table 3.3 and Table 3.4, the first column of the table indicates the number of checkpoint processors used in each test. If the number of checkpoint processors used in a run is zero, then there is no checkpoint in this run. For all experiments, we ran PCG for 2000 iterations and checkpoint every 100 iterations.

Table 3.3 reports the execution time of each test. In order to reduce the disturbance of the noise of the program execution time to the checkpoint time, we measure the time used for checkpointing separately for all experiments.

Table 3.3: PCG execution time (in seconds) with checkpoint

Time	Prob #1	Prob #2	Prob #3	Prob #4
0 ckpt	480.3	1052.2	2241.8	6606.9
1 ckpt	482.7	1055.1	2247.5	6614.5
2 ckpt	484.4	1057.9	2250.3	6616.9
3 ckpt	486.5	1059.9	2252.4	6619.7
4 ckpt	488.1	1062.2	2254.7	6622.3
5 ckpt	489.9	1064.3	2256.5	6625.1

Table 3.4: PCG checkpointing time (in seconds)

Time	Prob #1	Prob #2	Prob #3	Prob #4
1 ckpt	2.6	3.8	5.5	7.8
2 ckpt	4.4	5.8	8.5	10.6
3 ckpt	6.0	7.9	10.2	12.8
4 ckpt	7.9	9.9	12.6	15.0
5 ckpt	9.8	11.9	14.1	16.8

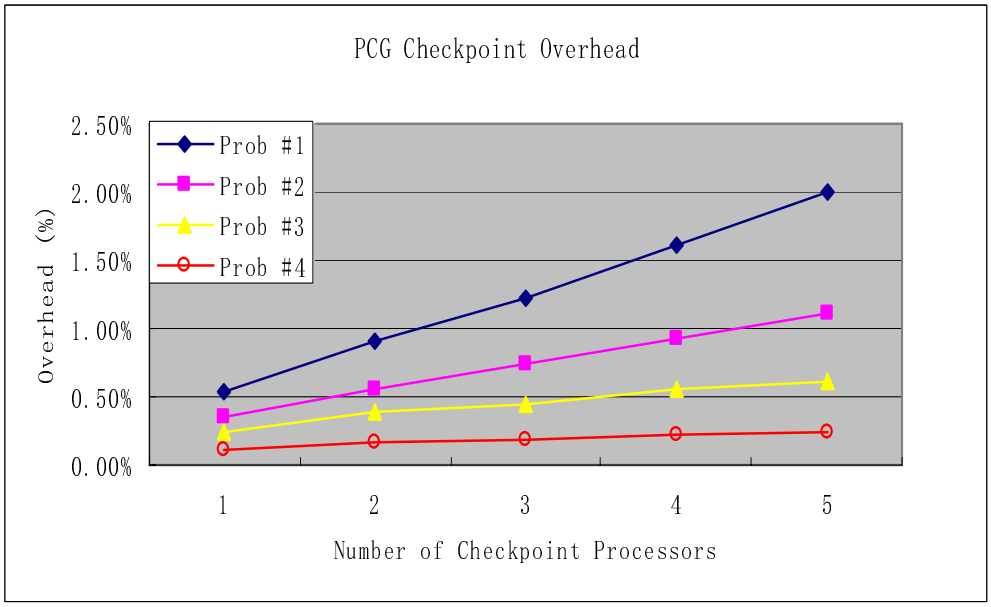


Figure 3.10: PCG checkpoint overhead

Table 3.4 reports the individual checkpoint time for each experiment. Figure 3.10 compares the checkpoint overhead (%) of surviving different numbers of simultaneous processor failures for different size of problems.

Table 3.4 indicates that as the number of checkpoint processors increases, the time for checkpointing in each test problem also increases. The increase in time for each additional checkpoint processor is approximately the same for each test problem. However, the increase of the time for each additional checkpoint processor is smaller than the time for using only one checkpoint processor. This is because from no checkpoint to checkpoint with one checkpoint processor PCG has to first set up the checkpoint environment and then do one encoding. However, from checkpoint with k (where $k > 0$) processors to checkpoint with $k + 1$ processors, the only additional work is to perform

one more encoding.

Note that we are performing checkpoint every 100 iterations and run PCG for 2000 iterations; therefore, from Table 3, we can calculate the checkpoint interval for each test. Our checkpoint interval ranges from 25 seconds (Prob #1) to 330 seconds (Prob #4). In practice, there is an optimal checkpoint interval which depends on the failure rate, the time cost of each checkpoint and the time cost of each recovery. Much literature about the optimal checkpoint interval [29, 54, 63] is available. We will not address this issue further here.

From Figure 3.10, we can see that even if we checkpoint every 25 seconds (Prob #1), the performance overhead of checkpointing to survive five simultaneous processor failures is still within 2% of the original program execution time, which actually falls into the noise margin of the program execution time. If we checkpoint every 5.5 minutes (Prob #4) and assume a processor fails one after another (one checkpoint processor case), then the overhead is only 0.1%.

3.5.3 Performance Overhead of Performing Recovery

The third set of experiments is designed to measure the performance overhead to perform recovery. All experiment configurations are the same as in the previous section except that we simulate a failure of k (k equals the number of checkpoint processors in the run) processors by exiting k processes at the 1000-th iteration in each run.

Table 3.5 reports the execution time of PCG with recovery. In order to reduce the disturbance of the noise of the program execution time to the recovery time, we measure

Table 3.5: PCG execution time (in seconds) with recovery

Time	Prob #1	Prob #2	Prob #3	Prob #4
0 proc	480.3	1052.2	2241.8	6606.9
1 proc	485.8	1061.3	2256.0	6634.0
2 proc	488.1	1063.6	2259.7	6633.5
3 proc	490.0	1066.1	2262.1	6636.3
4 proc	492.6	1068.8	2265.4	6638.2
5 proc	494.9	1070.7	2267.5	6639.7

Table 3.6: PCG recovery time (in seconds)

Time	Prob #1	Prob #2	Prob #3	Prob #4
1 proc	3.2	5.0	8.7	18.2
2 proc	3.7	5.5	9.2	18.8
3 proc	4.0	6.0	9.8	20.0
4 proc	4.5	6.5	10.4	20.9
5 proc	4.8	7.0	11.1	21.5

the time used for recovery separately for all experiments. Table 3.6 reports the recovery time in each experiment. Figure 3.11 compares the recovery overhead (%) from different numbers of simultaneous processor failures for different sizes of problems.

From Table 3.6, we can see the recovery time increases approximately linearly as the number of failed processors increases. However, the recovery time for a failure of one processor is much longer than the increase of the recovery time from a failure of k (where $k > 0$) processors to a failure of $k + 1$ processors. This is because, from no failure to a failure with one failed processor, the additional work the PCG has to perform includes first setting up the recovery environment and then recovering data. However, from a

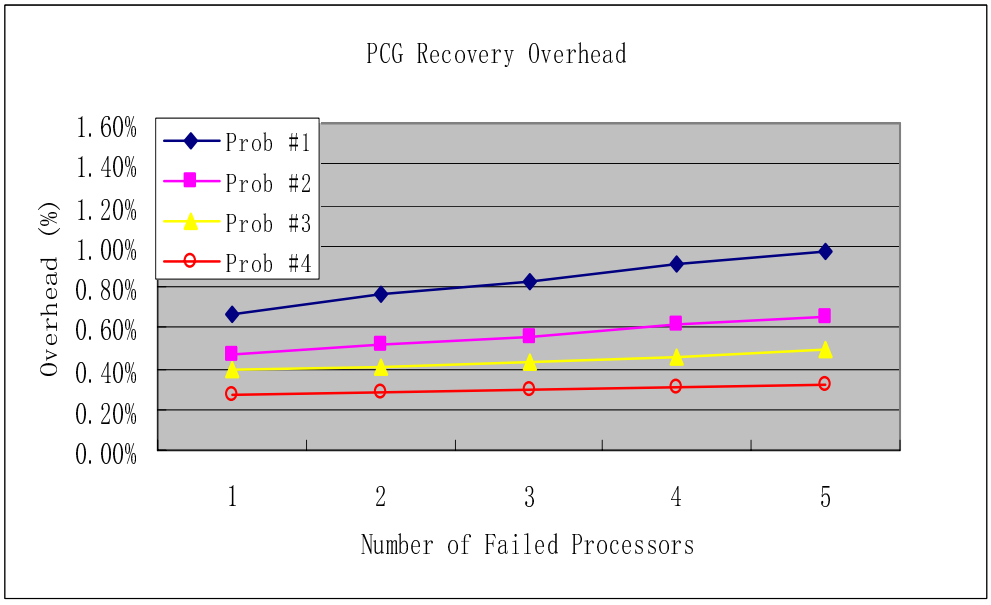


Figure 3.11: PCG recovery overhead

failure with k (where $k > 0$) processors to a failure with $k + 1$ processors, the only additional work is to recover data for an additional processor.

From Figure 3.11, we can see that the overheads for recovery in all tests are within 1% of the program execution time, which is again within the noise margin of the program execution time.

3.5.4 Numerical Impact of Round-Off Errors in Recovery

As discussed in Section 3.1, our diskless checkpointing schemes are based on floating-point arithmetic encodings, which introduce round-off errors into the checkpointing system. The experiments in this sub-section are designed to measure the numerical impact of the round-off errors in our checkpointing system. All experiment configurations

Table 3.7: Numerical impact of round-off errors in PCG recovery

Residual	Prob #1	Prob #2	Prob #3	Prob #4
0 proc	3.050e-6	2.696e-6	3.071e-6	3.944e-6
1 proc	2.711e-6	4.500e-6	3.362e-6	4.472e-6
2 proc	2.973e-6	3.088e-6	2.731e-6	2.767e-6
3 proc	3.036e-6	3.213e-6	2.864e-6	3.585e-6
4 proc	3.438e-6	4.970e-6	2.732e-6	4.002e-6
5 proc	3.035e-6	4.082e-6	2.704e-6	4.238e-6

are the same as in the previous section except that we report the norm of the residual at the end of each computation.

Note that if no failures occur, the computation proceeds with the same computational data as without checkpoint. Therefore, the computational results are affected only when there is a recovery in the computation. Table 7 reports the norm of the residual at the end of each computation when there are 0, 1, 2, 3, 4, and 5 simultaneous process failures.

From Table 3.7, we can see that the norms of the residuals are different for different numbers of simultaneous process failures. This is because, after recovery, due to the impact of round-off errors in the recovery algorithm, the PCG computations are performed based on slightly different recovered data. However, table 7 also indicates that the residuals with recovery do not have much difference from the residuals without recovery.

3.6 Discussion

The size of the checkpoint affects the performance of any checkpointing scheme. The larger the checkpoint size is, the higher the diskless checkpoint overhead will be. In the PCG example, we only need to checkpoint three vectors and two scalars periodically, therefore, the performance overhead is very low.

Diskless checkpointing is good for applications that modify a small amount of memory between checkpoints. There are many such applications in the high performance computing field. For example, in typical iterative methods for sparse matrix computation, the sparse matrix is often not modified during the program execution, only some vectors and scalars are modified between checkpoints. For this type of application, the overhead for surviving a small number of processor failures is very low.

Even for applications that modify a relatively large amount of memory between two checkpoints, reasonable performance for surviving single processor failure were reported in [39].

The basic weighted checksum scheme implemented in the PCG example has a higher performance overhead than other schemes discussed in Section 3. When an application is executed on a large number of processors, to survive general multiple simultaneous processor failures, the one dimensional weighted checksum scheme will achieve a much lower performance overhead than the basic weighted checksum scheme. If processors fail one after another (i.e. no multiple simultaneous processor failures), the neighbor based schemes can achieve even lower performance overhead. It was shown in [12]

that neighbor-based checkpointing is an order of magnitude faster than parity-based checkpointing, but takes twice as much storage overhead.

Diskless checkpointing can not survive a failure of all processors. Also, to survive a failure occurring during checkpoint or recovery, the storage overhead would double. If an application needs to tolerate these types of failures, a two level recovery scheme [62] which uses both diskless checkpointing and stable-storage-based checkpointing is a good choice.

Another drawback of our fault tolerance approach is that it requires the programmer to be involved in the fault tolerance. However, if the fault tolerance schemes are implemented into numerical software packages such as LFC [10], then transparent fault tolerance can also be achieved for programmers using these software tools.

3.7 Conclusions and Future Work

We have presented how to build fault survivable high performance computing applications with FT-MPI using diskless checkpointing. We have introduced floating-point arithmetic encodings into diskless checkpointing and discussed several checkpoint encoding strategies in detail. We have also implemented a fault survivable example application (PCG) that can survive general multiple simultaneous processor failures. Experimental results show that FT-MPI is at least comparable to other state-of-the-art MPI implementations with respect to performance and can support fault survivable MPI applications at the same time. Experimental results further demonstrate that our fault

tolerance approach can survive a small number of simultaneous processor failures with low performance overhead and little numerical impact.

For the future, we will evaluate our fault tolerance approach on systems with larger numbers of processors. We would also like to evaluate our fault tolerance approach with more applications and more diskless checkpointing schemes.

Chapter 4

Algorithm-Based Checkpoint-Free Fault Tolerance

As the size of today's high performance computers increases from hundreds, to thousands, and even tens of thousands of processors, node failures in these computers are becoming frequent events. Although checkpoint/rollback-recovery is the typical technique to tolerate such failures, it often introduces considerable overhead. Algorithm-based fault tolerance is a very cost-effective method to incorporate fault tolerance into matrix computations. However, previous algorithm-based fault tolerance methods for matrix computations are often derived using algorithms that are seldom used in the practice of today's high performance matrix computations and have mostly focused on platforms where failed processors produce incorrect calculations. To fill this gap, this chapter extends the existing algorithm-based fault tolerance for parallel matrix com-

putations to the volatile computing platform where the failed processor stops working. Instead of taking checkpoints periodically, a coded global consistent state of the critical application data is maintained in memory by modifying applications to operate on encoded data. Because no periodical checkpoint or rollback-recovery is involved in this approach, partial node failures can often be tolerated with surprisingly low overhead. We show the practicality of this technique by applying it to the ScaLAPACK matrix-matrix multiplication kernel which is one of the most important kernels for the ScaLAPACK library to achieve high performance and scalability. Experimental results demonstrate that the proposed approach is able to survive process failures with very low performance overhead.

4.1 Motivation

Today's long running scientific applications typically deal with faults by checkpoint/restart approaches in which all process states of an application are saved into stable storage periodically. The advantage of this approach is that it is able to tolerate the failure of the whole system. However, in this approach, if one process fails, usually all surviving processes are aborted and the whole application is restarted from the last checkpoint. The major source of overhead in all stable-storage-based checkpoint systems is the time it takes to write checkpoints into stable storage [53]. The checkpoint of an application on a, say, ten-thousand-processor computer implies that all critical data for the application on all ten thousand processors have to be written into stable storage periodically, which

may introduce an unacceptable amount of overhead into the checkpointing system. The restart of such an application implies that all processes have to be recreated and all data for each process have to be re-read from stable storage into memory or re-generated by computation, which often brings a large amount of overhead into restart. It may also be very expensive or unrealistic for many large systems such as grids to provide the large amount of stable storage necessary to hold all process state of an application with thousands of processes.

In order to tolerate partial failures with reduced overhead, diskless checkpointing [53] has been proposed by Plank et. al. By eliminating stable storage from checkpointing and replacing it with memory and processor redundancy, diskless checkpointing removes the main source of overhead in checkpointing [53]. Diskless checkpointing has been shown to achieve a decent performance to tolerate single process failure in [39]. For applications that modify a small amount of memory between checkpoints, it is shown in [11] that ,even to tolerate multiple simultaneous process failures, the overhead introduced by diskless checkpointing is still negligible.

However, for applications such as matrix-matrix multiplication that modify a large mount of memory between checkpoints, due to the large checkpoint size, even diskless checkpointing still introduces a considerable overhead into applications. Firstly, a local in memory checkpoint has to be maintained in diskless checkpointing, thus introducing large amount of memory overhead and hurts the efficiency of applications. Secondly, the local checkpoint in diskless checkpointing has to be taken and encoded periodically,

which introduce a considerable performance overhead into applications. Although the checksum and reverse computation technique in [39] has reduced the memory overhead, the overhead to calculate the checkpoint encodings periodically does not change. Furthermore, after failures, this technique increases the recovery overhead by reversing the computation.

Inspired by the existing algorithm-based fault tolerance idea in [35], in this chapter, we present an algorithm-based checkpoint-free fault tolerance approach in which, instead of taking checkpoints periodically, a coded global consistent state of the critical application data is maintained in memory by modifying applications to operate on encoded data. Although this approach is not as generally applicable as typical checkpoint approaches, in parallel matrix computations where it usually works, because no periodic checkpoint and rollback-recovery are involved in this approach, fault tolerance for partial node failures can often be achieved with a surprisingly low overhead.

Despite the fact that there has been much research on algorithm-based fault tolerance [35] in which applications are modified to operate on encoded data to determine the correctness of some mathematical calculations on parallel platforms where failed processors produce incorrect calculations, to the best of our knowledge, this is the first time that applications are modified to operate on encoded data to maintain a global consistent state on parallel and distributed systems where failed processors stop working.

We show the practicality of this technique by applying it to the ScaLAPACK [6]

matrix-matrix multiplication kernel which is one of the most important kernels for the ScaLAPACK library to achieve high performance and scalability. We address the practical numerical issue in this technique by proposing a class of numerically good real number erasure codes based on random matrices. Experimental results for matrix-matrix multiplication demonstrate that the proposed approach is able to survive a small number of process failures with a very low performance overhead.

Although the algorithm-based checkpoint-free fault tolerance approach presented in this chapter is non-transparent and algorithm-dependent, it is meaningful in that

1. It can often achieve a surprisingly low overhead in parallel matrix computations where it usually works.

2. It is often possible to build it into frequently used numerical libraries such as ScaLAPACK to relieve the involvement of the application programmer.

4.2 Algorithm-Based Checkpoint-Free Fault Tolerance

In this section, we present the basic idea of algorithm-based checkpoint-free fault tolerance. We restrict our scope to long running numerical computing applications only. As indicated in Section 5, this approach can mainly be applied to linear algebra computations on parallel and distributed systems.

4.2.1 Failure Detection and Location

It is assumed that fail-stop failures can be detected and located with the aid of the programming environment. Many current programming environments such as PVM [59], Globus [24], FT-MPI [20, 21, 19], and Open MPI [26] do provide this kind of failure detection and location capability. We assume that the loss of some processes in the message passing system does not cause the aborting of the surviving processes and that it is possible to replace the failed processes in the message passing system and continue the communication after the replacement. FT-MPI [20] is one such programming environment that supports all these functionalities. In the rest of this section, we will mainly focus on how to recover the application.

