Algorithm-Based Fault Tolerance for Two-Sided Dense Matrix Factorizations

A Dissertation Presented for the

Doctor of Philosophy

Degree

The University of Tennessee, Knoxville

Yulu Jia

December 2015

© by Yulu Jia, 2015
All Rights Reserved.

To my wife and my parents for their unconditional love and support.

Acknowledgements

I would like to thank my advisor, Dr. Jack Dongarra, for his guidance, motivation and support during my graduate study at the Innovative Computing Laboratory. Dr. Dongarra has given me tremendous help while I was searching for my dissertation topic and during my research. His kindness, patience and encouragement made my study a pleasant experience. I am also grateful to Dr. Dongarra for providing generous financial support for my research and for the abundant opportunities to attend various conferences, research forums and workshops.

Thanks to Dr. James Plank, Dr. Michael Berry, and Dr. Ohannes Karakashian for agreeing to serve on my dissertation committee. I greatly appreciate their time and invaluable guidance on this dissertation.

I would also like to thank Dr. Piotr Luszczek, Dr. George Bosilca, Dr. Thomas Herault and Dr. Aurelien Bouteiller for their valuable discussions and suggestions. Thanks to Dr. Fengguang Song, who has been very helpful in providing opinions on my research and suggestions on various logistical issues. Special thanks to Dr. Peng Du who contributed time and effort from 2500 miles away while I was writing my first paper. His selfless support was instrumental in motivating my dissertation research. Thank you also to Mr. Sam Crawford for reviewing and editing my papers.

Lastly, I am grateful to my family and friends. Thanks to my loving wife, Rui Ma, for always being thoughtful and supportive. Thanks to my parents, for all the hardship they have gone through to raise me and my brother, and for the love they gave us. Thanks to my brother of being supportive to me and for taking care of our

parents while I am far away from home. Thanks to Hao Zhang, Essam Elkhouly, Yu Zhang and my fellow students at ICL, whose friendship has enlightened and entertained me over the years.

"In mathematics you don't understand things. You just get used to them."

– *Johann von Neumann*

Abstract

The mean time between failure (MTBF) of large supercomputers is decreasing, and future exascale computers are expected to have a MTBF of around 30 minutes. Therefore, it is urgent to prepare important algorithms for future machines with such a short MTBF. Eigenvalue problems (EVP) and singular value problems (SVP) are common in engineering and scientific research. Solving EVP and SVP numerically involves two-sided matrix factorizations: the Hessenberg reduction, the tridiagonal reduction, and the bidiagonal reduction. These three factorizations are computation intensive, and have long running times. They are prone to suffer from computer failures.

We designed algorithm-based fault tolerant (ABFT) algorithms for the parallel Hessenberg reduction and the parallel tridiagonal reduction. The ABFT algorithms target fail-stop errors. These two fault tolerant algorithms use a combination of ABFT and diskless checkpointing. ABFT is used to protect frequently modified data . We carefully design the ABFT algorithm so the checksums are valid at the end of each iterative cycle. Diskless checkpointing is used for rarely modified data. These checkpoints are in the form of checksums, which are small in size, so the time and storage cost to store them in main memory is small. Also, there are intermediate results which need to be protected for a short time window. We store a copy of this data on the neighboring process in the process grid.

We also designed algorithm-based fault tolerant algorithms for the CPU-GPU

hybrid Hessenberg reduction algorithm and the CPU-GPU hybrid bidiagonal reduction algorithm. These two fault tolerant algorithms target silent errors. Our design employs both ABFT and diskless checkpointing to provide data redundancy. The low cost error detection uses two dot products and an equality test. The recovery protocol uses reverse computation to roll back the state of the matrix to a point where it is easy to locate and correct errors.

We provided theoretical analysis and experimental verification on the correctness and efficiency of our fault tolerant algorithm design. We also provided mathematical proof on the numerical stability of the factorization results after fault recovery. Experimental results corroborate with the mathematical proof that the impact is mild.

Table of Contents

1 Introduction **1**

1.1 Problem Statement . . . 3

1.2 Contribution . . . 3

1.2.1 Hard Errors . . . 3

1.2.2 Soft Errors . . . 5

1.3 Dissertation Outline . . . 6

2 Background **7**

2.1 Impact of Faults . . . 7

2.2 Existing technologies . . . 8

2.2.1 Hardware Duplication . . . 9

2.2.2 ECC . . . 10

2.2.3 Software Duplication . . . 11

2.2.4 Checkpoint Restart . . . 12

2.2.5 Algorithm Based Fault Tolerance . . . 14

3 Parallel Reduction to Hessenberg Form with Algorithm-Based Fault Tolerance **16**

3.1 Introduction . . . 17

3.2 Related Work . . . 20

3.3 ScaLAPACK Hessenberg Reduction . . . 22

3.3.1 2D Block Cyclic Data Distribution . . . 22

3.3.2 Failure Model Under 2D Block Cyclic Data Distribution . . . 23
3.3.3 Non-blocked Hessenberg Reduction . . . 24
3.3.4 Blocked Hessenberg Reduction . . . 24
3.4 Encoding The Input Matrix . . . 26
3.5 The Algorithm . . . 28
3.5.1 Maintaining Data Redundancy in the Factorization . . . 28
3.5.2 Checksum Duplication . . . 31
3.5.3 Recovery . . . 33
3.6 Performance Analysis . . . 34
3.7 Experiments . . . 38
3.7.1 Overhead Without Failure . . . 39
3.7.2 Overhead With Failure . . . 40
3.7.3 Numerical Stability After Recovery From a Failure . . . 42
3.8 Conclusions and Future Work . . . 46

4 Parallel Reduction to Tridiagonal Form with Algorithm-Based Fault Tolerance 47

4.1 Introduction . . . 47
4.2 Related Work . . . 50
4.3 Contribution . . . 52
4.4 Background . . . 53
4.4.1 2D block cyclic data distribution . . . 53
4.4.2 Tridiagonal Reduction . . . 54
4.4.3 Blocked tridiagonal reduction . . . 55
4.4.4 Failure model . . . 56
4.5 The Fault Tolerant Algorithm . . . 57
4.5.1 Initial checksum setup . . . 57
4.5.2 The fault tolerant algorithm . . . 60
4.5.3 The invariant checksum relationship . . . 61

4.5.4 Protecting the upper triangular matrix . . . 63
4.5.5 Recovery . . . 64
4.6 Complexity analysis . . . 67

5 Hessenberg Reduction with Transient Error Resilience on GPU-Based Hybrid Architectures 71
5.1 Introduction . . . 71
5.2 Related work . . . 74
5.3 Hessenberg reduction on GPU enabled hybrid architectures . . . 76
5.3.1 The Unblocked Hessenberg Reduction . . . 76
5.3.2 The Blocked Hessenberg Reduction . . . 77
5.3.3 Hessenberg Reduction in MAGMA . . . 78
5.4 Soft error resilient Hessenberg reduction algorithm . . . 80
5.4.1 Failure Model . . . 80
5.4.2 Encoding the Input Matrix . . . 82
5.4.3 The Fault Tolerant Algorithm . . . 83
5.4.4 The Checksum Relationship . . . 86
5.4.5 Protecting Q . . . 87
5.4.6 Recovery . . . 88
5.5 Performance Evaluation . . . 89
5.6 Experiments . . . 92
5.6.1 Performance Study . . . 92
5.6.2 Numerical Stability . . . 94
5.6.3 Orthogonality of Q . . . 96
5.7 Conclusion . . . 96

6 CPU-GPU Hybrid Bidiagonal Reduction With Soft Error Resilience 98
6.1 Introduction . . . 98
6.2 Related Work . . . 100
6.3 Block Bidiagonal Reduction in MAGMA . . . 101

6.4 Error Propagation 102
6.5 Fault Tolerant **DGEBRD** 104
6.5.1 Data Redundancy 105
6.5.2 Error Detection 105
6.5.3 Error Location and Correction 106
6.5.4 Multiple Concurrent Errors 106
6.5.5 Range of Application 107
6.6 Experiment Results 107
6.7 Conclusion 108

7 Conclusions and Future Directions **110**
7.1 Conclusion 110
7.2 Future Directions 112

Bibliography **113**

Vita **126**

List of Tables

3.1 Residual Comparison 46

5.1 Detailed specification of the test platform. 92
5.2 Numerical Stability A1, A2, A3 94
5.3 Orthogonality of Q A1, A2, A3 96

List of Figures

3.1 A matrix mapped to a 2×3 process grid. 23
3.2 Global view of the matrix when a process fails. 24
3.3 One iteration of PDGEHRD 26
3.4 An encoded matrix mapped to a 2×3 process grid. 27
3.5 Partitions of the matrix 33
3.6 Overhead of FT-Hess without failures and with one failure. Platform: Titan, $NB = 80$, Algorithm 2 41
3.7 Overhead of FT-Hess without failures. Platform: Titan, $NB = 80$, Algorithm 3 42

4.1 A matrix mapped to a 2×3 process grid. 54
4.2 One iteration of PDSYTRD 56
4.3 Global view of the matrix when a process fails. 57
4.4 An encoded matrix mapped to a 2×3 process grid. 59
4.5 Partition for recovery 64
4.6 State of the matrix after the i-th iteration 66
4.7 Partition for recovery 66

5.1 One iteration of DGEHRD 80
5.2 Propagation pattern of errors at different locations 82
5.3 The encoded initial matrix 83
5.4 One iteration of FT_DGEHRD 85

5.5 Maintaining the checksums for Q 88
5.6 Overhead of FT-Hess 93

6.1 One iteration of `magma_dgebrd` 103
6.2 Propagation pattern of an error. 103
6.3 Performance of the FT-BRD 108

Chapter 1

Introduction

Mainstream supercomputers are well into the peta-scale era, with the number of components sharply increasing in recent years. Early supercomputers relied on a small number of specially designed powerful processors, and the first supercomputer, the CDC 6600, had only one CPU. As the designer of CDC 6600 put it, "If you were plowing a field, which would you rather use: Two strong oxen or 1024 chickens?" That configuration was typical and lasted into the 1990s, when massively parallel supercomputers became dominant. Massively parallel supercomputers usually contain tens of thousands of commodity microprocessors. These thousands of processors are connected by fast interconnects for data communication. Modern supercomputers are now also equipped with accelerators, which are specialized in data parallel work loads. Tianhe-2 (at National Supercomputer Center, Guangzhou, China), the number one machine on the June 2015 TOP500 list [69], has 3,120,000 cores at its disposal. The increase in the number of components is likely to continue [31], in which case even the most optimistic predictions about the failure rate of a particular component, in terms of tens of years, depicts a gloomy future. A future where the Mean Time To Interrupt (MTTI) of the entire machine falls under a few hours, drastically affecting individual applications running on the system [80], with a lasting impact, not only on the scientific throughput, but directly on the cost of the scientific simulations.

Various techniques to recover from a process failure exist, encompassing completely automatic solutions such as Checkpoint/Restart (C/R) and algorithm-level techniques such as Algorithm Based Fault Tolerance (ABFT). All of these methods are applicable to linear algebra computations, and each has its advantages and drawbacks. The major advantage of the C/R approach is the generality: it can be applied to a wide range of applications — not just linear algebra software. The major drawback of C/R is its relatively high overhead. The advantage of the ABFT techniques is the potential lower overheads, in exchange for algorithmic alterations. The major disadvantage of the ABFT technique is that it can only be applied to protect numerical software.

In this dissertation, we explore methods to provide fault resilience for two-sided dense matrix factorizations, namely the Hessenberg reduction, tridiagonal reduction, and bidiagonal reduction. The Hessenberg reduction [85] is an important step in calculating the eigenvalues and/or eigenvectors of a dense non-symmetric matrix or for solving the regular generalized eigenvalue problem. The tridiagonal reduction is used in solving the eigenvalue problem of a dense symmetric matrix. The bidiagonal reduction is used in solving the singular value problem of a general dense matrix. The common characteristics of these three factorizations are that their computation complexity is high, and they are rich in level 2 BLAS operations. These two characteristics result in long running times. The longer the running time is, the more likely a component will fail during the execution. Once a failure strikes, the application cannot continue because of data loss. The application has to start over from the beginning, and all associated cost (machine time and electricity) is wasted. We developed algorithms to enable fault resilience for these two-sided dense matrix factorization algorithms. Our goal is to achieve fault tolerance with less overhead than the widely used C/R approach. We also envision a version for soft error resilience of the two-side matrix factorizations for CPU-GPU hybrid version.

1.1 Problem Statement

In this dissertation we consider two different kinds of errors in two-sided dense matrix factorizations: hard errors and soft errors. We define hard errors as process failures. When a hard error occurs a process is lost, the associate data is also gone. The factorization cannot continue in the case of hard errors because the data is incomplete. Soft errors are bit-flips in the factorization data. When a soft error occurs, a bit in the factorization data is flipped, and the factorization keeps running only to produce wrong results. Soft errors are more dangerous because if the application does not actively detect them and correct them, they will not be noticed.

1.2 Contribution

The contribution of this dissertation consists of two major parts: fault tolerance algorithms for hard errors and fault tolerance algorithms for soft errors. We designed the fault tolerance algorithms which combine the advantages of ABFT and disckless checkpointing. ABFT is used to protect the frequently modified data, which greatly reduces the overhead when compared to checkpointing. For the part of the data that is rarely modified, we use diskless checkpointing to provide protection.

1.2.1 Hard Errors

- **Pseudo column checksums for the panel.** Data in the original input matrix is protected by checksums. The checksum blocks at the bottom of the global matrix are called pseudo column checksums. If the global matrix is distributed over a $P \times Q$ process grid in a 2D block cyclic fashion, these checksum blocks on the bottom are calculated assuming the block column is distributed over Q processors instead of P processors. This is why the column checksums are called "pseudo" checksums. Pseudo checksums are needed in order to maintain the validity of the checksum blocks on the right side of the global input matrix.

The checksum blocks on the right side are the data redundancy used to recover from a failure later on.

