Fault Tolerant Matrix Operations for Parallel and Distributed Systems

A Dissertation

Presented for the

Doctor of Philosophy

Degree

The University of Tennessee, Knoxville

Youngbae Kim

August 1996

Acknowledgments

I dedicate this dissertation to my parents, as an expression of love and gratitude for providing me with the best education. Without their love, sacrifices, care and encouragement, I would not have written this dissertation. And I am very grateful to my wife Jiyeon for her endless love, and devotion. She was an endless source of inspiration and motivation that were crucial to the completion of my studies in the U.S.A. And to my son June Ho and daughter June Young, I am grateful for the love, understanding, and precious moments we have shared during my study.

I would like to express my appreciation to my advisor, Jack Dongarra, for his upholding of high standards in doing quality research in high-performance computing. I am grateful for his comments on this manuscript, and also for his guidance and support during my study at the University of Tennessee, Knoxville (UTK). I also wish to express my gratitude to Jim Plank for many constructive discussions and his comments on this manuscript and my previous papers. He helped me understand the inner details of fault tolerance and checkpointing. I thank Mike Berry for his comments on this manuscript and continuous encouragement and guidance during my study at UTK. I would like to thank Ohannes Karakashian for serving on my committee. I also thank Sven Hammarling for his constructive discussions on the error analysis presented in this dissertation.

I would like to express my appreciation to the people working in Dongarra land at UTK and ORNL. I especially thank Bob Manchek, R. Clint Whaley, and Jaeyoung Choi for their software and support. I also acknowledge with appreciation the support of the ScaLAPACK group, and the excellent administrative support of Tracy Rafferty, Judi Talley, and other staff.

Several people have helped make my life easier during my stay at UTK. I would like to thank Ajay Kalhan for his valuable help and great friendship. I also thank Randy Bond, Chris Jepeway, and their staff for the great system administrative support, and J. Mike Hammond for his quick and precise proofreading of this manuscript.

Special thanks to God for His Love, without which I can't do anything in my life.

Abstract

With the proliferation of parallel and distributed systems, it is an increasingly important problem to render parallel applications fault-tolerant because such applications are more prone to failures with an increasing number of processors.

This dissertation explores fault tolerance in a wide variety of matrix operations for parallel and distributed scientific computing. It proposes a novel computing paradigm to provide fault tolerance for numerical algorithms. This fault-tolerant computing paradigm relies on checkpointing and rollback recovery using processor and memory redundancy. The paradigm is an algorithm-based approach, in which fault tolerance techniques are tailored into each numerical algorithm without redesigning the algorithm and replicating the processes. The paradigm tolerates the changing and failure-prone nature of a computing platform, thereby allowing users to run their parallel codes dynamically and efficiently.

This dissertation describes the fault-tolerant implementations of various classes of high-performance matrix operations in a parallel programming environment. The implementations are currently applicable to networks of workstations. An empirical performance evaluation of the implementations on a network of workstation confirms that the advantages of our paradigm are its low overhead, simplicity, ease of implementation, and feasibility to scientific applications. This evaluation also demonstrates that the paradigm is an effective approach to achieve fast, reliable scientific computations on networks of workstations.

Contents

List of Tables

List of Figures

List of Symbols

Symbols	*Description*
m, n	row/column dimension of an $m \times n$ matrix.
b	block size.
P, Q	row/column dimension of a processor grid.
$N = \frac{n}{b}$	total number of iterations.
K	checkpointing interval (in iterations).
α	network startup time for communicating a message.
β	network per item transfer time of network.
γ	time to perform a floating point operation or a bitwise exclusive-or (`XOR`).
T_A	running time of an algorithm without checkpointing.
T_C	time overhead of checkpointing.
$T = T_A + T_C$	running time of an algorithm with checkpointing.
T_R	time overhead of rollback recovery.
M_C	amount of memory required for checkpointing.
$T_{copy}(m, n)$	time for each processor to copy an $m \times n$ matrix in memory.
$T_{pre}(m, n, P, Q)$	time to premultiply an $m \times n$ matrix by an orthogonal matrix.
$T_{post}(m, n, P, Q)$	time to postmultiply an $m \times n$ matrix by an orthogonal matrix.
$T_{solve}(m, n, P, Q)$	time to solve a triangular system over a $P \times Q$ processor grid.
$T_{mult}(m, n, k, P, Q)$	time to multiply an $m \times k$ matrix by a $k \times n$ matrix over a $P \times Q$ processor grid.
$T_{rank}(m, n, k, P, Q)$	time to perform one rank-k update over a $P \times Q$ processor grid.
$T_{barrier}(P, Q)$	time to synchronize all processors.
$T_{fact}(m, n, P, Q)$	time to factor an $m \times n$ matrix over a $P \times Q$ processor grid.

Chapter 1

Introduction

1.1 Overview

An important issue in computer systems is the ability of a system to continue operation in the presence of hardware failures and software errors. The designers and users of a computer system generally endeavor to render the system fault-tolerant.

In parallel systems the statistical chance that a processor will fail generally increases with the number of processors. With N processors having exponentially distributed lifetimes with a failure rate λ (failures per unit time), the mean-time-to-failure (MTTF) of the system is $\frac{1}{N\lambda}$ [Jal94].

In a real-world parallel system, a failure of one of the processors causes the entire program to fail, thus doubling its execution time in the worst case. If a parallel application is run on a small-scale parallel system with few processors and no hard-time constraint, such a situation may be acceptable because processor failures are unlikely to occur. However, as a parallel system becomes larger, and if the parallel application is time-critical (as in weather prediction) or has a running time of days, weeks, or months, the chance of processor failure is no longer negligible. In this situation, fault tolerance becomes significant.

One objective of current research is to develop techniques and system software to manage these failures in parallel computers comprising thousands of processors. Specifically, such

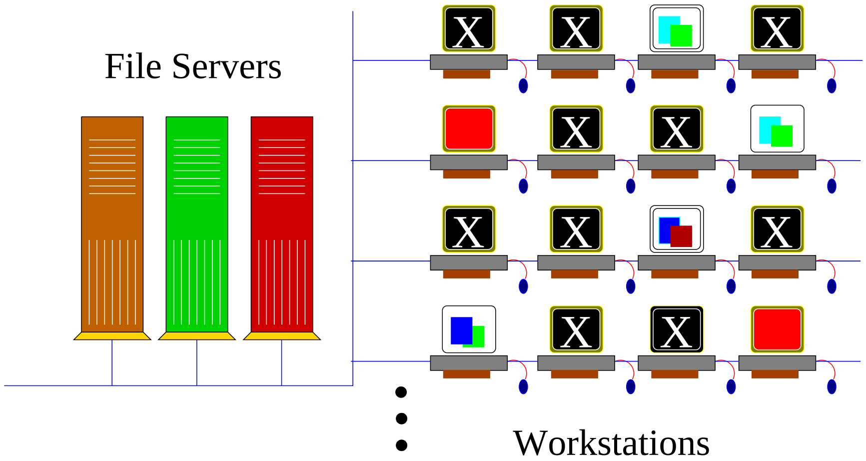


Figure 1.1: A configuration of networks of workstations.

management involves ensuring that the system is available (although perhaps at less than full performance), produces correct results, and gets back to full speed within an acceptable period. Under such conditions, reduced system performance resulting from component failure or from downtime for preventive maintenance is considered acceptable.

This dissertation explores fault tolerance in numerical algorithms for parallel scientific applications. Chapter 1 introduces a fault-tolerant paradigm and describes the platforms for which the paradigm is targeted. This chapter also shows our paradigm's contribution to the field of fault tolerance and examines related work.

1.2 Networks of Workstations

A distributed computing system consists of many autonomous computing elements called *nodes* interacting by message passing over a communication network. For example, a local area network of workstations is a distributed computing system comprising workstations loosely coupled by a local area network. A typical configuration of such a network of workstations is illustrated in Figure 1.1.

Recently, researchers have expressed widespread interest in using networks of workstations for parallel computing. Several factors have spurred this interest. First is the realization that workstations have large memory and significant computing power, which can be combined to tackle large scientific applications [HP94, THM+94, RH95, Sko94]. Second, new programming systems such as Express [Cor88], Linda [CG89], PVM [GBD+94], and MPI [SOHL+96] have made it easier to program in such heterogeneous environments. Third, networks of workstations have become more like massively parallel processing systems (MPPs) such as the IBM SP2 and Cray T3E. In particular, both workstations and MPPs use the same processors and even the same operating systems, and the local area network of workstations is similar to the interconnect of MPPs [AOST93, Bea95].

The advantages of using networks of workstations are clear:

1. They are cost-effective. Little or no additional hardware need be procured—only the software that provides the user interface to the network and OS facilities [Che93].

2. They provide high-bandwidth low-latency communication mechanisms; good scaling characteristics; and standard low-level primitives for communications, synchronization, and scheduling across architectures.

3. They are suitable for coarse-grained applications that require only minimal coordination among processors.

4. They can take advantage of considerable unused processing capacity. Networks of workstations consist of a dynamically varying set of machines that perform long-running applications during idle periods. The underlying idea is that a single application can be parallelized by assigning different parts of the application to different processors on the network. Since these processors are running a time-sharing system such as Unix, available cycles that otherwise would be wasted can be invisibly put to use.

Influenced by these advantages, many universities and laboratories have hooked together their existing workstations via a network with fast switches such as HiPPI, ATM, or FDDI, and are using them for long-lived, computationally intensive scientific applications. This expanded use raises an important issue about fault tolerance in distributed systems.

1.3 Fault Tolerance in Networks of Workstations

Fault tolerance in networks of workstations is in some ways less restrictive, and in other cases more restrictive, than in supercomputers. On the positive side, whereas in supercomputers a failure or maintenance check usually causes the whole system to be shut down, in a distributed network of workstations the rest of its nodes remain fully functional because of their autonomous nature. This feature of continued execution in the presence of partial failures provides great potential for fault tolerance, since tasks running on a failed processor can be moved to live processors to continue execution.

On the negative side, however, the probability of a failure grows as execution time or the number of workstations increases [Jal94]. In addition, while a program is running on a network of workstations, if one of the workstations is heavily loaded or unavailable, the overall execution slows down and hence suffers greatly reduced performance.

Recent studies in [ML91, Mut92, LCP91] examined the characteristics of the usage patterns of workstations and estimated capacity for sharing in a cluster of workstations. In [ML91], the authors showed that on average more than 75% of workstations of one environment were left idle by their users and that the workstations typically were either available for most of the hour, or unavailable for the entire hour. In [Mut92], the author observed that with eight of the workstations in a partition allocated to a user, at least one preemption occurred within one hour. The study also confirmed that workstations are more likely to be preempted during the day while the preemption rate is low for evening and night hours, and that the preemption rate grows as the number of workstations allocated or the duration of the jobs increases. In [LCP91], the authors monitored 68,000 hosts via the Internet and revealed that on average workstations failed once within 21 days.

Thus, networks of workstations change over time due to load, availability, and/or failure. It becomes a major concern for the users of networks of workstations to efficiently run a parallel application by exploiting unused workstation cycles.

Despite the importance of fault tolerance, the system designer often ignores it in most general-purpose computer systems because it is too costly to support in hardware unless the system runs time-critical applications. Software-implemented approaches, while inexpensive, have also been generally ignored—except for methods focusing on wholesale failures. These facts motivate the need for a new look at software-implemented fault tolerance in parallel applications.

1.4 Objective

Toward this purpose, we investigate the integration of fault tolerance in parallel applications rather than by providing transparent fault tolerance at the system level. Specifically, we propose a novel paradigm to provide fault tolerance for numerical algorithms. This new paradigm enables long-lived parallel applications to run dynamically and efficiently by not only tolerating failures of components but also reacting to the changing nature of a distributed system due to load and unavailability.

We target the matrix operations that are most fundamental and widely used in long-running computation-intensive scientific applications—matrix multiplication; the preconditioned conjugate gradient method for solving a large sparse linear system; LU, QR, and Cholesky factorizations for solving a dense linear system; and Hessenberg reduction for solving an eigenvalue problem. These matrix operations have been implemented and are widely available in software packages such as LAPACK (Linear Algebra PACKage) [ABD+95], ScaLAPACK (Scalable LAPACK) [CDO+96], and the PIM (Parallel Iterative Methods) package [dCH95].

1.5 A Fault-Tolerant Computing Paradigm

Our paradigm is designed to incorporate fault tolerance into the algorithms of an application. The underlying idea is to adopt the $N+1$ parity used by Gibson to provide reliability in RAID (Redundant Array of Inexpensive Disks) [Gib90]. The paradigm employs checkpointing and rollback recovery based on processor and memory redundancy with no reliance on disk. This idea, called "diskless checkpointing," was described by Plank and Li [PL94a] as a means for avoiding a major problem in consistent checkpointing. The checkpointing scheme is effective at providing fault tolerance for single-processor failures as long as there is enough memory to hold single checkpoints in memory. To reduce the memory requirements, incremental checkpointing can be used, and compression can be helpful in reducing the load on network bandwidth [FB89, Joh89b, LNP91, EJZ92, PL94a].

Our paradigm is also an algorithm-based approach, implemented using checkpointing and recovery so that fault tolerance is tailored into parallel applications. With algorithm-based checkpointing, instead of checkpointing transparently as in Fail-Safe PVM [LFS93], the programmer is responsible for the placement of checkpoints, what data are checkpointed, how to take checkpoints, and the recovery method for each algorithm. This strategy is beneficial for several reasons. First, the checkpoints can be placed at synchronization points in the program to construct a consistent recovery line. Second, the checkpointed state can be minimized because the checkpointer knows exactly what to save and how to reconstruct a consistent state. This is in contrast to a transparent checkpointer that must save all program states because it knows nothing about the program. Third, with transparent checkpointing, checkpoints are binary memory dumps, which rules out a heterogeneous recovery. With algorithm-based checkpointing, the recovery routines can plan for recovery by a different processor. In short, algorithm-based checkpointing is good when it is well tailored into the algorithm because it enables the checkpointing to be as efficient as possible [LMJ93].

A potential disadvantage of this paradigm is that the programmer must deal with each algorithm specifically with regard to the placement of checkpoints, what data should be checkpoint, and how to take checkpoints. Nevertheless, the extra work is justifiable if the algorithms being checkpointed can be provided for the user as frequently used library calls [SVS94].

1.6 Contributions

The contributions of this dissertation are fourfold:

1. A new paradigm to incorporate fault tolerance into parallel applications.

2. An exploration of the checkpointing and rollback recovery scheme using processor and memory redundancy with no reliance on disk—a significant issue in distributed systems.

3. A new checkpointing and rollback technique based on checksum and reverse computation to provide fault tolerance for matrix operations that require large memory for checkpointing.

4. A multiple checkpointing scheme that uses multiple checkpointing processors to tolerate a certain set of multiple failures and reduce the checkpointing and recovery overhead.

These contributions are intended to enable scientific programmers to run their code on a changing or failure-prone pool of machines.

A performance evaluation based on the analytic models and experimental results of various fault-tolerant implementations confirms that the following advantages of our fault tolerance techniques are realized in practice.

1. User-driven fault tolerance techniques provide low overhead and are feasible and easy to implement in scientific applications.

2. The checksum and reverse computation-based technique is a feasible method to provide fault tolerance for matrix operations that require large amounts of extra memory for checkpointing.

3. The multiple checkpointing technique is more efficient and reliable than single checkpointing techniques. It tolerates multiple failures and provides concurrency for checkpointing and recovery. It checkpoints at a finer-grain interval with relatively less memory.

4. These fault-tolerant implementations can be used in fault-tolerant numerical libraries for scientific computations in distributed systems.

1.7 Related Work

1.7.1 Overview

The main purpose of a fault-tolerant system is to produce correct results in the presence of faulty units. Two fundamentally different approaches have been used to achieve fault tolerance [AL81, SS82, LA94, Jal94]. They are based on redundancy by *replication of data and computation* and on *checkpointing and rollback recovery.*

In the first approach, the system is designed to perform correctly by hiding the effect of a faulty unit. This approach can be applied either in space or in time. The most popular redundancy-based technique for fault tolerance is NMR (N-Modular Redundancy), which employs a majority voting scheme. NMR systems can be designed with or without spare processors. When spare processors are used, upon detection of a faulty processor, the system is reconfigured using a spare one. An NMR system detects permanent as well as transient faults. The disadvantage of the NMR technique is its enormous overhead in hardware, especially for large values of N.

Conventional time redundancy involves computing the same problem more than once, at carefully designed time intervals, until successive results match. Such a system can detect intermittent and transient error conditions. However, errors due to permanent faults will not be detected. Once an error has been detected, the unit must be replaced by a spare unit that is known to be fault-free. While the hardware overhead in a temporally redundant system is minimal, the performance penalty can be very large.

Checkpointing and rollback recovery, also known as *backward error recovery*, is an attractive software approach to providing fault tolerance [BBG83, KT87, EJZ92]. In this approach, every process periodically saves its own state on stable storage as a checkpoint during normal operation. If a failure occurs, the processes will use the latest checkpoints on stable storage to restore a *global consistent state* and restart execution from that state, instead of restarting the computation from scratch.

Checkpointing can be embedded in the operating system and hence provide fault tolerance for parallel applications transparently without any change to applications. Unfortunately, although conceptually simple, checkpointing is quite costly in terms of performance overhead. The performance overhead results mainly from saving the checkpoints on stable storage and communicating between processes to build a global consistent state.

Both redundancy and checkpointing have been implemented in diverse systems, including CIRCUS [Coo85], LOCUS [PWC+81], and Clouds [DJAR90].

Recently, researchers have also expressed increasing interest in encoded techniques such as RAID [CLG+94] and in the process group approach and causal communications popularized by ISIS [Bir93].

In addition to these general approaches, certain approaches for fault tolerance have been designed specifically for some common architectures, for example, tree and hypercube architectures [Pra86]. These approaches are based on hardware redundancy by adding additional processors to the existing system based on a specific network architecture. Most of these special techniques are costly because of the required redundancy.

Of all these various approaches, checkpointing remains the method of choice to transparently provide fault tolerance for parallel applications at the system level. This method has been implemented on parallel and distributed systems in various ways [BBG+89, CJ91, CA94, EJZ92, JZ90, LY87, LNP94, PL94b, SSCC95, SY85, SJF95, WF93, BP93]. To provide fault tolerance services using such checkpointing techniques, researchers have implemented several process migration or replication tools on top of the existing parallel programming environments [CCK+95, PL94d, Ste96]. In addition, efforts such as Fail-Safe PVM [LFS93], FT-Linda [BS89], and Pact [Mai93] have dealt with fault tolerance in existing parallel programming environments.

Detailed discussion of these various efforts is beyond the scope of this dissertation. Instead, in this section, we examine in detail the existing work related to fault-tolerant matrix operations in floating-point arithmetic for parallel and distributed systems. We also review the approaches implemented at the software system level.

1.7.2 An Error-Checking Device in Linear Computations

In 1951, Dwyer [Dwy51] used row and column checksums as a checking device to eliminate computation of "mistakes" and to study the accumulation of errors in matrix computations. The underlying idea of his checksum encoding scheme is that the checksum property is preserved throughout some matrix computations.

For example, in Gaussian elimination, the sum of the coefficients of a row (i.e., the row checksum) always holds throughout the elimination of columns of the matrix. In solving a system with Gaussian elimination, this checking method is continued through the forward solution and can be carried through the back solution. Also, the basic final check can be used to verify that the computed solutions do actually satisfy all the original equations. The use of this final verification has been advised as a general computational procedure called "backward error analysis" [Wil65].

1.7.3 Algorithm-Based Fault Tolerance

Algorithm-based fault tolerance (ABFT) is a new approach to fault tolerance, which was first proposed by Huang and Abraham [HA84] for detecting transient errors on systolic arrays. In this approach, fault tolerance is achieved by tailoring the fault tolerance scheme to the algorithm that is to be performed on the parallel machine.

The fundamental idea of ABFT techniques is characterized by three steps: (1) encoding data at a higher level using checksum schemes; (2) redesigning algorithms to operate on the encoded data; and (3) distributing the computation steps of the redesigned algorithm among computational processors such that failure of any processor affects as little data as possible.

One of the most important issues of ABFT is the efficient encoding of data in terms of error detection and correction. A good scheme of encoding data should be able to detect all errors and locate and correct as many errors as possible—including multiple, especially grouped, errors.

A number of previous ABFT methods introduce various checksum encoding schemes for error detection and correction and fault location. These techniques are based on weighted checksum encoding schemes in [HA84, JA84, AL88, LP88b, Par92], real number codes in [NA90], Z_{2^m} code in [FRK94], general linear codes in [NA88], and error-coded operands in [Lo94]. Unfortunately, the introduction of checksum encoding schemes in algorithm-based fault detection has a disadvantage: it often leads to false alarms as a result of round-off errors and floating-point overflow that occur during data encoding or major computations. Recently, a simple example in [BGM$^+$95] dramatically pointed out how such classical checksum-based ABFT techniques can completely fail to detect even catastrophic errors as well as certain floating-point errors.

ABFT techniques have been extended to parallel algorithms executing on distributed memory MIMD (Multiple Instruction Multiple Data) machines [BRS$^+$90, PB92, Bal91, BRS$^+$88]. Some work [BRS$^+$88, NHA92, Roy92] has also been done on evaluating the effectiveness of algorithm-based techniques by computing the coverage for floating-point faults in the algorithm.

In the following subsections, important problems and issues regarding ABFT techniques are discussed. Particular attention is given to their effectiveness and applicability for tolerating faults in floating-point arithmetic on parallel and distributed computing systems.

Problems and Issues with ABFT Techniques

The platforms for which ABFT methods are targeted are SIMD (Single Instruction Multiple Data) machines such as systolic arrays or some regular processor arrays. In such machines, two kinds of error arise: transient and permanent. It is assumed that single or multiple faulty processors can cause both transient and permanent errors.

Various checksum codes have been suggested for fault-tolerant matrix computations on processor arrays. Use of these codes has been limited, however, because of potential roundoff, overflow, and underflow errors. Although ABFT techniques can provide fault tolerance at a low cost in terms of running time and hardware overhead, in general they can significantly complicate algorithm development. Fortunately, a wide variety of scientific and engineering applications depend on just a handful of related matrix operations. As a result, ABFT methods have been designed for several matrix-based computations that are used frequently in signal processing applications, such as matrix multiplication, matrix inversion, LU decomposition, QR decomposition, and FIR (Finite Impulse Response) filtering [HA84, JA86, LP88a, RB90, VJ90]. In addition, an algorithm-based approach makes it possible to develop fault tolerant implementations of the fast Fourier transform [WJ94, CM88a, THC90], sorting [CM88b], and median filtering [VJ90]. Research also has resulted in methods for the analysis and design of ABFT schemes [BA86, VJ93, AL88, NA88, RR93, BRS+88, NA90, NHA92, GRR90]. Additionally, some work has been done in developing new encoding schemes for ABFT, such as generalized checksum codes [NA88], and the sum-of-square technique for the detection and correction of errors in matrix manipulations.

The work in [HA84, JA86] provides a strong theoretical background for the field of ABF detection and tolerance; however, many issues remain to be addressed. In particular, although such encoding schemes provide a low-cost method for achieving fault tolerance, any ABFT method must deal effectively with the numerical limitations of finite-precision arithmetic. Hence, the method must allow for a certain amount of error due to numerical inaccuracy. One approach has been to assume only a fixed-point number system, but this does not provide general ABFT methods for scientific computation.

Extending the weighted checksum technique to floating-point arithmetic introduces a number of complications. First, the algorithms must tolerate a certain error between the correct and computed checksums. Such error tolerance constitutes the largest roundoff error that the scheme can encounter without signaling a fault. If the tolerance is set too low, typical numerical errors could be misinterpreted as faults (false alarms). Setting the tolerance too high, on the other hand, can significantly reduce fault coverage. Determination of the appropriate tolerance level depends heavily on the nature of the data and the computation, as well as on the size of the matrix involved. Hence, it is difficult to scale the work in [JA86] to massively parallel systems that use floating-point arithmetic.

Furthermore, an ABFT scheme must address the possibility of overflow. In particular, the weighted checksums may be extremely large if the number of elements whose checksum is required is large. Also, for a large matrix, the size of the larger weights can obliterate the effect of the smaller weights, thus making the detection and correction of many errors virtually impossible. Luk [AL88] suggests the weight $w_j = j$ instead of $w_j = 2^{j-1}$ for the jth element to avoid the problem. However, this new weight can still become prohibitively large. Luk's scheme also introduces complications: it requires more time to multiply and can introduce numerical error. In contrast, the exponentially weighted checksums are easily computed in a floating-point system without loss of precision because multiplication by a power of 2 can be performed simply by changing exponents. Moreover, the smaller spacing of Luk's weights makes fault location more difficult. As a result of the accumulation of roundoff error and error in checksum division (the checksum division becomes j if a_j is a faulty element), an error of ± 0.5 can lead to erroneous fault location.

An alternative approach, called *general linear codes*, is introduced in the paper [NA88]. Although these codes improve the numerical properties of ABFT schemes, the techniques are still extremely sensitive to roundoff errors, particularly in computations involving large matrices. Fault-tolerant algorithms must not only handle error detection, location, and correction, but must also tolerate certain inaccuracies due to roundoff errors in floating-point arithmetic. In [RJ94], the authors consider possible ways to scale linear code for algorithm-based error detection, correction, and fault location techniques to floating-point matrix operations in massively parallel systems. They suggest incorporating general linear codes into a partitioning scheme, thereby enhancing the numerical properties of ABFT encoding methods.

Multiple-Error Detection

In [Par92], the author introduces a *unified checksum method* for multiple transient error detection in three different matrix triangularizations: LU decomposition, Gaussian elimination with pairwise pivoting, and QR decomposition. This method was motivated by Luk and Park's unified scheme for single transient error [LP88b]. Also developed was a linear algebraic approach for multiple-error detection in matrix triangularizations by using checksum schemes.

In [LP88b], a technique is described for computing the correct decomposition from an erroneous one by factorization updating schemes. The technique was used to detect all the transient errors that occur in a maximum of t different columns by introducing t checksum vectors. Also presented is a floating-point error analysis, to determine the effects of the roundoff errors in using the checksum method for multiple-error detection and correction. Unfortunately, the problem of locating and correcting the errors seems to be intractable with this scheme. Even if the block checksum scheme allows one to locate and correct multiple errors, it reduces the portion of the matrix in which multiple errors are allowed. Moreover, even a single transient error can make the corrected results useless because of rounding errors. Thus, research is needed to solve the difficult problem of multiple-error location and correction for matrix triangularizations.

ABFT for MIMD Systems

Several papers deal with ABFT schemes for parallel and distributed computing systems. Of particular interest is the work in [RB94], in which the authors demonstrate the practicality of their ABFT schemes, based on weighted checksum codes, for fault location and correction. They present results with three popular numerical algorithms on an Intel iPSC/2 hypercube multicomputer. Also presented is a scheme that achieves forward error recovery (recovery without having to restart the algorithm from the beginning) for the case when t processors might be faulty, where t is a design parameter. Furthermore, the authors propose approaches for locating and correcting errors due to multiple faulty processors, by a suitable encoding of parallel algorithms for general-purpose distributed-memory multicomputers. They have implemented a version of a matrix multiplication algorithm that works on data encoded initially in such a manner that, even in the event of multiple-processor failures, the resulting matrix may be recovered from the encoded data on the fault-free processors. Their approach is based on a generalization of the weighted checksum approach, discussed in [JA84]. Finally, the authors present error coverage and timing overhead results on three algorithms: matrix multiplication, QR factorization, and Gaussian elimination.

Their experiments show that timing overheads are small, especially for moderate-sized problems. Nevertheless, the authors focus mainly on error detection in the cases of single- or multiple-processor failures. Although the presence of errors is always detected, there are cases in which the *locations* of the errors are not detected.

Remarks on ABFT

When using ABFT techniques with row and column checksums added to matrices, the following situation arises. If one computes an element of a matrix incorrectly when performing matrix multiplication, LU factorization, addition, or transposition, or if faulty processors have corrupted the computation and produced incorrect results due to transient, intermittent, or permanent faults, *all processors* detect, locate, and correct the errors subject to the faulty processors. In floating-point matrix computations—where no computation is exact—the difficulty of error detection and correction is exacerbated by the presence of roundoff errors. Hence, it is necessary to distinguish errors caused by hardware faults from errors that could be caused just by normal roundoff errors.

Most previous approaches for error detection and correction based on the use of checksums have dealt with such issues. However, none of these approaches locate and correct all the faults completely.

Another difference between our computing model and ABFT computing models deserves mention. ABFT techniques are quite appropriate for systolic processor array or regular SIMD machines. In systolic processor arrays, the input data are fed into the processor arrays, and each processor is responsible for computing a subset of the output elements. If a processor fails, all that is lost is the output element. Also, it is assumed that communication still works and that the input is still available. However, neither of these assumptions is valid on the platforms targeted in this dissertation. Our computing model assumes that any processor that fails will be unavailable until it is fixed or not available at all. Thus, a processor failure means that the system has lost both a processor and its data.

1.7.4 Backward Error Assertion

Unlike the previously discussed checksum-based ABFT methods, Boley et al. [BGM+95] introduce a new approach for detecting and partially correcting faults when using floating-point arithmetic. This approach is not based on any coding scheme, but is an assertion scheme based on *backward error analysis* [Wil65] for error detection in algorithms that solve dense systems of linear equations. This *backward error assertion* model is specifically designed to operate in an environment of floating-point arithmetic subject to roundoff errors. The authors have implemented the model in a Watchdog environment that can guarantee that the solution is correct (to the extent that any floating-point computation can be guaranteed correct) and either correct many transient errors or signal that an uncorrectable transient fault occurred during the solution process. Since the model uses a technique based on iterative refinement [Wil65] designed to minimize roundoff errors, it can automatically and silently correct many small errors, whether from roundoff or from hardware faults.