4.2.2 Single Failure Recovery

Today's long running scientific programs typically deal with faults by checkpoint and rollback recovery in which all process states of an application are saved into certain storage periodically. If one process fails, the data on *all* processes have to be recovered from the last checkpoint. The checkpoint and rollback of an application on a, say, ten-thousand-processor computer implies that all critical data for the application on all ten thousand processors have to be saved into and recovered from some storage periodically, which may introduce an unacceptable amount of overhead (both time and storage) into the checkpointing system. Considering that all data on all surviving processes are still effective, it is interesting to ask: *is it possible to recover only the lost data on the failed*

processes?

Consider the simple case where there will be only one process failure. Before the failure actually occurs, we do not know which process will fail; therefore, a scheme to recover only the lost data on the failed process actually needs to be able to recover data on *any* process. It seems difficult to be able to recover data on any process without saving all data on all processes somewhere. However, if we assume, at any time during the computation, that the data on the i^{th} process P_i satisfy

$$P_1 + P_2 + \cdots + P_{n-1} = P_n, \tag{4.1}$$

where n is the total number of process used for the computation, then the lost data on *any* failed process can be able to be recovered from (4.1). Assume the j^{th} process fails, then the lost data P_j can be recovered from

$$P_j = P_n - (P_1 + \cdots + P_{j-1} + P_{j+1} + \cdots + P_{n-1})$$

In this very special case, we are lucky enough to be able to recover the lost data on *any* failed process without checkpoint due to the special *checksum relationship* (4.1). In practice, this kind of special relationship is by no means natural. However, it is natural to ask: *is it possible to design an application to maintain such a special checksum relationship on purpose?*

Assume the original application is designed to run on n processes. Let P_i denote

the data on the i^{th} computation process. The special checksum relationship above can actually be designed on purpose as follows:

- Add another encoding process into the application. Assume that the data on this encoding process is C. For numerical computations, P_i is often an array of floating-point numbers; therefore, at the beginning of the computation, we can create a checksum relationship among the data of all processes by initializing the data C on the encoding process as

$$P_1 + P_2 + \cdots + P_n = C \tag{4.2}$$

- During the execution of the application, redesign the algorithm to operate both on the data of computation processes and on the data of encoding process in such a way that the checksum relationship (4.2) is always maintained.

The specially designed checksum relationship (4.2) actually establishes an equality between the data P_i on computation processes and the encoding data C on the encoding process. If any processor fails, then the equality (4.2) becomes an equation with one unknown. Therefore, the data on the failed processor can be reconstructed through solving this equation.

4.2.3 Multiple Failure Recovery

The specially designed checksum relationship in the last sub-section can only survive one process failure. However, in today's high performance computers, there are usually multiple processors on each node. Hence, it is usual to run multiple processes on one node, which implies that the failure of one node often causes the loss of multiple processes. Furthermore, as the number of nodes increases, the possibility of lost multiple nodes also increases. Therefore, it is often necessary to be able to survive multiple simultaneous process failures. In this section, we present a scheme that can be used to recover multiple simultaneous process failures.

Suppose there are n processes used for computation. Assume P_i represents the data on the i-th computation process. In order to be able to reconstruct the lost data on m failed processes, another m processes are dedicated to hold m encodings (weighted checksums) of the computation data. At the beginning of the application, the weighted checksum C_j on the jth encoding process can be calculated from

$$\begin{cases} a_{11}P_1 + \ldots + a_{1n}P_n & = C_1 \\ & \vdots \\ a_{m1}P_1 + \ldots + a_{mn}P_n & = C_m, \end{cases} \tag{4.3}$$

where a_{ij}, $i = 1, 2, ..., m$, $j = 1, 2, ..., n$, is the weight we need to choose. During the executing of the application, the application needs to be re-designed to operate both on

the data of computation processes and on the data of encoding processes in such a way that the relationship (4.3) is always maintained.

We call the relationship (4.3) the *weighted checksum relationship*. We call $A = (a_{ij})_{mn}$ the *encoding matrix* for the weighted checksum relationship. The specially designed weighted checksum relationship (4.3) actually establishes m equalities between the data P_i on computation processes and the encoding data C_i on the encoding processes. If some processes fail, then the m equalities become a system of linear equations. Therefore, the lost data on the failed processes may be able to be reconstructed through solving the system of linear equations.

Supposing that k computation processes and $m - h$ encoding processes have failed, then $n - k$ computation processors and h encoding processes have survived. If we look at the data on failed processors as unknowns, then (4.3) becomes m equations with $m - (h - k)$ unknowns.

If $k > h$, then there are fewer equations than unknowns, and there is no unique solution for (4.3). The lost data on the failed processes cannot be recovered.

However, if $k < h$, then there are more equations than unknowns. By appropriately choosing A, a unique solution for (4.3) can be guaranteed. Therefore, the lost data on the failed processes can be recovered by solving (4.3).

Without loss of generality, we assume: (1) the computational processes j_1, j_2, ..., j_k failed and the computational processes j_{k+1}, j_{k+2}, ..., j_n survived; (2) the encoding processes i_1, i_2, ..., i_h survived and the encoding processes i_{h+1}, i_{h+2}, ..., i_m failed. Then, in

equation (4.3), $P_{j_1}, ..., P_{j_k}$ and $C_{i_{h+1}}, ..., C_{i_m}$ become unknowns after the failure occurs. If we re-structure (4.3), we can get

$$\begin{cases} a_{i_1 j_1} P_{j_1} + ... + a_{i_1 j_k} P_{j_k} & = C_{i_1} - \sum_{t=k+1}^{n} a_{i_1 j_t} P_{j_t} \\ & \vdots \\ a_{i_h j_1} P_{j_1} + ... + a_{i_h j_k} P_{j_k} & = C_{i_h} - \sum_{t=k+1}^{n} a_{i_h j_t} P_{j_t} \end{cases} \tag{4.4}$$

and

$$\begin{cases} C_{i_{h+1}} & = a_{i_{h+1}1} P_1 + \ldots + a_{i_{h+1}n} P_n \\ & \vdots \\ C_{i_m} & = a_{i_m 1} P_1 + \ldots + a_{i_m n} P_n. \end{cases} \tag{4.5}$$

Let A_r denote the coefficient matrix of the linear system (4.4). If A_r has full column rank, then $P_{j_1}, ..., P_{j_k}$ can be recovered by solving (4.4), and $C_{i_{h+1}}, ..., C_{i_m}$ can be recovered by substituting $P_{j_1}, ..., P_{j_k}$ into (4.5).

Whether we can recover the lost data on the failed processes or not directly depends on whether A_r has full column rank or not. However, A_r in (4.4) can be any sub-matrix (including minor) of A depending on the distribution of the failed processors. If any square sub-matrix (including minor) of A is non-singular and there are no more than m process failed, then A_r can be guaranteed to have full column rank. Therefore, to be able to recover from any no more than m failures, the encoding matrix A has to satisfy: *any square sub-matrix (including minor) of A is non-singular.*

How can we find such kind of matrices? It is well known that some structured matrices such as Vandermonde matrix, Cauchy matrix, and DFT matrix satisfy any square sub-matrix (including minor) of the matrix is non-singular.

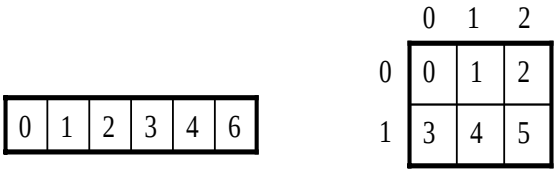
4.3 Checkpoint-Free Fault Tolerance for Matrix Multiplication

As an example to demonstrate how the algorithm-based checkpoint-free fault tolerance works in practice, in this section, we apply this technique to the ScaLAPACK matrix-matrix multiplication kernel which is one of the most important kernels for ScaLAPACK to achieve high performance and scalability.

Actually, it is also possible to incorporate fault tolerance into many other ScaLAPACK routines through this approach. However, in this section, we will restrict our presentation to the matrix-matrix multiplication kernel. For the simplicity of presentation, in this section, we only discuss the case where there is one process failure. However, just as described in the last section, it is straightforward to extend the result here to the multiple simultaneous process failures case by simply using the weighted checksum scheme in Section 4.2.3.

4.3.1 Two-Dimensional Block-Cyclic Distribution

It is well-known [6] that the layout of an application's data within the hierarchical memory of a concurrent computer is critical in determining the performance and scalability



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

Figure 4.1: Process grid in ScaLAPACK

of the parallel code. By using two-dimensional block-cyclic data distribution [6], ScaLAPACK seeks to maintain load balance and reduce the frequency with which data must be transferred between processes.

For reasons described above, ScaLAPACK organizes the one-dimensional process array representation of an abstract parallel computer into a two-dimensional rectangular process grid. Therefore, a process in ScaLAPACK can be referenced by its row and column coordinates within the grid. An example of such an organization is shown in Figure 4.1.

The two-dimensional block-cyclic data distribution scheme is a mapping of the global matrix onto the rectangular process grid. There are two pairs of parameters associated with the mapping. The first pair of parameters is (mb, nb), where mb is the row block size and nb is the column block size. The second pair of parameters is (P, Q), where P is the number of process rows in the process grid and Q is the number of process columns in the process grid. Given an element a_{ij} in the global matrix A, the process

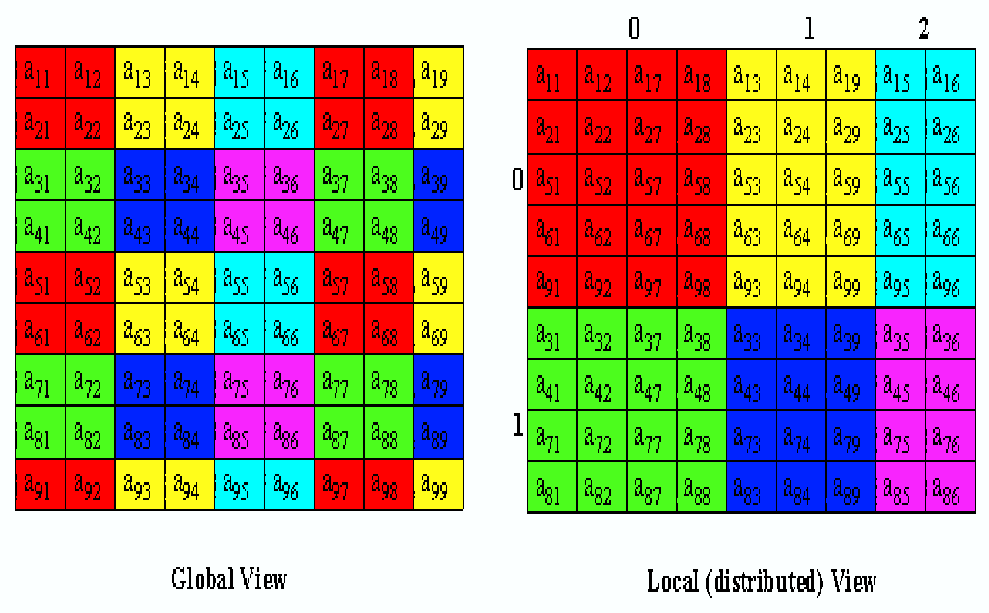


Figure 4.2: Two-dimensional block-cyclic matrix distribution

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

$$\begin{cases} p_i = \lfloor \frac{i}{mb} \rfloor \bmod P, \\ q_j = \lfloor \frac{j}{nb} \rfloor \bmod Q, \end{cases}$$

The local coordinate (i_{p_i}, j_{q_j}) which a_{ij} resides in the process (p_i, q_j) can be calculated according to the following formula

$$\begin{cases} i_{p_i} = \lfloor \frac{\lfloor \frac{i}{mb} \rfloor}{P} \rfloor . mb + i \mod mb, \\ j_{q_j} = \lfloor \frac{\lfloor \frac{j}{nb} \rfloor}{Q} \rfloor . nb + j \mod nb, \end{cases}$$

Figure 4.2 is an example of mapping a 9 by 9 matrix onto a 2 by 3 process grid according

two-dimensional block-cyclic data distribution with $mb = nb = 2$.

4.3.2 Encoding Two-Dimensional Block Cyclic Matrices

In this section, we will construct different encoding schemes which can be used to design checkpoint-free fault tolerant matrix computation algorithms in ScaLAPACK.

Assume a matrix M is originally distributed in a P by Q process grid according to the two dimensional block cyclic data distribution. For the convenience of presentation, assume the sizes of the local matrices in each process are the same. We will explain different coding schemes for the matrix M with the help of the example matrix in Figure 4.3. Figure 4.3 (a) shows the global view of an example matrix. After the matrix is mapped onto a 2 by 2 process grid with $mb = nb = 1$, the distributed view of this matrix is shown in Figure 4.3 (b).

Suppose we want to tolerate a single process failure. We dedicate another $P+Q+1$ additional processes and organize the total $PQ+P+Q+1$ process as a $P+1$ by $Q+1$ process grid with the original matrix M distributed onto the first P rows and Q columns of the process grid.

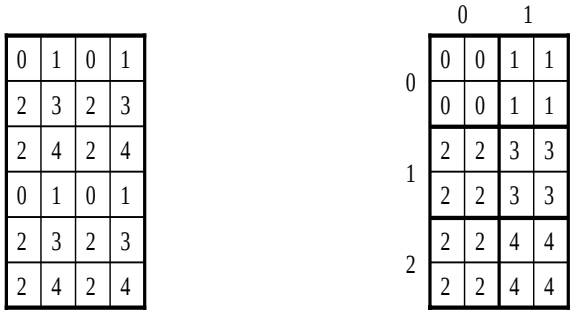
0	1	0	1
2	3	2	3
0	1	0	1
2	3	2	3

(a). Original matrix from global view

	0	0	1	1
0	0	0	1	1
	2	2	3	3
1	2	2	3	3

(b). Original matrix from distributed view

Figure 4.3: An example matrix with two-dimensional block cyclic distribution



(a). Column checksum matrix from global view (b). Column checksum matrix from distributed view

Figure 4.4: Distributed column checksum matrix of the example matrix

The *distributed column checksum matrix* M^c of the matrix M is the original matrix M plus the part of data on the $(P+1)^{th}$ process row which can be obtained by adding all local matrices on the first P process rows. Figure 4.4 (b) shows the distributed view of the column checksum matrix of the example matrix from Figure 4.1. Figure 4.4 (a) is the global view of the column checksum matrix.

The *distributed row checksum matrix* M^r of the matrix M is the original matrix M plus the part of data on the $(Q+1)^{th}$ process columns which can be obtained by adding all local matrices on the first Q process columns. Figure 4.5 (b) shows the distributed view of the row checksum matrix of the example matrix from Figure 4.1. Figure 4.5 (a) is the global view of the row checksum matrix.

The *distributed full checksum matrix* M^f of the matrix M is the original matrix M, plus the part of data on the $(P+1)^{th}$ process row which can be obtained by adding all local matrices on the first P process rows, plus the part of data on the $(Q+1)^{th}$ process column which can be Figure 4.6 (b) shows the distributed view of the full checksum

0	1	1	0	1	1
2	3	5	2	3	5
0	1	1	0	1	1
2	3	5	2	3	5

(a). Row checksum matrix from global view

	0		1		2	
0	0	0	1	1	1	1
	0	0	1	1	1	1
1	2	2	3	3	5	5
	2	2	3	3	5	5

(b). Row checksum matrix from distributed view

Figure 4.5: Distributed row checksum matrix of the original matrix

0	1	1	0	1	1
2	3	5	2	3	5
2	4	6	2	4	6
0	1	1	0	1	1
2	3	5	2	3	5
2	4	6	2	4	6

(a). Full checksum matrix from global view

	0		1		2	
0	0	0	1	1	1	1
	0	0	1	1	1	1
1	2	2	3	3	5	5
	2	2	3	3	5	5
2	2	2	4	4	6	6
	2	2	4	4	6	6

(b). Full checksum matrix from distributed view

Figure 4.6: Distributed full checksum matrix of the original matrix

matrix of the example matrix from Figure 4.3. Figure 4.6 (a) is the global view of the full checksum matrix.

4.3.3 Scalable Universal Matrix Multiplication Algorithm

To achieve high performance, the matrix-matrix multiplication in ScaLAPACK uses the blocked outer product version of algorithm. Let A_j denote the j^{th} column block of the matrix A and B_j^T denote the j^{th} row block of the matrix B. The following Figure 4.8 is the algorithm to perform the matrix matrix multiplication. Figure 4.7 shows the j^{th} step of the matrix matrix multiplication algorithm.

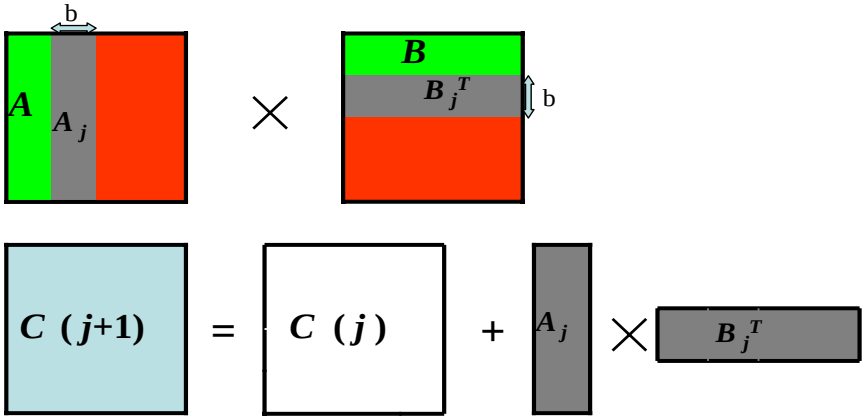


Figure 4.7: The j^{th} step of the matrix-matrix multiplication algorithm in ScaLAPACK

```
for j = 0, 1, ...
    row broadcast A_j;
    column broadcast B_j^T;
    C = C + A_j * B_j^T;
end
```

Figure 4.8: Scalable universal matrix-matrix multiplication algorithm in ScaLAPACK

4.3.4 Maintaining Global Consistent States by Computation

Assume A, B and C are distributed matrices on a P by Q process grid with the first element of each matrix on process (0,0). Let A^c, B^r and C^f denote the corresponding distributed checksum matrix. Let A^c_j denote the j^{th} column block of the matrix A^c and B^{rT}_j denote the j^{th} row block of the matrix B^r. We first prove the following fundamental theorem for matrix matrix multiplication with checksum matrices.

Theorem 2 *Let* $S_j = C^f + \sum_{k=0}^{j-1} A^c_k * B^{rT}_k$, *then* S_j *is a distributed full checksum matrix.*

Proof. It is straightforward that $A^c_k * B^{rT}_k$ is a distributed full checksum matrix and the sum of two distributed full checksum matrices is a distributed checksum matrix. S_j is the sum of j distributed full checksum matrices, therefore it is a distributed full checksum matrix. □

Theorem 2 tells us that at the end of each iteration of the matrix matrix multiplication algorithm with checksum matrices, the checksum relationship of all checksum matrices is still maintained. This tells us that a coded global consistent state of the critical application data is maintained in memory at the end of each iteration of the matrix matrix multiplication algorithm if we perform the computation with related checksum matrices.

