UC Riverside
UC Riverside Electronic Theses and Dissertations

Title
Silent Data Corruption Resilient Matrix Factorizations on Distributed Memory System

Permalink
https://escholarship.org/uc/item/22262301

Author
Wu, Panruo

Publication Date
2016

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA
RIVERSIDE

Silent Data Corruption Resilient Matrix Factorizations on Distributed Memory System

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Panruo Wu

December 2016

Dissertation Committee:

Dr. Zizhong Chen, Chairperson
Dr. Laxmi Bhuyan
Dr. Rajiv Gupta
Dr. Zhijia Zhao

Copyright by
Panruo Wu
2016

The Dissertation of Panruo Wu is approved:

____

____

____

____

Committee Chairperson

University of California, Riverside

Acknowledgments

I am extremely grateful to my advisor, Dr. Zizhong Chen, for his patience, guidance, and generous support throughout my PhD years. I would like to thank him for leading me into the high performance computing field and the great help in crafting my career. Dr. Zizhong Chen has made my graduate school experience very rewarding by giving me intellectual freedom in my work, engaging me in new ideas, supporting my attendance at many conferences, and demanding high quality work. Additionally, I would also like to express my appreciation to Dr. Rajiv Gupta, Dr. Laxmi Bhuyan, and Dr. Zhijia Zhao for serving on my dissertation committee and providing valuable feedback in improving my research.

I am fortunate to have collaborated with many brilliant people. In particular I would like to thank Dr. Dong Li, Dr. Jeffrey Vetter for providing me the opportunity of an internship in Oak Ridge National Laboratory, and Dr. Qiang Guan, Dr. Nathan DeBardeleben for the internship in New Mexico Consortium. In addition, this dissertation work would not have been possible without the discussion, inspiration, support, and professional work of my collaborators Sean Blanchard, Jieyang Chen, Dingwen Tao, Xin Liang, Ouyang Kaiming, Sihuan Li, Chong Ding, Longxiang Chen, Teresa Davies, Christer Karlsson.

I am grateful for the funding sources that allowed me to pursue my graduate school studies: the National Science Foundation, Los Alamos National Laboratory,

and Oak Ridge National Laboratory, the Department of Energy, and the department of Computer Science and Engineering in University of California Riverside.

Finally, I would like to acknowledge friends and family who supported me during my time here. To name just a few, Longxiang Chen, Dr. Luming Liang and Chong Ding made my time in Colorado unforgettable. To Jianbo Ye, whom I have been a good friend with since in college, I wish our friendship will last forever. I am lucky to be friends with Zening Li, Peng Deng, and Linchao Liao and we had a lot of fun and memories. I owe a debt of gratitude to all members of UCR SuperLab. I would also like to thank the members of the basketball club we had over the years: playing basketball with you is so much fun. And thank you to Shuyue Wang, for all her friendship, love, and support which I can perfectly feel even across the big ocean between us! To my parents and extended family, my words cannot adequately express my gratitude: I hope I can make you proud of me.

To my parents for all the support.

ABSTRACT OF THE DISSERTATION

Silent Data Corruption Resilient Matrix Factorizations
on Distributed Memory System

by

Panruo Wu

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, December 2016
Dr. Zizhong Chen, Chairperson

The lack of efficient resilience solutions is expected to be a major problem for the coming exascale supercomputers, as the chance that a long running large scale computation can finish without faults is diminishing quickly. In this dissertation I try to develop algorithmic techniques to provide fault tolerance for the commonly used matrix factorization algorithms and its high performance implementation in distributed memory massively parallel systems, with very low overhead and high scalability.

Specifically, I design numerical error correcting encoding of matrix and the corresponding algorithms to tolerate hardware faults during matrix factorizations. It is in common with error correcting codes (ECC) used widely in communication and storage systems that use codes to detect and correct errors occured during communication or at rest in storage cells. The salient difference is that while ECC protects invariable data, I need an ECC for variable matrix that is under factorization. My previous

and current work covers the design of such algorithmic fault tolerance techniques for the six most widely used matrix factorizations LU, QR, Cholesky, SVD, Hessenberg reduction, and tridiagonal reduction which comprise the core functionality of the de facto dense linear algebra package ScaLAPACK (Scalable Linear Algebra PACKage). The novel approach I used extensively is the on-line ABFT which not only designs the numerical codes but also modifies the algorithm to maintain the checksum in flight. For LU/QR/Cholesky factorizations, the on-line transformation results in vastly improved fault tolerance at a small extra cost. For SVD/Hessenberg/tridiagonal factorizations where no ABFT exist, the on-line ABFT fills this void and produces similarly highly scalable, resilient, and efficient algorithms and implementations.

Contents

List of Figures **xii**

List of Tables **xv**

1 Introduction **1**
- 1.1 Problem Statement . . . 4
- 1.2 Thesis Statement . . . 5
- 1.3 Contributions . . . 5
  - 1.3.1 For one-sided matrix factorizations . . . 5
  - 1.3.2 For two-sided matrix factorizations . . . 7
  - 1.3.3 For fault tolerant high performance linpack . . . 7
- 1.4 Limitations . . . 8
- 1.5 Dissertation Organization . . . 8

2 Related Work **9**
- 2.1 Faults . . . 9
- 2.2 Checksum based algorithmic fault tolerance . . . 11
- 2.3 System level fault tolerance . . . 17
- 2.4 Energy efficiency and resilience . . . 19
- 2.5 Numerical issues . . . 22

3 Matrix-matrix Multiplication **23**
- 3.1 Failure Classification . . . 24
- 3.2 Error Detection, Location, and Correction . . . 25
  - 3.2.1 Fault tolerance for single error . . . 27
  - 3.2.2 Checksum relationship maintained during block outer product matrix multiplication algorithm . . . 28
  - 3.2.3 Threshold to distinguish roundoff error and soft error . . . 32
- 3.3 Performance Analysis . . . 34
  - 3.3.1 Overhead for encoding . . . 34

3.3.2 Overhead for computation . . . 35
3.3.3 Overhead for detecting errors . . . 36
3.3.4 Overhead for Recovery . . . 36
3.4 Experimental Evaluation . . . 37
3.4.1 The overhead of our on-line FT-DGEMM . . . 38
3.4.2 Performance comparison: TMR vs ABFT vs on-line ABFT . . . 42
3.4.3 Performance comparison: ATLAS DGEMM vs our on-line FT-DGEMM 47

4 One-sided Matrix Factorizations 48
4.1 Background . . . 50
4.1.1 ABFT . . . 51
4.1.2 Block algorithms . . . 53
4.2 On-line ABFT design framework . . . 55
4.2.1 A Separation: checksum and matrix . . . 56
4.2.2 Double checksums . . . 58
4.2.3 An example: on-line ABFT Cholesky factorization . . . 59
4.2.4 A framework to design on-line ABFT for block algorithms . . . 62
4.3 On-line ABFT enabled one-sided factorizations . . . 64
4.3.1 LU . . . 65
4.3.2 QR . . . 67
4.3.3 Partial factorization . . . 70
4.3.4 Duplicate to protect panel blocks . . . 74
4.4 Performance analysis and experimental evaluation . . . 75
4.4.1 Performance and scalability analysis . . . 75
4.4.2 Experimental evaluation . . . 79

5 Two-sided Matrix Factorizations 83
5.1 Background . . . 86
5.1.1 Checksums encoded matrix and its operations . . . 86
5.1.2 Error correction with multiple checksums . . . 87
5.1.3 Maintaining checksum for matrix factorizations . . . 89
5.1.4 Householder reflector for unitary diagonalization . . . 90
5.2 Unblocked version . . . 92
5.2.1 Bidiagonal reduction . . . 93
5.2.2 Hessenberg reduction and tridiagonal reduction . . . 96
5.3 Blocked version . . . 98
5.3.1 Bidiagonal Reduction . . . 100
5.3.2 Hessenberg and Tridiagonal Reduction . . . 102
5.4 Distributed Blocked Version . . . 103
5.4.1 Data distribution . . . 104
5.5 Analysis . . . 108
5.5.1 Fault coverage . . . 108

5.5.2 Overhead 112
5.6 Experiments 113
5.6.1 Fault injections 113
5.6.2 Execution time overheads 115
5.6.3 Overheads and scalability 116

6 Towards Practical Algorithm Based Fault Tolerance 119
6.1 Fault Model 121
6.2 The Checksum Scheme 122
6.2.1 Error patterns and correction 122
6.2.2 Checksum scheme in LU decomposition 127
6.2.3 The complete picture as in HPL 131
6.2.4 Round-off error bounds 136
6.3 Overhead, Performance, Scalability, and fault tolerance capability . . 138
6.3.1 Fault tolerance capability 138
6.3.2 Execution time overhead 140
6.3.3 Error correction overhead 141
6.3.4 Memory overhead 142
6.3.5 Impact on scalability 142
6.3.6 Tradeoffs between resilience and overhead 142
6.4 Experimental Study 145
6.4.1 Fault injection for fault coverage 145
6.4.2 Fault injection experiments 146
6.4.3 Overheads of fault free execution and error correction 148

7 Conclusions and Future Work 154
7.1 Conclusions 154
7.2 Future Work 156

Bibliography 158

List of Figures

3.1 Overall Overhead in percentage. 38
3.2 Overhead of online FTGEMM (matrix size: 10000) on ALAMODE. . 39
3.3 Overhead of on-line FTGEMM (matrix size: 15000) on ALAMODE. . 40
3.4 Overhead of on-line FTGEMM (matrix size: 10000) on MIO. 40
3.5 Overhead of on-line FT-DGEMM (matrix size: 15000) on MIO. . . . 41
3.6 Performance of different strategies (matrix size: 10000) on ALAMODE. 43
3.7 Performance of different strategies (matrix size: 15000) on ALAMODE. 44
3.8 Performance of different strategies (matrix size: 10000) on MIO. . . . 44
3.9 Performance of different strategies (matrix size: 15000) on MIO. . . . 45
3.10 Performance for different failure rate (matrix size: 10000). 45
3.11 Performance for different failure rate (matrix size: 15000). 46
3.12 Performance for different failure rate (matrix size: 10000). 46
3.13 Performance for different failure rate (matrix size: 15000). 47

4.1 Traditional ABFT matrix matrix multiplication and the separation of checksums with the matrices 57
4.2 **On-line ABFT Cholesky** (a) the snapshot of the second iteration in a right-looking block Cholesky factorization algorithm. (b) the first step in this iteration is unblocked Cholesky factorization on A_{11}. (c) the second step in this iteration is a triangular solve to update the panel matrix; the checksums can be updated accordingly. (d) the third step is matrix multiplication to update trailing matrix; the checksums can be updated accordingly. 63

4.3 **On-line ABFT LU** (a) a snapshot of one iteration in a right looking block LU factorization algorithm: before and after. (b) the first step in this iteration is to update the column panel matrix by unblocked LU; the checksums are updated to checksums of right factor U_{11}, U_{21}. (c) second step is to update the row panel matrix by triangular solve; the checksums of the panel matrix can be updated accordingly. (d) the third step is matrix multiplication to update the trailing matrix; the checksums of the trailing matrix can be updated accordingly. 66
4.4 **On-line ABFT QR** (a) a snapshot of one iteration in right-looking block QR algorithm: before and after. (b) first step is updating column panel matrix by unblocked QR; checksums are updated to the checksums of right factor R_{11}, R_{21}. (c) the second step is to derive factor matrix T (d) the third step is to update the trailing matrix by matrix multiplication; the checksums can be updated accordingly. 68
4.5 Run time for fault tolerant versions of LU, QR, and Cholesky in FT-ScaLAPACK and their original versions in ScaLAPACK. The fault tolerant versions can detect, locate, and correct one error in every row of every bolck on every processor at every iteration. On average, one iteration takes roughly one seconds in these experiments. 82

5.1 The effect of (left) Householder reflector to a column vector. 91
5.2 Householder Reduction to Bidiagonal Form 94
5.3 Householder reduction to Hessenberg form 97
5.4 Modification to distributed matrix-vector multiplication to include local checksums 106
5.5 Local checksum equivalent to global checksum with specific weights. . 107
5.6 The fault coverage for the three checksum schemes 115
5.7 The execution time overhead for various checksum schemes compared to original `DGEBRD` 116
5.8 Fault free execution time for fault tolerant FT-PDGEBRD and non-fault-tolerant PDGEBRD from ScaLAPACK. Local matrix is fixed at 1000×1000; the global matrix size scales as the number of processors scales. 118
5.9 Fault free execution time for fault tolerant FT-PDGEBRD and non-fault-tolerant PDGEBRD from ScaLAPACK. Processor grid is fixed at $1024(32 \times 32)$. The global matrix size scales as the local matrix size scales. 118

6.1 Error patterns for a single fault in matrix multiplication 123
6.2 Checksums for matrix multiplication 124
6.3 The checksum scheme that can tolerate single arithmetic fault or memory fault 126

6.4 Tiled right-looking LU algorithm, one iteration 128
6.5 Tiled right-looking LU algorithm with checksums, one iteration . . . 129
6.6 Tiled right-looking LU algorithm with checksums, one iteration. Shaded area are incorrect due to error propagation. Note that the affected checksums are also incorrect but the checksums are inconsistent therefore can be used to detect errors. 132
6.7 2D block cyclic matrix distribution. 134
6.8 One process view in a 4x4 process grid: PF stands for (left and top) panel factorization and TU for trailing matrix update. The red diamond represents checksum verification point. 144
6.9 The execution time of FAULT, FULL, OPT, and ORIG HPL with varying number of nodes as X-axis. Each node comes with 32 computing cores. 150
6.10 Fault free execution time for optimized fault-tolerant HPL. 153

List of Tables

4.1 The meaning of the variables in equation 4.4, according to [6] 76
4.2 The value of the factor C_f, C_v, C_m for LU, QR, and Cholesky factorizations in ScaLAPACK according to [6] 76

6.1 Fault coverage for different ABFT techniques. "Before" means the fault affects data that is produced but not yet used. "Middle" means the fault affects data that is undergoing repeated use. 146
6.2 Fault tolerant for dense linear algebra: costs and fault tolerance capability. "Yes" means the faults can be tolerated; "No" means otherwise. The percentage indicates the execution time overhead against non fault tolerant LU implementation (PDGESV/PDGESV in ScaLAPACK, HPL_pdgesv in HPL). 147

Chapter 1

Introduction

The extreme scale high performance computing (HPC) systems that are expected by the end of this decade poses several challenges including performance, power efficiency, and reliability. Due to the large amount of components in these systems,shrinking feature size, and the severe constrains in power, the probability that an extreme scale application experiences faults during its execution is projected to be non negligible. Resilience to faults have been widely accepted as critical for exascale HPC applications[94, 23, 13].

Faults are malfunctions of the hardware or software, and are the underlying causes for observable errors. When the fault does not interrupt the execution of a process the program can continue execution normally, but the results may be corrupted. Such silent data corruptions cannot be tolerated by checkpoint/restart (C/R) alone unless they can be frequently detected. Silent data corruptions may be the consequence of

soft faults caused by cosmic rays and radiation from packaging materials, and are usually one time events that corrupt the state of the machine but not its overall functionality. We restrict our scope to silent data corruptions (SDC) in this work. Note that since soft errors which are caused by single event upset frequently corrupt data silently, SDC handling is also often discussed in context of soft errors.

Faults in storage and communication systems are often effectively tolerated by error correction codes (ECC) because the data stored or communicated are not changing. However, faults in logic units that transform the data are harder to detect and tolerate. Typically some kind of double modular redundancy (DMR) is needed to detect soft faults in logic units and triple modular redundancy (TMR) is needed to tolerate SDCs. Although modular redundancy requires at least 100% resource overhead and often incurs significant execution time overhead, it is sometimes the only general system level solution to tolerate SDCs [39, 96].

System level SDC solutions can be prohibitively expensive for HPC systems. An alternative solution is to implement fault tolerance in applications, which can take advantage of the semantics and structure of a specific application resulting in much lower cost. Algorithm based fault tolerance (ABFT) represents a middle ground between application specific fault tolerance and architecture fault tolerance. At one end application specific fault tolerance is highly diverse that often require ad-hoc solutions, at the other end system fault tolerance is general but too costly and unscalable.

Algorithms thus presents just enough semantics to take advantage and structure to be generally useful.

ABFT has first been proposed in a seminal work by Huang and Abraham [58] for matrix-matrix multiplication on systolic arrays. The idea of ABFT can be seen as an adaption of ECC to numeric structures like matrices or vectors. The significant difference is that for ECC the data is static but for ABFT the data is under transformation. In ABFT the central problem is that the codes must maintain after transformation in order to be able to detect errors using the codes. We will focus on the most important operations in numerical linear algebra: the LU factorization, QR factorization, Cholesky factorization, and unitary bidiagonal/tridiagonal/Hessenberg reduction. The (partial pivoted) LU factorization is *the* algorithm to solve general linear system which has applications in all science and engineering problems. QR factorization is often used to solve least square problem which is important in statistics. Cholesky factorization is used to solve the symmetric linear system. Unitary bidiagonal reduction is the prerequisite to Singular Value Decomposition (SVD) that is used in rank-deficient linear square problems and linear systems and certain data analytics problems such as Principal Component Analysis (PCA) and Linear Discriminant Analysis (LDA). The Hessenberg and tridiagonal reductions are prerequisites for eigenvalue decomposition that have many applications in structural engineering (vibration modes), quantum mechanics (Shrodinger's equation), and in data analytics (PageRank from Google search). In summary, these matrix factorizations are the

core computational tools in linear algebra that provide the basic building blocks for computational science and engineering, and more recently the big data analytics.

1.1 Problem Statement

The core problem in this proposal is to design efficient and scalable distributed memory one-sided and two-sided factorizations that are resilient to silent data corruptions. To make the statement more concrete some definition and clarification is in order. Efficient and scalable means that in general: 1) fault-free execution time should incur overhead less than 10%; 2) memory overhead should be less than 10%; 3) the resilient algorithm should be at least as scalable as the original counterparts. The algorithms and implementations are for large scale distributed memory computers that communicate by message passing. One-sided matrix factorizations include LU, QR, and Cholesky factorizations; two-sided matrix factorizations include bidiagonal, tridiagonal, and Hessenberg reductions, as they are the main computational subroutines in the ScaLAPACK (Scalable Linear Algebra PACKage). Silent data corruptions include bit flips in memory systems or in communications and computation faults in arithmetic logic unit, in particular the FPU.

1.2 Thesis Statement

Linear matrix encoding and appropriate modification to the matrix factorizations can enable timely detection and correction of silent data corruptions with minor overheads.

1.3 Contributions

1.3.1 For one-sided matrix factorizations

In this paper, we present the design and implementation of FT-ScaLAPACK, a fault tolerant version ScaLAPACK that is able to detect, locate, and correct soft errors in Cholesky, QR, and LU factorizations on-line in the middle of the computation in a timely manner before the errors propagate and accumulate. FT-ScaLAPACK has been validated with thousands of cores on Stampede at the Texas Advanced Computing Center. Experimental results demonstrate that FT-ScaLAPACK is able to achieve comparable performance and scalability with the original ScaLAPACK library. More specifically, our contributions include:

- **Cholesky Factorization:** We designed an on-line scheme to correct soft errors in Cholesky factorization before the errors propagate and accumulate, where the existing best schemes [56, 3] cannot correct errors. Existing schemes need to

restart the whole computation if any error occurs, therefore, introduces much higher overhead than our on-line scheme.

- **QR Factorization:** We designed an on-line scheme to correct soft errors in QR factorization before the errors propagate and accumulate, where the existing best schemes [35, 26] can only correct errors off-line at the end the computation after the errors propagated and accumulated. While the overhead of the existing off-line schemes increases at least quadratically as the number of errors increases, the overhead of our on-line scheme is much lower and increases only linearly.

- **LU Factorization:** We designed a new on-line scheme to correct soft errors in LU factorization without global communications or synchronizations, where the existing best schemes [19] are on-line, but involve expensive global communications and synchronizations.

- **Software Implementation:** We made the widely used ScaLAPACK library core routines (Cholesky, QR, LU) fault tolerant without modifying the library interfaces. Existing HPC applications that use ScaLAPACK library can now make use of our new FT-ScaLAPACK library to tolerate soft errors by just linking to the new library without any modification on source codes.

1.3.2 For two-sided matrix factorizations

- We propose the first comprehensive online ABFT schemes against soft errors for three two-sided factorizations.

- We analytically and empirically evaluate the fault coverage and efficiency of the proposed scheme and demonstrate the superiority to the current state of the art.

- We implemented the proposed technique in the Scalable Linear Algebra Package (ScaLAPACK) for easy adoption without changes to the interfaces.

1.3.3 For fault tolerant high performance linpack

New fault model We use a fault model that allows logic faults and memory system faults that are comprehensive temporally and spatially and design ABFT schemes that can effectively detect and correct errors caused by these faults.

New checksum scheme We propose a novel process local checksum scheme, multiple checksums for error detection and correction by studying the syndrome (error patterns) caused by the faults.

Validation and software implementation We test and validate the resilience using an architectural fault injector. We implement the new ABFT schemes in the latest Netlib HPL-2.1.

1.4 Limitations

The main limitation of this dissertation work is the fault model, which we generally assumed to be transient floating point arithmetic error, and expanded to include memory bit flips errors in chapter 6. It is currently unclear to what extent our fault model captures the actual faults. However this dissertation should provide foundations for the design space to accomondate different fault models that may proved to be more fidel.

1.5 Dissertation Organization

The rest of this dissertation is organized as follows. Chapter 2 gives a brief review of the background as well as the related work. Chapter 3 discusses the design of online ABFT for matrix multiplications. Chapter 4 develops on-line checksum schemes for one-sided factorizations. Chapter 5 further develops on-line checksum schemes for two-sided factorizations based on Householder transformations. Chapter 6 presents variants of checksum schemes for High Performance Linpack (HPL), a highly scalable and performant implementation of LU factorization routinely used to rank the largest supercomputers in the world (TOP500.org), which works for a more comprehensive fault model. Chapter 7 concludes the dissertation and discusses future work.

Chapter 2

Related Work

In this section we put our work in context by surveying the field, especially the work closest to this dissertation. Due to the complexity and high performance demand of high performance computing, there is no single solution to the reliability problem. There are techniques at differetn layers of the computer system stacks that mitigate this problem, ranging from the lowest circuit layer to the application specific layer. This dissertation work on the algorithm layer, which is right beneath the application layer that is at a advantageous position to have just enough semantics for efficient fault tolerance and also generality to service a range of applications.

2.1 Faults

Soft errors: The first report on soft errors due to alpha particles in computer chips was from Intel in 1978 [70]. The first report on soft errors due to cosmic

radiations in computer chips was in 1984 [108]. In 1996, Norman [79] studied error logs of several large computer systems and reported a number of incidents of cosmic ray strikes. In 2005, Hewlett-Packard admitted that the ASC Q supercomputer located in Los Alamos National Laboratory experienced frequent crashes because of cosmic ray strikes to its parity protected cache tag arrays. The machine is particularly susceptible because of the 7000ft altitude of the installation location [72]. The book by Mukherjee [77] surveys extensively the architectural techniques to design architectures for soft errors.

Memory System: The most commonly used and effective protection for memory system is the error detecting and/or error correcting codes used in storage system and communication system. Such codes range from simple parity code to Hamming codes [50] or Hsiao codes [55] that provide single bit correction and double bit detection (SEC-DED) capability. To correct multi-bit errors more complex and expensive codes such as double-bit-error-correcting and triple-bit-error-detecting (DEC-TED) [88], and Reed-Solomon (RS) codes [85]. The main problem of multi-bit correcting codes are the cost in increased circuit, storage, and computing overhead [89]. To handle memory chip failures chipkill-correct codes are adopted [64]. ECC alone cannot ensure that correct execution of applications as the application execution utilizes not only memory system but also computation logics.

Computational Logic: As the chip technology goes into ever smaller feature size, logic errors will be major problem [73, 74]. At the circuit level latches redundancy

based techniques are proposed but suffer from large area time overhead [68, 75, 84]. At the architectural level modular or execution redundancy is used to provide fault tolerance [77]. Coding approach is also devised for some arithmetic operations, such as AN code [69] usually for integer addition. Although residue code is not directly applicable to floating point arithmetic it can be used in different stages independently [61]. Integer arithmetic can be used to detect errors in floating point arithmetics concurrently such as demonstrated in [105]. Code-based methods are cost effective but inflexible to apply to a wide range of hardware structures. More flexible methods include replicating the execution of some logic units and verifying the result such as in IBM systems [71].

2.2 Checksum based algorithmic fault tolerance

The term algorithm based fault tolerance (ABFT) was proposed by Abraham and Huang [58] for matrix-matrix multiplication on systolic arrays to detect and correct transient errors that have occurred during the computation. The technique was subsequently extended to cover one-sided matrix factorizations (LU, Cholesky, and QR) by [65, 67, 2]. Later developments extend the idea to handle fail-stop errors [27, 9], to tolerate multiple fail-continue errors (soft errors) for LU [31] and QR [34]. For soft errors these developments share the common characteristics that they are offline problem specific. Offline means the error detection and correction happen after the

computation is done; problem specific means the technique works regardless of which specific algorithm is used to solve the matrix factorization problems.

Online, algorithm specific ABFT: To handle multiple soft errors more efficiently, recent research proposed online variant of the ABFT idea on matrix-matrix multiplication [100, 101] one-sided matrix factorizations [20, 99, 103]. The characteristics of these recent developments are that the techniques are online algorithm specific, as opposed to the offline problem specific discussed in the previous paragraph. Online means that the error detection and correction is continuously working during the computation instead of only at the end; algorithm specific means the ABFT scheme has to be designed with a particular matrix factorization algorithm in mind instead of algorithm agnostic.

Previous ABFT work in two sided factorizations: As for two sided matrix factorizations (Hessenberg, tridiagonalization, bidiagonalization), two recent studies [59, 60] discussed ABFT scheme for Hessenberg factorization against fail-stop errors and bidiagonalization against fail-continue errors. These two works are based on matrix-matrix multiplication and ignore other operations in the factorizations thus only provide partial fault coverage. To provide comprehensive protection for the two sided matrix factorizations, the challenge is that there is no obvious problem specific checksum scheme to base on. It is therefore not obvious to design offline ABFT for two-sided matrix factorizations; algorithm specific online ABFT is necessary. This

further entails that substantial algorithm modification must be incorporated together with the checksum scheme, which will be shown in the following sections.

ABFT and fault model: Algorithm based fault tolerance was first proposed by Abraham and Huang [58]. The original ABFT was proposed for matrix multiplication and LU on systolic arrays for real time signal processing. The fault model used is logic faults that produces erroneous results. Storage cell faults such as in memory, latch, and registers are assumed to be handled by traditional error correction codes. In matrix multiplication, as a single arithmetic fault causes only a single error in the result matrix, this ABFT scheme can effectively detect and correct it. In LU decomposition, because of error propagation, a single fault will cause an overwhelmingly large amount of errors in the results, thus making this ABFT scheme unable to tolerate a single fault algorithmically. The limited correction capability is due to three factors: 1) inability to tolerate multiple errors in the checksum scheme, 2) massive error propagation in matrix triangularization, and 3) offline error correction. These three factors conspire to make algorithmic error correction difficult in matrix triangularization. Later Luk and Park [67] described an elegant analytical model for ABFT in matrix triangularization. The analytical model assumes an abstract fault model that a transient error occurs at some intermediate iteration in the triangularization. Even though the single error will propagate in later stages and become uncorrectable at the end, it can be shown that the error can be cast back as a single rank perturbation to the original input matrix, much like the widely used backward

error analysis [98]. Then assuming two row checksums the correct result can be derived based on the backward fault model. This is a powerful technique that avoids the error propagation problem but it has three limitations: 1) the fault model assumes single error not necessarily single fault, as we have seen that single fault may cause multiple errors; 2) this checksum scheme has no column checksums thus may fail to even detect certain faults as pointed out by a recent work by Yao [104]; and 3) the method can only tolerate at most one fault during the decomposition. As the scale of supercomputing marches towards exascale, fault tolerance is becoming a key aspect in achieving the required performance at reasonable cost [93, 24, 13]. And assuming only one fault during the application run seems not appropriate in future large scale systems any more. To address more than one error in matrix triangularization, Du [33, 29] proposed a technique to tolerate two errors in solving linear system using partial pivoting LU decomposition. In this case, the decomposition cannot be corrected, but the result to the linear system can be recovered using the Sherman-Morrison-Woodbury formula. Handling beyond two errors would be more expensive than the LU decomposition itself. The fault model used is the same as in Luk and Park [67] thus suffers from the same problem. Some researchers went in another direction in order to tolerate more faults effectively. Realizing that the offline approach taken by the traditional ABFT techniques have to face catastrophic error propagation at the end, researchers attempted to adapt checksum schemes for online error detection and correction [21, 100, 99]. The idea is that online ABFT catches

errors early on when they are not propagated far away, therefore making it easier to correct. Online ABFT also can tolerate more errors that spread in time by avoiding errors compounding each other. The fault model used however is still arithmetic faults, and there still is no column checksums due to the difficulty in row pivoting. A recent study [104] discovers that the fault models used in the previous ABFT works are not adequate even in detecting faults (Section 3 in [104]). This work proposes a global row and column checksums that can effectively detect errors and it is also an online approach. However error correction is not considered.

In chapter 6 we do not use an abstract fault model; rather we assume an architectural fault model and aim to detect and correct multiple errors. The architectural fault model is closer to what happens in real world and not only include all the fault models discussed above but also more improvements.

The pioneering works of on-line ABFT on matrix-matrix multiplication [15, 102] and LU factorization in HPL [19] are the main inspiration of our work. While those works went at length to design, validate, and analyze the checksum schemes for the specific algorithms they aimed at, we used a unified approach guided by a high level framework that we show can make the design, validation, and analysis of on-line ABFT checksum schemes with block linear algebra algorithms almost mechanical. We think this higher level viewpoint helps mitigating the disadvantage of ABFT that major efforts have to be made to design on-line ABFT for each algorithm. By making on-line ABFT design systematic and easier, we hope to see more and more commonly

used algorithms and other fault tolerance techniques such as checkpoint/rollback, ECC [63], and hybrid memory systems using non-volatile memories enjoying the benefits of on-line ABFT.

Offline ABFT methods usually can detect errors and correct up to a certain number of errors using some formulas after the operation [66]. Although offline ABFT techniques usually have very small runtime overheads [30], they cannot stop errors from propagation. Instead, they try to recover from the mess after the errors spread, which limits the number and type of SDCs that can be tolerated. Our approach on the other hand, tries to contain the errors from propagating thus is more powerful in tolerating more errors at the cost of slightly larger runtime overhead.

The state-of-the-art algorithmic technique for tolerating soft/hard errors are extensively studied in the recent PhD thesis by P. Du [26] ABFT based approach to survive hard errors (fail-stop) for LU, QR, and Cholesky factorizations was well studied in [28]. But to survive soft errors in the matrix factorization subroutines, the jobs are arguably more difficult. Some study on LU [30] and QR [35] exist to deal with soft errors. The technical report by Du [30] proposes a technique to tolerate soft errors in LU, which is based on a mathematical model of treating soft error during LU factorization as rank-one perturbation to the original matrix and recovering the solution of $Ax = b$ with the Sherman-Morrison [43] formula. Although our one-sided factorizations we are dealing with here and the LU factorization discussed in paper [30] have the same goal, our approach is different in many fundamental aspects.

Firstly while their work guarantees the solution x to the linear system $Ax = b$ is correct, we aim at ensuring that the factorization of $A = LL^T, A = LU, A = QR$ is correct. Secondly, their work casts soft errors occur during the factorization process to the original matrix thus avoiding consideration of timing of errors, our work deals directly with errors as they occur, as the factorization proceeds. Thirdly, after making some tradeoffs their works can effectively tolerate two soft errors in each block with minimum overhead, our approach has the potential to tolerate significantly more soft errors at the expense of a little bit more overhead. Lastly, unlike LU decomposition, Cholesky factorization can easily break down if soft errors happen to invalidate the positive definiteness of the matrix, in which case all off-line approaches would fail since the Cholesky factorization would simply not finish.

2.3 System level fault tolerance

There are other fault tolerant techniques that can deal with SDCs in parallel factorizations that have various characteristics in terms of error detection and correction capability, runtime overhead and resources overhead. Among them, we compare our approach to node (MPI task) level TMR (RedMPI [40]). So far the RedMPI approach is the most general and powerful method to detect and tolerate silent (soft) errors. RedMPI uses multiple "replicas" for each MPI task, and check the MPI messages from the "replicas" to try to detect and recover silent errors. The idea is that, a SDC in one replica will eventually produce a corrupted MPI message, and that message

will be detected by comparing the messages from all the replicas for the same MPI task. Therefore, by checking the MPI message we can detect and tolerate SDCs. According the the paper [40], RedMPI introduces overhead between 20% to 60% for triple redundancy, and 13% to 45% for double redundancy, depending on applications. To detect error at least double redundancy is required, which means 2x nodes are required while to correct errors by voting at least triple redundancy or 3x nodes are required. Our approach is not as general and requires adaptions for each algorithm considered, but has much less overhead both in run time and node numbers.