- **Diskless checkpointing for the panel factorization result.** In the two-sided matrix factorizations, the intermediate matrices resulting from the panel factorization cannot be recomputed as they are in the one-sided matrix factorizations. The reason is that the panel factorization has data dependency on the trailing matrix. Any changes in the trailing matrix after the panel factorization will make it impossible to recompute the panel factorization. We use diskless checkpointing to protect the panel factorization results. Diskless checkpoints are stored in the neighboring process to the right.

- **Use the built in data redundancy in the parallel tridiagonal reduction.** The parallel tridiagonal reduction algorithm takes advantage of the symmetry of the matrix to save floating point operations by only performing updates on the lower triangular part of the trailing matrix. The upper triangular part of the global matrix is not accessed throughout the factorization, but the checksums on the right side of the global matrix encodes the full matrix. Once a failure occurs, we fill the necessary matrix blocks in the upper triangular part using matrix blocks from the lower triangular part. We are able to do this because of the symmetry of the trailing matrix.

- **Formal and experimental performance analysis.** We provide a thorough examination of the theoretical computation complexity of the fault tolerant algorithms. We calculate the number of extra floating point operations (FLOPS) incurred by the fault tolerant algorithms, and we compare the extra FLOPS with the FLOPS of the original algorithms. We also verify the theoretical analysis through experiments. We also evaluate the numerical stability of our fault tolerant algorithms.

1.2.2 Soft Errors

- **Low cost error detection.** We append a row of column checksums at the bottom of the input matrix, and a column of row checksums at the right side of the input matrix. The checksum row and the checksum column are maintained through computation, so they are valid at the end of each iteration. The mathematical operations required to maintain the checksums are the same operations used in updating the trailing matrix, and they can be combined to take advantage of the optimized computation kernels. Error detection can be achieved by comparing the sum of the checksum row elements and the sum of the checksum column elements at the end of each iteration. The extra FLOPS needed by the error check is $O(n)$ in each iteration, whereas the computation complexity of the original algorithm is $O(n^3)$.

- **Reverse computation.** After an error is detected, our algorithm reverses the computation to the state at the end of the last iteration. The state at the end of the last iteration is a clean state which means the checksums on the right of the matrix and the checksums on the bottom of the matrix are valid checksums. At this state we can use the checksums to locate and correct the error. Rolling back the application data through computation is preferable because computation is much faster than using checkpoints.

- **Modified panel factorization kernel.** The panel factorization kernel takes a block column as input, factorizes the block column to the desired form (tridiagonal, bidiagonal, or Hessenberg form), and then returns the Householder vectors used in the factorization. The group of Householder vectors are used to update the trailing matrix. In order to maintain the validity of the checksums on the right, we need checksums of the Householder vectors. These checksums can be generated separately after the panel factorization has finished, but this adds more tasks on the critical path of the algorithm, and will have more effects in slowing down the factorization. We fuse the checksum generation with the

panel factorization so that minimum time penalty is incurred due to generating these checksums.

1.3 Dissertation Outline

The rest of this dissertation is organized as follows:

Chapter 2 introduces the background and related work on different types of fault tolerance technologies. This chapter also presents related fault tolerance work on dense linear algebra, and two-sided dense matrix factorizations in particular. Chapter 3 presents the hard error resilient parallel Hessenberg reduction algorithm. The fault tolerant algorithm combines the strengths of ABFT and diskless checkpointing to achieve fault tolerance against hard errors with low performance penalty. Chapter 4 describes the hard error resilient parallel tridiagonal reduction algorithm. This fault tolerant algorithm also uses diskless checkpointing and ABFT to protect the matrix data. In addition, this fault tolerant algorithm uses the symmetry of the matrix to retrieve data from the lower triangular part of the matrix in the recovery protocol. Chapter 5 describes the fault tolerant Hessenberg reduction algorithm for soft errors on CPU-GPU hybrid platforms. Chapter 6 describes the fault tolerant bidiagonal reduction algorithm for soft errors on CPU-GPU hybrid platforms. Finally Chapter 7 concludes the dissertation and outlines future directions.

Chapter 2

Background

In this chapter we review studies on the impact of faults in high performance computing and common technologies to mitigate the impact of faults on applications.

2.1 Impact of Faults

Faults and component failures have been unavoidable since the birth of computers. The impact of computer faults is so significant that companies exist that focus solely on manufacturing fault-tolerant computers. Avizienis and Laprie [3] proposed taxonomy in an effort to facilitate easy communication on dependable computing. They define failure/error as the deviation of the actual service state from the correct service state. The cause of a failure/error is called a fault.

Faults can be rooted in hardware malfunctions or software issues. Hardware may have hidden defects that are not detected in the manufacturing process, and these defects or bugs manifest as faults after installation. Hardware faults can also be caused by aging of components. Another source of hardware faults is cosmic rays. In fact, cosmic rays were shown to be the most prevalent sources of transient errors. Transient errors may happen at different levels in the hardware hierarchy, such as communication links or digital logic, but transient errors occur most commonly in the semiconductor storage. Software faults, on the other hand, can be caused by

bugs. As much as programmers want to write their software correctly, there are always bugs left behind. Parallel software is especially difficult to debug. If the bug is in system software and the bug causes the system to provide incorrect service to application codes, these bugs are considered as faults.

Faults can be categorized as hard faults and soft faults based on their effects. If the fault causes the program to abort or hang, this fault is called a fail-stop fault, or hard fault. If a fault is transient and goes unrecognized by the runtime, this fault is called a fail-continue fault, or transient fault/soft fault. Hard faults can cause huge waste of computing resources on peta-scale machines or future exa-scale machines. If an application has been running on a large portion of a peta-scale machine for a prolonged time period, then a hard fault could trash all the computation leading up to the fault, resulting in the aforementioned waste. Hard faults can also prevent large applications from ever completing their tasks on future exa-scale machines. The mean time between failure (MTBF) of an exa-scale machine is expected to be around 30 minutes, which means jobs which take more than 30 minutes to complete may never actually finish. Compared to hard errors, soft errors are more dangerous and more catastrophic, because soft errors can cause incorrect numerical results and there is no way to confirm the correctness of the results if the results fall within the allowed range. Soft errors could also corrupt memory address pointers and control flow of applications, which in turn cause the application to crash or hang.

2.2 Existing technologies

Significant research efforts have been leveraged towards overcoming the impact of faults in high end computing. Several practical fault-tolerance technologies have been developed and provide satisfactory results on current peta-scale supercomputers. Some of these fault tolerance technologies tackle the problem using a hardware approach, and some tackle the problem using a software approach.

2.2.1 Hardware Duplication

A natural way to tolerate hardware failures is to use hardware redundancy. Hardware redundancy can be divided into two categories: passive hardware redundancy and active hardware redundancy (also known as dynamic hardware redundancy). Passive hardware redundancy is also called static hardware redundancy. Examples of passive hardware redundancy include NMR (N-Modular Redundancy) and TMR (Triple Modular Redundancy). TMR is a special case of NMR. In TMR, the same program is executed by three independent modules. Once every module obtains the result, they carry out a vote, the majority wins, and the final result is obtained. TMR was first implemented in the SAPO computer in Czechoslovakia [78, p.97]. TMR circuits were also used in the Launch Vehicle Digital Computer in the Saturn IB and Saturn V boosters. In Active hardware redundancy, only one unit of all the redundant units is running. If the running unit fails, a backup will take over and the faulty unit stops running. STAR (Self Testing And Repair), designed by Avizienis, is one example of a computer based on active hardware redundancy [78, p.97]. STAR implements hardware redundancy in a different way. Every component in STAR has several backup units, and at any given time only one unit is powered and working. STAR executes each piece of program twice to detect component failures; if a failure is detected, then the failed component is replaced by a backup component. Once swapped out, the failed component is powered off to save energy. This is desirable because STAR is designed for spacecraft where electricity is a scarce resource. Hardware replication is an effective method to increase the MTBF (mean time between failure) of the entire system, but the financial cost of the system also increases proportionally with the number of backup units. This makes the hardware redundancy approach impractical even for current peta-scale machines. Current top end supercomputers typically have tens of thousands of compute nodes, and hundreds of thousands of processors, which requires a huge financial investment, and using TMR for fault tolerance would triple the already high cost of the machines. Therefore, TMR

is typically used in small or special purpose computers such as the computer systems on spacecraft as mentioned before.

2.2.2 ECC

ECC (Error Correcting Code) is a method to protect data read from the memory or transmission bus. After data are written into memory, several factors can flip bits in the data. This kind of bit flip often occurs in main memory which is made of DRAM (Dynamic random-access memory). DRAM stores every bit in a capacitor, which itself has two states: charged or discharged. These two states can be used to represent the only two values in the binary computer world. Moore's law also applies to DRAM. As technology advances, the density of capacitors on the DRAM die becomes higher and higher, while at the same time the feature size of the capacitors becomes smaller and smaller. The charge capacity of individual capacitors therefore decreases, which makes it easier for external charges to change the state of the capacitors. As a result, measures must be taken to protect data stored in DRAM from being corrupted. This is where ECC comes into the play. ECC works by using some extra bits to provide data redundancy for individual words. These extra bits are calculated using a coding algorithm. The most common error correcting code is SECDED (single-error correction and double-error detection) Hamming code. As the name indicates, this code is able to correct a single bit corruption and detect a double-bit corruption. Another common SECDED code is the Hsiao code [53]. When data are written to the DRAM, ECC calculates the redundant bits and stores them also in the DRAM. When data stored in the DRAM are requested, the redundant bits are calculated again, and the newly generated bits are compared to the redundant bits stored in the DRAM. If there is a mismatch between these two copies of redundant bits, there are corrupted bits in the codeword, which is the requested data. ECC can continue to correct the error if there is only one bit corrupted. More advanced forms of ECC exists. Chipkill [29] is IBM's trademarked ECC technology. Chipkill uses a RAID-like

approach to protect data stored in main memory. In Chipkill, the redundant bits are not calculated based on a codeword on a single chip, but instead Chipkill references bits from different DRAM chips. If a bit in one DRAM chip gets corrupted, only one bit in the ECC codeword is corrupted. ECC is able to recover from this error. In a much worse scenario, if an entire DRAM chip is corrupted, we get a group of corrupted ECC codewords, but still only one bit in every ECC codeword is corrupted. ECC is still be able to recover from this DRAM chip failure. Sun Microsystems has a similar technology called Extended ECC, and HP also has a similar technology named Advanced ECC. In rare cases, ECC is used to protect the data bus [2].

2.2.3 Software Duplication

Software duplication is another natural way to provide resilience to failures. In software duplication, the program being executed is duplicated. In practice there are various ways to implement software duplication. The most naive approach is to run two instances of the same program without any modification to the code. When one instance fails, the other one will hopefully run to completion. This approach requires the least effort from the programmer, and is applicable to any existing code. However, this approach clearly only tolerates one fail-stop error. If the running time of the application is longer than twice of the MTBF of the machine, the application still will not be able to run to completion. More advanced forms of software duplication include process level duplication and thread level duplication. Both of these two approaches require modifications to existing legacy codes. Process level duplication runs two instances of each process in the application. If one process fails due to an external influence, the application can keep running because the remaining instance of the same process has all the necessary data. Moreover, the failed instance can be recovered by cloning the remaining live process. Now that redundancy is resumed, the application can tolerate the next fail-stop failure. Besides fail-stop failures, process level duplication can be used to detect soft errors. Since there are two instances of

the same process running, two copies of data exist in these two instances. These two copies of data can be compared periodically, and mismatches between the two copies of data indicate that soft errors have occurred. For this soft error detection functionality to work, the application has to be bitwise reproducible. RedMPI [44] is a library that performs duplicate executions of MPI applications transparently to the programmer. RedMPI is a layer between the MPI application and the MPI library. MPI calls made in the application are intercepted and handled by RedMPI. RedMPI duplicates the call and then calls functions in the MPI library to carry out the actual MPI operations. RedMPI can detect soft errors by comparing the two copies of messages. By doing this all soft errors in the transmitted messages can be caught, and presumably soft errors in the application data can be caught because soft errors in the data owned by the MPI process will eventually manifest themselves in the MPI messages.

2.2.4 Checkpoint Restart

While other methods for fail/stop errors exist, checkpoint restart is still the most mature and reliable choice for production computer systems. The general idea of Checkpoint/Restart is to store the application state to stable memory at a time interval. The time interval is a parameter supplied by the user. In case of a fail stop error, the data stored on reliable memory persists. The application can restart itself, read its last saved state from reliable memory, and continue from there. Checkpoint/Restart can be implemented at different levels in the software stack. System level checkpointing [42] is performed at the operating system level, where the operating system periodically saves the application states to reliable memory. The operating system is oblivious to the progress of the application, so the points at which the checkpoints are saved may not be optimal in that the amount of data saved may exceed the amount necessary to recover from a failure. The advantage of system level checkpointing is that it is transparent to users, meaning that it requires no effort from

the programmer. If system level checkpointing is enabled, any code running on the system will be protected by Checkpoint/Restart automatically. However, most of the time system level Checkpoint/Restart is not optimal since the operating system does not have knowledge of the application state. The operating system cannot choose the best time point to save the application data to reliable memory. Examples of system level Checkpoint/Restart include BLCR (Berkeley Lab Checkpoint Restart) by Lawrence Berkeley National Laboratory, the IRIX operating system by SGI, and Cray's UNICOS/CLE operating system. Application level checkpoint, on the other hand, wins in the performance and efficiency aspects. In application level Checkpoint/Restart, the application calls utility functions to save its own state to reliable memory. It is the programmer's responsibility to decide when and what to save to reliable memory. The programmer has intimate knowledge of the application, so he or she can choose the optimal point in time to write checkpoints to reliable memory. Optimal time point means that the data saved to reliable memory is as small as possible – just enguoth data to recover from the fault. Often times, the optimal time point is the end of the iterative loop in the application. When less data has to be written to reliable memory, less burden is placed on the I/O stack. This is particularly important since I/O is the bottleneck of the computer system, and in particular the bottleneck of the Checkpoint/Restart method. As a result, application level Checkpoint/Restart is much more efficient than the system level approach. The difficult part of application level C/R is that it requires a non-trivial amount of effort on the part of the programmer. The programmer has to write checkpointing code for each individual application, and the coding effort is not portable across applications, nor across platforms. Application level Checkpoint/Restart provides better efficiency than system level Checkpoint/Restart, but it still suffers from the inherent performance bottleneck of C/R; that is the I/O traffic needed for writing checkpoints to reliable memory, which is usually hard drives. Even in application level Checkpoint/Restart, frequent disk access with large amounts of data is costly. Checkpointing a large scale application could take hours to finish. Research efforts

have been made to optimize the amount of data needed to save to disk. One notable trend is multi-level checkpointing. Multi-level checkpointing provides different levels of reliability by writing checkpoints to different storage at different frequencies. Local memory such as RAM and local disk has higher bandwidth and small latency, and checkpoints can be written to this fast storage at higher frequency. Remote storage and the parallel file systems are expensive to access, but they are more reliable than faster local storage. Checkpoints can be written to this slow and more reliable storage at lower frequencies. Multi-level checkpointing can provide better efficiency than single level checkpointing. Examples of multi-level checkpointing are FTI (Fault Tolerance Interface) [5] and SCR (Scalable Checkpoint/Restart))[71].