The basic idea of this approach is to check that the computed solution satisfies an *a priori* error bound for the particular method used. Their experiments with implementations of three different numerical algorithms for solving systems of linear equations show its effectiveness and validity for fault tolerance in terms of fault coverage and percentage of errors detected, signaled, and corrected. Finally, they propose some implementations in a parallel environment with a Watchdog process and claim that this approach does not preclude the use of other checksum-based scheme.

1.7.5 Other Related Work

The following two approaches are very different from the previous work described as well as from the approach used in the dissertation. The approaches do not rely on any checkpointing and rollback recovery, but instead rely on redundancy or replication of process. These methods must be supported at the system or compiler level.

Recursive Algorithm for Fault Tolerance

The first approach to fault tolerance in parallel and distributed systems is to employ redundancy *dynamically* [Agr88]. For example, the recursive algorithm for fault tolerance (RAFT) [Agr88] is an algorithm for achieving fault tolerance (both diagnosis and recovery) without dedicated redundant processors or checkpointing. Fault tolerance is achieved by duplicating the computation on different processors and comparing the signatures derived from their results. Comparison-based diagnosis techniques have also been used by others; see, for example [Agr88].

A processor is assumed to produce the result of the computation and a signature. The signature is a good representative of the processor outcome and can be obtained by using data compression techniques. The approach works as follows. Assume that the processors are numbered. Initially, an incoming task is scheduled to a pair of processors, say P_1, P_2. These produce results R_1 and R_2 and signatures S_1 and S_2. The two signatures are compared. If they match, one of the results is returned. If the signatures do not match, the job is scheduled on another processor, say P_3. The signature S_3 is compared with the earlier signatures S_1 and S_2. If a pair of matching signatures is found, the result R_3 is returned. Otherwise, the job is scheduled on another processor, until a matching pair of signatures is found.

Dependable Execution of Parallel Programs

In [GC95], the authors propose an approach that ensures that application software tolerates processor failures on a MIMD system. The approach differs from checkpointing and rollback recovery techniques, and also from distributed algorithms techniques (which are essentially system-oriented), and does not rely on system-level constructs for failure recovery.

Parallel programs are usually composed of interdependent tasks, each of which is a unit of sequential processing accomplished on a single processor. The underlying idea of the approach is to detect failures in a parallel program by using a detection algorithm based on task dependencies of the program and then to restart the failed tasks by rescheduling the tasks using a reallocation algorithm aimed at redistributing task responsibilities from the unreliable processors to their reliable counterparts. The approach works well as long as the program can be decomposed into a set of communicating tasks. The proposed algorithms can be embedded by the compiler into any parallel program to transform it into a resilient version able to run on a multiprocessor platform. Nevertheless, although simulation results confirm the successful completion of parallel computations with modest detection and restart costs, the costs have been shown to increase drastically as the failure rate increases.

1.8 Dissertation Organization

The rest of this dissertation is organized as follows. Chapter 2 introduces the novel fault-tolerant computing paradigm to provide fault tolerance for numerical algorithms. It also presents the target system model and failure assumptions that the paradigm must satisfy and presents the fault tolerance techniques developed using checkpointing and rollback recovery based on processor and memory redundancy. Chapter 3 describes how the different fault tolerance techniques can be integrated efficiently into the algorithms of various matrix operations and gives analytic models of the fault-tolerant matrix operations. Chapter 4 describes the implementations of fault-tolerant matrix operations and presents the experimental results and performance evaluation of these implementations. Chapter 5 draws conclusions and discusses further work for research.

Chapter 2

A Fault-Tolerant Computing Paradigm for Matrix Operations

This chapter discusses a paradigm to provide fault tolerance for a wide variety of matrix operations. In Sections 2.1 through 2.3, we describe the system model, failure model, and failure detection that our paradigm must satisfy. Section 2.4 presents the basic checkpointing and recovery scheme employed by the paradigm. Section 2.5 introduces two different techniques to incorporate fault tolerance into various matrix operations. Section 2.6 presents the metrics to characterize performance of our fault tolerance techniques.

2.1 System Model

A distributed system consists of many computing elements, frequently referred to as *nodes*, that are geographically separated but connected by a communication network. Each node is autonomous and communicates with other nodes by message passing through the communication network. In such distributed systems, nodes are treated as a complete entity, and typically a node failure cannot make another node fail or behave incorrectly. Here, we focus on a distributed system as the computing platform targeted by the fault-tolerant computing paradigm. Specifically, in our system model, a system consists of the nodes connected by a network. Each node is a processor that has enough private memory to store its state and perform computations based on the current state. The nodes communicate by message passing through a point-to-point or broadcast communication network. The system offers space sharing such that several users can use the system simultaneously.

Suppose that we develop a scientific application that runs best on N processors in a distributed system. We assume that either a partition of a system or an entire system is dedicated to the application. The partition running the application consists of a set of $N + m$ autonomous processors $P = \{P_1, ..., P_{N+m}\}$ divided into a subset of N processors, *application processors*, $P_A = \{P_1, ..., P_N\}$, and a subset of m processors, *spare processors*, $P_C = \{P_{N+1}, ..., P_{N+m}\}$, where $m > 0$. The partition continues its own execution independently of other partitions in the system. Only application processors participate in the computation of the application; spare processors are available for the lifetime of the application. If any processors that are running the computation must leave the partition immediately because of failure, unavailability, or heavy load, they can be replaced quickly by spare processors available in the partition. Failed processors may rejoin the current partition when they become functional. Whenever processors of other partitions become idle, they are able to join the partition. However, the number of application processors running the application remains unchanged for the lifetime of the application.

2.2 Failure Model

We assume that failures respect the *fail-stop* model, as defined in [SS83]. Under this model, (1) components of a system fail by halting without initiating erroneous actions or sending incorrect messages, and (2) these failures are detectable. Of course, in a real-world system, asynchronous communication and unpredictable delays limit the validity of these assumptions: failure detection by timeout can never be completely accurate. In addition to such permanent failures, there are transient failures such as an unresponsive swapping device or a temporary communication outage. The model does not explicitly guarantee that such transient failures can be detected. If, however, such transient failures can be handled like halting failures, and if they are also detectable, they are treated as failures in our model.

2.3 Failure Detection

It is assumed that fail-stop failures can be detected with the aid of the programming environment or using an error-encoding scheme [MS77] and that all members of the partition are notified of such failures. The detection of transient failures may also be supported by the system programming environment. For example, PVM [GBD+94] provides a facility to notify the virtual machine of such processor failures, additions, and deletions, and also incorporates a timeout technique in its message passing. In PVM, when a failure occurs, a specific message is received by all the processors. Failure detection can be performed by checking this message when an application needs communications among processors.

However, if failure detection is not supported by the system, it should be addressed by using an error-detection technique. Many well-known techniques exist, such as parity checks and address range checks [AL81, Joh89a, Jal94]. In our system model, we assume that the system provides an adequate failure detection mechanism.

2.4 Checkpointing and Recovery

Checkpointing and rollback recovery (or backward error recovery) enable a system with fail-stop failures to tolerate failures by periodically saving the entire state and rolling back to the save state if a failure occurs [SY85]. In our paradigm, the basic checkpointing and rollback recovery scheme adopts the idea of providing reliability in disk arrays [Gib92]. This scheme is based on using the spare processors in the system model and spare memory of the application processors to provide redundancy.

To make our checkpointing scheme as efficient as possible, we implement algorithm-based checkpointing by exploiting the semantics of the algorithms of the target matrix operations.

In the following subsections, we introduce the fundamental idea of RAID and discuss the basic checkpointing scheme based on this idea. We also describe the recovery models to tolerate a single failure and multiple failures and give a general description of algorithm-based checkpointing for matrix operations, i.e., how the basic checkpointing scheme can be tailored into matrix operations.

2.4.1 Redundant Disk Arrays

Redundant Arrays of Inexpensive Disks (RAID) are a cost-effective means of simulating high-capacity disks [PGK88]. The data is distributed over multiple disks by using "bit-interleaving." In bit-interleaving, different bits of a data word are stored on different disks. The objective is to provide high I/O bandwidth. However, an array of disks used in this way is not reliable, since the failure of any one of the disks can cause all of the data to become unavailable. The solution is to have some redundant disks in the array, so that the failure of a few disks does not cause all of the data to become lost.

Various implementations exist for RAID [PGK88]. The different implementations provide different levels of cost benefit, reliability, and performance. In one implementation, a number of disks are partitioned into groups. Each group has some data disks and some *check disks*. The check disks are the redundant disks. The data disks and check disks together store the data by using some coding technique that allows recovery of the original data in the event of failure of a few disks in the group. The number of disks depends on the coding technique used. In general, an $N + m$ RAID implementation consists of N data disks and m check disks and is therefore able to tolerate up to m failures if a proper encoding scheme is applied.

One particular implementation employs one check disk for a group of disks. In Figure 2.1, such an $N+1$ RAID implementation is illustrated for $N = 4$. This implementation consists of $N + 1$ disks with one parity block per N blocks of data, each stored on a different disk. If, say, D_i is a block of data stored in disk i, the parity block D_C is a parity block computed from all of the D_i's. If one disk fails, each data block on the failed disk can be reconstructed from the N other blocks in each set of blocks. Figure 2.1 shows how D_1's data is recovered if D_1 fails.

$$D_C = D_1 \oplus D_2 \oplus D_3 \oplus D_4.$$
$$D_1 = D_C \oplus D_2 \oplus D_3 \oplus D_4.$$

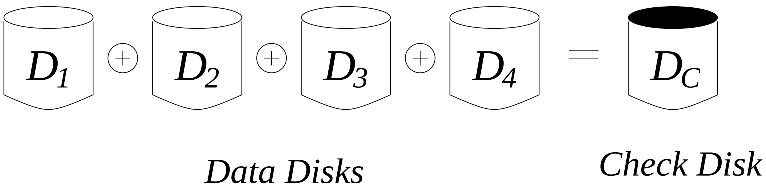


Figure 2.1: An $N + 1$ RAID implementation, where $N = 4$

2.4.2 Basic Checkpointing Scheme

For rollback recovery, a consistent state as a recovery point is constructed by saving periodically, during normal execution, all the information needed to restart the application. Under the system model assumed in Section 2.1, the N application processors and m spare processors (called checkpointing processors) cooperate to take periodic global checkpoints. If certain subsets of processors fail, the system can roll back to the most recent checkpoint, and the remaining processors will continue to run the application.

Specifically, each processor checkpoints its own *local checkpoints* in local physical memory. A parity or checksum of encoding the application processors' checkpoints is also maintained in the checkpointing processors. If any of the application processors fails, it can be replaced by a checkpointing processor or a spare processor, and the lost information of the failed processor can be recovered, if necessary, from the checkpointing processor's global checkpoints and the application processors' local checkpoints. If any of the checkpointing processors fails, it can be replaced, if necessary, by a spare processor. As more processors become available to the system, they can be used as extra checkpointing processors.

In the following two subsections, two recovery models are described—one for tolerating any single processor failure and the other for tolerating multiple processor failures.

2.4.3 Single-Failure Recovery Model

This model, consisting of N application processors and m spare processors, can handle m single failures during the lifetime of the application. The program executes on N processors; there is a single checkpointing processor. Figures 2.2 and 2.3 depict how to construct checkpoints and how to recover in the presence of a single failure. As shown, a spare processor becomes the new checkpointing processor after recovery, if one is available. The model therefore tolerates m single failures.

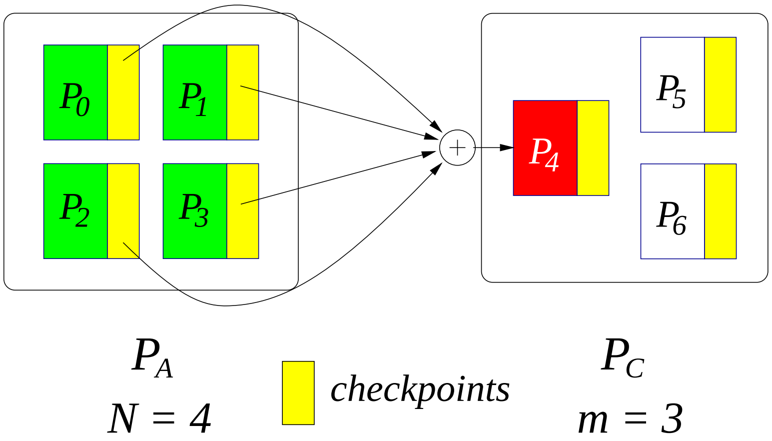


Figure 2.2: Single-failure recovery model: before a failure

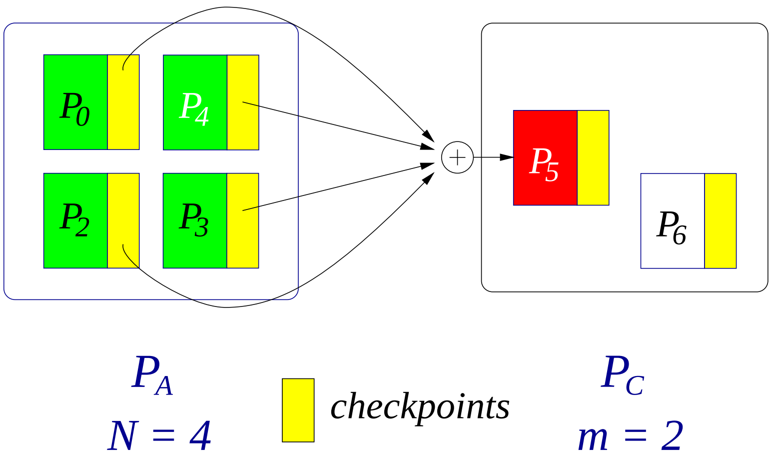


Figure 2.3: Single-failure recovery model: after a failure

2.4.4 Multiple-Failure Recovery Model

A generalization of the single-failure recovery model, the multiple-failure recovery model consists of $N + m$ processors that can tolerate m failures at once. Instead of having one dedicated processor for checkpointing, the entire set of application processors is divided into m groups, and one checkpointing processor is dedicated to each group. When one failure occurs in a group, the checkpointing processor in the group will replace the failed one, and the application will roll back and resume at the last checkpoint. Figure 2.4 shows the application processors logically configured into a two-dimensional mesh, with a checkpointing processor dedicated to each row of processors. This model enables the

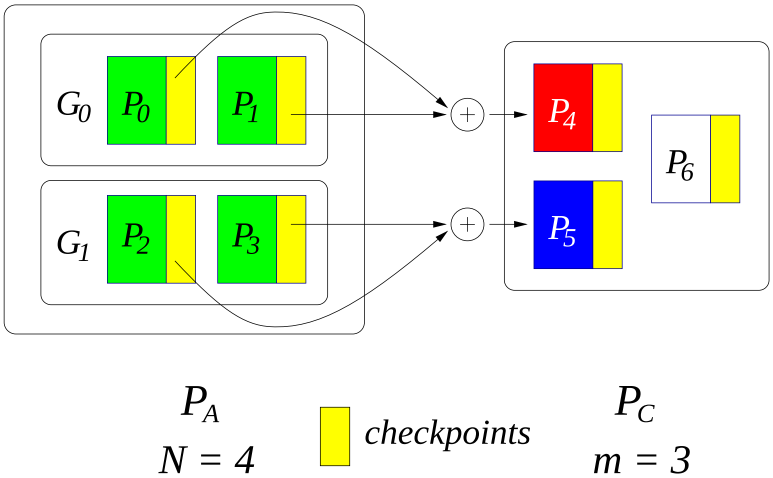


Figure 2.4: A multiple-failure recovery model

algorithm to tolerate a certain set of multiple failures simultaneously, one failure for each group (e.g. each row or column of processors). This is often called the *one-dimensional parity scheme* [GHKP89].

Several schemes have been developed to configure extra checkpointing processors to tolerate multiple processor failures. For example, the paper [GHKP89] presents two-dimensional parity or multidimensional parity, in which the coding information is distributed in two-dimensional or multidimensional fashion, respectively. Another paper [BBBM94] introduces

EVENODD parity, with which two extra processors may be used to tolerate any two failures in the system. More complicated coding schemes have been suggested to tolerate m failures with m checkpointing processors for arbitrary m [MS77, BM93, PFL94, Par95].

2.4.5 Algorithm-Based Checkpointing

For matrix operations like the general factorizations, algorithm-based checkpointing can become a simple task. Such matrix operations handle a static matrix data structure rather than a complicated dynamic data structure. In this section, we discuss the issues in algorithm-based checkpointing: how to take checkpoints, where to place checkpoints, and what data gets checkpointed.

It is a straightforward operation to adopt our basic checkpointing scheme by exploiting the semantics of each matrix algorithm. In matrix operations, since all processors iterate the same number of times, it would be sufficient to place a global consistent checkpoint after every K iterations, where $K > 0$. This is beneficial because the programmer could determine how often checkpoints are taken and hence tune its performance. In addition, these matrix operations iterate by modifying a part of the input matrix (e.g. row or column blocks). Once the entire matrix is checkpointed at the beginning of the algorithm, only the part of the matrix modified between two checkpoints needs to be checkpointed.

2.5 Fault Tolerance Techniques

This section describes two different techniques employed to incorporate fault tolerance into various classes of matrix operations. The techniques differ in the data encoding scheme used for checkpointing and in the rollback mechanism based on the computation aspects of each matrix operation.

2.5.1 Parity-Based Technique

The first fault tolerance technique employs parity (bitwise exclusive-or of floating-point numbers) as the coding information. While the application processors build *local checkpoints* in local memory, the parity of the application processors is maintained in the memory of the checkpointing processors. When processors fail, the lost data can be recovered by computing the parity from the remaining live processors and the checkpointing processors.

The fundamental idea of the parity-based technique is to checkpoint initially the entire data and then checkpoint the part of the data that has been modified since the last checkpoint. Algorithms that benefit most from this technique modify only a moderate amount of data between two checkpoints, thereby allowing the local checkpoint size to be small. Such algorithms include left-looking and top-looking variants of the Cholesky, LU, and QR factorization algorithms [DDSvdV91].

An iterative method for solving a linear system $Ax = b$ is the most appropriate for this technique. An iterative method such as PCG (preconditioned conjugate gradient) takes as input A and b for a linear system, and these are never changed. The same computation steps (e.g., matrix-vector multiplications and inner products of two vectors) are repeated to compute intermediate solutions x in a small amount of workspace, $O(n)$, where n is the size of matrix A. Since only the workspace and the intermediate solution x are modified at each iteration, only these are checkpointed. While each iteration has $O(n^2)$ time complexity, each checkpoint has only $O(n)$. Hence, the total overhead for checkpointing and recovery is small compared with the execution time for the failure-free algorithm without checkpointing. One can further reduce overhead by checkpointing every arbitrary number of iterations rather than every iteration.

Thus, this technique is efficient in that checkpointing to provide fault tolerance for an algorithm uses a very moderate amount of time and memory. However, for algorithms such as matrix multiplication, parity-based fault-tolerant implementations may not be easily developed because of memory constraints. For example, in order to checkpoint an algorithm such as matrix multiplication [AGZ94] using this technique, each processor must have enough memory to save its entire matrix. Moreover, right-looking variants of general factorizations perform faster than other variants, even on most parallel machines, by taking advantage of more parallelism and less memory access when the lower-right corner of the matrix is updated after the current panel of column blocks is factored [DDSvdV91]. The parity-based technique is not easily applied to reduction algorithms such as Hessenberg reduction for eigenvalue problems.

2.5.2 Checksum and Reverse Computation-Based Technique

As discussed in Section 2.5.1, when we render right-looking variants of matrix operations fault-tolerant using the parity-based technique, sufficient extra memory to store the entire matrix every iteration is required.

If a smaller amount of extra memory is available in each processor, we need to devise an alternative that recovers the state of the last checkpoint rather than saving it. One solution is to reverse the computations performed since the last checkpoint to recover the state at the most recent checkpoint.

One drawback of this technique is that it may introduce floating-point roundoff errors due to reverse computation. This roundoff error may change any bit in the binary representation of a floating-point number. Thus, incorrect data would be generated when the bitwise exclusive-or operation is performed. Such data is totally unpredictable and may be neither recovered or corrected by the parity-based technique.

Therefore, in a reverse computation technique, we propose to use a checksum, rather than a parity, to encode the data. Since the checksum is a floating-point addition, roundoff errors are minimized, rather than magnified as they are by parity. Admittedly, the possibility exists for overflow, underflow, and roundoff errors due to cancellation when the checksum is computed [Ste73, GL89]. The effect of such problems is considered negligible, however, because each checksum element is composed of just one floating-point number per processor. These issues will be discussed in detail in Section 3.4.6.

2.5.3 Multiple Checkpointing Technique

As discussed in Section 2.4.4, the multiple checkpointing technique is based on the multiple-failure recovery model of using multiple checkpointing processors. It can be used together with either of the previous two techniques to tolerate multiple failures. In our fault-tolerant implementations, one-dimensional parity is implemented in such a way that one checkpointing processor is dedicated to checkpoint a column of processors. Since this technique distributes both checkpointing and recovery processes locally into some groups of processors, the overhead is considerably reduced.

2.6 Performance Metrics

The metrics to evaluate performance of fault-tolerant matrix operations include the checkpointing overhead, extra memory requirements, and the recovery overhead in the presence of failures. The extra memory requirement for checkpointing can be minimized by adapting incremental checkpointing to fault-tolerant implementations. The checkpointing overhead is affected by the following:

1. the checkpointing interval: how often checkpoints are taken.
2. the total checkpoint size taken throughout the algorithm.
3. the additional communication requirement for checkpointing.
4. the computation of the encoding, e.g. bitwise exclusive-or (parity) or addition (checksum) of floating-point numbers.

The recovery overhead is dominated by the following costs:

1. The time to reconfigure the system. This is negligible, since it is simply done by synchronization of all the processors.
2. The time to roll the computations back to the most recent checkpoint. This cost for checksum and reverse computation-based fault-tolerant implementations varies with regard to when failures occur during the lifetime of the algorithm.
3. The time to restore the states of the failed processors. The states are reconstructed from the global checkpoint and live local checkpoints.
4. The execution time required to replay the computations done since the latest checkpoint.

The last two costs are inherent to any rollback recovery scheme.

Chapter 3

Fault-Tolerant Matrix Operations

This chapter describes how different fault tolerance techniques are incorporated into the target matrix operations—LU, QR, and Cholesky factorizations; Hessenberg reduction; the preconditioned conjugate gradient (PCG) iterative method; and matrix multiplication. These matrix operations are well explained in the literature [GL89, DDSvdV91, CDO+96, CDW95, ABD+95, AGZ94, BBC+95, dCH95]. This chapter also presents analytic models of fault-tolerant matrix operations and gives theoretical bounds for checkpointing and recovery overhead for each fault-tolerant matrix operation.

Section 3.1 describes basic concepts on matrix partitioning, mapping, and distribution, and variants of basic factorizations applied to our target matrix operations. Section 3.2 gives an analysis of checkpointing and recovery overhead, which is fundamental in the performance analysis of fault-tolerant matrix operations. Sections 3.3 and 3.4 discuss fault-tolerant matrix operations developed using parity technique and checksum and reverse computation technique, respectively.

3.1 Basic Concepts

In matrix operations, we assume that an $m \times n$ input matrix A is partitioned into square blocks, each of an equal block size b. With such matrix partitioning, a sequential block algorithm of matrix operations is performed in steps, one for each column of blocks, which is called a *column block*. For a parallel implementation, the partitioned matrix A is distributed among the N_p processors logically reconfigured as a $P \times Q$ mesh using two-dimensional block cyclic data distribution, where $N_p = P \times Q$. Therefore, each processor holds $\frac{m}{P}$ row blocks and $\frac{n}{Q}$ column blocks of block size b, where it is assumed that P and Q divide m and n, respectively. Figure 3.1 illustrates the block cyclic data distribution of a matrix with 6×6 blocks over a 2×2 mesh of four processors.

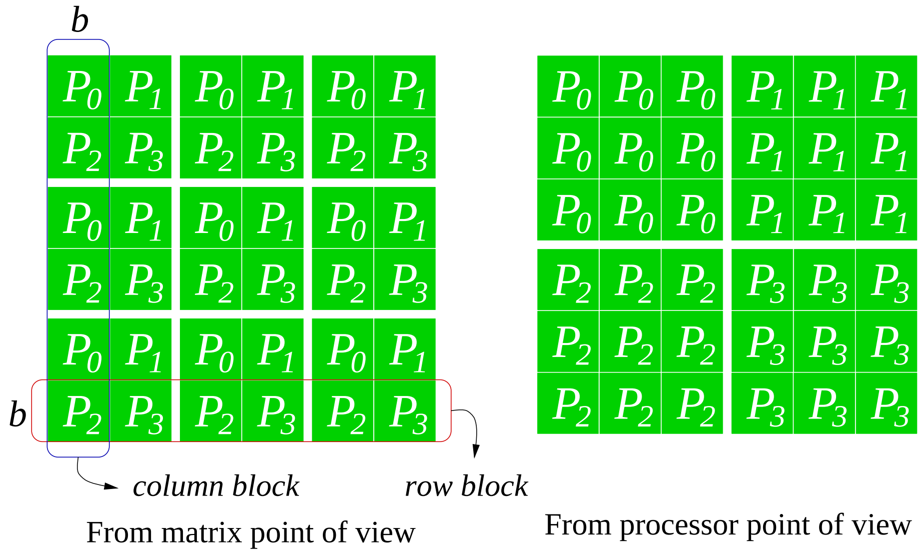


Figure 3.1: Data distribution of a matrix with 6×6 blocksover a 2×2 mesh of processors

Various block algorithms exist for matrix factorizations. In the following subsections, we start with a brief description of block algorithms of matrix factorizations. The remaining matrix operations are discussed when their fault-tolerant algorithms are described.

3.1.1 LU Factorization

In LU factorization, an $m \times n$ real matrix A is factored into a lower triangular matrix L and a upper triangular matrix U, i.e., $PA = LU$, where P is a permutation matrix. At each iteration, one column block is factored and a permutation matrix P is generated, if necessary. The LU factorization is performed in place, and P is stored as a one-dimensional array of the pivoting indices.

Three variants exist for implementing LU factorization on sequential machines. These three block algorithms of LU factorization can be constructed as follows. Suppose that we have factored A as $A = LU$. We write the factors in block form as follows:

$$\begin{pmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{pmatrix} = \begin{pmatrix} L_{11} & 0 & 0 \\ L_{21} & L_{22} & 0 \\ L_{31} & L_{32} & L_{33} \end{pmatrix} \begin{pmatrix} U_{11} & U_{21} & U_{31} \\ 0 & U_{22} & U_{32} \\ 0 & 0 & U_{33} \end{pmatrix}.$$

Multiplying L and U together and equating terms with A, we have

$$\begin{array}{lll} A_{11} = L_{11}U_{11}, & A_{12} = L_{11}U_{12}, & A_{13} = L_{11}U_{13}, \\ A_{21} = L_{21}U_{11}, & A_{22} = L_{21}U_{12} + L_{22}U_{22}, & A_{23} = L_{21}U_{13} + L_{22}U_{23}, \\ A_{31} = L_{31}U_{11}, & A_{32} = L_{31}U_{12} + L_{32}U_{22}, & A_{33} = L_{31}U_{13} + L_{32}U_{23} + L_{33}U_{33}. \end{array}$$

With these relationships, we can develop three variants by manipulating the order in which computations are formed and maintaining the final result of computations in place. These variants are called *ijk variants* [DGK84, DDSvdV91] or, more specifically, *right-looking*, *Crout* or *top-looking*, and *left-looking*, respectively. They differ in which regions of data are accessed and computed during each reduction step. Figure 3.2 shows the different data access patterns among the three variants.

3.1.2 Cholesky Factorization

Cholesky factorization factors an $n \times n$ real, symmetric, positive definite matrix A into a lower triangular matrix L and its transpose L^T, i.e., $A = LL^T$ (or U^TU, where U is upper triangular). Because of the symmetric, positive definite property of the matrix A, Cholesky factorization is also performed in place on either an upper or lower triangular matrix and

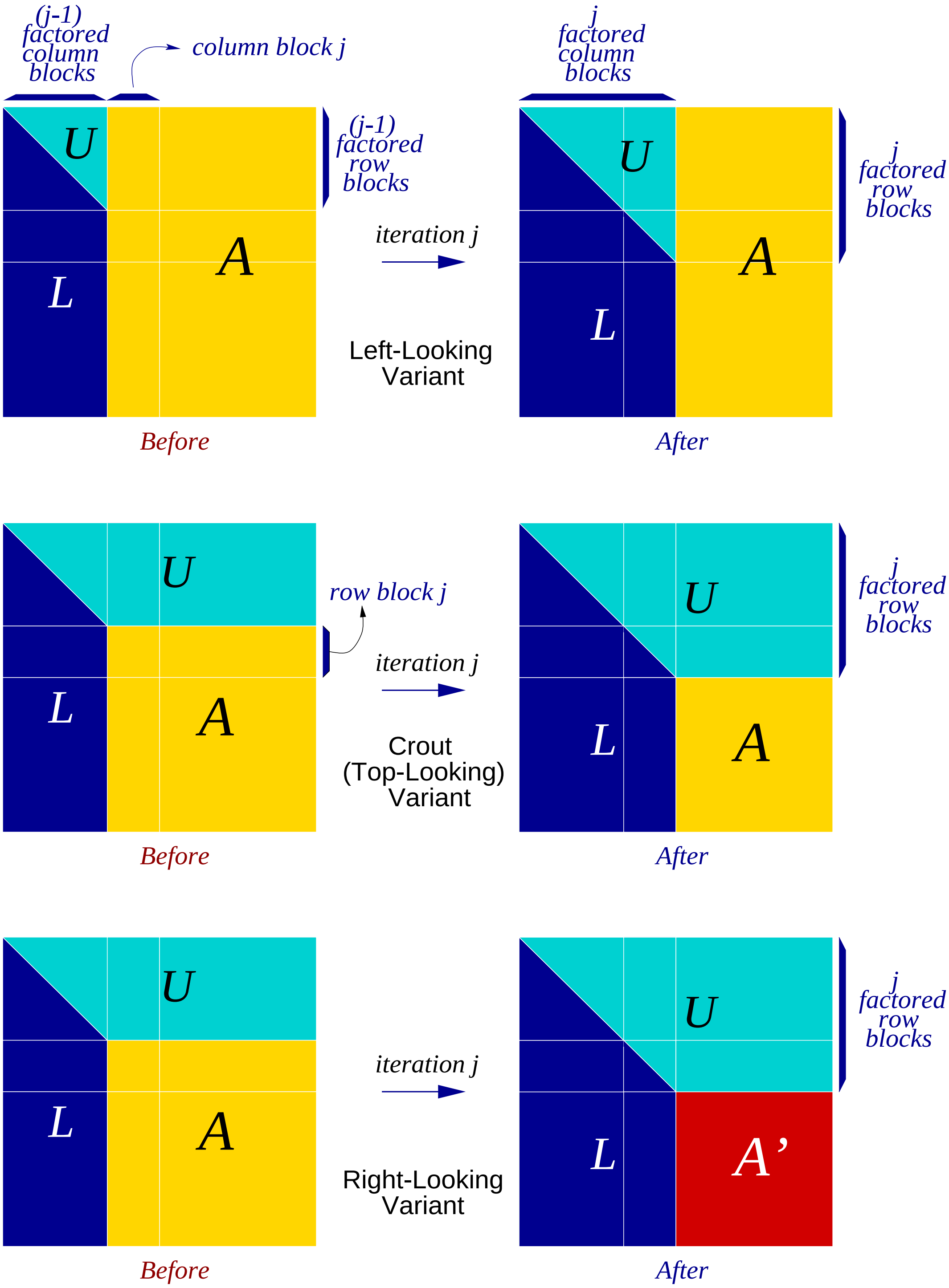


Figure 3.2: Three variants of LU factorizations

involves no pivoting. Three different variants of the Cholesky factorization can be developed as above.