Besides the redundancy based fault tolerance, another popular fault tolerance technique is the checkpointing and rollback (C/R). A very comprehensive survey of C/R can be found in [36]. The intuition of the C/R idea is to periodically save the execution state into stable storage and rollback to the saved state should a failure occurred. Checkpointing can be either initiated by the system transparently to the application such as in BLCR [52], or initiated by the application with the latter being more popular due to its simplicity, flexibility, and better performance. The bottleneck of a disk-based C/R scheme is usually the I/O system. It also severely limits the scalability of the C/R system. One particular competitive C/R variant is diskless checkpointing [82] such as SCR [76] that uses another node's memory as the checkpointing destination. Diskless checkpointing usually is much faster and scalable than disk checkpointing at the expensive of massive memory space overheads.

2.4 Energy efficiency and resilience

Architectural works: These works [87, 86] attempts to break the contract between hardware and software for the sake of energy efficiency: the software can no longer expect all operations in hardware are correct. The solutions to the loss of reliability promise is to redesign applications to use stochastic and self-stabilizing algorithms [90, 87] that are resilient to hardware faults.

System fault tolerance: The most commonly used (fail-stop) fault tolerant techniques, checkpoint/rollback has been studied very extensively in the literature. The most popular approach is application driven checkpointing, where the programmer defines the state to checkpoint and when to checkpoint. System level checkpoint Representative works include the Berkeley Lab Checkpoint/Restart (BLCR) [52], diskless checkpointing [82, 81], hierachical checkpointing (SCR [76], FTI [4]). Popular distributed memory parallel computing middleware such as MPI and Charm++ all support checkpointing. Note that checkpointing by itself can only tolerate fail-stop errors. From the perspective of energy efficiency, diskless checkpointing clearly wins for its much lower performance overhead and better scalability.

Another general approach that can tolerate both fail-stop and fail-continue failures is modular redundancy, or called replication. The idea is to use redundant resources for data and computation. Representative works include rMPI [38] and RedMPI [41] which replicates MPI ranks. From the perspective of energy efficiency replication

seems hardly justified for its large resource thus energy overhead (100% for detection, 200% for correction).

Algorithmic fault tolerance: In recent years tremendous progress has been made in the area of algorithmic fault tolerance mainly for its clever use of algorithmic structure to achieve very low overhead and very good scalability. Here we only sample some of the works. For dense linear algebra, encoding matrices with checksums has been studied in matrix-matrix multiplication [58] and matrix factorizations [2, 67, 65] mainly on systolic arrays. Recently, fueled by the looming reliability issues in large scale supercomputers, those checksum-based algorithmic fault tolerance has attracted extensive research in the context of HPC for both fail-continue failures [99, 103, 20, 31] and fail-stop failures [22, 27]. For sparse iterative methods, some algorithms exhibit inherent fault tolerance [11, 14, 87], and some algorithms can be efficiently augmented using algorithmic detection and/or correction [16, 91, 92]

Energy/power and reliability: Extensive research has been performed to save energy and preserve system reliability for real-time embedded processors and systems-on-chip. Zhu *et al.* [107] discussed the effects of energy management via frequency and voltage scaling on system failure rates. This work is later extended to reliability-aware energy saving scheduling that allocates slack for multiple real-time tasks [106], and a generalized Standby-Sparing technique for multiprocessor real-time systems, considering both transient and permanent faults [48]. These studies made some assumptions suitable for real-time embedded systems, but not applicable to large-scale

HPC systems with complex hardware and various types of faults. Pop *et al.* [83] explored heterogeneity in distributed embedded systems and developed a logic programming solution to identify a reliable scheduling scheme that saves energy. This work ignored runtime process communication, which is an important factor of performance and energy efficiency for HPC systems and applications. The Razor work [37] implemented a prototype 64-bit Alpha processor design that combines circuit and architectural techniques for low-cost speed path error detection/correction from operating at a lower supply voltage. With minor error recovery overhead, substantial energy savings can be achieved while guaranteeing correct operations of the processor. Similar power-saving and resilient-against-error hardware techniques have been proposed such as Intel's Near-Threshold Voltage (NTV) design [62] on a full x86 microprocessor.

For processors traditionally they are designed for the worst-case which are expensive in terms of area and power. Better than worst case (BTWC) design approaches based on timing speculation (TS) [37], are for average case and also have high yields. Such TS based methods generally allow timing violations to occur and try to detect and correct timing errors. The basic idea behind [37] is to supplement critical flip-flops with a shadow latch that strobes the output of a logic stage at a fixed delay after the main flip-flop. If a timing violation occurs, the main flip-flop and the shadow flip-flop will have different values and the timing error is detected. The correction involves recovering the correct value stored in the shadow flip-flop. The limitations of

Razor method is the requirement on the circuit delay behaviors. The Razor method thus uses circuit level fault tolerance to compensate the unreliability.

2.5 Numerical issues

As the floating point arithmetic is inexact, there is necessarily a threshold problem when verifying the checksum, as the checksum and the data would not agree exactly. How much deviation is accepted as a pass? There are two strategies in general: 1) conservative strategy that derives a priori error bounds for the algorithm; 2) optimistic strategy that takes into account the probablistic distribution of floating point numbers and arithmetic. The conservative strategy as discussed in section 3.2 where the maximum possible rounding error is derived. Any discrepancy larger than the bound necessarily indicate an error but the error bound can be too pessimistic in practice. The optimistic strategy such as shown in [10] is a more optimistic error bound that bounds the rounding error in high probability. This results in a much tighter bound that mandates more strict error checking. The optimistic bound depends on an interesting property of floating point arithmetic that tends to generate numbers following the inverse logrithm distribution [51].

Chapter 3

Matrix-matrix Multiplication

Matrix matrix multiplication is a widely used operation in science and engineering. ABFT has been extended to correct soft errors in the solution of system of linear equations . using Sherman-Morrison formula. While the proposed approach can be treated as an extension of ABFT, The most prominent difference of the proposed approach is that it tolerates soft errors **on-line**, which means soft errors are detected, located, and corrected in the middle of the computation during the program execution.

The on-line property of our approach introduces lower fault tolerance overhead and allows better reliability and flexibility. In our on-line approach, soft errors can be detected and located before they propagate. Corrupted computations can be stopped or corrected in the middle of the program execution in a timely manner. Therefore, computation efficiency can be improved significantly. Furthermore, the frequency

of the error detection can be flexibly adjusted according to the failure rate of the computing platform.

The idea of our online fault tolerance is not limited to matrix matrix multiplication. It can also be applied to other commonly used linear algebra subroutines. Matrix matrix multiplication is of special interest because it is the simplest case and it is widely used in other matrix operations. Linear algebra packages such as `LAPACK` and `ScaLAPACK` rely heavily on matrix matrix multiplication to achieve high performance. Many scientific applications spend majority of their execution times on such common linear algebra operations. Therefore protecting such most frequently used subroutines from soft errors can provide significant degree of fault tolerance for the whole application.

Experimental results demonstrate that the proposed technique can correct one error every minute with negligible (i.e., less than 1%) performance penalty over the ATLAS `dgemm()`.

3.1 Failure Classification

When a failure occurs during an application execution, if the failed process continues working, we define the failure as fail-continue failure. Fail-continue failures include computation errors by logic circuit and bit-flips in memory. They may be caused by many reasons including alpha particles from package decay, cosmic rays,

thermal neutrons, and random noises. Fail-continue failures do **not** interrupt the program execution. But the computation results can not be trusted any more.

Soft errors are one-time events that corrupt the state of a computing system but not its overall functionality. When a soft error occurs, it may or may not cause the crash of the system. Therefore a soft error may or may not be a fail-continue failure.

In this paper, we restrict our scope to fail-continue soft errors.

3.2 Error Detection, Location, and Correction

Data redundancy can be exploited to tolerate faults against memory errors and CPU logic errors. In this paper we implement the data redundancy through checksum matrices. For description we introduce some notions and terms of checksum matrices. A column checksum matrix of matrix A, denoted by A^c, is defined by $A^c := \begin{pmatrix} A \\ v^T A \end{pmatrix}$. The checksum vector v is typically set as a all-one column vector. Similarly, A row checksum matrix of matrix B is defined as $B^r := \begin{pmatrix} B & Bw \end{pmatrix}$, where w is a column checksum vector. Finally the full checksum matrix of matrix C is $C^f := \begin{pmatrix} C & Cw \\ v^T C & v^T Cw \end{pmatrix}$.

Instead of multiplying matrices A by B we use their checksum versions, and instead of obtaining $C := AB$ we get its full checksum version $C^f := (AB)^f$.

$$\begin{aligned} A^c \times B^r &= \begin{pmatrix} A \\ v^T A \end{pmatrix} \times \begin{pmatrix} B & Bw \end{pmatrix} \\ &= \begin{pmatrix} AB & ABw \\ v^T AB & v^T ABw \end{pmatrix} =: (AB)^f = C^f \end{aligned}$$

This extra information of the multiplication result can be used to detect, locate and possibly recover faults occurred during computation. If a soft error occurred, either because of memory fault or CPU computation fault, the faulty part will violate the checksum relationship of the result. If only one error occurred, exactly one row and one column of the result will not satisfy the checksum matrix definition. This relationship can be used to detect and locate the error. Furthermore, by solving a linear equation(s) it is possible to recover the faulty entry in C.

Our improvement over the traditional ABFT matrix multiplication is, by using the outer product version of matrix multiplication algorithm, we revealed that there's enormous opportunities during the operation to do the above checking and recovering. Instead of checking only at the end of whole matrix computation we can do the checking many times as the computation progresses, which greatly improve the ability to detect and correct errors. Moreover, the frequency of checking during computations

is flexible; it's possible to adjust it in accordance to the expected error rates of a computer.

The detailed method will be discussed in the following subsections.

3.2.1 Fault tolerance for single error

To tolerate single error we assume checksum vectors v, w to be all-one vector. Thus in a full checksum matrix C^f the sum of the each rows of C are stored in the extra column, and the sum of each columns of C are stored in the extra row of C^f. For brevity we assume A, B are n by n square matrix. The checksum relationship can be mathematically expressed as:

$$c_{i,n+1} = \sum_{j=1}^{n} c_{i,j}$$
$$c_{n+1,j} = \sum_{i=1}^{n} c_{i,j}$$
$$c_{n+1,n+1} = \sum_{i,j=1}^{n} c_{i,j}$$

If one entry of matrix C, say $c_{i,j}$, is corrupted, it's easy to locate the fault by examining the above checksum relationship, in which case exactly one row i and one column j will fail the examination above. Once the fault is detected we may use either the row

sum or column sum to correct the faulty entry $c_{i,j}$ by:

$$c_{i,j}^{\text{correct}} = c_{i,n+1} - \sum_{j=1,j\neq i}^{n} c_{i,j}$$
$$c_{i,j}^{\text{correct}} = c_{n+1,j} - \sum_{i=1,i\neq j}^{n} c_{i,j}$$

To tolerate multiple errors in C more sophisticated checksum vectors should be used. Readers are referred to paper [57] for details. However in our approach we are able to tolerate errors during computation so we don't need tolerating multiple errors in a single phase. We thus stick to the simple scheme outlined above.

This scheme of encoding A, B, C into checksum matrix A^c, B^r, C^f can only tolerate faults in C. If all A, B, C are to be protected we may encode both A, B into its full checksum matrix, and only use their partial checksum matrix(row checksum matrix for B and column checksum matrix for A) in multiplication. Then all A^f, B^f, C^f should go through the checksum relationship examination.

3.2.2 Checksum relationship maintained during block outer product matrix multiplication algorithm

Traditional ABFT matrix multiplication algorithms only check the checksum relationship (detecting, locating and correcting faults) at the end of the whole multiplication process. By employing block outer product matrix multiplication algorithm we could do as much as n times fault tolerance during the multiplication process. The

point of our scheme is that the checksum relationship is maintained during block outer product matrix multiplication, or more exactly at the end of each outer iteration of the block outer product algorithm.

We first define outer product version of matrix multiplication algorithm.

Algorithm 1 Outer product matrix multiplication

```
Require: A, B
Ensure: C ← A × B
C ← 0
for s = 1 → n do
    C ← C + A(1 : n, s) × B(s, 1 : n)
end for
output C
```

The validity of this matrix multiplication algorithm can be easily verified by decomposing A, B into column and row blocks.

$$AB = [A_1, \ldots, A_n] \times [B_1, \ldots, B_n]^T$$

$$= A_1B_1 + \ldots + A_nB_n$$

In the above outer product algorithm if we input checksum matrices A^c, B^r, we prove that at the end of every iteration(after each rank 1 update) the partial result C which we denote by $C_s(s = 1, 2, \ldots, n)$ is a full checksum matrix. Let $A_s := A(1 : n, 1 : s)$ and $B_s := B(1 : s, 1 : n)$, then

$$
\begin{aligned}
C_s &= A_s^c \times B_s^r \\
&= \begin{bmatrix} A_s \\ v^T A_s \end{bmatrix} \times \begin{bmatrix} B_s & B_s w \end{bmatrix} \\
&= (A_s B_s)^f
\end{aligned}
$$

The equation above shows that the partial product C_s is a full checksum matrix, which makes it possible to do fault tolerating at the end of every iteration s.

In practice implementations of matrix multiplication using outer product multiplication are unlikely to be efficient. In order to make better use of cache system of a computer we need to use blocking algorithms. The modified version of outer product multiplication(the block outer product algorithm) is to do a rank k update each iteration instead of a rank 1 update. For interface compatibility we use the same convention as in BLAS in which $op(A)$ means A or A^T depending on parameters passed onto the subroutine.

Algorithm 2 Blocked outer product matrix multiplication

Require: A, B, C, α, β
Ensure: $C \leftarrow \beta C + \alpha op(A) \times op(B)$
for $s = 1; s \leq \lfloor \frac{n}{k} \rfloor k; s \leftarrow s + k$ **do**
  $C \leftarrow \beta C$
  $C \leftarrow C + \alpha op(A)(1 : n, s : s + k - 1) \times op(B)(s : s + k - 1, 1 : n)$
end for
if $s \leq n$ **then**
  $C \leftarrow C + \alpha op(A)(1 : n, s : n) \times op(B)(s : n, 1 : n)$
end if
output C

Recalling that checksum relationship is closed under addition, similar to the outer product algorithm the partial result after each iteration in the corresponding block version above is also a full checksum matrix. Therefore we have opportunities to insert checking procedures into the algorithm, and checks the checksum relationship regularly at the frequency as we see fit. This results in the online version of ABFT matrix multiplication algorithm:

Algorithm 3 On-line ABFT matrix multiplication

Require: A, B, C, α, β
Ensure: $C \leftarrow \beta C + \alpha op(A) \times op(B)$
encode $op(A), op(B), C$ as $op(A)^c, op(B)^r, C^f$
for $s = 1; s \leq \lfloor \frac{n+1}{k} \rfloor k; s \leftarrow s + k$ **do**
  if s reaches the point we need a check **then**
    verify the *checksum relationship* of C^f
    if checksum relation *does not* hold **then**
      if only one line(i) and one column(j) fails the checksum verification **then**
        recover the faulty entry $C^f_{ij} \leftarrow C^f_{i,n+1} - \sum_{k \neq i}^{n} C^f_{i,k}$
      else
        recompute $C^f = \beta C^f + \alpha op(A)^c * op(B)^r$
      end if
    end if
  end if
  $C^f \leftarrow C^f + \alpha op(A)^c(1 : n+1, s : s+k-1) \times op(B)^r(s : s+k-1, 1 : n+1)$
end for
if $s \leq n + 1$ **then**
  $C^f \leftarrow C^f + \alpha op(A)^c(1 : n+1, s : n+1) \times op(B)^r(s : n+1, 1 : n+1)$
end if
output C as $C^f(1 : n, 1 : n)$

While the Our algorithm introduces much better reliability and flexibility than the original ABFT matrix multiplication algorithms.

3.2.3 Threshold to distinguish roundoff error and soft error

In our algorithms we need to verify the checksum relationship, which requires adding a row or a column of entries and comparing their sums against the entries in the last row or column in C_f. Because of roundoff errors in floating point computations we cannot expect the checksum relationship to hold exactly. Then how to distinguish roundoff errors from soft errors becomes an issue. Apparently manually

enforcing a threshold and assuming computation correct if differences are less than that threshold is unreliable. Too large a threshold may hide soft errors while a too small one may interrupt correct computations. Therefore it's helpful to develop a reasonable threshold based on concrete analysis.

A well known bound of matrix product roundoff error [45] is

$$||fl(AB) - AB||_\infty \le \gamma_n ||A||_\infty ||B||_\infty \tag{3.1}$$

where $\gamma_n = \frac{nu}{1-nu}$, and u is the unit roundoff error of the target machine, and n is the common dimension of the matrix A and B. We begin by assuming the computations are correct, i.e. $C^f = A^c B^r$. As a convention the floating point version of a variable has a hat over the corresponding name. Then we have

$$\begin{aligned}
\left|\sum_{j=1}^{n} \hat{c}_{ij} - \hat{c}_{i,n+1}\right| &= \left|\sum_{j=1}^{n}(\hat{c}_{ij} - c_{ij}) - (\hat{c}_{i,n+1} - c_{i,n+1})\right| \\
&\le \left|\sum_{j=1}^{n}(\hat{c}_{ij} - c_{ij})\right| + |(\hat{c}_{i,n+1} - c_{i,n+1})| \\
&\le ||fl(C^f) - C^f||_\infty \\
&\le \gamma_n ||A^c||_\infty ||B^r||_\infty =: \lambda
\end{aligned}$$

Therefore if the difference of the computed result satisfies $|\sum_{j=1}^{n} \hat{c}_{ij} - \hat{c}_{i,n+1}| \le \lambda$ it's reasonable to claim that no errors other than roundoff errors have occurred.

Otherwise we should regard it as a failure of checksum relationship examination and do the fault tolerating procedure.

Similarly for the column sum there is a corresponding constant that serves as the threshold which is $\mu := \gamma_n ||A||_1 ||B||_1$.

3.3 Performance Analysis

To quantify the efficiency of our approach, in this section, we analyze the overhead introduced by our fault tolerance approach theoretically. Let $\frac{1}{\gamma}$ denote the number of floating-point arithmetic operation per second (FLOPS) and N denote size of the matrix (i.e., the matrix is of size $N \times N$). We assume $\frac{1}{\gamma}$ as average FLOPS. Let $T_{overall}$ denote the time for matrix matrix multiplication, then it is well known that $T_{overall} = O(N^3)$.

3.3.1 Overhead for encoding

The time complexity of generating row or column checksum for input matrices can be expressed as follow

$$T_{encode} = N^2 \gamma \tag{3.2}$$

The overhead can be calculated by

$$\frac{T_{encode}}{T_{overall}} = O(\frac{1}{N})$$

The time and overhead to construct a full checksum matrix are

$$T_{encode_fullchecksum} = 2N^2\gamma \tag{3.3}$$

$$\frac{T_{encode_fullchecksum}}{T_{overall}} = O(\frac{1}{N})$$

3.3.2 Overhead for computation

Encoded information which helps detect and recover errors will introduce overhead to compute $A^c \times B^r$ instead of just computing $A \times B$. The additional computation time due to the increase of the matrix size is

$$\begin{aligned} T_{comp} &= 2(N+1) \times N \times (N+1) - 2N^3 \\ &= (4N^2 + 2N)\gamma \\ &\approx 4N^2\gamma \end{aligned} \tag{3.4}$$

The overhead, which has nothing to do with the number of errors, is

$$\frac{T_{comp}}{T_{overall}} = O(\frac{1}{N})$$

3.3.3 Overhead for detecting errors

The process which scans a whole $(N+1) \times (N+1)$ matrix with full checksum once needs $2 \times N^2$ addition operations and $2 \times N$ branch operations. If the program is to tolerate m errors, the time and overhead to detect a matrix is:

$$T_{detect} = 2mN^2\gamma \tag{3.5}$$

$$\frac{T_{detect}}{T_{overall}} = O(\frac{1}{N})$$

3.3.4 Overhead for Recovery

For the simple case that matrix C has only one encoded column and row, corrupted data can be recovered from the checksum relationship by just solving a linear equation. The overhead recovering data depends on the number N and how many errors the fault tolerant matrix multiplication recovers as well. Assuming the program is designed to tolerate at most m errors, the time complexity of recovery is

$$T_{recovery} = mN\gamma \tag{3.6}$$

The recovery overhead is

$$\frac{T_{recovery}}{T_{overall}} = O(\frac{1}{N^2})$$

The formula derived above shows the complexity of matrix multiplication dominates the complexity of the recovery overhead as the matrix size N approaches infinity.

3.4 Experimental Evaluation

In this section, we experimentally evaluate the performance of our fault tolerance approach. Three sets of experiments are performed to quantify the

- Performance and overhead of our on-line `FT-DGEMM` (Fault Tolerant General Matrix Matrix Multiplication).
- Performance comparison between our on-line `FT-DGEMM` and ABFT as well as TMR.
- Performance comparison between our on-line `FT-DGEMM` and `ATLAS DGEMM`.

All tests are performed on *Alamode* and *Mio* provided by CCIT and GECO in the Colorado School of Mines. The CPU on Alamode is Intel(R) Core(TM) i7-2600 CPU with 3.40GHz. The CPU on Mio is Intel(R) Xeon(R) CPU E5530 with 2.40GHz.

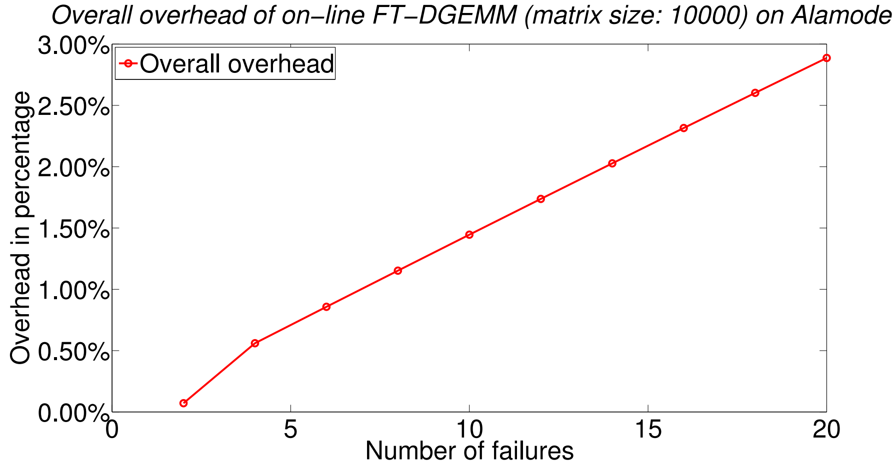


Figure 3.1: Overall Overhead in percentage.

The *number of failures* in the x-axis of all figures and texts in this section refers to the number of soft errors we can tolerate during one execution of our matrix matrix multiplication.

3.4.1 The overhead of our on-line `FT-DGEMM`

Overall Overhead

The first set of experimental results report the overall overhead over the `ATLAS DGEMM` for our on-line fault tolerant matrix matrix multiplication with different number of failures (i.e., different failure rates) but fixed matrix size(i.e., 10000). Figure 3.1 shows the percentage of overhead our approach introduces when different number of failures are injected into the program execution. Two failures during the program execution equal to 0.8 failures per minute and twenty failures during the program execution equal to 7.5 failures per minute. Figure 3.1 demonstrates that the proposed

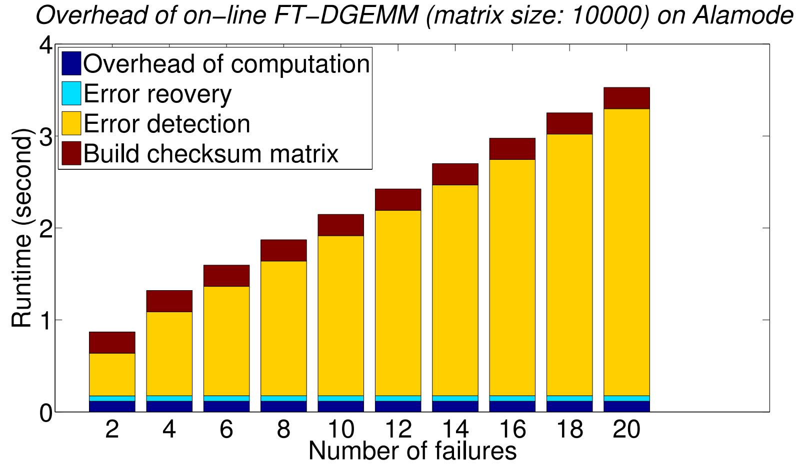


Figure 3.2: Overhead of online FTGEMM (matrix size: 10000) on ALAMODE.

technique can correct one error every ten seconds (i.e., six errors during the whole program execution) with negligible (i.e., less than 1%) performance penalty over the `ATLAS dgemm()`.

Overhead in detail

In the experiments, overhead of each part is timed to verify the derivations in section 3.3. Result matches the analysis in the section 3.3. Overhead is shown in a stack in each independent run. Runtime of overhead of each part is clearly displayed to show the portion they take in the overall overhead. Figure 3.2 to figure 3.5 show the execution time of overhead in stack bar figure. The runtime increases linearly as the failure rate grows. As we observe that the main growth is the overhead of detection whose frequency is determined by the failure rate.

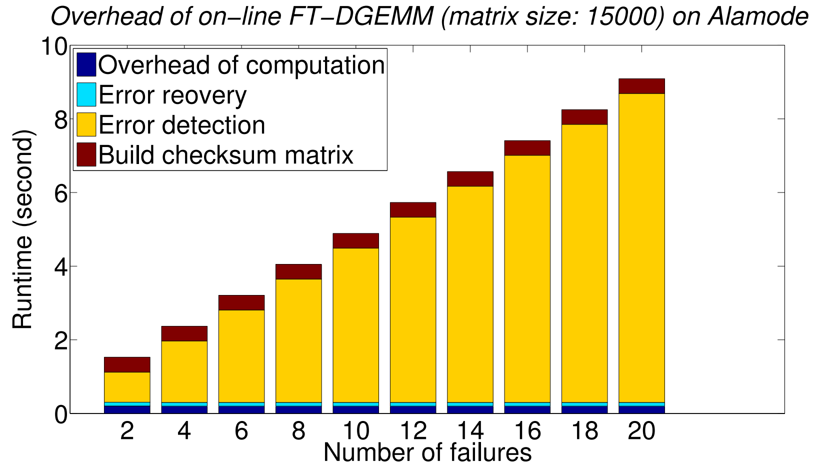


Figure 3.3: Overhead of on-line FTGEMM (matrix size: 15000) on ALAMODE.

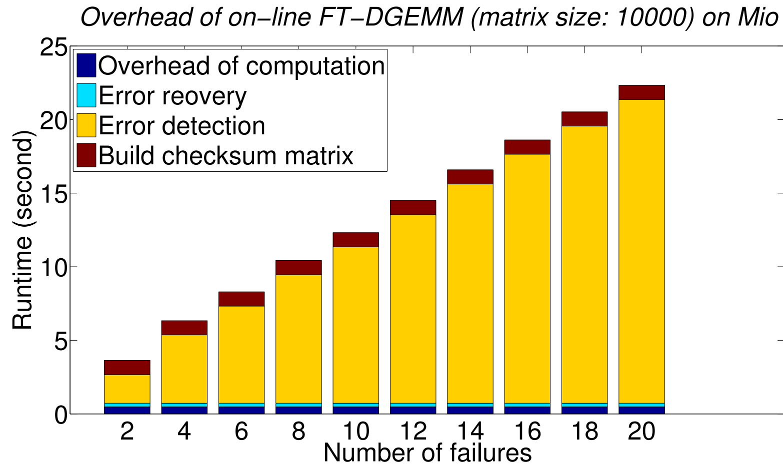


Figure 3.4: Overhead of on-line FTGEMM (matrix size: 10000) on MIO.

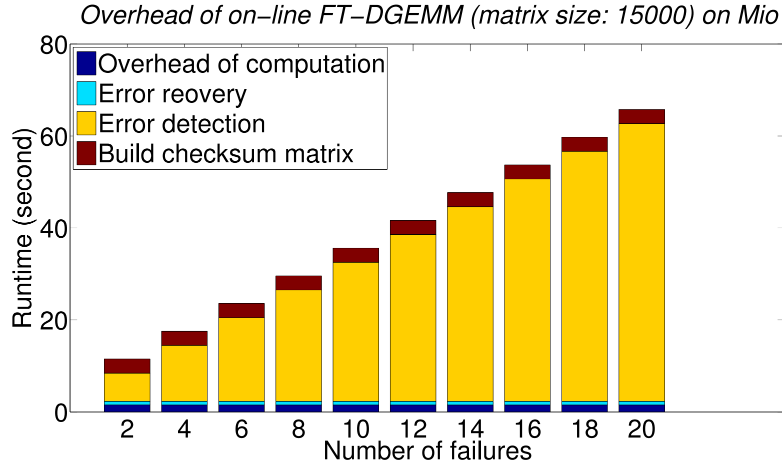


Figure 3.5: Overhead of on-line FT-DGEMM (matrix size: 15000) on MIO.

According to figures 3.2 - 3.5, the execution time for generating column checksum for matrix A and row checksum for matrix B is unchanged for fixed matrix size which is a $4N^2$ operation.

Detecting errors is a process to calculate summation of each row and column of matrix C to test whether they match the value on row and column checksum. It's a $O(mN^2)$ FLOPs computation. From figures 3.2 - 3.5, we can see that this portion of overhead increases linearly with the number of failures per execution, or failure rate the parameter of our implementation.

Recovery routines are implemented to try to recover corrupted entry in partial result C. If faults are found in detection phase, the row and column index would be flagged and errors in the intersection of problem row and problem column would be corrected by our mechanism, which is a $O(N)$ operation that is negligible.

The overhead of computation is introduced due to the increase of the matrix size from $N \times N$ to $(N+1) \times (N+1)$. As shown in the equation 3.4, it's $O(N^2)$ FLOP computation, which should be unchanged with fixed matrix size.

3.4.2 Performance comparison: TMR vs ABFT vs on-line ABFT

The set of data in Figures 3.6 to 3.9 demonstrate performance comparison between on-line `FT-DGEMM`, ABFT and TMR with the same matrix size 10000 and under three different actual failure rates. ABFT is a very famous technique to check the correctness of most matrix operation and recover the corrupted data which can tolerate fail-continue failures. TMR is a fault-tolerant mechanism in which three systems perform an identical process and the result is processed by a voting system to produce a single correct output. In our emulated TMR, we run the same program three times to tolerate faults during computation which results in incorrect data instead of fail of the device.

To tolerate two errors the on-line `FT-DGEMM` simply examines the checksum relationship twice during the calculation. However traditional ABFT can only detect one error (in the worst case) at the end of computation given the same checksum matrices. If two errors occur, traditional ABFT has to re-run the program in order to get the correct result. If there are two errors occur again in the re-run of the program, then a third run has to performed. So the execution time of traditional ABFT is at least

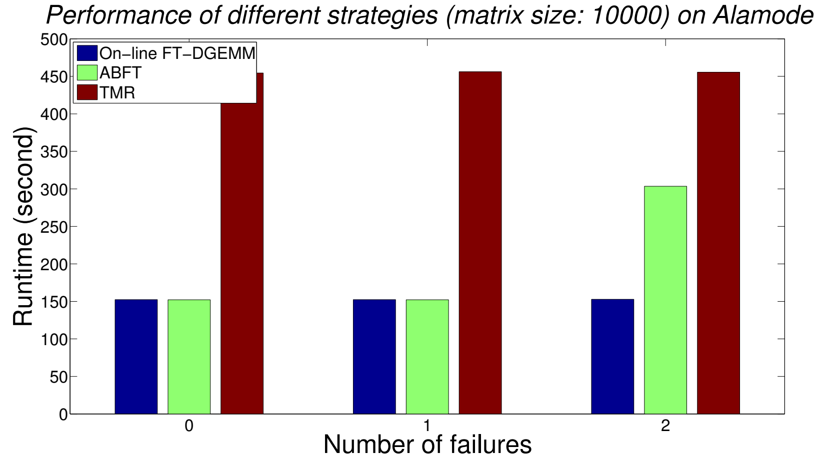


Figure 3.6: Performance of different strategies (matrix size: 10000) on ALAMODE.

twice as much as the execution time of our approach, in case two failures occurred during one matrix multiplication execution. In TMR, the same program has to be executed three times to produce three computation results for voting. Therefore, the total computation time to obtain correct result is at least three times the execution of the `ATLAS DGEMM`.

Figure 3.6 to 3.9 indicate that our on-line `FT-DGEMM` has much higher efficiency than ABFT and TMR.