2.2.5 Algorithm Based Fault Tolerance

ABFT techniques are highly efficient when applied to appropriate numerical algorithms. The basic idea of algorithm-based fault tolerance is to add data redundancy to the original data in the numerical algorithm. The redundant data is usually in the form of checksums. Then the programmer needs to study the numerical algorithm carefully and design methods to update the checksums to make sure they are the correct checksums for the data in the original algorithm. When applicable, algorithm based fault tolerance techniques provide resilience with very low overheads. The very first work to employ the algorithm based approach in numerical algorithms was by Huang and Abraham [54] on matrix-matrix multiply and addition, scalar product, and LU factorization matrix transposition. Their work focuses on dealing with soft errors in those matrix operations, but the method can be extended to provide fault tolerance to many numerical algorithms whose core operations are basic matrix operations. In addition to protection against soft errors, the algorithm based fault tolerance approach can be used to provide provide protection against hard errors. The original work deals with algorithms running on systolic arrays, which are similar to today's single node machines in that the address space is shared

by the processing elements in the systolic arrays. Researchers have also designed algorithm based fault tolerant algorithms on distributed memory machines. The ABFT techniques that work on distributed memory machines focus on one-sided dense matrix factorizations [36, 26, 28, 95]. Algorithm based fault tolerance algorithms for two-sided dense matrix factorizations are rarely studied. The only literature we have found relating to algorithm based fault tolerance for two-sided matrix factorizations is by Chen and Abraham [25]. Their work explored algorithm based methods to detect soft errors in the Hessenberg reduction, the QR algorithm, and the singular value problem.

Chapter 3

Parallel Reduction to Hessenberg Form with Algorithm-Based Fault Tolerance

This chapter studies the resilience of a two-sided factorization and presents a generic algorithm-based approach capable of making two-sided factorizations resilient. We establish the theoretical proof of the correctness and the numerical stability of the approach in the context of a Hessenberg Reduction (HR) and present the scalability and performance results of a practical implementation. Our method is a hybrid algorithm combining an Algorithm Based Fault Tolerance (ABFT) technique with diskless checkpointing to fully protect the data. We protect the trailing and the initial part of the matrix with checksums, and protect finished panels in the panel scope with diskless checkpoints. Compared with the original HR (the ScaLAPACK **PDGEHRD** routine) our fault-tolerant algorithm introduces very little overhead, and maintains the same level of scalability. We prove that the overhead shows a decreasing trend as the size of the matrix or the size of the process grid increases.

3.1 Introduction

Mainstream supercomputers are well into the peta-scale era, with the number of components on a sharp increase over the years. Only one year ago, Jaguar, hosted at the Oak Ridge National Laboratory, included 224,162 cores. During its 537 days of operation, an average of 2.33 failures per day [68] occured, or on average less than 10 continuous hours of operation. Already today, the new configuration of Jaguar, called Titan, has a remarkable 299,008 Opteron cores, over 18,688 compute nodes, without taking into account the number of computing units on the accelerators, which would put the count in millions. This sharp increase in the number of components is likely to continue [31], in which case even the most optimistic predictions about the failure rate of a particular component, in terms of tens of years, depict a gloomy future. A future where the Mean Time To Interrupt (MTTI) of the entire machine falls under a few hours, drastically affecting individual applications running on the system [80], with a lasting impact, not only on the scientific throughput, but directly on the cost of the scientific simulations.

Numerical libraries are an important category of large scale applications which can easily utilize hundreds of thousands of cores and run for a prolonged period of time as building blocks of even longer running applications. Any node failure will render the time already spent running the application useless. Existing numerical libraries for high performance computers were designed and implemented when the size of the systems were modest and component failures were not yet a concern. Altering these numerical libraries and algorithms by adding reliability capabilities is critical to enabling them to become suitable for the future architectures with million-way parallelism. This process will directly benefit all applications built on top of these basic building blocks.

Libraries with eigenvalue solvers are the method of choice for spectral clustering of graphs [91] and eigenvector centrality and its widely known form: the PageRank [6, 17, 19, 62]. Hessenberg form is a common intermediate representation for eigenvalue

calculations.

The Hessenberg reduction [85] is an important step in calculating the eigenvalues and/or eigenvectors of a dense non-symmetric matrix or for solving the regular generalized eigenvalue problem. The orthogonal transformations are commonly used for this reduction for their guaranteed stability even though their accumulated cost becomes high in terms of both: computation and communication. One of the most common algorithms that stably computes eigenvalues of a dense matrix is the QR algorithm [85, 47]. There are two steps in the QR algorithm. In the first step, the matrix A is reduced to Hessenberg form H by a sequence of similarity transformations: $A = QHQ^{\top}$. A Hessenberg form, H, is a square matrix in which all the entries below the first subdiagonal are 0. The second step further reduces H to an upper triangular form T. The elements on the diagonal of T are the eigenvalues of matrix A. A Hessenberg matrix is also required for obtaining Hessenberg triangular form of the matrix pair (A, B) of the regular generalized eigenvalue problem of the form $(A - \lambda B)x = 0$ when using the QZ algorithm that originated from the implicitly shifted QR algorithm [45, 46]. More recent work involves efficient implementations of various QR iteration methods on modern multicore and distributed memory systems [58, 16, 49, 59, 50].

The Hessenberg reduction routine is provided in virtually all major numerical libraries, both for shared and distributed memory architectures. LAPACK [1] contains the routine **DGEHRD** for Hessenberg reduction. ScaLAPACK [8] is the open source linear algebra library providing LAPACK equivalent functionalities for distributed memory machines, its Hessenberg reduction routine is **PDGEHRD**. Commercial numerical libraries often provide optimized implementations of LAPACK and ScaLAPACK for specific architectures (such as LibSci for Cray XT architectures).

The high arithmetic complexity overall and low arithmetic intensity of its building blocks make the Hessenberg reduction a rather costly operation. In spite its high computational complexity of $O(\frac{10}{3}n^3)$, the Hessenberg reduction only achieves a fraction of the theoretical machine peak performance (unlike one-sided factorizations

such as LU and QR). While its long running time makes the Hessenberg reduction routine more exposed to fail-stop failures, with few exceptions, no algorithmic solutions to tolerate fail-stop failures have been proposed. Common fault tolerant techniques such as checkpointing and algorithm based fault tolerance (ABFT) have limitations when applied to the Hessenberg reduction. Checkpointing stores application data to stable memory at certain time intervals. In Hessenberg reduction, the whole trailing matrix, which accounts for a significant portion of application data, is modified very frequently, annihilating even the potential benefits of incremental checkpointing. Moreover, the checkpointing technique introduces too much overhead due to frequent write-to-memory accesses (either hard disk or remote main memory). Similarly, the usual ABFT techniques cannot provide protection for the lower left part of the matrix during the reduction.

The focus of this chapter is to investigate the possibility and effectiveness of ABFT techniques in the context of the Hessenberg reduction, to make the algorithm resilient to process failures. The fault tolerant algorithm we propose is a hybrid approach. We add row checksums to the right hand side of the matrix, and column checksums at the bottom of the matrix, which is similar to classic ABFT. We prove that the checksum relationship between the row checksums on the right hand side and the data matrix is invariant thus it provides protection to the trailing matrix during the whole factorization process. Any process failure and data loss in the trailing matrix can be recovered using the row checksums. The finished part of the matrix is protected with another group of row checksums. This group of checksums is computed only once for a group of column blocks upon their completion, thus the cost is very low. The group of panels currently being factorized are protected with a checkpoint. Due to the data dependencies of the Hessenberg reduction algorithm, this checkpointing procedure cannot be avoided. However, there is only one block column that needs to be checkpointed at any given time, and the overhead caused by this checkpoint is modest still. Our algorithm can tolerant more than one process failures at a time assuming that there is at most one failure in one processor row.

The rest of the chapter is organized as follows: Section 3.2 presents previous research work in fault tolerance for matrix computations. Section 3.3 introduces the Hessenberg reduction algorithm and its implementation, and highlights the challenges in applying ABFT to the Hessenberg reduction. Sections 3.4 and 3.5, describe the encoding used to provide the redundancy on the input matrix and the algorithm to maintain it through the computation. Section 3.6 provides a formal analysis of the overhead and costs, while Section 3.7 experimentally validates them. Section 3.8 summarizes the results of this chapter and presents future work.

3.2 Related Work

Diverse techniques to recover from a process failure exist, encompassing completely automatic solutions such as Checkpoint/Restart (C/R) and algorithm-level techniques such as Algorithm Based Fault Tolerance (ABFT). All these methods are applicable to linear algebra computations and each has its advantages and drawbacks.

The major advantage of the C/R approach is the generality: it can be applied to a wide range of applications not only linear algebra software. In the C/R technique, consistent snapshots of program data in main memory are saved to stable storage (usually a disk drive) at certain time intervals. Once a failure happens, the entire application rolls back to the latest snapshot and computation resumes from that point on (we ignore the complexity related to the consistent view of the entire application in terms of message or file accesses). In a distributed environment the major cost of this method comes from obtaining the consistent snapshots and disk access to write the snapshots, which highlights the major drawback of such approaches, the relatively high overhead. Langou and Dongarra [61] investigated several checkpoint/recovery techniques and a checkpoint-free lossy fault tolerant technique for parallel iterative methods. Robert and Vivien [13, 15] presented a unified model for several common checkpoint/restart protocols, extended in [21] to cover process replication. Diskless checkpointing [75, 48, 63] stores checkpoints in main memory to avoid disk accesses.

The advantage of the ABFT techniques is the potential lower overheads, in exchange for algorithmic alterations. The algorithm based approach considers the mathematical operations carried out in the algorithm, and it takes advantage of the mathematical relationship between different parts of the data to recover from erroneous data. Algorithm-based techniques do not require disk accesses. The extra cost entailed is a requirement of a small amount of local memory storage and some floating point operations. Since CPU speed is orders of magnitude faster than disk accesses on modern computers, an algorithm based approach has a much smaller overhead compared against the C/R approach.

Huang and Abraham [54] proposed a system-level method to tolerate errors in matrix computations in the context of systolic arrays. The matrix is encoded and operations are carried out on the encoded data. A single failure can be corrected during the computation. This technique has been successfully applied to matrix addition and multiplication, scalar product and the LU decomposition. Later, Luk and Park [64] extended Huang's method to make it more efficient to correct transient errors in Gaussian elimination and QR decomposition on systolic arrays. They proposed methods to compute checksums of the original matrix. Their method does not need a rollback in order to correct the error. Kim and Plank [60] presented a technique based on checksum and reverse computation to tolerate process failures in matrix operations. Chen and Dongarra [26] designed and implemented an algorithm based fault tolerance algorithm to tolerate process failures in the ScaLAPACK PDGEMM routine. Bosilca and Langou [14] also designed and implemented an algorithm based fault tolerance algorithm for the ScaLAPACK PDGEMM routine and developed performance models to predict its overhead. Hakkarinen and Chen [51] implemented an algorithm based fault tolerance algorithm for Cholesky factorization, an algorithm tolerating a single process failure at a time. Du and Dongarra et al. [36] designed algorithm based fault tolerance algorithms for LU and QR factorizations and implemented them in the ScaLAPACK framework. Their methods have a low overhead and scale well with the increase of matrix size and process grid size. Davies

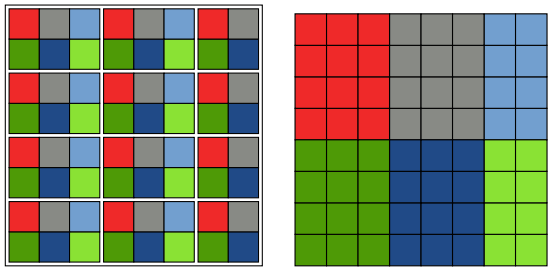
et al. [28] also applied the ABFT technique to HPL [74, 32] which is a highly optimized right-looking LU factorization. Yao and Wang [95] proposed a non-stop algorithm based fault tolerant scheme to recover the solution vector from fail-stop process failures in HPL 2.0. Bland et al. [10, 9] proposed a checkpoint-on-Failure protocol for fault recovery in dense linear algebra.

3.3 ScaLAPACK Hessenberg Reduction

ScaLAPACK uses a 2D block cyclic data distribution to achieve good load balancing. The Hessenberg reduction routine **PDGEHRD** in ScaLAPACK also distributes data in this way. The ScaLAPACK implementation of the Hessenberg reduction is a blocked algorithm. It first reduces a panel of columns using Householder reflections and accumulates the Householder reflectors along the way. Later it applies the group of reflectors all at once to the trailing matrix.