3.1.3 QR Factorization

Given an $m \times n$ real matrix A, QR factorization factors A such that

$$A = Q \begin{pmatrix} R \\ 0 \end{pmatrix},$$

where Q is an $m \times m$ orthogonal matrix and R an $n \times n$ upper triangular matrix. Q is computed by applying a sequence of Householder transformations to the current column block of the form, $H_i = I - \tau_i v_i v_i^T$, where $i = 1, \ldots, b$. In one block QR algorithm, Q can be applied or manipulated through the identity $Q = H_1 H_2 \ldots H_b = I - VTV^T$, where V is a lower triangular matrix of "Householder" vectors v_i, and T is an upper triangular matrix constructed from the triangular factors v_i and τ_i of the Householder transformations. When the factorization is complete, V is stored in the lower triangular part of the original matrix A, R is stored in the upper triangular part of A, and the τ_i's are stored in the diagonal entries of A.

The complete details of this algorithm are described in [GL89] and [DDSvdV91]. Both left-looking and right-looking variants can be constructed.

3.2 Analysis of Checkpointing and Recovery

In this section, we analyze the time complexity of checkpointing and recovering matrices and portions of matrices. This analysis will provide a basic formula for computing the overhead of checkpointing and recovery in each fault-tolerant matrix operation.

3.2.1 Analysis of Basic Checkpointing

The basic checkpointing operation works on a panel of blocks, where each block consists of X floating-point numbers, and the processors are logically configured in a $P \times Q$ mesh (See Figure 3.3). The processors take the checkpoint with a combine operation of `XOR` or addition. This works in a spanning-tree fashion in three parts. The checkpoint is first taken rowwise, then taken columnwise, and then sent to P_C. The first part therefore takes $\lceil \log P \rceil$ steps, and the second part takes $\lceil \log Q \rceil$ steps. Each step consists of sending and then performing either `XOR` or addition on X floating-point numbers. The third part consists of sending the X numbers to P_C.

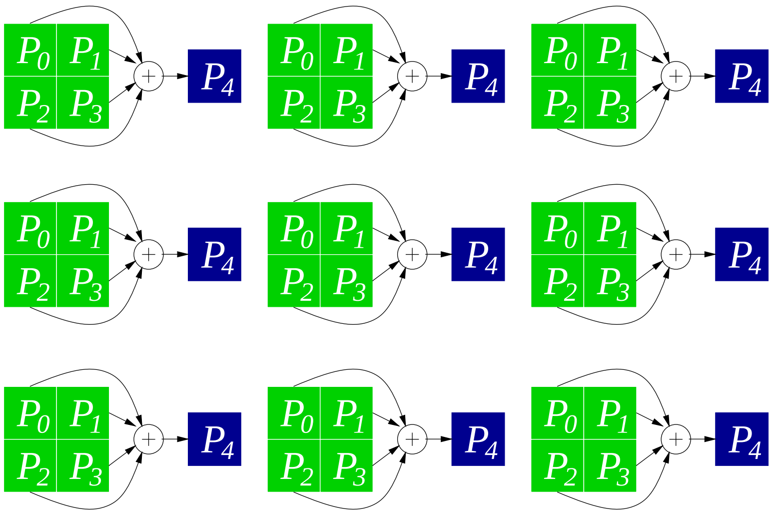


Figure 3.3: Checkpointing the matrix of Figure 3.1

We define the following terms:

- γ: The time for performing a floating-point addition or `XOR` operation.
- α: The startup time for sending a message.
- β: The time to transfer one floating-point number.

The first part takes $\lceil \log P \rceil(\alpha + X(\beta + \gamma))$, the second part takes $\lceil \log Q \rceil(\alpha + X(\beta + \gamma))$, and the third takes $\alpha + X\beta$.

Thus, the total time to checkpoint a panel is the following:

$$T_{panelckpt}(X, P, Q) = (\lceil \log P \rceil + \lceil \log Q \rceil)(\alpha + X(\beta + \gamma)) + (\alpha + X\beta).$$

If we assume that X is large, the α terms disappear, and $T_{panelckpt}$ can be approximated by the following equation:

$$T_{panelckpt}(X, P, Q) \approx X(\beta + (\lceil \log P \rceil + \lceil \log Q \rceil)(\beta + \gamma)). \tag{3.1}$$

To simplify our equations in the subsequent chapters, we define the function

$$T_{ckpt}(P, Q) = \frac{\beta + (\lceil \log P \rceil + \lceil \log Q \rceil)(\beta + \gamma)}{PQ}. \tag{3.2}$$

Note that $T_{panelckpt}(X, P, Q) \approx PQXT_{ckpt}(P, Q)$. For constant values of P and Q, $T_{ckpt}(P, Q)$ is a constant. Thus, $T_{panelckpt}(X, P, Q)$ is directly proportional to X.

Checkpointing an Entire Matrix

At times, an entire $m \times n$ matrix must be checkpointed. If one assumes that m and n are large, the time complexity of this operation is

$$T_{matckpt}(m, n, P, Q) = mnT_{ckpt}(P, Q). \tag{3.3}$$

Checkpointing a Column Block

Often, a column block (block size b) of a matrix must be checkpointed. This is equivalent to checkpointing an $m \times b$ matrix that is distributed along a $P \times 1$ processor grid (i.e., a column of processors). The time complexity of this operation is

$$T_{cbckpt}(m, b, P, Q) = mbT_{ckpt}(P, 1). \tag{3.4}$$

Checkpointing a Row Block

Similarly, a row block (block size b) of a matrix is often checkpointed. This is equivalent to checkpointing a $b \times n$ matrix that is distributed along a $1 \times Q$ processor grid (*i.e.* a row of processors). The time complexity of this operation is

$$T_{rbckpt}(b, n, P, Q) \quad = \quad bnT_{ckpt}(1, Q). \tag{3.5}$$

If $P = Q$, then $T_{ckpt}(1, Q) = T_{ckpt}(P, 1)$. These terms will be used frequently in the analyses that follow.

3.2.2 Analysis of Recovery

When failures occur, the entire matrix should be recovered from the live processors and the checkpointing processor. This operation differs from the operation of checkpointing in that it requires only taking the checkpoint rowwise and then columnwise. Similarly, we also define the function

$$T_{recv}(P, Q) \quad = \quad \frac{(\lceil \log P \rceil + \lceil \log Q \rceil)(\beta + \gamma)}{PQ}. \tag{3.6}$$

Then the time complexities of recovering an entire $m \times n$ matrix, a column block, and a row block are given respectively by

$$\begin{aligned} T_{matrecv}(m, n, P, Q) \quad &= \quad mnT_{recv}(P, Q), &(3.7)\\ T_{cbrecv}(m, b, P, Q) \quad &= \quad mbT_{recv}(P, 1), &(3.8)\\ T_{rbrecv}(b, n, P, Q) \quad &= \quad bnT_{recv}(1, Q). &(3.9) \end{aligned}$$

In practice, $T_{recv}(P, Q)$ is approximated by $T_{ckpt}(P, Q)$.

3.2.3 Analysis of Multiple Checkpointing

In analyzing multiple checkpointing, we assume that one checkpointing processor is dedicated to checkpointing the data over a column of processors (i.e., a $P \times 1$ processor grid). One checkpoint is taken over each column of processors and is then sent to the corresponding checkpointing processor (see Figure 2.4). The first part takes $\lceil \log P \rceil (\alpha + X(\beta + \gamma))$ time, and the second takes $\alpha + X\beta$ time.

Thus, as discussed before, the time overhead of checkpointing a panel of X floating-point numbers can be approximated by the following equation:

$$T_{panelckpt}(X, P, Q) \approx X(\beta + \lceil \log P \rceil (\beta + \gamma)). \tag{3.10}$$

We also define the function

$$T_{mckpt}(P, Q) = \frac{\beta + \lceil \log P \rceil (\beta + \gamma)}{PQ}.. \tag{3.11}$$

The time overhead of checkpointing an $m \times n$ matrix, a column block, and a row block is then

$$T_{matckpt}(m, n, P, Q) = mnT_{mckpt}(P, Q), \tag{3.12}$$

$$T_{cbckpt}(m, n, P, Q) = mbT_{mckpt}(P, 1), \tag{3.13}$$

$$T_{rbckpt}(m, n, P, Q) = bnT_{mckpt}(1, 1). \tag{3.14}$$

Similarly, the time complexity of recovering the matrix is

$$T_{matrecv}(m, n, P, Q) = mnT_{mrecv}(P, Q), \tag{3.15}$$

$$T_{mrecv}(P, Q) = \frac{\lceil \log P \rceil (\beta + \gamma)}{PQ}. \tag{3.16}$$

3.2.4 The Rate of Checkpointing

We define the *checkpointing rate* R to be the rate of sending a message and performing bitwise `XOR` on the message, measured in bytes per second. We assume that a message is a 64-bit floating-point number. The relationship between R and $T_{ckpt}(P,Q)$ is given as follows. For the single checkpointing case,

$$R \approx \frac{\lceil \log P \rceil + \lceil \log Q \rceil}{PQ} \frac{8}{T_{ckpt}(P,Q)}. \tag{3.17}$$

For the multiple checkpointing case,

$$R \approx \frac{\lceil \log P \rceil}{PQ} \frac{8}{T_{mckpt}(P,Q)}. \tag{3.18}$$

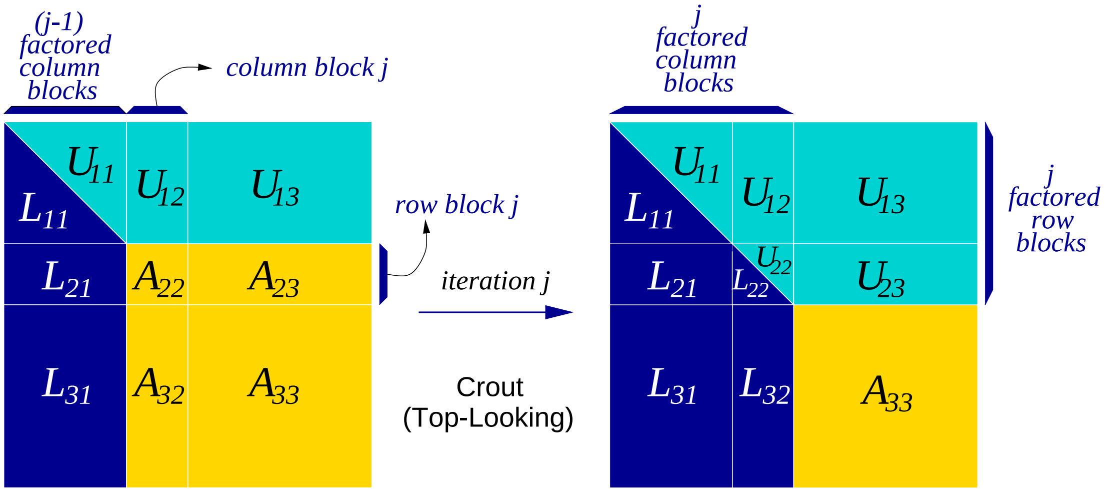
We will use the checkpointing rate R to evaluate the performance of the different fault tolerance techniques in the following chapters.

3.3 Parity-Based Matrix Operations

Note that the parity-based technique is appropriate for matrix operations in which a moderate amount of data is updated between two checkpoints, making the checkpoint size small. This technique can be used to incorporate fault tolerance into the left-looking and top-looking variants of LU, QR, and Cholesky factorizations and the PCG iterative method. In the following subsections, we describe how to take checkpoints and to recover from failures in these matrix operations based on parity. For the purpose of time complexity analysis, we assume that the input matrix is square.

3.3.1 Crout LU Factorization

We begin with a parallel Crout LU factorization, discussing checkpointing and recovery in turn.



I. Update the jth row and column blocks

$$
\begin{aligned}
A_{23} &\leftarrow A_{23} - L_{21}U_{13} \\
\begin{pmatrix} A_{22} \\ A_{32} \end{pmatrix} &\leftarrow \begin{pmatrix} A_{22} \\ A_{32} \end{pmatrix} - \begin{pmatrix} L_{21} \\ L_{31} \end{pmatrix} U_{12}
\end{aligned}
$$

II. Factor the jth column block

$$
P_2 \begin{pmatrix} A_{22} \\ A_{32} \end{pmatrix} = \begin{pmatrix} L_{22} \\ L_{32} \end{pmatrix} U_{22}
$$

III. Pivoting

$$
\begin{aligned}
\begin{pmatrix} A_{23} \\ A_{33} \end{pmatrix} &\leftarrow P_2 \begin{pmatrix} A_{23} \\ A_{33} \end{pmatrix} \\
\begin{pmatrix} L_{21} \\ L_{31} \end{pmatrix} &\leftarrow P_2 \begin{pmatrix} L_{21} \\ L_{31} \end{pmatrix}
\end{aligned}
$$

IV. Solve a triangular system

$$
U_{23} = L_{22}^{-1} A_{23}
$$

Figure 3.4: Iteration j of the Crout LU algorithm

At the beginning of iteration j in the Crout LU algorithm, the leftmost $j-1$ column blocks and the uppermost $j-1$ row blocks have been factored, and the remaining column blocks are unchanged. At iteration j, the jth column block is updated first and then factored. Then, the jth row block is updated. Thus, each iteration looks like that in Figure 3.4.

Time Complexity of the Algorithm:

$$
\begin{aligned}
T_{LU}(m,n,P,Q) &= \sum_{j=1}^{\frac{n}{b}} (T_{mult}(b, jb, n-jb, P, Q) + T_{mult}(m-jb, jb, b, P, Q) \\
&\quad + T_{fact}(m-jb+b, b, P, Q) + T_{swap}(b, n, P, Q))
\end{aligned}
$$

$$
\begin{aligned}
&+ T_{solve}(b, n - jb, 1, Q)) \\
\approx\ & \frac{1}{PQ}\left(mn^2 - \frac{n^3}{3}\right)\gamma + O(n^2).
\end{aligned}
$$

For a square matrix, that is, $m = n$,

$$
T_{LU}(n, n, P, Q) \approx \frac{2n^3}{3PQ}\gamma + O(n^2).
$$

Checkpointing

To checkpoint the Crout LU algorithm, we use one spare processor as the checkpointing processor P_C. For each panel of $P \times Q$ blocks in the matrix, one block in P_C contains the bitwise exclusive-or of each block in the panel (see Figure 3.3). Moreover, each processor (including P_C) allocates memory for an extra column block and two extra row blocks for pivoting. An integer array of size b is also required for saving the current pivoting indices. These extra workspaces are represented by W_C, W_R, W_P, and W_ρ, respectively. In addition to storing and checkpointing the current column block and row block, the pivoting rows and indices should be stored and checkpointed in each iteration. The complete description is given as follows.

1. All the processors checkpoint the matrix A initially.
2. For each iteration j,
   - (a) The processors owning the jth column save their corresponding column blocks into W_C.
   - (b) The processors owning the jth row save their corresponding row blocks into W_R.
   - (c) The checkpointing processor P_C saves the row and column blocks corresponding to the jth row and column into W_R and W_C, respectively.
   - (d) All processors save the b pivoting indices corresponding to the jth row into W_ρ.
   - (e) The application processors perform Steps I and II in Figure 3.4 for the jth column block.
   - (f) The processors owning any pivoting row save their pivoting row into W_P.
   - (g) P_C saves its rows corresponding to the pivoting rows into W_P.
   - (h) The application processors perform the remaining computation steps.

(i) The processors checkpoint the jth column block, row block, the pivoting rows, and indices.

(j) The processors synchronize and go to the next iteration.

Memory Requirement for W_C, W_R, W_P, W_ρ: As stated above, the following extra memory in floating-point number per processor is required for checkpointing. We assume that an integer size is half of that of a double-precision floating-point number.

$$M_C = \left(\frac{n}{P} + 2\frac{n}{Q} + \frac{1}{2}\right) b.$$

Time Complexity for Checkpointing T_C: The time overhead of checkpointing consists of the time to checkpoint the initial matrix, which is equivalent to $T_{matckpt}$, and the time overhead of checkpointing the current column block, row block, pivoting rows, and indices generated at each iteration. For large n, the time of copying and checkpointing the pivoting indices is negligible compared with that of copying and checkpointing row and column blocks.

$$\begin{aligned}
T_C &= T_{matckpt}(n, n, P, Q) + \sum_{j=0}^{N-1} \Big(T_{copy}(\frac{n-jb}{P}, b) + 2T_{copy}(b, \frac{n}{Q}) + T_{copy}(\frac{b}{2}, 1) \\
&\quad + T_{cbckpt}(n - jb, b, P, Q) + 2T_{rbckpt}(b, n, P, Q) + T_{cbckpt}(\frac{b}{2}, 1, 1, 1) \Big) \\
&\approx n^2 T_{ckpt}(P, Q) + \sum_{j=0}^{N-1} \left((n - jb) b T_{ckpt}(P, 1) + 2bn T_{ckpt}(1, Q)\right) \\
&\approx n^2 \left(T_{ckpt}(P, Q) + \frac{1}{2} T_{ckpt}(P, 1) + 2T_{ckpt}(1, Q) \right).
\end{aligned}$$

Note that $N = \frac{n}{b}$ is the total number of iterations performed during the lifetime of the algorithm.

Recovery

For rollback recovery, the following steps are performed by the live processors and P_C when a failure occurs.

1. Reconfigure the processor grid: the failed processor is replaced by P_C or a spare processor.

2. Restore the current column block, row block, and pivoting rows and indices from W_C, W_R, W_P, and W_ρ, respectively.

3. Recover the failed processor's data from the parity and the live processors' data.

4. Resume the computation from the beginning of the current iteration.

Time Complexity for Recovery T_R: The time overhead of recovery consists of the time of reconfiguring (i.e., synchronization) in Step 1, the time of copying in Step 2, and the time of recovering the matrix in Step 3. Since Steps 1 and 2 are $O(1)$ and $O(n)$ operations, respectively, Step 3 dominates the time overhead of recovery. In the analyses that follow, the time of copying and recovering the pivoting indices as well as the synchronization time is also neglected.

$$
\begin{aligned}
T_R &= T_{barrier}(P,Q) + T_{copy}(\frac{n}{P}, b) + 2T_{copy}(b, \frac{n}{Q}) + T_{copy}(\frac{b}{2}, 1) \\
&\quad + T_{matrecv}(n, n, P, Q) + T_{matrecv}(\frac{n}{2}, 1, P, Q) \\
&\approx n^2 T_{recv}(P,Q).
\end{aligned}
$$

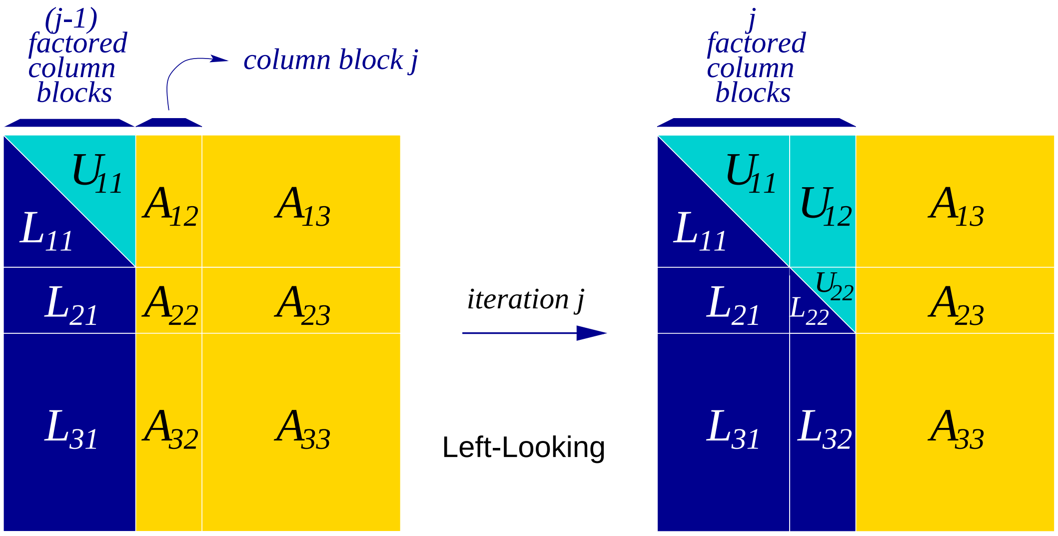
The complexity summary of the Crout LU algorithm is presented in Table 3.1.

Table 3.1: Complexity summary of the Crout LU algorithm

T_{LU}	$n^3 \frac{2}{3PQ}\gamma + O(n^2)$
T_C	$n^2 \left(\frac{1}{PQ}((\lceil\log P\rceil + \lceil\log Q\rceil)(\tau+\beta)+\beta) + \frac{1}{2P}(\lceil\log P\rceil(\tau+\beta)+\beta) + \frac{2}{Q}(\lceil\log Q\rceil(\tau+\beta)+\beta)\right) + O(n)$
T_R	$n^2 \frac{1}{PQ}(\lceil\log P\rceil + \lceil\log Q\rceil)(\tau+\beta) + O(n)$
M_C	$\left(\frac{n}{P} + 2\frac{n}{Q} + \frac{1}{2}\right) b$

3.3.2 Left-Looking LU Factorization

At the beginning of iteration j in the left-looking LU algorithm, the leftmost $j-1$ column blocks have been factored, and the remaining column blocks are unchanged (Figure 3.5). In iteration j, the jth column block is updated and then factored.



I. Solve a triangular system

$$U_{12} \quad = \quad L_{11}^{-1} A_{12}$$

II. Update the jth column block

$$\begin{pmatrix} A_{22} \\ A_{32} \end{pmatrix} \quad \leftarrow \quad \begin{pmatrix} A_{22} \\ A_{32} \end{pmatrix} - \begin{pmatrix} L_{21} \\ L_{31} \end{pmatrix} U_{12}$$

III. Factor the jth column block

$$P_2 \begin{pmatrix} A_{22} \\ A_{32} \end{pmatrix} \quad = \quad \begin{pmatrix} L_{22} \\ L_{32} \end{pmatrix} U_{22}$$

IV. Pivoting

$$\begin{pmatrix} A_{23} \\ A_{33} \end{pmatrix} \quad \leftarrow \quad P_2 \begin{pmatrix} A_{23} \\ A_{33} \end{pmatrix},$$
$$\begin{pmatrix} L_{21} \\ L_{31} \end{pmatrix} \quad \leftarrow \quad P_2 \begin{pmatrix} L_{21} \\ L_{31} \end{pmatrix}$$

Figure 3.5: Iteration j of the left-looking LU algorithm

The checkpointing and recovery technique of the left-looking LU algorithm differs from that of the top-looking LU algorithm only in that it involves all blocks of the current column. Its complexity summary is presented in Table 3.2.

Table 3.2: Complexity summary of the left-looking LU algorithm

T_{LU}	$n^3 \frac{2}{3PQ}\gamma + O(n^2)$
T_C	$n^2 \left(\frac{1}{PQ}((\lceil \log P \rceil + \lceil \log Q \rceil)(\tau+\beta)+\beta) + \frac{1}{P}(\lceil \log P \rceil(\tau+\beta)+\beta) + \frac{1}{Q}(\lceil \log Q \rceil(\tau+\beta)+\beta) \right) + O(n)$
T_R	$n^2 \frac{1}{PQ}(\lceil \log P \rceil + \lceil \log Q \rceil)(\tau+\beta) + O(n)$
M_C	$\left(\frac{n}{P} + 2\frac{n}{Q} + \frac{1}{2} \right) b$

3.3.3 Left-Looking Cholesky Factorization

We consider the left-looking algorithm of Cholesky factorization. Again, we discuss the parallel algorithm and methods for checkpointing and recovery.

A Parallel Algorithm

The left-looking Cholesky algorithm is a special case of the Crout LU algorithm without pivoting. Figure 3.6 depicts the preservation of symmetry.

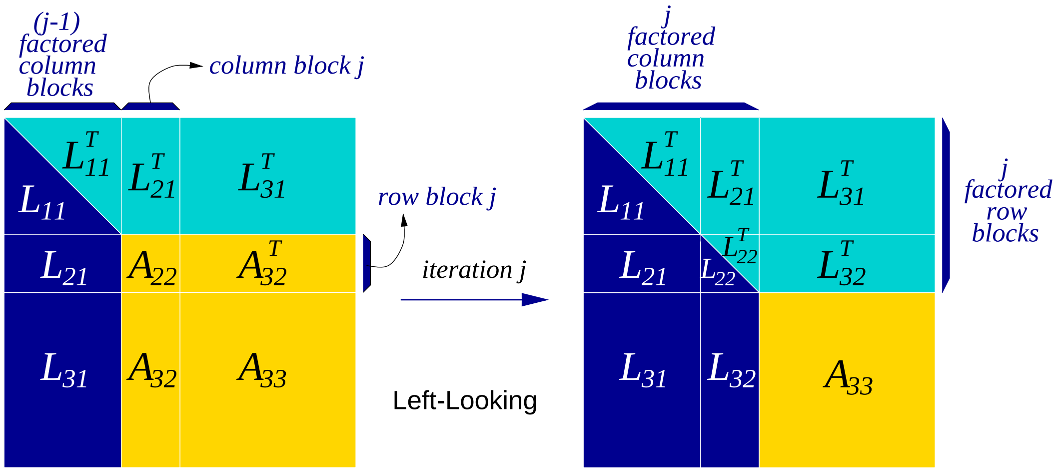
Time Complexity of the Algorithm:

$$
\begin{aligned}
T_{LLT}(n,n,P,Q) &= \sum_{j=1}^{\frac{n}{b}} (T_{rank}(b,jb,b,P,Q) + T_{fact}(b,b,1,1) \\
&\qquad + T_{mult}(n-jb-b,b,jb,P,Q) + T_{solve}(b,n-jb-b,P,Q)) \\
&\approx \frac{n^3}{3PQ}\gamma + O(n^2).
\end{aligned}
$$

Checkpointing

To checkpoint the algorithm, the only extra memory required is for storing the current column. This memory is represented by W_C.

1. All the processors checkpoint the matrix A initially.



I. Update the jth column blocks

$$\begin{pmatrix} A_{22} \\ A_{32} \end{pmatrix} \leftarrow \begin{pmatrix} A_{22} \\ A_{32} \end{pmatrix} - \begin{pmatrix} L_{21} \\ L_{31} \end{pmatrix} L_{21}^T$$

II. Factor the jth column block

$$A_{22} = L_{22} L_{22}^T$$

III. Solve a triangular system

$$L_{32}^T = L_{22}^{-1} A_{32}^T$$

Figure 3.6: Iteration j of the left-looking Cholesky algorithm

2. For each iteration j,
   (a) Save the jth column block into W_C.
   (b) Perform factorization on the jth column block.
   (c) Checkpoint the jth column block.

Memory Requirement for W_C: $M_C = \frac{n}{P} b$.

Time Complexity for Checkpointing T_C: Because of symmetry, only half of the entire matrix needs to be checkpointed. Thus, T_C is approximated by

$$\begin{aligned} T_C &\approx \frac{1}{2} T_{matckpt}(n, n, P, Q) + \sum_{j=0}^{N-1} T_{cbckpt}(n - jb, b, P, Q) \\ &\approx n^2 \left(\frac{1}{2} T_{ckpt}(P, Q) + \frac{1}{2} T_{ckpt}(P, 1) \right). \end{aligned}$$

Recovery

1. Reconfigure the processor grid.
2. Restore the current column block from W_C.
3. Recover the failed processor's data from the parity data.
4. Resume the computation from the beginning of the current iteration.