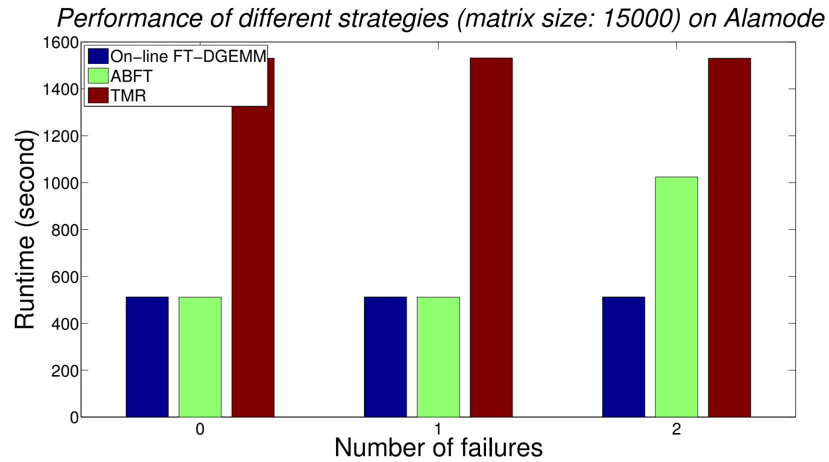


Figure 3.7: Performance of different strategies (matrix size: 15000) on ALAMODE.

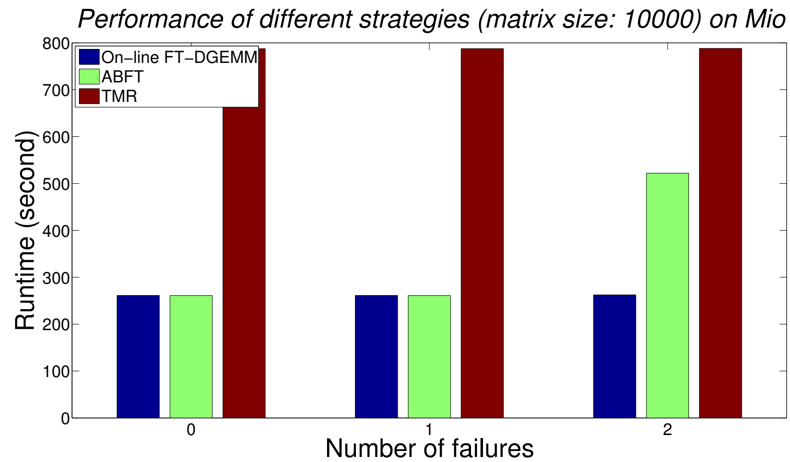


Figure 3.8: Performance of different strategies (matrix size: 10000) on MIO.

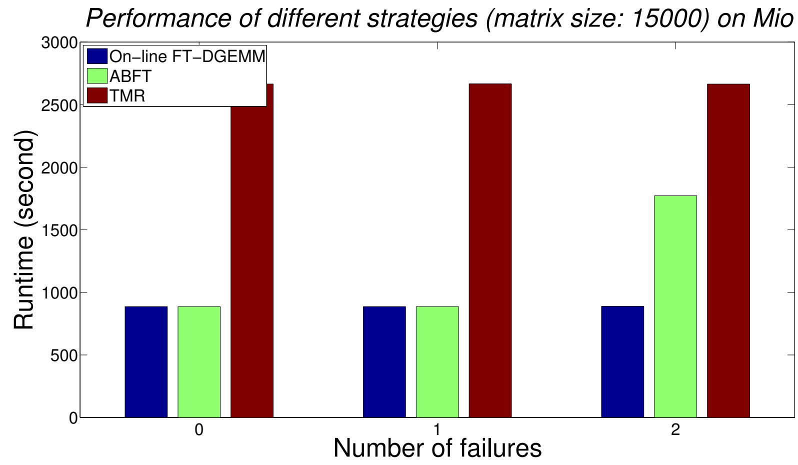


Figure 3.9: Performance of different strategies (matrix size: 15000) on MIO.

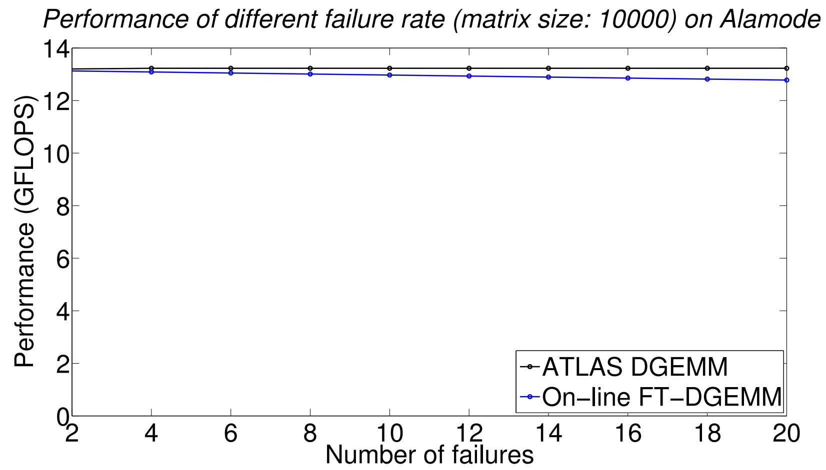


Figure 3.10: Performance for different failure rate (matrix size: 10000).

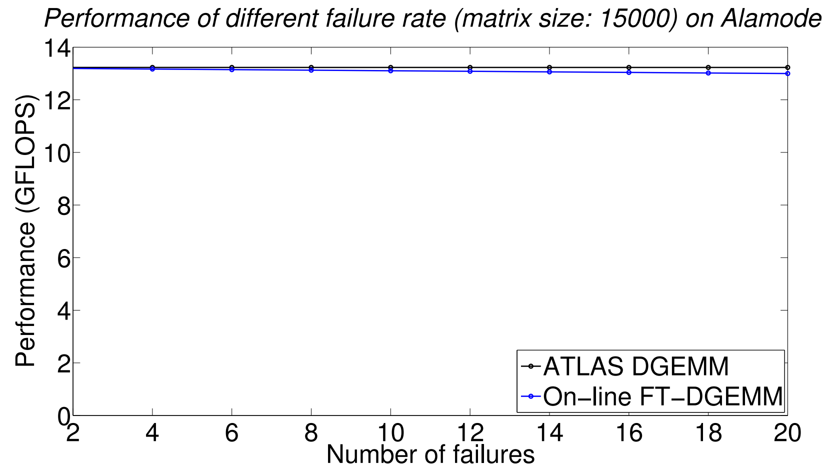


Figure 3.11: Performance for different failure rate (matrix size: 15000).

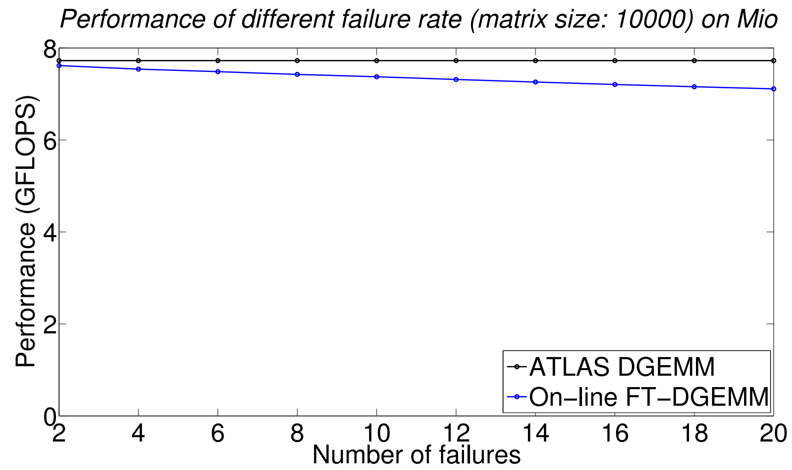


Figure 3.12: Performance for different failure rate (matrix size: 10000).

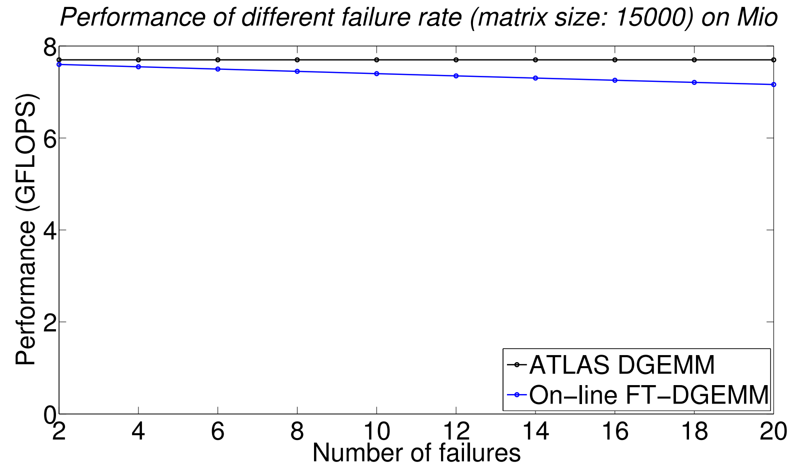


Figure 3.13: Performance for different failure rate (matrix size: 15000).

3.4.3 Performance comparison: `ATLAS DGEMM` vs our on-line `FT-DGEMM`

In this section, we show the comparison of performance between non fault tolerant `ATLAS DGEMM` and our on-line `FT-DGEMM`. Two sets of tests are performed on both Alamode and Mio.

Figures 3.10 - 3.13 indicate that our online `FT-DGEMM` increases the reliability dramatically by only introducing a very low overhead. As we can see on the figures, the performance drops by no more than 5% when tolerating up to twenty soft errors during the program execution if errors come one after another. The flexibility and high reliability of our approach make it possible to adopt it to various situations with acceptable overhead.

Chapter 4

One-sided Matrix Factorizations

It is well known that soft errors in linear algebra operations can be detected efficiently at the end of the computation (i.e., off-line) using algorithm-based fault tolerance (ABFT) [56, 3]. However, the ABFT technique in [56, 3] usually cannot correct even one soft error [26, 19, 95] in Cholesky, QR, and LU factorizations because one error in one matrix element will be propagated to many other matrix elements and hence cause too many errors to correct.

Recently, tremendous progresses have been made to correct soft errors in LU and QR factorizations. In [32], Sherman-Morrison-Woodbury formula was successfully used to correct one soft error in the solutions of linear systems obtained via LU factorization. In [30], Sherman-Morrison-Woodbury formula was extended to correct multiple soft errors. In [35], Spike-Eliminating and QR-Update techniques were used to correct one soft error in QR factorization. In [19], an online technique

was designed to correct soft errors in LU factorization of the high-performance Linpack(HPL) benchmark.

In this chapter, we present the design and implementation of FT-ScaLAPACK, a fault tolerant version ScaLAPACK that is able to detect, locate, and correct soft errors in Cholesky, QR, and LU factorizations on-line in the middle of the computation in a timely manner before the errors propagate and accumulate. FT-ScaLAPACK has been validated with thousands of cores on Stampede at the Texas Advanced Computing Center. Experimental results demonstrate that FT-ScaLAPACK is able to achieve comparable performance and scalability with the original ScaLAPACK library. More specifically, our contributions include:

- **Cholesky Factorization:** We designed an on-line scheme to correct soft errors in Cholesky factorization before the errors propagate and accumulate, where the existing best schemes [56, 3] cannot correct errors. Existing schemes need to restart the whole computation if any error occurs, therefore, introduces much higher overhead than our on-line scheme.

- **QR Factorization:** We designed an on-line scheme to correct soft errors in QR factorization before the errors propagate and accumulate, where the existing best schemes [35, 26] can only correct errors off-line at the end the computation after the errors propagated and accumulated. While the overhead of the existing off-line schemes increases at least quadratically as the number of errors increases, the overhead of our on-line scheme is much lower and increases only linearly.

- **LU Factorization:** We designed a new on-line scheme to correct soft errors in LU factorization without global communications or synchronizations, where the existing best schemes [19] are on-line, but involve expensive global communications and synchronizations.

- **Software Implementation:** We made the widely used ScaLAPACK library core routines (Cholesky, QR, LU) fault tolerant without modifying the library interfaces. Existing HPC applications that use ScaLAPACK library can now make use of our new FT-ScaLAPACK library to tolerate soft errors by just linking to the new library without any modification on source codes.

4.1 Background

This section provides the necessary background required to understand the idea of this paper. At first we give a brief introduction to traditional ABFT (we refer to as off-line ABFT); then in order to describe on-line ABFT we need to have a big picture of the so-called block version of algorithms for dense linear algebra algorithms, which are essential for high performance on modern hierarchical memory systems and distributed memory systems.

4.1.1 ABFT

ABFT was first introduced by Abraham and Huang [56]. The idea is that, for some matrix (or matrices) operation $P(A, \ldots) = (X, \ldots)$, we first encode the operands into their checksum form, for example $A \xrightarrow{\text{encode}} A^f := \begin{bmatrix} A & Ae \\ e^T A & e^T Ae \end{bmatrix}$ where e is a pre-defined (column) vector; then apply the operation on the encoded matrix (matrices) and the results are automatically "encoded":

$$P(A^{\text{enc}}, \ldots) = (X^{\text{enc}}, \ldots)$$

An example is the ABFT enabled matrix-matrix multiplication (matmul), in which case the operator P is $P(A, B) = A \times B = X$. The classic way to encode the operands A, B is as follows:

$$A \xrightarrow{\text{encode}} A^c = \begin{bmatrix} A \\ e^T A \end{bmatrix}, \quad B \xrightarrow{\text{encode}} B^r = \begin{bmatrix} B & Be \end{bmatrix}$$

And it can be shown that the result is also in some checksum form:

$$P(A^c, B^r) = X^f, \text{where} X = AB$$

The superscripts c, r, f of the encoded matrices A^c, B^r, X^f stand for "column, row, full" checksum matrices respectively. By definition, column checksums are the

(weighted) sums of every columns of matrix A, or mathematically $e^T A$. Row checksums are (weighted) sums of every row in A or mathematically Ae.

Another example is the LU factorization $A = LU$ where L is a unit lower triangular matrix and U is an upper triangular matrix. We may define the LU factorization operation as $P(A) = (L, U)$. To make LU factorization ABFT enabled, we encode the operand A into its full checksum form:

$$A \xrightarrow{\text{encode}} A^f = \begin{bmatrix} A & Ae \\ e^T A & e^T Ae \end{bmatrix}$$

then we apply P on A^f and the results are two automatically checksum encoded matrices:

$$P(A^f) = (L^c, U^r)$$

After the operation on the encoded matrix, we end up with the encoded result that includes our desired output and its checksums. We can verify the result by checking the matrix against its checksums: a match means the operation is carried out correctly and a mismatch indicates a problem. Of course, "match" here means within roundoff error bounds since floating point arithmetics are not exact.

4.1.2 Block algorithms

Modern dense linear algebra algorithms are arranged in such a way that level 3 BLAS operations (basically matrix-matrix multiplication) are used as much as possible for high performance. This results in block versions of the algorithms that "defer" many lower level BLAS operations and aggregate them together into a single level 3 operation later. It's called "block" because matrices are divided into rectangular blocks which, instead of scalars, are the basic units for the description of block algorithms. For example, the block version of the Cholesky factorization (as implemented in ScaLAPACK [18]) is given as follows.

Cholesky factorization turns a symmetric, positive definite square matrix into the product of a lower triangular matrix and its transpose: $A = LL^T$. The factorization happens in-place: the result L overwrites the original A. At each iteration, we write the $n \times n$ matrix A as four blocks:

$$
\begin{aligned}
\begin{bmatrix} A_{11} & A_{21}^T \\ A_{21} & A_{22} \end{bmatrix} &= \begin{bmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{bmatrix} \begin{bmatrix} L_{11}^T & L_{21}^T \\ 0 & L_{22}^T \end{bmatrix} \\
&= \begin{bmatrix} L_{11}L_{11}^T & L_{11}L_{21}^T \\ L_{21}L_{11}^T & L_{21}L_{21}^T + L_{22}L_{22}^T \end{bmatrix}
\end{aligned}
$$

The northwest block A_{11} is a $n_b \times n_b$ block; A_{21} is $(n - n_b) \times n_b$ block; A_{22} is $(n - n_b) \times (n - n_b)$ block. It follows that

$$A_{11} = L_{11}L_{11}^T \tag{4.1}$$

$$A_{21} = L_{21}L_{11}^T \tag{4.2}$$

$$A_{22} = L_{21}L_{21}^T + L_{22}L_{22}^T \tag{4.3}$$

If we solve the first Cholesky factorization of a block A_{11} from the first equation using unblocked algorithm we get L_{11}; then from the second equation we can solve L_{21} and from the third we have a new Cholesky factorization problem on a smaller $(n - n_b) \times (n - n_b)$ matrix A':

$$A' := A_{22} - L_{21}L_{21}^T = L_{22}L_{22}^T$$

Note that this step involves a matrix-matrix multiplication. It also turns out that this step accounts for the majority of computations. Repeat the above 3-step procedures, until the the whole matrix is factorized. In summary, each iteration in the block right-looking Cholesky factorization algorithm is a 3-step procedure:

1. xPOTF2: Cholesky factorize the diagonal block A_{11} from (4.1) and L_{11} overwrites A_{11}.

2. xTRSM: Solve the column panel L_{21} from (4.2) and L_{21} overwrites A_{21}.

3. xSYRK: Update the trailing matrix from (4.3) by $A_{22} \leftarrow A_{22} - L_{21}L_{21}^T$

4.2 On-line ABFT design framework

Off-line ABFT works regardless of the actual algorithms used to carry out a particular matrix operation. For example, there are quite some different algorithms for matrix-matrix multiplication; off-line ABFT works with any one of them. No matter which algorithm is chosen to do matrix matrix multiplication, the checksums will maintain at the end of the operation. However, the checksums do not necessarily maintain *in the middle* of the multiplication; in fact, Chen etc [15] showed that among the algorithms to do matrix matrix multiplication, only one of them (outer product algorithm) can maintain the checksums *during* the multiplication. This seems to imply that designing on-line ABFT is harder than off-line ABFT since we have to choose a specific algorithm that has the special property to maintain checksums during the operation.

However, in this paper we'll show that it is possible to design on-line ABFT for *any* block algorithms, not only for one particular algorithm. For this we deploy two strategies: 1) we attach checksums to each block instead of to the whole matrix; 2) the checksums are only loosely related to their corresponding blocks but not part of the blocks. We found that these strategies allow us to easily design on-line ABFT checksum schemes for all one-sided factorizations and potentially for any other block algorithms that exhibit similar structures of one-sided factorization.

4.2.1 A Separation: checksum and matrix

Inspired by this work [47], we develop a different view on ABFT that separates checksums from the matrix. Let us see an example on matrix matrix multiplication. Originally, the idea of ABFT is that the product of two checksum encoded matrices is also a checksum encoded matrix, which is illustrated in Fig 4.1 (a). In (a), the checksums of A, B, C are parts of the encoded matrices. A different point of view is to separate the checksums from their corresponding matrices, as shown in (b). It follows that the row and column checksums of C can be obtained by multiplying A with the row checksums of B and the column checksums of A with B, shown in (c) and (d). Using this point of view on ABFT, we are no longer relying on the "automatic" update of the checksums as part of the matrix operation on the encoded matrix. Instead, we can manipulate the checksums freely, so long as the checksums are updated to remain consistent with their corresponding matrices at certain points. It then became possible to maintain the consistency of checksums with their corresponding matrix blocks in the middle of the matrix operation instead of only at the end, at the expense of having to manually update the checksums rather than relying on the automatic update by the matrix operation.

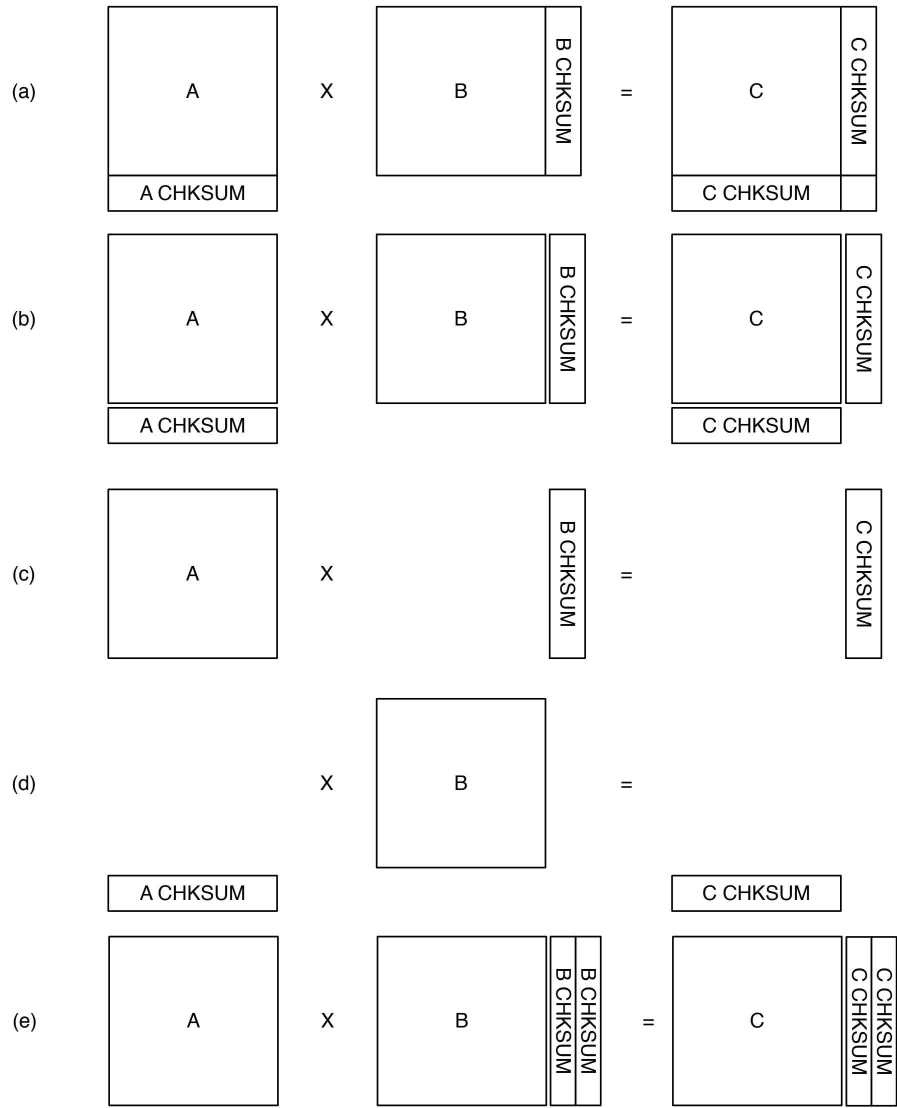


Figure 4.1: Traditional ABFT matrix matrix multiplication and the separation of checksums with the matrices

4.2.2 Double checksums

As shown in Figure 4.1 (a), there are two checksums for C—a row checksum and column checksum. Actually, to detect errors in C, one checksum is sufficient. However, to locate and correct errors, multiple checksums are required. In fact, to be able to correct m errors, at least $m + 1$ checksums have to be used. In the simplest setting, two checksums can detect errors and correct up to 1 error. And it does not have to a row and column checksum; two row checksums or two column checksums also work for up to 1 error. Fig 4.1 (e) shows two row checksums for matrix matrix multiplication. We found that two row checksums or two column checksums work best, for reasons that should be clear when the on-line ABFT Cholesky is presented later.

Let us see a simple example on how two checksums can detect and correct up to 1 error in a matrix row or column. In this example we use two checksums with different weights: $e_1 = [1, 1, \ldots, 1]^T, e_2 = [1, 2, \ldots, n]^T$. Assume that a matrix row is $a = [a_1, a_2, \ldots, a_n]$ and it is supposed to have two checksums r_1, r_2 available

$$r_1 = ae_1 = \sum_{i=1}^{n} a_i$$

$$r_2 = ae_2 = \sum_{i=1}^{n} ia_i$$

Now we have a computed $a' = [a'_1, a'_2, \ldots, a'_n]$ with up to 1 erroneous element $a'_j \neq a_j$ where the error position j is unknown to us. We know the error exists if

$$\delta_1 = \sum_{i=1}^{n} a'_i - r_1 = a'_j - a_j \neq 0$$

$$\delta_2 = \sum_{i=1}^{n} i a'_i - r_2 = j(a'_j - a_j) \neq 0$$

And a simple division $\delta_2/\delta_1 = j$ would give us the error position j; further $\delta_1 = a'_j - a_j$ gives us the magnitude of the error from which the correct a_j can be recovered from erroneous computed value a'_j using δ_1 and position j.

Note that this example only shows a single row and its two checksums; the same procedure can be applied to each row of a matrix and its column checksums; therefore with double row checksums we can tolerate up to 1 error per matrix row at a time. After the current error is corrected, the same scheme can be used to correct the next potential error. **Therefore, this scheme can correct multiple errors if errors arrive one after another.**

4.2.3 An example: on-line ABFT Cholesky factorization

The objective of designing on-line ABFT is to maintain checksum consistency during the matrix operation. Specifically, as shown in Fig 4.2, we want the checksums of every involved blocks after each step. For example, after the first step that factorizes the block A_{11} into lower triangular block L_{11}, we want to update the checksums of A_{11}

(denoted by $R(A_{11})$) to $R(L_{11})$. Similarly, we want to update the checksums involved in the second step and the third step $R(A_{21}), R(A_{22})$ to the checksums of the outputs $R(L_{21}), R(A'_{22})$. Fig 4.2 illustrates the idea of updating the checksums of the involved blocks at the end of every step in every iteration.

In Fig 4.2, (a) illustrates the 2nd iteration of the block right looking Cholesky factorization algorithm. Every iteration consists of 3 steps: the first step factorizes the diagonal block A_{11}; the second step updates the panel matrix blocks A_{21}; the third step updates the trailing matrix A_{22}. Subfigures (b), (c), (d) illustrate the three steps that update A_{11}, A_{21}, A_{22} to L_{11}, L_{21}, L_{22} respectively. Note that after updating the blocks, we also need to update their corresponding checksums, as shown in subfigures (b), (c), (d) lower parts. For example, in subfigure (b), we need to somehow update $R(A_{11})$ to $R(L_{11})$ in such a way that if $R(L_{11})$ remains the checksums of L_{11} if and only if the xPOTF2 procedure is carried out correctly.

The method to update of $R(A_{11})$ can be derived by examine the partial unblocked Cholesky factorization (xPOTF2) on matrix $\begin{bmatrix} A_{11} \\ R(A_{11}) \end{bmatrix}$. However, the result is independent of the specific algorithm used in xPOTF2. See the first part of subsection 4.3.3 for details.

After the update $R(A_{11}) \to R(L_{11})$, the updated checksum will be consistent with the updated block unless the Cholesky factorization on A_{11} is faulty. This is the basis on which we can use the checksum to check whether the xPOTF2 is carried out without faults.

We further need to do the same thing on the second and third steps. Fortunately, it is easier than the first step.

The second step is to update the panel matrix A_{21} using the result of the first step L_{11}. The triangular solve can be described mathematically:

$$A_{21} \times (L_{11}^T)^{-1} \rightarrow L_{21}$$

The obvious way to update the checksums of the panel matrix A_{21} is

$$R(A_{21}) \times (L_{11}^T)^{-1} \rightarrow R(L_{21})$$

Similarly, the third step is mathematically:

$$A_{22} - L_{21} \times (L_{21})^T \rightarrow L_{22}$$

The checksums of the trailing matrix A_{22} can be updated by

$$R(A_{22}) - R(L_{21}) \times (L_{21})^T \rightarrow R(L_{22})$$

The validity of the updates to the second and third steps can be easily seen and understood, once we note that the checksums of a matrix is just a product of a matrix

and checksum vector:

$$R(A) = e^T \times A$$

where e is a predefined (column) vector or a predefined matrix consisting of several (column) vectors.

4.2.4 A framework to design on-line ABFT for block algorithms

Now that we can update the checksums after each step and we can use the checksums to verify every step in every iteration of the Cholesky factorization, we in effect achieved "on-line" ABFT on this particular Cholesky factorization algorithm. Because the correctness of each step is verified, no error can escape and propagate. In the meantime, simple errors (single error per column) can be effectively corrected by checksums. These two properties of "on-line" ABFT significantly improves the resilience of ABFT approaches. Furthermore, the ability to verify the correctness continuously also provides otherwise missing information for other layers of fault tolerance such as checkpoint/rollback etc.

After examining the right-looking block Cholesky factorization algorithm and the method to maintain checksums, we can summarize a systematic way to derive checksum schemes for potentially many block linear algebra algorithms. First, we look at the description of the block algorithm at the block level; it usually consists of several

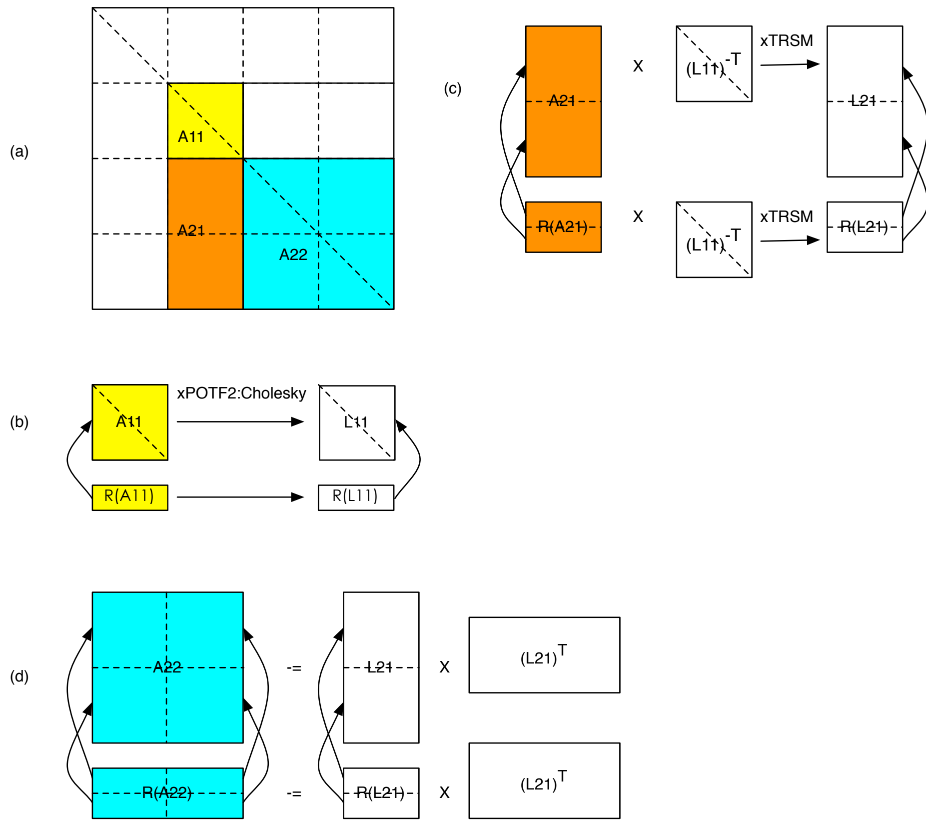


Figure 4.2: **On-line ABFT Cholesky** (a) the snapshot of the second iteration in a right-looking block Cholesky factorization algorithm. (b) the first step in this iteration is unblocked Cholesky factorization on A_{11}. (c) the second step in this iteration is a triangular solve to update the panel matrix; the checksums can be updated accordingly. (d) the third step is matrix multiplication to update trailing matrix; the checksums can be updated accordingly.

steps in every iteration. The first step is usually updating the diagonal block (A_{11}) using unblocked factorization. The following steps update the panel matrix (A_{21}) and trailing matrix (A_{22}) using some kind of matrix multiplication. To update the checksums for the first step, we just need to attach the checksums to the block and do a "partial" factorization on them (see subsection 4.3.3). To update the checksums for the following matrix multiplication steps, we just need to multiply the checksums with proper matrix, as shown in Fig 4.2 (c) (d).

Using this framework, we can derive "on-line" ABFT for all one-sided factorizations LU, QR, and Cholesky; the details vary but the principles apply. LU usually comes with partial pivoting which involves swapping rows; QR has a more complicated matrix multiplication update step. See section 4.3 for details on how to customize the checksum scheme for all one-sided factorization.

4.3 On-line ABFT enabled one-sided factorizations

In this section, additional one-sided factorization LU and QR and their customized "on-line" ABFT checksum schemes are discussed in detail. The missing part on how to update the checksums after the unblocked factorization step will be discussed thoroughly. Note in this section, in order to save space we may also use $[A; B]$ to denote a vertically stacked matrix $\begin{bmatrix} A \\ B \end{bmatrix}$, while $[A, B]$ denotes a horizontally stacked matrix $\begin{bmatrix} A & B \end{bmatrix}$.

4.3.1 LU

The LU factorization is essentially Gaussian elimination that factorizes a $M \times N$ matrix into $A = PLU$, where A and L is $M \times N$ matrices and U is $N \times N$ matrix. L is unit lower triangular and U is upper triangular; P is permutation matrix, which is stored in a vector IPIV.

The block right-looking LU factorization algorithm follows very similar structure of the Cholesky factorization algorithm described in the previous section. It is a series of iterations, with each iteration processing the trailing matrix of the previous iteration. The second iteration is illustrated in fig 6.4 (a). Every iteration is a 4-step process which can be described mathematically as follows.

1. xGETF2: Apply the (unblocked) LU factorization on the panel matrix $\left[A_{11}; A_{21}\right]$. This results in the upper triangular matrix U_{11} and lower triangular matrix L_{11} and the updated panel matrix blocks L_{21}, as shown in fig 6.4 (a).