3.3.1 2D Block Cyclic Data Distribution

There are several possible ways to distribute a matrix across distributed memory machines. Among them, the 2D block cyclic distribution was chosen for ScaLAPACK based on its good scalability properties and the ability to use Level 3 BLAS routines. Figure 3.1 illustrates the 2D block cyclic data distribution with an example. A matrix is partitioned into small $nb \times nb$ square blocks. nb is called the blocking factor. These blocks are mapped to a 2×3 process grid. If a data block is mapped to a process it means the data block is physically stored in the local memory associated with that process. All the data blocks assigned to the same process are stored contiguously. Figure 3.1(a) shows the global matrix from a logical point of view. Each of the six colors represents a process. Data blocks are assigned to the processes in a round-robin fashion in both horizontal and vertical directions. Figure 3.1(b) is the processes' view of the distribution. Same as in Figure 3.1(a), each color represents a process. Each



(a) Matrix View (b) Process View

Figure 3.1: A matrix mapped to a 2×3 process grid.

process's own part of the matrix is stored contiguously in its local memory in column major. Each process is assigned roughly the same amount of data, which means they are responsible for roughly the same amount of total floating point operations. Block algorithms in ScaLAPACK proceed from left to right. As the algorithm continues, each process has roughly the same amount of work load left. This avoids prolonged idle time and keeps all the processes busy most of the time.

In this 2D block cyclic distribution, each process's data correspond to blocks scattered across the entire global matrix. When a process fails during the Hessenberg reduction, we get corrupted data blocks in every part of the global matrix.

3.3.2 Failure Model Under 2D Block Cyclic Data Distribution

In this work, we consider process failures. When a process fails in the process grid the data resident on that process will be all gone. Figure 3.2 shows the status of the matrix when a process failure happens. The colored squares are the data blocks owned by the live processes. The blank squares with question marks are the data blocks owned by the failed process. After we have recovered the process grid, the replacement process contains invalid data. These invalid data blocks need to be recovered to their state before the failure happened. If we continue the Hessenberg reduction without recovering the lost data the final result will be completely wrong.

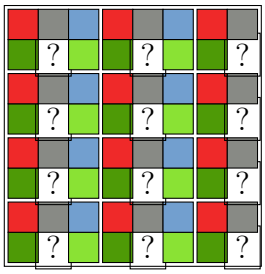


Figure 3.2: Global view of the matrix when a process fails.

3.3.3 Non-blocked Hessenberg Reduction

The Hessenberg reduction takes a general nonsymmetric square matrix $A \in \mathbb{R}^{n \times n}$ and decomposes it: $A = UHU^\top$. U is an orthogonal matrix, H is a Hessenberg matrix. The non-blocked Hessenberg reduction is an iterative process, $n-1$ Householder transformations are applied to the matrix A from left and right $H_{n-1}H_{n-2}\ldots H_2H_1AH_1^\top H_2^\top \ldots H_{n-2}^\top H_{n-1}^\top = H$. The orthogonal matrix U is $U = H_1H_2\ldots H_{n-2}H_{n-1}$. A Householder transformation H_i can be generated efficiently [85, page 83]. The non-blocked version uses Level 2 BLAS operations which have a low flop/transfer ratio and are slow. ScaLAPACK uses a blocked Hessenberg reduction algorithm which has a larger number of efficient Level 3 BLAS operations.

3.3.4 Blocked Hessenberg Reduction

In the blocked Hessenberg reduction [35] nb (the blocking factor in the 2D block cyclic distribution) Householder reflectors are accumulated and applied to the trailing matrix together using Level 3 BLAS. Using the WY representation [7, 79] the reduction can be written as:

$$H_{nb}^\top \cdots H_1^\top A H_1 \cdots H_{nb} = A - VW - YV^\top \tag{3.1}$$

where V is the matrix formed by the nb Householder vectors used to reduce the first nb columns, T is an $nb \times nb$ upper triangular matrix, $W = T^\top V^\top A$, $Y = AV^\top$

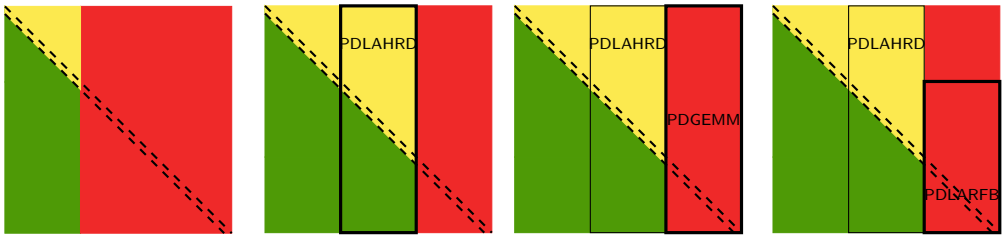
Algorithm 1 is the pseudo code for **PDGEHRD**. The function call **PDLAHRD**

Algorithm 1 PDGEHRD

1: **for** every panel **do**
2: PDLAHRD on the panel, return V, T, Y
3: PDGEMM: $trail(A) = trail(A) - YV^\top$
4: PDLARFB: $trail(A) = trail(A) - VT^\top V^\top \cdot trail(A)$
5: **end for**

reduces a panel with a sequence of Householder transformations. It takes the trailing submatrix, reduces the first panel of nb columns, and overwrites the bottom part of the panel with the Householder reflectors. Although this panel factorization routine only modifies the panel, it has a data dependency on the trailing matrix. In other words, once the trailing matrix is modified and we lose data inside the panel, the panel factorization cannot be repeated. This poses a challenge for our fault tolerant algorithm design as explained in later sections.

Figure 3.3 illustrates one iteration of the ScaLAPACK Hessenberg reduction algorithm. In Figure 3.3(a) the yellow part is part of the final result of the Hessenberg matrix. This part will not be touched once they have been computed. Columns in the green part are the Householder reflectors used to transform the matrix. The red part is the trailing matrix which will be reduced in future iterations. As other factorizations in ScaLAPACK, **PDGEHRD** is an iterative algorithm. In each iteration, **PDLAHRD** reduces the first block column which is called the *panel*. This call produces the final result of the desired Hessenberg matrix (the yellow upper trapezoid in Figure 3.3(b)) and nb Householder reflectors (the green lower trapezoid in Figure 3.3(b)). This call also generates intermediate matrices V and Y which are used by the **PDGEMM** and **PDLARFB** immediately following the panel reduction to update the trailing submatrix. When this iteration finishes, we get a smaller trailing submatrix: the red part on the right in Figure 3.3(e). In the next iteration, the same process is repeated on the shrunk trailing submatrix which further reduces it to a smaller size. This algorithm is a right-looking algorithm, in that, the updates only access data to the right of the current panel. Matrix entries to the left of the current panel are never touched again after the panel computation proceeded to the right.



(a) Beginning of iteration (b) Factorize the panel (c) Right update (d) Left update

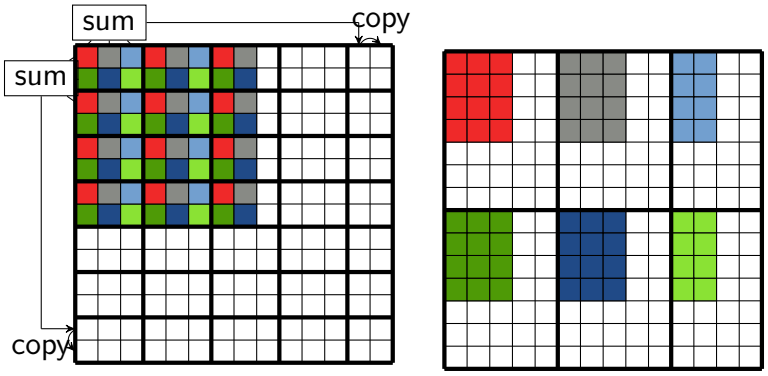
(e) End of iteration

Figure 3.3: One iteration of PDGEHRD

3.4 Encoding The Input Matrix

The essential part of ABFT technique is to expand the original matrix data with redundant data and maintain the relationship between the original matrix and the redundant data through computation. In our fault tolerant Hessenberg reduction algorithm, we chose to append the matrix with row checksums to the right of the original matrix. We show the checksum scheme with an example in Figure 3.4. A matrix of $N \times N$ blocks is mapped to a $P \times Q$ process grid in the 2D block cyclic fashion (here $N = 8$, $P = 2$, $Q = 3$). Each process will be assigned at most $\lceil N/P \rceil \times \lceil N/Q \rceil$ data blocks. We add $\lceil N/Q \rceil \times 2$ block columns to the right as checksum blocks. Data blocks in the same position of different processes of the same process row are added together element-wise to form a checksum block. This checksum block is duplicated and stored next to itself. The details are shown in Figure 3.4(a).

We also expand the original matrix with checksum blocks at the bottom. Only the storage is allocated, the actual checksums are not actually calculated in the beginning. The extra storage at the bottom will be used for pseudo checksums of the V matrix



(a) Matrix View (b) Process View

Figure 3.4: An encoded matrix mapped to a 2×3 process grid.

which contains the block Householder reflectors. The number of pseudo checksum block rows at the bottom is the same as the number of checksum block columns to the right of the matrix. And each pseudo checksum block is calculated in this way: pretend the matrix is distributed over a $Q \times Q$ process grid (despite that it is actually distributed over a $P \times Q$ grid), then we sum corresponding data blocks in different processes in the same process column element-wise and obtain a pseudo checksum block. The summing relationship is also shown in Figure 3.4(a). In this figure, the pseudo checksum block is the sum of the first three blocks, because had we distributed the matrix over a 3×3 process grid, the first three data blocks would be the first blocks in the three processes in the their respective local matrices.

These checksum blocks are treated as normal matrix data and distributed across the process grid. Figure 3.4(b) shows each process's local matrix containing the checksum blocks. Each black box represents a process. The white blocks are the checksum blocks. Note that the example in Figure 3.4 uses a small process grid, the checksum data are relatively large compared to the original input matrix. But in practice the process grid is rarely this small. The checksum data only accounts for a small portion of the input matrix when the size of the process grid increases.

3.5 The Algorithm

Algorithm 2 ABFT Hessenberg Reduction (non-delayed)

1: Compute the row checksum of matrix A, get A_e
2: **for** each i in N_e iterations **do**
3: **if** $i \equiv 0 \mod Q$ **then**
4: Take a snapshot of the panel scope.
5: **end if**
6: PDLAHRD on the panel, return V, T, Y
7: Calculate column pseudo checksum of V, get V_e
8: Send V to the next process column.
9: The process column owning the ith panel make a copy of its Y and T, send Y, T to the next process column.
10: PDGEMM: $trail(A_e) = trail(A_e) - Y(V_e)^\top$
11: PDLARFB:
 $trail(A_e) = trail(A_e) - VT^\top V^\top \cdot trail(A_e)$
12: Recover from failure if there is any.
13: **end for**

3.5.1 Maintaining Data Redundancy in the Factorization

Two versions of ABFT Hessenberg reduction algorithms are shown in Algorithm 2 and Algorithm 3. These two versions are mathematically equivalent, but their actual implementations have different performance characteristics due to the behavior of PBLAS routines. We use Algorithm 2 to explain how the method works. In iteration i, we refer to the Q block columns starting from $\lfloor i/Q \rfloor$ to $\lceil i/Q \rceil$ (inclusive) as the *panel scope*. N is the dimension of the original matrix, nb is the blocking factor.

Algorithm 2 first calculates row checksums for each block row in line 1. This is achieved with a reduction operation on each block row. Calculating this global checksum for the entire matrix requires many reduction operations and large communication volume. But this checksum is computed only once at the beginning of the algorithm. The cost is not high compared to the time cost of the actual Hessenberg reduction.

In line 4, the algorithm takes a snapshot of the panel scope before starting the factorization of the block columns in the panel scope. The final Hessenberg matrix contains zeros in its lower part, below the first subdiagonal. In order to save storage, the ScaLAPACK Hessenberg reduction algorithm stores the Householder reflectors in the lower part of the matrix. Because the zero entries are overwritten with the Householder reflectors, the row checksum relationship between the current panel scope and its checksum no longer holds. Once a process failure causes data loss in the trailing matrix part of the current panel, we can retrieve the pre-update data from the snapshot and reapply all updates from the beginning of the current panel scope. By so doing, we can restore the lost data to their state right before the failure.

Lines 8 and 9 record the state of the panel scope after each panel factorization. These two lines also record the state of Y and T which are the results of panel factorization. Y and T are stored in a separate workspace apart from A. The newly calculated Householder reflectors are stored in-place in the lower portion of A. These reflectors do not have any protection mechanism. Unlike a recent implementation of QR factorization [36], the panel factorization in the Hessenberg reduction has a data dependence on the trailing submatrix. The **PDLAHRD** routine needs the unmodified trailing submatrix to factorize the panel. This means that the panel factorization result has to be protected right away after it is obtained. We do not delay the recording of the state of the panel result (V, Y and T) till either the **PDGEMM** call or the **PDLARFB** call. This is because in the case of a process failure, data loss would occur in the panel result. This is possible for a failure that happens after the panel factorization and after the start of the trailing matrix update. The panel result cannot be recovered in that case by a rollback of the panel and re-factorizing it despite the fact that we can manage to recover the panel data right before its factorization. It is possible to reverse the effect of the trailing matrix update if we store the V, Y and T matrices which were used to update the trailing matrix. But they are not available since these three matrices are exactly what are supposed to be recovered.

Line 12 recovers data that were lost due to a process failure. The details of the recovery procedure are explained in section 3.5.3.

The following theorem shows how the correctness of the checksum is maintained throughout the algorithm.

Theorem 1. *The row checksums for block columns after the current panel scope are valid at the end of each iteration.*

Proof. We proceed by showing that the checksum remains correct after each step. Suppose A is of size $m \times n$, e is a column vector of 1's of length n. For simplicity, in the following proof we assume the block size nb is 1, and the process grid is $m \times n$, so each process takes one entry of the matrix but the proof holds true for any nb value and process grid size.

1. Before the **for** loop all the row checksums are just calculated, no data has been modified. Thus the checksums are valid.

2. In the first iteration, after the **PDLAHRD**, the checksum for the first panel scope is destroyed. But the checksums for the block columns after the first panel scope are still valid, because both the original matrix data and the checksums haven't been modified.