However, in a distributed environment, different processes may update their local data asynchronously. Therefore, if when some process has updated its local matrix and some process is still in the communication stage, a failure happens, then the relationship

of the data in the distributed matrix would not be maintained and the data on all processes would not form a consistent state. But this could be solved by simply performing a synchronization before performing local update. Therefore, in the following algorithm in Figure 4.10, there will always be a coded global consistent state (i.e. the checksum relationship) of the matrix A^c, B^r and C^f in memory. Hence, a single process failure at any time during the matrix matrix multiplication would be able to recovered from the checksum relationship. Figure 4.9 shows the j^{th} step of the fault tolerant matrix matrix multiplication algorithm.

In this algorithm, the only modification to the library routine is to perform a synchronization before local update. However the amount of modification necessary to maintain a consistent state is highly dependent on the characteristic of an algorithm. For example, in LU factorization, due to the damage of the linear relationship by the global row pivoting, one also needs to adjust the encodings appropriately when performing pivoting to maintain a consistent encoded state in memory.

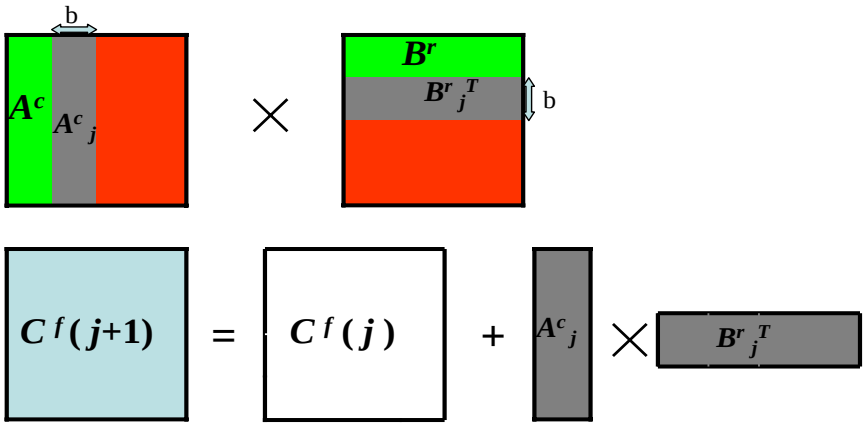


Figure 4.9: The j^{th} step of the fault tolerant matrix-matrix multiplication algorithm

```
construct checksum matrices A^c, B^r, and C^f;
for j = 0, 1, ...
    row broadcast A^c_j;
    column broadcast B^{rT}_j;
    synchronize;
    C^f = C^f + A^c_j * B^{rT}_j;
end
```

Figure 4.10: A fault tolerant matrix-matrix multiplication algorithm

4.3.5 Overhead and Scalability Analysis

In this section, we analyze the overhead introduced by the algorithm-based checkpoint-free fault tolerance for matrix matrix multiplication.

For the simplicity of presentation, we assume all three matrices A, B, and C are square. Assume all three matrices are distributed onto a P by P process grid with m by m local matrices on each process. The size of the global matrices is $P \times m$ by $P \times m$. Assume all elements in the matrices are 8-byte double precision floating-point numbers. Assume every process has the same speed and that 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. Let γ denote the time it takes for a process to perform one floating-point arithmetic operation.

Time Complexity for Parallel Matrix Matrix Multiplication

Note that the sizes of all three global matrices A, B, and C are all $P \times m$; therefore, the total number of floating-point arithmetic operations in the matrix matrix multiplication is $2P^3m^3$. There are P^2 process with each process executing the same number of floating-point arithmetic operations. Hence, the total number of floating-point arithmetic operations on each process is $2Pm^3$. Therefore, the time T_{matrix_comp} for the computation in matrix matrix multiplication is

$$T_{matrix\_comp} = 2Pm^3\gamma.$$

In the parallel matrix matrix multiplication algorithm in Figure 4.8, the columns of A and the rows of B also need to broadcast to other column and row processes respectively. To broadcast one block columns of A using a simple *binary tree* broadcast algorithm, it takes $2(\alpha + 8bm\beta)\log_2 P$, where b is the row block size in the two dimensional block cyclic distribution. Therefore, the time T_{matrix_comm} for the communication in matrix matrix multiplication is

$$T_{matrix\_comm} = 2\alpha\frac{Pm}{b}\log_2 P + 16\beta Pm^2\log_2 P.$$

Therefore, the total time to perform parallel matrix matrix multiplication is

$$\begin{aligned} T_{matrix\_mult} &= T_{matrix\_comp} + T_{matrix\_comm} \\ &= 2Pm^3\gamma + 2\alpha\frac{Pm}{b}\log_2 P \\ &\quad +16\beta Pm^2\log_2 P. \end{aligned} \tag{4.6}$$

Overhead for Calculating Encoding

To make matrix matrix multiplication fault tolerant, the first type of overhead introduced by the algorithm-based checkpoint-free fault tolerance technique is (1) constructing the distributed column checksum matrix A^c from A; (2) constructing the distributed row checksum matrix B^r from B; (3) constructing the distributed full checksum matrix C^f from C.

The distributed checksum operation involved in constructing all these checksum matrices performs the summation of P local matrices from P processes and saves the result into the $(P+1)^{th}$ process. Let T_{each_encode} denote the time for one checksum operation and T_{total_encode} denote the time for constructing all three checksum matrices A^c, B^r, and C^f, then

$$T_{total\_encode} = 4T_{each\_encode}$$

By using a *fractional tree* reduce style algorithm [55], the time complexity for one

checksum operation can be expressed as

$$T_{each\_encode} = 8m^2\beta\left(1+O\left(\left(\frac{\log_2 P}{m^2}\right)^{1/3}\right)\right) + O(\alpha\log_2 P) + O(m^2\gamma)$$

Therefore, the time complexity for constructing all three checksum matrices is

$$T_{total\_encode} = 32m^2\beta\left(1+O\left(\left(\frac{\log_2 P}{m^2}\right)^{1/3}\right)\right) + O(\alpha\log_2 P) + O(m^2\gamma). \quad (4.7)$$

In practice, unless the size of the local matrices m is very small or the size of the process grid P is extremely large, the total time for constructing all three checksum matrices is almost independent of the size of the process grid P.

The overhead (%) R_{total_encode} for constructing checksum matrices for matrix matrix multiplication is

$$\begin{aligned} R_{total\_encode} &= \frac{T_{total\_encode}}{T_{matrix\_mult}} \\ &= O(\frac{1}{Pm}) \end{aligned} \quad (4.8)$$

From (4.8), we can conclude

1. If the size of the data on each process is fixed (m is fixed), then as the number of processes increases to infinite (that is $P \to \infty$), the overhead (%) for constructing

the checksum matrices decreases to zero with a speed of $O(\frac{1}{P})$

2. If the number of processes is fixed (P is fixed), then as the size of the data on each process increases to infinite (that is $m \to \infty$) the overhead (%) for constructing the checksum matrices decreases to zero with a speed of $O(\frac{1}{m})$

Overhead for Performing Computations on Encoded Matrices

The fault tolerant matrix matrix multiplication algorithm in Figure 4.10 performs computations using checksum matrices which have larger size than the original matrices. However, the total number of processes devoted to computation also increases. A more careful analysis of the algorithm in Figure 4.10 indicates that the number of floating-point arithmetic operations on each process in the fault tolerant algorithm (Figure 4.10) is actually the same as that of the original non-fault tolerant algorithm (Figure 4.8).

As far as the communication is concerned, in the original algorithm (in Figure 4.8), the column (and row) blocks are broadcast to P processes. In the fault tolerant algorithms (in Figure 4.10), the column (and row) blocks now have to be broadcast to $P+1$ processes.

Therefore, the total time to perform matrix matrix multiplication with checksum matrices is

$$\begin{aligned} T_{matrix\_mult\_checksum} &= 2Pm^3\gamma + 2\alpha\frac{Pm}{b}\log_2(P+1) \\ &\quad +16\beta Pm^2\log_2(P+1). \end{aligned}$$

Therefore, the overhead (time) to perform computations with checksum matrices is

$$\begin{aligned} T_{overhead\_matrix\_mult} &= T_{matrix\_mult\_checksum} - T_{matrix\_mult} \\ &= (2\alpha\frac{Pm}{b} + 16\beta Pm^2)\log_2(1+\frac{1}{P}). \end{aligned} \tag{4.9}$$

The overhead (%) $R_{overhead\_matrix\_mult}$ for performing computations with checksum matrices in fault tolerant matrix matrix multiplication is

$$\begin{aligned} R_{overhead\_matrix\_mult} &= \frac{T_{overhead\_matrix\_mult}}{T_{matrix\_mult}} \\ &= O(\frac{1}{Pm}) \end{aligned} \tag{4.10}$$

From (4.10), we can conclude that

1. If the size of the data on each process is fixed (m is fixed), then as the number of processes increases to infinite (that is $P \to \infty$), the overhead (%) for performing computations with checksum matrices decreases to zero with a speed of $O(\frac{1}{P})$

2. If the number of processes is fixed (P is fixed), then as the size of the data on each process increases to infinite (that is $m \to \infty$) the overhead (%) for performing computations with checksum matrices decrease to zero with a speed of $O(\frac{1}{m})$

Overhead for Recovery

The failure recovery contains two steps: (1) recover the programming environment; (2) recover the application data.

The overhead for recovering the programming environment depends on the specific programming environment. For FT-MPI [20], on which we perform all our experiments, it introduces a negligible overhead (refer Section 4.5.3).

The procedure to recover the three matrices A, B, and C is similar to calculating the checksum matrices. For matrix C, it can be recovered from either the row checksum or the column checksum relationship. Therefore, the overhead to recover data is

$$\begin{aligned} T_{recover\_data} &= 24m^2\beta\left(1+O\left(\left(\frac{\log_2 P}{m^2}\right)^{1/3}\right)\right) \\ &\quad +O(\alpha\log_2 P)+O(m^2\gamma) \end{aligned} \tag{4.11}$$

In practice, unless the size of the local matrices m is very small or the size of the process grid P is extremely large, the total time for recover all three checksum matrices is almost independent of the size of the process grid P.

The overhead (%) $R_{recover\_data}$ for constructing checksum matrices for matrix matrix multiplication is

$$\begin{aligned} R_{recover\_data} &= \frac{T_{recover\_data}}{T_{matrix\_mult}} \\ &= O(\frac{1}{Pm}) \end{aligned} \tag{4.12}$$

4.4 Practical Numerical Issues

The algorithm-based checkpoint-free fault tolerance presented in Section 4.2 involves solving a system of linear equations to recover multiple simultaneous process failures. Therefore, in the practice of the algorithm-based checkpoint-free fault tolerance, the numerical issues involved in recovering multiple simultaneous process failures have to be addressed.

In Section 4.2.3, it has been derived that, to be able to recover from any no more than m failures, the encoding matrix A has to satisfy: any square sub-matrix (including minor) of A is non-singular. This requirement for the encoding matrix coincides with the properties for the generator matrices of real number Reed-Solomon style erasure correcting codes. In fact, our weighted checksum encoding in Section 4.2.3 can be viewed as a version of the Reed-Solomon erasure coding scheme [49] in the real number field. Therefore any generator matrix from real number Reed-Solomon style erasure codes can actually be used as the encoding matrix of algorithm-based checkpoint-free fault tolerance

In the existing real number or complex-number Reed-Solomon style erasure codes in the literature, the generator matrices mainly include the following: Vandermonde matrix (Vander) [32], Vandermonde-like matrix for the Chebyshev polynomials (Chebvand) [7], Cauchy matrix (Cauchy), Discrete Cosine Transform matrix (DCT), and Discrete Fourier Transform matrix (DFT) [23]. Theoretically, these generator matrices can all be used as the encoding matrix of the algorithm-based checkpoint-free fault

tolerance scheme.

However, in computer floating point arithmetic where no computation is exact due to round-off errors, it is well known [25] that, in solving a linear system of equations, a condition number of 10^k for the coefficient matrix leads to a loss of accuracy of about k decimal digits in the solution. Therefore, in order to get a reasonably accurate recovery, the encoding matrix A actually has to satisfy the condition that *any square sub-matrix (including minor) of A is well-conditioned.*

The generator matrices from above real number or complex-number Reed-Solomon style erasure codes all contain ill-conditioned sub-matrices. Therefore, in these codes, when certain error patterns occur, an ill-conditioned linear system has to be solved to reconstruct an approximation of the original information, which can cause the loss of precision of possibly all digits in the recovered numbers.

We will address these practical numerical issues by introducing a class of new codes based on Gaussian random matrices in Chapter 5 and 6.

4.5 Experimental Evaluation

In this section, we experimentally evaluate the performance overhead of applying the algorithm-based checkpoint-free fault tolerance technique to the ScaLAPACK matrix-matrix multiplication kernel. We performed four sets of experiments to answer the following four questions:

1. What is the performance overhead of constructing checksum matrices?

Table 4.1: Experiment configurations

Size of original matrix	12,800	19,200	25,600
Size of full checksum matrix	19,200	25,600	32,000
Process grid without FT	2 by 2	3 by 3	4 by 4
Process grid with FT	3 by 3	4 by 4	5 by 5

2. What is the performance overhead of performing computations with checksum matrices?

3. What is the performance overhead of recovering FT-MPI programming environments?

4. What is the performance overhead of recovering checksum matrices ?

For each set of experiments, the size of the problems and the number of computation processes used are listed in Table 4.1.

All experiments were performed on a cluster of 32 Pentium IV Xeon 2.4 GHz dual-processor nodes. Each node of the cluster has 2 GB of memory and runs the Linux operating system. The nodes are connected with a Gigabit Ethernet. The timer we used in all measurements was MPI_Wtime.

The programming environment we used was FT-MPI [20]. FT-MPI is a fault tolerant version of MPI that is able to provide basic system services to support fault survivable applications. FT-MPI implements the complete MPI-1.2 specification, some parts of the MPI-2 document, and extends some of the semantics of MPI for allowing the application the possibility to survive process failures. FT-MPI can survive the failure

of n-1 processes in a n-process job, and, if required, can re-spawn the failed processes. However, the application is still responsible for recovering the data structures and the data of the failed processes.

Although FT-MPI provides basic system services to support fault survivable applications, prevailing benchmarks show that the performance of FT-MPI is comparable [21] to other current state-of-the-art MPI implementations.

4.5.1 Overhead for Constructing Checksum Matrices

The first set of experiments is designed to evaluate the performance overhead of constructing checksum matrices. We keep the amount of data in each process fixed (that is the size of local matrices m fixed), and increase the size of the global test matrices (hence the size of process grid).

Table 4.2 reports the time for performing computations on original matrices and the time for constructing the three checksum matrices A^c, B^r, and C^f.

From Table 4.2, we can see that, as the size of the global matrices increases, the time for constructing checksum matrices increases only slightly. This is because, in the formula (4.7), when the size of process grid P is small, $32m^2\beta$ is the dominate factor in

Table 4.2: Time and overhead (%) for constructing checksum matrices

Size of original matrix	12,800	19,200	25,600
Exec. time for original matrix	442.9	695.0	989.8
Time for calculating encoding	38.0	40.8	43.2
Overhead (%) for encoding	8.6%	5.9%	4.4%

the time to constructing checksum matrices. Table 4.2 also indicates that the overhead (%) for constructing checksum matrices decreases as size of matrices increases, which is consistent with our theoretical formula (4.8) about the overhead for constructing checksum matrices.

4.5.2 Overhead for Performing Computations on Encoded Matrices

The algorithm-based checkpoint-free fault tolerance technique involves performing computations with checksum matrices, which introduces some overhead into the fault tolerance scheme. The purpose of this experiment is to evaluate the performance overhead of performing computations with checksum matrices.

Table 4.3 reports the execution time for performing computations on original matrices and the execution time for performing computations on checksum matrices for different size of matrices.

Table 4.3 indicates that the amount time increased for performing computations on checksum matrices increases slightly as the size of the matrices increases. The reason for this increase is that, when performing computations with checksum matrices, column

Table 4.3: Time and overhead (%) for performing computations on encoded matrices

Size of original matrix	12,800	19,200	25,600
Size of full checksum matrix	19,200	25,600	32,000
Exec. time for original matrix	442.9	695.0	989.8
Exec. time for encoded matrix	462.6	716.4	1013.3
Increased time	19.7	21.4	23.5
Overhead (%)	4.4%	3.1%	2.4%

Table 4.4: Time and overhead (%) for recovering FT-MPI environment

Size of original matrix	12,800	19,200	25,600
Exec. time for original matrix	442.9	695.0	989.8
Time for recovery FT-MPI	0.6	1.1	1.6
Overhead (%)	0.14%	0.16%	0.16%

blocks of A^c (and row blocks of B^r) have to be broadcast to one more process. The dominant time for parallel matrix matrix multiplication is the time for computation which is the same for both fault tolerant algorithm and non-fault tolerant algorithm. Therefore, the amount of time for fault tolerant algorithm increases only slightly as the size of matrices increases.

4.5.3 Overhead for Recovering FT-MPI Environment

The overhead for recovering programming environments depends on the specific programming environments. In this section, we evaluate the performance overhead of recovering the FT-MPI environment.

Table 4.4 reports the time for recovering the FT-MPI communication environment following a single process failure. Table 4.4 indicates that the overhead for recovering FT-MPI is less than 0.2% which is negligible in practice.

4.5.4 Overhead for Recovering Application Data

The purpose of this set of experiments is to evaluate the performance overhead of recovering application data following a single process failure.

Table 4.5: Time and overhead (%) for recovering application data

Size of original matrix	12,800	19,200	25,600
Exec. time for original matrix	442.9	695.0	989.8
Time for recovery data	28.5	30.6	32.4
Overhead (%)	6.4%	4.4%	3.3%

Table 4.5 reports the time for recovering the three checksum matrices A^c, B^r, and C^f in the case of single process failure. Table 4.5 indicates that ,as the size of the matrices increases, the time for recovering checksum matrices increases slightly and the overhead for recovering checksum matrices decreases, again confirming the theoretical results in Section 4.3.5.

4.6 Discussion

This chapter presented an algorithm-based checkpoint-free fault tolerance approach in which, instead of taking checkpoints periodically, a coded global consistent state of the critical application data is maintained in memory by modifying applications to operate on encoded data. Although the algorithm-based checkpoint-free fault tolerance in this chapter shares the same basic idea of modifying applications to operate on encoded data with the traditional the algorithm-based fault tolerance [35], they assume very different a failure model.

Compared with the typical checkpoint/restart approaches, the algorithm-based checkpoint-free fault tolerance in this chapter can only tolerate partial process failures. It needs

support from the programming environment to detect and locate failures. It requires the programming environment to be robust enough to survive node failures without suffering complete system failure. Both the overhead of and the additional effort to maintain a coded global consistent state of the critical application data in algorithm-based checkpoint-free fault tolerance is usually highly dependent on the specific characteristic of the application. Therefore, it is possible that the algorithm-based checkpoint-free fault tolerance approach introduces higher overhead than checkpoint approaches.