2. xLASWP: Apply row interchanges to the left and right of the panel.

3. xTRSM: Solve the row panel U_{12} by $U_{12} \leftarrow (L_{11})^{-1} A_{12}$.

4. xGEMM: Update the trailing matrix A_{22} by $A'_{22} \leftarrow A_{22} - L_{21} U_{12}$.

Applying the framework described in section 4.2, we can derive the scheme to update checksums after each step.

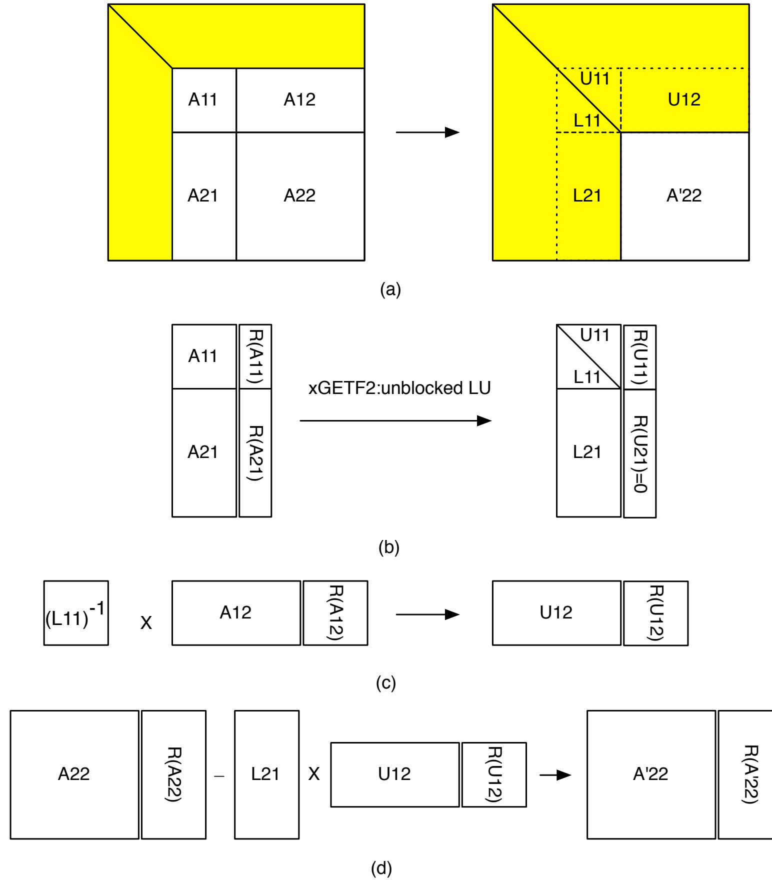


Figure 4.3: **On-line ABFT LU** (a) a snapshot of one iteration in a right looking block LU factorization algorithm: before and after. (b) the first step in this iteration is to update the column panel matrix by unblocked LU; the checksums are updated to checksums of right factor U_{11}, U_{21}. (c) second step is to update the row panel matrix by triangular solve; the checksums of the panel matrix can be updated accordingly. (d) the third step is matrix multiplication to update the trailing matrix; the checksums of the trailing matrix can be updated accordingly.

The update to the checksums of the first steps can be derived from partial LU factorization on $\begin{bmatrix} A_{11} & R(A_{11}) \\ A_{21} & R(A_{21}) \end{bmatrix}$, which will be discussed in subsection 4.3.3. This results in the checksums of upper triangular block U_{11} and the lower part U_{21} which is zero.

The second step, not shown in fig 6.4 is applying row interchanges to the left and right of the panel. If the checksums rows are interchanged according to the matrix rows the checksums will stay consistent after the update.

The third and fourth steps are essentially matrix multiplications. Again, we update the checksums by multiplying appropriate checksums with matrices, as shown in fig 6.4 (c) and (d).

4.3.2 QR

The QR factorization takes a $M \times N$ matrix A and factorizes it into the product $A = QR$ where Q is a $M \times M$ orthogonal matrix and R is upper triangular matrix.

The computation of block QR algorithm can be summarized as three steps in every iteration:

1. xGEQR2: Apply the (unblocked) QR factorization on the panel matrix $\begin{bmatrix} A_{11}; A_{21} \end{bmatrix}$. This results in the upper triangular matrix R_{11} and the Householder vectors stored in V, as shown in Figure 4.4 (a).

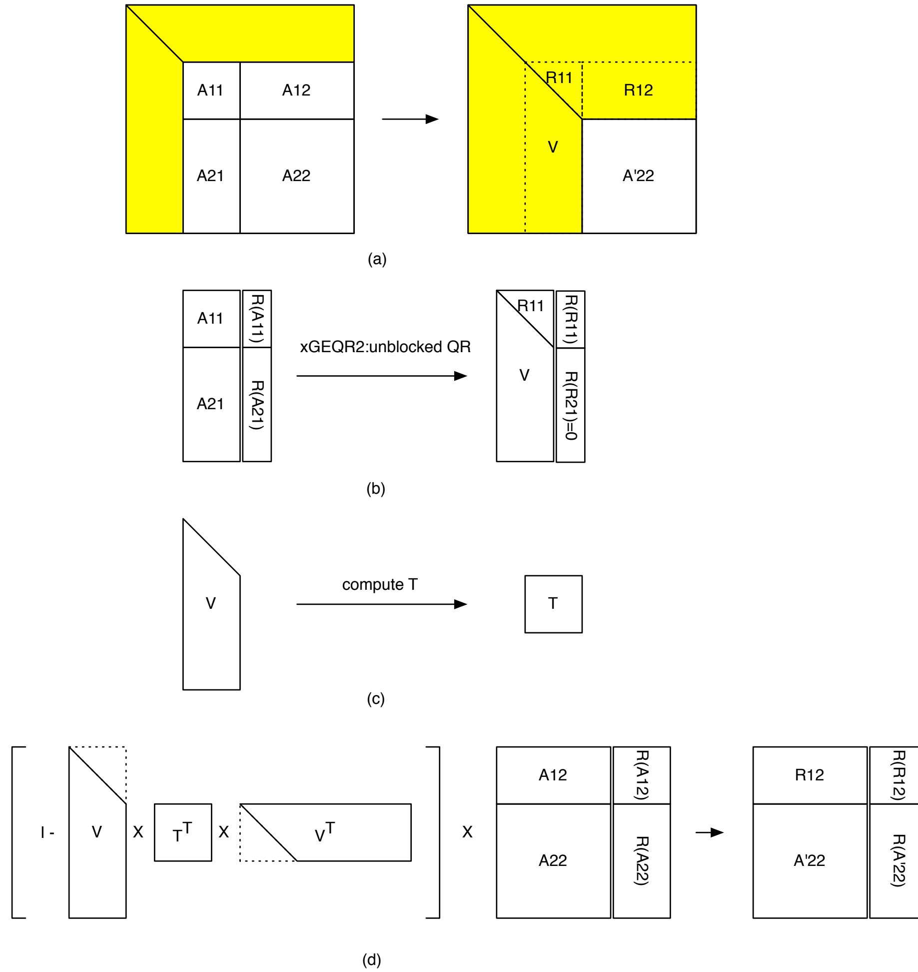


Figure 4.4: **On-line ABFT QR** (a) a snapshot of one iteration in right-looking block QR algorithm: before and after. (b) first step is updating column panel matrix by unblocked QR; checksums are updated to the checksums of right factor R_{11}, R_{21}. (c) the second step is to derive factor matrix T (d) the third step is to update the trailing matrix by matrix multiplication; the checksums can be updated accordingly.

2. xLARFT: From the Householder vectors a factor matrix T is computed, such that the Householder matrix factor is $Q = V^T \times T \times V$

3. xLARFB: Update the trailing matrix A_{22} by $A'_{22} \leftarrow (I - VT^TV^T)A_{22}$.

Applying the "on-line" ABFT block algorithm framework, we want to update the checksums of the involved blocks for every step. Like in the case in LU, the update to the checksums for the first step can be derived by doing partial QR factorization on $\begin{bmatrix} A_{11} & R(A_{11}) \\ A_{21} & R(A_{21}) \end{bmatrix}$ which will result in the checksums of the upper triangular block R_{11} and the lower part R_{21} which is 0.

The second step is computing Householder factor T from Householder vectors in V. This step cannot be protected by checksums; however we could test the orthogonality of $K = I - VT^TV^T$ by verifying the property that orthogonal transformations preserve 2-norm. In other words, for any vector x, we should have $||Kx||_2 = ||x||_2$. To efficiently evaluate Kx we can use the associativity of matrix multiplication to reduce the computation cost by $Kx = (I - VT^TV^T)x = x - V(T^T(V^Tx))$. The runtime overhead for the verification can be shown to be insignificant, and upon failure of the test, we can regenerate T from V.

The third step is essentially matrix multiplication. We update the checksums by multiplying the left factor to the checksums of A_{22} to obtain the checksums of A'_{22}.

4.3.3 Partial factorization

This subsection will discuss the missing part of the whole scheme—how to do partial factorization and derive the method to update checksums after the first step in LU, QR, and Cholesky factorizations.

First let us take the Cholesky factorization for an example. The (unblocked) Cholesky procedure that factorizes the block A_{11}—xPOTF2—is the outer-product version of Cholesky factorization. It can be described as Algorithm 4:

Algorithm 4 (out product unblocked Cholesky) Given a symmetric positive definite $A \in R^{n\times n}$, the following algorithm computes the L such that $A = LL^T$ and L overwrites the lower triangular part of A.

1: **for** $j = 1 : n$ **do**
2: $\quad A(j,j) \leftarrow \sqrt{A(j,j)}$
3: $\quad$ **if** $j < n$ **then**
4: $\quad\quad A(j+1 : n, j) \leftarrow A(j+1 : n, j)/A(j,j)$
5: $\quad\quad A(j+1 : n, j+1 : n) \leftarrow A(j+1 : n, j+1 : n) - A(j+1 : n, j) \cdot A(j+1 : n, j)^T$
6: $\quad$ **end if**
7: **end for**

Suppose before the factorization we have the column checksum of A which we denote as $r := e^T A$. After the above factorization, A is factorized and overwritten by the factor L. We want to update r so that r equals to $e^T L$ and r can be used to check L (apparently we cannot sum L up to get r, in which case r cannot be used to check L). The method to derive such update is to do the above factorization over $[A; r]$ partially instead of over A fully. If we plug in $[A; r]$ into the above algorithm, the last row which is our checksum r will be updated as in algorithm 5.

Algorithm 5 Given a positive definite matrix $A \in \mathbb{R}^{n\times n}$ and its column checksum $r = e^T A$, after Cholesky factorization $A = LL^T$, this algorithm updates the checksum r such that r will be the row checksum of L, i.e $r = e^T L$.

```
1: for j = 1 : n do
2:     r(j) ← r(j)/A(j, j)                                  ▷ line 4 in Alg 4
3:     r(j + 1 : n) ← r(j + 1 : n) − r(j) · A(j + 1 : n, j)^T  ▷ line 5 in Alg 4
4: end for
```

Because the checksum r is updated as a part in $[A; r]$ and the line 4 and 5 in algorithm 4 are all linear operations, the resulting r will still be checksum of A (which has been overwritten by L) at the end; i.e. $r = e^T L$.

Now let us look at the LU case. As usual, we first write down the outer product unblocked LU factorization as algorithm 6.

Algorithm 6 (outer product unblocked LU) Given a matrix $A \in \mathbb{R}^{m\times n}$ $(m \geq n)$, the following algorithm factorizes A into $A = PLU$ where $L \in \mathbb{R}^{m\times n}$ is a unit lower triangular matrix that overwrites the lower triangular part of A, and $U \in \mathbb{R}^{n\times n}$ is a upper triangular matrix that overwrites the upper triangular part of A. P is permutation matrix stored in a vector $IPIV$

```
1: for j = 1 : n − 1 do
2:     Find the pivot row index i = arg max_{j≤k≤m} |A(k, j)|
3:     IPIV(j) ← i
4:     Swap row i with row j: A(i, :) ↔ A(j, :)
5:     if A(j, j) ≠ 0 then
6:         A(j + 1 : m, j) = A(j + 1 : m, j)/A(j, j)
7:         A(j+1 : m, j+1 : n) = A(j+1 : m, j+1 : n)−A(j+1 : m, j)A(j, j+1 : n)
8:     end if
9: end for
```

Since algorithm 6 involves row swapping, column checksums are hard to maintain. We therefore choose to use a row checksum $r = Ae$. If we swap the rows of checksums in accordance with the swapping of matrix A in this algorithm, the row checksum

will remain consistent. Similar to the Cholesky case, we apply this algorithm 6 on $[A; r]$ instead of on A; the resulting update procedure is described in algorithm 7.

Algorithm 7 Given a matrix $A \in \mathbb{R}^{m \times n}$ $(m \geq n)$ and its column checksum $r = Ae$, after LU factorization $A = PLU$, this algorithm updates the checksum r such that r will become the column checksum of $[U; 0] \in \mathbb{R}^{m \times n}$, i.e. $r = [Ue; 0] \in \mathbb{R}^m$

```
1: for j = 1 : n − 1 do
2:     Swap row j with row IPIV(j): r(j) ↔ r(IPIV(j))         ▷ line 4 in Alg 6
3:     if A(j, j) ≠ 0 then
4:         r(j + 1 : m) = r(j + 1 : m) − A(j + 1 : m, j)r(j)   ▷ line 7 in Alg 6
5:     end if
6: end for
```

Again, because the operations in line 4 and 7 in algorithm 6 are all linear, after LU on A and the algorithm 7 on r, the updated A and r will still be consistent in the sense that $r = [U; 0]e$ holds.

The last case of the one-sided factorizations is QR. As usual, we first write down the outer product unblock QR factorization. It will be more complicated than the previous two factorizations, but still the structures are the same the same design can be applied.

We also choose to use row checksum $r = Ae$. Applying algorithm 8 on $[A, r]$ instead of on A we derive the algorithm 9 that updates the checksum r so that $r = [R; 0]e$

After performing algorithm 9 on r, the updated r will become the row checksum of $[R; 0]$ $(r = [R; 0]e)$. This concludes our description on how to derive the method to

Algorithm 8 (outer product unblocked QR) Given a matrix $A \in \mathbb{R}^{m\times n}$ ($m \geq n$), this algorithm finds Householder matrices $H_1, \ldots, H_n$ such that if $Q = H_1 \cdots H_n$, then $A = QR$ where $Q \in \mathbb{R}^{m\times m}$ is an orthogonal matrix and $R \in \mathbb{R}^{n\times n}$ is upper triangular matrix. Householder vectors $H_1, \ldots, H_n$ overwrite the lower triangular part of A and R overwrites the upper triangular part of A.

1: **for** $j = 1 : n$ **do**
2: $[v, \beta] \leftarrow \text{householder}(A(j : m, j))$
3: $A(j : m, j : n) \leftarrow (I - \beta vv^T)A(j : m, j : n)$
4: **if** $j < m$ **then**
5: $A(j + 1 : m, j) \leftarrow v(2 : m - j + 1)$
6: **end if**
7: **end for**

Algorithm 9 Given a matrix $A \in \mathbb{R}^{m\times n}$ ($m \geq n$) and its column checksum $r = Ae$, after the QR factorization $A = QR$ this algorithm updates the checksum r such that r becomes column checksum of $[R; 0]$, i.e. $r = [R; 0]e$.

1: **for** $j = 1 : n$ **do**
2: $v \leftarrow \begin{bmatrix} 1 \\ A(j+1 : m, j) \end{bmatrix}$ ▷ line 5 in Alg 8.
3: $r(j : m) \leftarrow (I - \beta vv^T)Ar(j : m)$ ▷ line 3 in Alg 8.
4: **end for**

update checksums for the first steps in LU, QR, and Cholesky factorization. Note that this subsection only derives the update methods (algorithms 5, 7, 9) based on but *not* relied on outer product unblocked LU, QR, and Cholesky algorithms (algorithms 4, 6, 8); actually, the update algorithms 5, 7, 9 work with *any* LU, QR, Cholesky factorization algorithms, not only with the out product versions.

4.3.4 Duplicate to protect panel blocks

Note in the LU and QR case, after the first step that factorizes the panel matrix $[A_{11}; A_{21}]$ and its row checksums, the checksums will become the checksums of the right factor (U in LU and R in QR) as shown in fig 6.4 (b) and fig 4.4 (b); the left factors L and Q will have no checksums to protect them. This means that even though errors in L and Q will be detected as erroneous left factor leads to erroneous right factor, the left factors have no redundant information to correct errors. To be able to tolerate errors in left factors we can duplicate the panel matrix in memory before the first step factorization so that if the first step proves to be faulty, we can rollback using that duplicate to repeat the first step factorization. This procedure only duplicate the current panel matrix A_{11}, A_{21} thus inducing little run time and memory overhead.

4.4 Performance analysis and experimental evaluation

In this section we first introduce a model to analyze the overhead of incorporating the proposed on-line ABFT to ScaLAPACK factorizations. We then show the performance of our implementation of "on-line" ABFT enabled LU,QR, and Cholesky on up to 1600 processes.

4.4.1 Performance and scalability analysis

According to ScaLAPACK user manual [6], ScaLAPACK is "scalable" in the sense that, maintaining constant memory use per process (n^2/P), the overall efficiency should be maintained no matter how many processes are used. We argue that, adding "on-line" fault tolerance as described in this paper into the one-sided factorization subroutines in ScaLAPACK the scalability should remain the same. The overhead should be bounded by a small constant.

A simple model of run time of one-sided factorizations in ScaLAPACK [6] is decomposing the time into computation, message bandwidth, and message latencies. It is not a very accurate model but a good enough first order approximate for the run time ScaLAPACK one-sided factorization subroutines. For our purpose to show the

Table 4.1: The meaning of the variables in equation 4.4, according to [6]

Variable	Description
$C_f n^3$	Total number of floating-point operations
$C_v n^2$	Total number of data items communicated
$C_m n/NB$	Total number of messages
t_f	Time per floating-point operation
t_v	Time per data item communicated
t_m	Time per message
n	Matrix size
P	Number of processes
NB	Data distribution block size

Table 4.2: The value of the factor C_f, C_v, C_m for LU, QR, and Cholesky factorizations in ScaLAPACK according to [6]

	C_f	C_v	C_m
LU	2/3	$3 + 1/4 \log_2 P$	$NB(6 + \log_2 P)$
Cholesky	1/3	$2 + 1/2 \log_2 P$	$4 + \log_2 P$
QR	4/3	$3 + \log_2 P$	$2(NB \log_2 P + 1)$

performance and scalability impact of "on-line" ABFT, we list the model here:

$$T(n, P) = \frac{C_f n^3}{P} t_f + \frac{C_v n^2}{\sqrt{P}} t_v + \frac{C_m n}{NB} t_m \tag{4.4}$$

where n is the matrix size (assuming the matrix is $n \times n$ square matrix), P the number of processes, C_f, C_v, C_m the computation, message size and message number factors respectively (see table 4.2), and t_f, t_v, t_m the time per FLOP, interconnection bandwidth and its latency; see table 4.1.

The parallel efficiency [6] of a one-sided factorization is

$$E(n,P) = (1 + \frac{1}{NB}\frac{C_m t_m}{C_f t_f}\frac{P}{n^2} + \frac{C_v t_v}{C_f t_f}\frac{\sqrt{P}}{n})^{-1} \quad (4.5)$$

Overhead for maintaining checksums

In order to correct errors in every row of every block at every iteration, two checksums for each row in each block are needed. After introducing checksums we are actually factorizing a matrix of size $(1 + \frac{2}{NB})n$. Our algorithms behave similar to the original ScaLAPACK subroutines. Therefore, similar to 4.4, our computation time can be modeled by

$$T'(n,P) = \frac{C'_f n^3}{P} t_f + \frac{C'_v n^2}{\sqrt{P}} t_v + \frac{C'_m n}{NB} t_m \quad (4.6)$$

where

$$C'_f = (1 + \frac{4}{NB})C_f, C'_v = (1 + \frac{4}{NB})C_v, C'_m = C_m \quad (4.7)$$

Plugging in new C's into equation 4.5 gives us the efficiency of “on-line” ABFT factorizations:

$$E'(n,P) = (1 + \frac{1}{NB+4}\frac{C_m t_m}{C_f t_f}\frac{P}{n^2} + \frac{C_v t_v}{C_f t_f}\frac{\sqrt{P}}{n})^{-1} \quad (4.8)$$

From the two running time equations 4.4 and 4.6, the performance overhead introduced by adding on-line fault tolerance is bounded by a constant factor $\frac{4}{NB}$, which is independent of P and often small because NB is usually in hundreds in today's optimzied library codes. Notice that, the only difference between equations 4.5 and 4.8 is the factor before the second term in parentheses. ScaLAPACK is scalable in the sense that, maintaining the local memory usage $\frac{n^2}{P}$ on each process will lead to maintained efficiency; and the larger the local memory usage per process the better efficiency will be. Equation 4.8 shows that our online ABFT factorizations are also scalable in the same sense as in ScaLAPACK.

Overhead for error detection

In order to detect errors, all elements in the same row of a block need to be added together and then compared to existing checksums maintained in the factorizations. The total number of FLOPs (floating point operations) to verify all checksums for all blocks in any processor is $\frac{n^2}{P}$. Therefore, the overhead for error detection is approximately $\frac{1}{n}$. Even if we verify correctness at every iteration (i.e., approximately every second in our experiments in Section 5.2) of the factorization, the performance overhead is still only $\frac{1}{NB}$, which is independent of P and negligible since NB is usually in hundreds in today's library codes. Hence, the scalability will be the same as the orginal ScaLAPACK

Overhead for error location

After an error is detected, in order to locate the error, the weighted checksum of the faulty row (or column) needs to be calculated. The total number of extra FLOPs to locate an error is NB. Therefore, the performance overhead for error location is approximately $\frac{NB}{n^3/P}$. Even if there is an error on every processor at every iteration of the factorization, the total overhead is only $\frac{P}{n^2}$, which is again negligible because the problem size n is often much larger than the number of processors P in today's supercomputing applications. Furthermore, the total overhead will not incease as the processor P increases if the size of the local matrix $\frac{n}{\sqrt{P}}$ on each processor does not decrease. Therefore, the scalability is not affected.

Overhead for error correction

After an error is located, the error correction is just simply adding the error back to the corrupted matrix element (see Section 3.1 for details), which needs only one FLOPs. Correcting k errors needs only k FLOPs.

4.4.2 Experimental evaluation

Our FT-ScaLAPACK implementation is based on ScaLAPACK 2.0.2. ScaLAPACK uses the so-called 2D block cyclic matrix distribution [7] to spread matrix and computation onto multiple processes to achieve load balance. The natural strategy to implement the "on-line" checksum scheme is placing the checksum on the same pro-

cess where its corresponding matrix block resides. In this way checksums are always local to their corresponding blocks and all checking and correcting error procedures can be performed locally without inter process communication. Also, it is convenient to treat the checksums of the matrix blocks as a normal ScaLAPACK distributed matrix so that common operations involving the checksums can be performed using ScaLAPACK infrastructure. However, in order to eliminate extra communications to update checksums, we must aggregate the communication of matrix blocks and their checksums to avoid communicating checksums separately. In this way we can keep the number of messages unchanged but increase the message size a little bit which is reflected in Equation 4.7.

We implemented the double precision "on-line" ABFT enabled PDGETRF (LU), PDGEQRF (QR), and PDPOTRF (Cholesky) subroutines, indicated with prefix "FT-". We use double checksums for each block which means we can correct an error in every row/column of every bolck at every iterations in every processor. As we can see from section 4.3, except for Cholesky which has symmetry, checksums for the other two factorizations can only protect one factor of the factorization result: in $A = PLU$ checksums only protect U but not L and in $A = QR$ checksums only protect R. We used duplicates described in subsection 4.3.4 to tolerate errors in L or Q.

We run both FT-ScaLAPACK and ScaLAPACK subroutines on the Stampede supercomputer at the Texas Advanced Computing Center, which is a 10 PFLOPS

Dell Linux cluster ranking #7 at the current TOP500 Supercomputers List. Each node has 2x 8-core Xeon E5 processors, with each core peaking 21.6GFLOPS. The interconnect is FDR 56 Gb/s mellanox switches organized in a fat-tree topology. We use all 16 cores in every node and 1 MPI task per core.

Software side, ScaLAPACK 2.0.2 from netlib is compiled against MKL 13.0.2.146 using Intel compiler 13.1.0 and Intel MPI 4.1.0.030. We only use MKL for its BLAS and LAPACK functions. For each factorization subroutine, we perform weak scaling tests; i.e. we scale the problem size with the number of processes and maintain the memory usage per process. For simplicity, we use square process grids $8 \times 8, 16 \times 16, 24 \times 24, 32 \times 32, 40 \times 40$. Since matrix A is distributed almost evenly on $P \times P$ processes, if we keep the memory use per process fixed at $F \times F$ double precision floats (for QR is $2F \times F$) then the size of matrix A is $(FP) \times (FP)$ (for QR is $(2FP) \times (FP)$). The parameters F, NB, M, N are indicated in the figure headers, where $M \times N$ is the shape of matrix A.

Figure 4.5 indicates: (1). The fault tolerance overhead fluctuates around 5%, which does not increase with the number of processes P and is not far from the theoretical estimation in section 5.5; (2). The total program execution times for both FT-ScaLAPACK and ScaLAPACK routines increase linearly as $\sqrt{P}$ increases, which implies both versions have constant efficiency and hence the same scalability. These experimental results confirmed our theoretical performance and scalability analysis in section 5.5.

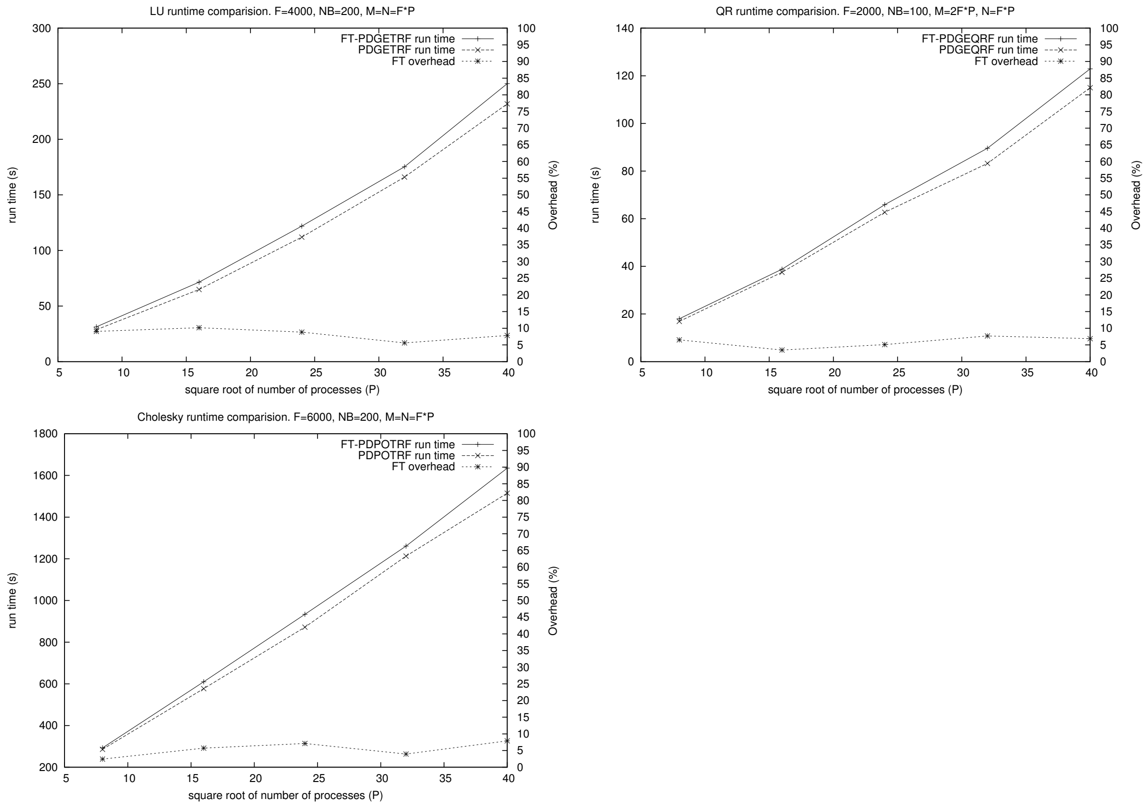


Figure 4.5: Run time for fault tolerant versions of LU, QR, and Cholesky in FT-ScaLAPACK and their original versions in ScaLAPACK. The fault tolerant versions can detect, locate, and correct one error in every row of every bolck on every processor at every iteration. On average, one iteration takes roughly one seconds in these experiments.

Chapter 5

Two-sided Matrix Factorizations

Algorithm based fault tolerance (ABFT) provides an appealing alternative to modular redundancy for soft error detection and correction. ABFT essentially provides ECC functionality for higher level data that is under transformation. Both techniques can detect errors in the encoded data and provide forward error recovery at low cost. The salient difference between ECC and ABFT is that ECC protects static data while ABFT must cope with dynamic data. For ECC, once the data word is written its codeword is calculated and stored. When the data is read the codeword is checked for integrity. The data are supposed to remain unchanged. Thus ECC protects against bit flips in storage system and communication. However ABFT protects data that are under certain transformations (linear algebra operations). Before computation the target data objects (usually matrix and vector) are encoded and after the computation the encoded data object is checked for integrity. This requires

the encoding to remain valid under the computation, and often modification to the algorithms used.

Direct eigen solvers including the computation of eigenvalue decomposition and singular value decomposition are prevalent in science, engineering, and more recently in data analytics. In science and engineering, common problems such as vibration mode in mechanical engineering and Schrodinger's equation in quantum mechanics. In data analytics, eigen solvers are used in principal component analysis (PCA), Linear Discriminant Analysis (LDA), spectral graph theory, and in particular Google's PageRank [12]. To solve such eigenproblems the matrix is usually first unitarily reduced to the simplest form possible (Hessenberg for asymmetric matrix, tridiagonal for symmetric matrix, and bidiagonal for SVD decomposition for any matrix) and then is solved iteratively. The first step, the reduction to simple forms, usually dominates in execution time and computing resources. Once reduced, the eigenproblems can be solved quickly in a single machine. Therefore we focus on the first phase reduction in this paper.

Two recent publications [59, 60] discussed algorithm based fault tolerance for Hessenberg factorization against fail-stop errors, and bidiagonalization against fail-continue errors, respectively. The algorithmic protection technique used is based on matrix-matrix multiplication similar to [58, 17, 49, 99, 100]. For two sided factorizations that are of interest here, matrix-matrix multiplication only constitutes half

of the work in terms of floating point operations (flop) and less than 20%[1] of the execution time. Furthermore, the checksums of the matrices cannot be automatically maintained during the factorizations thus have to be regenerated in every iteration. The checksum regeneration adds overhead and compromises fault coverage, as regenerated checksum cannot be used to verify correctness for prior operations. In contrast, this paper proposes an algorithm based fault tolerance scheme against soft errors that covers all BLAS2 and BLAS3 operations and with automatically maintained checksums. The results are substantially improved fault coverage (almost all floating point operations) with lower overhead. Additionally we consider the complete set of three two-sided factorizations for eigenproblems and analyze them in a unified framework.

The contribution of this chapter is as follows:

- We propose the first comprehensive online ABFT schemes against soft errors for three two-sided factorizations.
- We analytically and empirically evaluate the fault coverage and efficiency of the proposed scheme and demonstrate the superiority to the current state of the art.
- We implemented the proposed technique in the Scalable Linear Algebra Package (ScaLAPACK) for easy adoption without changes to the interfaces.

[1] Assuming BLAS2 speed is 20% of speed of BLAS3, the coverage in time is 16%. For example on Intel Sandy Bridge and AMD Piledriver BLAS2 speed is around 1/5 and 1/7 that of of BLAS3[97].

5.1 Background

This section introduces necessary backgrounds on checksum based fault tolerance and the two sided matrix factorization algorithms.

5.1.1 Checksums encoded matrix and its operations

Checksum encoded matrix are the basis of algorithm based fault tolerance in linear algebra. Generally speaking checksums are linear combinations of the rows or columns of the matrix. The column checksum of a matrix is a row vector that is the (weighted) sum of all the rows in the matrix; The row checksum of a matrix is a column vector that is the (weighted) sum of all the columns in the matrix. The weights are usually predefined and independent of the matrix. Mathematically, the column checksum of a matrix (denoted with superscript c) can be represented as:

$$A^c = \begin{bmatrix} A \\ e^T A \end{bmatrix}$$

The row checksum can be written as:

$$A^r = \begin{bmatrix} A & Ae \end{bmatrix}$$

The e is a (column) vector with all elements being 1. It is usually used as the default weights; when we want to emphasize a non-uniform weight w is commonly

used. The full checksum encoded matrix is a matrix with both row and column checksum:

$$A^f = \begin{bmatrix} A & Ae \\ e^T A & e^T Ae \end{bmatrix}$$

A basic result of checksum based ABFT is as follows: if $AB = C$.

$$\begin{bmatrix} A \\ e^T A \end{bmatrix} \begin{bmatrix} B & Ae \end{bmatrix} = \begin{bmatrix} C & Ce \\ e^T C & e^T Ce \end{bmatrix}$$

or using the notation of checksum encoded matrix:

$$A^c B^r = C^f \tag{5.1}$$

This equation shows the maintenance of checksum in matrix multiplication: column checksum encoded matrix multiply row checksum encoded matrix gives us a full checksum encoded matrix. This property can be used to detect errors in the matrix multiplication.