3. In the first iteration, after the **PDGEMM**, the checksums for the block columns after the first panel scope are still valid.

$$
\begin{aligned}
A_e - Y(V_e)^\top &= \begin{bmatrix} A & Ae \end{bmatrix} - Y \begin{bmatrix} V \\ e^\top V \end{bmatrix}^\top \\
&= \begin{bmatrix} A & Ae \end{bmatrix} - Y \begin{bmatrix} V^\top & (e^\top V)^\top \end{bmatrix} \\
&= \begin{bmatrix} A & Ae \end{bmatrix} - Y \begin{bmatrix} V^\top & V^\top e \end{bmatrix} \\
&= \begin{bmatrix} A & Ae \end{bmatrix} - \begin{bmatrix} YV^\top & YV^\top e \end{bmatrix} \\
&= \begin{bmatrix} A - YV^\top & Ae - YV^\top e \end{bmatrix} \\
&= \begin{bmatrix} A - YV^\top & (A - YV^\top)e \end{bmatrix}
\end{aligned}
$$

4. In the first iteration, after the **PDLARFB**, the checksums for the block columns after the first panel scope are still valid.

$$\begin{aligned} & A_e - VT^\top V^\top A_e \\ = \; & (I - VT^\top V^\top) A_e \\ = \; & (I - VT^\top V^\top) \begin{bmatrix} A & Ae \end{bmatrix} \\ = \; & \begin{bmatrix} (I - VT^\top V^\top)A & (I - VT^\top V^\top)Ae \end{bmatrix} \\ = \; & \begin{bmatrix} (A - VT^\top V^\top A) & (A - VT^\top V^\top A)e \end{bmatrix} \end{aligned}$$

By mathematical induction, the row checksums for block columns after the current panel scope are valid at the end of each iteration. □

3.5.2 Checksum Duplication

We protect the row checksums appended to the right of the matrix by maintaining two copies of exactly the same checksums. Because the checksums are distributed as normal matrix data over the process grid, any process failure will also cause loss of the checksums resident on the failed process. To solve this problem we maintain two copies of the checksums as in [36]. Both are kept valid through updating them independently. These two copies are stored next to each other so they are distributed to different process columns. Since only one process could fail, we always have one valid copy and can use this copy to recover the other copy. This approach does not need dedicated checksum processes, and does not have to assume that the checksum processes never fail. This approach also has good load balancing property. These traits are preferable because it does not require users of the ScaLAPACK library to change their application or the way they run their application. It is also easier to implement since the code has clear logic. The update of the checksum data does not need special treatment, the only thing needed is to change the dimensions of the trailing matrix during the update step of the original ScaLAPACK code.

Algorithm 3 ABFT Hessenberg Reduction (delayed)

1: Compute the row checksum of matrix A, get A_e
2: **for** each i in 1 to $\lceil N/nb \rceil$ iterations **do**
3: **if** $i \equiv 0 \mod Q$ **then**
4: Take a snapshot of the panel scope.
5: **end if**
6: PDLAHRD on the panel, return V, T, Y
7: **if** a process owns parts of V, T, Y **then**
8: Store V, T, Y in its neighbor in the next process column.
9: **end if**
10: **if** $i \equiv 0 \mod Q$ **then**
11: Calculate column checksums of V from the last Q block columns, get V_e
12: **end if**
13: PDGEMM: $trail(A_e) = trail(A_e) - Y(V_e)^\top$
14: PDLARFB:
 $trail(A_e) = trail(A_e) - VT^\top V^\top \cdot trail(A_e)$
15: **if** $i \equiv 0 \mod Q$ **then**
16: Update the row checksums at the right side of the original matrix using the V, Y, T matrices from the last Q panel factorizations.
17: **end if**
18: **if** a failure happens **then**
19: Compute column checksums of V from the already factorized panels in the current panel scope, get V_e.
20: Update the row checksums at the right side of the original matrix.
21: Recover from failure.
22: **end if**
23: **end for**

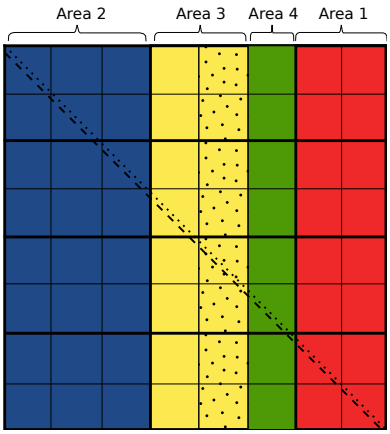


Figure 3.5: Partitions of the matrix. The dotted block column in area 3 has just been factorized. Area 1 (red) is the trailing matrix after the current panel scope. Area 2 (blue) is the finished part of the matrix. Area 3 (yellow) is the block columns in the current panel scope that have been factorized. Area 4 (green) is part of the current panel scope which belongs to the trailing matrix.

3.5.3 Recovery

When the Hessenberg factorization is in progress, the matrix can be divided into different areas based on the status of the data as shown in Figure 3.5. Different areas of the matrix data need different methods to recover.

The recovery process:

1. Recover the runtime system. Replace the lost process and restore the process grid.

2. Recover lost checksums using the duplication.

3. Recover lost data in area 1 and 2 using the row checksum on the right and the data on the live processes. First calculate the sum of data blocks on different processes in the same process row, then subtract this partial sum from the checksum to get the lost data blocks. Send the recovered data blocks to the replacement process.

4. Recover the lost data in area 3 using the checkpoint.

5. Recover the lost data in area 4. First retrieve the backup data from the snapshot, then apply all the left updates and right updates since the last snapshot.

6. Resume computation as usual. Ready to recover from the next failure.

3.6 Performance Analysis

In this section we use N to refer to the dimension of the global matrix.

There are several sources where the overhead of the fault tolerant Hessenberg reduction comes from. Firstly, it carries out more floating point operations than the ScaLAPACK version. Secondly, we need to perform bookkeeping for the panel results. Thirdly, we need to generate vertical pseudo checksums for V after the panels are factorized.

Global row checksums have to be calculated at the beginning of the factorization. On a $P \times Q$ process grid, every process row calculates the checksums inside the process row using reduction operations. Every process row performs the reductions in parallel with other process rows. Hence the total time cost is the same as the time cost in any one process row. There are $N/(nb \cdot Q)$ block columns in one process, for every one block column there is one reduction operation. Let T_Q be the time cost of one reduction operation among Q processes, the overhead incurred by the global checksum calculation at the beginning of the fault tolerance Hessenberg reduction algorithm is:

$$T_Q \frac{N}{nb \cdot Q}$$

This part of the overhead is a one time cost. The Hessenberg reduction is computation intensive, and the total floating point operation count is $O(\frac{10}{3}N^3)$. As the size of the matrix N increases, the operation count increases quickly, this initial one-time

checksum cost becomes insignificantly small very quickly compared to the total cost of the original ScaLAPACK Hessenberg reduction routine.

Extra floating point operations are needed to maintain the correct global checksum on the right side of the matrix. The panel factorization will stop at the end of the original matrix, so no panel factorization has to be done on the checksum block columns. The trailing matrix updates have to be performed on the checksums. In every iteration there is a right update which is a **PDGEMM**, and there is a left update which is a **PDLARFB**. The **PDLARFB** contains three steps: a **PDGEMM**, a **PDTRMM** and another **PDGEMM**. For the right updates, the number of checksum block columns decreases as the factorization proceeds. The reason is that the block columns to the left of the current panel and in the current panel scope are not protected by the right side checksum, and we do not need to update these not used checksums anymore. For the left updates on the checksums, not only does the number of columns of the checksums decrease, but also the number of rows decreases.

The amount of extra floating point operations caused by the right update (**PDGEMM**) is:

$$\begin{aligned} FLOP_{\text{pdgemm}} &= \sum_{i=1}^{N/nb-1} 2N\,(2nb)\,nb \cdot Q \\ &= 2\tfrac{N^3}{Q} - 2N^2 nb \end{aligned}$$

The amount of floating point operations introduced by the left update (**PDLARFB**)

is:

$$
\begin{aligned}
& FLOP_{\text{pdlarfb}} \\
= & \sum_{I=1}^{\frac{N}{nb}-1} \left[2nbQ\left(2nb \cdot I\right)\left(2nb \cdot I + 2\right) + \left(2nb \cdot I\right) nb^2\right] Q \\
= & \frac{8}{3}\frac{N^3}{Q} - 4N^2nb + 4N^2 + \frac{N^2nb}{Q} + \frac{4}{3}NQnb^2 \\
& - 4NQ \cdot nb - Nnb^2
\end{aligned}
$$

The total amount of extra floating point operations by maintaining the checksum is

$$
FLOP_{\text{Extra}} = \sum_{i=1}^{N/nb-1} \left[FLOP_{\text{pdgemm}} + FLOP_{\text{pdlarfb}}\right]
$$

The total count of floating point operations of the original ScaLAPACK Hessenberg reduction routine is:

$$
FLOP_{\text{Orig}} \approx \frac{10}{3}N^3
$$

So the overhead introduced by maintaining the checksums is given by:

$$
\begin{aligned}
Overhead &= \frac{FLOP_{\text{Extra}}}{FLOP_{\text{Orig}}} \\
&= \frac{FLOP_{\text{pdgemm}} + FLOP_{\text{pdlarfb}}}{FLOP_{\text{Orig}}} \\
&= \frac{3}{10}\left(\frac{2}{3}\frac{1}{Q} - \frac{6nb}{N} + \frac{4}{N} + \frac{nb}{NQ} + \frac{4}{3}\frac{Qnb^2}{N^2}\right. \\
&\quad \left. - \frac{4Qnb}{N^2} - \frac{nb^2}{N^2}\right)
\end{aligned}
$$

These extra floating point operations are all in matrix matrix multiplies which are efficiently implemented, so the overhead in terms of floating point operation count can also be interpreted as overhead in terms of running time. From the formula above, we observe that as the size of the matrix is big enough, N tends to infinity, and the

terms containing N in the denominator tend to 0:

$$\lim_{N\to\infty} Overhead = \frac{1}{5Q} \tag{3.2}$$

which means that the theoretical lower bound of the overhead introduced by maintaining the checksums is $1/(5Q)$. By "theoretical" we mean the ideal case where there is no time cost for memory accesses, and no time cost for communications between processes. When we keep the blocking factor nb unchanged and keep increasing the matrix size N, the least amount of overhead we have to pay is $1/(5Q)$ of the ScaLAPACK Hessenberg reduction routine. In practice it is not possible to access memory and transfer data between processes without time costs. The actual observed overhead introduced by these extra floating point operations should be higher than the above theoretical lower bound.

The second part of the overhead comes from bookkeeping the panel factorization results after panels are factorized. The bookkeeping is done by sending the matrices to the neighboring process in the next process column and storing them there. There are three matrices which have to be saved: the panel itself, Y and T. Let T_{sr} be the time cost to perform a Send-Receive operation between two processes, the total overhead incurred by bookkeeping the panel factorization results is:

$$T_{sr}\frac{N}{nb}$$

The value of T_{sr} varies depending on the MPI implementation and the network between processes.

Also there is the overhead of computing the vertical pseudo checksum of V. Every pseudo checksum block calculation involves a reduction operation, and this pseudo checksum has to be calculated in every iteration. Let T_P denote the time cost to perform a reduction among P processes, the total time cost of calculating this

checksum is given by:

$$T_P \frac{N}{nb \cdot Q}$$

Storage overhead. Extra storage is necessary for the checksums and for bookkeeping the panel factorization results. We keep two copies of the row checksums on the right of the original matrix. The amount of memory needed for this is:

$$2N\frac{N}{Q}$$

We also need the same amount of storage for the pseudo checksum of V. This makes the total amount of checksum memory:

$$4N\frac{N}{Q}$$

The amount of memory needed to store the snapshot of the panel scope is $N\left(N/Q + 2nb\right)$, the amount of memory needed by checkpointing Y and T is:

$$N\left(N/Q\right) + nb\left(N/Q\right)$$

Adding them all together, the total amount of storage overhead is:

$$4\frac{N^2}{Q} + \left(N + nb\right)\left(N/Q\right)$$

3.7 Experiments

In this section we evaluate the performance of our fault tolerant Hessenberg reduction algorithm through experiments. We used DOE's Titan as our test platform.

Titan is a hybrid supercomputing system located at Oak Ridge National Laboratory. It is the fastest parallel computer on the current TOP500 list (Nov.,

2012). Since we are only using the traditional CPU section of the machine, information about NVIDIA GPUs on Titan is not reported. Titan is composed of 18,688 nodes with 299,008 cores, for a CPU peak performance in double precision of 2.63 PFlop/s.

3.7.1 Overhead Without Failure

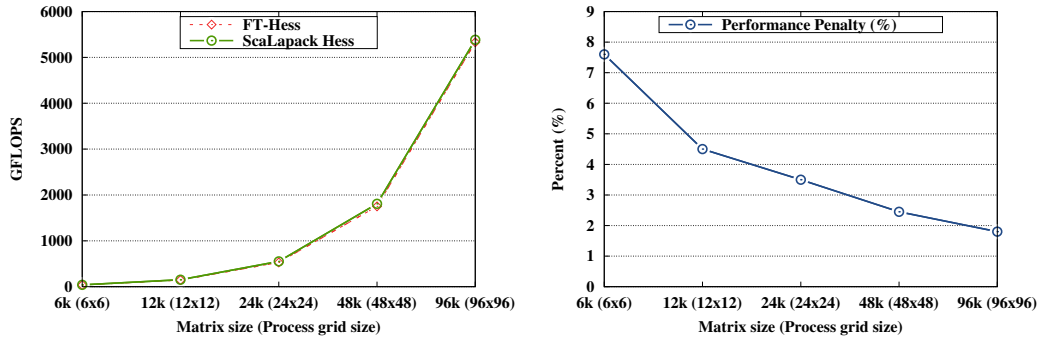
Figure 3.6(a) and Figure 3.6(b) shows the overhead of our fault tolerant Hessenberg reduction on Titan when no failure happens in the factorization. The overhead measured in the percentage of performance penalty drops as the problem size increases. The performance of Hessenberg reduction is not as high as the one-sided factorizations (LU, QR and Cholesky) on both distributed memory machines and shared memory machines. The reason is that Hessenberg reduction is rich in Level 2 BLAS (**GEMV**). Level 2 BLAS routines have a 1-to-1 flop to word ratio. These routines are memory bound and hence their performance is limited by the bandwidth of the memory. In terms of performance, our fault tolerant algorithm has a small overhead. The overhead with a matrix of size 6000 on a 6×6 process grid is 7.6%. The overhead keeps decreasing as the matrix size increases and the process grid increases. The overhead drops to 1.8% for a matrix of size 96000 (process grid 96×96). This overhead includes the overhead of calculating the initial checksum, the computation overhead incurred by updating the checksum, the overhead of calculating the vertical pseudo checksum of V after each panel factorization, and the overhead of the recovery process. Equation 3.2 states that the overhead caused by extra computation on the checksums asymptotically decreases to $1/(5Q)$. It accounts for a decreasing portion of the total overhead as the problem size and process grid become large. The overhead caused by saving the results of the panel factorization (**PDLAHRD**) becomes the major contributor of the total overhead. Over the course of the factorization, the total communication volume of this saving process is roughly two times the global matrix data volume. Depending on the network bandwidth between the processes,

this part of the overhead can account for different percentages of the total overhead. Generally, this part of the overhead tends to a small constant percentage when the problem size increases.