Unlike in typical checkpoint/restart approaches which involve periodical checkpoint and rollback-recovery, there is no checkpoint or rollback-recovery involved in this approach. Furthermore, in the algorithm-based checkpoint-free fault tolerance in this chapter, whenever process failures occur, it is only necessary to recover the lost data on the failed processes. Therefore, for many applications, it is also possible for this approach to achieve a much lower fault tolerance overhead than typical checkpoint/restart approaches. As shown in Section 4.3.5 and 4.5, for matrix matrix multiplication, which is one of the most fundamental operations for computational science and engineering, as the size N of the matrix increases, the fault tolerance overhead decreases with the speed of $\frac{1}{N}$.

Although the algorithm-based checkpoint-free fault tolerance approach presented in this chapter is non-transparent and algorithm-dependent, it is meaningful in that

1. It can often achieve a surprisingly low overhead in parallel matrix computations where it usually works.

2. It is often possible to build it into frequently used numerical libraries such as ScaLAPACK to relieve the involvement of the application programmer.

4.7 Conclusions and Future Work

In this chapter, we presented an algorithm-based checkpoint-free fault tolerance approach in which, instead of taking checkpoint periodically, a coded global consistent state of the critical application data is maintained in memory by modifying applications to operate on encoded data. Although the applicability of this approach is not so general as the typical checkpoint/rollback-recovery approach, in parallel matrix computations where it usually works, because no periodical checkpoint or rollback-recovery is involved in this approach, process failures can often be tolerated with a surprisingly low overhead.

We showed the practicality of this technique by applying it to the ScaLAPACK matrix-matrix multiplication kernel which is one of the most important kernels for ScaLAPACK library to achieve high performance and scalability. Experimental results demonstrated that the proposed checkpoint-free approach is able to survive process failures with a very low performance overhead.

For the future, we plan to incorporate this fault tolerance technique into more ScaLAPACK library routines and more high performance computing applications. We would also like to evaluate this technique on systems with larger numbers of processors.

Chapter 5

Numerically Stable Real Number Codes Based on Random Matrices

Error correction codes are often defined over finite fields. However, in many applications, error correction codes defined over finite fields do not work. Instead, codes defined over real-number or complex-number fields have to be used to detect and correct errors. For example, in algorithm-based fault tolerance [7] [35] [41] [45] and fault tolerant dynamic systems [32], to provide fault tolerance in computing, data are first encoded using error correction codes and then algorithms are re-designed to operate (using floating point arithmetic) on the encoded data. Due to the impact of the floating-point arithmetic on the binary representation of these encoded data, codes defined over finite fields do

not work. But codes defined over real-number and complex-number fields can be used in these applications to correct errors in computing by taking advantage of certain relationships, which are maintained only when real-number (or complex-number) codes are used.

However, most real-number and complex-number codes in literature are quite suspect in their numerical stability. Error correction procedures in most error correction codes involve solving linear system of equations. In computer floating point arithmetic where no computation is exact due to round-off errors, it is well known [25] that, in solving a linear system of equations, a condition number of 10^k for the coefficient matrix leads to a loss of accuracy of about k decimal digits in the solution. In the generator matrices of most existing real-number and complex-number codes, there exist ill-conditioned sub-matrices. Therefore, in these codes, when certain error patterns occur, an ill-conditioned linear system of equations has to be solved in the error correction procedure, which can cause the loss of precision of possibly all digits in the recovered numbers.

The numerical issue of the real-number and complex-number codes has been recognized and studied in the literature. In [7], Vandermonde-like matrix for the Chebyshev polynomials was introduced to relieve the numerical instability problem in error correction for algorithm-based fault tolerance. In [22] [23] [33] [43], the numerical properties of the Discrete Fourier Transform codes were analyzed and methods to improve the numerical properties were also proposed. To some extent, these efforts have alleviated

the numerical problem of the real-number and complex-number codes. However, how to construct real-number and complex-number codes without numerical problem is still an open problem.

In this chapter, we introduce a class of real-number and complex-number codes that are numerically much more stable than existing codes in the literature. Our codes are based on random generator matrices over real-number and complex-number fields. The rest of this chapter is organized as follow: Section 5.1 specifies the problem we focus on. In Section 5.2, we first study the properties of random matrices and then introduce our codes. Section 5.3 compares our codes with most existing codes in both burst error correction and random error correction. Section 5.4 concludes the chapter and discusses the future work.

5.1 Problem Specification

Let $x = (x_1, x_2, ..., x_N)^T \in \mathcal{C}^{\mathcal{N}}$ denote the original information, and G denote a M by N real or complex matrix. Let $y = (y_1, y_2, ..., y_M)^T \in \mathcal{C}^{\mathcal{M}}$, where $M = N + K$, denote the encoded information of x with redundancy. The original information x and the encoded information y are related through

$$y = Gx. \tag{5.1}$$

Our problem is: how to choose the matrix G such that, after any no more than K erasures in the elements of the encoded information y, a good approximation of the original information x can still be reconstructed from y.

When there are at most K elements of y lost, there are at least N elements of y available. Let J denote the set of indexes of any N available elements of y. Let y_J denote a sub-vector of y consisting of the N available elements of y whose indexes are in J. Let G_J denote a sub-matrix of G consisting of the N rows whose indexes are in J. Then, from (5.1), we can get the following relationship between x and y_J:

$$y_J = G_J x. \tag{5.2}$$

When the matrix G_J is singular, there are an infinite number of solutions to (5.2). But, if the matrix G_J is non-singular, then (5.2) has one and only one solution, which is the original information vector x.

For real-number and complex-number arithmetic on modern computers where no computation is exact due to round-off errors, it is well known [25] that, in solving a linear system of equations, a condition number of 10^k for the coefficient matrix leads to a loss of accuracy of about k decimal digits in the solution. Therefore, in order to reconstruct a good approximation of the original information x, G_J has to be well-conditioned.

For any N by N sub-matrix G_J of G, there is an erasure pattern of y that requires to solve a linear system with G_J as the coefficient matrix to reconstruct an approximation of the original x. Therefore, to guarantee that a reasonably good approximation of x can be reconstructed after any no more than K erasures in y , the generator matrix G must satisfy: *any N by N sub-matrix of G is well-conditioned.*

5.2 Real Number Codes Based on Random Matrices

In this section, we will introduce a class of new codes that are able to reconstruct a very good approximation of the original information with high probability regardless of the erasure patterns in the encoded information. Our new codes are based on random matrices over real or complex number fields.

5.2.1 Condition Number of Random Matrices from Standard Normal Distribution

In this sub-section, we mainly focus on the probability that the condition number of a random matrix is large and the expectation of the logarithm of the condition number. Let $G(m,n)$ be an $m \times n$ real random matrix whose elements are independent and identically distributed standard normal random variables and let $\widetilde{G}(m,n)$ be its complex counterpart.

Theorem 3 *Let κ denote the condition number of $G(n,n)$, $n > 2$, and $t \geq 1$, then*

$$\frac{0.13n}{t} < P(\kappa > t) < \frac{5.60n}{t}. \tag{5.3}$$

Moreover,

$$E(\log(\kappa)) = \log(n) + c + \epsilon_n, \tag{5.4}$$

where $c \approx 1.537$, $lim_{n\to\infty}\epsilon_n = 0$,

Proof. The inequality (5.3) is from Theorem 1 of [3]. The formula (5.4) can be obtained from Theorem 7.1 of [15]. □

Theorem 4 *Let $\widetilde{\kappa}$ denote the condition number of $\widetilde{G}(n,n)$, and $t \geq \sqrt{n}$, then*

$$1-\left(1-\frac{1}{t^2}\right)^{n^2-1} \leq P(\widetilde{\kappa} > t) \leq 1-\left(1-\frac{n}{t^2}\right)^{n^2-1}. \tag{5.5}$$

Moreover,

$$E(\log(\widetilde{\kappa})) = \log(n) + c + \epsilon_n, \tag{5.6}$$

where $c \approx 0.982$, $lim_{n\to\infty}\epsilon_n = 0$,

Proof. Let $\widetilde{\kappa}_D$ denote the scaled condition number (see [16] for definition) of $\widetilde{G}(n,n)$, then

$$P(\frac{\widetilde{\kappa}_D}{\sqrt{n}} > t) \leq P(\widetilde{\kappa} > t) \leq P(\widetilde{\kappa}_D > t). \tag{5.7}$$

From Corollary 3.2 in [16], we have

$$P(\widetilde{\kappa}_D > t) = 1-\left(1-\frac{n}{t^2}\right)^{n^2-1}. \tag{5.8}$$

Therefore,

$$P(\frac{\widetilde{\kappa}_D}{\sqrt{n}} > t) = P(\widetilde{\kappa}_D > \sqrt{n}t) = 1-\left(1-\frac{1}{t^2}\right)^{n^2-1}. \tag{5.9}$$

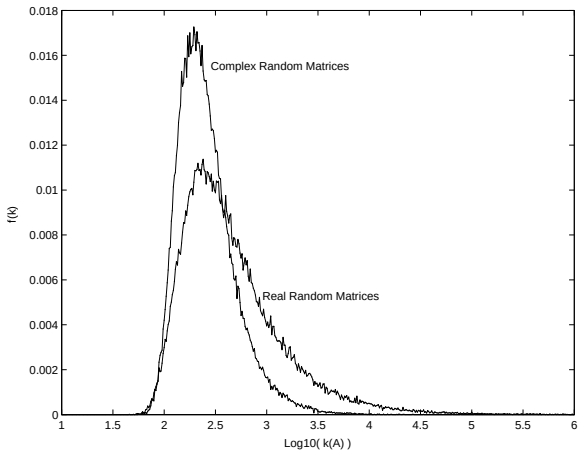


Figure 5.1: The probability density functions of the condition numbers of $G(100, 100)$ and $\widetilde{G}(100, 100)$.

The inequality (5.5) can be obtained from (5.7), (5.8) and (5.9). The formula (5.6) can be obtained from Theorem 7.2 of [15]. □

In error correction practice, all random numbers used are pseudo random numbers, which have to be generated through a random number generator. Figure 5.1 shows the empirical probability density functions of the condition numbers of the pseudo random matrix $G(100, 100)$ and $\widetilde{G}(100, 100)$, where $G(100, 100)$ is generated by *randn(100, 100)* and $\widetilde{G}(100, 100)$ is generated by *randn(100, 100)* + $\sqrt{-1}$ * *randn(100, 100)* in MATLAB. From these density functions, we know that most pseudo random matrices also have very small condition numbers. And, for the same matrix size, the tail of the condition number for a complex random matrix is thinner than that of a real one.

We have also tested some other random matrices. Experiments show a lot of other random matrices, for example, uniformly distributed pseudo random matrices, also

have small condition numbers with high probability. For random matrices of non-normal distribution, we will report our experiments and some analytical proofs of their condition number properties in a future work.

5.2.2 Real Number Codes Based on Random Matrices

In this sub-section, we introduce a class of new codes that are able to reconstruct a very good approximation of the original information with very high probability regardless of the erasure patterns in the encoded information.

In the real number case, we propose to use $G(M, N)$ or uniformly distributed M by N matrices with mean 0 (denote as $U(M, N)$) as our generator matrices G. In the complex number case, we propose to use $\widetilde{G}(M, N)$ or uniformly distributed M by N complex matrices with mean 0 (denote as $\widetilde{U}(M, N)$) as our generator matrices G.

Take the real-number codes based on random matrix $G(M, N)$ as an example. Since each element of the generator matrix $G(M, N)$ is a random number from the standard normal distribution, each element of any $N \times N$ sub-matrix $(G_J)_{N\times N}$ of $G(M, N)$ is also a random number from the standard normal distribution. According to the condition number results in Subsection 5.2.1 , the probability that the condition number of $(G_J)_{N\times N}$ is large is very small. Hence, any N by N sub-matrix $(G_J)_{N\times N}$ of G is well-conditioned with very high probability. Therefore, no matter what erasure patterns occur, the error correction procedure is numerically stable with high probability.

We admit that our real-number and complex-number codes are not perfect. Due to the probability approach we used, the drawback of our codes is that, no matter how

small the probability is, there is a probability that a erasure pattern may not be able to be recovered accurately.

However, compared with the existing codes in literature, the probability that our codes fail to recover a good approximation of the original information is negligible (see Section 5.3 for detail). Moreover, in the error correction practice, we may first generate a set of pseudo random generator matrices and then test each generator matrix until we find a satisfied one.

5.3 Comparison with Existing Codes

In the existing codes in the literature, the generator matrices mainly include: Vandermonde matrix (Vander) [32], Vandermonde-like matrix for the Chebyshev polynomials (Chebvand) [7], Cauchy matrix (Cauchy), Discrete Cosine Transform matrix (DCT), Discrete Fourier Transform matrix (DFT) [23]. These generator matrices all contain ill-conditioned sub-matrices. Therefore, in these codes, when certain error patterns occur, an ill-conditioned linear system has to be solved to reconstruct an approximation of the original information, which can cause the loss of precision of possibly all digits in the recovered numbers. However, in our codes, the generator matrices are random matrices. Any sub-matrix of our generator matrices is still a random matrix, which is well-conditioned with very high probability. Therefore, no mater what erasure patterns occur, the error correction procedure is numerically stable with high probability. In this section, we compare our codes with existing codes in both burst erasure correction and

random erasure correction.

5.3.1 Burst Erasure Correction

We compare our codes with existing codes in burst error correction using the following example.

Example: Suppose $x = (1, 1, 1, ..., 1)^T$ and the length of x is $N = 100$. G is a 120 by 100 generator matrix. $y = Gx$ is a vector of length 120. Suppose y_i, where $i = 101, 102, ...120$, are lost. We will use y_j, where $j = 1, 2, ...100$, to reconstruct x through solving (5.2) .

Table 5.1 shows how the generator matrix of each code is generated. Table 5.2 reports the accuracy of the recovery for each code. All calculations are done using MATLAB. The machine precision is 16 digits. Table 5.2 shows our codes are able to reconstruct the original information x with much higher accuracy than the existing

Table 5.1: The definition of different codes

Name	The generator matrix $G = (g_{mn})_{120\times100}$
Vander	$\left((m+1)^{100-n-1}\right)_{120\times100}$
Chebvand	$(T_{m-1}(n))_{120\times100}$, where T_{m-1} is the chebyshev polynomial of degree $n-1$
Cauchy	$\left(\frac{1}{m+n}\right)_{120\times100}$
DCT	$\left(\sqrt{\frac{i}{120}}\cos\frac{\pi(2n+1)m}{240}\right)_{120\times100}$, where if $m = 0, i = 1$, and if $m \neq 0, i = 2$
DFT	$\left(e^{-j\frac{2\pi}{120}mn}\right)_{120\times100}$, where $j = \sqrt{-1}$
RandN	randn(120,100) in MATLAB
RandN-C	randn(120,100) + j * randn(120,100) in MATLAB, where $j = \sqrt{-1}$
RandU	rand(120,100) - 0.5 in MATLAB
RandU-C	rand(120,100) - 0.5 + j * (rand(120,100) - 0.5) in MATLAB,

Table 5.2: Burst erasure recovery accuracy of different codes

Name	$\kappa(G_J)$	$\frac{\|x-\widehat{x}\|_2}{\|x\|_2}$	Accurate digits	Number of digits lost
Vander	3.7e+218	2.4e+153	0	16
Chebvand	Inf	1.7e+156	0	16
Cauchy	5.6e+17	1.4e+03	0	16
DCT	1.5e+17	2.5e+02	0	16
DFT	2.0e+16	1.6e+00	0	16
RandN	7.5e+2	3.8e-14	14	2
RandN-C	4.5e+2	6.8e-14	14	2
RandU	8.6e+2	3.7e-14	14	2
RandU-C	5.7e+2	2.6e-14	14	2

codes. The reconstructed x from all existing codes lost all of their sixteen effective digits. However, the reconstructed x from the codes we proposed in the last section lost only about two effective digits.

5.3.2 Random Erasure Correction

For any N by N sub-matrix G_J of G, there is an erasure pattern of y which requires to solve a linear system with G_J as the coefficient matrix to reconstruct an approximation of the original x. A random erasure actually results in a randomly picked N by N sub-matrix of G. In Table 5.3, we compare the proportion of 100 by 100 sub-matrices whose condition number is larger than 10^i, where $i = 4, 6, 8$, and 10, for different kinds of generator matrices of size 150 by 100. All generator matrices are defined in Table 5.1. All results in Table 5.3 are calculated using MATLAB based on 1,000,000 randomly (uniformly) picked sub-matrices.

From Table 5.3, we can see, of the 1,000,000 randomly picked sub-matrices from

Table 5.3: Percentage of 100 by 100 sub-matrices (of a 150 by 100 generator matrix) whose condition number is larger than 10^i, where $i = 4, 6, 8$, and 10.

Name	$\kappa \geq 10^4$	$\kappa \geq 10^6$	$\kappa \geq 10^8$	$\kappa \geq 10^{10}$
Vander	100.000%	100.000%	100.000%	100.000%
Chebvand	100.000%	100.000%	100.000%	100.000%
Cauchy	100.000%	100.000%	100.000%	100.000%
DCT	96.187%	75.837%	48.943%	28.027%
DFT	92.853%	56.913%	21.644%	5.414%
RandN	1.994%	0.023%	0.000%	0.000%
RandN-C	0.033%	0.000%	0.000%	0.000%
RandU	1.990%	0.018%	0.000%	0.000%
RandU-C	0.036%	0.000%	0.000%	0.000%

any of our random generator matrices, there are 0.000% sub-matrices whose condition number is larger than 10^8. However, for all existing codes in literature that we have tested, there are at least 21.644% sub-matrices whose condition number is larger than 10^8. Therefore, our codes are much more stable than the existing codes in literature.

5.4 Conclusions and Future Work

In this chapter, we have introduced a class of real-number and complex-number codes based on random generator matrices over real-number and complex-number fields. We have compared our codes with existing codes in both burst erasure correction and random erasure correction. Experiment results demonstrate our codes are numerically much more stable than existing codes in the literature.

For the future, we will compare real-number codes based on different random matri-

ces with different probability distributions. We would also like to investigate what are the numerically optimal real number codes.

Chapter 6

Condition Numbers of Gaussian Random Matrices

In the study of real-number and complex-number error correction codes based on random matrices in Chapter 5 and their applications in fault tolerant high performance computing in Chapter 3 and 4, in order to estimate the numerical stability and reliability of our coding schemes, we need to estimate the probabilities that the condition numbers of small random rectangular matrices are large. For example, what is the probability that the condition number of a 10×5 random matrix is larger than 10^2?

In this chapter, we investigate the tails of the condition number distributions of random rectangular matrices whose elements are independent and identically distributed standard normal real or complex random variables. We establish upper and lower bounds for the tails of the condition number distributions of these matrices. Upper

bounds for the expected logarithms of the condition numbers of these matrices are also given.