Time Complexity for Recovery T_R: The time overhead of recovery is dominated by Step 3 and approximated by

$$T_R \approx \frac{1}{2}T_{matrecv}(n, n, P, Q) \approx n^2\frac{1}{2}T_{recv}(P, Q).$$

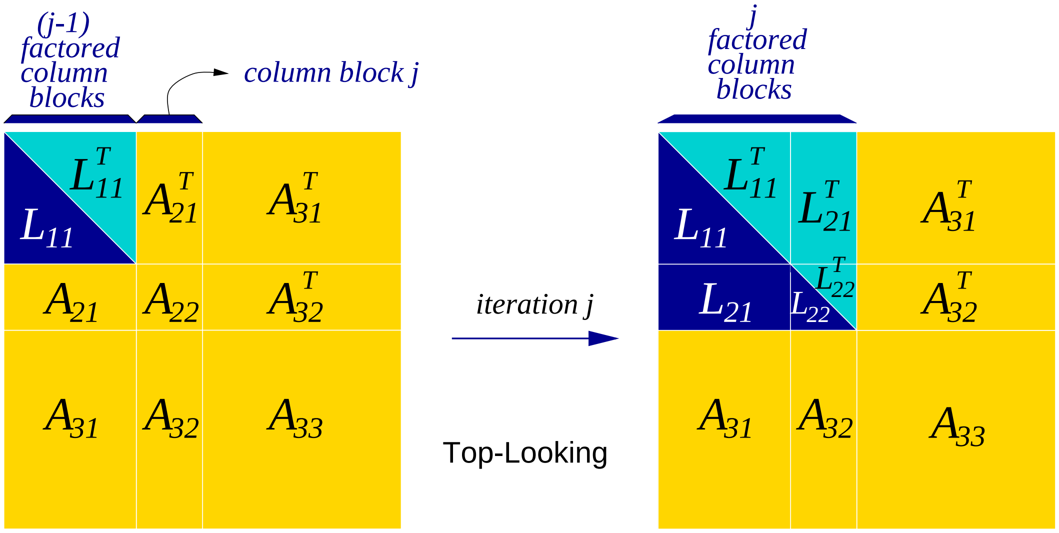
Table 3.3 presents the complexity summary of the left-looking Cholesky algorithm.

Table 3.3: Complexity summary of the left-looking Cholesky algorithm

T_{LLT}	$n^3 \frac{1}{3PQ}\gamma + O(n^2)$
T_C	$n^2 \left(\frac{1}{2PQ}\left((\lceil\log P\rceil + \lceil\log Q\rceil)(\tau+\beta)+\beta\right) + \frac{1}{2P}(\lceil\log P\rceil(\tau+\beta)+\beta)\right) + O(n)$
T_R	$n^2 \frac{1}{2PQ}((\lceil\log P\rceil + \lceil\log Q\rceil)(\tau+\beta) + O(n)$
M_C	$\left(\frac{n}{P}\right) b$

3.3.4 Top-Looking Cholesky Factorization

Each iteration in the top-looking Cholesky algorithm looks like that in Figure 3.7. The nature of the top-looking Cholesky algorithm with regards to fault tolerance is similar to the left-looking Cholesky algorithm. Since Step I updates the current row block, it is involved in both checkpointing and recovery. Its complexity analysis is presented in Table 3.4.



I. Solve a triangular system

$$L_{21}^T \quad = \quad L_{11}^{-1} A_{21}^T$$

II. Update the diagonal block

$$A_{22} \quad \leftarrow \quad A_{22} - L_{21} L_{21}^T$$

III. Factor the jth column block

$$A_{22} \quad = \quad L_{22} L_{22}^T$$

Figure 3.7: Iteration j of the top-looking Cholesky algorithm

Table 3.4: Complexity summary of the top-looking Cholesky algorithm

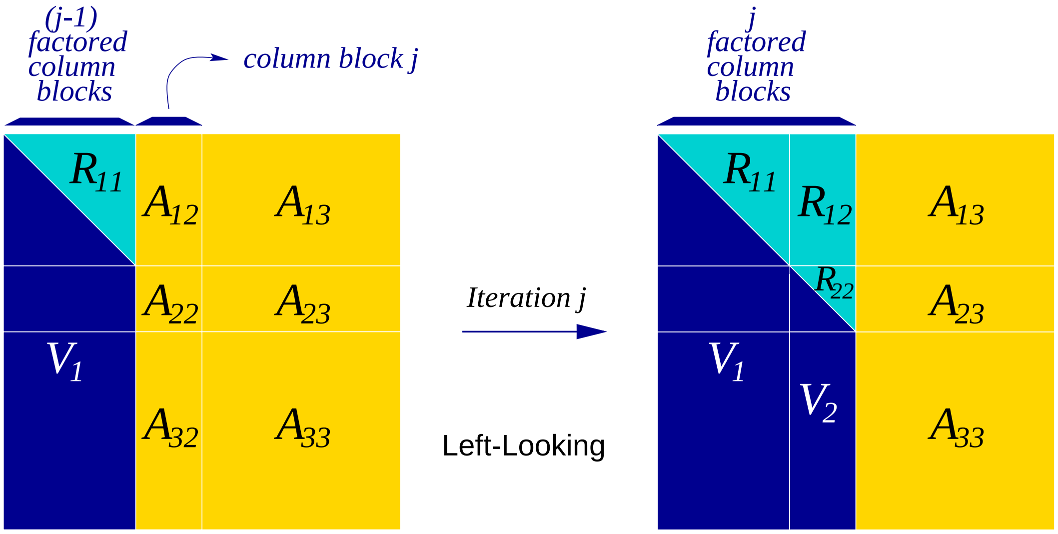
T_{LLT}	$n^3 \frac{1}{3PQ}\gamma + O(n^2)$
T_C	$n^2 \left(\frac{1}{2PQ}((\lceil \log P \rceil + \lceil \log Q \rceil)(\tau + \beta) + \beta) + \frac{1}{P}(\lceil \log P \rceil (\tau + \beta) + \beta) \right) + O(n)$
T_R	$n^2 \frac{1}{2PQ}(\lceil \log P \rceil + \lceil \log Q \rceil)(\tau + \beta) + O(n)$
M_C	$\left(\frac{n}{P} + \frac{n}{Q} \right) b$

3.3.5 Left-Looking QR Factorization

Next, we consider the left-looking QR factorization.

A Parallel Algorithm

At the beginning of iteration j, the leftmost $j-1$ column blocks have been factored, and the remaining column blocks are unchanged (Figure 3.8). In Step I, the jth column block must be updated using the previous Householder matrix Q_1. In Step II, the jth column block is factored.



I. Update the jth column block.

$$\begin{pmatrix} R_{12} \\ A_{22} \\ A_{32} \end{pmatrix} \leftarrow Q_1^T \begin{pmatrix} A_{12} \\ A_{22} \\ A_{32} \end{pmatrix},$$

where $Q_1 = (I - V_1 T_1 V_1^T)$.

II. Factor the jth column block.

$$\begin{pmatrix} A_{22} \\ A_{32} \end{pmatrix} = Q_2 \begin{pmatrix} R_{22} \\ 0 \end{pmatrix},$$

where $Q_2 = (I - V_2 T_2 V_2^T)$.

Figure 3.8: Iteration j of the left-looking QR algorithm

Time Complexity of the Algorithm:

$$T_{QR}(m,n,P,Q) = \sum_{j=0}^{N-1} (T_{pre}(m,b,P,Q) + T_{fact}(m-jb,b,P,1))$$

$$\approx \frac{2n^2}{PQ}\left(m - \frac{n}{3}\right)\gamma + O(mn).$$

If $m = n$,

$$T_{QR}(n, n, P, Q) \approx \frac{4n^3}{3PQ}\gamma + O(n^2).$$

Checkpointing

1. All the processors checkpoint the matrix A initially.
2. For each iteration j,
   (a) Save the jth column block into W_C.
   (b) Perform factorization on the jth column block.
   (c) Checkpoint the current column block.

Memory requirement for W_C: $M_C = \frac{n}{P}b$.

Time Complexity for Checkpointing T_C:

$$\begin{aligned} T_C &= T_{matckpt}(n, n, P, Q) + \sum_{j=1}^{\frac{n}{b}} T_{cbckpt}(n, b, P, Q) \\ &\approx n^2\left(T_{ckpt}(P, Q) + T_{ckpt}(P, 1)\right). \end{aligned}$$

Recovery

1. Reconfigure the processor grid.
2. Restore the current column block from W_C.
3. Recover the failed processor's data from the parity data.
4. Resume the computation from the beginning of the current iteration.

Time Complexity for Recovery T_R: Since Step 3 also dominates T_R, it is given by

$$T_R \approx T_{matrecv}(n, n, P, Q) \approx n^2 T_{recv}(P, Q).$$

Table 3.5 presents the complexity summary of the left-looking QR algorithm.

Table 3.5: Complexity summary of the left-looking QR algorithm

T_{QR}	$n^3 \frac{4}{3PQ}\gamma + O(n^2)$
T_C	$n^2 \left(\frac{1}{PQ}((\lceil \log P \rceil + \lceil \log Q \rceil)(\tau + \beta) + \beta) + \frac{1}{P}(\lceil \log P \rceil(\tau + \beta) + \beta)\right) + O(n)$
T_R	$n^2 \frac{1}{PQ}(\lceil \log P \rceil + \lceil \log Q \rceil)(\tau + \beta) + O(n)$
M_C	$\frac{n}{P} b$

3.3.6 An Iterative Equation Solver (PCG)

Iterative equation solvers are used for the following problem: Given a large sparse matrix A and a vector b, find the vector x such that $Ax = b$. Iterative equation solvers work by providing an initial approximation to x and then iteratively refining this approximation until $Ax = b$ to within some error tolerance. Unfortunately, no single iterative method is robust enough to solve all sparse linear systems accurately and efficiently. Therefore, we limit our scope to one such method, known as "preconditioned conjugate gradient" (PCG).

If A is symmetric positive definite, PCG can be used to solve the system $Ax = b$ by projecting A onto a "Krylov subspace" and then solving the system in this subspace. In PCG, two matrices M_1 and M_2, called *left-* and *right-preconditioners*, respectively, are often employed to improve the method's convergence rate. The details of the algorithm can be found in [GL89, DDSvdV91, BBC$^+$95]. Its mechanics as they affect fault tolerance are simple. First, the sparse matrix A and the two preconditioners M_1 and M_2 are represented in a dense form and are then distributed along with b among the application processors. After this point, A, M_1, M_2, and b are not altered.

Now, the vectors p_0, r_0, w_0 and ξ_0 are calculated from A, M_1, M_2, and b. These intermediate vectors are used to calculate the vector x_0, which is the first approximation to x. The algorithm then iterates as follows: The values of A, M_1, M_2, b, x_{i-1}, p_{i-1}, r_{i-1}, w_{i-1} and ξ_{i-1} are used to calculate p_i, r_i, w_i and ξ_i. These are then used to calculate x_i, the ith approximation to x. The iterations continue until $Ax_i = b$ to within a given error tolerance.

1. $r_0 = M_1(b - AM_2x_0)$
2. $p_0 = r_0$
3. $\rho_0 = r_0^T r_0$
4. $\omega_0 = M_1AM_2p_0$
5. $\xi_0 = p_0^T \omega_0$

`for` $i = 1, 2, \ldots$

6. $\alpha_{i-1} = \rho_{i-1}/\xi_{i-1}$
7. $x_i = x_{i-1} + \alpha_{i-1}p_{i-1}$
8. $r_i = r_{i-1} - \alpha_{i-1}w_{i-1}$
9. `check stopping criterion`
10. $s_i = M_1AM_2r_i$
11. $\rho_i = r_i^T r_i$
12. $\delta_i = r_i^T s_i$
13. $\beta_i = \rho_i/\rho_{i-1}$
14. $p_i = r_i + \beta_i p_{i-1}$
15. $w_i = s_i + \beta_i w_{i-1}$
16. $\xi_i = \delta_i - \beta_i^2 \xi_{i-1}$

`end`

Figure 3.9: A failure-free PCG algorithm

Checkpointing and Recovery

It is straightforward to incorporate fault tolerance into the PCG algorithm. First, the processors distribute A, M_1, M_2, b and allocate memory W_i for x_i, p_i, r_i, w_i, and ξ_i. The checkpointing processor P_C is initialized to contain the bitwise exclusive-or of each these variables. Now, each processor (including P_C) must include extra vectors for each of x, p, r, w, and ξ. These extra vectors are maintained just as is W_C in the factorization examples. They hold the values of x_{i-1}, p_{i-1}, r_{i-1}, w_{i-1}, and ξ_{i-1} during iteration i so that the iteration can be rolled back following a failure.

Note that in PCG, we can checkpoint every K iterations, by copying x_i, p_i, r_i, w_i, and ξ_i to the extra vectors and computing the bitwise exclusive-or of x, p, r, w, and ξ only at the end of every K iterations. Each PCG iteration takes $O(n^2)$ time, where n is the matrix size, because of the matrix-vector multiplication. Each checkpoint requires $O(n)$ time and recovery requires $O(n)$ time if A, M_1, and M_2 are sparse and hence stored in a dense form. W_i requires $O(n)$ extra memory.

3.3.7 Summary

The fundamental idea of the parity-based technique is to checkpoint initially the entire data and then checkpoint the part of the data that has been modified since the last checkpoint. To take advantage of this technique, we have incorporated the algorithms that have the characteristic that a moderate amount of data is modified between two checkpoints. The natural choice includes left-looking or top-looking variants of Cholesky, LU, and QR factorization algorithms. Also, an iterative method such as PCG is shown to be appropriate for this technique. In such algorithms, the same computation repeats on some moderate amount of data, for example, a row or column block or a small number of vectors.

From the analytic models presented in the preceding sections, the total overhead of $O(n^2)$ time for checkpointing and recovery is small compared with the time of $O(n^3)$ that the algorithm without checkpointing takes. Thus, the technique is efficient when one use a moderate amount of memory to provide fault tolerance for an algorithm. However, there are other algorithms, such as matrix multiplication, whose fault-tolerant versions may not be easily developed using this scheme. To checkpoint such an algorithm, one must have sufficient memory to save its entire data; this requirement clearly wastes resources. In the next section, we discuss how another technique is incorporated into this class of matrix operations.

The multiple checkpointing technique can also be incorporated to tolerate multiple failures. An an example, with a two-dimensional processor grid, if one-dimensional parity is used, one checkpointing processor is allocated to either each row or each column of processors.

3.4 Checksum and Reverse Computation-Based Matrix Operations

The following subsections describe the fault-tolerant algorithms based on the checksum and reverse computation technique. This technique is useful for those matrix operations in which the checkpoint size is too large to save in extra memory space. In this section, we discuss fault-tolerant versions of several matrix operations: matrix multiplication, right-looking factorization algorithms, and Hessenberg reduction.

3.4.1 Matrix Multiplication

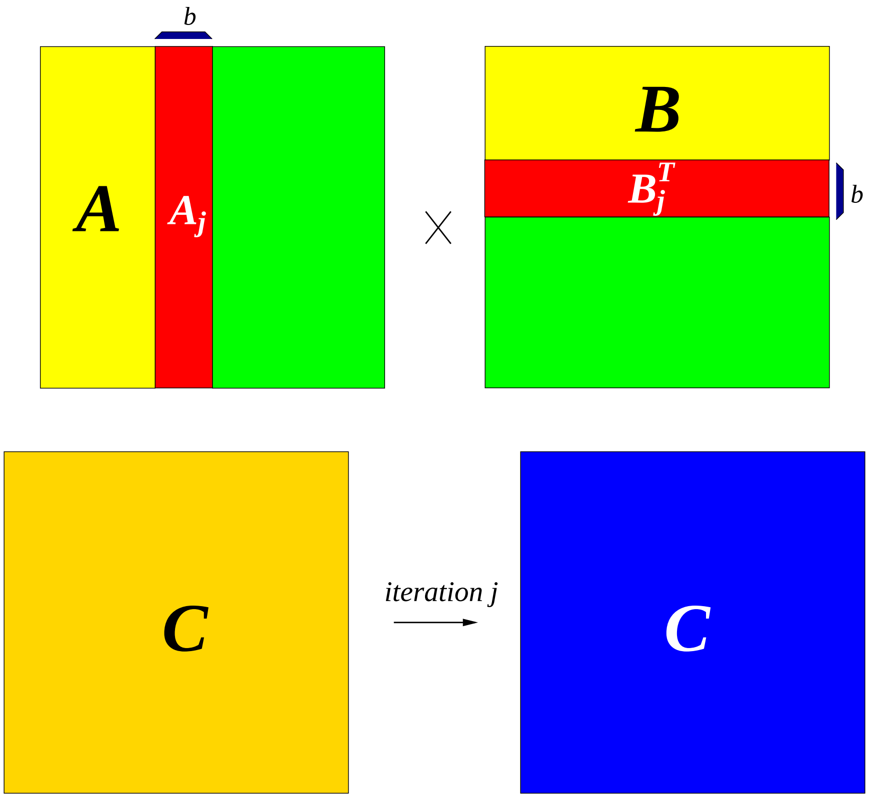
A Parallel Algorithm

Let an $m \times k$ matrix A be multiplied by a $k \times n$ matrix B to produce the $m \times n$ matrix C. Matrix-matrix multiplication can be formulated as a sequence of rank-one updates by

$$C = C + \sum_{j=0}^{k} A_j B_j^T,$$

where A_j is the jth column vector of A and B_j^T is the jth row vector of B. Assuming that the matrices are partitioned into blocks of size b, its corresponding block algorithm is depicted in Figure 3.10. The block algorithm performs the rank-b updates of a column block A_j and a row block B_j^T. Note that one rank-b update operation is equivalent to the multiplication of an $m \times b$ matrix by a $b \times n$ matrix so that the result is added to an existing matrix.

A parallel algorithm can be obtained by parallelizing the rank-b update at each iteration [AGZ94]. Each iteration j in the parallel algorithm must broadcast a column block A_j of A within the column, broadcast a row block B_j^T of B within the row, and then do the rank-b update. Assuming that the matrices are distributed over a $P \times Q$ processor grid using a two-dimensional block cyclic data distribution with block size b and that a minimum



$$C \leftarrow C + A_j \times B_j^T$$

Figure 3.10: Iteration j of the algorithm of matrix multiplication

spanning tree broadcast is used, the time complexity of the algorithm is given by

$$\begin{aligned} T_{MM}(m,n,k,P,Q) &= \sum_{j=1}^{\frac{k}{b}} T_{rank}(m,n,b,P,Q) \\ &\approx \frac{k}{b}\left(\frac{2mn}{PQ}b\gamma + \lceil \log Q \rceil(\alpha + \frac{m}{P}b\beta) + \lceil \log P \rceil(\alpha + \frac{n}{Q}b\beta)\right). \end{aligned}$$

Here, if $m = n = k$, it is given by

$$T_{MM}(n,n,n,P,Q) = \frac{n}{b}\left(\frac{2n^2}{PQ}b\gamma + \lceil \log Q \rceil(\alpha + \frac{n}{P}b\beta) + \lceil \log P \rceil(\alpha + \frac{n}{Q}b\beta)\right)$$

$$\approx \quad n^3 \frac{2\gamma}{PQ} + O(n^2).$$

Checkpointing

It is straightforward to incorporate fault tolerance into the algorithm. Since the result matrix C is modified by one rank-b update at each iteration, a fault-tolerant algorithm needs to checkpoint only the matrix C periodically at the end of an iteration. In our implementation, the matrix C is checkpointed at the end of every K of iterations, where K is chosen by the programmer. Specifically, the fault-tolerant algorithm works as follows:

1. Checkpoint the matrices A, B, and C initially.
2. Perform K rank-b updates.
3. Checkpoint the matrix C, where this checkpoint is done with addition.
4. Synchronize and go to Step 2.

Memory requirement: No extra memory is required for checkpointing.
Time complexity for checkpointing, T_C: The time of performing Step 1 is equivalent to $3 \times T_{matckpt}(n, n, P, Q)$. The time of performing Step 3 at each checkpoint is given by $T_{matckpt}(n, n, P, Q)$. Thus, T_C is

$$\begin{aligned} T_C &= 3T_{matckpt}(n, n, P, Q) + \frac{n}{Kb} T_{matckpt}(n, n, P, Q) \\ &\approx (3 + \frac{n}{Kb}) n^2 T_{ckpt}(P, Q). \end{aligned}$$

Recovery

Suppose that each checkpoint is taken every K of iterations and that the last checkpoint is taken at iteration i and hence the next checkpoint is taken at iteration $i + K$. Assuming that a failure occurs at iteration f in the current checkpointing interval, then the following operations are performed for recovery.

1. Reconfigure the processor grid.
2. Recover the failed processor's data of A and B using the checksum and the live processors' data.

3. Reverse the computations of the live processors from the current iteration l to the iteration $i+1$ by performing the following computations:

$$C \leftarrow C - \sum_{j=l}^{i+1} A_j B_j^T.$$

4. Recover the failed processor's data of C using the checksum of P_C.

5. Resume the computation at iteration $i+1$.

As shown above, recovery involves both reversing the computation and recovering three matrices using the checksum.

Time complexity for recovery, T_R: The time overhead of recovery consists of the time of performing Steps 3 and 4. Thus, T_R is approximated by

$$\begin{aligned} T_R &= T_{MM}(n, n, Kb, P, Q) + 3T_{matrecv}(n, n, P, Q) \\ &\approx n^2 \frac{2Kb}{P}\gamma + 3n^2 T_{recv}(P, Q). \end{aligned}$$

Table 3.6 presents the complexity summary of the matrix multiplication algorithm.

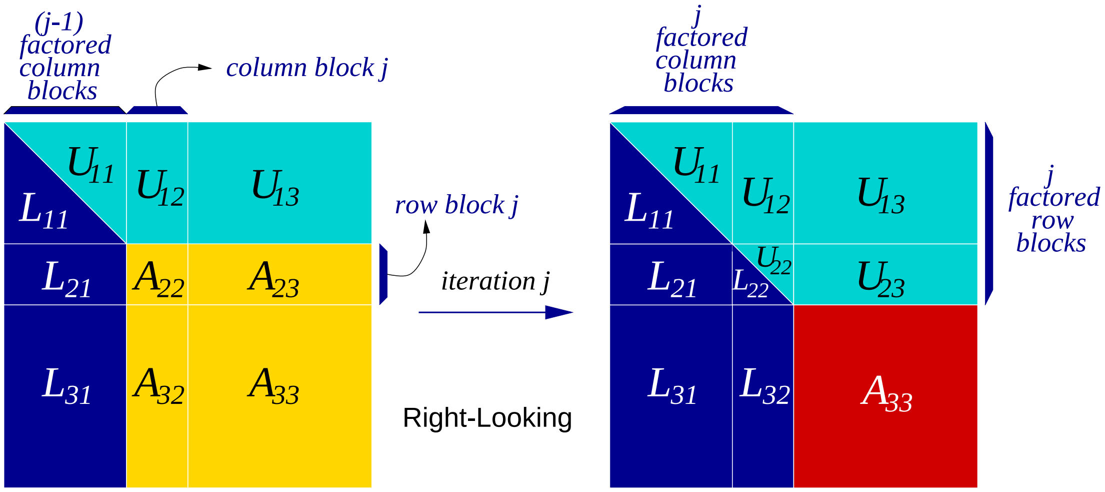
Table 3.6: Complexity summary of the algorithm of matrix multiplication

T_{MM}	$n^3 \frac{2\gamma}{PQ} + O(n^2)$
T_C	$(\frac{n}{Kb} + 3)n^2 \frac{1}{PQ}(((\lceil \log P \rceil + \lceil \log Q \rceil)(\gamma + \beta) + \beta) + O(n^2)$
T_R	$n^2 \left(\frac{2Kb}{P}\gamma + \frac{3}{PQ}(\lceil \log P \rceil + \lceil \log Q \rceil)(\gamma + \beta) \right) + O(n)$
M_C	None

3.4.2 Right-Looking LU Factorization

A Parallel Algorithm

Unlike the other variants, the right-looking algorithms take advantage of matrix multiplication as the dominant operation. In the right-looking LU algorithm, at the beginning of iteration j the leftmost $j-1$ column blocks and uppermost $j-1$ row blocks have been factored. At iteration j, the current column block is factored first and then the remaining matrix is updated by one rank-b update using the current column block and row block. Pivoting is also done before the rank-b update. Thus, each iteration looks as in Figure 3.11.



I. Factor the jth column block

$$P_2 \begin{pmatrix} A_{22} \\ A_{32} \end{pmatrix} = \begin{pmatrix} L_{22} \\ L_{32} \end{pmatrix} U_{22}$$

II. Pivoting

$$\begin{pmatrix} A_{23} \\ A_{33} \end{pmatrix} \leftarrow P_2 \begin{pmatrix} A_{23} \\ A_{33} \end{pmatrix},$$

$$\begin{pmatrix} L_{21} \\ L_{31} \end{pmatrix} \leftarrow P_2 \begin{pmatrix} L_{21} \\ L_{31} \end{pmatrix}$$

III. Solve a triangular system.

$$U_{23} = L_{22}^{-1} A_{23}$$

IV. Update the remaining matrix.

$$A_{33} \leftarrow A_{33} - L_{32} U_{23}$$

Figure 3.11: Iteration j of the right-looking LU algorithm

Time complexity of the Algorithm:

$$
\begin{aligned}
T_{LU}(n,n,P,Q) &= \sum_{j=1}^{\frac{n}{b}} (T_{fact}(n,b,P,1) + T_{swap}(n,b,P,Q) \\
&\qquad + T_{solve}(b, n-jb, b, 1, Q) + T_{rank}(n-jb, n-jb, b, P, Q)) \\
&\approx \frac{2n^3}{3PQ}\gamma + O(n^2).
\end{aligned}
$$

Checkpointing

Assume that one checkpoint is taken every K of iterations. Like the top-looking LU algorithm, extra memory for W_C, W_R, W_P, and W_ρ is required for checkpointing the right-looking LU algorithm. The $\lceil \frac{K}{Q} \rceil$ row and column blocks to be factored during the current checkpointing interval of K iterations are saved in W_R, and W_C, respectively. The pivoting rows and indices to be generated during the current interval are saved in W_P and W_ρ, respectively.

At an iteration j, the factored L_j and U_j are to be checkpointed as well as the pivoting rows and indices. At the end of every checkpointing interval the newly modified remaining matrix is checkpointed. The entire algorithm of checkpointing is described as follows.

1. All the processors checkpoint the matrix A initially.
2. For every K of iterations,
   (a) For each iteration j $(i+1 \leq j \leq i+K)$,
      i. The processors owning the jth column save their corresponding column blocks into W_C.
      ii. The processors owning the jth row save their corresponding row blocks into W_R.
      iii. All the processors save the jth block of the pivoting indices into W_ρ.
      iv. P_C saves the jth row and column blocks corresponding to the jth row and column into W_R and W_C, respectively, and save the jth block of pivoting indices into W_ρ.
      v. All P_A's perform factorization on the jth column block.
      vi. The processors owning any pivoting row save the pivoting row into W_P.
      vii. P_C saves the rows corresponding to the pivot rows into W_P.

viii. Update the remaining matrix by the rank-b update.

$$A \;\leftarrow\; A - L_j U_j$$

ix. Checkpoint the jth row and column blocks and the pivoting rows and indices. At this point, P_C maintains the checksums of all L_j's, U_j's, the pivoting rows, and indices for the current checkpointing interval.

(b) Checkpoint the remaining matrix at the last iteration of the current checkpointing interval.

Memory requirement for W_C, W_R, W_P, and W_ρ: Assuming that the checkpoint is taken every K of iterations, extra memory M_C is required for storing the jth column blocks for $i+1 \le j \le i+K$, which are distributed over Q columns of processors.

$$M_C \;=\; \lceil \frac{K}{Q} \rceil \left(\frac{n}{P} + 2\frac{n}{Q} + \frac{1}{2} \right) b,$$

Time complexity for checkpointing, T_C: In addition to the initial checkpointing, the jth row and column blocks are to be saved, and all L_j's, U_j's, and the jth pivoting rows and indices are to be checkpointed for every interval of K iterations, ,.e., $i+1 \le j \le i+K$. Finally, at the end of every checkpointing interval of K iterations, the remaining matrix is to be checkpointed. To simplify the analysis, we ignore the time overhead of copying.

$$\begin{aligned} T_C \;\approx\; & T_{matckpt}(n, n, P, Q) + \sum_{j=1}^{\frac{n}{Kb}} \left(T_{matckpt}(n - jKb, n - jKb, P, Q) \right) \\ & + \sum_{j=1}^{\frac{n}{b}} \left(T_{cbckpt}(n - jb + b, b, P, Q) + 2T_{rbckpt}(b, n, P, Q) + 2T_{cbckpt}(\frac{b}{2}, 1, 1, 1) \right). \end{aligned}$$

For large n the last term dominates the time overhead of checkpointing. Then, T_C is approximated by

$$T_C \;\approx\; \sum_{j=1}^{\frac{n}{Kb}} (n - jKb)^2 T_{ckpt}(P, Q) \approx n^3 \frac{1}{3Kb} T_{ckpt}(P, Q).$$

Recovery

Suppose that a failure occurs at the lth iteration of a checkpointing interval, where $i+1 \leq l \leq i+K$. Since several iterations have already been performed without checkpointing the remaining matrix, we need to recover all of the data at the last checkpoint including the old data of the updated trailing matrix. The following operations are performed for recovery. Let L_j and U_j represent the jth column and row blocks at iteration j after being updated during the current interval, respectively.

1. Reconfigure the processor grid.
2. Recover L_j's and U_j's for $i+1 \leq j \leq l$ from their corresponding checksum.
3. Recover their pivoting rows and indices for iteration j ($i+1 \leq j \leq l$) from their corresponding checksum.
4. Reverse the computation for the remaining matrix from iteration l to $i+1$ as follows:
   (a) $A \leftarrow A + L_j U_j$
   (b) Restore the jth pivoting rows and indices from their corresponding W_P and W_ρ.
5. Restore their column and row blocks from W_C and W_R.
6. Recover the failed processor's portion of the matrix from the checksum.
7. Resume the computation at iteration $i+1$.