Figure 3.7(b) shows the overhead of Algorithm 3 on Titan. We see that the performance overhead keeps dropping in the beginning, but it starts to go up again at grid size 96×96. There are three main reasons which cause the overhead increase. Firstly, when we delay the updates of the global checksums at the end of each panel scope, these updates resulting from each panel factorization are applied sequentially. When the process grid size increases, the number of panels in the panel scope also becomes larger. The sequence of updates to the global checksums takes longer to finish. Secondly, when updating the checksums separately from the trailing matrix, the updates (**PDLARFB** and **PDGEMM**) are applied to a tall and skinny matrix. These two routines perform best when applied to more rectangular matrices. Also, splitting the calls to these routines disrupts their internal communication pipeline that hides latency and creates additional synchronization points upon exit and then entry into these routines. Thirdly, updating the checksums separately causes extra communication between processes owning V and processes owning the checksums. These overheads are critical in the context of an already communication-rich operation such as the Hessenberg reduction, and they inhibit scalability as the Figure 3.7(b) indicates.

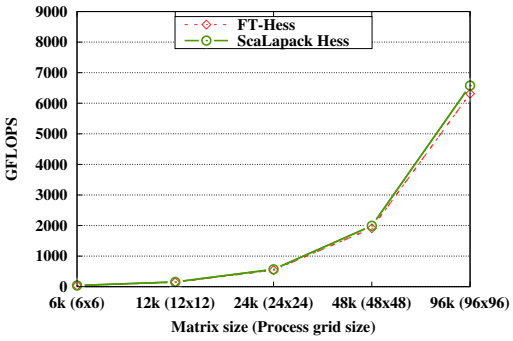
3.7.2 Overhead With Failure

Figure 3.6(c) and Figure 3.6(d) shows the performance and performance overhead of our fault tolerant Hessenberg reduction algorithm on Titan when one failure happens in the factorization. Compared with Figure 3.6(a) and Figure 3.6(b)and the performance overhead shown in the Figure 3.6(c) and Figure 3.6(d) include one more factor: the recovery overhead. The recovery process involves a global row-wise reduction operation on the entire global matrix. Before this global reduction the data

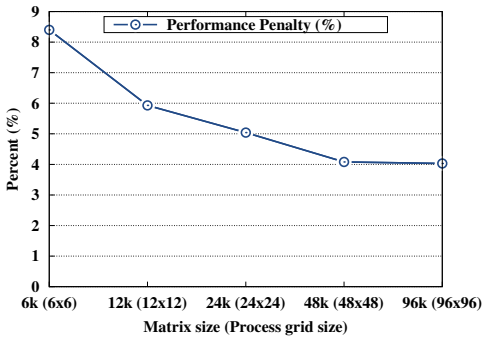


(a) Without failures

(b) Without failures



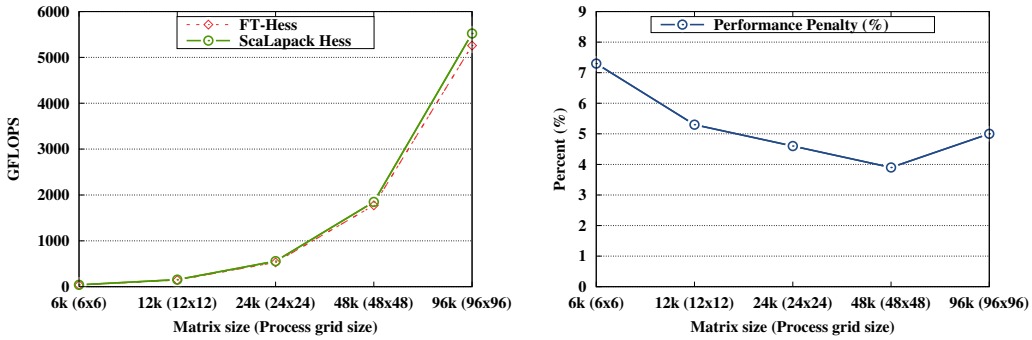
(c) With failures



(d) With failures

Figure 3.6: Overhead of FT-Hess without failures and with one failure. Platform: Titan, $NB = 80$, Algorithm 2

on the replacement process are set to zero. This global reduction operation calculates a new global checksum. The lost data belonging to Area 1 and Area 2 in Figure 3.5 are recovered using the new checksum and the old checksum that we have been maintaining along with the factorization. The cost of this global reduction depends on the bandwidth of the link between the processes. This cost accounts for a small portion of the total running time of the Hessenberg reduction. Figure 3.6(d) shows that, even with the recovery cost included, the total overhead of our fault tolerant Hessenberg reduction algorithm is still very low and it decreases as the problem increases. It is down to 4.03% for the matrix of size 96000 (process grid dimension: 96×96).



(a) Performance

(b) Penalty in percentage

Figure 3.7: Overhead of FT-Hess without failures. Platform: Titan, $NB = 80$, Algorithm 3

3.7.3 Numerical Stability After Recovery From a Failure

In this subsection, we show how our fault tolerant Hessenberg reduction algorithm maintains the same level of numerical stability as the original ScaLAPACK algorithm.

Floating point numbers are represented in IEEE 754 format in modern computers, floating point operations are not carried out in exact arithmetic. Standard error analysis for the reduction of a general matrix A to Hessenberg form H by means of similarity transformations shows the process to be backward stable [92, page 363]. In particular, the process reduces a nearby problem $\hat{A} = A + E$ into $\hat{H}$ with a set of similarity transformations U and at the end we get:

$$\hat{H} = U^{\top} \hat{A} U \tag{3.3}$$

The bound on the residual error E [92, page 351] is

$$\|E\|_F / \|A\|_F \leq \phi(N)\epsilon \tag{3.4}$$

where ϕ is a low degree polynomial [92, page 351, Table 1] and ϵ is the *unit roundoff* (machine precision). This is an expected result since the transformation only employs orthogonal transformations and therefore does not introduce rounding errors larger

than those already existing in the data. In fact, its backward error analysis has been used in a scheme that detects soft errors in linear algebra operations at runtime [11].

The ScaLAPACK **PDGEHRD** routine uses the following factorization residual to verify the factorization result

$$r_\infty = \frac{\|A - UHU^\top\|_\infty}{\|A\|_\infty N\epsilon}$$

where r_∞ is a slowly growing function of N. For practical purposes r_∞ may be checked against a constant threshold r_t. We consider the reduction correct if the residual r_∞ is smaller than the threshold $r_t = 3$.

To show backward stability of the recovery process, we use the technique of projecting the error (resulting from a fault) back into the original matrix A [64]. We then exploit the fact that the backward error analysis already involves a perturbation to A and the reduction is shown to provide a solution to a nearby problem $\hat{A}$ with a satisfactory bound on the perturbing error. Then, using a standard dot-product error analysis [22], we show that the numerical stability is not affected by the recovery from the fault. The dot-product analysis applies to our checksum procedure with only a slight modification.

There are three sources of errors in addition to the error existing in the original algorithm after the recovery:

- from the initial encoding of the input matrix,
- from updating the global checksum,
- from recovering the lost data in the case of a failure.

Errors from encoding the input matrix. The initial checksums are calculated through a simple summation operation. On a $P \times Q$ process grid, each checksum element involves at most $Q - 1$ addition operations . The rounding error (denoted by

E_1) introduced by encoding the input matrix is bounded by

$$E_1 \leq (Q-1)\,\epsilon \tag{3.5}$$

This upper bound is reached in the worst case scenario when rounding errors happen in every element and all have the same sign. In reality, rounding errors do not happen for every operation and/or do not all have the same sign. A pair of rounding errors with opposite signs will cancel each other out. The actual error is much smaller than the upper bound – the suggested approximation is the square root of quantities dependent on the problem size [93].

Errors from updating the global checksum. The global checksums on the right hand side of the input matrix are updated by two routines **PDGEMM** and **PDLARFB**, both of them perform matrix-matrix multiplications. These two routines are numerically stable which means the rounding error of the input data does not grow after the calculation.

Errors from recovering the lost data in the case of a failure. During recovery we calculate a new checksum of the data on the still live processes. In the worst case scenario the rounding error (denoted by E_2) could be

$$E_2 \leq (Q-1)\,\epsilon \tag{3.6}$$

In the worst case, E_2 has the opposite sign to E_1, which gives the worst case error in the recovered data compared against the lost data

$$E_3 = E_1 + E_2 \leq 2\,(Q-1)\,\epsilon \tag{3.7}$$

If the failure happens in the i-th iteration, denote the accumulated transformations

so far by $U_{(i)}$, we have

$$\begin{aligned}
\hat{r}_\infty &= \frac{\|A - \left(UHU^\top + U_{(i)}E_3U_{(i)}^\top\right)\|_\infty}{\|A\|_\infty N\epsilon} \\
&= \frac{\|\left(A - U_{(i)}E_3U_{(i)}^\top\right) - UHU^\top\|_\infty}{\|A\|_\infty N\epsilon} \\
&= \frac{\|\left(A - U_{(i)}E_3U_{(i)}^\top\right) - UHU^\top\|_\infty}{\|A - U_{(i)}E_3U_{(i)}^\top\|_\infty N\epsilon} \times \frac{\|A - U_{(i)}E_3U_{(i)}^\top\|_\infty}{\|A\|_\infty} \\
&= c \times \frac{\|\left(A - U_{(i)}E_3U_{(i)}^\top\right) - UHU^\top\|_\infty}{\|A - U_{(i)}E_3U_{(i)}^\top\|_\infty N\epsilon}
\end{aligned}$$

where

$$\begin{aligned}
c &= \frac{\|A - U_{(i)}E_3U_{(i)}^\top\|_\infty}{\|A\|_\infty} \\
&\leq \frac{\|A\|_\infty + \|U_{(i)}E_3U_{(i)}^\top\|_\infty}{\|A\|_\infty} \\
&= 1 + \frac{\|U_{(i)}E_3U_{(i)}^\top\|_\infty}{\|A\|_\infty} \\
&\leq 1 + N/P \times 2(Q-1)\epsilon
\end{aligned}$$

Again, this is the theoretical upper bound assuming the worst possible cases. In reality rounding errors are mostly likely random, so they will cancel each other out. The recovery process will not cause observable extra backward errors.

Table 3.1 shows a comparison of the residual r obtained in our fault tolerant algorithm when a failure happens and the residual obtained in the fault-free ScaLAPACK routine. We can see that our fault tolerant algorithm computes answers on the same order of magnitude as the original ScaLAPACK algorithms, with minor differences due to the randomness of the initial matrix and the lack of bitwise reproducibility of the algorithm. Overall, our fault tolerant Hessenberg reduction algorithm is as backward stable as the ScaLAPACK version.

Table 3.1: Residual Comparison

Grid Size	FT-Hess	ScaLAPACK Hess
6×6	5.208026×10^{-3}	5.014403×10^{-3}
12×12	3.099298×10^{-3}	2.348654×10^{-3}
24×24	2.166615×10^{-3}	1.174153×10^{-3}
48×48	1.361631×10^{-3}	6.350293×10^{-4}
96×96	1.038104×10^{-3}	3.379741×10^{-4}

3.8 Conclusions and Future Work

This chapter describes a hybrid fault tolerant Hessenberg reduction algorithm combining diskless checkpointing and algorithm based fault tolerance techniques under the fail/stop failure model, capable to recover from one process failure at a time. After the successful recovery, the computation is resumed and ready to progress and to tolerate the next process failure. We use algorithm based fault tolerance techniques to protect the trailing matrix, and checksums to protect the left part of the Hessenberg matrix, while the panel scope is protected through diskless checkpointing. We confirmed the low overhead and good scalability of our approach both from a theoretical standpoint and through experiments on various scales. The overhead decreases when the matrix size or the process grid size increases, making this approach a good candidate for large scale environments. Future work would include exploring methods to tolerate multiple simultaneous failures and designing fault tolerant algorithms for other two-sided factorizations in large scale parallel computing environments.

Chapter 4

Parallel Reduction to Tridiagonal Form with Algorithm-Based Fault Tolerance

4.1 Introduction

Today's massively parallel computer systems are more vulnerable to failures than ever before. Supercomputers in the early days only had a few processors. Over time the increasing demand for computational power and the development computer hardware impacted the design of supercomputers, as as result supercomputers containing tens of thousands of commodity processors become the norm. A large processor count provides high computational power, meanwhile it also brings lower reliability of the entire system. A study in 2007 on the failure rates of LANL systems show that the failure rate reached 1100 failures per year [80]. That is the system can run without interrupt for 12 hours. The 2011 version of the Oak Ridge National Lab's ten-year exascale road map projected that in year 2020, the total core count of the machine will be on the order of $O(billion)$, and the Mean Time to Interrupt will be 22 - 120 minutes [43]. When High performance computing enters the exascale era, the

reliability of high performance computing systems become a practical concern for large scale, long running applications.