Based on our results, for random rectangular matrices whose elements are independent and identically distributed standard normal real or complex random variables, we are able to estimate the probabilities that their condition numbers are large. For example, based on our results, we are able to tell, for a 10×5 real random matrix whose elements are independent and identically distributed standard normal random variables, the probability that the condition number is larger than 10^2 is less than 6×10^{-7}.

Our main results for the 2-norm condition number κ of an $m \times n$ real random matrix whose elements are independent and identically distributed standard normal random variables are:

$$\frac{1}{\sqrt{2\pi}} \left(\frac{c}{x}\right)^{|n-m|+1} < P\left(\frac{\kappa}{n/(|n-m|+1)} > x\right) < \frac{1}{\sqrt{2\pi}} \left(\frac{C}{x}\right)^{|n-m|+1},$$

and

$$E(\log \kappa) < \log \frac{n}{|n-m|+1} + 2.258,$$

where $0.245 \leq c \leq 2.000$ and $5.013 \leq C \leq 6.414$ are universal positive constants independent of m, n and x, and $m \geq 2$, $n \geq 2$ and $x \geq |n-m|+1$.

For an $m \times n$ complex random matrix whose elements are independent and identically distributed standard normal random variables, our main results for the 2-norm condition

number κ are:

$$\frac{1}{2\pi}\left(\frac{c}{x}\right)^{2(|n-m|+1)} < P\left(\frac{\kappa}{n/(|n-m|+1)} > x\right) < \frac{1}{2\pi}\left(\frac{C}{x}\right)^{2(|n-m|+1)},$$

and

$$E(\log\kappa) < \log\frac{n}{|n-m|+1} + 2.240,$$

where $0.319 \leq c \leq 2.000$ and $5.013 \leq C \leq 6.298$ are universal positive constants independent of m, n and x, and $m \geq 2$, $n \geq 2$ and $x \geq |n-m|+1$.

After finishing the manuscript of this work, we communicated with Edelman and learned that similar problems were also being studied independently by Edelman and Sutton [17]. After simple formatting, the upper bounds in both work actually can be unified into the same format

$$P\left(\kappa > x\right) \leq C(m,n,\beta)\left(\frac{1}{x}\right)^{\beta(|n-m|+1)},$$

where $\beta = 1$ for real random matrices and $\beta = 2$ for complex random matrices, and $C(m,n,\beta)$ is a function of m, n, and β. However, the function $C(m,n,\beta)$ in the two works do take very different forms and imply very different meanings.

On one hand, the bounds in [17] are asymptotically tight as $x \to \infty$ while the bounds in this work are not. On the other hand, the bounds in this work involve only elementary functions. Hence they are much simpler than the asymptotically tight bounds in [17] which involve high order moments of the largest eigenvalues of Wishart matrices.

Although for the special case of large square random matrices, simple estimations for $C(m, n, \beta)$ are given in [17], for general rectangular matrices, no simple estimation is available.

It is well-known that the joint eigenvalue density function of a Wishart matrix has a closed form expression [37]. Therefore, $P(\kappa > x)$ can actually be expressed *accurately* as a high dimensional integration of this joint eigenvalue density function. One of the key aspects to estimate $P(\kappa > x)$ is to find a simple-to-use estimation of this accurate (but not simple-to-use) high dimensional integral expression. This work is meaningful in that it finds out such a simple-to-use estimation by giving out simple upper and lower bounds which involve only elementary functions. We refer interested readers to [17] for more accurate asymptotically tight bounds and other related bounds for the tails of the condition numbers of general β-Laguerre ensembles.

Above and in what follows in this chapter, the constant C and c denote universal positive constants independent of m, n and x; however, identical symbols may represent different numbers in different places.

6.1 Preliminaries and Basic Facts

Let X be an $m \times n$ matrix. If $\sigma_1 \geq \sigma_2 \geq ... \geq \sigma_p$, where $p = \min\{m, n\}$, are the p singular values of X, then the *2-norm condition number* of X is

$$\kappa_2(X) = \frac{\sigma_1}{\sigma_p}.$$

For any $m \times n$ matrix X, X^T is an $n \times m$ matrix and $\kappa_2(X) = \kappa_2(X^T)$. So, without loss of generality, in discussing the condition numbers of random matrices, it is enough to only consider random matrices with no more rows than columns. Therefore, from now on, when we speak of an $m \times n$ matrix, we will assume $m \leq n$ in the rest of this chapter.

Let $G_{m\times n}$ be an $m \times n$ real random matrix whose elements are independent and identically distributed standard normal random variables. Let $W_{m,n}$ denote the $m \times m$ random matrix $G_{m\times n}G_{m\times n}^T$. $W_{m,n}$ is the well known Wishart matrix named after John Wishart who first studied its distribution.

Similar to [15], in this chapter, we will study the condition number of $G_{m\times n}$ through investigating the eigenvalues of the Wishart matrix $W_{m,n}$. The following lemma establishes a simple relationship between the condition number of $G_{m\times n}$ and the eigenvalues of $W_{m,n}$.

Proposition 5 *If λ_{max} is the largest eigenvalue of $W_{m,n}$, and λ_{min} is the smallest eigenvalue of $W_{m,n}$, then the 2-norm condition number of $G_{m\times n}$ satisfies*

$$\kappa_2(G_{m\times n}) = \sqrt{\frac{\lambda_{max}}{\lambda_{min}}}.$$

Remarkably enough, the exact joint probability density function for the m eigenvalues of the Wishart matrix $W_{m,n}$ can be written down in a closed form [37].

Lemma 6 *If $\lambda_1 \geq ... \geq \lambda_m$ are the m eigenvalues of $W_{m,n}$, then the joint probability*

density function of $\lambda_1 \geq ... \geq \lambda_m$ *is*

$$f(x_1, ..., x_m) = K_{m,n} e^{-\frac{1}{2}\sum_{i=1}^{m} x_i} \prod_{i=1}^{m} x_i^{\frac{1}{2}(n-m-1)} \prod_{i=1}^{m-1} \prod_{j=i+1}^{m} (x_i - x_j), \tag{6.1}$$

where

$$K_{m,n}^{-1} = \left(\frac{2^n}{\pi}\right)^{m/2} \prod_{i=1}^{m} \Gamma\left(\frac{n-m+i}{2}\right) \Gamma\left(\frac{i}{2}\right). \tag{6.2}$$

Let $N(0,1)$ denote the standard normal distribution. Let $\widetilde{N}(0,1)$ denote the distribution of $u + iv$, where u and v are independent and identically distributed $N(0,1)$ random variables, and $i = \sqrt{-1}$. Let $\widetilde{G}_{m\times n}$ be an $m \times n$ complex random matrix whose elements are independent and identically distributed $\widetilde{N}(0,1)$ random variables. Let $\widetilde{W}_{m,n}$ denote the $m \times m$ random matrix $\widetilde{G}_{m\times n}\widetilde{G}_{m\times n}^H$. In literature, $\widetilde{W}_{m,n}$ is called the complex Wishart matrix.

Similar to the real case, there is also a simple relationship between the condition number of $\widetilde{G}_{m\times n}$ and the eigenvalues of $\widetilde{W}_{m,n}$.

Proposition 7 *If* $\widetilde{\lambda}_{max}$ *is the largest eigenvalue of* $\widetilde{W}_{m,n}$*, and* $\widetilde{\lambda}_{min}$ *is the smallest eigenvalue of* $\widetilde{W}_{m,n}$*, then the 2-norm condition number of* $\widetilde{G}_{m\times n}$ *satisfies*

$$\kappa_2(\widetilde{G}_{m\times n}) = \sqrt{\frac{\widetilde{\lambda}_{max}}{\widetilde{\lambda}_{min}}}.$$

Like the real case, the exact joint probability density function for the m eigenvalues of the complex Wishart matrix $\widetilde{W}_{m,n}$ can also be written down in a closed form [37].

Lemma 8 *If $\widetilde{\lambda}_1 \geq ... \geq \widetilde{\lambda}_m$ are the m eigenvalues of $\widetilde{W}_{m,n}$, then the joint probability density function of $\widetilde{\lambda}_1 \geq ... \geq \widetilde{\lambda}_m$ is*

$$\widetilde{f}(x_1, ..., x_m) = \widetilde{K}_{m,n} e^{-\frac{1}{2}\sum_{i=1}^{m} x_i} \prod_{i=1}^{m} x_i^{n-m} \prod_{i=1}^{m-1} \prod_{j=i+1}^{m} (x_i - x_j)^2, \tag{6.3}$$

where

$$\widetilde{K}_{m,n}^{-1} = 2^{mn} \prod_{i=1}^{m} \Gamma(n-m+i)\,\Gamma(i). \tag{6.4}$$

In the process of deriving our upper and lower bounds for the tails of the condition number distributions, some bounds for Gamma and incomplete Gamma functions are very useful.

Lemma 9 *Assume $a > 0$, and $b > 0$. If $t \leq \frac{b}{a}$, then*

$$\int_0^t e^{-ax} x^b dx \leq e^{-at} t^{b+1}.$$

Proof. Let $f(t) = \int_0^t e^{-ax} x^b dx - e^{-at} t^{b+1}$, then $f'(t) = e^{-at} t^b (1 + at - (b+1))$. So $f(t)$ decreases on $[0, \frac{b}{a}]$ and increases on $[\frac{b}{a}, \infty)$. Since $f(0) = 0$, and $f(\infty) = \int_0^\infty e^{-ax} x^b dx > 0$, if $t \leq \frac{b}{a}$, then $f(t) < 0$. Therefore, if $t \leq \frac{b}{a}$, then $\int_0^t e^{-ax} x^b dx \leq e^{-at} t^{b+1}$. □

Lemma 10 *Assume $a > 0$, $b > 0$, and $k > \frac{1}{a}$. If $t \geq \frac{kb}{ka-1}$, then*

$$\int_t^\infty e^{-ax} x^b dx \leq k e^{-at} t^b.$$

Proof. Let $f(t) = \int_t^\infty e^{-ax} x^b dx - k e^{-at} t^b$, then $f'(t) = e^{-at} t^b (-1 + ka - \frac{kb}{t})$. So $f(t)$

decreases on $[0, \frac{kb}{ka-1}]$ and increases on $[\frac{kb}{ka-1}, \infty)$. Since $f(0) = \int_0^\infty e^{-ax} x^b dx > 0$, and $f(\infty) = 0$. So, if $t \geq \frac{kb}{ka-1}$, then $f(t) < 0$. Therefore, if $t \leq \frac{kb}{ka-1}$, then $\int_t^\infty e^{-ax} x^b dx \leq ke^{-at} t^b$. □

Lemma 11 *If* $\Gamma(x) = \int_0^\infty e^{-t} t^{x-1} dt$, *where* $x > 0$, *then*

$$\sqrt{2\pi} x^{x+\frac{1}{2}} e^{-x} < \Gamma(x+1) < \sqrt{2\pi} x^{x+\frac{1}{2}} e^{-x+\frac{1}{12x}}, \tag{6.5}$$

and

$$\Gamma(x + \frac{1}{2}) < \Gamma(x)\sqrt{x}. \tag{6.6}$$

Proof. (6.5) follows straightforwardly from 6.1.38 in [42], and (6.6) can be obtained from the answer to Problem 9.60 in [30]. □

6.2 Bounds for Eigenvalue Densities of Wishart Matrices

In this section, we will prove some bounds for the probability density functions of the eigenvalues of Wishart matrices. These bounds are very useful in the derivation of the bounds for the tails of the condition number distributions.

Let λ_{max} denote the largest eigenvalue of $W_{m,n}$, and λ_{min} denote the smallest eigenvalue of $W_{m,n}$. In the following lemma, we prove an upper bound for the joint probability density function of λ_{max} and λ_{min}.

Lemma 12 *Let* $f_{\lambda_{max},\lambda_{min}}(x, y)$ *denote the joint probability density function of* λ_{max}

and λ_{min}. *Then* $f_{\lambda_{max},\lambda_{min}}(x,y)$ *satisfies:*

$$f_{\lambda_{max},\lambda_{min}}(x,y) \leq C_{m,n} e^{-\frac{1}{2}(x+y)} x^{\frac{1}{2}(n+m-3)} y^{\frac{1}{2}(n-m-1)}, \tag{6.7}$$

where

$$C_{m,n} = \frac{1}{4\Gamma(m-1)\Gamma(n-m+1)}. \tag{6.8}$$

Proof. Let $R_{x,y} = \{(x_2, x_3, ..., x_{m-1}) : x \geq x_2 \geq ... \geq x_{m-1} \geq y\} \subseteq R^{m-2}$. From the joint probability density function of the m eigenvalues of $W_{m,n}$ in Lemma 6, we have

$$\begin{aligned} f_{\lambda_{max},\lambda_{min}}(x,y) &= \int_{R_{x,y}} f(x, x_2, ..., x_{m-1}, y) dx_2 dx_3 ... dx_{m-1} \\ &= K_{m,n} e^{-\frac{1}{2}(x+y)} x^{\frac{1}{2}(n-m-1)} y^{\frac{1}{2}(n-m-1)} \\ &\quad \int_{R_{x,y}} e^{-\frac{1}{2}\sum_{i=2}^{m-1} x_i} \prod_{i=2}^{m-1} x_i^{\frac{1}{2}(n-m-1)} \\ &\quad (x-y) \prod_{i=2}^{m-1} (x - x_i)(x_i - y) \prod_{i=2}^{m-2} \prod_{j=i+1}^{m-1} (x_i - x_j) \prod_{i=2}^{m-1} dx_i. \end{aligned} \tag{6.9}$$

Let $R_{m-2} = \{(x_2, x_3, ..., x_{m-1}) : x_2 \geq ... \geq x_{m-1} \geq 0\}$, then $R_{m-2} \subseteq R_{x,y}$. Note that, in (6.9), $x \geq x_i \geq y$ for $i = 2, 3, ..., m-1$. Replacing $x - y$ and $x - x_i$ by x, and

$x_i - y$ by x_i for $i = 2, 3, ..., m-1$, and $R_{x,y}$ by R_{m-2}, then we get

$$\begin{aligned} f_{\lambda_{max},\lambda_{min}}(x,y) &\leq K_{m,n} e^{-\frac{1}{2}(x+y)} x^{\frac{1}{2}(n+m-3)} y^{\frac{1}{2}(n-m-1)} \\ &\int_{R_{m-2}} e^{-\frac{1}{2}\sum_{i=2}^{m-1} x_i} \prod_{i=2}^{m-1} x_i^{\frac{1}{2}(n-m+1)} \prod_{i=2}^{m-2} \prod_{j=i+1}^{m-1} (x_i - x_j) \\ &\prod_{i=2}^{m-1} dx_i. \end{aligned} \tag{6.10}$$

Note that $f(x_1, x_2, ..., x_m)$ in (6.1) is a probability density function, therefore, for any $m \leq n$, we have

$$\int_{R_m} e^{-\frac{1}{2}\sum_{i=1}^{m} x_i} \prod_{i=1}^{m} x_i^{\frac{1}{2}(n-m-1)} \prod_{i=1}^{m-1} \prod_{j=i+1}^{m} (x_i - x_j) \prod_{i=1}^{m} dx_i = K_{m,n}^{-1},$$

where $R_m = \{x_1 \geq x_2 \geq ... \geq x_m \geq 0\} \subseteq R^m$. Therefore, we have

$$\int_{R_{m-2}} e^{-\frac{1}{2}\sum_{i=2}^{m-1} x_i} \prod_{i=2}^{m-1} x_i^{\frac{1}{2}(n-m+1)} \prod_{i=2}^{m-2} \prod_{j=i+1}^{m-1} (x_i - x_j) \prod_{i=2}^{m-1} dx_i = K_{m-2,n}^{-1}. \tag{6.11}$$

Substitute (6.11) into (6.10), we obtain

$$f_{\lambda_{max},\lambda_{min}}(x,y) \leq \frac{K_{m,n}}{K_{m-2,n}} e^{-\frac{1}{2}(x+y)} x^{\frac{1}{2}(n+m-3)} y^{\frac{1}{2}(n-m-1)}. \tag{6.12}$$

From (6.2), we have

$$\begin{aligned}\frac{K_{m,n}}{K_{m-2,n}} &= \frac{\pi}{2^n}\frac{1}{\Gamma\left(\frac{m-1}{2}\right)\Gamma\left(\frac{m}{2}\right)\Gamma\left(\frac{n-m+1}{2}\right)\Gamma\left(\frac{n-m+2}{2}\right)} \\ &= \frac{1}{4\Gamma\left(m-1\right)\Gamma\left(n-m+1\right)}.\end{aligned} \tag{6.13}$$

Substituting (6.13) into (6.12), we get (6.7) and (6.8) . □

Let $\widetilde{\lambda}_{max}$ denote the largest eigenvalue of $\widetilde{W}_{m,n}$, and $\widetilde{\lambda}_{min}$ denote the smallest eigenvalue of $\widetilde{W}_{m,n}$. Similar to the real case, in the following lemma, we give an upper bound for the joint probability density function of $\widetilde{\lambda}_{max}$ and $\widetilde{\lambda}_{min}$. The upper bound in complex case can be proved using the same techniques used in the real case. Therefore, we omit the proof and only give the result here.

Lemma 13 *Let $\widetilde{f}_{\widetilde{\lambda}_{max},\widetilde{\lambda}_{min}}(x,y)$ denote the joint probability density function of $\widetilde{\lambda}_{max}$ and $\widetilde{\lambda}_{min}$ Then $\widetilde{f}_{\widetilde{\lambda}_{max},\widetilde{\lambda}_{min}}(x,y)$ satisfies:*

$$\widetilde{f}_{\widetilde{\lambda}_{max},\widetilde{\lambda}_{min}}(x,y) \leq \widetilde{C}_{m,n}e^{-\frac{1}{2}(x+y)}x^{n+m-2}y^{n-m}, \tag{6.14}$$

where

$$\widetilde{C}_{m,n} = \frac{1}{2^{2n}\Gamma(m-1)\Gamma(m)\Gamma(n-m+1)\Gamma(n-m+2)}. \tag{6.15}$$

Bounds for the probability density functions of the smallest eigenvalues are also very useful in the derivation of the bounds for the tails of the condition number distributions.

In the following lemma, we prove upper and lower bounds for the probability density function of the smallest eigenvalue of a real Wishart matrix.