5.1.2 Error correction with multiple checksums

The checksums can be used for error detection and it can also be used for error correction if we use multiple checksums with different weights. All the discussions in this paper can be extended to multiple checksums with arbitrary weights. Here is a simple

double checksum scheme to locate and correct up to 1 errors per vector. Suppose we encode a vector using two different weights $e_1 = [1, 1, \ldots, 1]^T, e_2 = [1, 2, \ldots, n]^T$. The vector is $a = [a_1, \ldots, a_n]$ and we have two correct encoded checksums of a:

$$r_1 = ae_1 = \sum_{i=1}^{n} a_i, \quad r_2 = ae_2 = \sum_{i=1}^{n} ia_i$$

Now suppose the computed $a' = [a'_1, \ldots, a'_n]$ has up to one erroneous element $a'_j \neq a_j$, where the location j is unknown to us. However when we verify the checksums:

$$\delta_1 = \sum_{i=1}^{n} a'_i - r_1 = a'_j - a_j \neq 0$$
$$\delta_2 = \sum_{i=1}^{n} ia_i - r_2 = j(a'_j - a_j) \neq 0$$

Then a simple division δ_2/δ_1 gives us the location j. The correct value of a_j can then be recovered using the correct checksum and the other correct elements of a: $a_j = a'_j - \sum_{i=1, i\neq j}^{n} a'_i$. Therefore, if we have two checksums for a vector we can correct up to 1 erroneous element in it. If we have full double checksums for a matrix we can correct up to 1 erroneous element in each column/row. In the following text we only assume single checksum for brevity of notations; extending to multiple checksums with different weights is a easy to do.

5.1.3 Maintaining checksum for matrix factorizations

There are three common one sided factorizations: LU, Cholesky, and QR. The LU factorization factorizes a matrix A into a lower triangular matrix L and an upper triangular matrix U. It can be shown that (almost) unmodified LU algorithm operating on the full checksum encoded matrix A^f would result in column checksum encoded L^c and row checksum encoded U^r.

$$A = LU, A^f = \begin{bmatrix} L \\ e^T L \end{bmatrix} \begin{bmatrix} U & Ue \end{bmatrix}$$

The same works for Cholesky factorization:

$$A = LL^T, A^f = \begin{bmatrix} L \\ e^T L \end{bmatrix} \begin{bmatrix} L^T & L^T e \end{bmatrix}$$

For QR factorization $A = QR$ because the left factor Q is supposed to be orthogonal, unmodified QR algorithm operating on A^f would *not* result in the desired $Q^c R^r$ as QR produces the orthogonal left factor while Q^c cannot be orthogonal as Q^c is singular. However if we only perform QR on the row checksum encoded A^r then we have:

$$A = QR, A^r = QR^r$$

We see that the the QR factorization preserves the row checksum. This is possible because the right factor R is supposed to be upper triangular and R^r can conform to this requirement.

For two sided factorizations however the left factor and right factor are both orthogonal thus we cannot hope to use unmodified algorithm to do the factorization that preserves the checksum. For example bidiagonalization factors a matrix into:

$$A = QBP^T, A^f = Q'B'P'^T$$

There is no possibility that $Q' = Q^c$ or $P' = P^c$ as Q', Q, P, P' are all supposed to be orthogonal. Nor can we expect unmodified two sided factorization algorithms to preserve row/column checksums for A^r or A^c. For the other two two-sided factorizations the checksums cannot be automatically maintained without modification to the factorization algorithm, as in both cases the left and right factors are orthogonal.

5.1.4 Householder reflector for unitary diagonalization

Householder reflector is an orthogonal matrix that maps a target vector to the first axis. The effect is shown as in figure 5.1.

The Householder matrix F is constructed based on vector x as follows:

$$v = x + \text{sign}(x_1)||x||_2 e_1, \quad F = I - \frac{2}{v^T v} v v^T \tag{5.2}$$

$$x = \begin{bmatrix} \times \\ \times \\ \times \\ \vdots \\ \times \end{bmatrix} \quad \xrightarrow{F} \quad Fx = \begin{bmatrix} ||x|| \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix} = ||x||e_1$$

Figure 5.1: The effect of (left) Householder reflector to a column vector.

in which e_1 is the vector $(1, 0, 0, \ldots, 0)^T$. It can be shown that applying the left Householder matrix to a column vector x will reflect x to the first axis, eliminating all other components except the first one. Note that since the Householder reflector is symmetric and orthogonal, it serves as the primary tool to unitarily triangularize (zero out lower triangular part) a matrix. Likewise, applying a Householder matrix on the right to a row vector will have similar effect and can be used to zero out the upper triangular part.

To simplify notation we normalize the Householder vector v: $v = v/||v||_2$. The Householder matrix thus becomes $F = I - 2vv^T$. Regarding the checksum preserving property of Householder reflection we have the following important lemma that is the basis of all the automatic maintenance of checksums in two-sided factorizations:

Lemma 1 *Suppose* $(I - 2vv^T)A = A'$, *then we have the corresponding checksum result:*

$$A^f - 2v^c v^T A^r = (A')^f$$

Proof.

$$\text{LHS} = \begin{bmatrix} A & Ae \\ e^T A & e^T Ae \end{bmatrix} - 2 \begin{bmatrix} v \\ e^T v \end{bmatrix} v^T \begin{bmatrix} A & Ae \end{bmatrix}$$

$$= \begin{bmatrix} A' & A'e \\ e^T A' & e^T A'e \end{bmatrix} = \text{RHS}$$

■

There is a symmetric case that applies F to the right of A. The significance of this lemma is that, if we have the checksum of Householder vector v and the matrix A, we can operate on the checksum augmented vector/matrix to obtain the desired output, *and* its associated checksums. In other words, we have defined a certain application of Householder matrix in such a way that the checksum is preserved. The Householder transformation will be modified in this way in the two sided matrix factorizations in order to maintain the checksum encoding, thus serving as the foundation for checksum based ABFT for them.

5.2 Unblocked version

In this section we describe the checksum based ABFT for unblocked bidiagonal reduction and Hessenberg/tridiagonal reduction. Unblocked algorithms are easy to understand but not practical in practice due to its inefficiency in using modern hi-

erarchical memory system. It will however illustrate the basic technique for further more complex blocked algorithms and distributed algorithms.

5.2.1 Bidiagonal reduction

Bidiagonal reduction (or bidiagonalization) is the first step in computing singular value decomposition. It is the process of alternating left and right orthogonal transformations to reduce a general $m \times n$ matrix into a upper diagonal form (if $m \geq n$), in which all but the diagonal and first superdiagonal entries are zeros:

$$A = QBP^T, \text{Q and P are orthogonal, B is bidiagonal.} \tag{5.3}$$

Householder reflectors are usually used to introduce zeros into the subdiagonal. Note that in general no direct methods can reduce A to bidiagonal form. The key method in the factorization is through Householder transformation; applying Householder reflector on the left of the matrix reduces eliminates the entries below diagonal in the current column; applying it on the right eliminate the entries that are right of the first superdiagonal in the current row. The process is shown in figure 5.2.

Let us first consider the unblocked version of the bidiagonalization algorithm used in LAPACK DGEBD2 subroutine. Suppose we are given a thin matrix A of size $m \times n$ and $m \geq n$. The (upper) bidiagonalization algorithm is described in algorithm 10

$$
\begin{bmatrix} X & X & X & X \\ X & X & X & X \\ X & X & X & X \\ X & X & X & X \\ X & X & X & X \\ X & X & X & X \end{bmatrix}_{A}
\xrightarrow{U_1^* \cdot}
\begin{bmatrix} \mathbf{X} & \mathbf{X} & \mathbf{X} & \mathbf{X} \\ \mathbf{0} & \mathbf{X} & \mathbf{X} & \mathbf{X} \\ \mathbf{0} & \mathbf{X} & \mathbf{X} & \mathbf{X} \\ \mathbf{0} & \mathbf{X} & \mathbf{X} & \mathbf{X} \\ \mathbf{0} & \mathbf{X} & \mathbf{X} & \mathbf{X} \\ \mathbf{0} & \mathbf{X} & \mathbf{X} & \mathbf{X} \end{bmatrix}_{U_1^* A}
\xrightarrow{\cdot V_1}
\begin{bmatrix} X & \mathbf{X} & 0 & 0 \\ & \mathbf{X} & \mathbf{X} & \mathbf{X} \\ & \mathbf{X} & \mathbf{X} & \mathbf{X} \\ & \mathbf{X} & \mathbf{X} & \mathbf{X} \\ & \mathbf{X} & \mathbf{X} & \mathbf{X} \\ & \mathbf{X} & \mathbf{X} & \mathbf{X} \end{bmatrix}_{U_1^* A V_1}
$$

$$
\xrightarrow{U_2^* \cdot}
\begin{bmatrix} X & X & & \\ & \mathbf{X} & \mathbf{X} & \mathbf{X} \\ & \mathbf{0} & \mathbf{X} & \mathbf{X} \\ & \mathbf{0} & \mathbf{X} & \mathbf{X} \\ & \mathbf{0} & \mathbf{X} & \mathbf{X} \\ & \mathbf{0} & \mathbf{X} & \mathbf{X} \end{bmatrix}_{U_2^* U_1^* A V_1}
\xrightarrow{\cdot V_2}
\begin{bmatrix} X & X & & \\ & X & \mathbf{X} & 0 \\ & & \mathbf{X} & \mathbf{X} \\ & & \mathbf{X} & \mathbf{X} \\ & & \mathbf{X} & \mathbf{X} \\ & & \mathbf{X} & \mathbf{X} \end{bmatrix}_{U_2^* U_1^* A V_1 V_2}
\xrightarrow{\cdots}
\begin{bmatrix} X & X & & \\ & X & X & \\ & & X & X \\ & & & X \\ & & & \\ & & & \end{bmatrix}_{U_n^* \ldots U_1^* A V_1 \ldots V_n}
$$

Figure 5.2: Householder Reduction to Bidiagonal Form

Let us walk through the bidiagonalization procedure in algorithm 10. Line 2-4 computes the left Householder vector and matrix; line 5 applies the resulting Householder matrix to the left of the matrix to eliminate the column (see the first and third transformation in figure 5.2). Note that Similarly, line 6-8 computes the right Householder vector and matrix; line 9 applies the resulting Householder matrix to the right of the matrix to eliminate the row (see the second and fourth transformation in figure 5.2).

Now it is time to design a checksum scheme and modification to the algorithm 10 in such a way that the checksum encoding will remain valid at certain point during the factorization. In fact we will aim at maintaining checksum encoding at the end

Algorithm 10 Householder Reduction to (Upper) Bidiagonal Form

1: Require: matrix $A \in \mathbb{R}^{m \times n}, m \geq n$
2: Ensure: $A \rightarrow QBP^T$ where B is (upper) bidiagonal, Q, P are orthogonal
3: **for** $k = 1 : m - 1$ **do**
4: $\quad x = A_{k:m,k}$
5: $\quad q = x + \text{sign}(x_1)||x||_2 e_1$
6: $\quad q = q/||q||_2$
7: $\quad A_{k:m,k:n} = A_{k:m,k:n} - 2q(q^T A_{k:m,k:n})$
8: $\quad x = A_{k,k+1:n}$
9: $\quad p = x + \text{sign}(x_1)||x||_2 e_1$
10: $\quad p = p/||p||_2$
11: $\quad A_{k:m,k+1:n} = A_{k:m,k+1:n} - 2(A_{k:m,k+1:n} p)p^T$
12: $\quad$Store q, p
13: **end for**

of each Householder application (line 5 and 9) in each loop iteration. We summarize the design problem we are facing now:

> Problem: Given a fully checksum encoded original matrix A^f, how do we modify the algorithm 10 such that at the end of line 5 and 9 the matrix A^f or its submatrix is properly encoded?

The basis is the lemma 1. Our design is present in the algorithm 11.

Theorem 2 *In the modified algorithm 11 at the end of line 5 or line 9 in each iteration, the matrix A remains full checksum encoded. At the end of the algorithm, the result will be the full checksum encoded result of that of algorithm 10 performed on the matrix without encoding.*

Algorithm 11 Modified Householder Reduction to (Upper) Bidiagonal Form

1: Require: A full checksum encoded matrix $A \in \mathbb{R}^{(m+1)\times(n+1)}, m \geq n$
2: Ensure: $A \rightarrow Q^c B (P^r)^T$ where B is (upper) bidiagonal, Q, P are orthogonal
3: **for** $k = 1 : m-1$ **do**
4: $\quad x = A_{k:m+1,k}$
5: $\quad q = x + \text{sign}(x_1)||x_{1:\text{last}-1}||_2 e_1^c$
6: $\quad q = q/||q_{1:\text{last}-1}||_2$
7: $\quad A_{k:m+1,k:n+1} = A_{k:m+1,k:n+1} - 2qq^T_{1:\text{last-}1}A_{k:m,k:n+1}$
8: $\quad x = A_{k,k+1:n}$
9: $\quad p = x + \text{sign}(x_1)||x_{1:\text{last}11}||_2 e_1^c$
10: $\quad p = p/||p_{1:\text{last}11}||_2$
11: $\quad A_{k:m+1,k+1:n+1} = A_{k:m+1,k+1:n+1} - 2A_{k:m+1,k+1:n}p_{1:\text{last}+1}p^T$
12: $\quad$ Store q, p
13: **end for**

To see why theorem 2 holds intuitively, it is instructive to compare algorithm 11 to algorithm 10 line by line. For example, by induction before line 5 the pre-condition that the matrix A is full checksum encoded. Applying lemma 1 to line 5 leads to the post-condition that the after line 5 the matrix A remains full checksum encoded. Also note that by construction q and p in algorithm 11 is the column checksum encoded q and p in algorithm 10. This leads to the conclusion that after line 5 the matrix A in algorithm 11 is the full checksum encoded A in algorithm 10. These two conclusions combined to prove theorem 2. We thus illustrated that algorithm 11 provides a solution to our stated problem.

5.2.2 Hessenberg reduction and tridiagonal reduction

Hessenberg reduction is the first step in computing eigenvalue decomposition. It is the process of applying the same orthogonal transformation to both left and right

$$
\underset{A}{\begin{bmatrix} X & X & X & X & X \\ X & X & X & X & X \\ X & X & X & X & X \\ X & X & X & X & X \\ X & X & X & X & X \end{bmatrix}}
\xrightarrow{Q_1^* \cdot}
\underset{Q_1^* A}{\begin{bmatrix} X & X & X & X & X \\ \mathbf{0} & \mathbf{X} & \mathbf{X} & \mathbf{X} & \mathbf{X} \\ \mathbf{0} & \mathbf{X} & \mathbf{X} & \mathbf{X} & \mathbf{X} \\ \mathbf{0} & \mathbf{X} & \mathbf{X} & \mathbf{X} & \mathbf{X} \\ \mathbf{0} & \mathbf{X} & \mathbf{X} & \mathbf{X} & \mathbf{X} \end{bmatrix}}
\xrightarrow{\cdot Q_1}
\underset{Q_1^* A Q_1}{\begin{bmatrix} X & X & X & X & X \\ X & \mathbf{X} & \mathbf{X} & \mathbf{X} & \mathbf{X} \\ & \mathbf{X} & \mathbf{X} & \mathbf{X} & \mathbf{X} \\ & \mathbf{X} & \mathbf{X} & \mathbf{X} & \mathbf{X} \\ & \mathbf{X} & \mathbf{X} & \mathbf{X} & \mathbf{X} \end{bmatrix}}
$$

$$
\xrightarrow{Q_2^* \cdot}
\underset{Q_2^* Q_1^* A Q_1}{\begin{bmatrix} X & X & X & X & X \\ X & X & X & X & X \\ & \mathbf{X} & \mathbf{X} & \mathbf{X} & \mathbf{X} \\ & \mathbf{0} & \mathbf{X} & \mathbf{X} & \mathbf{X} \\ & \mathbf{0} & \mathbf{X} & \mathbf{X} & \mathbf{X} \end{bmatrix}}
\xrightarrow{\cdot Q_2}
\underset{Q_2^* Q_1^* A Q_1 Q_2}{\begin{bmatrix} X & X & \mathbf{X} & \mathbf{X} & \mathbf{X} \\ X & X & \mathbf{X} & \mathbf{X} & \mathbf{X} \\ & X & \mathbf{X} & \mathbf{X} & \mathbf{X} \\ & & \mathbf{X} & \mathbf{X} & \mathbf{X} \\ & & \mathbf{X} & \mathbf{X} & \mathbf{X} \end{bmatrix}}
\xrightarrow{\cdots}
\underset{Q_n^* \dots Q_1^* A Q_1 \dots Q_n}{\begin{bmatrix} X & X & X & X & X \\ X & X & X & X & X \\ & X & X & X & X \\ & & X & X & X \\ & & & X & X \end{bmatrix}}
$$

Figure 5.3: Householder reduction to Hessenberg form

of the matrix to reduce the a square matrix into upper Hessenberg form in which all but the upper triangular part and the subdiagonal are zero:

$$A = QHQ^T, \text{Q is orthogonal, H is Hessenberg matrix.} \tag{5.4}$$

Note the difference from bidiagonal reduction: the left factor and right factor are both Q. The reduction is also carried out through Householder reflectors. The additional constraint that the left factor and right factor must be the same in general prevents direct reduction to bidiagonal as in the case of bidiagonal reduction case; the best we can do is the Hessenberg form. If the matrix A is symmetric then the H is both Hessenberg and symmetric therefore tridiagonal. In our discussion tridiagonal reduction can be treated as a special case of Hessenberg reduction thus will not be discusses separately unless necessary.

To modify the Householder Hessenberg reduction algorithm, again the basis is lemma 1. The result is very similar to the bidiagonal reduction case so we only give the modified algorithm 12.

Algorithm 12 Modified Householder Reduction to Hessenberg Form

1: Require: A full checksum encoded matrix $A^f \in \mathbb{R}^{(m+1)\times(m+1)}$
2: Ensure: $A^f \to Q^c, H^f, (Q^T)^r$ where $A = QHQ^T$, B is (upper) bidiagonal, Q, P are orthogonal
3: **for** $k = 1 : m-2$ **do**
4: $\quad x = A_{k+1:m+1,k}$
5: $\quad q = x + \text{sign}(x_1)||x_{1:\text{last}-1}||_2 e_1^c$
6: $\quad q = q/||q_{1:\text{last}-1}||_2$
7: $\quad A_{k+1:m+1,k:m+1} = A_{k+1:m+1,k:m+1} - 2qq^T_{1:\text{last-1}}A_{k+1:m+1,k:m+1}$
8: $\quad A_{1:m+1,k+1:m+1} = A_{1:m+1,k+1:m+1} - 2A_{1:m+1,k+1:m+1}q_{1:\text{last-1}}q^T$
9: $\quad$Store q, p
10: **end for**

We then have a theorem similar to theorem 2 asserting similar checksum maintenance.

5.3 Blocked version

In contrast to the previous unblocked algorithms for two sided factorizations, in this section we move on to the blocked algorithms which are substantially more efficient and complex. The unblocked algorithms are rich in level 1 and 2 BLAS operations (vector-vector and matrix-vector) which have low computation intensity. The blocked version aggregates some of the BLAS 2 into single BLAS 3 (matrix-matrix multiplication) operations. BLAS 3 operations have much better computation

Algorithm 13 Modified Householder reduction to bidiagonal form, Blocked Version in matlab notation. Red color marks the modifications.

Input: Full checksum encoded $A_{1:m+1,1:n+1}$

WHILE $m > 0$:

1. `DLABRD`: REduce the kth panel of the matrix and compute X, Y. [Repeat B times for $i = 1, \ldots, B$.]
   (a) Update the jth column of A: $A_{i:m+1,i} = A_{i:m+1,i} - A_{j:m+1,1:i-1} * Y'_{i,1:i-1} - X_{j:m+1,1:i-1} * A_{1:i-1,i}$.
   (b) Verify $A_{i:m+1,i} = A^c_{i:m,i}$
   (c) Compute the ith column Householder vector of A, $A_{i:m+1,i}$
   (d) Compute $Y_{i+1:n+1,i} = \tau_v(A'_{i:m,i+1:n+1} * A_{i:m,i} - Y_{i+1:n+1,1:i-1} * A'_{i:m,1:i-1} * A_{i:m,i} - A'_{1:i-1,i+1:n+1} * X'_{i:m,1:i-1} * A_{i:m,i})$
   (e) Update the jth row of A: $A_{i,i+1:n+1} = A_{i,i+1:n+1} - A_{i,1:i}Y^T_{i+1:n+1,1:i} - X_{1:i-1,i}A_{1:i-1,i+1:n+1}$.
   (f) Verify $A_{i,i+1:n+1} = A^r_{i,i+1:n}$
   (g) Compute the ith row Householder vector of A, $A_{i,i+1:n+1}$.
   (h) Compute $X_{i+1:m+1,i} = \tau_u(A_{i+1:m+1,i+1:n} * A'_{i,i+1:n} - A_{i+1:m+1,i+1:n} * Y'_{i+1:n,1:i} * A'_{i,i+1:n} - X_{i+1:m+1,1:i-1} * A_{1:i-1,i+1:n} * A'_{i,i+1:n})$.
2. `DGEMM`: Update A with V and Y, $A_{B+1:m+1,B+1:n+1} = A_{B+1:m+1,B+1:n+1} - A_{B+1:m+1,1} * Y'_{B+1m+1,1:B}$.
3. `DGEMM`: Update A with X and U, $A_{B+1:m+1,B+1:n+1} = A_{B+1:m+1,B+1:n+1} - X_{B+1:m+1,1} * A_{1:B,B+1:n+1}$.
4. Verify $A_{B+1:m+1,B+1:n+1} = A^f_{B+1:m,B+1:n}$

$A = A_{B+1:m+1,B+1:n+1}, m = m - B, n = n - B$

END WHILE

Process the remaining A with unblocked algorithm.

intensity therefore better performance on modern computers. The method to modify the blocked algorithms to maintain checksums is similar but not quite the same as the unblocked algorithms. The details are presented in this section.

5.3.1 Bidiagonal Reduction

Algorithm 14 Modified Blocked Householder Reduction to Bidiagonal Form, Blocked Version in matrix notation.. Red colored marks the modifications.

1. `DLABRD`: Reduce the kth panel of the matrix and compute V, U, X, Y. [Repeat B times for $i = 1, \ldots, B$ (let $j = (k-1)B + i$).]
   (a) Update the jth column of A: $A^c_{:,j} = A^c_{:,j} - V^c_{i-1} y_i^T - X^c_{i-1} u_i^T$
   (b) Compute the ith column Householder vector of A, v_i^c.
   (c) Compute $(y_i)^c = \tau_v((A^r)^T v_i - (Y_{i-1})^c V_{i-1}^T v_i - U^c_{i-1} X_{i-1}^T v_i)$
   (d) Update the jth row of A: $(A^r_{j,:})^T = (A^r_{j,:})^T - Y_i^c (v_I^T) - U^c_{i-1} x^T_{i-1}$
   (e) Compute the ith row Householder vector of A, u_i^c
   (f) Compute $x_i^c = \tau_u(A^c u_i - X^c_{i-1} U^T_{i-1} u_i - V_i^c Y_i^T u_i)$.
2. `DGEMM`: Update A with V and Y, $A^f = A^f - V^c (Y^c)^T$.
3. `DGEMM`: Update A with X and U, $A^f = A^f - X^c (U^c)^T$.

The blocked bidiagonal reduction (xGEBRD in LAPACK) solves the same factorization problem in equation 5.3 by delaying the application of line 5 and 9 in algorithm 10 for a few iterations and apply the aggregated effect in a single matrix multiplication. The detailed process is illustrated in algorithm 14. The algorithm is summarized from [25] which is used in LAPACK.

The blocked algorithm 14 looks quite a bit more elaborate compared to algorithm 10. The algorithm 14 describes an iteration in a loop. The main complexity is in calculating the temporary matrices X, Y that represent the effect of rank-2 trailing matrix update (line 5 and 9 in algorithm 10). The X, Y "store" the update to the trailing matrix and are used to apply the update in bulk of B rank-2 updates into matrix multiplications, or rank-$2B$ update (step 2, 3 in algorithm 14).

To modify the blocked algorithm 14 such that it maintains checksum encoding when operating on fully checksum encoded matrix A, lemma 1 cannot be directly used since the rank-1 Householder transformation is delayed and applied in bulk. An apparent idea is to relate the matrix multiplications in step 2 and 3 in algorithm 14 to equation 5.1. If we can have a column checksum encoded U, V, X, Y and full checksum encoded A, the trailing matrix update (step 2,3) that subtract $VY^T + XU^T$ from A leaves a fully checksum encoded A as we desired. This approach is used by [60]. The problem with this approach is that the checksums of matrices U, V, X, Y must be recalculated every time before step 2 and 3. The inability to maintain the checksums of U, V, X, Y not only adds additional overheads but also compromises fault coverage: the faulty calculation of U, V, X, Y cannot be detected as they are not encoded. We would like to avoid the recalculation of checksums and develop a automatically maintained checksum throughout the blocked algorithm. The U, V, X, Y should all be encoded throughout the algorithm, and any faulty calculation in the algorithm 14 would lead to inconsistently encoded matrix.

To do that we follow an intuitive guideline similar to the modification we made in algorithm 11: we include the checksums in vector, matrix-vector, and matrix-matrix operations when it makes sense. If the result is supposed to be a full checksum encoded matrix, the left factor should be column checksum encoded and the right factor should be row checksum encoded; if the result is supposed to be column checksum encoded, the left factor should be column checksum encoded and the rest checksum is not included. The resulting modified algorithm is presented in algorithm 13.

We claim that after step 2,3 in algorithm 13, the matrix A is full checksum encoded, and after step 1.(a) or 1.(e) the updated vector in A is row checksum encoded. To see why this is so we need to go into the derivation of blocked algorithm and the equivalence between unblocked and blocked versions. Carefully applying the basic maintenance of checksums in matrix-matrix/vector multiplication proves this claim. The details are omitted here due to limited space.

5.3.2 Hessenberg and Tridiagonal Reduction

The Hessenberg and tridiagonal reduction is similar in spirit to the bidiagonal case. We list only the Hessenberg reduction algorithm here in algorithm 15; The tridiagonal reduction is Hessenberg reduction on symmetric matrix.

Algorithm 15 Modified orthogonal reduction to Hessenberg form, blocked version in matrix notation.. Red colored marks the modifications.

1. `DLAHRD`: Reduce the kth panel of the matrix and compute V, Y, T. [Repeat B times for $i = 1, \ldots, B$ (let $j = (k-1)B + i$).]
   (a) Computer the Householder vector v_i^c.
   (b) Compute $y_i^c = \tau(A^c v_i - Y_{i-1}^c V_{i-1}^T v_i)$.
   (c) Compute $t_i = -\tau T_{i-1} V_{i-1}^T v_i$
   (d) Update the $(j+1)th$ column of A if necessary:
      i. Apply the block Householder vector from the right: $A^c_{:,j+1} = A^c_{:,j+1} - Y_i^c [V_i]_{j,:}$
      ii. Apply the block Householder vector from the left: $A^c_{:,j+1} = A^c_{:,j+1} - V_i^c T_i^T V_i^T A_{:,j}$
2. `DGEMM`: Update A with V and Y, $A^f = A^f - Y^c (V^c)^T$.
3. `DLARFB`: Apply the block Householder vector from the left, $A^f = A^f - V^c T^T V^T A^r$.

5.4 Distributed Blocked Version

In this section we discuss how to adapt the ABFT scheme for the distributed two sided factorizations in the ScaLAPACK package. First we will briefly introduce the 2D cyclic data distribution and how ScaLAPACK implements the blocked factorizations discussed previously on a distributed memory cluster system. Then we discuss the adaption of the ABFT scheme to distributed memory system and the modification to the corresponding distributed algorithm.

5.4.1 Data distribution

The ScaLAPACK uses the two dimensional cyclic block data distribution [8] for the matrix on distributed memory system because of its efficient support for high performance matrix-matrix multiplication and load balance for parallel speed up and scalability. In many linear algebra algorithms such as the matrix factorizations, the algorithm works on ever smaller submatrix as the algorithm progresses thus posing a load balance problem. To mitigate this problem 2D block cyclic distribution assigns matrix blocks to processes in a round-robin fashion. Therefore each process will possess blocks with a constant stride. Two dimensional means the matrix is partitioned into a 2D array of blocks and the processes are arranged into a 2D grid. The blocks is assigned according to its row and column index, as shown in figure

In the previous section we attach checksums to the global matrix and modify the algorithms to maintain the checksum during factorization. Because ScaLAPACK mostly reuses the LAPACK algorithm and implementation for the two sided factorizations, the same design can be used in ScaLAPACK. However because the matrix is distributed onto multiple computing nodes, the checksum verification procedure would require communication that involves all the processes. To avoid the communication in checksum verification, we would like to attach checksums to the matrix local to each process instead of to the global matrix. In addition, local checksums are desirable in numerical accuracy and scalability in correcting errors. The question

is then how to modify the ScaLAPACK two-sided factorizations in such a way that local checksums are maintained in similar way that global checksums are maintained.

The guideline to modify the ScaLAPACK software follows the same principle of LAPACK; the difference is that in LAPACK the matrix-vector and matrix-matrix operations are on global matrix but in ScaLAPACK they are on local matrix. Previously we modify the matrix-vector and matrix-matrix multiplications by including the appropriate checksums global to the matrix/vector, now on the distributed matrix we modify the corresponding operations by including the appropriate checksum local to the matrix/vector. Take matrix-vector multiplication for example as illustrated in figure 5.4. Suppose we would like to perform a global matrix-vector multiplication $y = Ax$ and the matrices and vectors are distributed as illustrated in 5.4. First each process owning a part of matrix A would do the matrix-vector multiplication on its local data $y_L = A_L x_L$, and then all processes that contribute to the same part of y would sum up their contribution to form the part of global y. To modify this distributed matrix-vector multiplication to include the local checksums, we simply include the local checksums in the local matrix-vector multiplication, and then sum up the checksum encoded result vector y_L^c. Modifying distributed matrix-matrix multiplication is similar; the difference is that in matrix-matrix multiplication both row checksums and column checksums are involved.

We now claim that during the modified distributed blocked two-sided factorizations, the process *local* matrix will be properly checksum encoded just like the global

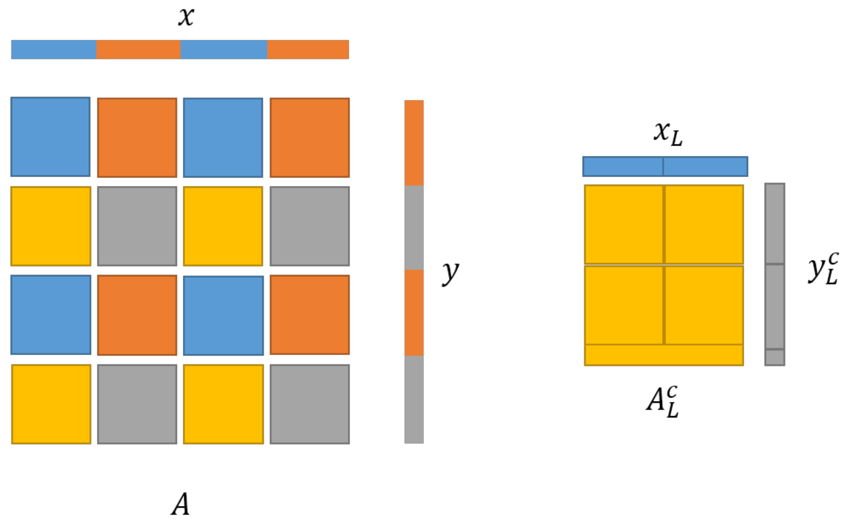


Figure 5.4: Modification to distributed matrix-vector multiplication to include local checksums

matrix is encoded in the non-distributed case. To see why this is true, we note that 1) the checksum is a weighted sum of the participating elements, and all of our discussion will work with any weights; and 2) the following insight:

> Key insight: Each process owns a portion of the global matrix. Checksum of the local matrix is equivalent to checksum of the global matrix with selective weights; see figure 5.5. This is possible because all the distributions of all the matrices (A, X, Y) are compatible.