Over the years, many techniques have been proposed to provide resilience for applications/restart. The most successful technique among them is checkpointing/restart. Checkpointing is a general purpose resilience technique which stores the application state into reliable storage from time to time. When a failure strikes and some process loses its data, the application reads the last checkpoint and restarts from that point. Usually checkpoints are taken periodically. The checkpoint interval is a optimization parameter. More frequent checkpointing incurs higher bottom line overhead, which is the overhead no matter failures occurs or not. Checkpointing/restart can be implemented at the system level or the user level. In system level checkpointing, the operating system provides the checkpointing functionality, the user is not required to put any effort to obtain resilience for the application. An example of system level checkpoint/restart implementation is the Berkeley Lab Checkpoint/Restart (BLCR) for LINUX [41]. User level checkpointing calls for more user effort. It is the user's responsibility to decide when to checkpoint and what content to checkpoint. In order to do this the user need to understand the code and modify the code to gain fault tolerance.

The checkpointing technique works well on a broad range of applications, but for certain class of codes, we can do better using algorithm based fault tolerance (ABFT). Algorithm based fault tolerance was brought forward by Huang and Abraham [55] and later gained interest from more researchers. Like the checkpoint/restart technique, the ABFT technique also keeps redundant data for error correction. There are two main differences between checkpoint/restart and ABFT. Firstly, unlike checkpoint/restart which stores redundant data (checkpoints) to stable storage (slow), ABFT stores redundant data in main memory (fast). The average access time of hard drives is 200 times slower than the average DRAM. Secondly, the redundant data in ABFT are mathematically computed from the application data, and the mathematical relationship between the redundant data and the application data is kept an invariant

through modest computation. The size of redundant data is small. Whereas the redundant data in checkpoint/restart are simply a copy of the application data. ABFT is naturally suitable for numerical software whose main operations are mathematical operations.

Matrix factorizations are critical building blocks of many scientific codes. They often run for a prolonged period of time, hence are easily exposed to interrupts. Researchers have been trying to incorporate ABFT into parallel matrix factorizations. ABFT enabled versions of the three most important one-sided matrix factorization algorithms (LU, QR, Cholesky) have been developed. Compared with one-sided factorizations, it is harder to protect two-sided factorizations (Hessenberg, tridiagonal, bidiagonal) with ABFT because two-sided factorization algorithms are more complex and more effort is needed to capture the algorithm properties in order to use ABFT to provide resilience for them.

In this chapter we present a fault tolerant parallel tridiagonal reduction algorithm with ABFT. The tridiagonal reduction of a dense matrix is the first step in solving the symmetric eigenvalue problem. The eigenvalue problem is encountered very often in structural mechanics and electrodynamics. The eigenvalues are related to the resonance frequencies of systems, the eigenvectors are related to the invariant probability measures of stochastic processes [24, 12]. The tridiagonal reduction algorithm for real matrices is implemented as **DSYTRD** in LAPACK [1], as **PDSYTRD** in ScaLAPACK [8]. The tridiagonal reduction stage is the most time consuming step in solving the symmetric eigenvalue problem. Providing fault tolerance to the tridiagonal reduction stage can greatly improve the chance of completing the symmetric eigenvalue problem on large scale computers.

The main contribution of this chapter is a ABFT enabled parallel tridiagonal reduction algorithm as implemented in ScaLAPACK. The tridiagonal reduction algorithm proceeds in a panel factorization–trailing matrix update loop. We protect the factorization using checksums. Checksums are appended to the right side of the original matrix and are updated the same way as the trailing matrix update in the

non-ABFT tridiagonal reduction algorithm. We prove that the appended checksums remain valid at the end of each loop iteration. When a process failure strikes, we subtract matrix elements from the corresponding checksum, this way data on the failed process are recovered. In ScaLAPACK the upper (or lower, depending on user preference) is not accessed in order to save computation and data movement, so the upper (or lower) half of the matrix contains invalid data after the factorization starts. However, the checksums are valid assuming both the upper and lower triangle of the matrix contain valid data. Before we recover the lost data, we fill up the necessary part in the upper triangle using data from the lower triangle. We can do this because the trailing matrix is always symmetric.

The rest of the chapter is organized as follows: we give a description of algorithm based fault tolerance and its integration in linear algebra software in section 4.2. In Section 4.3 we list our contributions. In Section 4.4 we explain the tridiagonal reduction algorithm as implemented in ScaLAPACK, the data distribution pattern in ScaLAPACK and the type of failure our fault tolerant algorithm is meant to deal with. Section 4.5 is devoted to the ABFT enabled fault tolerant tridiagonal reduction algorithm. In Section 4.6 we analyze the asymptotic computation overhead and storage overhead of the fault tolerant algorithm.

4.2 Related Work

ScaLAPACK [8] is a widely used software package which contains distributed dense linear algebra routines. ScaLAPACK is available on virtually all distributed memory high performance computers. The reference implementation of ScaLAPACK is available on netlib. Major computer vendors usually offer optimized versions of ScaLAPACK for their own machines. For example, Cray's scientific computing package *libsci* contains a version of ScaLAPACK optimized for Cray machines.

There has been very few work to protect the tridiagonal reduction algorithm. Early work on the fault tolerant symmetric eigenvalue problem focused on systolic

arrays [25]. A systolic array is a group of data processing units connected together through a mesh network. Each data processing unit performs some operation on one data element at a time. Data flow through the network of processing units much like the pulsing blood flow in the human body driven by the heart, hence the name systolic array. Systolic array is an interesting architecture, but it is not commonly seen nowadays.

Later, in the 1990s, Kim et. al. [60] designed a fault tolerant scheme for the Hessenberg reduction which is the first step of solving the nonsymmetric eigenvalue problem. Their scheme works in a distributed memory environment. The algorithm takes a checkpoint of the matrix at the end of each iteration. The checkpoint is in the form of the sum of data blocks on different nodes, this sum is called a checksum. Upon a failure, the algorithm rolls back to the end of the previous iteration using reverse computation, then uses the checksum to recover the failure. The advantage of their method is that they uses checksums as a in-memory checkpoint which not only reduces the size of the data to bookkeep but also saves frequent disk accesses. The drawback is that in their method the checksums need to be recalculated in every iteration. This brings extra synchronization cost and data transfer cost.

The regular checkpoint/restart technique can also be applied to the tridiagonal reduction algorithm, but no implementation has been done.

The Algorithm Based Fault Tolerance (ABFT) technique is a narrow spectrum fault tolerance method. When applicable ABFT can provide failure protection for the target algorithm at extremely low cost. Huang and Abraham [55] first used ABFT to protect matrix-matrix multiply. The basic idea is to encode the input matrices A, B and C with checksums, the checksums are appended to the input matrices to form extended input matrices. The matrix-matrix multiply is carried out as usual on the extended input matrices. Based on the proof in their chapter, the checksums in the resulting extended matrix are still valid checksums. Based on this valid checksum relationship, faults can be detected and corrected. Later the ABFT technique for LU, QR and Cholesky factorizations have been developed for soft errors on systolic

arrays. It is hard to capture the algorithm characteristics in matrix factorizations to design algorithm based fault tolerant algorithms for them.

In recent years the ABFT technique is revisited and ABFT enabled matrix factorizations are designed for large scale distributed supercomputers. Many of these new ABFT algorithms are based on the ScaLAPACK package. Unlike systolic arrays, large scale supercomputers have much more processors and each processor operates on a data set instead of a single element as in systolic arrays. This complexity calls for a new design of the ABFT scheme. All of the three one-sided matrix factorizations are protected against process failures: Du [36] designed ABFT based LU and QR factorizations, Chen [51] designed ABFT based Cholesky factorization.

4.3 Contribution

We proved the checksum relationship, designed a fault tolerant algorithm, implemented our design, and gave an analysis on the overhead of our algorithm:

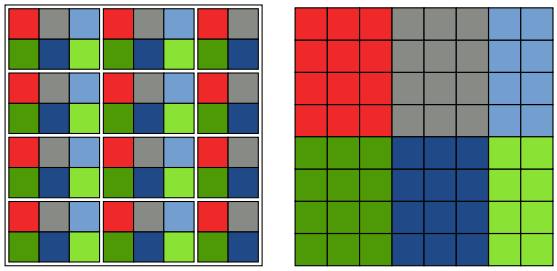
- **Invariant checksum relationship** We proved the invariant checksum relationship throughout the factorization.
- **ABFT based tridiagonal reductoin algorithm** We designed a fault tolerant tridiagonal reduction algorithm fully based on ABFT. The panel scope is also protected by checksums.
- **Overhead analysis** We provided a thorough analysis on the overhead of our fault tolerant tridiagonal reduction algorithm. The analysis shows the asymptotic performance overhead and asymptotic storage overhead approach 0.

4.4 Background

Before we explain our fault tolerant tridiagonal reduction algorithm, it is important to understand how the non-fault tolerant version of the algorithm works in ScaLAPACK.

4.4.1 2D block cyclic data distribution

ScaLAPACK operates on dense matrices on distributed memory machines. Data are stored on different processors in a two dimensional block cyclic fashion. Processors are arranged into a two dimensional $P \times Q$ grid (P rows and Q columns). Matrices are partitioned into blocks, these blocks are then distributed to the $P \times Q$ processor grid in a 2D block cyclic manor as shown in Figure 4.1. In this example the available processors are arranged into a 2×3 processor grid, each color represents a different processor. Figure 4.1(a) shows the global view of the matrix when it is distributed over a 2×3 process grid. Figure 4.1(b) shows each process's view of the data. Data blocks owned by one processor are assembled to a local matrix, this local matrix is stored contiguously in each process's memory. Each process stores a matrix descriptor, based on this descriptor a process can compute the global position of a local data block given its local position. Global position can be converted to local position based on this descriptor. 2D block cyclic data distribution scheme has several advantages [8]. Firstly, it has good load balancing properties. Matrix data are distributed evenly across all available processors. As the computation proceeds, data participating in the computation are still evenly distributed. Secondly, this distribution scheme enables the use of Level 3 BLAS on the local matrix. Level 3 BLAS operations are well optimized and has the highest flop:word ratio. Lastly, this distribution scheme has good scaling properties.



(a) Matrix View (b) Process View

Figure 4.1: A matrix mapped to a 2×3 process grid.

4.4.2 Tridiagonal Reduction

The tridiagonal reduction algorithm of a dense matrix is used in solving the symmetric eigenvalue problem. Solving the symmetric eigenvalue problem $Ax = \lambda x$ (A is symmetric) means to find the decomposition $A = Q\Lambda Q^{-1}$ where Λ is diagonal with Λ_{ii} being the i-th eigenvalue of A, Q is composed of the eigenvectors of A with $Q(:, i)$ being the eigenvector associated with Λ_i. $A = Q\Lambda Q^{-1}$ is also called the spectral decomposition of A. The symmetric eigenvalue problem can be solved using the power method or the QR algorithm. These two methods are usually too expensive in practice. So instead, an adapted QR algorithm is commonly used. First the matrix A is reduced to tridiagonal form using orthogonal similarity transformation $U^T AU = T$. Similarity transformations preserve eigenvalues, so T has the same eigenvalues as A. The eigenvalues and eigenvectors of the tridiagonal form T can be found much more easily than the eigenvalues and the eigenvectors of A [73, p. 119]. Then it is easy to compute the eigenvectors of the original matrix A using U and the eigenvectors of T.

The reduction of matrix A to tridiagonal form can be achieved using Householder reflections. We can find a Householder matrix U_1 so that $A_1 = U_1^T AU_1$ is tridiagonal in the first column and the first row. We continue this process and find a sequence of Householder matrices $U_2, U_3, \ldots U_n$ so that $(U_1U_2U_3 \ldots U_n)^T A(U_1U_2U_3 \ldots U_n) = A_n = T$ is in tridiagonal form. Implementing this process on a computer yields poor performance because it involves a lot of matrix-vector multiplies. Matrix-vector multiply is a BLAS 2 operation which have low performance on computers with a

hierarchical memory system. ScaLAPACK's tridiagonal reduction routine PDSYTRD implements a blocked version of the aforementioned Householder transformation method.

4.4.3 Blocked tridiagonal reduction

ScaLAPACK implements blocked version of the tridiagonal reduction algorithm. The blocked version is rich in BLAS 3 operations which have high flop:word ratios and are highly optimized on common computer architectures. In the blocked version, the input matrix A is partitioned into block columns of width nb. Householder reflectors from nb columns are accumulated then applied to the trailing matrix all at once.

Algorithm 4 is the ScaLAPACK tridiagonal reduction algorithm. **PDSYTRD**

Algorithm 4 PDSYTRD

1: **for** every panel **do**
2: PDLATRD on the panel, return V and W
3: PDSYR2K: $trail(A) = trail(A) - VW^\top - WV^T$
4: **end for**

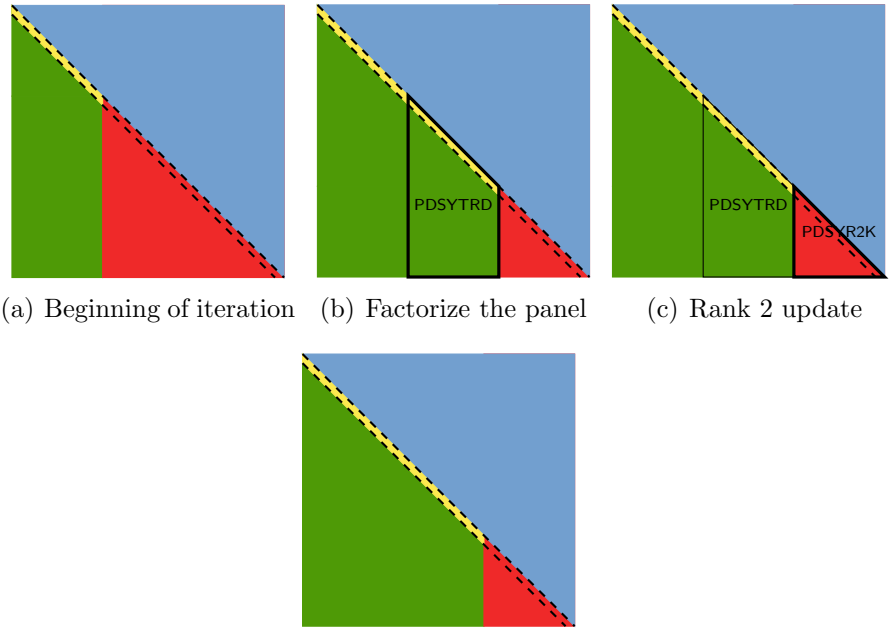
reduces a panel of nb columns of A to tridiagonal form. **PDSYR2K** performes a rank 2 update on the trailing submatrix. Algorithm 5 shows the operations performed by **PDSYTRD**. Figure 4.2 shows the memory footprint of one iteration of Algorithm 4.