Lemma 14 *Let $f_{\lambda_{min}}(x)$ denote the probability density function of the smallest eigenvalue of $W_{m,n}$. Then $f_{\lambda_{min}}(x)$ satisfies:*

$$L_{m,n}e^{-\frac{m}{2}x}x^{\frac{1}{2}(n-m-1)} \leq f_{\lambda_{min}}(x) \leq L_{m,n}e^{-\frac{1}{2}x}x^{\frac{1}{2}(n-m-1)}, \tag{6.16}$$

where

$$L_{m,n} = \frac{2^{\frac{n-m-1}{2}}\Gamma\left(\frac{n+1}{2}\right)}{\Gamma\left(\frac{m}{2}\right)\Gamma\left(n-m+1\right)}. \tag{6.17}$$

Proof. Let $R_x = \{(x_1, x_2, ..., x_{m-1}) : x_1 \geq ... \geq x_{m-1} \geq x\} \subseteq R^{m-1}$. From the joint probability density function of the eigenvalues of $W_{m,n}$ in Lemma 6, we have

$$\begin{aligned} f_{\lambda_{min}}(x) &= \int_{R_x} f(x_1, x_2, ..., x_{m-1}, x)dx_1dx_2dx_{m-1} \\ &= K_{m,n}e^{-\frac{1}{2}x}x^{\frac{1}{2}(n-m-1)}\int_{R_x} e^{-\frac{1}{2}\sum_{i=1}^{m-1}x_i}\prod_{i=1}^{m-1}x_i^{\frac{1}{2}(n-m-1)} \\ &\quad \prod_{i=1}^{m-1}(x_i - x)\prod_{i=1}^{m-2}\prod_{j=i+1}^{m-1}(x_i - x_j)\prod_{i=1}^{m-1}dx_i. \end{aligned}$$

For the lower bound part, taking the transformation $y_i = x_i - x$, where $i =$

$1, 2, ..., m-1$, we have

$$f_{\lambda_{min}}(x) = K_{m,n} e^{-\frac{m}{2}x} x^{\frac{1}{2}(n-m-1)} \int_{R_y} e^{-\frac{1}{2}\sum_{i=1}^{m-1} y_i} \prod_{i=1}^{m-1} (y_i + x)^{\frac{1}{2}(n-m-1)} \prod_{i=1}^{m-1} y_i \prod_{i=1}^{m-2} \prod_{j=i+1}^{m-1} (y_i - y_j) \prod_{i=1}^{m-1} dy_i,$$

where $R_y = \{y_1 \geq y_2 \geq ... \geq y_{m-1} \geq 0\} \subseteq R^{m-1}$.

Replacing $y_i + x$ by y_i for $i = 1, 2, ..., m-1$, we obtain

$$f_{\lambda_{min}}(x) \geq K_{m,n} e^{-\frac{m}{2}x} x^{\frac{1}{2}(n-m-1)} \int_{R_y} e^{-\frac{1}{2}\sum_{i=1}^{m-1} y_i} \prod_{i=1}^{m-1} y_i^{\frac{1}{2}(n-m+1)} \prod_{i=1}^{m-2} \prod_{j=i+1}^{m-1} (y_i - y_j) \prod_{i=1}^{m-1} dy_i.$$

Note that

$$\int_{R_y} e^{-\frac{1}{2}\sum_{i=1}^{m-1} y_i} \prod_{i=1}^{m-1} y_i^{\frac{1}{2}(n-m+1)} \prod_{i=1}^{m-2} \prod_{j=i+1}^{m-1} (y_i - y_j) \prod_{i=1}^{m-1} dy_i = K_{m-1,n+1}^{-1}.$$

Therefore, we obtain

$$f_{\lambda_{min}}(x) \geq \frac{K_{m,n}}{K_{m-1,n+1}} e^{-\frac{m}{2}x} x^{\frac{1}{2}(n-m-1)}. \tag{6.18}$$

For the upper bound part, from [14], we have

$$f_{\lambda_{min}}(x) \leq \frac{K_{m,n}}{K_{m-1,n+1}} e^{-\frac{1}{2}x} x^{\frac{1}{2}(n-m-1)}. \tag{6.19}$$

From (6.2), we have

$$\begin{aligned}\frac{K_{m,n}}{K_{m-1,n+1}} &= \frac{\sqrt{\pi}\left(\frac{1}{2}\right)^{\frac{n-m+1}{2}}\Gamma\left(\frac{n+1}{2}\right)}{\Gamma\left(\frac{m}{2}\right)\Gamma\left(\frac{n-m+1}{2}\right)\Gamma\left(\frac{n-m+2}{2}\right)} \\ &= \frac{2^{\frac{n-m-1}{2}}\Gamma\left(\frac{n+1}{2}\right)}{\Gamma\left(\frac{m}{2}\right)\Gamma\left(n-m+1\right)}.\end{aligned} \tag{6.20}$$

Substitute (6.20) into (6.19) and (6.18), we get (6.16) and (6.17) . □

Similar to the real case, in the following lemma, we give upper and lower bounds for the probability density function of the smallest eigenvalue $\widetilde{\lambda}_{min}$ of a complex Wishart matrix. These bounds can be proved using the same techniques used in the real case. Therefore, we omit the proof and only give the result here.

Lemma 15 *Let $\widetilde{f}_{\widetilde{\lambda}_{min}}(x)$ denotes the probability density function of the smallest eigenvalue of $\widetilde{W}_{m,n}$, then $\widetilde{f}_{\widetilde{\lambda}_{min}}(x)$ satisfies:*

$$\widetilde{L}_{m,n}e^{-\frac{m}{2}x}x^{n-m} \leq \widetilde{f}_{\widetilde{\lambda}_{min}}(x) \leq \widetilde{L}_{m,n}e^{-\frac{1}{2}x}x^{n-m}, \tag{6.21}$$

where

$$\widetilde{L}_{m,n} = \frac{\Gamma(n+1)}{2^{n-m+1}\Gamma(m)\Gamma(n-m+1)\Gamma(n-m+2)}. \tag{6.22}$$

6.3 The Upper Bounds for the Distribution Tails

In this section, we will derive the upper bounds for the tails of the condition number distributions of random rectangular matrices whose elements are independent and identically distributed standard normal random variables. Our main results are Theorem 20 for real random matrices, and Theorem 21 for complex random matrices.

Lemma 16 *For any $A > 0$, $x > 0$, and $n \geq m \geq 2$, the largest eigenvalue λ_{max} and the smallest eigenvalue λ_{min} of $W_{m,n}$ satisfy*

$$P\left(\frac{\lambda_{max}}{\lambda_{min}} > x^2, \lambda_{min} \leq \frac{A^2 n}{x^2}\right) < \frac{1}{\Gamma(n-m+2)}\left(\frac{An}{x}\right)^{n-m+1}.$$

Proof. From the upper bound for the probability density function of λ_{min} in Lemma 14, we have

$$\begin{aligned} P\left(\frac{\lambda_{max}}{\lambda_{min}} > x^2, \lambda_{min} \leq \frac{A^2 n}{x^2}\right) &< P\left(\lambda_{min} \leq \frac{A^2 n}{x^2}\right) \\ &= \int_0^{\frac{A^2 n}{x^2}} f_{\lambda_{min}}(t)dt \\ &< L_{m,n}\int_0^{\frac{A^2 n}{x^2}} t^{\frac{1}{2}(n-m-1)}dt \\ &= \frac{\Gamma\left(\frac{n+1}{2}\right)}{\Gamma\left(\frac{m}{2}\right)\left(\frac{n}{2}\right)^{\frac{n-m+1}{2}}}\frac{1}{\Gamma(n-m+2)}\left(\frac{An}{x}\right)^{n-m+1}. \end{aligned}$$

Since $m \leq n$, by applying (6.6) repeatedly, we can prove

$$\Gamma\left(\frac{m}{2}\right)\left(\frac{n}{2}\right)^{\frac{n-m+1}{2}} > \Gamma\left(\frac{n+1}{2}\right).$$

Therefore, we have

$$P\left(\frac{\lambda_{max}}{\lambda_{min}} > x^2, \lambda_{min} \leq \frac{A^2 n}{x^2}\right) \quad < \quad \frac{1}{\Gamma(n-m+2)}\left(\frac{An}{x}\right)^{n-m+1}.$$

□

Similar to real random matrices, for complex random matrices, we have the following Lemma 17. Lemma 17 can be proved using the same techniques as Lemma 16, so we will omit the proof and only give the result.

Lemma 17 *For any $A > 0$, $x > 0$, and $n \geq m \geq 2$, the largest eigenvalue $\widetilde{\lambda}_{max}$ and the smallest eigenvalue $\widetilde{\lambda}_{min}$ of $\widetilde{W}_{m,n}$ satisfy*

$$P\left(\frac{\widetilde{\lambda}_{max}}{\widetilde{\lambda}_{min}} > x^2, \widetilde{\lambda}_{min} \leq \frac{A^2 n}{x^2}\right) < \frac{1}{\Gamma(n-m+2)^2}\left(\frac{A^2 n^2}{2x^2}\right)^{n-m+1}.$$

The proof of the following Lemma 18 is based on the upper bound for the joint probability density function of λ_{max} and λ_{min} in Lemma 12 and the upper bound of the incomplete Gamma function in Lemma 10.

Lemma 18 *For any $A \geq 2.32$, $x > 0$, and $n \geq m \geq 2$, the largest eigenvalue λ_{max} and the smallest eigenvalue λ_{min} of $W_{m,n}$ satisfy*

$$P\left(\frac{\lambda_{max}}{\lambda_{min}} > x^2, \lambda_{min} > \frac{A^2 n}{x^2}\right) < 0.017\frac{1}{\Gamma(n-m+2)}\left(\frac{An}{x}\right)^{n-m+1}.$$

Proof. From the upper bound for the joint probability density function of λ_{max} and λ_{min} in Lemma 12, we have

$$\begin{aligned} P\left(\frac{\lambda_{max}}{\lambda_{min}} > x^2, \lambda_{min} > \frac{A^2 n}{x^2}\right) &= \int_{\frac{A^2 n}{x^2}}^{\infty} \int_{tx^2}^{\infty} f_{\lambda_{max},\lambda_{min}}(s,t) ds dt \\ &< \int_{\frac{A^2 n}{x^2}}^{\infty} \int_{tx^2}^{\infty} C_{m,n} e^{-\frac{1}{2}t} t^{\frac{1}{2}(n-m-1)} e^{-\frac{1}{2}s} s^{\frac{1}{2}(n+m-3)} ds dt. \end{aligned}$$

Taking the transform $u = tx^2$, we have

$$\begin{aligned} P\left(\frac{\lambda_{max}}{\lambda_{min}} > x^2, \lambda_{min} > \frac{A^2 n}{x^2}\right) &= C_{m,n} \left(\frac{1}{x}\right)^{n-m+1} \int_{A^2 n}^{\infty} e^{-\frac{u}{2x^2}} u^{\frac{1}{2}(n-m-1)} \\ &\quad \left(\int_u^{\infty} e^{-\frac{1}{2}s} s^{\frac{1}{2}(n+m-3)} ds\right) du. \end{aligned}$$

According to Lemma 10, with $k = 4$,if $u \geq 2(n+m-3)$, then

$$\int_u^{\infty} e^{-\frac{1}{2}s} s^{\frac{1}{2}(n+m-3)} ds \leq 4 e^{-\frac{1}{2}u} u^{\frac{1}{2}(n+m-3)}.$$

Since $A \geq 2.32$ and $n \geq m$, hence, $u \geq A^2 n \geq 2(n+m-3)$. Therefore, we have

$$\begin{aligned} P\left(\frac{\lambda_{max}}{\lambda_{min}} > x^2, \lambda_{min} > \frac{A^2 n}{x^2}\right) &\leq 4 C_{m,n} \left(\frac{1}{x}\right)^{n-m+1} \int_{A^2 n}^{\infty} e^{-\frac{u}{2x^2} - \frac{1}{2}u} u^{n-2} du \\ &\leq 4 C_{m,n} \left(\frac{1}{x}\right)^{n-m+1} \int_{A^2 n}^{\infty} e^{-\frac{1}{2}u} u^{n-2} du. \end{aligned}$$

Since $A \geq 2.32$, so $A^2 n \geq 4(n-2)$. Apply Lemma 10 again, we have

$$\begin{aligned} P\left(\frac{\lambda_{max}}{\lambda_{min}} > x^2, \lambda_{min} > \frac{A^2 n}{x^2}\right) &\leq 16 C_{m,n} e^{-\frac{1}{2}A^2 n} A^{2n-4} n^{n-2} \left(\frac{1}{x}\right)^{n-m+1} \\ &= \frac{4e^{-\frac{A^2 n}{2}} A^{2n-4} n^{m-3}}{\Gamma(m-1)\Gamma(n-m+1)} \left(\frac{n}{x}\right)^{n-m+1} \\ &\leq \frac{4e^{(2\ln A - \frac{A^2}{2})n}}{A^4} \frac{n^{m-2}}{\Gamma(m-1)} \frac{1}{\Gamma(n-m+2)} \left(\frac{n}{x}\right)^{n-m+1} \end{aligned} \tag{6.23}$$

Note that, for any $2 \leq m \leq n$, it can be proved that

$$\frac{n^{m-2}}{\Gamma(m-1)} < \frac{e^n}{\sqrt{4\pi}}. \tag{6.24}$$

Substitute (6.24) into (6.23), we have

$$P\left(\frac{\lambda_{max}}{\lambda_{min}} > x^2, \lambda_{min} > \frac{A^2 n}{x^2}\right) \leq \frac{4e^{(2\ln A - \frac{A^2}{2}+1)n}}{\sqrt{4\pi} A^4} \frac{1}{\Gamma(n-m+2)} \left(\frac{n}{x}\right)^{n-m+1}.$$

Since $A \geq 2.32$, therefore, we have

$$e^{(2\ln A - \frac{A^2}{2}+1)n} < 1.$$

Therefore, when $A \geq 2.32$, we have

$$\begin{aligned} P\left(\frac{\lambda_{max}}{\lambda_{min}} > x^2, \lambda_{min} > \frac{A^2 n}{x^2}\right) &\leq \frac{4}{\sqrt{4\pi} A^4} \frac{1}{\Gamma(n-m+2)} \left(\frac{n}{x}\right)^{n-m+1} \\ &\leq 0.017 \frac{1}{\Gamma(n-m+2)} \left(\frac{An}{x}\right)^{n-m+1}. \end{aligned}$$

□

Similar to real random matrices, for complex random matrices, we have the following Lemma 19. Lemma 19 can be proved using the same techniques as Lemma 18, so we will omit the proof and only give the result.

Lemma 19 *For any $A \geq 3.2735$, $x > 0$, and $n \geq m \geq 2$, the largest eigenvalue $\widetilde{\lambda}_{max}$ and the smallest eigenvalue $\widetilde{\lambda}_{min}$ of $\widetilde{W}_{m,n}$ satisfy*

$$P\left(\frac{\widetilde{\lambda}_{max}}{\widetilde{\lambda}_{min}} > x^2, \widetilde{\lambda}_{min} > \frac{A^2 n}{x^2}\right) < 0.0016 \frac{1}{\Gamma(n-m+2)^2}\left(\frac{A^2 n^2}{2x}\right)^{n-m+1}.$$

We are now prepared to prove our first main result about the condition numbers of real random matrices whose elements are independent and identically distributed standard normal random variables.

Theorem 20 *For any $n \geq m \geq 2$ and $x \geq n - m + 1$, the 2-norm condition number of $G_{m\times n}$ satisfies*

$$P\left(\frac{\kappa_2(G_{m\times n})}{n/(n-m+1)} > x\right) < \frac{1}{\sqrt{2\pi}}\left(\frac{C}{x}\right)^{n-m+1}, \tag{6.25}$$

where $C \leq 6.414$ is a universal positive constant independent of m, n, and x.

Proof. For any $L > 0$, inspired by [3], we first break down $P(\kappa_2(G_{m\times n}) > x)$ into two parts.

$$\begin{aligned} P(\kappa_2(G_{m\times n}) > x) &= P\left(\frac{\lambda_{max}}{\lambda_{min}} > x^2\right) \\ &= P\left(\frac{\lambda_{max}}{\lambda_{min}} > x^2, \lambda_{min} \le \frac{L^2 n}{x^2}\right) + P\left(\frac{\lambda_{max}}{\lambda_{min}} > x^2, \lambda_{min} > \frac{L^2 n}{x^2}\right). \end{aligned}$$

Let $L = 2.32$, then based on Lemma 16 and Lemma 18, we can get

$$\begin{aligned} P(\kappa_2(G_{m\times n}) > x) &< \frac{1}{\Gamma(n-m+2)}\left(\frac{Ln}{x}\right)^{n-m+1} \\ &\quad +0.017\frac{1}{\Gamma(n-m+2)}\left(\frac{Ln}{x}\right)^{n-m+1} \\ &< \frac{1}{\Gamma(n-m+2)}\left(\frac{1.017Ln}{x}\right)^{n-m+1}. \end{aligned}$$

Note that, from Lemma 11, we have

$$\Gamma(n-m+2) > \sqrt{2\pi(n-m+1)}(n-m+1)^{n-m+1}e^{-(n-m+1)}.$$

Therefore, we have

$$P(\kappa_2(G_{m\times n}) > x) < \frac{1}{\sqrt{2\pi(n-m+1)}}\left(\frac{1.017eL\frac{n}{n-m+1}}{x}\right)^{n-m+1}.$$

Therefore

$$\begin{aligned} P\left(\frac{\kappa_2(G_{m\times n})}{n/(n-m+1)} > x\right) &< \frac{1}{\sqrt{2\pi(n-m+1)}}\left(\frac{1.017eL}{x}\right)^{n-m+1} \\ &< \frac{1}{\sqrt{2\pi}}\left(\frac{6.414}{x}\right)^{n-m+1}. \end{aligned}$$

Let $C = 6.414$, then we get (6.25). □

Remark:

1. The upper bound in Theorem 20 is for arbitrary $n \geq m \geq 2$ and $x \geq n-m+1$. For some special case of m and n, more precise upper bound can be obtained. For example, for the special case of real random $2\times n$ matrices, based on the exact probability density function of $\kappa_2(G_{2\times n})$ in [15], we can get

$$P\left(\kappa_2(G_{2\times n}) > x\right) = \left(\frac{2x}{x^2+1}\right)^{n-1} < \left(\frac{2}{x}\right)^{n-1}.$$

2. For the special case of real random $m \times m$ matrices, where $m \geq 3$, it has been proved in [3] that

$$P\left(\kappa_2(G_{m\times m}) > m\,.\,x\right) < \frac{C'}{x}, \tag{6.26}$$

where $C' \leq 5.60$ is a universal positive constant independent of x and m.

In Theorem 20, if we take $m = n$, then we have

$$P\left(\kappa_2(G_{m\times m}) > m\,.\,x\right) < \frac{2.60}{x},$$

which is consistent with (6.26) except that we improved the upper bound for the constant C' from 5.60 to 2.60. From the following (6.27), we know that the constant C' in (6.26) actually must at least be 2.

3. For the special case of large real random $m \times m$ matrices, it has been proved in [15] that

$$\lim_{m\to\infty} P\left(\frac{\kappa_2(G_{m\times m})}{m} < x\right) = e^{-\frac{2}{x}-\frac{2}{x^2}}.$$

Therefore, we have

$$\lim_{m\to\infty} P\left(\frac{\kappa_2(G_{m\times m})}{m} > x\right) = 1 - e^{-\frac{2}{x}-\frac{2}{x^2}} \sim \frac{2}{x} \tag{6.27}$$

as $x \to \infty$. Hence, the smallest possible universal constant C in Theorem 20 must be no smaller than $2\sqrt{2\pi}$. Therefore, the universal constant C in Theorem 20 actually must satisfy

$$C \geq 2\sqrt{2\pi} \approx 5.013. \tag{6.28}$$

Similar to real random matrices, for complex random matrices, we have the following Theorem 21. Theorem 21 can be proved using the same techniques as Theorem 20, so we will omit the proof and only give the result.