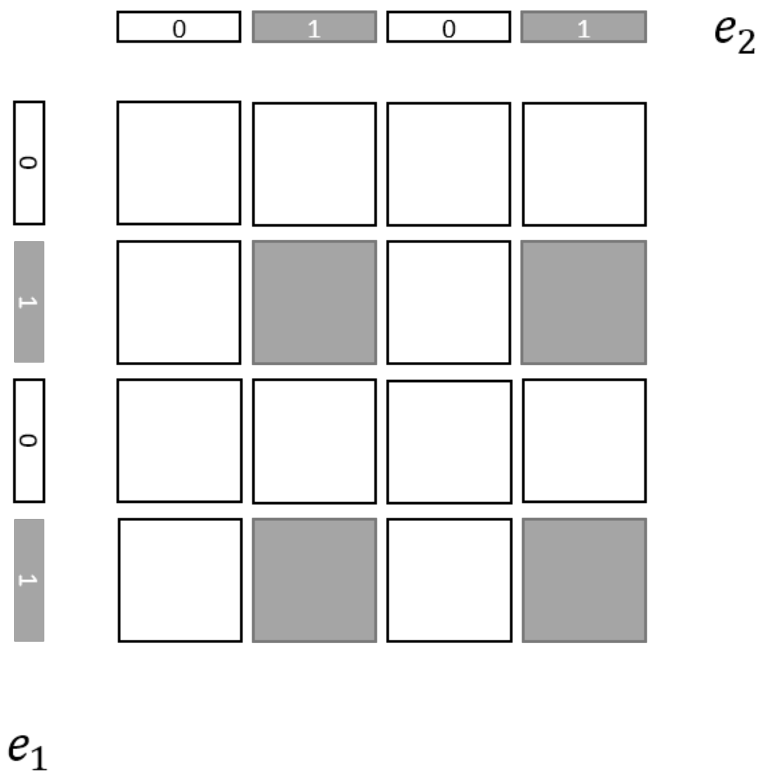


Figure 5.5: Local checksum equivalent to global checksum with specific weights.

This leads to the interesting view that each process in the distributed factorizations use its own selective weights. The global matrix thus have multiple distributed checksums with different weights.

5.5 Analysis

In this section we analyze three aspects of the algorithms we described in these aspects: 1) the fault coverage; 2) the overhead and scalability.

5.5.1 Fault coverage

In this section we first attempt to quantify the (detection) fault coverage of different error detection techniques. There are various kinds of error detection techniques; they might have different strength in detecting errors. Intuitively if we only design checksum scheme for part of the data structure and only for some operations, the fault coverage is low because corruption to data or operation outside of the protection will not be detected. On the algorithm abstraction layer we can define a metric of (algorithmic) fault coverage as portion of data structures and portion of time for which the data structures are *not* vulnerable. A data structure is vulnerable for a period of time during which a corruption to it cannot in general be detected. The corruption can be in the form of logic error (miscalculation) or bit flips. If all data structures are protected for the whole duration of the algorithm, the fault coverage is

100%. If half the data structures are protected for half of the time in the algorithm, the fault coverage is $50\% \times 50\% = 25\%$. The fault coverage conceptually reflects the probability of error detection if fault happens randomly and uniformly in space and time.

We compare the three checksum schemes for the fault coverage analysis. The first is our proposed checksum scheme without checksum regeneration. The second is the previous checksum scheme from [59, 60] that covers only the matrix-matrix multiplications with checksum regeneration. The third builds upon the second such that covers the matrix-matrix and matrix-vector multiplication with checksum regeneration. We denote the three checksum schemes by FT1, FT2, and FT3 respectively. The checksum regenerations in FT2 and FT3 are necessary because checksums are not automatically maintained across the matrix-vector/matrix operations thus have to be regenerated before the operation begins. The checksums also have to be checked before the result is overwritten and the checksum regenerated, otherwise the fault will elude detection. We will analyze the fault coverage of FT1, FT2, and FT3 based on the LAPACK `dgebrd` subroutine shown as algorithm 14.

For FT1 shown in algorithm 13 all operations except for the four matrix-vector operations in 1.(c) and 1.(f) are covered. The unprotected matrix-vector operations are the four intermediate calculation $V_{i-1}^T v_i, X_{i-1}^T v_i, U_{i-1}^T u_i, Y_i^T u_i$. At each iteration they affect at most $2M \times B$ elements, and takes

$$\sum_{k=1}^{B}(m+n-k+1)(k-1) \leq \frac{mB^2}{2}$$

Summing the above for all iterations ($m = M-N, M-N+B, M-N+2B, \ldots, M$)yields:

$$\sum_{j=1}^{(M-N)/B}(M-N+jB)B^2 \leq \frac{1}{2}BMN$$

Noting that the whole matrix takes space $M \times N$ and the whole algorithm takes FLOPs $\frac{4}{3}N^2(2M-N)$, we can estimate the fault coverage ratio of FT1:

$$\text{Cov(FT1)} \geq 1 - \frac{\frac{1}{2}BMN \times 2MB}{\frac{4}{3}N^2(2M-N) \times MN} \geq 1 - \frac{3}{4}\frac{M}{N}\left(\frac{B}{N}\right)^2$$

We see that the fault coverage of FT1 scheme is in general very high as the blocking size is usually within $[32, 128]$ and the problem matrix can be M, N can be at least 100x that of B. The ratio tends to be larger than 99.9% for reasonably large and square matrix.

For the FT2 scheme that only protects the matrix-matrix factorization, suppose the matrix is a square ($M = N$). In each iteration, the two matrix-matrix multiplication (step 2, 3 in algorithm 14) protects data size n^2 for duration of $4Bn^2$. The whole matrix has size N^2 and duration $\frac{8}{3}N^3$. Taking consideration of all iterations ($n = B, 2B, \ldots, N$) we get:

$$\text{Cov(FT2)} = \frac{\sum_{j=1}^{N/B} 4B(jB)^2 \times (jB)^2}{N^2 \times \frac{8}{3}N^2} = 30\%$$

For the FT3 scheme the fault coverage is almost 100% as the only unprotected parts is the generation of Householder vector (step 1.(b)(e) in algorithm 14) which affect very few elements for very brief time. The problem of this approach is the high overhead due to checksum regeneration and frequent verification to achieve high fault coverage which FT1 does not have. The overhead will analyzed in the next subsection and also empirically shown in the next section.

What about the error correction coverage? Detailed analysis is difficult as the whether the error is correctable depends on specific the timing and location of the error. However we note that detection fault coverage is the upper bound for correction fault coverage: errors cannot be corrected unless they are detected. We will explore the correction coverage empirically in the next section.

As a side note, the algorithmic fault coverage discussed here is based on the algorithm abstraction layer; the objects and operations are rather high level and need to be mapped to a program, and subsequently mapped to a computer instructions, and eventually to electrical circuits for execution. Some of the lower abstraction constructs such as the indices, loops, pointers in the programming language layer that are needed to carry out the algorithm are not discussed here; their misbehavior may or may not be covered by higher abstraction layer. The fault tolerance mechanisms for other abstraction layers are important in determining the resilience of the final application

execution, but they are out of the scope of this paper. Fault tolerance at different layers have their peculiar tradeoffs between complexity, cost, and effectiveness and it is desirable to integrate them to combine their individual advantages.

5.5.2 Overhead

In this subsection we analyze the overhead in introducing checksum in checksum scheme FT1, FT2, and FT3. The execution time overhead comes from three parts: 1) checksum (re)generation; 2) extra computation/communications that involves the checksums; 3) checksum verification. The memory space overhead comes from the extra space to store checksums. Suppose BLAS2 operation execution rate is α_2 and BLAS3 operation execution rate is α_3. On a typical modern CPU such as Intel Xeon α_3 is about 5x that of α_2.

For FT1, the overhead is

$$O_{\mathrm{FT1}} = \frac{\alpha_2 N^2 + (\alpha_2 + \alpha_3)/2 \times 8N^2 + \alpha_2 \frac{N^3}{3B}}{\frac{4}{3}(\alpha_2 + \alpha_3) N^3} \tag{5.5}$$

$$O_{\mathrm{FT2}} = \frac{\alpha_2 \frac{N^3}{3B}}{\frac{4}{3}(\alpha_2 + \alpha_3) N^3} \tag{5.6}$$

$$O_{\mathrm{FT3}} = \frac{\frac{4}{3}\alpha_2 N^3 + \alpha_2 \frac{N^3}{3B}}{\frac{4}{3}(\alpha_2 + \alpha_3) N^3} \tag{5.7}$$

Under reasonable assumption of $B = 32, N \approx 1000s$, the O_{FT1} and O_{FT2} should be at around a few percent, and O_{FT3} can approach 100% primarily due to the frequent checksum regeneration.

5.6 Experiments

In this section we empirically study the proposed fault tolerance two sided factorizations for the aspects discussed in the previous section.

5.6.1 Fault injections

In this subsection we empirically examine the fault coverage of the three checksum schemes FT1(proposed in this paper), FT2, and FT3. We use a fault injector to inject faults into the program and repeat this test to see if the fault can be detected. The fault injection is aimed to assess the algorithmic fault coverage therefore only floating point data and operations are targeted; the indices, pointers, and other control data and operations are not. Corruptions to the matrices and vectors seldom lead to crash of the process thus is almost always a silent data corruption; corruption to control data and operation quite likely will lead to process crash.

The fault injector we use is based on debugger GDB. It works as follows. First the target program is run under the debugger and profiled for its execution time. Then we randomly generate a timer event during the execution to stop the program. The debugger then either randomly decide to corrupt a random element in the numerical data, or modify the current floating point arithmetic instructions. If the current instruction is not a floating point instruction the debugger executes one instruction at a time until it finds a floating point instruction. The debugger then corrupt the destination register to simulate an FPU fault. We compiled the program using specific

compiler flags to only generate x87 FPU code which is easy to target, as there are fewer instruction types and x87 uses floating point registers as a stack. The result of the current x87 instruction is usually placed on the top of the stack. We also disables optimization flags to gain more information of the injection site, and as a result the matrix-matrix multiplication runs at the same speed as matrix-vector multiplication. The experimental results should be close to the analysis in the previous section that assumes the same speed for all operations. With optimized BLAS library BLAS3 is much faster than BLAS2 the fault coverage of FT2 would have been reduced.

We have implemented the three checksums schemes into the blocked serial bidiagonal reduction subroutine `DGEBRD()` in LAPACK 3.6.0. The matrix size is set to 500x500. Each program will be subject to 1000 error injections; each time a single fault is injected and the program will report whether it has detected the fault. All injections will affect the result if untreated. The results are summarized in the figure 5.6. We can see that FT1 can detect almost all errors and correct a fairly large fraction of them. FT2 can only detect less than 50% errors and correct most of the detected errors. FT3 on the other hand does not perform too well primarily due to the significant added computations of checksum regenerations that are themselves vulnerable, and the non-persistent checksums that cannot detect memory corruptions to already computed data.

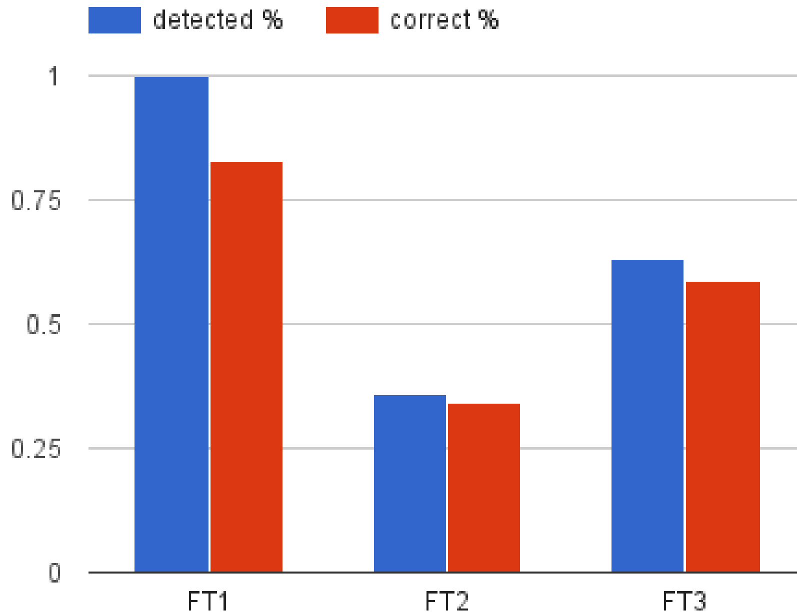


Figure 5.6: The fault coverage for the three checksum schemes

5.6.2 Execution time overheads

Here we empirically evaluate the execution time overhead of the checksum schemes. We use the same implementation as the previous subsection but here we enable aggressive optimization flags of compiler and highly optimized BLAS library [97] available on the AMD Opteron system. The problem size is set at 1000x1000 and the block size $B = 32$ as the LAPACK 3.6.0 defaults to. The execution time for the three checksum scheme implementation based on LAPACK `DGEBRD` subroutine is shown in figure 5.7. From the figure we can see that FT1 as predicted only incurs less than 5% overhead; FT2 has more overhead due to frequent checksum regeneration; and

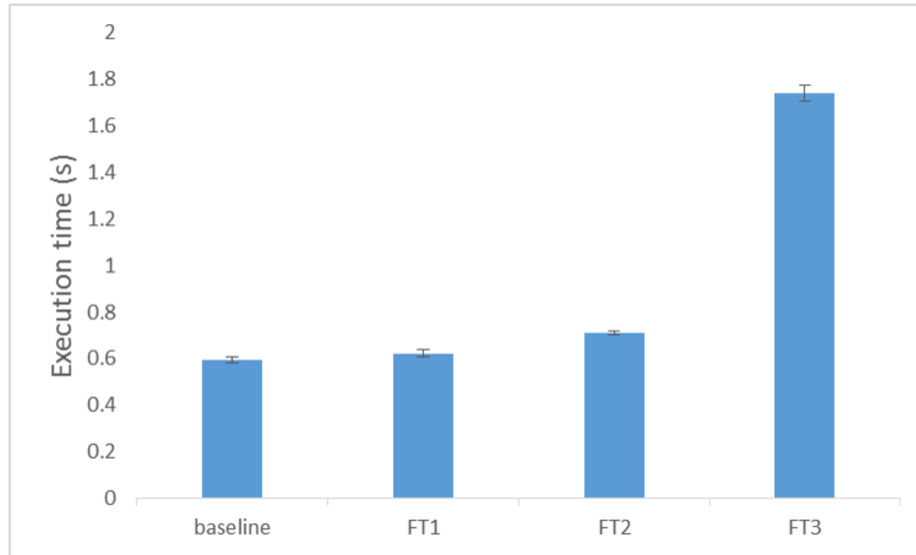


Figure 5.7: The execution time overhead for various checksum schemes compared to original `DGEBRD`

FT3 has almost 200% overhead due to the very frequent checksum regeneration that is BLAS2 operation.

5.6.3 Overheads and scalability

Now we turn to the distributed memory bidiagonal reduction subroutine `PDGEBRD` in ScaLAPACK 2.0.2 and see how the hardened version scale to more computing cores and nodes. We have two scaling tests. In the first test, we fix the memory usage per processor and scale the number of processors from 4 to 4096. In the second test, we fix the processor grid at 1024 (32x32) and scale the problem size from 500x500 to 1500x1500. This works fills the last major missing parts of dense matrix factorization algorithms in ScaLAPACK. This set of experiments are performed on the TACC

Stampede supercomputer, currently ranked at #12 on top500.org list. The Stampede machine consists of Xeon E5-2680 8C 2.7GHz processors and Infiniband FDR fat-tree network. Xeon Phi is not utilized. The compiler is intel/14.0.1.106 and MPI implementation is impi/4.1.0.030. The optimized BLAS library is MKL that comes with the Intel compilers. Each time measurement is repeated several times until the 95% confidence interval of the average (indicated as error bar) is within 5% of the average measurements [53], except for the 4096 cores case in figure 5.8 in which case the variance of the measurements are too high and repeating measurements do not seem to reduce it. The reason might be that allocating 4096 processors has a higher chance of including unhealthy nodes that are slower and drag down the performance. The high overhead 10.6% is therefore of low statistical significance. We see that in figure 5.8 and 5.9 in general the hardened FT-PDGEBRD maintains low overhead and the scalability of the original PDGEBRD subroutine in the scaling tests.

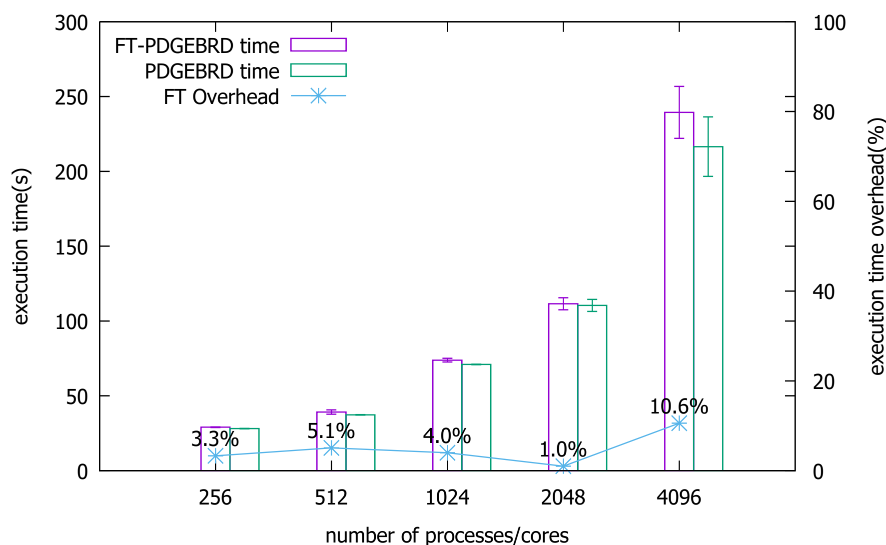


Figure 5.8: Fault free execution time for fault tolerant FT-PDGEBRD and non-fault-tolerant PDGEBRD from ScaLAPACK. Local matrix is fixed at 1000×1000; the global matrix size scales as the number of processors scales.

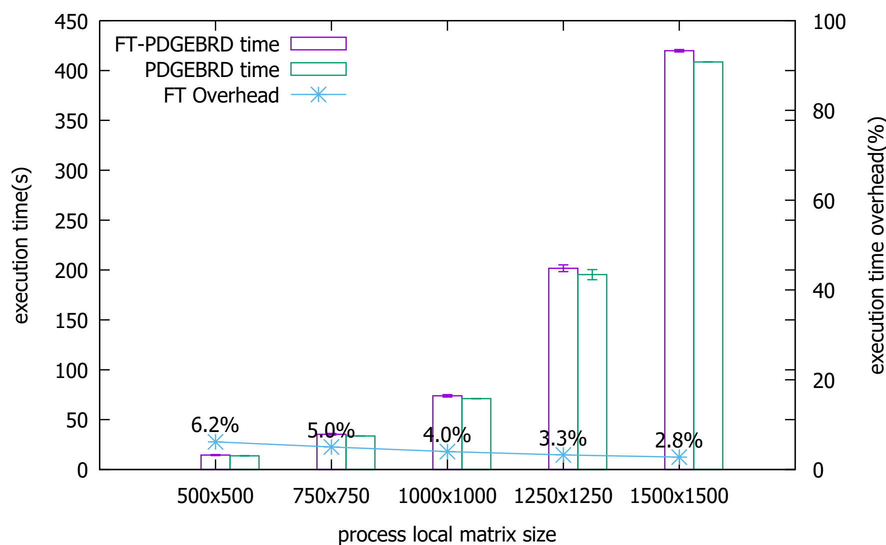


Figure 5.9: Fault free execution time for fault tolerant FT-PDGEBRD and non-fault-tolerant PDGEBRD from ScaLAPACK. Processor grid is fixed at $1024(32 \times 32)$. The global matrix size scales as the local matrix size scales.

Chapter 6

Towards Practical Algorithm Based Fault Tolerance

ABFT has first been proposed in a seminal work by Huang and Abraham [58] for matrix-matrix multiplication on systolic arrays. The idea of ABFT can be seen as an adaption of ECC to numeric structures like matrices or vectors. The significant difference is that for ECC the data is static but for ABFT the data is under transformation. In ABFT the central problem is that the codes must maintain after transformation in order to be able to detect errors using the codes. The fault model is a deciding factor in the design of ABFT codes and adaption to the associated algorithm. However the fault models used in existing ABFT research are either too abstract [33, 67, 44] or too simplistic [21, 99, 100] limiting their use where the architectural fault models do not fit. In this work we rethink the fault model and explore the challenges if

we use a comprehensive architectural fault model that allows both logic/arithmetic faults and storage faults in main memory, on-chip memory, and other datapaths. We demonstrate that with this fault model we still can design highly efficient and resilient ABFT techniques for dense linear algebra and use high performance linpack (HPL) to show that the new techniques can be implemented efficiently in complex real world high performance and highly scalable applications. The design is validated empirically by a QEMU [5] based architectural fault injector, F-SEFI [1], which implements the comprehensive fault model. We incorporate the new ABFT techniques into the latest Netlib HPL-2.1 and empirically show that the resulting FT-HPL incurs low overhead and maintains high scalability of the original HPL.

The contributions of this chapter are:

New fault model We use a fault model that allows logic faults and memory system faults that are comprehensive temporally and spatially and design ABFT schemes that can effectively detect and correct errors caused by these faults.

New checksum scheme We propose a novel process local checksum scheme, multiple checksums for error detection and correction by studying the syndrome (error patterns) caused by the faults.

Validation and software implementation We test and validate the resilience using an architectural fault injector. We implement the new ABFT schemes in the latest Netlib HPL-2.1.

6.1 Fault Model

The fault model for silent soft errors includes arithmetic faults that result in a wrong answer, for example 1+1=3. The other important fault is the memory system fault, manifesting as corrupted bits in storage cells. Memory faults could happen in main memory, in caches, registers, and other datapaths. We suppose one memory fault only affects one memory word; it can be multiple bits or single bit corruption.

It is useful to see how the architectural level faults manifest themselves in the algorithm level. Typically numerical algorithms deal with scalar numbers, vectors, and matrices. A variable may be mapped to multiple memory devices. For example the variable may be mapped to main memory, and cached in on-chip cache. It may also live in a register temporarily. The fault that affects the variable may be caused by corruptions in one of the mapped physical devices, and manifest themselves differently. For example if the main memory is corrupted, the mapped variable may read the corrupted value continuously until the memory is overwritten. If the corruption happens in cache, the variable may read incorrect value until the cache line is flushed. Therefore, a corrupted data element in program may sometimes read correct value but at other times read corrupted value.

6.2 The Checksum Scheme

It is important to make a distinction between fault and error. For our purpose, a fault is a malfunction in the architecture, such as a bit flip in memory, cache, or registers. An error is the symptom due to the fault. Thus faults are the cause and errors are what we observe that are not correct. In designing numerical algorithms, errors are erroneous floating point variables. A single bit fault may lead to multiple errors, depending on how the faulty value is used. For algorithm designers and implementers, the problem to design fault tolerant algorithms is to find ways to detect and tolerate errors. In online ABFT framework, the problem can be further specified as to detect and tolerate errors resulting from one for *every error handling interval*. In this section, we will first study the error patterns of a single fault and how to tolerate them; then we will discuss how to design checksum schemes in LU decomposition; we will discuss how to put this technique in use in the very high performance LU decomposition package HPL; last we drop the assumption of precise arithmetic and deal with finite precision floating point arithmetic.

6.2.1 Error patterns and correction

We begin by studying the error patterns caused by a single fault in matrix multiplication, as matrix multiplication is the simplest dense matrix operation and it is an important part of the LU decomposition. We will see that memory faults may lead

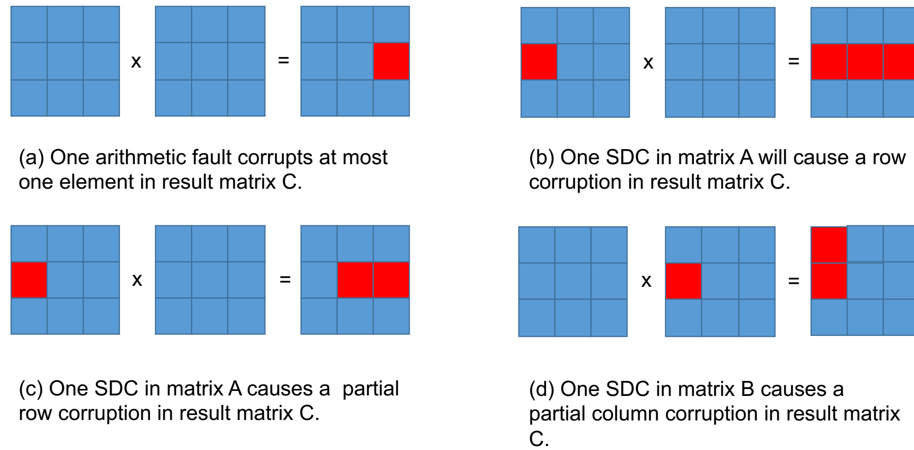
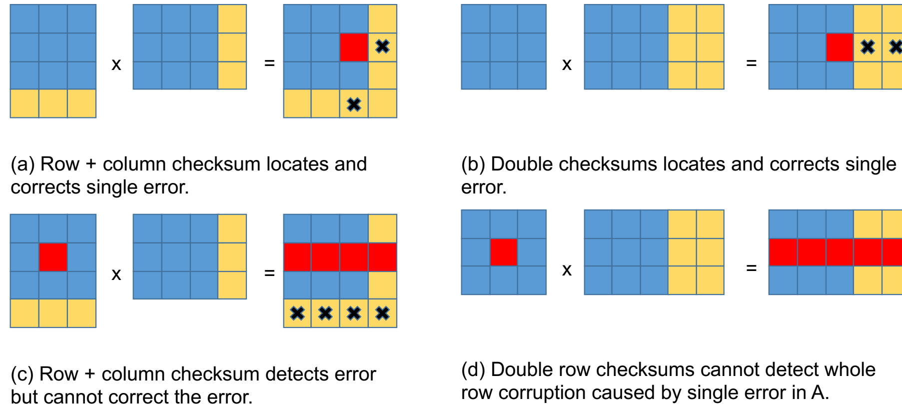


Figure 6.1: Error patterns for a single fault in matrix multiplication

to multiple errors, while in contrast one arithmetic fault will only lead to one error in matrix multiplication.

Figure 6.1 shows four cases when one fault strikes. The fault could be an arithmetic fault or a silent data corruption (SDC). The red elements indicate errors. In subfigure (a), a single arithmetic error can only corrupt one element in the result, because the intermediate value produced by the faulty arithmetic operation is only used to calculate one element. In subfigure (b), a SDC in matrix A corrupts the whole row in the result C, because the corrupted element in A is used to calculate the whole row. In subfigure (c), the SDC occurs not in memory but in for example cache, or occurs later during the matrix multiplication. In this case a single SDC in matrix A causes partial row corruptions in C. In subfigure (d) a single SDC in matrix B causes partial column corruption in C. The important observation here is that *a single fault cannot*

cause errors in more than one row or column. This observation enables us to design checksums that can correct all the error patterns caused by a single fault.



(a) Row + column checksum locates and corrects single error.

(b) Double checksums locates and corrects single error.

(c) Row + column checksum detects error but cannot correct the error.

(d) Double row checksums cannot detect whole row corruption caused by single error in A.

Figure 6.2: Checksums for matrix multiplication

Next we discuss how to design checksum schemes to detect and correct up to one fault based on the fault patterns in figure 6.1. A matrix can have two types of checkums along its two dimensions: the checksum at the bottom of a matrix is called column checksum and the checksum to the right of a matrix is called row checksum. The column checksum encoded matrix is often denoted by a superscript A^c and the row checksum encoded matrix by superscript A^r. If a matrix has both then it is called fully checksummed and denoted by A^f. Mathematically, let e be the weight vector (or matrix in the case of multiple checksums) then:

$$A^c = \begin{bmatrix} A \\ e^T A \end{bmatrix}, \quad B^r = \begin{bmatrix} B & Be \end{bmatrix}, \quad C^f = \begin{bmatrix} C & Ce \\ e^T C & \end{bmatrix}$$

As shown in figure 6.2, we have multiple configurations of checksums. The yellow blocks are row or column checksums associated with the matrix. The red block indicates an incorrect element, and a black cross on a row/column checksum indicates that the row/column checksum is inconsistent with the respective row/column in matrix. We need at least two checksums to correct up to one error because the location and the magnitude of the error are two unknowns. For a single error in a matrix, either two row checksum, two column checksum, or one row plus one column checksums can detect and correct one error in matrix C. In subfigure (a), the error can be located at the intersection of the inconsistent row and column. The error can be recovered using either the row or column checksum [58], because the *checksums are correct*. In subfigure (b), a single error in matrix C can be detected and corrected using two row (weighted) checksums with different weights [99]. The location of the error and the magnitude of the error can be solved from the two checksums. In subfigure (c), a single SDC in matrix A causes a whole row corruption that result in an incorrect but consistent row. Because the row checksum is corrupted, it leaves us with only one column checksum which is inadequate to correct the errors. In subfigure (d), a single SDC in matrix A causes a whole row corruption with incorrect but consistent checksums. In this case the checksum scheme cannot detect the errors.

It is now clear that we need both row and column checksums to avoid the error detection failure. And to correct row/column corruptions we need two row checksums and column checksums, as shown in figure 6.3. In figure 6.3, a SDC in matrix A causes

a whole row corruption in C detectable by the column checksums. The errors can be located and corrected on per column basis using the two correct column checksums. The row checksums are neither able to locate the errors nor correct them.

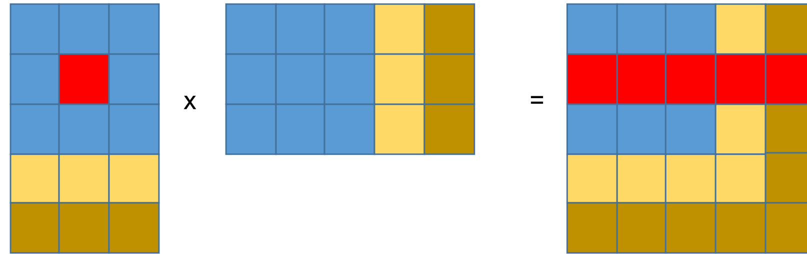


Figure 6.3: The checksum scheme that can tolerate single arithmetic fault or memory fault

Specifically, how do we locate and correct one erroneous element using two checksums? There is an easy to use encoding method. Suppose we encode a vector using two different weights $e_1 = [1, 1, \ldots, 1]^T, e_2 = [1, 2, \ldots, n]^T$. The vector is $a = [a_1, \ldots, a_n]$ and we have two correct encoded checksums of a:

$$r_1 = ae_1 = \sum_{i=1}^{n} a_i, \quad r_2 = ae_2 = \sum_{i=1}^{n} ia_i$$

Now suppose the computed $a' = [a'_1, \ldots, a'_n]$ has up to one erroneous element $a'_j \neq a_j$, where the location j is unknown to us. However when we verify the checksums:

$$\delta_1 = \sum_{i=1}^{n} a'_i - r_1 = a'_j - a_j \neq 0$$
$$\delta_2 = \sum_{i=1}^{n} ia_i - r_2 = j(a'_j - a_j) \neq 0$$

Then a simple division δ_2/δ_1 gives us the location j. The correct value of a_j can then be recovered using the correct checksum and the other correct elements of a: $a_j = a'_j - \sum_{i=1,i\neq j}^{n} a'_i$.

In this subsection the error patterns in matrix multiplication are discussed and checksums are devised to detect and correct errors, given that we have the desired checksums available. In the following subsection, how to maintain the checksums online is discussed in LU decomposition. Note that in LU decomposition the matrix multiplication is actually $C \leftarrow C - A \times B$ instead of $C \leftarrow A \times B$ so *correction through re-computation cannot be used* because the original C is overwritten.

6.2.2 Checksum scheme in LU decomposition

In this subsection the right-looking LU decomposition is briefly introduced. We first show that LU decomposition maintains global row and column checksums. Then we discuss the two adaptions to the LU decomposition that are essential in achieving good performance on modern cache based system and parallel computing.

LU decomposition factors a matrix A into the product of two triangular matrices (lower) L and (upper) U: $A \rightarrow L \times U$. The tiled right-looking variant of the LU algorithm works as shown in figure 6.4.

Figure 6.4 shows the state before and after an iteration in the algorithms. The algorithm is a series of iterations that keeps shrinking the trailing matrix until done. The yellow parts of the matrix indicate areas that have been factored and not active.

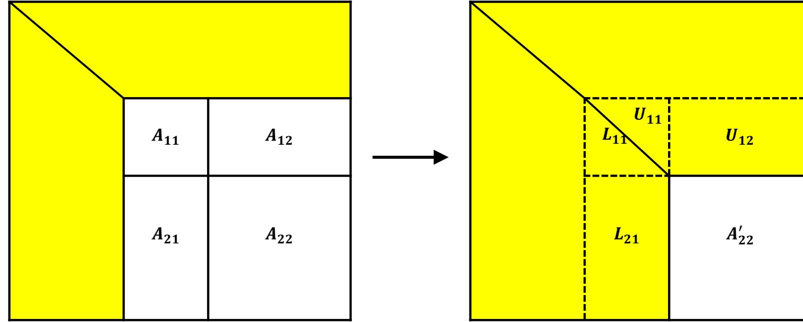


Figure 6.4: Tiled right-looking LU algorithm, one iteration

For a certain iteration, the algorithms follows three steps: left panel factorization, top panel update, and trailing matrix update, described by the following equations:

$$\begin{bmatrix} A_{11} \\ A_{21} \end{bmatrix} \rightarrow \begin{bmatrix} L_{11} \\ L_{21} \end{bmatrix} \times U_{11} \tag{6.1}$$

$$A_{12} \rightarrow L_{11} \times U_{12} \tag{6.2}$$

$$A'_{22} \leftarrow A_{22} - L_{21} \times U_{12} \tag{6.3}$$

The maintenance of checksums offline: In the original Huang and Abraham ABFT paper [58] it has been shown that if we LU decompose a full checksummed matrix A^f, we will end up with column checksummed L^c and row checksummed U^r:

$$\begin{bmatrix} A & Ae \\ e^T A & \end{bmatrix} \rightarrow \begin{bmatrix} L \\ e^T L \end{bmatrix} \times \begin{bmatrix} U & Ue \end{bmatrix} \tag{6.4}$$

where vector e is the checksum weights vector. This relationship can only be used to detect errors but not correct errors because in LU the errors will propagate to checksums too.