Time complexity for recovery T_R: The time overhead of recovery varies depending on the iteration at which a failure occurs. In addition to the time for recovering the column and row blocks and the pivoting rows and indices modified since the last checkpoint, the time is required for recovering the failed processor's L_j and U_j and for performing up to K rank-b updates by reverse computation. Since the time overhead of recovery is dominated by the time for performing Steps 4 and 6 (i.e., the last two terms), T_R is approximated as follows.

$$\begin{aligned} T_R \approx\ & T_{barrier}(P,Q) \\ & + T_{matrecv}(n, Kb, P, Q) + 2T_{matrecv}(Kb, n, P, Q) + T_{matrecv}(\frac{Kb}{2}, 1, P, Q) \end{aligned}$$

$$
\begin{aligned}
&+\sum_{j=l}^{i+1} T_{rank}(n-jb, n-jb-b, b, P, Q) + T_{matrecv}(n, n, P, Q) \\
\approx\ & n^2 \frac{2Kb}{PQ}\gamma + n^2 T_{recv}(P, Q).
\end{aligned}
$$

Table 3.7 presents the complexity summary of the right-looking LU algorithm.

Table 3.7: Complexity summary of the right-looking LU algorithm

T_{LU}	$n^3 \frac{2}{3PQ}\gamma + O(n^2)$
T_C	$n^3 \frac{1}{3KbPQ}((\lceil \log P \rceil + \lceil \log Q \rceil)(\gamma+\beta)+\beta) + O(n^2)$
T_R	$n^2 \frac{1}{PQ}\left(2Kb\gamma + (\lceil \log P \rceil + \lceil \log Q \rceil)(\gamma+\beta)\right) + O(n)$
M_C	$\lceil \frac{K}{Q} \rceil \left(\frac{n}{P} + 2\frac{n}{Q}\right) b$

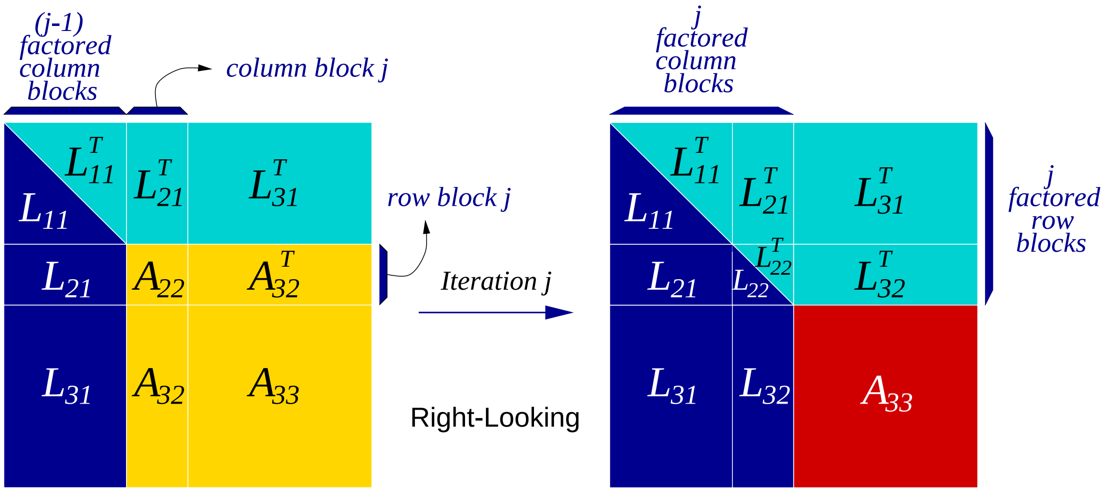
3.4.3 Right-Looking Cholesky Factorization

A Parallel Algorithm

At the beginning of iteration j in the right-looking algorithm, the leftmost $j-1$ column blocks and uppermost $j-1$ row blocks have been factored (Figure 3.12). At iteration j, the jth column block is factored and updated first and then the remaining matrix is updated by one rank-b update using the factored column block L_j.

Time complexity of the algorithm:

$$
\begin{aligned}
T_{LLT}(n, n, P, Q) &= \sum_{j=1}^{\frac{n}{b}} (T_{fact}(b, b, 1, 1) + T_{solve}(b, n-jb, 1, Q) \\
&\qquad + T_{rank}(n-jb, n-jb, b, P, Q)) \\
&\approx \frac{n^3}{3PQ}\gamma + O(n^2).
\end{aligned}
$$



I. Factor the jth column block

$$A_{22} = L_{22}L_{22}^T$$

II. Solve a triangular system.

$$L_{32}^T = L_{22}^{-1}A_{32}^T$$

III. Update the remaining matrix.

$$A_{33} \leftarrow A_{33} - L_{32}L_{32}^T$$

Figure 3.12: Iteration j of the right-looking Cholesky algorithm

Checkpointing

To checkpoint the right-looking Cholesky algorithm, assuming that each checkpoint is taken every K of iterations, first all the processors allocate memory W_C for $\lceil \frac{K}{Q} \rceil$ column blocks. W_C will be used to save the column blocks to be updated during a checkpointing interval. The current column block is saved and the factored current column L_j is checkpointed for each iteration. At the end of each interval of K iteration, the newly modified remaining part of A is checkpointed. The complete steps are explained below.

1. All the processors checkpoint the matrix A initially.
2. For every K of iterations,
   (a) For each iteration j $(i + 1 \leq j \leq i + K)$,
      i. The processors owning the jth column save their jth column block into W_C.
      ii. Perform factorization on the jth column block.

iii. Update the remaining matrix by the rank-b update.
iv. Checkpoint the jth column block modified at this iteration.

(b) Checkpoint the remaining matrix at iteration $i+K$.

Memory requirement for W_C: Memory is required for storing the K column blocks, which are distributed over Q columns of processors. Thus, M_C is given by

$$M_C = \lceil \frac{K}{Q} \rceil \left(\frac{n}{P}\right) b.$$

Time complexity for checkpointing, T_C: In addition to the checkpointing overhead of the top-looking Cholesky algorithm, the right-looking Cholesky algorithm requires checkpointing the updated trailing matrix. For large n, we assume that Step 2(b) dominates the time overhead of checkpointing. Then, T_C is given by

$$\begin{aligned} T_C &\approx \sum_{j=1}^{\frac{n}{Kb}} \frac{1}{2} T_{matckpt}(n - jKb, n - jKb, P, Q) \\ &\approx \sum_{j=1}^{\frac{n}{Kb}} \frac{1}{2}(n - jKb)^2 T_{ckpt}(P, Q) \\ &\approx n^3 \frac{1}{6Kb} T_{ckpt}(P, Q). \end{aligned}$$

Recovery

1. Reconfigure the processor grid.
2. Recover the failed processor's L_j from the checksum of the L_j's, if necessary. At this point all L_j's for $i+1 \le j \le l$ have been recovered.
3. Reverse the computation for the updated part of the matrix by performing the following operations.

$$A \;\leftarrow\; A + \sum_{j=l}^{i+1} L_j L_j^T.$$

4. Restore A_j's during the current checkpointing interval from W_C, if it has been updated.

5. Recover the failed processor's data of A from the checksum.

6. Resume the computation at iteration $i+1$.

Time complexity for recovery, T_R: In addition to the time for recovering the column blocks modified since the last checkpoint, time is required for recovering the failed processor's L_j and for performing up to K rank-b updates by reverse computation. The recovery overhead is dominated by the time for performing Steps 3 and 5.

$$\begin{aligned} T_R &\approx \sum_{j=l}^{i+1} T_{rank}(n-jb, n-jb, P, Q) + \frac{1}{2} T_{matrecv}(n, n, P, Q) \\ &\approx n^2 \frac{2Kb}{PQ}\gamma + \frac{1}{2} n^2 T_{recv}(P, Q). \end{aligned}$$

Table 3.8 presents the complexity summary of the right-looking Cholesky algorithm.

Table 3.8: Complexity summary of the right-looking Cholesky algorithm

T_{LLT}	$n^3 \frac{1}{3PQ}\gamma + O(n^2)$
T_C	$n^3 \frac{1}{6KbPQ}((\lceil \log P \rceil + \lceil \log Q \rceil)(\gamma+\beta)+\beta) + O(n^2)$
T_R	$n^2 \frac{1}{PQ}\left(2Kb\gamma + \frac{1}{2}(\lceil \log P \rceil + \lceil \log Q \rceil)(\gamma+\beta)\right) + O(n)$
M_C	$\lceil \frac{K}{Q} \rceil \left(\frac{n}{P}\right) b$

3.4.4 Right-Looking QR Factorization

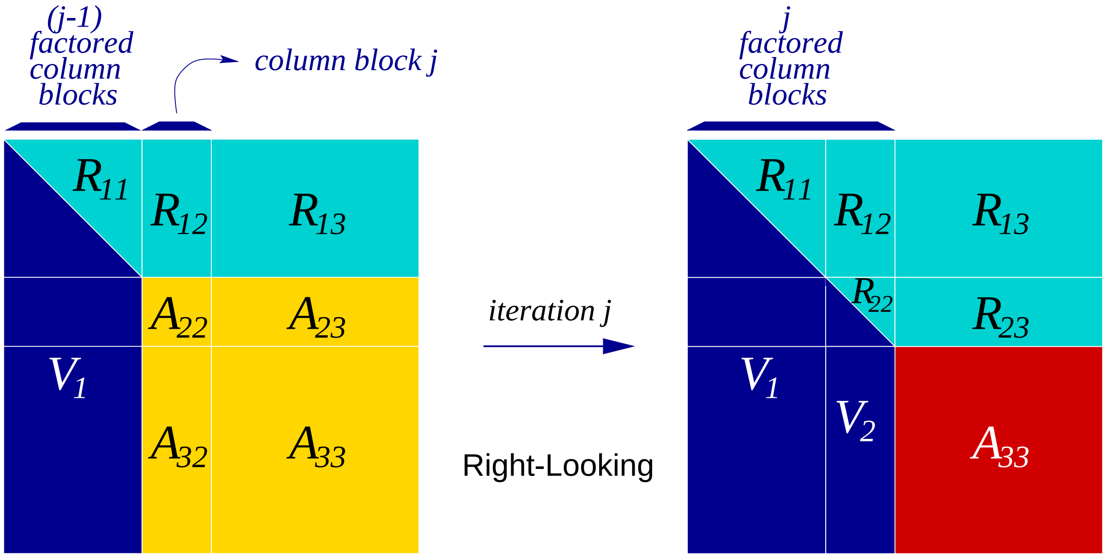
A Parallel Algorithm

In the right-looking QR algorithm, updating the remaining matrix is performed after the current column block is factored. As depicted in Figure 3.13, at each iteration j, the jth column block is factored and then the computation in Step II is performed to update the remaining matrix. To take advantage of matrix-matrix multiplication, the updating procedure is modified in the following form [DDSvdV91, CDO+96]. We may write the update to A performed at each iteration as

$$A \leftarrow (I - VTV^T)^T A = A - VT^T V^T A = A - VW^T,$$

where $W^T = T^T V^T A$.

Thus, in one implementation, Step II first computes T from a sequence of Householder vectors and τ_i's and then computes W_2^T. Finally it performs a rank-b update with V_2 and W_2^T.



I. Factor the j column block.

$$\begin{pmatrix} A_{22} \\ A_{32} \end{pmatrix} = Q_2 \begin{pmatrix} R_{22} \\ 0 \end{pmatrix},$$

where $Q_2 = I - V_2 T_2 V_2^T$

II. Update the trailing matrix.

$$\begin{aligned} \begin{pmatrix} R_{23} \\ A_{33} \end{pmatrix} &\leftarrow \left(I - V_2 T_2 V_2^T\right)^T \begin{pmatrix} A_{23} \\ A_{33} \end{pmatrix} \\ &= \begin{pmatrix} A_{23} \\ A_{33} \end{pmatrix} - V_2 T_2^T V_2^T \begin{pmatrix} A_{23} \\ A_{33} \end{pmatrix}. \end{aligned}$$

Figure 3.13: Iteration j of the right-looking QR algorithm

Time complexity for the algorithm:

$$T_{QR}(n, n, P, Q) = \sum_{j=0}^{N-1} (T_{fact}(n - jb, b, P, 1) + T_{pre}(n - jb, n - jb - b, P, Q)]$$

$$\approx \frac{4n^3}{3PQ}\gamma + O(n^2).$$

Checkpointing

In the right-looking QR algorithm, the updating procedure requires an intermediate computation of W_j^T for the jth iteration. Subsequently, the computation of W_j^T involves the remaining matrix, which is distributed over the processor grid. If a failure occurs when this update is being performed, the failed processor's portion of the remaining matrix can't be recovered because other processors' data has already been modified. Thus, this requires the intermediate result W_j^T to be saved and checkpointed for reverse computation. Checkpointing the algorithm can be described in the following steps, assuming that each checkpoint is taken every K of iterations and memory for W_C and W_W is allocated to store the column blocks and W_j^T's to be computed during a checkpointing interval.

1. All the processors checkpoint the matrix A.
2. For every K of iterations,
   (a) For each iteration j $(i+1 \leq j \leq i+K)$,
      i. The processors owning the jth column save their corresponding column blocks into W_C.
      ii. Perform factorization for the jth column block.
      iii. Compute T_j and the intermediate row vector W_j^T and store W_j^T into W_W.
      iv. Update the remaining matrix by performing

$$A \leftarrow A - V_j W_j^T$$

      v. Checkpoint V_j and W_j^T.
   (b) Checkpoint the remaining matrix at iteration $i+K$.

Memory requirement for W_C and W_W: Assuming that a checkpoint is taken every K of iterations, M_C is given

$$M_C = \lceil \frac{K}{Q} \rceil \left(\frac{n}{P} + \frac{n}{Q} \right) b.$$

Time complexity for checkpointing, T_C: In addition to initial checkpointing, the W_j^T's and A_j's are to be saved and checkpointed every iteration. At the end of every checkpointing interval of K iterations, the remaining matrix is to be checkpointed. Note that for large n, Step 2(b) dominates the checkpointing overhead.

$$\begin{aligned} T_C &= \sum_{j=1}^{\frac{n}{Kb}} T_{matckpt}(n - jKb, n - jKb, P, Q) \\ &\approx \sum_{j=1}^{\frac{n}{Kb}} (n - jKb)^2 T_{ckpt}(P, Q) \\ &\approx n^3 \frac{1}{3Kb} T_{ckpt}(P, Q). \end{aligned}$$

Recovery

1. Reconfigure the processor grid.
2. Recover the failed processor's W_j^T and V_j, if necessary. At this point, all W_j^T's and V_j's for $j = i+1, \ldots, l$ have been recovered.
3. Do the following computations for the updated part of the matrix.

$$A \leftarrow A + \sum_{j=i+1}^{l} V_j W_j^T.$$

4. Restore the jth column blocks from W_C for $i + 1 \leq j \leq i + K$.
5. Recover the failed processor's data from the checksum matrix.
6. Resume the computation at iteration $i + 1$.

Time complexity for recovery, T_R: Similarly, T_R is approximated by

$$\begin{aligned} T_R &\approx \sum_{j=l}^{i+1} T_{rank}(n - jb, n - jb, P, Q) + T_{matrecv}(n, n, P, Q) \\ &\approx n^2 \frac{2Kb}{PQ} \gamma + n^2 T_{recv}(P, Q). \end{aligned}$$

Table 3.9 presents the complexity summary of the right-looking QR algorithm.

Table 3.9: Complexity summary of the right-looking QR algorithm

T_{QR}	$n^3 \frac{4}{3PQ}\gamma + O(n^2)$
T_C	$n^3 \frac{1}{3KbPQ}((\lceil \log P \rceil + \lceil \log Q \rceil)(\gamma + \beta) + \beta) + O(n^2)$
T_R	$n^2 \frac{1}{PQ}\left(2Kb\gamma + (\lceil \log P \rceil + \lceil \log Q \rceil)(\gamma + \beta)\right) + O(n).$
M_C	$\lceil \frac{K}{Q} \rceil \left(\frac{n}{P} + \frac{n}{Q}\right) b$

3.4.5 Hessenberg Reduction

A Parallel Algorithm

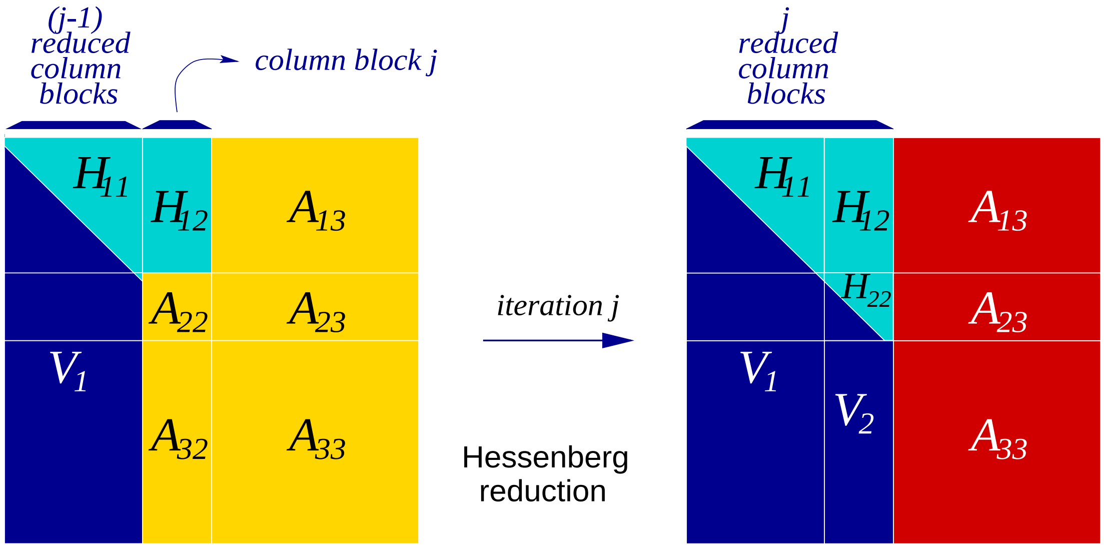
Hessenberg reduction reduces an $m \times n$ matrix A to an upper-Hessenberg matrix H using similarity transformation based on Householder reflectors. The algorithm can be formulated by

$$A = Q^T H Q,$$

where Q is an $m \times m$ orthogonal matrix and H an $m \times n$ upper-Hessenberg matrix.

The block algorithm proceeds by reducing each column block in each iteration. At each iteration j, the current column of processors reduces the jth column block to upper-Hessenberg form using Householder reflectors, and then a sequence of the Householder reflectors is applied to the remaining matrix by both pre- and post-multiplication (Figure 3.14). As with the right-looking QR algorithm, we may write the update (Step II) to A performed at an iteration of the block algorithm as

$$A \leftarrow (I - VTV^T)^T A (I - VTV^T) = (I - VT^T V^T)(A - YV^T),$$



I. Reduce the j column block.

$$\begin{pmatrix} A_{22} \\ A_{32} \end{pmatrix} = Q_2 \begin{pmatrix} H_{22} \\ 0 \end{pmatrix},$$

where $Q_2 = I - V_2 T_2 V_2^T$

II. Update the trailing matrix.

$$\begin{pmatrix} A_{13} \\ A_{23} \\ A_{33} \end{pmatrix} \leftarrow Q_2^T \begin{pmatrix} A_{13} \\ A_{23} \\ A_{33} \end{pmatrix} Q_2.$$

Figure 3.14: Iteration j of the Hessenberg reduction algorithm

where $Y = AVT$. Let $A \leftarrow A - YV^T$. Then, as with the update of the right-looking QR algorithm, it may be written by

$$A \leftarrow (I - VTV^T)^T A = A - VW^T,$$

where $W^T = T^T V^T A$. Based on this representation, Step II proceeds in the following five phases.

1. compute T.
2. compute $Y = AVT$.
3. do the rank-b update, $A \leftarrow A - YV^T$.
4. compute $W^T = T^T V^T A$.
5. do the rank-b update, $A \leftarrow A - VW^T$.

Time complexity of the algorithm:

$$
\begin{aligned}
T_{HRD}(n,n,P,Q) &= \sum_{j=1}^{\frac{n}{b}} (T_{fact}(n-jb+b,b,1,1) + T_{post}(n,n-jb,P,Q) \\
&\qquad + T_{pre}(n-jb+b,n-jb,P,Q)) \\
&\approx \frac{10n^3}{3PQ}\gamma + O(n^2).
\end{aligned}
$$

Checkpointing

Updating the remaining matrix by pre- and post-multiplication requires the intermediate computations $Y = AVT$ and $W^T = T^T V^T A$ as described in the previous section. Thus, for recovery, those intermediate results Y and W^T must be saved and checkpointed at each iteration. It is also assumed that a checkpoint is taken every K of iterations.

1. All the processors checkpoint the matrix A.
2. For K iterations,
   (a) For each iteration j $(i+1 \leq j \leq i+K)$,
      i. The processors owning the jth column save their corresponding column blocks into W_C.
      ii. Perform factorization for the jth column-block.
      iii. Update the remaining matrix by performing the following computation,

$$
A \;\leftarrow\; A - Y_j V_j^T - V_j W_j^T.
$$

      iv. Checkpoint Y_j and W_j^T.

   (b) Checkpoint the remaining matrix at iteration $i+K$.

Memory requirement for W_C, W_Y, and W_W: Extra memory is required for storing the column blocks to be factored within a checkpointing interval, and Y_j and W_j^T.

$$
M_C \;=\; \lceil \frac{K}{Q} \rceil \left(2\frac{n}{P} + \frac{n}{Q} \right) b.
$$

Time complexity for checkpointing, T_C:

$$
\begin{aligned}
T_C &\approx \sum_{j=1}^{\frac{n}{Kb}} T_{matckpt}(n, n - jKb, P, Q) \\
&\approx \sum_{j=1}^{\frac{n}{Kb}} n(n - jKb) T_{ckpt}(P, Q) \\
&\approx n^3 \frac{1}{2Kb} T_{ckpt}(P, Q).
\end{aligned}
$$

Recovery

1. Reconfigure the processor grid.
2. Recover the failed processor's V_j, Y_j, and W_j^T from their checksum, if necessary. At this point, all V_j's, Y_j's, and W_j^T's for $i+1 \leq j \leq l$ have been recovered.
3. Do the following computations for the updated part of the matrix.

$$
A \;\leftarrow\; A + \sum_{j=i+1}^{l} \left(V_j W_j^T + Y_j V_j^T \right).
$$

4. Restore the jth column blocks from W_C for $i+1 \leq j \leq l$
5. Recover the failed processor's data from the checksum matrix.
6. Resume the computation at iteration $i+1$.

Time complexity for recovery, T_R:

$$
\begin{aligned}
T_R &= \sum_{j=l}^{i+1} \left(T_{rank}(n - jb, n - jb - b, b, P, Q) + T_{rank}(n, n - jb - b, b, P, Q) \right) \\
&\qquad + T_{matrecv}(n, n, P, Q) \\
&\approx n^2 \frac{4Kb}{PQ} \gamma + n^2 T_{recv}(P, Q).
\end{aligned}
$$

Table 3.10 presents the complexity summary of the Hessenberg reduction algorithm.

Table 3.10: Complexity summary of the Hessenberg reduction algorithm

T_{HRD}	$n^3 \frac{10}{3PQ}\gamma + O(n^2)$
T_C	$n^3 \frac{1}{2KbPQ}((\lceil \log P \rceil + \lceil \log Q \rceil)(\gamma + \beta) + \beta) + O(n^2)$
T_R	$n^2 \frac{1}{PQ}\left(4Kb\gamma + (\lceil \log P \rceil + \lceil \log Q \rceil)(\gamma + \beta)\right) + O(n)$
M_C	$\lceil \frac{K}{Q} \rceil \left(2\frac{n}{P} + \frac{n}{Q}\right) b$

3.4.6 Summary

We have shown that the checksum and reverse computation technique is a feasible alternative to provide fault tolerance for matrix operations in which the checkpoint size is too large to save in extra memory space. Though the underlying idea of this technique is similar to that of the parity-based technique, it requires checkpointing a large amount of data. Hence, the checkpointing overhead may increase significantly compared with that for the parity-based matrix operations. As shown in the analytic models of various matrix operations, while the failure-free algorithms of matrix operations require $O(n^3)$ floating-point operations for a matrix of size n, the checkpointing steps require $O(\frac{n^3}{Kb})$ operations. Nevertheless, the technique provides the flexibility of selecting the checkpointing interval K to tune the overhead. The complexity analyses also show that there is a tradeoff between the extra memory requirement and the checkpointing overhead. In order to reduce the overhead, the checkpointing interval must get larger, and hence more memory is required to store the data (i.e., column blocks or row blocks) updated during each checkpointing interval.

It is well known that in the realm of floating-point computations no computation is exact [Wil63]. While the parity-based technique does not involve any floating-point computations for providing fault tolerance for numerical algorithms, the technique based on the checksum and reverse computation does involve both floating-point additions and matrix computations. Thus, there is a possibility of numerical problems in floating-point arithmetic subject to the checksum as well as reverse computation.

Some numerical problems do arise that are related to the checksum encoding. First, overflow and underflow might occur if the checksum is too large or small. However, since the checksum is computed over the data of the total number of processors, the possibility that overflow and underflow occur in computing the checksum is very low and usually considered to be negligible unless an element of the matrix is too big or too small. The overflow and underflow in floating-point arithmetic can be avoided just by organizing the computations differently, for example, normalizing each number by dividing it by the maximum among the numbers [Wil63, FMM77]. Second, roundoff errors from cancellation can be a serious numerical problem. It is well known that cancellation usually occurs when two numbers of approximately the same size are subtracted. Fortunately, the cancellation subject to the checksum can usually be avoided by performing the summation of the floating-point numbers in different order.

In the next section, we give a complete description of error analysis for the fault-tolerant algorithms based on checksum and reverse computation. In this analysis, it turns out that reverse computations neither introduce any significant roundoff errors nor affect the stability of the original algorithm.

3.5 Error Analysis

In this section, we outline the derivation of the error bound of each algorithm in the presence of failures and hence demonstrate how the roundoff error due to reverse computation may affect the stability of each matrix algorithm. Assuming that a checkpoint is taken every iteration, we only consider the effect of the roundoff errors made within a checkpointing interval of one iteration. The derivation presented can be easily extended to a checkpointing interval of multiple iterations.

3.5.1 Matrix Reductions

We first consider the LU, QR, and Cholesky factorizations and Hessenberg reduction. In general, we denote A_k as the matrix at the beginning of iteration k and T_k as the transformation computed at iteration k of each algorithm. Iteration k in the presence of a failure requires the following steps: (1) $A_{k+1} = T_k A_k$. Note that this step is equivalent to an iteration of the algorithm without any failure. (2) Assuming that a failure occurs at the end of iteration k, reverse computation involves the computation of $A_k = T_k^{-1} A_{k+1}$. (3) The algorithm repeats the iteration k by performing $A_{k+1} = T_k A_k$. Throughout this analysis we define $\tilde{A}$ as the computed matrix of A, i.e. $\tilde{A} = fl(A)$. Note that $A_k, A_{k+1}, T_k \in R^{n \times n}$.

In general, we consider the transformation $C = TA$ and let $\tilde{C} = fl(C) = C + E$. T is the transformation matrix generated in a matrix reduction algorithm. It can readily be shown [Hig96] that

$$||E|| \leq nu||T||||A||, \tag{3.19}$$

where u is the machine epsilon, otherwise known as the unit roundoff error.

Theorem 1 *Consider the computation $A_{k+1} = T_k(T_k^{-1}(T_k A_k))$, which proceeds in the following subsequent steps.*

$$\begin{aligned} A_{k+1} &= T_k A_k & (3.20)\\ A_k &= T_k^{-1} A_{k+1} & (3.21)\\ A_{k+1} &= T_k A_k. & (3.22) \end{aligned}$$

Then, the computed matrix of A_{k+1} in the presence of a failure satisfies

$$\tilde{A}_{k+1} = T_k(A_k + F) \tag{3.23}$$

so that the error is bounded with

$$||F|| \leq cnu||T_k^{-1}||||T_k||||A_k||, \tag{3.24}$$

where c is a constant.

Proof: First, from Equation 3.20, the computed matrix of A_{k+1} is given by

$$\begin{aligned}\tilde{A}_{k+1} &= fl(A_{k+1}) \\ &= A_{k+1} + E_1 \\ &= T_k A_k + E_1. \end{aligned} \tag{3.25}$$

Here, using Equation 3.19 the error E_1 satisfies

$$\|E_1\| \leq nu\|T_k\|\|A_k\|. \tag{3.26}$$

Second, from Equation 3.21 the computed matrix of A_k is given by

$$\begin{aligned}\tilde{A}_k &= fl(A_k) \\ &= A_k + E_2 \\ &= T_k^{-1}\tilde{A}_{k+1} + E_2 \\ &= T_k^{-1}(A_{k+1} + E_1) + E_2 \\ &= A_k + T_k^{-1}E_1 + E_2. \end{aligned} \tag{3.27}$$

The error E_2 also satisfies

$$\begin{aligned}\|E_2\| &\leq nu\|T_k^{-1}\|\|A_{k+1}\| \\ &\leq nu\|T_k^{-1}\|\|T_k\|\|A_k\|. \end{aligned} \tag{3.28}$$

Third, from Equation 3.21 the computed matrix of A_{k+1} is given by

$$\begin{aligned}\tilde{A}_{k+1} &= fl(A_{k+1}) \\ &= A_{k+1} + E_3 \\ &= T_k\tilde{A}_k + E_3 \\ &= T_k(A_k + T_k^{-1}E_1 + E_2) + E_3 \end{aligned}$$

$$
\begin{aligned}
&= T_k(A_k + T_k^{-1}E_1 + E_2 + T_k^{-1}E_3) \\
&= T_k(A_k + F).
\end{aligned}
$$

so that $F = T_k^{-1}E_1 + E_2 + T_k^{-1}E_3$, where

$$
\|E_3\| \leq nu\|T_k\|\|A_k\|. \tag{3.29}
$$

Hence, the error F is bounded with

$$
\begin{aligned}
\|F\| &\leq \|T_k^{-1}\|\|E_1\| + \|E_2\| + \|T_k^{-1}\|\|E_3\| \\
&\leq cnu\|T_k^{-1}\|\|T_k\|\|A_k\|. \ \Box
\end{aligned}
$$

From Equation 3.24, we draw the following conclusions on the stability of each fault-tolerant algorithm in the presence of failures.