Theorem 21 *For any $n \geq m \geq 2$ and $x \geq n - m + 1$, the 2-norm condition number of $\widetilde{G}_{m\times n}$ satisfies*

$$P\left(\frac{\kappa_2(\widetilde{G}_{m\times n})}{n/(n-m+1)} > x\right) < \frac{1}{2\pi}\left(\frac{\widetilde{C}}{x}\right)^{2(n-m+1)},$$

where $\widetilde{C} \leq 6.298$ is a universal positive constant independent of x, m, and n.

6.4 The Lower Bounds for the Distribution Tails

In this section, we will prove the lower bounds for the tails of the condition number distributions of random rectangular matrices whose elements are independent and identically distributed standard normal random variables. Our main results are Theorem 26 for real random matrices, and Theorem 27 for complex random matrices.

Lemma 22 *For any $B > 0$, $x > 0$, and $n \geq m \geq 2$, the smallest eigenvalue λ_{min} of $W_{m,n}$ satisfies*

$$P\left(\lambda_{min} \leq \frac{B^2 n}{x^2}\right) > \sqrt{\frac{2e^{\frac{5}{6}}}{3}} e^{-\frac{B^2 mn}{2x^2}} \frac{1}{\Gamma(n-m+2)} \left(\frac{e^{-\frac{1}{2}} Bn}{x}\right)^{n-m+1}.$$

Proof. From the lower bound for the probability density function of λ_{min} in Lemma 14, we have

$$\begin{aligned}
P\left(\lambda_m \leq \frac{B^2 n}{x^2}\right) &= \int_0^{\frac{B^2 n}{x^2}} f(\lambda_m) d\lambda_m \\
&> \int_0^{\frac{B^2 n}{x^2}} L_{m,n} e^{-\frac{m}{2}\lambda_m} \lambda_m^{\frac{1}{2}(n-m-1)} d\lambda_m \\
&> L_{m,n} e^{-\frac{B^2 mn}{2x^2}} \int_0^{\frac{B^2 n}{x^2}} \lambda_m^{\frac{1}{2}(n-m-1)} d\lambda_m \\
&= L_{m,n} e^{-\frac{B^2 mn}{2x^2}} \frac{2n^{\frac{n-m+1}{2}}}{n-m+1} \left(\frac{B}{x}\right)^{n-m+1} \\
&= e^{-\frac{B^2 mn}{2x^2}} \frac{\Gamma\left(\frac{n+1}{2}\right)}{\Gamma\left(\frac{m}{2}\right)\left(\frac{n}{2}\right)^{\frac{n-m+1}{2}}} \frac{1}{\Gamma(n-m+2)} \left(\frac{Bn}{x}\right)^{n-m+1}.
\end{aligned}$$

Note that

$$\frac{n+1}{2}\Gamma\left(\frac{n+1}{2}\right) > \sqrt{2\pi}\left(\frac{n+1}{2}\right)^{\frac{n+2}{2}} e^{-\frac{n+1}{2}},$$

and

$$\frac{m}{2}\Gamma\left(\frac{m}{2}\right) < \sqrt{2\pi}\left(\frac{m}{2}\right)^{\frac{m+1}{2}} e^{-\frac{m}{2}+\frac{1}{6m}}.$$

Therefore

$$\begin{aligned}\frac{\Gamma\left(\frac{n+1}{2}\right)}{\Gamma\left(\frac{m}{2}\right)\left(\frac{n}{2}\right)^{\frac{n-m+1}{2}}} &> e^{-\frac{n-m+1}{2}-\frac{1}{6m}}\sqrt{\frac{(n+1)^n}{m^{m-1}n^{n-m+1}}} \\ &= e^{-\frac{n-m+1}{2}-\frac{1}{6m}}\sqrt{\frac{n^{n+1}(1+1/n)^{n+1}}{(n+1)m^{m-1}n^{n-m+1}}} \\ &> e^{-\frac{n-m+1}{2}-\frac{1}{6m}}\sqrt{\frac{ne}{n+1}}.\end{aligned}$$

Since $2 \leq m \leq n$, therefore, we have

$$\frac{\Gamma\left(\frac{n+1}{2}\right)}{\Gamma\left(\frac{m}{2}\right)\left(\frac{n}{2}\right)^{\frac{n-m+1}{2}}} > \sqrt{\frac{2e^{\frac{5}{6}}}{3}}e^{-\frac{n-m+1}{2}}.$$

Therefore, we have

$$P\left(\lambda_m \leq \frac{B^2 n}{x^2}\right) > \sqrt{\frac{2e^{\frac{5}{6}}}{3}}e^{-\frac{B^2 mn}{2x^2}}\frac{1}{\Gamma(n-m+2)}\left(\frac{e^{-\frac{1}{2}}Bn}{x}\right)^{n-m+1}.$$

□

Similar to real random matrices, we have the following Lemma 23 for complex random matrices. Lemma 23 can be proved using the same techniques as Lemma 22, so we will omit the proof and only give the result.

Lemma 23 *For any $B > 0$, $x > 0$, and $2 \leq m \leq n$, the smallest eigenvalue $\widetilde{\lambda}_{min}$ of $\widetilde{W}_{m,n}$ satisfies*

$$P\left(\widetilde{\lambda}_{min} \leq \frac{B^2 n}{x^2}\right) > e^{1-\frac{1}{12m}}e^{-\frac{B^2 mn}{2x^2}}\frac{1}{\Gamma(n-m+2)^2}\left(\frac{e^{-1}B^2 n^2}{2x^2}\right)^{n-m+1}.$$

The proof of the following Lemma 24 is based on the upper bound of the joint probability density function of λ_{max} and λ_{min} in Lemma 12 and the upper bound of the incomplete Gamma function in Lemma 10.

Lemma 24 *For any $B \leq e^{-1.7}$, $x > 0$, and $2 \leq m \leq n$, the largest eigenvalue λ_{max} and the smallest eigenvalue λ_{min} of $W_{m,n}$ satisfy*

$$P\left(\lambda_{min} \leq \frac{B^2 n}{x^2}, \frac{\lambda_{max}}{\lambda_{min}} \leq x^2\right) \quad < \quad \frac{11 B^{m-1}}{4\sqrt{4\pi}} \frac{1}{\Gamma(n-m+2)} \left(\frac{e^{-\frac{1}{2}} Bn}{x}\right)^{n-m+1}.$$

Proof. From the upper bound for the joint probability density function of λ_{max} and λ_{min} in Lemma 12, we have

$$\begin{aligned} P\left(\lambda_{min} \leq \frac{B^2 n}{x^2}, \frac{\lambda_{max}}{\lambda_{min}} \leq x^2\right) &= \int_0^{\frac{B^2 n}{x^2}} \int_0^{tx^2} f_{\lambda_{max},\lambda_{min}}(s,t) ds dt \\ &< C_{m,n} \int_0^{\frac{B^2 n}{x^2}} \int_0^{tx^2} e^{-\frac{1}{2}t} t^{\frac{1}{2}(n-m-1)} e^{-\frac{1}{2}s} s^{\frac{1}{2}(n+m-3)} ds dt. \end{aligned}$$

Taking the transform $u = tx^2$, we have

$$\begin{aligned} P\left(\lambda_{min} \leq \frac{B^2 n}{x^2}, \frac{\lambda_{max}}{\lambda_{min}} \leq x^2\right) &= C_{m,n} \left(\frac{1}{x}\right)^{n-m+1} \int_0^{B^2 n} e^{-\frac{u}{2x^2}} u^{\frac{1}{2}(n-m-1)} \\ &\qquad \left(\int_0^u e^{-\frac{1}{2}s} s^{\frac{1}{2}(n+m-3)} ds\right) du. \end{aligned}$$

According to Lemma 2.5, if $u \leq n+m-3$, then

$$\int_0^u e^{-\frac{1}{2}s} s^{\frac{1}{2}(n+m-3)} ds \quad \leq \quad e^{-\frac{1}{2}u} u^{\frac{1}{2}(n+m-1)}.$$

Therefore, when $B \leq e^{-1.7}$, we have

$$\begin{aligned} P\left(\lambda_{min} \leq \frac{B^2 n}{x^2}, \frac{\lambda_{max}}{\lambda_{min}} \leq x^2\right) &\leq C_{m,n}\left(\frac{1}{x}\right)^{n-m+1} \int_0^{B^2 n} e^{-\frac{u}{2x^2}-\frac{1}{2}u} u^{n-1} du \\ &\leq C_{m,n}\left(\frac{1}{x}\right)^{n-m+1} \int_0^{B^2 n} e^{-\frac{1}{2}u} u^{n-1} du. \end{aligned}$$

Since $B \leq e^{-1.7}$, so $B^2 n \leq 2(n-1)$. Applying Lemma 2.5 again, we have

$$\begin{aligned} P\left(\lambda_{min} \leq \frac{B^2 n}{x^2}, \frac{\lambda_{max}}{\lambda_{min}} \leq x^2\right) &\leq C_{m,n}\left(\frac{1}{x}\right)^{n-m+1} e^{-\frac{B^2 n}{2}} B^{2n} n^n \\ &= \frac{e^{-\frac{B^2 n}{2}} B^{n+m-1} n^{m-1}}{4\Gamma(m-1)\Gamma(n-m+1)} \left(\frac{Bn}{x}\right)^{n-m+1}. \end{aligned}$$

From (6.24), we have

$$\frac{n^{m-2}}{\Gamma(m-1)} < \frac{e^n}{\sqrt{4\pi}}.$$

Therefore, we have

$$\begin{aligned} P\left(\lambda_{min} \leq \frac{B^2 n}{x^2}, \frac{\lambda_{max}}{\lambda_{min}} \leq x^2\right) &\leq \frac{e^n e^{-\frac{B^2 n}{2}} B^{n+m-1} n}{4\sqrt{4\pi}\Gamma(n-m+1)} \left(\frac{Bn}{x}\right)^{n-m+1} \\ &\leq \frac{B^{m-1} n(n-m+1) e^{\frac{3}{2}n} e^{-\frac{B^2 n}{2}} B^n}{4\sqrt{4\pi}} \\ &\qquad \frac{1}{\Gamma(n-m+2)} \left(\frac{e^{-\frac{1}{2}} Bn}{x}\right)^{n-m+1}. \end{aligned}$$

When $B \leq e^{-1.7}$, for all $n \geq m \geq 2$, we have

$$n(n-m+1)e^{\frac{3}{2}n}e^{-\frac{B^2 n}{2}}B^n \quad < \quad 11.$$

Therefore,when $B \leq e^{-1.7}$, we have

$$P\left(\lambda_{min} \leq \frac{Bn}{x^2}, \frac{\lambda_{max}}{\lambda_{min}} \leq x^2\right) \quad < \quad \frac{11B^{m-1}}{4\sqrt{4\pi}} \frac{1}{\Gamma(n-m+2)} \left(\frac{e^{-\frac{1}{2}}Bn}{x}\right)^{n-m+1}.$$

□

Similar to real random matrices, we have the following Lemma 25 for complex random matrices. Lemma 25 can be proved using the same techniques as Lemma 24, so we will omit the proof and only give the result.

Lemma 25 *For any $B^2 \leq e^{-1.2}$, $x > 0$, and $2 \leq m \leq n$, the largest eigenvalue $\widetilde{\lambda}_{max}$ and the smallest eigenvalue $\widetilde{\lambda}_{min}$ of $\widetilde{W}_{m,n}$ satisfy*

$$P\left(\widetilde{\lambda}_{min} \leq \frac{Bn}{x^2}, \frac{\widetilde{\lambda}_{max}}{\widetilde{\lambda}_{min}} \leq x^2\right) \quad < \quad 0.0352 \frac{1}{\Gamma(n-m+2)^2} \left(\frac{e^{-1}B^2n^2}{2x^2}\right)^{n-m+1}.$$

We are now prepared to derive the lower bounds for the tails of the condition number distributions of random matrices whose elements are independent and identically distributed standard normal random variables

Theorem 26 *For any $x \geq n-m+1$ and $n \geq m \geq 2$, the 2-norm condition number of*

$G_{m\times n}$ satisfies

$$P\left(\frac{\kappa_2(G_{m\times n})}{n/(n-m+1)} > x\right) > \frac{1}{\sqrt{2\pi}}\left(\frac{c}{x}\right)^{n-m+1}, \tag{6.29}$$

where $c \geq 0.245$ is a universal positive constant independent of $x, m,$ and n.

Proof. For any positive constant H, we have

$$\begin{aligned} P(\kappa_2(G_{m\times n}) > x) &= P\left(\frac{\lambda_1}{\lambda_m} > x^2\right) \\ &> P\left(\lambda_m \leq \frac{H^2 n}{x^2}, \frac{\lambda_1}{\lambda_m} > x^2\right) \\ &= P\left(\lambda_m \leq \frac{H^2 n}{x^2}\right) - P\left(\lambda_m \leq \frac{H^2 n}{x^2}, \frac{\lambda_1}{\lambda_m} \leq x^2\right). \end{aligned}$$

Let $H = e^{-1.7}$, then based on Lemma 22 and Lemma 24, we have

$$P(\kappa > x) > \left(\sqrt{\frac{2e^{\frac{5}{6}}}{3}}e^{-\frac{H^2 mn}{2x^2}} - \frac{11H^{m-1}}{4\sqrt{4\pi}}\right)\frac{1}{\Gamma(n-m+2)}\left(\frac{e^{-\frac{1}{2}}Hn}{x}\right)^{n-m+1}.$$

From Lemma 11, we have

$$\Gamma(n-m+2) < \sqrt{2\pi(n-m+1)}(n-m+1)^{n-m+1}e^{-(n-m+1)+\frac{1}{12(n-m+1)}}.$$

Note that, for $2 \leq m \leq n$, we have

$$\sqrt{n-m+1} < 1.21^{n-m+1}, \text{ and } \frac{1}{12(n-m+1)} \leq \frac{1}{12},$$

Therefore, we have

$$P\left(\kappa_2(G_{m,n}) > x\right) \quad > \quad \left(\sqrt{\frac{2e^{\frac{5}{6}}}{3}}e^{-\frac{H^2mn}{2x^2}} - \frac{11H^{m-1}}{4\sqrt{4\pi}}\right)\frac{e^{-\frac{1}{12}}}{\sqrt{2\pi}}\left(\frac{\frac{e}{1.21(n-m+1)}e^{-\frac{1}{2}}Hn}{x}\right)^{n-m+1}.$$

Since $H = e^{-1.7}$, $x \geq 1$, and $2 \leq m \leq n$, so we have

$$\left(\sqrt{\frac{2e^{\frac{5}{6}}}{3}}e^{-\frac{H^2mn}{2x^2}} - \frac{11H^{m-1}}{4\sqrt{4\pi}}\right)e^{-\frac{1}{12}} \quad > \quad 0.99.$$

Therefore, we have

$$\begin{aligned} P\left(\kappa_2(G_{m,n}) > x\right) \quad &> \quad \frac{0.99}{\sqrt{2\pi}}\left(\frac{0.248\frac{n}{n-m+1}}{x}\right)^{n-m+1} \\ &> \quad \frac{1}{\sqrt{2\pi}}\left(\frac{0.245\frac{n}{n-m+1}}{x}\right)^{n-m+1}. \end{aligned}$$

Therefore

$$P\left(\frac{\kappa_2(G_{m,n})}{n/(n-m+1)} > x\right) \quad > \quad \frac{1}{\sqrt{2\pi}}\left(\frac{0.245}{x}\right)^{n-m+1}.$$

Let $c = 0.245$, then we get (6.29). □

Remark:

1. The lower bound in Theorem 26 is for arbitrary $n \geq m \geq 2$ and $x \geq n - m + 1$. For some special case of m and n, more precise lower bound can be obtained. For example, for the special case of real random $m \times m$ matrices, where $m \geq 3$, it has been

proved in [3] that

$$P\left(\kappa_2(G_{m\times m}) > m\,.\,x\right) > \frac{c}{x},$$

where $c \geq 0.13$ is a universal positive constant independent of x and m.

In Theorem 26, however, if we take $m = n$, then we can only get

$$P\left(\kappa_2(G_{m\times m}) > m\,.\,x\right) > \frac{0.097}{x},$$

2. For the special case of real random $2\times n$ matrices, based on the exact probability density function of $\kappa_2(G_{2\times n})$ in [15], we can get

$$P\left(\kappa_2(G_{2\times n}) > x\right) = \left(\frac{2x}{x^2+1}\right)^{n-1} \sim \left(\frac{2}{x}\right)^{n-1}$$

as $x \to \infty$. Hence, the constant c in Theorem 26 is no larger than 2. Therefore, the constant c in Theorem 26 actually satisfies

$$0.245 \leq c \leq 2. \tag{6.30}$$

Similar to real random matrices, we have the following Theorem 27 for complex random matrices. Theorem 27 can be proved using the same techniques as Theorem 26, so we will omit the proof and only give the result.

Theorem 27 *For any $x \geq n-m+1$ and $n \geq m \geq 2$, the 2-norm condition number of $G_{m\times n}$ satisfies*

$$P\left(\frac{\kappa_2(\widetilde{G}_{m\times n})}{n/(n-m+1)} > x\right) > \frac{1}{2\pi}\left(\frac{c}{x}\right)^{2(n-m+1)},$$

where $c \geq 0.319$ is a universal positive constant independent of x, m, and n.

6.5 The Upper Bounds for the Expected Logarithms

For square Gaussian random matrix $G_{n\times n}$, in [58], Smale asked for $E(\log \kappa_2(G_{n\times n}))$. Similarly, for rectangular Gaussian random matrix $G_{m\times n}$, it is also interesting to investigate $E(\log \kappa_2(G_{m\times n}))$. In this section, we will derive upper bounds for $E(\log \kappa_2(G_{m\times n}))$ and $E(\log \widetilde{\kappa}_2(G_{m\times n}))$. Our main results are Theorem 28 and Theorem 29.