The maintenance of checksums online: If LU decompose a full checksum matrix, we will end up with a column checksummed L and row checksummed U. However multiple errors compound each other resulting in algorithmically uncorrectable errors. It would be desirable to detect and correct errors frequently during the factorizations to handle errors in a timely manner. In fact, we will show that at the end (or beginning) of each iteration, the factored left panel and top panel will be column checksummed and row checksummed, and the trailing matrix will be fully checksummed. We will show this claim inductively by first assuming the condition holds at the beginning of a certain iteration and prove that the condition holds at the end of the iteration. The initial condition clearly holds as we have a fully checksummed initial matrix.

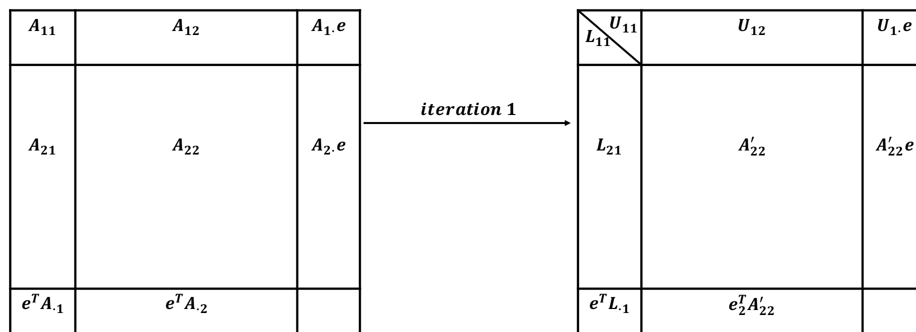


Figure 6.5: Tiled right-looking LU algorithm with checksums, one iteration

For simplicity we only examine the first iteration. As shown in figure 6.5, before the iteration we have the full checksums. After the left panel has been factorized according to equation (6.1), the column checksum associated with the left panel turns into the checksum of the factorized panel: $e^T A_{\cdot 1} \rightarrow e^T L_{\cdot 1}$. To see why this is true one only has to observe: 1) the factorized left panel will not be updated again therefore will stay unchanged through the end; 2) from equation (6.4) we know that at the end the left panel will be column checksummed. Thus we proved that the left panel factorization maintains column checksum. Similarly, the second step according to equation (6.2) maintains the row checksum of the top panel. Next we need to prove that after the trailing matrix update according to equation (6.5), the trailing matrix will be fully checksummed. To see that we only have to apply the matrix multiplication to the checksums. Take the column checksums for example. The transformation done to the column checksums is depicted by:

$$\begin{aligned} &= e^T \begin{bmatrix} A_{12} \\ A_{22} \end{bmatrix} - e^T \begin{bmatrix} L_{11} \\ L_{21} \end{bmatrix} \times U_{12} \\ &= e_1^T (A_{12} - L_{11} U_{12}) + e_2^T (A_{22} - L_{21} U_{12}) \\ &= e_2^T (A'_{22}) \end{aligned} \tag{6.5}$$

which proves that the trailing matrix is fully checksummed by the second part of the checksum weights vector e_2.

6.2.3 The complete picture as in HPL

The previous subsection discusses the algorithmic structure of tiled right-looking LU decomposition, and the maintenance of checksums at each iteration in fault free execution. In this subsection we discuss what happens when faults strikes, namely the error patterns. Once we know the error patterns we can describe correction procedures. We will also deal with two more complications in HPL: partial row pivoting for numerical stability and 2d cyclic block distributions of matrix for load balance in distributed computing.

Error patterns: We examine the error patterns in the three steps during one iteration, and discuss detection and correction procedures. First, we look at the first step and the second step according to equations (6.1) and (6.2), namely the left and top panel factorization. Our first claim is that *any single fault that occur during the left and top panel factorization will lead to inconsistent checksums*, provided that the arithmetic are precise, i.e. no round-off errors. In other words, the error detection by checksums is precise. The reason that the error detection is precise is because we have both row and column checksums. If for example only row checksums are used, as pointed out by figure 5 in [104], certain faults strike in lower triangular L will not be detected. In our case the fault will be detected by the inconsistent column checksums. Depending on the location and timing of the fault, the error pattern could be very complex and both the row and column checksums will be contaminated and there is no easy algorithmic corrections, as shown in figure 6.6 (a). For this case we can use

in-memory checkpointing and rollback specifically for the left and top panels. Once the checksum inconsistency is detected the computation can be rolled back to the beginning of the iteration. In HPL the in-memory checkpoint can be stored in the communication buffer for broadcasting L thus do not consume extra memory space. The overhead of memory copy of two panels is not significant.

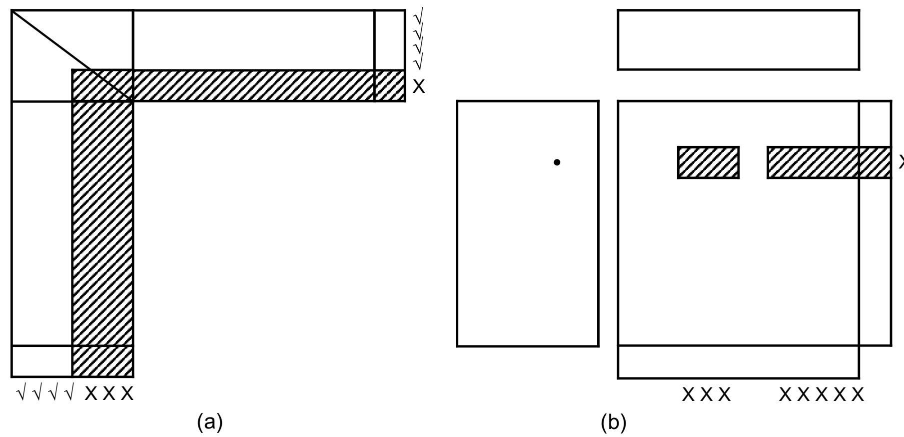


Figure 6.6: Tiled right-looking LU algorithm with checksums, one iteration. Shaded area are incorrect due to error propagation. Note that the affected checksums are also incorrect but the checksums are inconsistent therefore can be used to detect errors.

For the trailing matrix update, as discussed earlier a single arithmetic fault only affects one element in the result thus easily correctable. More interesting cases are memory faults within L_{21} or U_{12}. For a single SDC in L_{21} or U_{12}, *the errors cannot be in more than one row or one column.* Assuming precise arithmetic, a single fault will trigger at least one row checksum inconsistency and one column checksum inconsistency. Therefore the error detection in trailing matrix update is precise, and furthermore the error patterns are within our capability to correct. For example in the

case shown in figure 6.6 (b) a memory fault associated with an element in L_{21} causes partial row corruptions. In this case the errors are easily located by the intersection of the inconsistent row and column checksums and corrected by the correct column checksums. It seems that one row checksum and one column checksum is sufficient to locate and correct any single fault in the trailing matrix update. However this is not true and will be explained next.

Parallel LU decomposition and 2d cyclic block distribution: On a multiprocessor machine a matrix is usually distributed onto a PxQ grid of processes according to 2d block cyclic scheme for load balance and scalability. As shown in figure 6.7, a 4x4 block matrix is distributed onto four processes. In the previous discussion we only look at the logical (global) view of the matrix and the checksum scheme is applied to the whole matrix. This view has some drawbacks. First, the fault tolerance capability is not scalable with the size of the matrix. Second, as the checksums are associated with the global matrix that are distributed, the error detection and correction requires inter process communication. To avoid these two drawbacks, we instead apply checksums to the process local matrix rather than the global matrix. In this way, the fault tolerance capability is fixed per process, and increases proportionally with the number of processes or the size of the matrix. Error detection and correction only involve local information.

The online maintenance of the process local checksums are very similar to the global checksums. The error patterns can exhibit more patterns than that of global

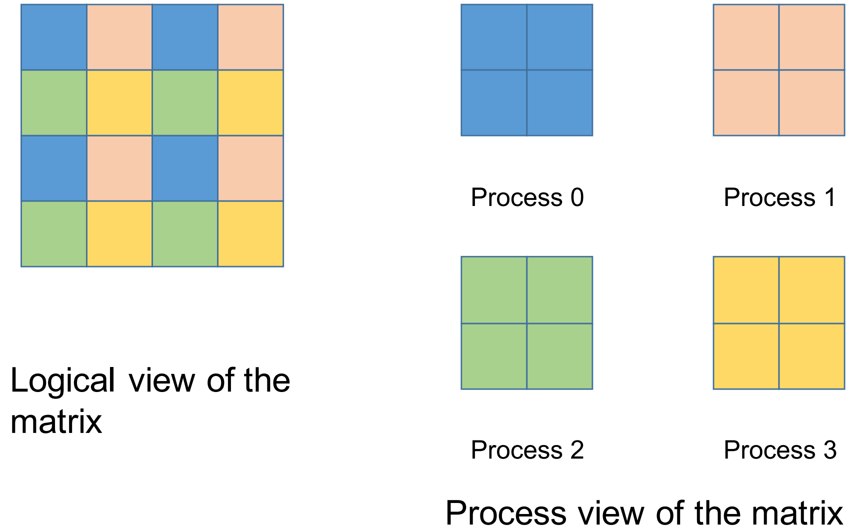


Figure 6.7: 2D block cyclic matrix distribution.

checksums. For example, consider the first iteration and the matrix distribution in figure 6.7. In the trailing matrix update, for process 0 and 2, a memory fault in left panel will always produce one inconsistent row checksum but that is not the case for process 1 and 3. For process 1 and 3, a persistent memory corruption in L causes the trailing matrix update to exhibit the error pattern shown in figure 6.2 (d) where all row checksums are incorrect but consistent. In this case a single column checksum can only detect error; two column checksums are required to correct the errors. For process 0 and 2 we show that even a persistent memory fault in L can produce one inconsistent incorrect checksums. Similar to equation (6.5) and figure 6.5, suppose after the left panel is factorized it is corrupted in one element $L_{21} \rightarrow \widehat{L_{21}} := L_{21} + \alpha e_i e_j^T$. Then the trailing matrix A_{22} and its row checksums will be updated by the corrupted $\widehat{L_{21}}$ in the following way (the symbol with a hat indicates a corruption):

$$
\begin{aligned}
\widehat{A'_{22}} &\leftarrow A_{22} - \widehat{L_{21}}U_{12} \\
\widehat{CS(A'_{22})} &\leftarrow \begin{bmatrix} A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} e_1 \\ \\ e_2 \end{bmatrix} \\
&= (A_{21} - \widehat{L_{21}}U_{11})e_1 + (A_{22} - \widehat{L_{21}}U_{12})e_2 \qquad (6.6) \\
CS(\widehat{A'_{22}}) &= \widehat{A'_{22}}e_2 = (A_{22} - \widehat{L_{21}}U_{12})e_2 \\
CS(\widehat{A'_{22}}) - \widehat{CS(A'_{22})} &= (A_{21} - \widehat{L_{21}}U_{11})e_1 \\
&= \alpha e_i e_j^T U_{11} e_1
\end{aligned}
$$

with the last equation indicating one inconsistent row checksum. Note that the equations confirm that only one row in the trailing matrix will be affected; the whole row is corrupted and so is the associated row checksum, but they are corrupted in a way that makes them inconsistent. The single row corruption can be handled by the double column checksum effectively. The above analysis also shows that the Example 3 in [104] is incorrect.

Partial row pivoting in LU: In practice unpivoted LU can easily break down due to numerical instability. To reduce the instability while not incur prohibitively high overhead, partial row pivoting is commonly used. However the row swapping in the pivoting disrupts the maintenance of the checksums. If the row checksums are swapped together with their respective rows the row checksums still maintain.

Column checksums need to be fixed and not swapped. In process local checksums however, maintaining column checksums require more care. One row may be swapped with a row from another process, thus invalidating both checksums. Therefore the checksums must be updated when inter process swapping happens.

Putting them together The pseudo code algorithm 16 summarizes the error detection and correction logic. For brevity it is in the point of view of global matrix.

Algorithm 16 The fault tolerant HPL algorithm, global view.

Require: Fully checksummed matrix A^f and right hand side b
  Ensure $x = A^{-1}b$ in the presence of floating point soft errors, or signal errors, n is the size of A, B the blocking factor
  for $i = 0$ to n step B **do**
    $A(i:n, i:n) =: \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$
    Factorize left panel $\begin{bmatrix} A_{11} \\ A_{21} \\ CS(A_{\cdot 1}) \end{bmatrix} \rightarrow \begin{bmatrix} L_{11} \backslash U_{11} \\ L_{21} \\ CS(L_{\cdot 1}) \end{bmatrix}$
    Factorize top panel $\begin{bmatrix} A_{12} & CS(A_{12}) \end{bmatrix} \rightarrow \begin{bmatrix} U_{12} & CS(U_{12}) \end{bmatrix}$
    Check column checksums for L and row checksums U
    if Errors not algorithmically correctable **then**
      Rollback to the start of this iteration
    end if
    Update the trailing matrix $A^f_{22} \leftarrow A^f_{22} - L^c_{21} U^r_{12}$
    Check and correct full checksum matrix A^f_{22}
  end for

6.2.4 Round-off error bounds

In the last section we have shown that if we limit the faults to one per error handling interval and assume precise arithmetic, the error detection is both sound and precise. In practice the floating point arithmetic are not precise, soundness and

precision cannot be attained simultaneously. As lack of soundness is not acceptable in fault tolerance, we thus strive to maintain soundness at some expense of precision. To do that, we derive *a priori* norm based error bounds for the round-off error, and use the upper bound as the threshold to distinguish architectural faults from floating point round-off errors. If the architectural faults alters the less significant bits in a floating point number and the result is still within round-off error bounds, no errors will be detected and the fault is deemed indistinguishable from round-off errors.

Specifically, when we are verifying the checksums we need to compare the calculated sums to the checksums. Because floating point arithmetic has finite precision, those two may differ even in fault free execution. Our problem is now to bound the difference that round-off errors such that round-off errors alone would not violate the bound. Consider the matrix multiplication $C = AB$. A well known norm bound of the round-off errors in matrix multiplication is as follows [46].

$$||fl(AB) - AB||_\infty \leq \gamma_n ||A||_\infty ||B||_\infty \tag{6.7}$$

Assuming that the encoded matrix multiplication $C^f = A^c B^r$ is carried correctly, and the variable with a hat represents its floating point representation, we have the

following result:

$$\begin{aligned}
\left|\sum_{j=1}^{n}\widehat{c_{ij}} - \widehat{c_{i,n+1}}\right| &= \left|\sum_{j=1}^{n}(\widehat{c_{ij}} - c_{ij}) - (\widehat{c_{i,n+1}} - c_{i,n+1})\right| \\
&\leq \left|\sum_{j=1}^{n}(\widehat{c_{ij}} - c_{ij})\right| + |(\widehat{c_{i,n+1}} - c_{i,n+1})| \\
&\leq ||fl(C^f) - C^f||_\infty \\
&\leq \gamma_n ||A^c||_\infty ||B^r||_\infty
\end{aligned} \tag{6.8}$$

where $\gamma_n = nu/(1 - nu)$ and u is the unit round-off error of the machine. For IEEE 754 64bit floating point number $u = 10^{-16}$. We thus obtained a bound of round-off errors that can be used as a threshold to distinguish architectural faults from floating point round-off errors. There is a similar bound to verify the row checksums.

6.3 Overhead, Performance, Scalability, and fault tolerance capability

In this section we model the fault tolerance capability, the execution time overhead, the scalability, and optimization of the proposed fault tolerant HPL.

6.3.1 Fault tolerance capability

For the error correction capability provided that errors can be detected, a natural question is how many errors or faults can be corrected? For each process in each error

handling interval, any number of errors during the left and top panel factorization can be tolerated by the rollback. Multiple errors or one fault can be tolerated during the trailing matrix update, provided that the errors are within one row or one column. Note that the number of faults that can be tolerated is scalable with the number of processes and problem size, so at large scale enormous number of errors or faults can be tolerated as long as the faults do not burst into one error handling interval.

Compared to online ABFT (FT-ScaLAPACK) [21, 99]: Online ABFT may fail to detect memory error in the trailing matrix update where the process is not engaging in the left panel factorization. FT-ScaLAPACK cannot correct the errors caused by faults in the left panel during the matrix multiplication.

Compared to offline ABFT (Du, Luk) [29, 33, 67] : Our FT-HPL is resilient to much more faults. For non permanently sticky memory fault, for example faults in cache or registers, offline ABFT correction based on casting the fault back to low rank perturbations to the initial matrix no longer work. In fact, any fault that do not corrupt a variable for its entire lifespan will fail in offline ABFT fault tolerance scheme, as the fault do not fit in the abstract fault model. Thus the tolerable faults in offline ABFT schemes is a small subset of the more comprehensive fault models considered in this paper.

6.3.2 Execution time overhead

The fault tolerant LU decomposition introduces overheads in maintaining checksums, checking checksums periodically, and correcting errors if detected. As the analysis here only serves as a first order approximation of the performance, we use a widely used simple machine model. The communication time is modeled as $T = \alpha + \beta L$ where α is network latency and β is the reciprocal of network bandwidth. The computation of matrices and vectors can be modeled by the product of compute rate γ and number of floating point operations (FLOPs). The compute rate of BLAS3 operation such as matrix multiplication is γ_3 and the compute rate of BLAS2 operation such as matrix vector multiplication is γ_2. On modern architectures γ_2 is much lower than γ_3 so it is important to make the distinction. Let N be the size of the matrix A, B be the blocking factor, $P \times Q$ be the dimension of the process grid, then the run time of HPL LU decomposition is as follows [80]:

$$\begin{aligned} T_{\text{hpl}} =& 2\gamma_3 \frac{N^3}{3PQ} + \beta N^2 \frac{3P+Q}{2PQ} + \\ & \alpha N \frac{(B+1)\log P + P}{B} \end{aligned} \tag{6.9}$$

Checksum maintenance overhead: The overhead of checksum maintenance can be considered as the effectively increased matrix size. Adding two row checksums and two column checksums to the process local matrix, the global checksum matrix is bounded by $\max(N(1+2P/N), N(1+2Q/N))$. In a reasonable configuration of HPL,

N/P and N/Q are the local dimensions of process local matrix that are around 10,000 therefore the enlargement of the global matrix size is around 0.02%. The resulting relative increase run time in equation 6.9 will be less than 0.1%, thus not a significant contribution to the run time overheads.

Checksum verification overhead: The periodical verification of the checksums is one major contribution to the run time overhead. The verification of checksums is a BLAS2 operation. The overhead of the verifications are:

$$
\begin{aligned}
T_{\text{check}} &= \frac{4\gamma_2}{PQ}(N^2 + (N-B)^2 + (N-2B)^2 + \cdots + B^2) \\
&= 4\gamma_2 \frac{N^3}{3BPQ}
\end{aligned}
\tag{6.10}
$$

Compared to equation 6.9 the relative overhead is

$$
\frac{T_{\text{check}}}{T_{\text{hpl}}} < \frac{2\gamma_2}{B\gamma_3} \tag{6.11}
$$

Assuming a blocking factor B around 200 and BLAS2 operation is 5x slower than BLAS3 operations, the overhead is less than 5%. Different machines will have different ratio and different relative overhead.

6.3.3 Error correction overhead

This overhead is only present when errors are detected and correctable. The algorithmic error correction using checksums are non-significant. For the errors that

are not algorithmically correctable by the checksums, the overhead is the lost work and rollback and recompute of the left panel factorization, which is empirically a small relative to the whole factorization.

6.3.4 Memory overhead

The fault tolerance needs extra memory space to store the checksums and the left panels. The extra space to store the checksums are less than 0.1% so not a significant overhead. The memory overhead of storing the left panel is more significant at $\frac{B}{N/Q}$. Again assuming a typical HPL configuration $B = 200, N/Q = 10,000$ the overhead is at 2%.

6.3.5 Impact on scalability

If we measure scalability by the parallel efficiency $\frac{T\text{ser}}{PQT\text{hpl}}$ which indicates how close it is to ideal parallel speedup, because the execution time overhead is bounded if memory usage per process is fixed and regardless of P, Q, the scalability of the fault tolerant HPL will remain the same as the original HPL which is excellent.

6.3.6 Tradeoffs between resilience and overhead

According to the overhead analysis and the detailed timing result from the experiments we found that the verification of the trailing matrix is one major overhead to the execution time. In fact when the trailing matrix verification is disabled the fault

free execution time overhead dropped by half. In this section we discuss the tradeoff between fault tolerance and overhead, and the insights to allow such tradeoffs to happen.

Let us take the point of view of one particular process. Suppose there is a grid of $P \times P$ processes and the matrix is distributed in 2d cyclic blocked manner. Since the LU decomposition works factorizes left and top panel sequentially from left to right and from top to bottom, the particular process engages into panel factorization every P iterations (in figure 6.8 $P = 4$). As we have discussed in the error patterns in matrix-matrix multiplication (TU), the errors propagate in a controlled way. In fact, if we skip the trailing matrix verification procedure at the end of iteration 1 and 2, we still can correct up to 1 fault happening during iteration 1,2, and 3 at the end of iteration 3. In this way we trade fault tolerance for reduced error checking overhead. The observation that allows us to make this tradeoff is that faults during iteration 1 will not propagate during iteration 2 and 3. However this is not true for PF as one fault in PF will propagate and cause massive errors in subsequent TU making the single fault uncorrectable. Thus the error handling procedure after each PF cannot be skipped to reduce fault tolerance.

The overhead analysis in the last section takes a global workload approach and assumes perfect load balance between the processes. But in parallel LU there is load imbalance during the panel factorizations where only a column of processes engage and other processes are waiting for the factorization result. It can be seen that PF is likely

to be on the critical path. As PF depends on the TU immediately before, that TU is also likely to be on the critical path. The TU verification before PF thus is likely to be on the critical path. In fact from the experiments we found that by disabling only the TU verification immediately before a PF (shown in figure 6.8 OPT) the overall execution time drops almost as much as by disabling *all* TU verifications altogether. This significant reduction in overhead is therefore highly desirable, however it seems to break the promise that single fault during one error handling interval is tolerable. To remedy this problem, we only need to observe that, one fault in the last TU will cause the immediate subsequent PF verification to fail. The PF can be made non-destructive and once the PF fails the checksum verification, the error handling procedure for the previous TU is automatically invoked and the PF will restart. Therefore, the best tradeoff between fault tolerance and overhead is to disable only

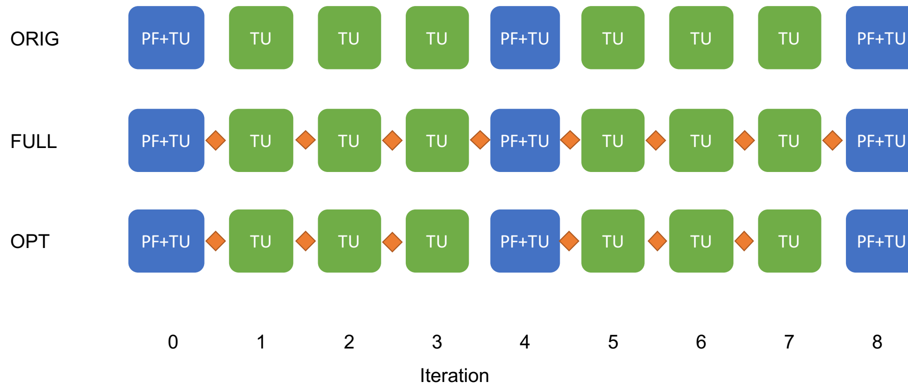


Figure 6.8: One process view in a 4x4 process grid: PF stands for (left and top) panel factorization and TU for trailing matrix update. The red diamond represents checksum verification point.

the TU verification immediately before a PF for every process. In this way the error

handling interval remains short and the critical TU verification overhead is reduced significantly.

6.4 Experimental Study

In this section we empirically evaluate: 1) the fault coverage of the proposed FT-HPL in comparison to the state-of-the-art ABFT techniques by targeted fault injection; 2) the resilience of the FT-HPL scheme and implementation by randomly injecting various faults; 3) the cost of introducing such fault tolerance by measuring large scale executions.

6.4.1 Fault injection for fault coverage

In this subsection we experimentally compare the fault coverage of the state-of-the-art ABFT techniques that can apply to LU decomposition and HPL. We inject both arithmetic faults and memory faults to various locations in code and data at various times during the execution. We select several representative stages in one iteration to inject faults. Specifically, during the first iteration of LU algorithm, we inject faults right before the iteration and in the middle of the iteration (at iteration 2 of trailing matrix update). The arithmetic fault is simulated by modifying the output of a floating point multiplication. The memory fault is injected to matrix element (2,1) by modifying the data value. To precisely control where and how to inject a

fault, we use the debugger GDB to stop the program and modify the program and data. During each run, we only inject one fault.

The fault coverage are summarized in table 6.1. As can be seen in table 6.1, no previous ABFT techniques provide as complete coverage to both arithmetic faults and memory faults happening at any time.

Table 6.1: Fault coverage for different ABFT techniques. “Before” means the fault affects data that is produced but not yet used. “Middle” means the fault affects data that is undergoing repeated use.

Fault category	Arithmetic	Memory	
Fault timing		Before	Middle
FT-HPL (this paper)	✓	✓	✓
FT-ScaLAPACK[99] /FTLU[21]	✓	✗	✗
FT-DGESV[33, 29]	✗	✓	✗

6.4.2 Fault injection experiments

We use a architectural fault injector F-SEFI [1] to implement the fault model and reveal the resilience of the FT-HPL implementation. Faults are injected at random time to a random instruction or memory locations that is to be used. Note that we inject faults into active memory to avoid masked faults that are never used. We model both floating point arithmetic faults and memory system faults. F-SEFI is based on QEMU, an architecture emulator. It works by intercepting the instructions of the application and alter the effect of the instructions to simulate arithmetic faults and

Table 6.2: Fault tolerant for dense linear algebra: costs and fault tolerance capability. "Yes" means the faults can be tolerated; "No" means otherwise. The percentage indicates the execution time overhead against non fault tolerant LU implementation (PDGESV/PDGESV in ScaLAPACK, HPL_pdgesv in HPL).

Fault category	No Error	Arithmetic Faults		Memory Faults	
Number of faults	0	≤ 2	many	≤ 2	many
FT-HPL	5%	Yes, 5%	Yes, 5-35%[a,b]	Yes, 5%	Yes, 5-35%[a,b]
FT-ScaLAPACK[99]/FTLU[21]	8%	Yes, 8%	Yes, ¿8%[b]	No	No
FT-DGESV[33, 29]	1%	Partial[c], 1%	No	Partial[c], 1%	No
RedMPI[42][d]	≥ 20%	Yes, ≥ 20%	Yes, ≥ 20%	Yes ≥ 20%	Yes,≥ 20%

[a] Overhead depends on the impacted phase in HPL.
[b] To tolerate multiple faults they must be spaced out in time thus not overwhelming one error handling interval.
[c] The fault must happen in specific time and location to fit the algebraic model in [33, 29]. See table 6.1.
[d] To tolerate faults RedMPI need 200% more processors to form TMR at MPI rank level.

memory faults. The application runs unmodified in the virtual machine and F-SEFI effectively simulate architecture correct execution (ACE) faults [78]. Memory system faults are modeled in detail: different level of stickiness associated with a memory address is used. In a cache based architecture, a variable in the program is mapped to multiple physical spaces in the memory hierarchy. When the image of the variable in different physical spaces is corrupted, the program perceives a certain stickiness of the error. For example, a corrupted main memory word is very sticky as it will read corrupted value until overwritten. On contrast, a corrupted cache word may only read corrupted value temporarily until it is flushed out, and subsequent read to the variable will read from main memory or lower level cache which has the correct value.

The configurations of the fault injection experiments are as follows. Four virtual machines are used with one MPI rank in each virtual machine. The problem size is 200x200 with blocking factor $B = 5$, which means that there are 40 intervals.

During each run of the experiment, 5 faults are injected at random times to a random memory locations that are active. We take care not to inject two faults into one error handling interval which our FT-HPL cannot handle. Note that this setting injects a considerable amount of faults into a small problem size to stress the fault tolerance mechanism.

In total 300 repetitions of the experiment are performed. Among them, 252 cases (84%) successfully tolerated the injected faults and passed the residual check of the HPL application. In all passed cases, the injected faults are detected and corrected by our algorithms. Another 21 cases (7%) run to completion but failed to pass the residual check because in HPL application not all data structures and operations can be protected by our algorithm. The remaining 27 (9%) cases crashed or hung. In contrast, when subject to 5 random memory faults both FT-ScaLAPACK/FTLU and FT-DGESV would have success rate of 0%.

6.4.3 Overheads of fault free execution and error correction

In this section we evaluate how much execution time overhead is during fault free execution, and the cost of error correction in the presence of faults. The experiments are conducted on two clusters: 1) a small cluster TARDIS (up to 512 cores) for detailed overhead reduction experiments, and 2) TACC Stampede for large scale (up to 4096 cores) scalability and overhead experiments. The TARDIS is a 16 node cluster; each node is equipped with two sockets AMD 6272 processors (32 cores) clocked

at 2.1GHz. Each node has 64 GB memory. The interconnect is Mellanox QDR InfiniBand. The TACC Stampede is currently the #10 on Top500.org November 2015 list. Each node has two Intel E5 8-core (Sandy Bridge) processors with core frequency 2.7GHz and 32 GB memory. Each core is capable to deliver 21.6GFLOP/s at maximum. The interconnect is FDR 56Gbps InfiniBand Mellanox switches using the 2-level Clos fat tree topology. Table 6.2 provides summarized comparison to state-of-the-art ABFT techniques in terms of overhead and fault coverage.

Overhead reduction and correction overhead

This set of experiments are done on TARDIS to investigate the overhead reduction effect discussed in subsection 6.3.6. In the fault free execution mode, four variants of implementations are measured: ORIG is the original unmodified Netlib HPL-2.1[80]; FULL implements the fault tolerance described in the last section; OPT implements an optimization technique that partially removes the trailing matrix checksum verification from the critical path; and FAULT is essentially FULL plus injected error that triggers all error correction procedures. In the non fault free execution, faults are injected via source code instrumentation to trigger all error checking and correction, thus demonstrating the maximum overhead of error correction. The process local matrix size is fixed at around 3000x3000, lower than a typical 10000x10000 configuration which will take much longer to complete. We use process grids $N \times 32$, with

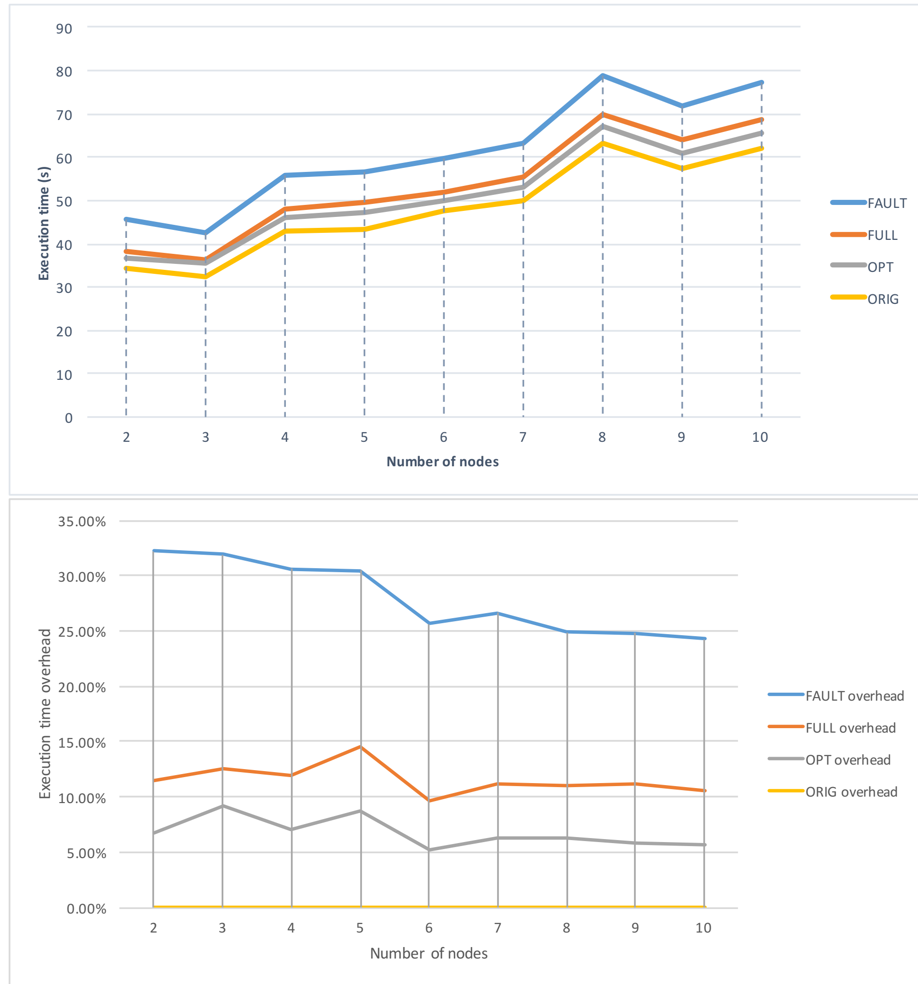


Figure 6.9: The execution time of FAULT, FULL, OPT, and ORIG HPL with varying number of nodes as X-axis. Each node comes with 32 computing cores.

the number of nodes N being from 2 to 10, and the matrix size N being from 24000 to 51000 The block size is fixed at $B = 200$.