1. LU and Cholesky factorizations: T_k and A_k of the computation in presence of a failure are equivalent to those of the failure-free computation. If the failure-free algorithm is stable, then its fault-tolerant algorithm is also stable.

2. QR factorization and Hessenberg reduction: Since $\|T_k\| = \|T_k^{-1}\| = 1$ for an orthogonal transformation T_k, the fault-tolerant algorithm in the presence of failures is always as stable as the failure-free algorithm.

In summary, for the LU, QR, and Cholesky factorizations and Hessenberg reduction, reverse computations performed in the presence of failures do not introduce any significant roundoff errors.

3.5.2 Matrix Multiplication

We now consider the fault-tolerant matrix multiplication algorithm $C = AB$. We denote C_k as the matrix at the beginning of iteration k. Assuming that a failure occurs at the end of iteration k, an iteration of the fault-tolerant algorithm in presence of a failure consists of the following three steps.

$$\begin{aligned} C^{(k+1)} &= C^{(k)} + xy^T & (3.30) \\ C^{(k)} &= C^{(k+1)} - xy^T & (3.31) \\ C^{(k+1)} &= C^{(k)} + xy^T, & (3.32) \end{aligned}$$

where $C^{(k)}, C^{(k+1)} \in R^{m\times n}$, $x \in R^{m\times 1}$, and $y \in R^{1\times n}$.

Theorem 1 *Consider the computation $C^{(k+1)} = ((C^{(k)}+xy^T)-xy^T)+xy^T$, which proceeds in Equations 3.30 through 3.32. Then, the computation is as stable as the failure-free computation of $C^{(k+1)} = C^{(k)} + xy^T$.*

Proof: First, consider the computation of Equation 3.30 and let $c_{ij}^{(k)}$ be an element of the matrix $C^{(k)}$ at the beginning of iteration k. Let $\tilde{c}_{ij}^{(k+1)}$ be a computed element of $C^{(k+1)}$. $\tilde{c}_{ij}^{(k+1)}$ is approximated by

$$\begin{aligned} \tilde{c}_{ij}^{(k+1)} &= fl(c_{ij}^{(k+1)}) \\ &= c_{ij}^{(k)}(1+\delta_1) + x_i y_j (1+\delta_2) \qquad (|\delta_i| \leq 2u) \\ &= c_{ij}^{(k)} + x_i y_j + c_{ij}^{(k)}\delta_1 + x_i y_j \delta_2 \\ &= c_{ij}^{(k+1)} + c_{ij}^{(k)}\delta_1 + x_i y_j \delta_2. & (3.33) \end{aligned}$$

Then,

$$\begin{aligned} |\tilde{c}_{ij}^{(k+1)} - c_{ij}^{(k+1)}| &\leq |c_{ij}^{(k)}||\delta_1| + |x_i y_j||\delta_2| \\ &\leq 2u(|c_{ij}^{(k)}| + |x_i||y_j|). & (3.34) \end{aligned}$$

Therefore,

$$\|\tilde{C}^{(k+1)} - C^{(k+1)}\| \leq c_1 n u (\|C^{(k)}\| + \|x\|\|y\|). \qquad (3.35)$$

Second, consider the computation of Equation 3.31.

$$\begin{aligned}\tilde{c}_{ij}^{(k)} &= fl(c_{ij}^{(k)}) \\ &= \tilde{c}_{ij}^{(k+1)}(1+\gamma_1) - x_i y_j (1+\gamma_2) \qquad (|\gamma_i| \le 2u)\end{aligned}$$

Using Equation 3.33,

$$\begin{aligned}\tilde{c}_{ij}^{(k)} &= (c_{ij}^{(k+1)} + c_{ij}^{(k)}\delta_1 + x_i y_j \delta_2)(1+\gamma_1) - x_i y_j (1+\gamma_2) \\ &\approx c_{ij}^{(k+1)} - x_i y_j + c_{ij}^{(k)}\delta_1 + c_{ij}^{(k+1)}\gamma_1 + x_i y_j \delta_2 - x_i y_j \gamma_2 \\ &= c_{ij}^{(k)} + c_{ij}^{(k)}(\delta_1+\gamma_1) + x_i y_j(\gamma_1 - \gamma_2 + \delta_2). \qquad (3.36)\end{aligned}$$

Then,

$$\begin{aligned}|\tilde{c}_{ij}^{(k)} - c_{ij}^{(k)}| &\le |c_{ij}^{(k)}|(|\delta_1| + |\gamma_1|) + |x_i y_j|(|\gamma_1| + |\gamma_2| + |\delta_2|) \\ &\le c_2 u(|c_{ij}^{(k)}| + |x_i||y_j|). \qquad (3.37)\end{aligned}$$

Therefore,

$$\|\tilde{C}^{(k)} - C^{(k)}\| \le c_3 n u(\|C^{(k)}\| + \|x\|\|y\|). \qquad (3.38)$$

Third, consider the computation of Equation 3.32.

$$\begin{aligned}\tilde{c}_{ij}^{(k+1)} &= fl(c_{ij}^{(k+1)}) \\ &= \tilde{c}_{ij}^{(k+1)}(1+\beta_1) + x_i y_j (1+\beta_2). \qquad (|\beta_i| \le 2u)\end{aligned}$$

Using Equation 3.36,

$$\begin{aligned}\tilde{c}_{ij}^{(k+1)} &= (c_{ij}^{(k)} + c_{ij}^{(k)}(\delta_1+\gamma_1) + x_i y_j(\gamma_1 - \gamma_2 + \delta_2))(1+\beta_1) + x_i y_j(1+\beta_2) \\ &\approx c_{ij}^{(k)} + x_i y_j + c_{ij}^{(k)}(\delta_1 + \gamma_1 + \beta_1) + x_i y_j(\gamma_1 - \gamma_2 + \delta_2 + \beta_2) \\ &\approx c_{ij}^{(k+1)} + c_{ij}^{(k)}(\delta_1 + \gamma_1 + \beta_1) + x_i y_j(\gamma_1 - \gamma_2 + \delta_2 + \beta_2). \qquad (3.39)\end{aligned}$$

Then,

$$|\tilde{c}_{ij}^{(k+1)} - c_{ij}^{(k+1)}| \quad \leq \quad c_4 u(|c_{ij}^{(k)}| + |x_i||y_j|). \tag{3.40}$$

Therefore,

$$\|\tilde{C}^{(k+1)} - C^{(k+1)}\| \quad \leq \quad c_5 nu(\|C^{(k)}\| + \|x\|\|y\|). \tag{3.41}$$

The error bounds expressed in Equations 3.35 and 3.41 are both tight. Hence, the fault-tolerant algorithm in the presence of failures is as stable as the failure-free algorithm. □

In summary, for matrix multiplication reverse computations do not introduce any significant roundoff errors.

3.6 Multiple Checkpointing Technique

As discussed in Section 2.5.3, a simple scheme for tolerating multiple failures with multiple checkpointing processors is to employ *one-dimensional parity*. This section describes how the multiple checkpointing technique based on one-dimensional parity can be incorporated into each different algorithm.

For the one-dimensional parity scheme, we assume that one checkpointing processor is dedicated to each column of a $P \times Q$ processor grid. Note that such a scheme allows the program to tolerate Q simultaneous failures as long as failures occur in different groups, for example, columns of processors. With multiple checkpointing processors, each column of processors, including its dedicated checkpointing processor, cooperates to checkpoint its part of the matrix independently from the other columns of processors. Since the checkpointing and recovery can be distributed into groups of processors (i.e., columns of processors), the overhead of both checkpointing and recovery can be reduced. In addition, when the checksum is used, it reduces the possibility of overflow, underflow, and cancellation because fewer processors are involved in each checksum. Note that this technique can be used together with any of the fault tolerance techniques presented in this chapter.

Chapter 4

Performance Evaluation

This chapter presents experimental results and an empirical performance evaluation of the fault-tolerant implementations of the matrix operations discussed in Chapter 3. The implementations presented in this chapter were developed in the PVM programming environment and run on a network of workstations. In this chapter we compare the performance of two different fault tolerance techniques and discuss the performance of a multiple checkpointing technique.

4.1 Implementations

The fault-tolerant matrix operations described in the preceding chapter were implemented in the PVM programming environment [GBD+94]. The failure-free versions of the matrix operations were implemented as part of ScaLAPACK [CDO+96]. The ScaLAPACK library includes core matrix operations developed by using two subroutine libraries: the PB-BLAS and the BLACS. The PB-BLAS (Parallel Block-Based Linear Algebra Subprograms) [CDW96] are an extended subset of the Level 2 and 3 BLAS [DCHD90] for distributed-memory computers. The BLACS (Basic Linear Algebra Communication Subprograms) [DW95] perform common matrix-oriented communication tasks. Both of these subroutine packages are also implemented in the PVM programming environment.

The fault-tolerant implementations were designed to provide the user with the same efficiency, high performance, and portability as the ScaLAPACK routines. We first developed the overall structure of the fault-tolerant versions of the underlying libraries by embedding failure detection and identification using the fault-tolerant features of PVM [GBD+94]. All aspects of fault tolerance were kept internal to the applications and as transparent to the programmer as possible. The structure of these implementations is depicted in Figure 4.1. The currently available fault-tolerant implementations are listed in Table 4.1. A fault-tolerant PCG iterative solver has been developed based on the structure.

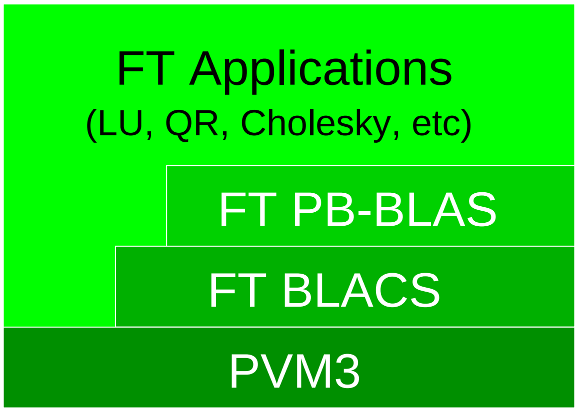


Figure 4.1: Structure of the fault-tolerant implementations

These routines were run on a network of 24 Sun Sparc-5 workstations connected by a fast, switched Ethernet network. The peak bandwidth of communication between two workstations in this network is 40 Mbits per second. Each workstation has 96 Mbytes of RAM. These experiments were performed with the network allocated for our exclusive use with no other user processes running.

For all of the implementations, the following set of tests was performed and timed:

- Failure-free algorithm without checkpointing.
- Fault-tolerant implementation with single checkpointing.
- Single checkpointing implementation with one random failure.

Table 4.1: Current fault-tolerant implementations of matrix operations

	Parity	Checksum & Reverse Computation	Multiple Checkpointing
LU	Crout, Left	Right	Crout, Left, Right
LLT	Left	Right	Left, Right
QR	Left	Right	Left, Right
HRD	—	HRD	HRD
MM	—	MM	MM
PCG	PCG	—	PCG

- Fault-tolerant implementation with multiple checkpointing.
- Multiple checkpointing implementation with multiple failures.

Note that the failures were forced to occur at the last iteration before the first checkpoint. The performance results of the implementations are evaluated in terms of the following parameters.

- Total elapsed wall-clock times of the algorithms in seconds (T_A, T).
- Checkpointing and recovery overheads in seconds (T_C, T_R).
- Checkpointing interval in iterations (K, $N_C = \frac{n}{Kb}$).
- Average checkpointing interval in seconds ($\frac{T - T_{init}}{N_C}$).
- Average checkpointing overhead in seconds (ΔT_C).
- Total size of checkpoints in bytes (M).
- Extra memory usage in bytes (M_C).
- Checkpointing rate in bytes per second (R).

4.2 Experimental Results

The single checkpointing version of each implementation was run on 17 processors, with 16 application processors logically configured into a 4×4 processor grid and one checkpointing processor. The multiple checkpointing version of each implementation was run on 20 processors, with a 4×4 processor grid and four checkpointing processors. The block size for all implementations was set at 50, and all implementations were developed for 64-bit floating-point arithmetic.

For the PCG iterative solver, a matrix derived from the five-point finite-difference discretization with grid size of $1,024$ of a partial differential equation (PDE) was tested up to $5,000$ iterations. The stopping criteria used was a residual tolerance of 10^{-10}. Both left and right preconditioners were applied for the PDE problem.

The matrix A is stored in compressed diagonal storage format [BBC$^+$95] as a $5 \times n$ matrix. The preconditioners are diagonal matrices stored as vectors of size $n = (1024)^2$. The checkpointing overhead therefore consists of the time to checkpoint A, M_1, M_2, and b at the beginning of the algorithm (a total of $8 \times n$ elements) and the time to checkpoint the intermediate vectors every K iterations (a total of $5 \times n$ elements) which is given by

$$\begin{aligned} T_C &= T_{matckpt}(8, n, P, Q) + KT_{matckpt}(5, n, P, Q) \\ &\approx (8 + 5K)nT_{ckpt}(P, Q). \end{aligned}$$

Note that K is the checkpointing interval in iterations.

Figures 4.2 through 4.6 present the experimental results of the parity-based implementations. Figures 4.7 through 4.11 present the experimental results of the checksum and reverse computation-based implementations. Considering network instability, we ran each implementation at three different times. The experimental results shown are obtained from the best one of the three different runs.

In those figures, the fifth column represents the average checkpointing interval in seconds and the eighth column represents the average time overhead of each checkpoint. Note that

T_C includes the initial checkpointing overhead T_{init}. Finally, K is the checkpointing interval in iterations and is chosen differently for each implementation to keep the checkpointing overhead small.

4.3 Performance Evaluation

The checkpointing performed in these implementations consists of data communication and either `XOR` or addition of floating-point numbers. We define the checkpointing rate R as the amount of data checkpointed in bytes per second. This metric has been used to evaluate the performance of various checkpointing schemes [EJZ92, PL94c]. In our case, the checkpointing rate is determined experimentally based on our analytic models of the fault-tolerant implementations. This checkpointing rate is used to compare the performance of the different fault tolerance techniques. Figures 4.12 and 4.13 plot the checkpointing rate for each implementation.

4.3.1 Parity-Based Technique

1. For the parity-based matrix operations, the total percentage overhead of checkpointing decreases as the problem size n increases.

2. The total overhead of recovery is dominated by the time for taking the bitwise exclusive-or of each processor's entire data. The time it takes to recover does not depend upon the location of the failure.

3. The multiple checkpointing implementations show performance improvement. LU factorizations benefit relatively more from the multiple checkpointing because of pivoting.

4. Figure 4.12 shows the checkpointing rate experimentally determined for each implementation. This presents the overall performance of the parity-based technique for matrix operations.

4.3.2 Checksum and Reverse Computation-Based Technique

1. The total checkpointing overhead can be kept small by expanding the checkpointing interval K or block size b. As K or b increases, more memory is required and the total recovery overhead increases. The selection of b affects the performance of the failure-free matrix operations. We suggest that K be increased instead of b.

2. The total recovery overhead depends partially upon the location of the failure.

3. Multiple checkpointing significantly reduces the overhead of checkpointing and recovery.

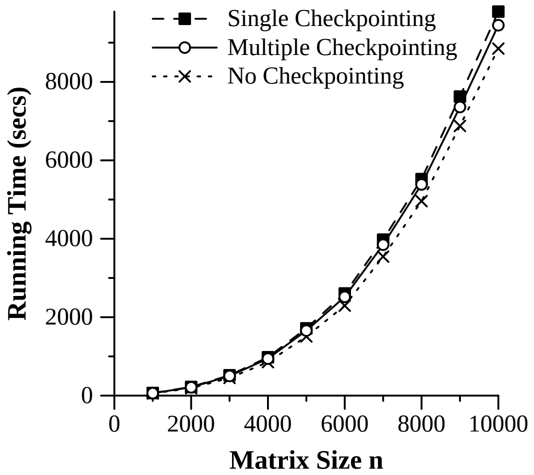
4. Good performance can be achieved with a moderate amount of extra memory for the right-looking QR algorithm and Hessenberg reduction.

5. For Hessenberg reduction (and small values of n), the extra memory used for checkpointing significantly improves the performance of the failure-free algorithm by saving some communication and computation. The values of checkpointing overhead and checkpointing rate reflect this fact.

6. Matrix multiplication is the simplest algorithm into which we incorporate the technique, and good performance is obtained without any memory overhead.

n	T_A (sec)	N_C +1	T (sec)	$\frac{T-T_{init}}{N_C}$ (sec)	T_C (sec)	%	T_{init} (sec)	$\frac{\Delta T_C}{N_C}$ (sec)	T_R (sec)
		With Single Checkpointing							
1000	56	21	66	3.1	10	17.9	3	0.3	3
2000	190	41	221	5.4	31	16.3	12	0.5	11
3000	451	61	519	8.5	68	15.1	25	0.7	24
4000	853	81	978	12.1	125	14.7	50	0.9	44
5000	1509	101	1715	17.0	206	13.7	79	1.3	69
6000	2294	121	2605	21.5	311	13.6	104	1.7	99
7000	3545	141	3978	28.2	433	12.2	148	2.0	136
8000	4959	161	5521	34.3	562	11.3	195	2.3	178
9000	6877	181	7627	42.1	750	10.9	253	2.7	248
10000	8850	201	9793	48.7	943	10.7	295	3.2	283
		With Multiple Checkpointing							
1000	56	21	63	3.0	7	12.5	2	0.2	2
2000	190	41	212	5.2	22	11.6	9	0.3	8
3000	451	61	501	8.2	50	11.1	21	0.5	19
4000	853	81	944	11.7	91	10.7	34	0.7	32
5000	1509	101	1658	16.4	149	9.9	53	1.0	54
6000	2294	121	2519	20.8	225	9.8	88	1.1	77
7000	3545	141	3849	27.3	304	8.6	114	1.3	104
8000	4959	161	5385	33.4	426	8.6	148	1.7	139
9000	6877	181	7360	40.7	483	7.0	155	1.8	153
10000	8850	201	9445	47.0	595	6.7	188	2.0	181

Checkpointing Interval $K = 1$, $\Delta T_C = T_C - T_{init}$, $T = T_A + T_C$



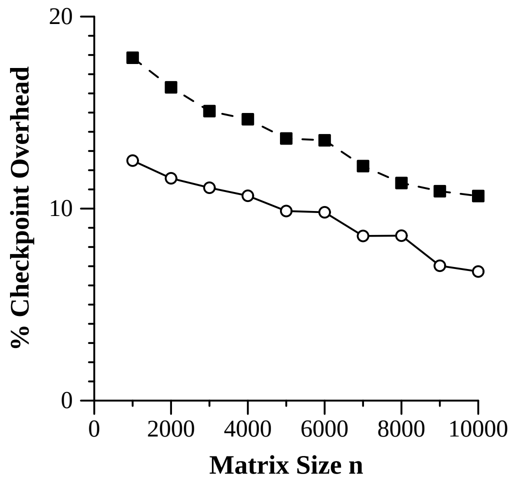
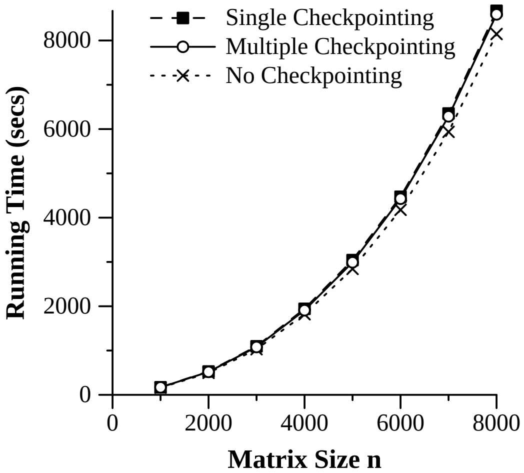


Figure 4.2: Crout LU: timing results

n	T_A (sec)	With Single Checkpointing N_C +1	T (sec)	$\frac{T-T_{init}}{N_C}$ (sec)	T_C (sec)	%	T_{init} (sec)	$\frac{\Delta T_C}{N_C}$ (sec)	T_R (sec)
1000	162	21	171	8.1	9	5.6	3	0.3	3
2000	496	41	527	12.9	31	6.2	12	0.5	11
3000	1030	61	1096	18.0	66	6.4	27	0.6	25
4000	1820	81	1942	24.0	122	6.7	49	0.9	45
5000	2843	101	3042	30.1	199	7.0	78	1.2	70
6000	4177	121	4472	37.0	295	7.1	111	1.5	100
7000	5937	141	6352	45.0	415	7.0	153	1.9	136
8000	8146	161	8670	53.9	524	6.4	200	2.0	178
		With Multiple Checkpointing							
1000	162	21	169	8.0	7	4.3	2	0.2	2
2000	496	41	519	12.7	23	4.6	8	0.4	7
3000	1030	61	1080	17.7	50	4.9	19	0.5	17
4000	1820	81	1910	23.6	90	4.9	34	0.7	29
5000	2843	101	2992	29.6	149	5.2	54	0.9	45
6000	4177	121	4428	36.6	251	6.0	87	1.4	78
7000	5937	141	6288	44.6	351	5.9	114	1.7	104
8000	8146	161	8590	53.4	444	5.5	149	1.8	138

Checkpointing Interval $K = 1$, $\Delta T_C = T_C - T_{init}$, $T = T_A + T_C$



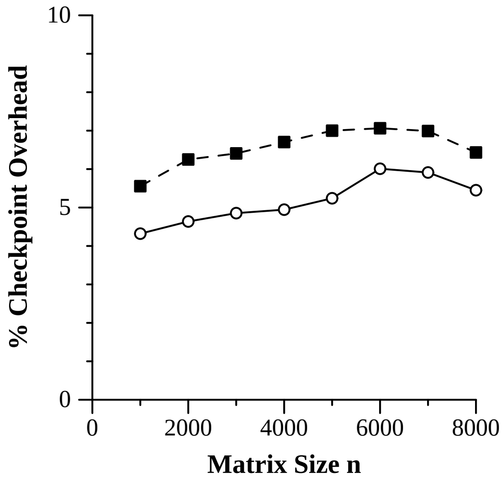


Figure 4.3: Left-looking LU: timing results

n	T_A (sec)	N_C +1	T (sec)	$\frac{T-T_{init}}{N_C}$ (sec)	T_C (sec)	T_C %	T_{init} (sec)	$\frac{\Delta T_C}{N_C}$ (sec)	T_R (sec)
		With Single Checkpointing							
1000	12	21	15	0.7	3	25.0	2	0.0	2
2000	58	41	70	1.7	12	20.7	6	0.1	6
3000	163	61	189	3.1	26	16.0	14	0.2	13
4000	331	81	379	4.7	48	14.5	26	0.3	23
5000	602	101	682	6.8	80	13.3	38	0.4	36
6000	992	121	1112	9.2	120	12.1	52	0.6	51
7000	1516	141	1681	11.9	165	10.9	70	0.7	69
8000	2244	161	2494	15.5	250	11.1	102	0.9	98
9000	3057	181	3367	18.6	310	10.1	134	1.0	132
10000	4112	201	4489	22.3	377	9.2	152	1.1	148
		With Multiple Checkpointing							
1000	12	21	14	0.7	2	16.7	1	0.0	1
2000	58	41	69	1.7	11	19.0	4	0.2	3
3000	163	61	182	3.0	19	11.7	8	0.2	8
4000	331	81	379	4.7	48	14.5	15	0.4	14
5000	602	101	668	6.6	66	11.0	24	0.4	24
6000	992	121	1097	9.1	105	10.6	35	0.6	32
7000	1516	141	1660	11.8	144	9.5	49	0.7	46
8000	2244	161	2457	15.3	213	9.5	64	0.9	62
9000	3057	181	3317	18.3	260	8.5	86	1.0	82
10000	4112	201	4446	22.1	334	8.1	105	1.1	102

Checkpointing Interval $K = 1$, $\Delta T_C = T_C - T_{init}$, $T = T_A + T_C$

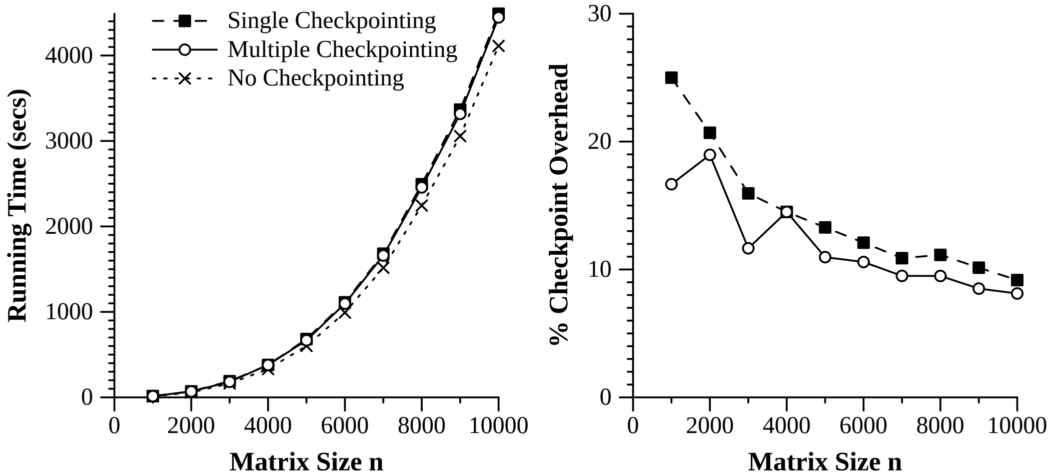
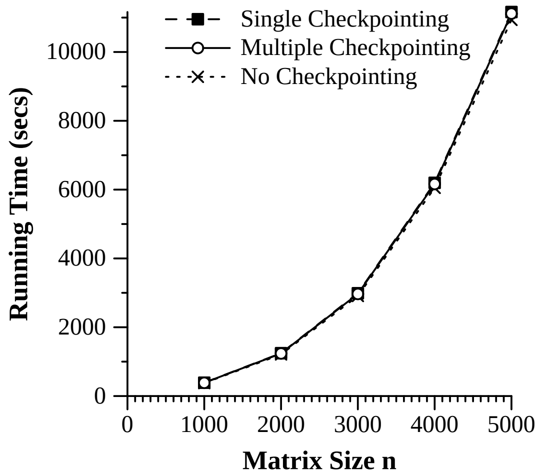


Figure 4.4: Left-looking Cholesky: timing results

n	T_A (sec)	N_C +1	T (sec)	$\frac{T-T_{init}}{N_C}$ (sec)	T_C (sec)	%	T_{init} (sec)	$\frac{\Delta T_C}{N_C}$ (sec)	T_R (sec)
		With Single Checkpointing							
1000	376	21	388	18.5	12	3.2	5	0.3	4
2000	1210	41	1246	30.4	36	3.0	14	0.5	12
3000	2909	61	2988	49.0	79	2.7	32	0.8	29
4000	6052	81	6197	76.5	145	2.4	55	1.1	52
5000	10934	101	11162	110.5	228	2.1	74	1.5	70
		With Multiple Checkpointing							
1000	376	21	385	18.3	9	2.4	3	0.3	3
2000	1210	41	1238	30.2	28	2.3	9	0.5	8
3000	2909	61	2968	48.7	59	2.0	21	0.6	19
4000	6052	81	6163	76.1	111	1.8	37	0.9	34
5000	10934	101	11117	110.1	183	1.7	55	1.3	46