Theorem 28 *For any $n \geq m \geq 2$, the 2-norm condition number of $G_{m\times n}$ satisfies*

$$E(\log \kappa_2(G_{m\times n})) < \log \frac{n}{n-m+1} + 2.258. \tag{6.31}$$

Proof. Let $f_\kappa(x)$ be the probability density function of $\kappa_2(G_{m\times n})$, then

$$\begin{aligned} E\log\left(\frac{\kappa_2(G_{m\times n})}{6.414\frac{n}{n-m+1}}\right) &= \int_1^\infty \log\left(\frac{x}{6.414\frac{n}{n-m+1}}\right) f_\kappa(x)dx \\ &< \int_{6.414\frac{n}{n-m+1}}^\infty \log\left(\frac{x}{6.414\frac{n}{n-m+1}}\right) f_\kappa(x)dx \\ &= \int_{6.414\frac{n}{n-m+1}}^\infty P(\kappa_2(G_{m\times n}) > x) \quad \frac{1}{x} \quad dx. \end{aligned}$$

From Theorem 20, we have

$$P\left(\kappa_2(G_{m\times n}) > x\right) < \frac{1}{\sqrt{2\pi}}\left(\frac{6.414\frac{n}{n-m+1}}{x}\right)^{n-m+1}.$$

Therefore, we have

$$\begin{aligned} E\log\left(\frac{\kappa_2(G_{m\times n})}{6.414\frac{n}{n-m+1}}\right) &< \frac{1}{\sqrt{2\pi}}\int_{6.414\frac{n}{n-m+1}}^{\infty}\left(\frac{6.414\frac{n}{n-m+1}}{x}\right)^{n-m+1}\frac{1}{x}\,dx \\ &= \frac{1}{(n-m+1)\sqrt{2\pi}} \\ &< 0.399. \end{aligned}$$

Therefore, we have

$$\begin{aligned} E\log(\kappa_2(G_{m\times n})) &< \log\frac{n}{n-m+1} + \log 6.414 + 0.399 \\ &< \log\frac{n}{n-m+1} + 2.258. \end{aligned}$$

□

Remark:

1. For the special case of real random $m \times m$ matrices, from the results in [60], we can get

$$E\log(\kappa_2(G_{m\times m})) \le \log m + \frac{3+3\log 2}{2} \approx 2.54. \tag{6.32}$$

In Theorem 28, if we take $m = n$, then we have

$$E \log(\kappa_2(G_{m\times n})) < \log n + 2.258.$$

which is a slightly improved version of (6.32).

2. The upper bound in Theorem 28 is for arbitrary $n \geq m \geq 2$. For some special case of m and n or large m and n, more precise results exist:

For the special case of real random $2 \times n$ matrices, it was shown in [14] that

$$E \log(\kappa_2(G_{2\times n})) = \frac{1}{2}\sqrt{\pi}\frac{\Gamma\left(\frac{n-1}{2}\right)}{\Gamma\left(\frac{n}{2}\right)}.$$

For real random $m \times m$ matrices, it has been proved in [14] that

$$E \log(\kappa_2(G_{m\times m})) = \log m + c + o(1)$$

as $m \to \infty$, where $c \approx 1.537$.

For rectangular matrix $G_{m_n\times n}$, if $\lim_{n\to\infty} m_n/n = y$ and $0 < y < 1$, then it has been proved in [14] that

$$E \log(\kappa_2(G_{m_n\times n})) = \log \frac{1+\sqrt{y}}{1-\sqrt{y}} + o(1)$$

as $n \to \infty$

Similar to real random matrices, we have the following Theorem 29 for complex random matrices. Theorem 29 can be proved using the same techniques as Theorem 28, so we will omit the proof and only give the result.

Theorem 29 *For any $n \geq m \geq 2$, the 2-norm condition number of $G_{m\times n}$ satisfies*

$$E(\log \kappa_2(\widetilde{G}_{m\times n})) < \log \frac{n}{n-m+1} + 2.240.$$

Chapter 7

Conclusions and Future Work

7.1 Conclusions of the Research

In this dissertation, we developed several scalable fault tolerance techniques to tolerate partial process failures in large-scale parallel and distributed computing.

We introduced several new encoding strategies into the existing diskless checkpointing techniques and reduced the overhead to tolerate k failures in p processes from $k(\beta + \gamma)m \ . \ \log p$ to $k(\beta + \gamma)m \ . \ (1 + O(\frac{1}{\sqrt{m}})$, where $\frac{1}{\gamma}$ is the rate to perform summation, $\frac{1}{\beta}$ is the network bandwidth between processors, and m is the size of local checkpoint per processor. The introduced checkpoint schemes are scalable in the sense that the overhead to tolerate k failures in p processes does not increase as the number of processes p increases. We evaluated the performance overhead of our fault tolerance approach by using a preconditioned conjugate gradient equation solver as an example. Experimental results demonstrate that our fault tolerance approach can survive a small

number of simultaneous processor failures with low performance overhead and little numerical impact.

We developed an algorithm-based checkpoint-free fault tolerance approach in which, instead of taking checkpoint periodically, a coded global consistent state of the critical application data is maintained in memory by modifying applications to operate on encoded data. We show the practicality of this technique by applying it to the ScaLAPACK matrix-matrix multiplication kernel which is one of the most important kernels for ScaLAPACK to achieve high performance and scalability.

We designed a class of numerically stable real number erasure codes based on random matrices which can be used for the algorithm-based checkpoint-free fault tolerance technique to tolerate multiple simultaneous process failures. Experimental results demonstrate our codes are numerically much more stable than existing codes in literature.

We established upper and lower bounds for the tails of the condition number distributions of Gaussian random matrices which demonstrate that the coding schemes in our algorithm-based checkpoint-free fault tolerance are numerically highly reliable. Let $G_{m\times n}$ be an $m \times n$ real random matrix whose elements are independent and identically distributed standard normal random variables, and let $\kappa_2(G_{m\times n})$ be the 2-norm condition number of $G_{m\times n}$. We proved that, for any $m \geq 2$, $n \geq 2$ and $x \geq |n-m| + 1$, $\kappa_2(G_{m\times n})$ satisfies $\frac{1}{\sqrt{2\pi}}\,(c/x)^{|n-m|+1} < P\left(\frac{\kappa_2(G_{m\times n})}{n/(|n-m|+1)} > x\right) < \frac{1}{\sqrt{2\pi}}\,(C/x)^{|n-m|+1}$, where $0.245 \leq c \leq 2.000$ and $5.013 \leq C \leq 6.414$ are universal positive constants in-

dependent of m, n and x. Moreover, for any $m \geq 2$ and $n \geq 2$, $E(\log \kappa_2(G_{m\times n})) < \log \frac{n}{|n-m|+1} + 2.258$. A similar pair of results for complex Gaussian random matrices was also established.

7.2 Future Work

One of the drawbacks of our fault tolerance approach is that it requires the application programmers to be involved in the fault tolerance. For the future, we plan to design some techniques and develop some software tools to relieve the fault tolerance burden from the application programmer. We will exploit the possibility of automating both the checkpointing and the recovery. We plan to build these fault tolerance techniques into numerical libraries such ScaLAPACK and PETSc. Another direction to extend our work is to develop an application level checkpointing library to help users to perform the diskless checkpointing.

Diskless checkpointing studied in this dissertation could not survive the failure of all processors. However, in the practice of today's high performance computing, the failure of the whole system is not rare. In the future, we would like to explore using a two level recovery scheme [62] which uses both diskless checkpointing and stable-storage-based checkpointing to tolerate both types of failures.

The real-number and complex-number codes proposed in the research are not perfect. Due to the probability approach we used, the drawback of our codes is that, no matter how small the probability is, there is a probability that a erasure pattern may not be

able to be recovered accurately. An interesting open problem is how to construct the numerically optimal codes over real-number and complex-number fields.

Recently, extensive research has been performed on low density parity check (LDPC) codes. One possible direction to further reduce the fault tolerance overhead and improve the scalability might be to explore the possibility of using some of the recent LDPC codes to replace the Reed-Solomon code used in this dissertation.

Bibliography

Bibliography

[1] N. R. Adiga and et al. An overview of the BlueGene/L supercomputer. In *Proceedings of the Supercomputing Conference (SC'2002), Baltimore MD, USA*, pages 1–22, 2002.

[2] L. Alvisi and K. Marzullo. Message logging: Pessimistic, optimistic, causal, and optimal. *IEEE Trans. Softw. Eng.*, 24(2):149–159, 1998.

[3] J. M. Azais and M.Wschebo. Upper and lower bounds for the tails of the distribution of the condition number of a gaussian matrix. *SIAM J. Matrix Anal. Appl*, 26(2):426–440, 2005.

[4] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. SIAM, Philadelphia, PA, 1994.

[5] A. Beguelin, E. Seligman, and P. Stephan. Application level fault tolerance in heterogeneous networks of workstations. *J. Parallel Distrib. Comput.*, 43(2):147–

155, 1997.

[6] L. S. Blackford, J. Choi, A. Cleary, A. Petitetand R. C. Whaley, J. Demmel, I. Dhillon, K. Stanley, J. Dongarra, S. Hammarling, G. Henry, and D. Walker. ScaLAPACK: a portable linear algebra library for distributed memory computers - design issues and performance. In *Supercomputing '96: Proceedings of the 1996 ACM/IEEE conference on Supercomputing (CDROM)*, 1996.

[7] D. L. Boley, R. P. Brent, G. H. Golub, and F. T. Luk. Algorithmic fault tolerance using the lanczos method. *SIAM Journal on Matrix Analysis and Applications*, 13:312–332, 1992.

[8] G. Bosilca, Z. Chen, J. Langou, and J. Dongarra. Recovery patterns for iterative methods in a parallel unstable environment. Submitted to SIAM J. Scientific Computing, 2004.

[9] G. Bronevetsky, D. Marques, K. Pingali, and P. Stodghill. Automated application-level checkpointing of mpi programs. *SIGPLAN Not.*, 38(10):84–94, 2003.

[10] Z. Chen, J. Dongarra, P. Luszczek, and K. Roche. Self-adapting software for numerical linear algebra and LAPACK for clusters. *Parallel Computing*, 29(11-12):1723–1743, November-December 2003.

[11] 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. *Pro-*

ceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 1, 2005.

[12] T. Chiueh and P. Deng. Evaluation of checkpoint mechanisms for massively parallel machines. In *FTCS*, pages 370–379, 1996.

[13] J.J. Dongarra, H.W. Meuer, and E. Strohmaier. TOP500 Supercomputer Sites, 24th edition. In *Proceedings of the Supercomputing Conference (SC'2004), Pittsburgh PA, USA*. ACM, 2004.

[14] A. Edelman. Eigenvalues and condition numbers of random matrices. *SIAM J. Matrix Anal. Appl.*, 9(4):543–560, 1988.

[15] A. Edelman. *Eigenvalues and Condition Numbers of Random Matrices.* Ph.D. dissertation, Dept. of Math., M.I.T., June 1989.

[16] A. Edelman. On the distribution of a scaled condition number. *Mathematics of Computation*, 58:185–190, 1992.

[17] A. Edelman and B. Sutton. Tails of condition number distributions. *SIAM J. of Matrix Anal. and Applic.*, 27(2):547–560, 2005.

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

[19] G. E. Fagg and J. Dongarra. FT-MPI: Fault tolerant MPI, supporting dynamic applications in a dynamic world. In *PVM/MPI 2000*, pages 346–353, 2000.

[20] G. E. Fagg, E. Gabriel, G. Bosilca, T. Angskun, Z. Chen, J. Pjesivac-Grbovic, K. London, and J. J. Dongarra. Extending the MPI specification for process fault tolerance on high performance computing systems. In *Proceedings of the International Supercomputer Conference, Heidelberg, Germany*, 2004.

[21] G. E. Fagg, E. Gabriel, Z. Chen, , T. Angskun, G. Bosilca, J. Pjesivac-Grbovic, and J. J. Dongarra. Process fault-tolerance: Semantics, design and applications for high performance computing. *Submitted to International Journal of High Performance Computing Applications*, 2004.

[22] P. Ferreira. Stability issues in error control coding in complex field, interpolation, and frame bound. *IEEE Signal Processing Letters*, 7(3):57–59, 2000.

[23] P. Ferreira and J. Vieira. Stable dft codes and frames. *IEEE Signal Processing Letters*, 10(2):50–53, 2003.

[24] I. Foster and C. Kesselman. The globus toolkit. In *The grid: blueprint for a new computing infrastructure*, pages 259–278, 1999.

[25] G. H. Golub and C. F. Van Loan. *Matrix Computations*. The John Hopkins University Press, , 1989.

[26] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. Dongarra, J. M. Squyres V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L.

Graham, and T. S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *PVM/MPI*, pages 97–104, 2004.

[27] A. Geist and C. Engelmann. Development of naturally fault tolerant algortihms for computing on 100,000 processors. *Submited to J. Parallel Distrib. Comput.*, 2002.

[28] A. G. Geist, J. A. Kohl, and P. M. Papadopoulos. CUMULVS: Providing fault-tolerance, visualization, and steering of parallel applications. *International Journal of Supercomputer Applications and High Performance Computing*, 11(3):224–236, 1997.

[29] E. Gelenbe. On the optimum checkpoint interval. *J. ACM*, 26(2):259–270, 1979.

[30] R.L. Graham, D.E. Knuth, and O. Patashnik. *Concrete Mathematics: A Foundation for Computer Science*. Addison-Wesley, Reading, MA, 1994.

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

[32] C.N. Hadjicostis and G.C. Verghese. Coding approaches to fault tolerance in linear dynamic systems. *Submitted to IEEE Transactions on Information Theory*, 2004.

[33] W. Henkel. Multiple error correction with analog codes. *Proceedings of AAECC*, pages 239–249, 1989.

[34] P. D. Hough, M. E. Goldsby, and E. J. Walsh. Algorithm-dependent fault tolerance for distributed computing. *Technical Report SAND2000-8219, Sandia Technical Report*, 2000.

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

[36] I. Foster and C. Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kauffman, San Francisco, 1999.

[37] A.T. James. Distributions of matrix variates and latent roots derived from normal samples. *Ann. Math. Statist*, 35(99):475–501, 1964.

[38] D. B. Johnson. *Distributed system fault tolerance using message logging and checkpointing*. PhD thesis, Rice University, 1990. Chairman-Willy Zwaenepoel.

[39] Y. Kim. *Fault Tolerant Matrix Operations for Parallel and Distributed Systems*. Ph.D. dissertation, University of Tennessee, Knoxville, June 1996.

[40] R. Koo and S. Toueg. Checkpointing and rollback-recovery for distributed systems. *IEEE Trans. Softw. Eng.*, 13(1):23–31, 1987.

[41] F.T. Luk and H.Park. An analysis of algorithm-based fault tolerance techniques. *Journal of Parallel and Distributed Computing*, 5:1434–1438, 1988.

[42] M. Abramowitz and I.A. Stegun, eds. *Handbook of Mathematical Functions*. , Dover, New York, 1970.

[43] F. Marvasti, M. Hasan, M. Echhart, and S. Talebi. An analysis of algorithm-based fault tolerance techniques. *Journal of Parallel and Distributed Computing*, 47(4):1065–1075, 1999.

[44] Message Passing Interface Forum. MPI: A Message Passing Interface Standard. Technical Report ut-cs-94-230, University of Tennessee, Knoxville, Tennessee, USA, 1994.

[45] V. S. S. Nair and J. A. Abraham. Real-number codes for fault-tolerant matrix operations on processor arrays. *IEEE Trans. Comput.*, 39(4):426–435, 1990.

[46] V. S. S. Nair, J. A. Abraham, and P. Banerjee. Efficient techniques for the analysis of algorithm-based fault tolerance (abft) schemes. *IEEE Trans. Comput.*, 45(4):499–503, 1996.

[47] J. S. Plank. *Efficient checkpointing on MIMD architectures.* PhD thesis, Princeton University, Princeton, NJ, USA, 1993.

[48] J. S. Plank. Improving the performance of coordinated checkpointers on networks of workstations using RAID techniques. In *15th Symposium on Reliable Distributed Systems*, pages 76–85, October 1996.

[49] J. S. Plank. A tutorial on Reed-Solomon coding for fault-tolerance in RAID-like systems. *Software – Practice & Experience*, 27(9):995–1012, September 1997.

[50] J. S. Plank, M. Beck, and G. Kingsley. Compiler-assisted memory exclusion for fast checkpointing. *IEEE Technical Committee on Operating Systems and Application Environments*, 7(4):10–14, Winter 1995.

[51] J. S. Plank, Y. Kim, and J. Dongarra. Fault-tolerant matrix operations for networks of workstations using diskless checkpointing. *J. Parallel Distrib. Comput.*, 43(2):125–138, 1997.

[52] J. S. Plank and K. Li. Faster checkpointing with n+1 parity. In *FTCS*, pages 288–297, 1994.

[53] J. S. Plank, K. Li, and M. A. Puening. Diskless checkpointing. *IEEE Trans. Parallel Distrib. Syst.*, 9(10):972–986, 1998.

[54] J. S. Plank and M. G. Thomason. Processor allocation and checkpoint interval selection in cluster computing systems. *J. Parallel Distrib. Comput.*, 61(11):1570–1590, November 2001.

[55] P. Sanders and J. F. Sibeyn. A bandwidth latency tradeoff for broadcast and reduction. *Inf. Process. Lett.*, 86(1):33–38, 2003.

[56] M. Schulz, G. Bronevetsky, R. Fernandes, D. Marques, K. Pingali, and P. Stodghill. Implementation and evaluation of a scalable application-level checkpoint-recovery scheme for mpi programs. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 38, Washington, DC, USA, 2004. IEEE Computer Society.

[57] M. Silva and G. Silva. An experimental study about diskless checkpointing. In *EUROMICRO'98*, pages 395–402, 1998.

[58] S. Smale. On the efficiency of algorithms of analysis. *Bull. Amer. Math. Soc.*, 13:87–121, 1985.

[59] V. S. Sunderam. PVM: a framework for parallel distributed computing. Concurrency: Pract. Exper., pages 315–339, 1990.

[60] S.J. Szarek. Condition numbers of random matrices. *J. Complexity*, 7(2):131–149, 1991.

[61] S. S. Vadhiyar and J. Dongarra. SRS: A framework for developing malleable and migratable parallel applications for distributed systems. *Parallel Processing Letters*, 13(2):291–312, 2003.

[62] N. H. Vaidya. A case for two-level recovery schemes. *IEEE Trans. Computers*, 47(6):656–666, 1998.

[63] J. W. Young. A first order approximation to the optimal checkpoint interval. *Commun. ACM*, 17(9):530–531, 1974.

Vita

Zizhong Chen was born in Chongqing, P. R. China. He received his high-school education from Tongliang Middle School, Chongqing, P. R. China from 1990 to 1993. He obtained a Bachelor of Science degree in Mathematics from Beijing Normal University, Beijing, P. R. China in 1997.

He moved to the University of Tennessee, Knoxville, to pursue a doctoral degree in August 2000. During his graduate studies, he worked as a Graduate Research Assistant in Innovative Computing Laboratory (ICL) under the guidance of Dr. Jack Dongarra. He was involved in the following federally funded projects related to high performance computing: Fault Tolerant MPI (FT-MPI), VGrADS, LAPACK for Clusters (LFC), ScaLAPACK, and Self-Adapting Numerical Software (SANS). His current research interests include high performance computing, parallel and distributed processing, cluster and grid computing, numerical analysis and scientific computing, and computational science and engineering. Zizhong Chen is expected to receive a Doctor of Philosophy degree in Computer Science in May 2006.