Figure 6.9 thus shows the execution time in weak scaling experiments. It can be seen that with fault free execution, the execution time overhead can be as low as 6% compared to the non fault tolerant original HPL implementation. This is the cost paid to be able to tolerate faults that can occur during the execution. Also the

error correction procedures are very cheap and cost between 25% to 35% execution overhead at the maximum of its fault tolerance capability. Note that this is the time it takes to handle hundreds of faults or thousands of errors caused by the faults.

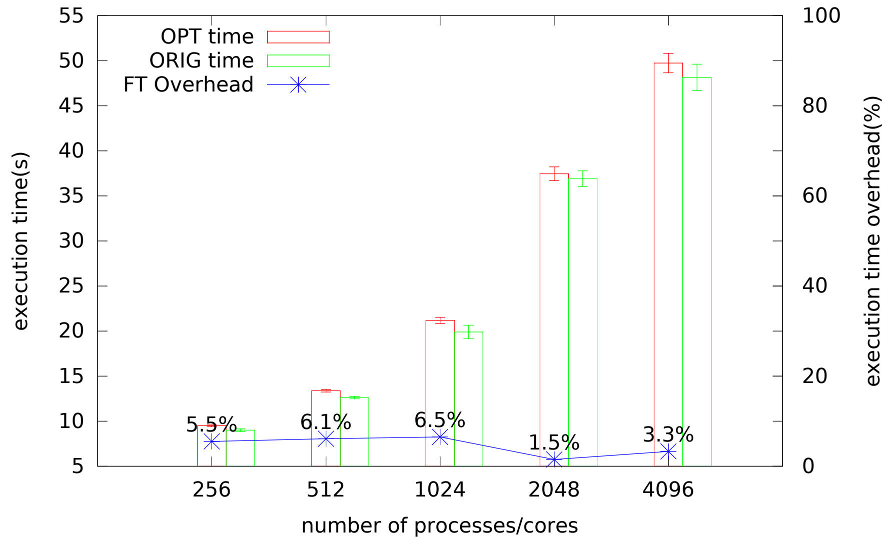
It is also worth noting that the OPT configuration has almost 50% overhead reduction over the FULL configuration which confirms the analysis that the trailing matrix verification immediately before panel factorization is on the critical path.

Scalability experiments

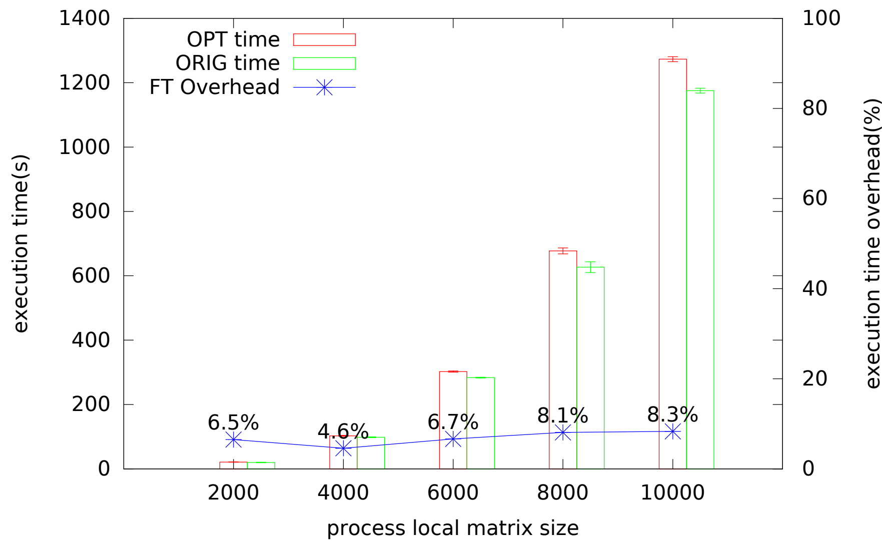
In the following texts we adopt the OPT strategy and look at the fault free overhead at large scale on TACC Stampede using up to 4096 cores (256 nodes, the maximum allowed scale without special request). For HPL the efficiency in terms of floating point operations per second (FLOP/s) per core increases when memory usage per process increases. In the first set of experiments we use only a small fraction of memory available to avoid exceedingly long experiment execution time (a single HPL run at its maximum problem size could take hours for 4096 cores). In the second set of experiments we fix the number of computing elements at 1024 cores and increase the problem size to observe the trend of overhead. From these two sets of experiments we can get an empirical idea of the overhead in introducing the resilience into HPL. The results are shown in figure 6.10

Reproducing large scale parallel experiments is difficult; so we strive to improve the interpretability [54] by providing more contexts and data. Since the execution

time of HPL on a typical computing cluster is slightly indeterministic on Stampede, we collected enough measurements until the 99% confidence interval is around 5% of the reported mean measurements, following the recommendations from [54]. Also for this particular experiments on TACC Stampede we strongly suspect that there was an abnormal node with significantly slower network interface. If such node is included in the resource allocation the job will be significantly delayed by at least 20%. We base this conclusion on the following two reasons: 1) the measurements strongly exhibit two clusters around two modes. Any one measurement belonging to one cluster will appear as outlier for the other cluster using Tukey's outlier classification method. 2) jobs involving more nodes have a higher portion of such abnormally slow measurements: for 1024 cores we got 1 every 20 measurements; for 2048 cores we got 1 every 10 measurements; for 4096 cores 1 every 2 measurements. To eliminate the interference of such slow node we remove the measurements that are abnormally slow.



(a) Fault free execution time for optimized fault-tolerant HPL (OPT) and the original HPL (ORIG). The process local matrix size is fixed at 2000×2000 while the number of processes/cores is scaling from 256 to 4096.



(b) Fault free execution time for optimized fault-tolerant HPL (OPT) and the original HPL (ORIG). The number of processes/cores is fixed at 1024 while the process local matrix size scales from 2000×2000 to 10000×10000.

Figure 6.10: Fault free execution time for optimized fault-tolerant HPL.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

In the matrix-matrix case (Chapter 3), I extended the traditional ABFT technique for fault tolerant matrix-matrix multiplication from off-line to on-line. The most prominent difference of the proposed approach is that it tolerates soft errors **on-line**. In our on-line fault tolerance approach, soft errors are detected, located, and corrected in the middle of the computation during the program execution. Because soft errors can be detected and located before they propagate and corrupted computations can be stopped and corrected in the middle of the program execution in a timely manner, computation efficiency can be improved significantly. Experimental results demonstrate that the proposed technique can correct one error every

ten seconds with negligible (i.e., less than 1%) performance penalty over the `ATLAS dgemm()`.

In the one-sided matrix factorization case (Chapter 4) I proposed a systematic way to design and reason about on-line ABFT that is resilient to soft errors for all three one-sided factorizations LU, QR, and Cholesky. By following a few principles, it is not hard to obtain correct and efficient on-line checksum schemes for LU, QR, and Cholesky subroutines used in ScaLAPACK and potentially many other block algorithms that share similar structure with them. We also showed that this approach can lead to efficient and easy to use implementations.

In Chapter 5, it is shown the current state-of-the-art algorithm based algorithm fault tolerance for two-sided matrix factorizations are not efficient. Since no obvious top-down checksum scheme exist such as those for one-sided factorization, bottom-up checksum schemes are either low in fault coverage or unnecessarily expensive due to the lack of automatic maintenance of checksums across lower level operations. In this work we explore a systematic way to modify three two-sided factorizations used in production codes LAPACK 3.6.0 and ScaLAPACK 2.0.2 to maintain checksums resulting in very low overhead and high fault coverage. Based on a modified Householder transformation, the two-sided matrix factorizations can be turned into coherent checksum preserving. Adaptions to blocked algorithms and distributed memory cases are also possible following guidelines obtained from the unblocked one.

Fault model is the deciding factor on design of ABFT algorithms (Chapter 6). To make ABFT towards practically useful we seek to close the gap between what occurs at the architecture level and what the algorithm expects. We explore the challenges in designing ABFT algorithms under a general architectural fault model that allows both arithmetic and memory system faults comprehensive both temporally and spatially. By dividing the execution into many error handling intervals and aim at tolerating single fault in each error handling interval, we build a process local checksum scheme that achieves scalable fault tolerance (one fault per iteration per process) at around 5% fault free execution time overhead and less than 35% execution time overhead when facing maximum number of faults. Targeted fault injection shows that the comprehensive fault model cannot be handled by existing state-of-the-art ABFT techniques but will be effectively tolerated by FT-HPL scheme. Random fault injection shows that our FT-HPL implementation can tolerate 84% of the cases where 5 faults occur within less than 1 second. Such low overhead and high fault tolerance under comprehensive fault model makes the new ABFT in dense linear algebra practical and attractive in extreme scale systems, on unreliable commodity hardwares, or in hostile environments.

7.2 Future Work

This line of research can be continued in several directions. The first one is that the ideas present here is for silent data corruptions; however the online checksum

maintainance means that it is possible to also recover from fail-stop errors (such as process kill). The second direction is that more algorithms or variants of algorithms can be covered using the same basic ideas explored here. There are many algorithmic progress in this area that rearranges the algorithm in some ways to increase execution speed; these algorithms need to be covered. The third direction is the combination with orthogonal techniques to provide protection for non-numerical data and control, as only numerical data structures are protected in this dissertation.

Bibliography

[1] F-SEFI: A Fine-Grained Soft Error Fault Injection Tool for Profiling Application Vulnerability. In *IPDPS'14*.

[2] C.J. Anfinson and F.T. Luk. A linear algebraic model of algorithm-based fault tolerance, 1988.

[3] Cynthia J. Anfinson and Franklin T. Luk. A linear algebraic model of algorithm-based fault tolerance. *IEEE Trans. Computers*, 37(12):1599–1604, 1988.

[4] Leonardo Bautista-Gomez, Seiji Tsuboi, Dimitri Komatitsch, Franck Cappello, Naoya Maruyama, and Satoshi Matsuoka. FTI. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis on - SC '11*, page 1, New York, New York, USA, 2011. ACM Press.

[5] Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. In *USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.

[6] L S Blackford, J. Choi, A Cleary, and E D'Azevedo. ScaLAPACK user's guide. 1997.

[7] L S Blackford, J. Choi, A Cleary, J Demmel, I Dhillon, J Dongarra, S Hammarling, G Henry, A. Petitet, K Stanley, D. Walker, and R C Whaley. ScaLAPACK: A Portable Linear Algebra Library for Distributed Memory Computers - Design Issues and Performance. In *Supercomputing, 1996. Proceedings of the 1996 ACM/IEEE Conference on*, page 5. IEEE Computer Society, 1996.

[8] L. S. Blackford, J. Society for Industrial and Applied Mathematics., A. Cleary, E. D'Azeuedo, J. Demmel, I. Dhillon, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and Jack J. Whaley, R. C./Dongarra. *ScaLAPACK user's guide*. SIAM, 1997.

[9] Aurelien Bouteiller, Thomas Herault, George Bosilca, Peng Du, and Jack Dongarra. Algorithm-based Fault Tolerance for Dense Matrix Factorizations, Multiple Failures and Accuracy.

[10] C. Braun, S. Halder, and H.J. Wunderlich. A-ABFT: Autonomous Algorithm-Based Fault Tolerance for Matrix Multiplications on Graphics Processing Units. pages 443–454, jun 2014.

[11] Greg Bronevetsky and Bronis de Supinski. Soft error vulnerability of iterative linear algebra methods. *Proceedings of the 22nd annual international conference on Supercomputing - ICS '08*, page 155, 2008.

[12] Kurt Bryan and Tanya Leise. The $25,000,000,000 Eigenvector: The Linear Algebra behind Google. *http://dx.doi.org/10.1137/050623280*, 2006.

[13] Franck Cappello, Al Geist, William Gropp, Sanjay Kale, Bill Kramer, and Marc Snir. Toward Exascale Resilience: 2014 update. *Supercomputing frontiers and innovations*, 1(1):5–28, jun 2014.

[14] Marc Casas, Bronis R. de Supinski, Greg Bronevetsky, and Martin Schulz. Fault Resilience of the Algebraic Multi-grid Solver. ICS '12, pages 91–100, New York, NY, USA, 2012. ACM.

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

[16] Zizhong Chen. Online-ABFT: An Online Algorithm Based Fault Tolerance Scheme for Soft Error Detection in Iterative Methods. PPoPP '13, pages 167–176, New York, NY, USA, 2013. ACM.

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

[18] J. Choi, J.J. Dongarra, L S Ostrouchov, and A.P. Petitet. Design and Implementation of the ScaLAPACK LU, QR, and Cholesky Factorization Routines - Scientific Programming - Volume 5, Number 3 / 1996 - IOS Press. *Scientific ...*, 1996.

[19] Teresa Davies and Zizhong Chen. Correcting soft errors online in LU factorization. In *HPDC '13: Proceedings of the 22nd international symposium on High-performance parallel and distributed computing*. ACM Request Permissions, June 2013.

[20] Teresa Davies and Zizhong Chen. Correcting soft errors online in LU factorization. pages 167–178. ACM, 2013.

[21] Teresa Davies and Zizhong Chen. Correcting soft errors online in LU factorization. In *Proceedings of the 22nd international symposium on High-performance parallel and distributed computing*, pages 167–178. ACM, 2013.

[22] Teresa Davies, Christer Karlsson, Hui Liu, Chong Ding, and Zizhong Chen. High Performance Linpack Benchmark : A Fault Tolerant Implementation without Checkpointing. pages 162–171, 2011.

[23] Nathan DeBardeleben, James Laros, John T Daly, Stephen L Scott, Christian Engelmann, and Bill Harrod. High-end computing resilience: Analysis of issues facing the HEC community and path-forward for research and development. *Whitepaper*, 2009.

[24] Nathan DeBardeleben, James Laros, John T Daly, Stephen L Scott, Christian Engelmann, and Bill Harrod. High-end computing resilience: Analysis of issues facing the HEC community and path-forward for research and development. *Whitepaper*, 2009.

[25] Jack J. Dongarra, Danny C. Sorensen, and Sven J. Hammarling. Block reduction of matrices to condensed forms for eigenvalue computations. *Journal of Computational and Applied Mathematics*, 27(1-2):215–227, sep 1989.

[26] P Du. Hard and Soft Error Resilience for One-sided Dense Linear Algebra Algorithms. Technical report, 2012.

[27] Peng Du, Aurelien Bouteiller, George Bosilca, Thomas Herault, and Jack Dongarra. Algorithm-based Fault Tolerance for Dense Matrix Factorizations. PPoPP '12, pages 225–234, New York, NY, USA, 2012. ACM.

[28] Peng Du, Aurelien Bouteiller, George Bosilca, Thomas Herault, and Jack Dongarra. Algorithm-based fault tolerance for dense matrix factorizations. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, pages 225–234, New York, NY, USA, 2012. ACM.

[29] Peng Du, P. Luszczek, and J. Dongarra. High Performance Dense Linear System Solver with Soft Error Resilience. In *2011 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 272–280, September 2011.

[30] Peng Du, P Luszczek, and J Dongarra. High Performance Linear System Solver with Resilience to Multiple Soft Errors. In *UT-CS report (UT-CS-11-683)*, pages 272–280, 2011.

[31] Peng Du and Piotr Luszczek. High Performance Dense Linear System Solver with Resilience to Multiple Soft Errors. *Procedia Computer Science*, 9:216–225, 2012.

[32] Peng Du, Piotr Luszczek, and Jack Dongarra. High performance dense linear system solver with soft error resilience. In *CLUSTER*, pages 272–280, 2011.

[33] Peng Du, Piotr Luszczek, and Jack Dongarra. High Performance Dense Linear System Solver with Resilience to Multiple Soft Errors. *Procedia Computer Science*, 9:216–225, 2012.

[34] Peng Du, Piotr Luszczek, Stan Tomov, and Jack Dongarra. Soft error resilient QR factorization for hybrid system with GPGPU. *Proceedings of the second workshop on Scalable algorithms for large-scale systems - ScalA '11*, page 11, 2011.

[35] Peng Du, Piotr Luszczek, Stan Tomov, and Jack Dongarra. Soft error resilient QR factorization for hybrid system with GPGPU. *Journal of Computational Science*, March 2013.

[36] E. N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A Survey of Rollback-recovery Protocols in Message-passing Systems. *ACM Comput. Surv.*, 34(3):375–408, sep 2002.

[37] D. Ernst, N. S. Kim, S. Das, S. Pant, R. Rao, T. Pham, C. Ziesler, D. Blaauw, T. Austin, K. Flautner, and T. Mudge. Razor: A low-power pipeline based on circuit-level timing speculation. In *Proc. MICRO*, pages 7–18, 2003.

[38] Kurt Ferreira, Rolf Riesen, Ron Oldfield, Jon Stearley, James Laros, Kevin Pedretti, and Ron Brightwell. rMPI : Increasing Fault Resiliency in a Message-Passing Environment. 2011.

[39] David Fiala. Detection and Correction of Silent Data Corruption for Large-Scale High-Performance Computing. *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, pages 2069–2072, may 2011.

[40] David Fiala. Detection and Correction of Silent Data Corruption for Large-Scale High-Performance Computing. In *Distributed Processing, Workshops and Phd Forum (IPDPSW)*, pages 2069–2072. IEEE, 2011.

[41] David Fiala, Frank Mueller, Christian Engelmann, Rolf Riesen, Kurt Ferreira, and Ron Brightwell. Detection and Correction of Silent Data Corruption for Large-scale High-performance Computing. SC '12, pages 78:1–78:12, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[42] David Fiala, Frank Mueller, Christian Engelmann, Rolf Riesen, Kurt Ferreira, and Ron Brightwell. Detection and Correction of Silent Data Corruption for Large-scale High-performance Computing. SC '12, pages 78:1–78:12, Los Alamitos, CA, USA, 2012.

[43] P Fitzpatrick and C.C Murphy. Fault tolerant matrix triangularization and solution of linear systems of equations. In *Application Specific Array Processors, 1992. Proceedings of the International Conference on*, pages 469–480, 1992.

[44] P. Fitzpatrick and C.C. Murphy. Fault tolerant matrix triangularization and solution of linear systems of equations. In *Proceedings of the International Conference on Application Specific Array Processors, 1992*, pages 469–480, August 1992.

[45] Gene Golub and Charles Van Loan. *Matrix computations (3rd ed.)*. Johns Hopkins University Press, 1996.

[46] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. JHU Press, December 2012.

[47] John A Gunnels, Daniel S Katz, Enrique S Quintana-Orti, and R A Van de Gejin. Fault-tolerant high-performance matrix multiplication: Theory and practice. In *Dependable Systems and Networks, 2001. DSN 2001. International Conference on*, pages 47–56. IEEE, 2001.

[48] Y. Guo, D. Zhu, and H. Aydin. Generalized standby-sparing techniques for energy-efficient fault tolerance in multiprocessor real-time systems. In *Proc. RTCSA*, pages 62–71, 2013.

[49] Doug Hakkarinen, Panruo Wu, and Zizhong Chen. Fail-Stop Failure Algorithm-Based Fault Tolerance for Cholesky Decomposition. *IEEE Transactions on Parallel and Distributed Systems*, 26(5):1323–1335, 2015.

[50] R. W. Hamming. Error Detecting and Error Correcting Codes. *Bell System Technical Journal*, 29(2):147–160, apr 1950.

[51] R. W. Hamming. On the Distribution of Numbers. *Bell System Technical Journal*, 49(8):1609–1625, oct 1970.

[52] Paul H Hargrove, Jason C Duell, Geist A et Al, Bode B et Al, Hargrove P Duell J, Roman E, Roman E, Zhong H J, Nieh, Su G Osman S, Subhraveti D, Nieh J, Hargrove P Duell J, Roman E, Barrett B Lumsdaine A Duell J Hargrove P Sankaran S, Squyres J M, and Roman E. Berkeley lab checkpoint/restart (BLCR) for Linux clusters. *Journal of Physics: Conference Series*, 46(1):494–499, sep 2006.

[53] Torsten Hoefler and Roberto Belli. Scientific Benchmarking of Parallel Computing Systems: Twelve Ways to Tell the Masses when Reporting Performance Results. SC '15, pages 73:1–73:12, New York, NY, USA, 2015. ACM.

[54] Torsten Hoefler and Roberto Belli. Scientific Benchmarking of Parallel Computing Systems: Twelve Ways to Tell the Masses when Reporting Performance Results. SC '15, pages 73:1–73:12, New York, NY, USA, 2015.

[55] M. Y. Hsiao. A Class of Optimal Minimum Odd-weight-column SEC-DED Codes. *IBM Journal of Research and Development*, 14(4):395–401, jul 1970.

[56] Kuang-Hua Huang and Abraham. Algorithm-Based Fault Tolerance for Matrix Operations. *IEEE Transactions on Computers*, C-33(6):518–528, June 1984.

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

[58] Kuang-hua Huang and Jacob A Abraham. Algorithm-Based Fault Tolerance for Matrix Operations. *IEEE Transactions on Computers*, C-33(6):518–528, jun 1984.

[59] Yulu Jia, George Bosilca, Piotr Luszczek, and Jack J. Dongarra. Parallel reduction to hessenberg form with algorithm-based fault tolerance. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis on - SC '13*, pages 1–11, New York, New York, USA, nov 2013. ACM Press.

[60] Yulu Jia, Piotr Luszczek, George Bosilca, and Jack J. Dongarra. CPU-GPU hybrid bidiagonal reduction with soft error resilience. In *Proceedings of the Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems - ScalA '13*, pages 1–5, New York, New York, USA, nov 2013. ACM Press.

[61] Jien-Chung Lo. Reliable floating-point arithmetic algorithms for error-coded operands. *IEEE Transactions on Computers*, 43(4):400–412, apr 1994.

[62] H. Kaul, M. Anders, S. Hsu, A. Agarwal, R. Krishnamurthy, and S. Borkar. Near-threshold voltage (NTV) design – opportunities and challenges. In *Proc. DAC*, pages 1153–1158, 2012.

[63] D Li, C Zizhong, W Panruo, and S Vetter Jeffrey. Rethinking Algorithm-Based Fault Tolerance with a Cooperative Software-Hardware Approach. *International Conference for High Performance Computing, Networking, Storage and Analysis*, 2013.

[64] Sheng Li, Ke Chen, Ming-Yu Hsieh, Naveen Muralimanohar, Chad D. Kersey, Jay B. Brockman, Arun F. Rodrigues, and Norman P. Jouppi. System implications of memory reliability in exascale computing. In *Proceedings of 2011*

International Conference for High Performance Computing, Networking, Storage and Analysis on - SC '11, page 1, New York, New York, USA, 2011. ACM Press.

[65] Franklin T. Luk and Haesun Park. An Analysis of Algorithm-based Fault Tolerance Techniques. *J. Parallel Distrib. Comput.*, 5(2):172–184, apr 1988.

[66] Franklin T Luk and Haesun Park. An analysis of algorithm-based fault tolerance techniques. *Journal of Parallel and Distributed Computing*, 5(2):172–184, April 1988.

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

[68] Diego Lunardini, Balaji Narasimham, Vishwa Ramachandran, Varadarajan Srinivasan, Ronald D Schrimpf, and William H Robinson. A performance comparison between hardened-by-design and conventional-design standard cells. In *2004 workshop on radiation effects on components and systems, radiation hardening techniques and new developments*, 2004.

[69] James L. Massey and Oscar N. García. Error-Correcting Codes in Computer Arithmetic. In *Advances in Information Systems Science*, pages 273–326. Springer US, Boston, MA, 1972.

[70] T.C. May and Murray H. Woods. Alpha-particle-induced soft errors in dynamic memories. *IEEE Transactions on Electron Devices*, 26(1):2–9, jan 1979.

[71] Patrick J Meaney, Scott B Swaney, Pia N Sanda, and Lisa Spainhower. Ibm z990 soft error detection and recovery. *IEEE Transactions on Device and Materials Reliability*, 5(3):419–427, 2005.

[72] Sarah E Michalak, Kevin W Harris, Nicolas W Hengartner, Bruce E Takala, Stephen Wender, and Others. Predicting the number of fatal soft errors in Los Alamos National Laboratory's ASC Q supercomputer. *Device and Materials Reliability, IEEE Transactions on*, 5(3):329–335, 2005.

[73] Subhasish Mitra, Tanay Karnik, Norbert Seifert, and Ming Zhang. Logic soft errors in sub-65nm technologies design and CAD challenges. In *Proceedings of the 42nd annual conference on Design automation - DAC '05*, page 2, New York, New York, USA, 2005. ACM Press.

[74] Subhasish Mitra, Ming Zhang, T. M. Mak, Norbert Seifert, Victor Zia, and Kee Sup Kim. Logic soft errors a major barrier to robust platform design. In *Proceedings - International Test Conference*, volume 2005, pages 687–696. IEEE, 2005.

[75] Kartik Mohanram and Nur A Touba. Cost-effective approach for reducing soft error failure rate in logic circuits. In *ITC*, volume 3, pages 893–901, 2003.

[76] A Moody, G. Bronevetsky, K. Mohror, and B.R. de Supinski. Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System. pages 1–11, nov 2010.

[77] Shubu Mukherjee. *Architecture design for soft errors*. Morgan Kaufmann, 2011.

[78] Shubu Mukherjee. *Architecture design for soft errors*. Morgan Kaufmann, 2011.

[79] Eugene Normand. Single event upset at ground level. *IEEE transactions on Nuclear Science*, 43(6):2742–2750, 1996.

[80] Antoine Petitet, R. Clint Whaley, Jack Dongarra, and A Cleary. HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers.

[81] J.S. Plank, Youngbae Kim Youngbae Kim, and J.J. Dongarra. Algorithm-based diskless checkpointing for fault tolerant matrix\noperations. In *Twenty-Fifth International Symposium on Fault-Tolerant Computing. Digest of Papers*, pages 1–20. IEEE Comput. Soc. Press, 1995.

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

[83] P. Pop, K. H. Poulsen, V. Izosimov, and P. Eles. Scheduling and voltage scaling for energy/reliability trade-offs in fault-tolerant time-triggered embedded systems. In *Proc. CODES+ISSS*, pages 233–238, 2007.

[84] Rajeev R Rao, David Blaauw, and Dennis Sylvester. Soft error reduction in combinational logic using gate resizing and flipflop selection. In *2006 IEEE/ACM International Conference on Computer Aided Design*, pages 502–509. IEEE, 2006.

[85] Irving S Reed and Gustave Solomon. Polynomial codes over certain finite fields. *Journal of the society for industrial and applied mathematics*, 8(2):300–304, 1960.

[86] John Sartori and Rakesh Kumar. Architecting processors to allow voltage/reliability tradeoffs. *2011 Proceedings of the 14th International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, pages 115–124, 2011.

[87] John Sartori, Joseph Sloan, and Rakesh Kumar. Stochastic Computing : Embracing Errors in Architecture and Design of Processors and Applications. *Proceedings of the 14th International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, pages 135–144, 2011.

[88] Lin Shu and Daniel J Costello. Error control coding: fundamentals and applications. *Englewood Cliffs, New Jersey*, 1983.

[89] C.W. Slayman. Cache and memory error detection, correction, and reduction techniques for terrestrial servers and workstations. *IEEE Transactions on Device and Materials Reliability*, 5(3):397–404, sep 2005.

[90] Joseph Sloan, David Kesler, Rakesh Kumar, and Ali Rahimi. A numerical optimization-based methodology for application robustification: Transforming applications for error tolerance. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 161–170, jun 2010.

[91] Joseph Sloan, Rakesh Kumar, and Greg Bronevetsky. Algorithmic approaches to low overhead fault detection for sparse linear algebra. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*, pages 1–12. IEEE, jun 2012.

[92] Joseph Sloan, Rakesh Kumar, and Greg Bronevetsky. An algorithmic approach to error localization and partial recomputation for low-overhead fault tolerance. pages 1–12. IEEE, 2013.

[93] Marc Snir and et. al. Addressing failures in exascale computing. *International Journal of High Performance Computing Applications*, 28(2):129–173, May 2014.

[94] Marc Snir, Robert W. Wisniewski, Jacob A. Abraham, Sarita V. Adve, Saurabh Bagchi, Pavan Balaji, Jim Belak, Pradip Bose, Franck Cappello, Bill Carlson, Andrew A. Chien, Paul Coteus, Nathan A. DeBardeleben, Pedro C. Diniz, Christian Engelmann, Mattan Erez, Saverio Fazzari, Al Geist, Rinku Gupta, Fred Johnson, Sriram Krishnamoorthy, Sven Leyffer, Dean Liberty, Subhasish Mitra, Todd Munson, Rob Schreiber, Jon Stearley, and Eric Van Hensbergen. Addressing failures in exascale computing. *International Journal of High Performance Computing Applications*, 28(2):129–173, may 2014.

[95] Michael Turmon, Robert Granat, Daniel Katz, and John Lou. Tests and tolerances for high-performance software-implemented fault detection. *IEEE Trans. Computers*, 52(5):579–591, 2003.

[96] John Von Neumann. Probabilistic logics and the synthesis of reliable organisms from unreliable components. *Automata Studies*, 34(34):43–98, 1956.

[97] Qian Wang, Xianyi Zhang, Yunquan Zhang, and Qing Yi. AUGEM. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis on - SC '13*, pages 1–12, New York, New York, USA, nov 2013. ACM Press.

[98] James Hardy Wilkinson, James Hardy Wilkinson, and James Hardy Wilkinson. *The algebraic eigenvalue problem*, volume 87. Clarendon Press Oxford, 1965.

[99] Panruo Wu and Zizhong Chen. FT-ScaLAPACK: Correcting Soft Errors On-line for ScaLAPACK Cholesky, QR, and LU Factorization Routines. HPDC '14, pages 49–60, New York, NY, USA, 2014. ACM.

[100] Panruo Wu, Chong Ding, Longxiang Chen, Teresa Davies, Christer Karlsson, and Zizhong Chen. On-line soft error correction in matrix-matrix multiplication. *Journal of Computational Science*, 4(6):165–172, nov 2013.

[101] Panruo Wu, Chong Ding, Longxiang Chen, Feng Gao, Teresa Davies, Christer Karlsson, and Zizhong Chen. Fault tolerant matrix-matrix multiplication. In *Proceedings of the second workshop on Scalable algorithms for large-scale systems - ScalA '11*, page 25, New York, New York, USA, 2011. ACM Press.

[102] Panruo Wu, Chong Ding, Longxiang Chen, Feng Gao, Teresa Davies, Christer Karlsson, and Zizhong Chen. Fault tolerant matrix-matrix multiplication: correcting soft errors on-line. In *ScalA '11: Proceedings of the second workshop on Scalable algorithms for large-scale systems*. ACM Request Permissions, November 2011.

[103] Panruo Wu, Qiang Guan, Nathan DeBardeleben, Sean Blanchard, Dingwen Tao, Xin Liang, Jieyang Chen, and Zizhong Chen. Towards Practical Algorithm Based Fault Tolerance in Dense Linear Algebra. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing - HPDC '16*, pages 31–42, New York, New York, USA, 2016. ACM Press.

[104] Erlin Yao, Jiutian Zhang, Mingyu Chen, Guangming Tan, and Ninghui Sun. Detection of soft errors in LU decomposition with partial pivoting using algorithm-based fault tolerance. *International Journal of High Performance Computing Applications*, page 1094342015578487, April 2015.

[105] Yaqi Zhang, Ralph Nathan, and Daniel J. Sorin. Reduced Precision Checking to Detect Errors in Floating Point Arithmetic. oct 2015.

[106] D. Zhu and H. Aydin. Energy management for real-time embedded systems with reliability requirements. In *Proc. ICCAD*, pages 528–534, 2006.

[107] D. Zhu, R. Melhem, and D. Mossé. The effects of energy management on reliability in real-time embedded systems. In *Proc. ICCAD*, pages 35–40, 2004.

[108] James F Ziegler and Helmut Puchner. *SER–history, Trends and Challenges: A Guide for Designing with Memory ICs*. Cypress, 2004.