Checkpointing Interval $K = 1$, $\Delta T_C = T_C - T_{init}$, $T = T_A + T_C$



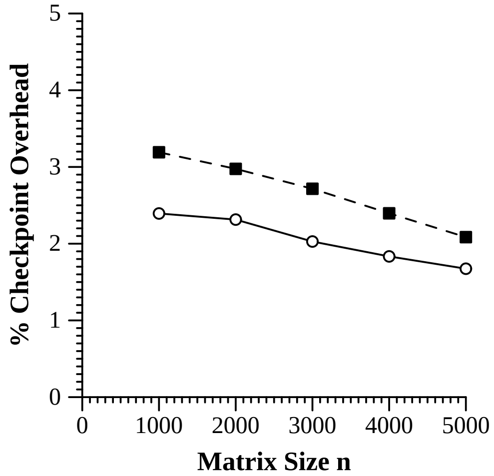


Figure 4.5: Left-looking QR: timing results

K	T_A (sec)	N_C +1	T (sec)	$\frac{T-T_{init}}{N_C}$ (sec)	T_C (sec)	T_C %	T_{init} (sec)	$\frac{\Delta T_C}{N_C}$ (sec)	T_R (sec)
		With Single Checkpointing							
10	5789	500+1	14742	29	8953	154.7	28.6	17.8	40
25	5789	200+1	9436	47	3647	63.0	28.9	18.1	41
50	5789	100+1	7607	76	1818	31.4	28.6	17.9	40
100	5789	50+1	6746	134	957	16.5	29.7	18.5	41
500	5789	10+1	6004	600	215	3.7	29.7	18.5	40
1000	5789	5+1	5909	1181	120	2.1	29.1	18.2	41
2500	5789	2+1	5853	2926	64	1.1	28.4	17.8	41
5000	5789	1+1	5835	5835	46	0.8	28.3	17.7	42
		With Multiple Checkpointing							
10	5789	500+1	11504	23	5715	98.7	18.2	11.4	23
25	5789	200+1	8123	40	2334	40.3	18.5	11.6	23
50	5789	100+1	7014	70	1225	21.2	19.3	12.1	23
100	5789	50+1	6397	127	608	10.5	18.9	11.8	23
500	5789	10+1	5919	591	130	2.2	17.9	11.2	23
1000	5789	5+1	5863	1172	74	1.3	17.9	11.2	23
2500	5789	2+1	5828	2914	39	0.7	17.3	10.8	23
5000	5789	1+1	5817	5817	28	0.5	17.2	10.8	23

$$T = T_A + T_C, \quad \Delta T_C = T_C - T_{init}$$

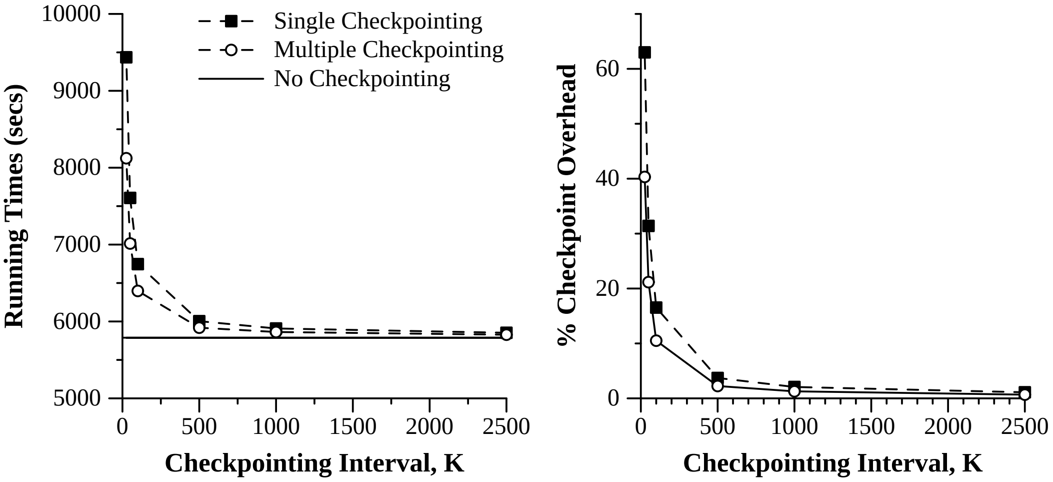


Figure 4.6: PCG iterative solver: timing results

n	T_A (sec)	N_C +1	T (sec)	$\frac{T-T_{init}}{N_C}$ (sec)	T_C (sec)	%	T_{init} (sec)	$\frac{\Delta T_C}{N_C}$ (sec)	T_R (sec)
		With Single Checkpointing							
1000	45	2	52	23.1	7	15.6	3	1.8	47
2000	153	3	180	51.4	27	17.6	11	4.6	86
3000	364	4	436	91.8	72	19.8	25	9.9	144
4000	745	6	884	147.3	139	18.7	43	16.0	190
5000	1293	7	1525	210.3	232	17.9	69	22.5	279
6000	2144	8	2525	297.1	381	17.8	98	33.3	399
7000	3211	9	3760	385.6	549	17.1	134	42.6	528
8000	4774	11	5590	508.2	816	17.1	175	58.3	646
9000	6268	12	7555	616.7	1287	20.5	229	86.4	811
10000	8651	13	10447	773.9	1796	20.8	282	112.1	982
		With Multiple Checkpointing							
1000	45	2	50	22.2	5	11.1	2	1.3	46
2000	153	3	170	48.6	17	11.1	7	2.9	81
3000	364	4	404	85.1	40	11.0	16	5.1	134
4000	745	6	825	137.5	80	10.7	29	8.5	175
5000	1293	7	1427	196.8	134	10.4	45	12.3	254
6000	2144	8	2368	278.6	224	10.4	65	18.7	363
7000	3211	9	3561	365.2	350	10.9	90	26.7	484
8000	4774	11	5271	479.2	497	10.4	115	34.7	581
9000	6268	12	7042	574.9	774	12.3	153	50.7	711
10000	8651	13	9732	720.9	1081	12.5	181	66.7	885

Checkpointing Interval $K = 16$, $\Delta T_C = T_C - T_{init}$, $T = T_A + T_C$

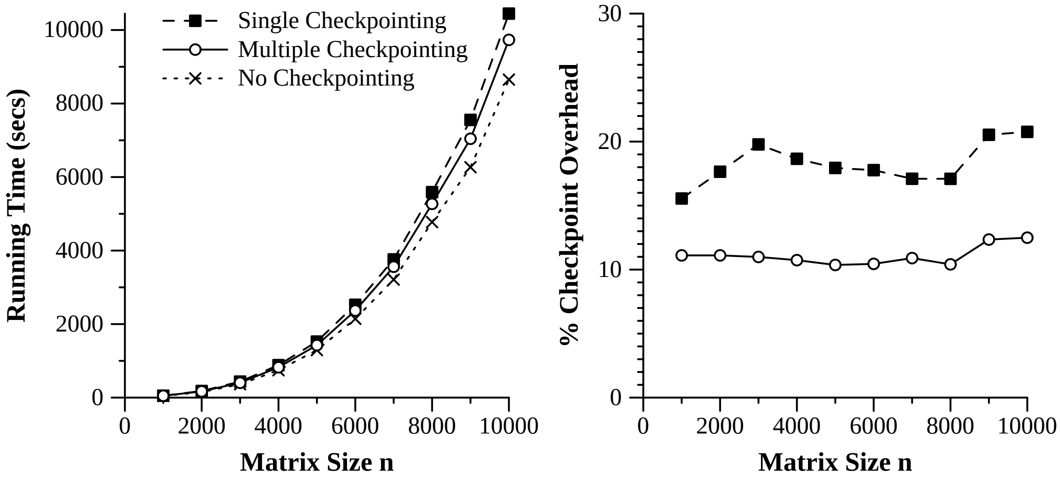


Figure 4.7: Right-looking LU: timing results

n	T_A (sec)	N_C +1	T (sec)	$\frac{T-T_{init}}{N_C}$ (sec)	T_C (sec)	T_C %	T_{init} (sec)	$\frac{\Delta T_C}{N_C}$ (sec)	T_R (sec)
		With Single Checkpointing							
1000	10	2	14	6.2	4	40.0	2	0.9	11
2000	52	3	70	20.0	18	34.6	6	3.4	32
3000	147	4	178	37.5	31	21.1	12	4.0	61
4000	332	6	391	65.2	59	17.8	23	6.0	88
5000	574	7	697	96.1	123	21.4	34	12.3	131
6000	942	8	1140	134.1	198	21.0	50	17.4	184
7000	1466	9	1754	179.9	288	19.6	68	22.6	261
8000	2145	11	2551	231.9	406	18.9	88	28.9	304
9000	3004	12	3581	292.3	577	19.2	114	37.8	387
10000	4068	13	4833	358.0	765	18.8	141	46.2	478
		With Multiple Checkpointing							
1000	10	2	12	5.3	2	20.0	1	0.4	11
2000	52	3	61	17.4	9	17.3	4	1.4	30
3000	147	4	166	34.9	19	12.9	8	2.3	57
4000	332	6	373	62.2	41	12.3	14	4.5	81
5000	574	7	645	89.0	71	12.4	23	6.6	119
6000	942	8	1052	123.8	110	11.7	32	9.2	167
7000	1466	9	1643	168.5	177	12.1	52	12.8	235
8000	2145	11	2390	217.3	245	11.4	58	17.0	272
9000	3004	12	3361	274.4	357	11.9	72	23.3	346
10000	4068	13	4553	337.3	485	11.9	90	29.3	428

Checkpointing Interval $K = 16$, $\Delta T_C = T_C - T_{init}$, $T = T_A + T_C$

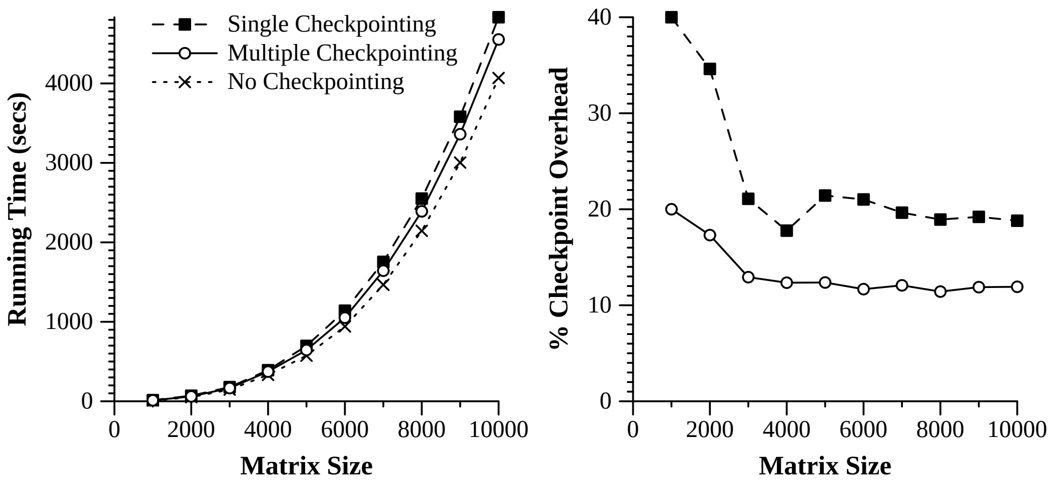


Figure 4.8: Right-looking Cholesky: timing results

n	T_A (sec)	N_C +1	T (sec)	$\frac{T-T_{init}}{N_C}$ (sec)	T_C (sec)	T_C %	T_{init} (sec)	$\frac{\Delta T_C}{N_C}$ (sec)	T_R (sec)
		With Single Checkpointing							
1000	52	6	59	9.8	7	13.5	3	0.7	13
2000	203	11	248	22.5	45	22.2	11	3.1	33
3000	535	16	677	42.3	142	26.5	25	7.3	67
4000	1152	21	1473	70.1	321	27.9	44	13.2	107
5000	2147	26	2733	105.1	586	27.3	69	19.9	162
6000	3545	31	4488	144.8	943	26.6	99	27.2	225
7000	5482	36	6983	194.0	1501	27.4	133	38.0	298
8000	8002	41	10336	252.1	2334	29.2	174	52.7	385
		With Multiple Checkpointing							
1000	52	6	55	9.2	3	5.8	2	0.2	11
2000	203	11	231	21.0	28	13.8	7	1.9	26
3000	535	16	607	37.9	72	13.5	16	3.5	54
4000	1152	21	1315	62.6	163	14.1	28	6.4	90
5000	2147	26	2419	93.0	272	12.7	44	8.8	136
6000	3545	31	4064	131.1	519	14.6	64	14.7	189
7000	5482	36	6404	177.9	922	16.8	87	23.2	250
8000	8002	41	9441	230.3	1439	18.0	114	32.3	324

Checkpointing Interval $K = 4$, $\Delta T_C = T_C - T_{init}$, $T = T_A + T_C$

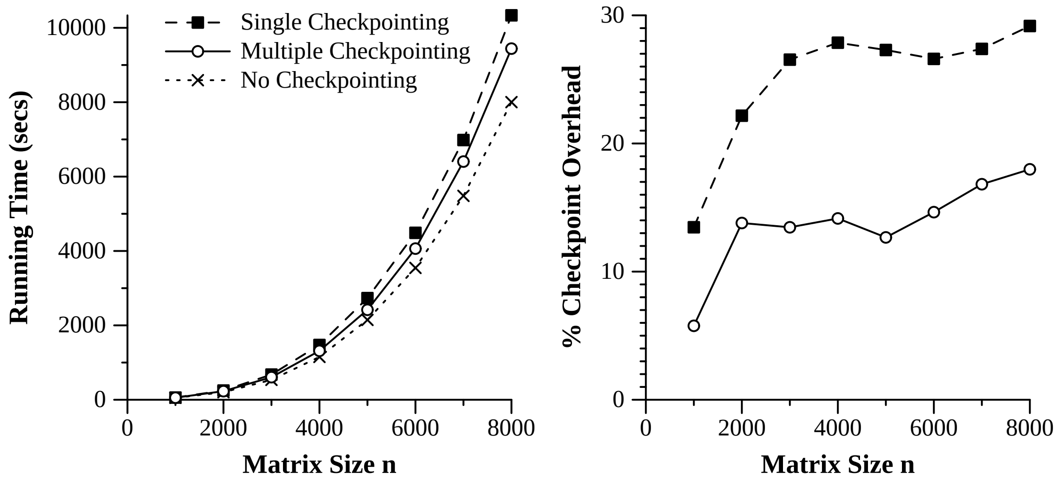


Figure 4.9: Right-looking QR: timing results

n	T_A (sec)	N_C $+1$	T (sec)	$\frac{T-T_{init}}{N_C}$ (sec)	T_C (sec)	T_C %	T_{init} (sec)	$\frac{\Delta T_C}{N_C}$ (sec)	T_R (sec)
		With Single Checkpointing							
1000	154	6	133	22.2	-21	-13.6	3	-4.0	32
2000	601	11	569	51.7	-32	-5.3	11	-3.9	74
3000	1524	16	1607	100.4	83	5.4	24	3.7	135
4000	3158	21	3491	166.2	333	10.5	44	13.8	211
5000	5744	26	6423	247.0	679	11.8	68	23.5	308
6000	9323	31	10653	343.6	1330	14.3	98	39.7	415
7000	14258	36	16478	457.7	2220	15.6	135	57.9	552
		With Multiple Checkpointing							
1000	154	6	130	21.7	-24	-15.6	2	-4.3	30
2000	601	11	560	50.9	-41	-6.8	7	-4.4	62
3000	1524	16	1572	98.2	48	3.1	16	2.0	121
4000	3158	21	3340	159.0	182	5.8	28	7.3	196
5000	5744	26	6122	235.5	378	6.6	45	12.8	277
6000	9323	31	10033	323.6	710	7.6	64	20.8	380
7000	14258	36	15585	432.9	1327	9.3	88	34.4	504

Checkpointing Interval $K = 4$, $\Delta T_C = T_C - T_{init}$, $T = T_A + T_C$

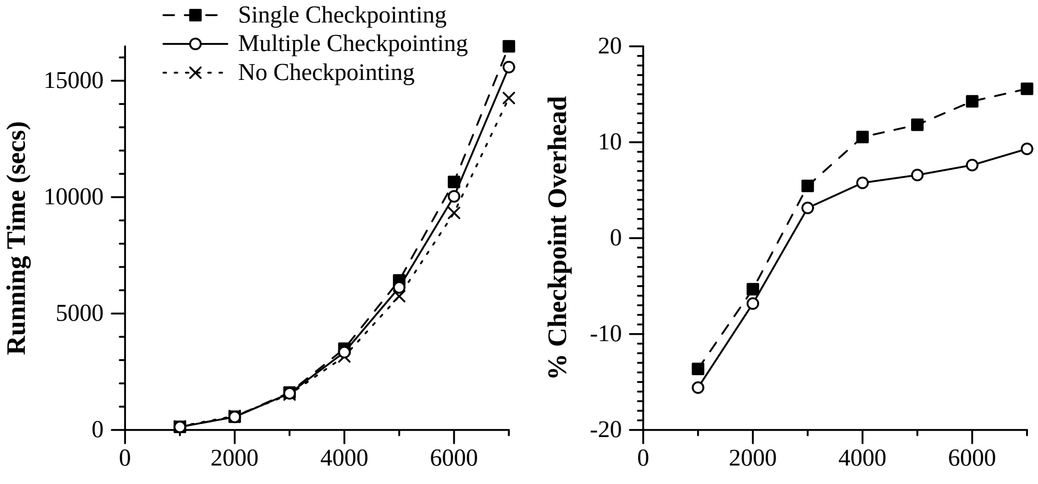


Figure 4.10: Hessenberg reduction: timing results

n	T_A (sec)	N_C +3	T (sec)	$\frac{T-T_{init}}{N_C}$ (sec)	T_C (sec)	%	T_{init} (sec)	$\frac{\Delta T_C}{N_C}$ (sec)	T_R (sec)
		With Single Checkpointing							
1000	29	2+3	43	19.1	14	48.3	8	2.7	33
2000	197	3+3	261	74.6	64	32.5	32	9.1	109
3000	644	4+3	816	171.8	172	26.7	73	20.8	244
4000	1547	6+3	1941	323.5	394	25.5	131	43.8	451
5000	2951	7+3	3682	507.9	731	24.8	205	72.6	638
6000	5036	8+3	6170	725.9	1134	22.5	295	98.7	938
7000	7920	9+3	9546	979.1	1626	20.5	406	125.1	1297
		With Multiple Checkpointing							
1000	29	2+3	37	16.4	8	27.6	5	1.3	28
2000	197	3+3	238	68.0	41	20.8	21	5.7	92
3000	644	4+3	756	159.2	112	17.4	48	13.5	214
4000	1547	6+3	1797	299.5	250	16.2	85	27.5	400
5000	2951	7+3	3418	471.4	467	15.8	149	43.9	568
6000	5036	8+3	5855	688.8	819	16.3	191	73.9	827
7000	7920	9+3	9108	934.2	1188	15.0	263	94.9	1179

Checkpointing Interval $K = 16$, $\Delta T_C = T_C - T_{init}$, $T = T_A + T_C$

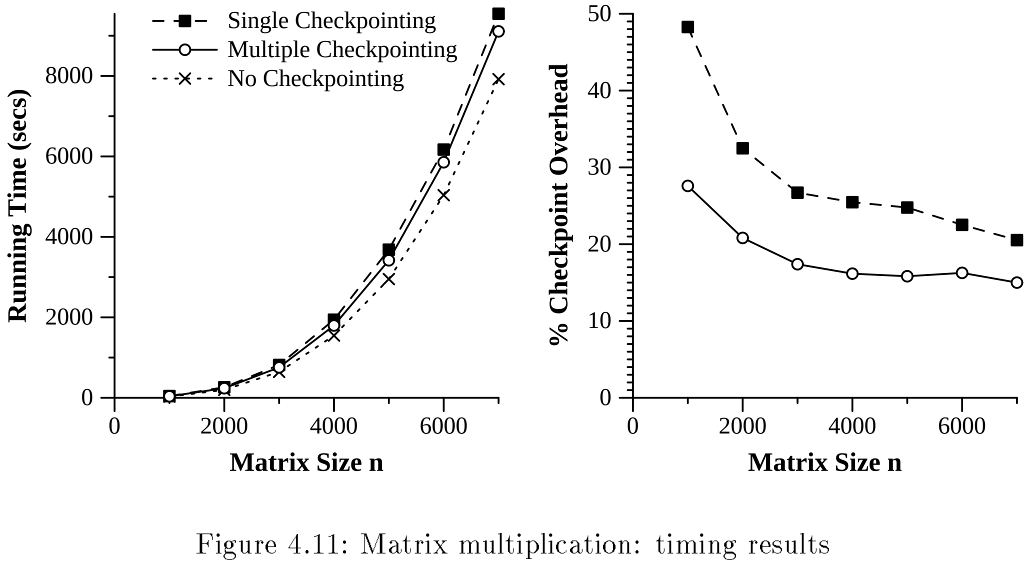


Figure 4.11: Matrix multiplication: timing results

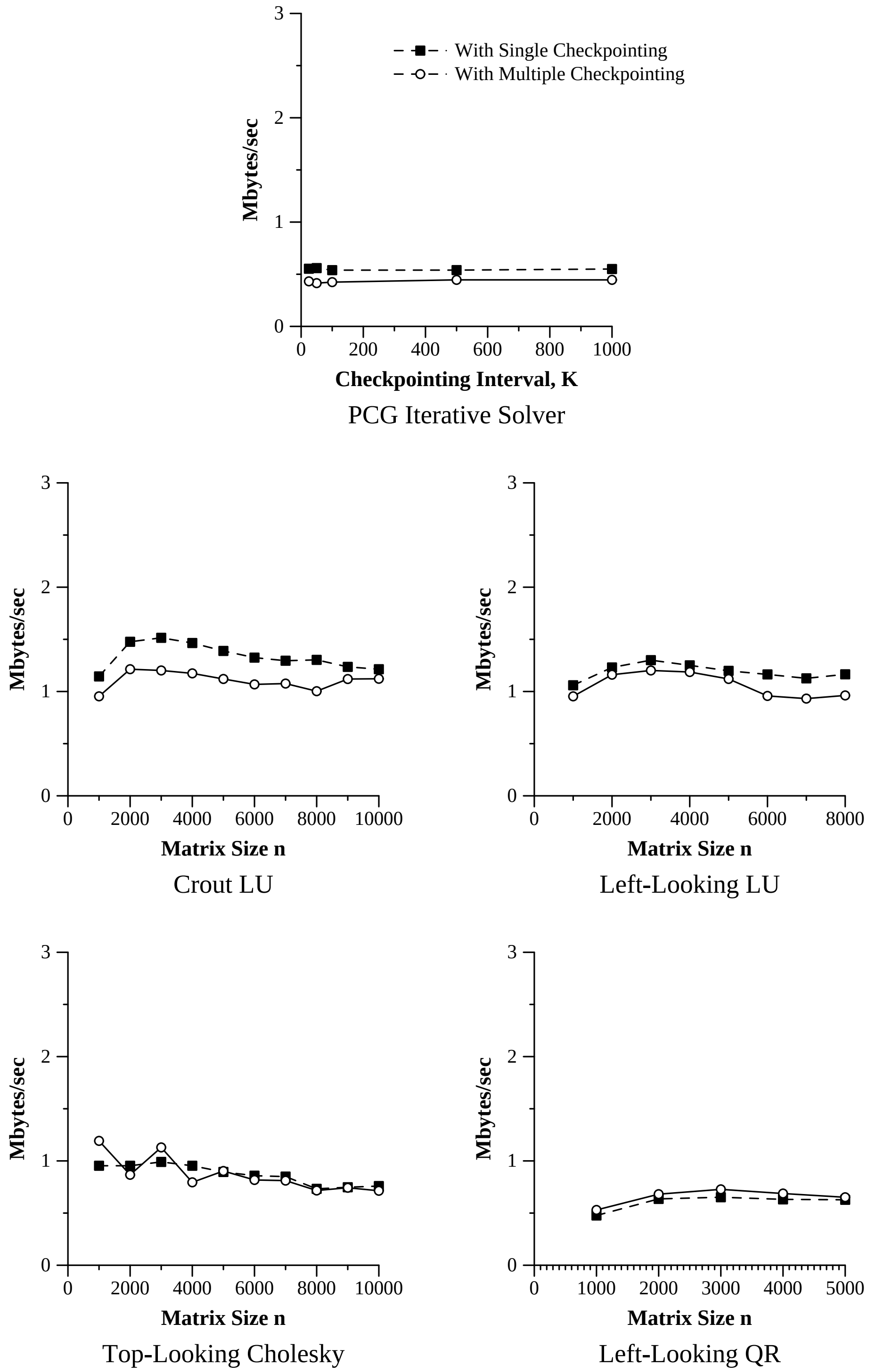
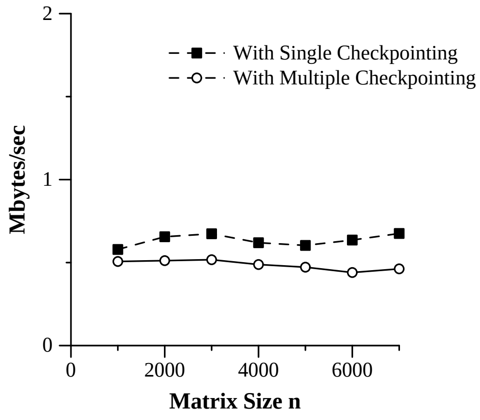
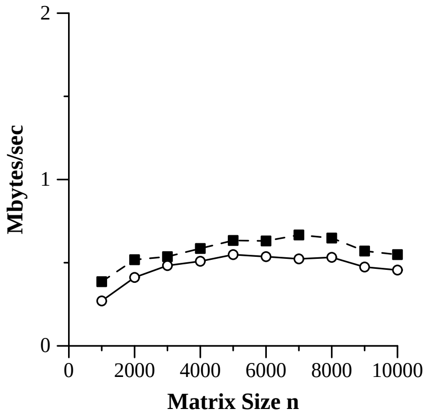


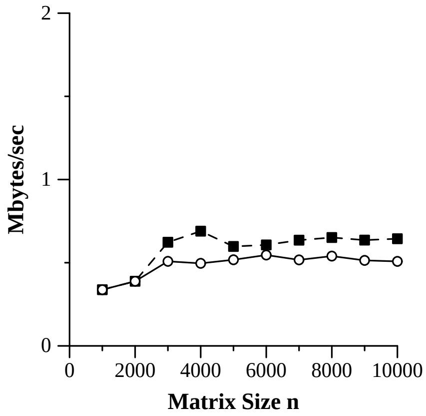
Figure 4.12: Experimental checkpointing rate



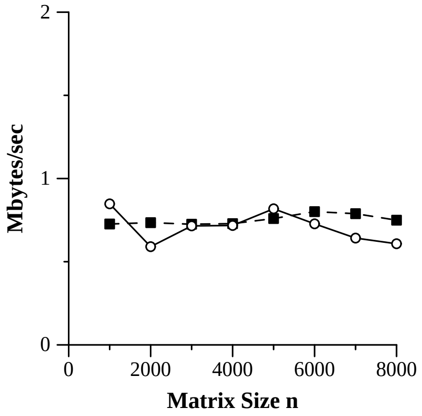
Matrix Multiplication



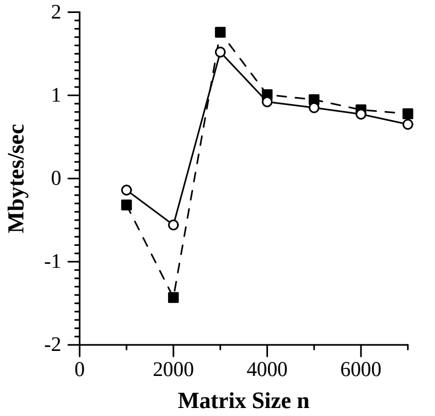
Right-Looking LU



Right-Looking Cholesky



Right-Looking QR



Hessenberg Reduction

Figure 4.13: Experimental checkpointing rate (cont'd)

4.3.3 Summary

Since the measured peak bandwidth of the network is 40 Mbits per second, we expect that the checkpointing rate should be somewhat lower than 5 Mbytes per second considering synchronization, copying, performing `XOR`, and message latency and network contention. As shown in Figures 4.12 and 4.13, the checkpointing rate determined experimentally is between 0.5 and 1 Mbytes per second for all the matrix operations with the exception of Hessenberg reduction.

Figure 4.14 presents the total running times of both the failure-free and fault-tolerant implementations of the three factorizations. The right-looking variant performs the best among the failure-free variants of each factorization because it benefits from less communication and more parallelism than the others. However, for the LU and Cholesky factorizations, the Crout or left-looking variants with checkpointing perform better than the right-looking variant with checkpointing.

For the QR factorization, no top-looking variant exists, and the left-looking variant performs much slower than the right-looking variant. The total checkpointing overhead of the left-looking variant is too high compared with the right-looking variant without checkpointing. The checksum and reverse computation-based variant is the only one available for Hessenberg reduction and matrix multiplication. Therefore, the checksum and reverse computation technique is the method of choice for QR factorization, Hessenberg reduction, and matrix multiplication.

On the other hand, as shown in the complexity analysis, note that the choice of a different processor grid slightly affects the overhead of checkpointing and recovery as well as the performance of the failure-free algorithm without checkpointing.

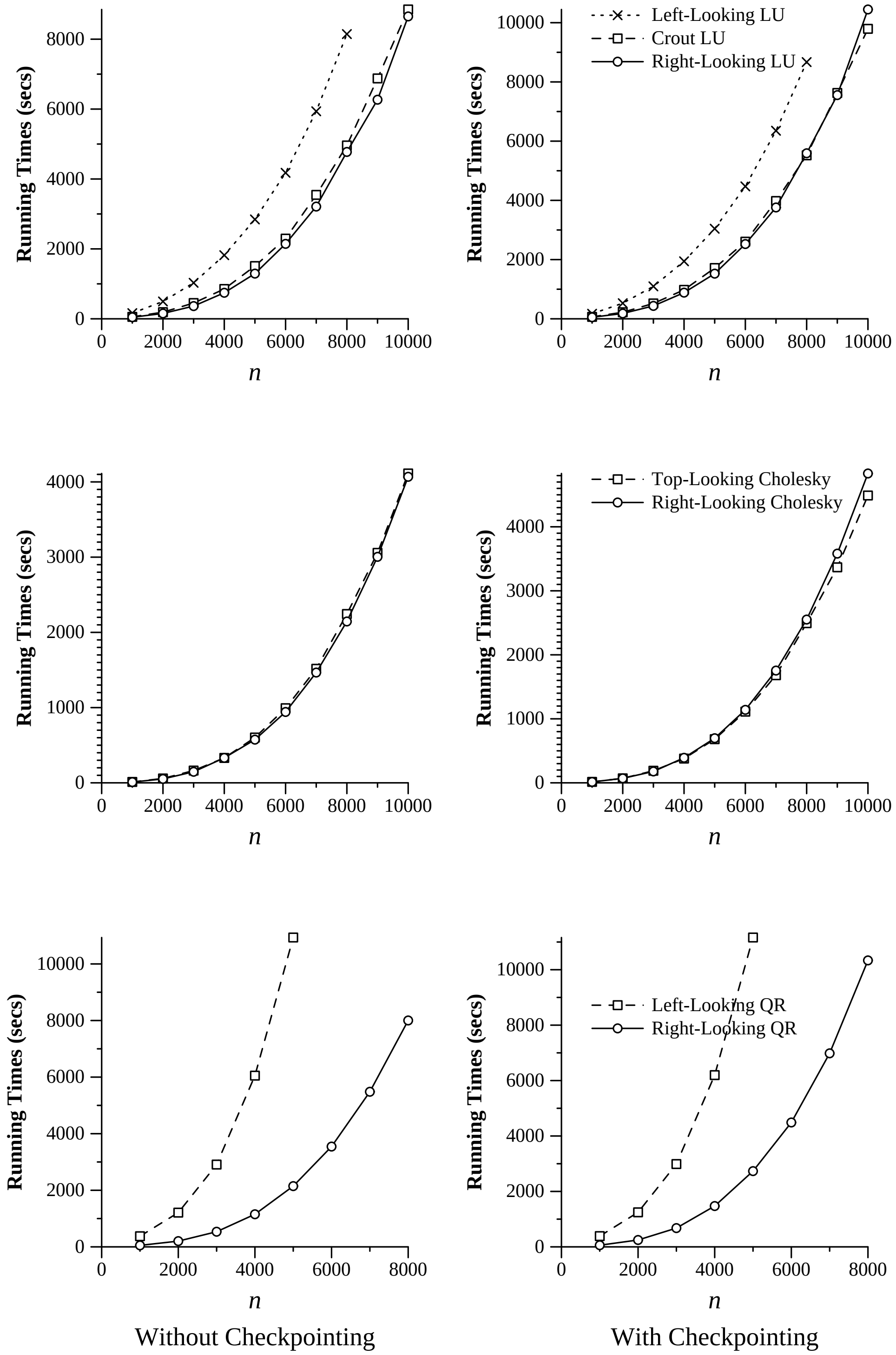


Figure 4.14: Comparison among factorization variants

Chapter 5

Conclusion and Future Work

In this dissertation, we have (1) provided a detailed evaluation of the fault tolerance techniques for matrix operations for distributed scientific computing, (2) proposed a new fault-tolerant computing paradigm, and (3) experimentally demonstrated the feasibility of this paradigm on a network of workstations. This chapter summarizes our conclusions and presents directions for future research.

5.1 Conclusions

In this section we detail our conclusions about three fault tolerance techniques: parity-based, checksum and reverse computation-based, and multiple checkpointing.

5.1.1 Parity-Based Technique

The parity-based algorithms under consideration require $O(n^3)$ floating-point operations, and fault tolerance requires only an additional $O(n^2)$ operations. The fault-tolerant matrix operations can be characterized as follows:

1. Very low overhead while checkpointing at a relatively fine-grain interval.

2. Robust and easy to incorporate this technique into numerical algorithms.

3. Checkpointing and recovery does not cause numerical problems such as overflow and underflow.

4. Block size has little impact on checkpointing and recovery overhead.

5. Usefulness is limited to those matrix operations in which a moderate amount of data is modified between two checkpoints.

5.1.2 Checksum and Reverse Computation-Based Technique

The checksum and reverse computation-based technique incurs checkpointing and recovery overhead that becomes relatively large compared with the operation count of the failure-free algorithm. We have, however, drawn the following conclusions:

1. This technique is a feasible alternative to provide fault tolerance for those numerical algorithms that cannot be rendered fault-tolerant by using the parity-based technique.

2. This technique is appropriate for those matrix operations that require too much extra memory for other fault tolerance techniques.

3. The checkpointing overhead can be reduced by expanding the checkpointing interval. This process requires additional memory usage.

4. Block size affects overhead, which can be compensated for by an appropriate choice of checkpointing interval.

5. Numerical problems such as overflow, underflow, and roundoff error due to cancellation may occur, but are of little practical concern.

5.1.3 Multiple Checkpointing Technique

The numerical results of the multiple checkpointing technique confirm that the technique is more efficient and reliable by not only distributing the process of checkpointing and rollback recovery over groups of processors but also by tolerating multiple failures in one of each group of processors. This technique has been shown to improve both the reliability of the computation and the performance of the checkpointing and recovery. In particular, for the checksum and reverse computation-based implementations, multiple checkpointing could reduce the checkpointing and recovery overhead without using more memory. In addition, the probability of overflow, underflow, and cancellation error can be reduced. Finally, it is easier to develop fault-tolerant implementations when multiple checkpointing processors are used for checkpointing and recovery.

5.1.4 Remarks about the Paradigm

This research presents a paradigm for executing certain scientific computations on a changing distributed computing platform. The paradigm allows a distributed computation to run on a platform where individual processors may leave due to failures, unavailability, or heavy load, and where processors may enter during the computation. If the computation is designed for N application processors, it may proceed efficiently as long as the number of processors in the distributed system remains greater than N and as long as processors leave the system singly or in groups such that only one processor failure occurs in the same checkpointing group.

In general, we have drawn the following conclusions about our new paradigm to provide fault tolerance for numerical algorithms for distributed scientific computations:

1. The proposed paradigm is feasible and effective.

2. Algorithm-based fault tolerance techniques efficiently exploit the semantics of the application at low cost in programmer effort.

3. The programming effort to render applications fault-tolerant is justifiable when the fault-tolerant implementations are provided as numerical libraries.

4. The paradigm provides an interesting way to allow reliability in computations performed on networks of computers.

5. The paradigm supports a heterogeneous network of workstations.

6. The paradigm can be adapted to the algorithms based on any data distribution, rather than being specific to the block cyclic distribution employed in the current implementations.

We emphasize that our scheme can be used to detect data corruption in a system where such detection is otherwise absent. In coding theory terms, our mechanism can provide detection in an error-prone environment or correction in an erasure environment [PW72].

It is obvious that many state-of-the-art MPPs such as the IBM SP2 and the Cray T3E have become more like networks of workstations. Fault tolerance is normally not provided in such parallel and distributed-computing platforms. If such computing platforms support our fail-stop processor model [SS83] and facilitate failure detection, we may use these fault tolerance techniques on MPPs in the same way as on workstation networks, barring only mass failures and scheduled shutdowns.

5.2 Future Work

Two general directions of research can follow on the work presented in this dissertation. The first is to improve the current implementations and apply the paradigm to different numerical algorithms. The second is to investigate some issues that were not considered in this dissertation.

Our implementations use a constant checkpointing interval in terms of iterations throughout the execution of the fault-tolerant algorithm. This results in higher checkpointing overhead as the amount of work per iteration decreases as the algorithm proceeds. A compensating increase in the checkpointing interval as the algorithm proceeds would result in consistent checkpointing overhead throughout the execution of the fault-tolerant algorithm.

Failure detection is of little concern in our computation models. However, if a particular parallel programming environment does not support a failure detection mechanism, such a mechanism should be developed at a general software level and hence will result in reasonable overhead. An algorithm-based error detection scheme as discussed in Section 1.7 would be a method of choice for failure detection. However, a separate encoding scheme should be employed.

The multiple checkpointing technique is shown to be more effective and reliable than the single checkpointing technique. There are several more complicated schemes for configuring multiple checkpointing processors to tolerate more general sets of multiple failures. These schemes include *two-dimensional parity* and *multi-dimensional parity* [GHKP89], the *Reed-Solomon* coding scheme [PW72, Rom92, PFL94], and *Evenodd parity* [BBBM94]. The paper [Par95] also proposes a scheme to tolerate any two failures in RAID. One possible direction of future research is to investigate how such schemes can be employed to tolerate different groups of multiple failures or a random set of multiple failures. We expect it to be challenging to implement any such fault-tolerant scheme into the target matrix operations discussed in this dissertation.

Finally, we would like to integrate this fault-tolerant computing paradigm with general load-balancing (e.g., CARMI [PL94d]). In other words, if a few processors are added to or deleted from the network of workstations, then the system continues running using the techniques presented in this dissertation. However, if the size of the processor pool changes by an order of magnitude, then it makes sense to reconfigure the system with a different value of N and redistribute the data over the new configuration. This solution can be another challenging direction of future research. Such an integration would represent a truly adaptive, high-performance methodology for scientific computations on distributed systems.

Bibliography

Bibliography

[ABD$^+$95] E. Anderson, Z. Bai, J. Demmel, J. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide*. SIAM Press, Philadelphia, PA, 1995.

[Agr88] P. Agrawal. Fault tolerance in multiprocessor systems without dedicated redundancy. *IEEE Transactions on Computers*, 37(3):358–362, March 1988.

[AGZ94] R. Agarwal, F. Gustavson, and M. Zubair. A high performance matrix multiplication algorithm on a distributed-memory parallel computer, using overlapped communication. *IBM Systems Journal*, 38(6):673–681, November 1994.

[AL81] T. Anderson and P. A. Lee. *Fault Tolerance Principles and Practice*. Prentice Hall International, Inc., Englewood Cliffs, New Jersey, 1981.

[AL88] C. J. Anfinson and F. T. Luk. A linear algebraic model of algorithm-based fault tolerance. *IEEE Transacions on Computers*, C-37(12):1599–1604, December 1988.

[AOST93] T. Anderson, S. Owicki, J. Saxe, and C. Thacker. High speed switch scheduling for local area networks. *ACM Transactions on Computer Systems*, pages 319–352, 1993.

[BA86] P. Banerjee and J. A. Abraham. Bounds on algorithm-based fault tolerance in multiple processor systems. *IEEE Transactions on Computers*, C-35(4):296–306, April 1986.

[Bal91] V. Balasubramanian. *The Analysis and Synthesis of Efficient Algorithm-Based Error Detection Schemes for Hypercube Multiprocessors*. PhD thesis, Univ. of Illinois, Urbana-Champaign, Urbana-Champaign, IL, February 1991.

[BBBM94] M. Blaum, J. Brady, J. Bruck, and J. Menon. EVENODD: An optimal scheme for tolerating double disk failures in RAID architectures. pages 245—254, April 1994.

[BBC+95] R. Barrett, M. Berry, Tony 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.* SIAM, Philadelphia, PA, second edition, 1995.

[BBG83] A. Borg, J. Baumbach, and S. Glazer. A message system supporting fault tolerance. In *ACM Symposium on Operating System Principles*, pages 90–99, Atlanta, GA, October 1983.

[BBG+89] A. Borg, W. Blau, W. Graetsch, F. Herrman, and W. Oberle. Fault tolerance under UNIX. *ACM Transactions on Computer Systems*, 7(1):1–24, Feb 1989.

[Bea95] N. J. Boden et al. Myrinet – a gigabet-per-second local-area network. *IEEE Micro*, 15(1):29–36, February 1995.

[BGM+95] D. Boley, G. H. Golub, S. Makar, N. Saxena, and E. J. McCluskey. Floating point fault tolerance with backward error assertions. *IEEE Transactions on Computers*, C-44(2):302–311, February 1995.

[Bir93] K. Birman. The process group approach to reliable distributed computing. *Communications of the ACM*, 36(12), December 1993.

[BM93] W. A. Burkhard and J. Menon. Disk array storage system reliability. pages 432–441, June 1993.

[BP93] N. S. Bowen and D. K. Pradhan. Processor- and memory-based checkpoint and rollback recovery. *IEEE Computer*, 26(2):22–31, 1993.

[BRS+88] P. Banerjee, J. T. Rahmeh, C. B. Stunkel, V. S. S. Nair, K. Roy, and J. A. Abraham. An evaluation of system-level fault tolerance on the intel hypercube multiprocessor. In *18th IEEE International Symposium on Fault Tolerant Computing*, pages 362–367, Tokyo, Japan, June 1988.

[BRS+90] P. Banerjee, J. T. Rahmeh, C. Stunkel, V. S. Nair K. Roy, V. Balasubramanian, and J. A. Abraham. Algorithm-based fault tolerance on a hypercube multiprocessor. *IEEE Transactions on Computers*, C-39(9):1132–1145, September 1990.

[BS89] D. E. Bakken and R. D. Schilchting. Supporting fault-tolerant parallel programming in Linda. *ACM Transactions on Computer Systems*, 7(1):1–24, Feb 1989.

[CA94] D. Cummings and L. Alkalaj. Checkpoint/rollback in a distributed system using coarse-grained dataflow. In *24th International Symposium on Fault-Tolerant Computing*, pages 424–433, June 1994.

[CCK+95] J. Casas, D. Clark, R. Konuru, S. Otto, R. Prouty, and J. Walpole. MPVM: A migration transparent version of PVM. *Computing Systems*, 8(2):171–216, Spring 1995.

[CDO+96] J. Choi, J. J. Dongarra, S. Ostrouchov, A. P. Petitet, D. W. Walker, and R. C. Whaley. The design and implementation of the ScaLAPACK LU, QR, and Cholesky factorization routines. *Scientific Programming (to appear)*, 1996.

[CDW95] J. Choi, J. J. Dongarra, and D. W. Walker. The design of a parallel, dense linear algebra software library: Reduction to Hessenberg, tridiagonal, and bidiagonal form. *Numerical Algorithms*, 10:379–399, 1995.

[CDW96] J. Choi, J. J. Dongarra, and D. W. Walker. PB-BLAS: A set of parallel block basic linear algebra subprograms. *Concurrency: Practice and Experience (to appear)*, 1996.

[CG89] N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, (4):444–458, April 1989.

[Che93] D. Cheng. A survey of parallel programming languages and tools. Technical Report RND-93-005, NASA Ames Research Center, 1993.

[CJ91] F. Cristian and F. Jahanain. A timestamp-based checkpointing protocol for long-lived distributed computations. In *10th Symposium on Reliable Distributed Systems*, pages 12–20, October 1991.

[CLG+94] P. Chen, E. Lee, G. Gibson, R. Katz, and D. Patterson. High-performance, reliable secondary storage. *ACM Computing Surveys*, 26(2):145–186, June 1994.

[CM88a] Y. H. Choi and M. Malek. A fault tolerant FFT processor. *IEEE Transactions on Computers*, C-37(5):617–621, May 1988.

[CM88b] Y. H. Choi and M. Malek. A fault tolerant systolic sorter. *IEEE Transactions on Computers*, C-37(5):621–624, May 1988.

[Coo85] E. Cooper. Replicated distributed programs. *Operating Systems Review*, 19(5):63–78, December 1985.

[Cor88] ParaSoft Corporation. *Express Version 1.0: A Communication Environment for Parallel Computers.* 1988.

[dCH95] R. D. da Cunha and T. Hopkins. *PIM 2.0 – The Parallel Iterative Methods Package for Systems of Linear Equations User's Guide.* University of Kent, Canterbury, UK, 1995.

[DCHD90] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. Duff. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 18(1):1–17, 1990.

[DDSvdV91] J. J. Dongarra, I. S. Duff, D. C. Sorensen, and H. A. van der Vorst. *Solving Linear Systems on Vector and Shared Memory Computers.* SIAM, Philadelphia, PA, second edition, 1991.

[DGK84] J. J. Dongarra, F. Gustavson, and A. Karp. Implementating linear algebra algorithms for dense matrices on a vector pipeline machine. *SIAM Review*, 26(1):91–112, January 1984.

[DJAR90] P. Dasgupta, R. LeBlanc Jr., M. Ahamad, and U. Ramachandran. The Coulds distributed operating system. *IEEE Computer*, 1990.

[DW95] J. J. Dongarra and R. C. Whaley. A user's guide to the BLACS v1.0. LAPACK Working Note 94, Technical Report CS-95-281, University of Tennessee, 1995.

[Dwy51] Paul Sumner Dwyer. *Linear Computations.* Wiley, New York, 1951.

[EJZ92] E. N. Elnozahy, D. B. Johnson, and W. Zwaenepoel. The performance of consistent checkpointing. In *11th Symposium on Reliable Distributed Systems*, pages 39–47, October 1992.

[FB89] S. Feldman and C. Brown. IGOR: A system for program debugging via reversible execution. *ACM SIGPLAN Notices, Workshop on Parallel and Distributed Debugging*, 24(1):112–123, Jan 1989.

[FMM77] G. E. Forsythe, M. A. Malcolm, and C. B. Moler. *Computer Methods for Mathematical Computations.* Prentice Hall, Englewood Cliffs, NJ, 1977.

[FRK94] G. L. Feng, T. R. N. Rao, and M. S. Kolluru. Error correcting codes over Z_{2^m} for algorithm-based fault tolerance. *IEEE Transactions on Computers*, C-43(3):370–374, March 1994.

[GBD+94] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine – A User's Guide and Tutorial for Networked Parallel Computing.* MIT Press, Cambridge, MA, 1994.

[GC95] E. Gelenbe and S. Chabridon. Dependable execution of distributed programs. *Simulation Practice and Theory*, 3(1):1–16, 1995.

[GHKP89] G. A. Gibson, L. Hellerstein, R. M. Karp, and D. A. Patterson. Failure correction techniques for large disk arrays. pages 123–132, April 1989.

[Gib90] G. A. Gibson. *Redundant Disk Arrays: Reliable, Parallel Secondary Storage.* PhD thesis, University of California, Berkeley, Berkeley, CA, December 1990.

[Gib92] G. A. Gibson. *Redundant Disk Arrays: Reliable, Parallel Secondary Storage.* The MIT Press, Cambridge, MA, 1992.

[GL89] G. Golub and C. Van Loan. *Matrix Computations.* Johns-Hopkins, Baltimore, second edition, 1989.

[GRR90] D. Gu, D. J. Rosenkrantz, and S. S. Ravi. Design and analysis of test schemes for algorithm-based fault tolerance. In *20th International Symposium on Fault-Tolerant Computing*, pages 106–113, June 1990.

[HA84] K-H. Huang and J. A. Abraham. Algorithm-based fault tolerance for matrix operations. *IEEE Transactions on Computers*, C-33(6):518–528, June 1984.

[Hig96] Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithms.* SIAM, 1996.

[HP94] A. Heddaya and K. Park. Mapping parallel iterative algorithms onto workstation networks. Technical Report BU-CS-94-003, Boston University, 1994.

[JA84] J.-Y. Jou and J. A. Abraham. Fault tolerant matrix operations on multiple processor systems using weighted checksums. In *SPIE Proceedings*, volume 495, August 1984.

[JA86] J.-Y. Jou and J. A. Abraham. Fault tolerant matrix arithmetic and signal processing on highly concurrent computing structures. In *Proceedings of IEEE*, volume 74(5), pages 732–741, May 1986.

[Jal94] P. Jalote. *Fault Tolerance in Distributed Systems.* Prentice Hall, Englewood Cliffs, NJ, 1994.

[Joh89a] B. W. Johnson. *Design and Analysis of Fault Tolerant Digital Systems.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1989.

[Joh89b] D. B. Johnson. *Distributed System Fault Tolerance Using Message Logging and Checkpointing.* PhD thesis, Rice University, December 1989.

[JZ90] D. B. Johnson and W. Zwaenepoel. Recovery in distributed systems using optimistic message logging and checkpointing. *Journal of Algorithms*, 11(3):462–491, September 1990.

[KT87] R. Koo and S. Toueg. Checkpointing and rollback-recovery for distributed systems. *IEEE Transactions on Software Engineering*, SE-13(1):23–31, January 1987.

[LA94] S.-T. Levi and A. K. Agrawala. *Fault Tolerant System Design.* Computer Engineering. McGraw-Hill, New York, NY, 1994.

[LCP91] D. D. E. Long, J. L. Carroll, and C. J. Park. A study of the reliability of Internet sites. In *10th Symposium on Reliable Distributed Systems*, pages 177–186, October 1991.

[LFS93] J. León, A. L. Fisher, and P. Steenkiste. Fail-Safe PVM: A portable package for distributed programming with transparent recovery. Technical Report CMU-CS-93-124, Carnegie Mellon University, February 1993.

[LMJ93] L. A. Laranjeira, M. Malek, and R. M. Jenevein. Space/time overhead analysis and experiments with techniques for fault tolerance. *Dependable Computing and Fault-Tolerant Systems*, 8(3):303–318, 1993.

[LNP91] K. Li, J. F. Naughton, and J. S. Plank. Checkpointing multicomputer applications. In *10th Symposium on Reliable Distributed Systems*, pages 1–11, October 1991.

[LNP94] K. Li, J. F. Naughton, and J. S. Plank. Low-latency, concurrent checkpointing for parallel programs. *IEEE Transactions on Parallel and Distributed Systems*, 5(8):874–879, August 1994.

[Lo94] J.-C. Lo. Reliable floating-point arithmetic algorithms for error-coded operands. *IEEE Transactions on Computers*, C-43(4):400–412, April 1994.

[LP88a] F. T. Luk and H. Park. An analysis of algorithm-based fault tolerance techniques. *Journal of Parallel and Distributed Computing*, 5:172–184, 1988.

[LP88b] F. T. Luk and H. Park. Fault-tolerant matrix triangularizations on systolic arrays. *IEEE Transactions on Computers*, C-37(11):1434–1438, November 1988.

[LY87] T. H. Lai and T. H. Yang. On distributed snapshots. *Information Processing Letters*, 25:153–158, May 1987.

[Mai93] J. Maier. Pact – a fault tolerant parallel programming environment. February 1993.

[ML91] M. W. Mutka and M. Livny. The available capacity of a privately owned workstation environment. *Perfomance Evaluation*, August 1991.

[MS77] F. J. MacWilliams and N. J. A. Sloane. *The Theory of Error-Correcting Codes, Part I.* North-Holland Publishing Company, 1977.

[Mut92] M. W. Mutka. Estimating capacity for sharing in a privately owned workstation environment. *IEEE Transactions on Software Engineering*, 18(4):319–328, April 1992.

[NA88] V. S. S. Nair and J. A. Abraham. General linear codes for fault-tolerant matrix operations on processor arrays. In *18th International Symposium on Fault-Tolerant Computing*, pages 180–185, June 1988.

[NA90] V. S. S. Nair and J. A. Abraham. Real number codes for fault-tolerant matrix operations on processor arrays. *IEEE Transactions on Computers*, C-39(4):426–435, April 1990.

[NHA92] V. S. S. Nair, Y. V. Hoskote, and J. A. Abraham. Probabilistic evaluation of on-line checks in fault-tolerant multiprocessor systems. *IEEE Transactions on Computers*, C-41(5):532–541, May 1992.

[Par92] H. Park. On multiple error detection in matrix triangularizations using checksum methods. *Journal of Parallel and Distributed Computing*, 14:90–97, 1992.

[Par95] C.-I. Park. Efficient placement of parity and data to tolerate two disk failures in disk array systems. *IEEE Transactions on Parallel and Distributed Systems*, 6(11):1177–1184, November 1995.

[PB92] M. Peercy and P. Banerjee. Design and analysis of software reconfiguration strategies for hypercube multicomputers under multiple faults. In *22nd International Symposium on Fault-Tolerant Computing*, pages 448–455, June 1992.

[PFL94] J. S. Plank, J. Friedman, and K. Li. A failure correction technique for parallel storage devices with minimal device overhead. Technical Report CS-94-243, University of Tennessee, August 1994.

[PGK88] D. A. Patterson, G. Gibson, and R. H. Katz. A case for redundant arrays of inexpensive disks (RAID). pages 109–116, 1988.

[PL94a] J. S. Plank and K. Li. Faster checkpointing with $N+1$ parity. In *24th International Symposium on Fault-Tolerant Computing*, pages 288–297, Austin, TX, June 1994.

[PL94b] J. S. Plank and K. Li. Ickp — a consistent checkpointer for multicomputers. *IEEE Parallel & Distributed Technology*, 2(2):62–67, Summer 1994.

[PL94c] J. S. Plank and K. Li. Performance results of *ickp* — a consistent checkpointer on the iPSC/860. In *Scalable High Performance Computing Conference*, pages 686–693, Knoxville, TN, May 1994.

[PL94d] J. Pruyne and M. Livny. Providing resource management services to parallel applications. May 1994.

[Pra86] D. K. Pradhan. Fault-tolerant multiprocessor and VLSI-based system communication architectures. In D. K. Pradhan, editor, *Fault-Tolerant Computing Theory and Techniques*, chapter Chapter 7. Prentice Hall, Englewood Cliffs, NJ, 1986.

[PW72] W. Peterson and E. J. Weldon. *Error-Correcting Codes*. MIT Press, Cambridge MA, second edition edition, 1972.

[PWC+81] G. Popek, B. Walker, J. Chow, D. Edwards, C. Kline, G. Rudisin, and G. Thiel. A network transparent, high reliability distributed system. pages 169–177, 1981.

[RB90] A. L. Narasimha Reddy and P. Banerjee. Algorithm-based fault detection for signal processing applications. *IEEE Transactions on Computers*, C-39(10):1304–1308, October 1990.

[RB94] A. Roy-Chowdhury and P. Banerjee. Algorithm-based fault location and recovery for matrix computations. In *24th International Symposium on Fault-Tolerant Computing*, pages 38–47, Austin, TX, June 1994.

[RH95] K. S. Ramesh and Sudheendra Hangal. Using PVM and MPI for scientific applications. Technical report, Centre for Development of Advanced Computing, Pune India, 1995.

[RJ94] J. Rexford and N. K. Jha. Partitioned encoding schemes for algorithm-based fault tolerance in massively parallel systems. *IEEE Transactions on Prallel and Distributed Systems*, 5(6):649–653, June 1994.

[Rom92] S. Roman. *Coding and Information Theory*. Springer-Verlag, 1992.

[Roy92] A. Roy-Chowdhury. Evaluation of algorithm based fault-tolerance techniques on multiple fault classes in the presence of finite precision arithmetic. Master's thesis, Univ. of Illinois, Urbana-Champaign, August 1992.

[RR93] D. J. Rosenkrantz and S. S. Ravi. Improved bounds for algorithm-based fault tolerance. *IEEE Transactions on Computers*, C-42(5):630–635, May 1993.

[SJF95] G. Sure, R. Janssens, and W. K. Fuchs. Reduced overhead logging for rollback recovery in distributed shared memory. pages 279–288, June 1995.

[Sko94] P. A. Skordos. Parallel simulation of subsonic fluid dynamics on a cluster of workstations. Technical Report A.I. Memo No. 1485, Massachusetts Institute of Technology, November 1994.

[SOHL+96] M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. J. Dongarra. *MPI: The Complete Reference*. MIT Press, Boston, MA, 1996.

[SS82] D. P. Siewiorek and R. S. Swarz. *The Theory and Practice of Reliable System Design*. Digital Press, Bedford, MA, 1982.

[SS83] R. D. Schlichting and F. B. Schneider. Fail-stop processors: An approach to designing fault-tolerant computing systems. *IEEE Transactions on Computers*, 1(3):222–238, August 1983.

[SSCC95] L. M. Silva, J. G. Silva, S. Chapple, and L. Clarke. *Portable Checkpointing and Recovery*. 1995.

[Ste73] G. W. Stewart. *Introduction to Matrix Computations*. Academic Prcess, San Diego, CA, 1973.

[Ste96] G. Stellber. CoCheck: Checkpointing and process migration for MPI. April 1996.

[SVS94] L. M. Silva, B. Veer, and J. G. Silva. Checkpointing SPMD applications on transputer networks. In *Scalable High Performance Computing Conference*, pages 694–701, Knoxville, TN, May 1994.

[SY85] R. E. Strom and S. Yemini. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems*, 3(3):204–226, August 1985.

[THC90] D. L. Tao, C. R. P. Hartmann, and Y. S. Chen. A novel concurrent error detection scheme for FFT networks. In *20th International Symposium on Fault-Tolerant Computing*, pages 114–121, June 1990.

[THM+94] C. Trefftz, C. C. Huang, P. K. McKinley, T. Y. Li, and Z. Zeng. Design and performance evaluation of a distributed eigenvalue solver on a workstation cluster. In *1994 IEEE 14th International Conference on Distributed Computing Systems*, pages 608–615, 1994.

[VJ90] B. Vinnakota and N. K. Jha. A dependence graph based approach to the design of algorithm computations. In *20th International Symposium on Fault-Tolerant Computing*, June 1990.

[VJ93] B. Vinnakota and N. K. Jha. Diagnosability and diagnosis of algorithm-based fault-tolerant systems. *IEEE Transactions on Computers*, C-42(8):924–937, August 1993.

[WF93] Y. M. Wang and W. K. Fuchs. Lazy checkpoint coordination for bounding rollback propagation. In *12th Symposium on Reliable Distributed Systems*, pages 78–85, October 1993.

[Wil63] J. H. Wilkinson. *Rounding Errors in Algebraic Processes.* Prentice Hall, Englewood Cliffs, NJ, 1963.

[Wil65] J. H. Wilkinson. *The Algebraic Eigenvalue Problem.* Oxford, Clarendon Press, 1965.

[WJ94] S.-J. Wang and N. K. Jha. Algorithm-based fault tolerance for FFT networks. *IEEE Transactions on Computers*, C-43(7):849–854, July 1994.

Vita

Youngbae Kim was born in Kwangju, Korea on February 21, 1959. He raised and graduated from the SooChang Elementary, the MooJin Middle, and the Kwangju JeIl High schools there. In 1978, he entered the College of Engineering and Science at the Seoul National University (SNU) in Seoul, Korea. He graduated with a Bachelor's degree in Electronics Engineering in 1982 and earned a Master's degree in 1984. His main interests lay in the fields of digital communication and signal processing. His Master's thesis is entitled "Time Delay Estimation using an Adaptive FIR (Finite Impulse Response) Filter".

In 1984, he began his first professional career as a software engineer at the Computer Systems Research and Development Division of GoldStar (Currently, LG Electronics) Laboratories. He developed several application programs to facilitate using Korean characters in various Unix platforms. After completing the Army Officer training course, he continued to work in the Laboratories until he came to the U.S.A.

In 1990, he completed a Master's degree supervised by Professor Harry Jordan in Electrical and Computer Engineering at the University of Colorado, Boulder. He worked on the Force, a parallel programming language for shared-memory multiprocessors, and developed a sparse triangular solver with the Force. He also worked on Nonsymmetric Eigenvalue problem solvers with Professor Elizabeth Jessup. Meanwhile, he was working as a software programmer at CADSWES (Center for Advanced Decision Support of Water Environmental Systems) and developed various graphical user interfaces for decision support system software.

In 1992, he moved to the Tennessee and continued a doctorate supervised by Professor Jack Dongarra in Computer Science Department at the University of Tennessee, Knoxville, and completed it in August 1996. His research interests include parallel and distributed computing focusing on numerical scientific computations; performance evaluation and benchmarking; network computing; checkpointing and fault tolerance. He will begin his professional career by taking a position in the National Energy Research Scientific Computing (NERSC) Center at Lawrence Berkeley National Laboratory.