Algorithm 5 PDLATRD

1: **for** the i-th column of the panel P (i from 1 to nb) **do**
2: Compute the Householder vector v_i which annihilates $P(i+2:end, i)$
3: Compute
$x_i = \tau(A^{(1)}v_i - W_{i-1}(V_{i-1}^T v_i) - V_{i-1}(W_{i-1}^T v_i))$
4: Compute $w_i = x_i - \tau v_i(v_i^T x_i)/2$
5: If $i < nb$, update the $i+1$-th column of P
6: **end for**

Figure 4.2(a) is the beginning the iteration. **PDSYTRD** takes advantage of the symmetry of A, only the lower triangular part of A is accessed and modified. The strictly upper triangular part of A (light blue) is not accessed by **PDSYTRD** and

remains unchanged through out the factorization. The green part is the factorized part, the red part on the right is the trailing matrix to be factorized. Figure 4.2(b) is the panel factorization. A panel of nb columns is reduced to tridiagonal form. The green part inside the panel stores the Householder vectors. Figure 4.2(c) is the rank 2 update to the trailing matrix. Taking advantage of the symmetry of the trailing matrix, only the lower triangular part is read and modified. Figure 4.2(d) is the state of the matrix at end of the iteration. This state is similar to the state at the beginning of the iteration, only that now the trailing matrix is nb columns smaller than that at the beginning of the iteration.



(a) Beginning of iteration (b) Factorize the panel (c) Rank 2 update

(d) End of iteration

Figure 4.2: One iteration of PDSYTRD

4.4.4 Failure model

Failures encountered in computer systems can be caused by various sources. They could be caused by hardware failures, software faults, cosmos rays or system overheating. Regardless of the cause, based on the effect of the failure we divide

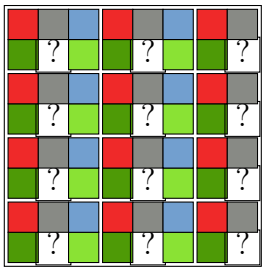


Figure 4.3: Global view of the matrix when a process fails.

failures into two categories: hard errors and soft errors. If a failure is perment (process crash, hardware malfunctional) and will be noticed by the system, we consider this failure a hard error. If a failure is transient and goes unnoticed by the system, we consider the failure a soft error. In this chapter we deal with hard errors.

In the case of **PDSYTRD**, a process failure will cause the data residing in that process to be lost. Figure 4.3 shows the status of the matrix in Figure 4.1(a) when one process fails. A White bloc with a question mark in it is a lost data block after the process fails. At this point the runtime is notified that a process has been lost. Measures have to be taken in order to proceed with the computation.

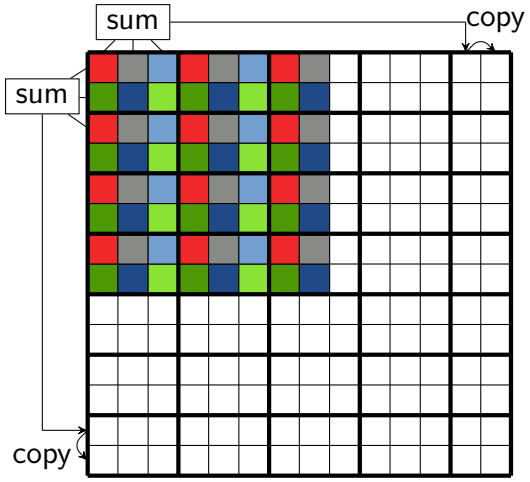
4.5 The Fault Tolerant Algorithm

In this section we describe our fault tolerant tridiagonal reduction algorithm. The main idea of our algorithm is to append checksums to the original matrix. These checksums can be updated through meaningful mathematical operations, so the checksums can be maintained with very little overhead.

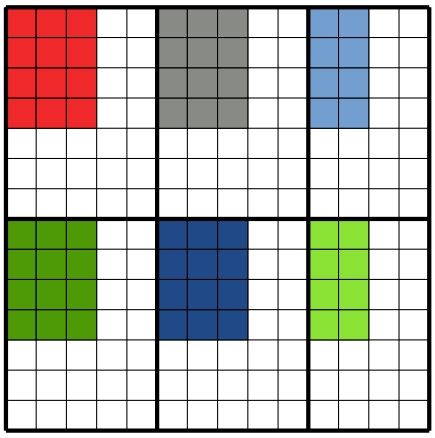
4.5.1 Initial checksum setup

In our algorithm we use two sets of checksums: column checksums at the bottom and row checksums on the right. The checksums are appended to the input matrix to form an extended matrix. The extended matrix is distributed over the original process grid in the 2D block cyclic fashion. In this layout, the distribution of the

original matrix data is exactly the same as in the case without the checksum data. In other words, if a process is assigned some data blocks of the matrix data in the stock ScaLAPACK **PDSYTRD** routine, this process will be assigned exactly the same data blocks of the original matrix plus some checksum blocks in our fault tolerant algorithm. Figure 4.4 shows the arrangement of the checksums of a matrix distributed over a 2×3 process grid. Colored blocks in the two graphs are data blocks belonging to the original matrix, white blocks are checksums appended to the original matrix. In Figure 4.4(a) is the global view of the extended matrix. In the global view the checksum blocks are located to the right and at the bottom of the original matrix. In this small example the number of checksum blocks seems relatively big comparing with the number of data blocks in the original matrix. But we will explain later that the proportion of the checksum blocks decreases to negligible when the process grid increases. The way the row checksums are calculated is as following: the first data block belonging to the same block row on each process are added together element-wise to form a checksum block. This checksum block is duplicated and stored next to itself. Since the checksum blocks are also distributed in the 2D block cyclic fashion, keeping two copies of the same checksum block next to each other ensures that these two checksum blocks will be assigned to two different processes. The benefit of this is that if one copy of the checksum block is lost due to a process failure, we still have another copy. Figure 4.4(b) shows the matrix from each process's perspective. In addition to the original matrix data (colored blocks), each process is assigned some checksum blocks (white blocks). Again, in this small example the checksum blocks seem a lot compared to the original blocks. But in reality when the process grid is large and the input matrix is large, the proportion of the checksums becomes very small.



(a) Matrix View



(b) Process View

Figure 4.4: An encoded matrix mapped to a 2×3 process grid.

4.5.2 The fault tolerant algorithm

If a matrix is distributed over a process grid of size $P \times Q$ where P is the number of rows and Q is the number of columns of the process grid, and the current iteration is factorizing the i-th block column, we define the *panel scope* as the group of Q block columns from the $\lfloor \frac{i}{Q} \rfloor$-th block column to the $(\lfloor \frac{i}{Q} \rfloor + Q - 1)$-th block column. Algorithm 6 shows our fault tolerant tridiagonal reduction algorithm. Line 1 computes the initial row checksums and column checksums. Before the factorization starts, the matrix is a symmetric matrix, the upper triangular part contains valid data. The initial checksum calculation is based on the entire matrix. After the factorization starts, the upper part of the matrix is not modified, it contains invalid data. But the checksums are updated in such a way that the checksums still encodes the entire symmetric matrix. That is to say if we replace the invalid upper triangular part of the matrix with valid data then compute the checksums, the checksum will equal the checksums we keep at the bottom of the matrix. Starting from line 2 the algorithm enters the factorization-update loop. In line 3 each process which owns a block column of the panel scope makes a local copy of that block column. This local copy will be used to recover lost data blocks inside the panel scope. Line 4 calls **FT-PDLATRD** to factorize the panel. **FT-PDLATRD** is a modified version of the stock **PDLATRD** routine in ScaLAPACK. It does everthing **PDLATRD** does, it also generates the checksums for the output V and W. We denote V and the checksums of V together as V_c. Similarly we denote W and the checksums of W together as W_c. Line 5 through line 7 recover from failures occurred during the panel factorization. Line 8 calls **FT-PDSYR2K** to apply a rank-2 update to the trailing matrix A_e. If a process failure occurs, line 9 through line 11 calls the routine **Recover** to recover data lost due to the failure.

Algorithm 7 is our modified panel factorization routine. It is very similar to the stock **PDLATRD** routine. The difference is that when computing the Householder vectors, **FT-PDLATRD** also scales the checksum part of the column so that the

resulting v_i is checksum protected. The output W is also checksum protected when **FT-PDLATRD** finishes. Notice that the checksums of V and W are updated through mathematical operations (GEMV and vector scaling). This approach to get the checksums is highly efficient comparing with performing a global reduction on the current panel.

Algorithm 6 Fault Tolerant Tridiagonal Reduction

1: Calculate the column checksum of the input matrix A, get A_e
2: **for** each i in $\lceil \frac{N}{nb} \rceil$ iterations **do**
3: Each process which owns part of the current panel makes a local copy of its own part of the panel
4: Call **FT-PDLATRD** on the panel, return V_c and W_c
5: **if** failure occurs **then**
6: Call **Recover** to recover from the failure
7: **end if**
8: Call **FT-PDSYR2K**, apply a block rank-2 update: $A_e^{(i+1)} = A_e^{(i)} - V_c W^T - W_c V^T$
9: **if** failure occurs **then**
10: Call **Recover** to recover from the failure
11: **end if**
12: **end for**

Algorithm 7 FT-PDLATRD

1: **for** the i-th column of the panel P (i from 1 to nb) **do**
2: Compute the Householder vector v_i which annihilates $P(i+2:N, i)$
3: Update the checksum of the i-th column $P(N+1:end, i)$
4: Compute $(x_e)_i = \tau(A_e^{(1)} v_i - (W_e)_{i-1}(V_{i-1}^T v_i) - (V_e)_{i-1}(W_{i-1}^T v_i))$
5: Compute $(w_e)_i = (x_e)_i - \tau(v_e)_i (v_i^T x_i)/2$
6: If $i < nb$, update the $i+1$-th column of P
7: **end for**

4.5.3 The invariant checksum relationship

In this section we prove the invariant checksum relationship which is the foundation of our ABFT algorithm.

Theorem 2. *The column checksums for block columns are valid at the end of each iteration.*

Proof. We use the phrase *current panel* to refer to the block column currently being factorized in the iteration. Our proof is based on mathematical induction. The proof uses Algorithm 6 as a reference. e is an all-one column vector $(1, 1, \ldots, 1)$.

1. Before the factorization the checksum blocks at the bottom of the original matrix are valid, because they are newly computed.

2. Assume the checksum blocks are valid before Algorithm 6 enters the i-th factorize-update loop, the checksum blocks are still valid before it enters the $(i+1)$-th factorize-update loop.

   (a) In the factorize-update loop, when the panel factorization is finished (line 4), the checksum blocks at the bottom are valid. The checksum blocks for the current panel have been updated to be valid according to Algorithm 7. The checksum blocks at the bottom for the trailing matrix have not been modified since the last iteration, so they are also valid.

   (b) After the rank-2 update to the trailing matrix, the checksum blocks at the

bottom of the trailing matrix are still valid.

$$
\begin{aligned}
& A_e^{(i)} - V_c W^T - W_c V^T \\
= & \begin{bmatrix} A^{(i)} \\ e^T A^{(i)} \end{bmatrix} - \begin{bmatrix} V \\ e^T V \end{bmatrix} \cdot W^T - \begin{bmatrix} W \\ e^T W \end{bmatrix} V^T \\
= & \begin{bmatrix} A^{(i)} \\ e^T A^{(i)} \end{bmatrix} - \begin{bmatrix} VW^T \\ e^T VW^T \end{bmatrix} - \begin{bmatrix} WV^T \\ e^T WV^T \end{bmatrix} \\
= & \begin{bmatrix} A^{(i)} - VW^T - WV^T \\ e^T A^{(i)} - e^T VW^T - e^T WV^T \end{bmatrix} \\
= & \begin{bmatrix} A^{(i)} - VW^T - WV^T \\ e^T (A^{(i)} - VW^T - WV^T) \end{bmatrix} \\
= & \begin{bmatrix} A^{(i+1)} \\ e^T A^{(i+1)} \end{bmatrix}
\end{aligned}
$$

3. By mathematical induction the checksum blocks are valid at the end of each factorize-update iteration.

□

4.5.4 Protecting the upper triangular matrix

In **PDSYTRD** the strictly upper triangular matrix is not accessed. On exit, the upper triangular matrix contains the same data as on entrance. In our fault tolerant algorithm we also maintain this behavior. This is where checksums on the right side come into the play. Before the factorization starts, we compute row-wise checksum blocks and store the checksum blocks on the right side of the original matrix. These checksum blocks only encodes the strictly upper triangular part of the matrix, each checksum block is duplicated and two copies of the same checksum block are stored next to each other horizontally. These checksum blocks are not accessed or changed

once they are computed. Once an failure occurs, a simple subtraction can recover the lost data block in the input matrix.

4.5.5 Recovery

At the end of the i-th iteration the state of the data matrix and the state of the checksum blocks are shown in Figure 4.6. Figure 4.6(a) is the state of the data matrix. The green part is the factorized part, it contains the Householder vectors. The light blue area is the untouched part. The red part is the trailing matrix. Figure 4.6(b) shows the left part of the bottom checksum blocks. The gray area at the bottom is the checksum blocks. The boxed gray area encodes the boxed green part. In Figure 4.6(c) the boxed gray part mathematically encodes the boxed the red part. In the ScaLAPACK implementation only the lower triangular of the boxed red part is stored. Figure 4.6(d) shows the checksum blocks on the right of the matrix encoding the upper triangular part of the data matrix. These checksum blocks are computed and written once, they are never changed ever since.

We partition the matrix into three areas (see Fgure 4.5) based on the state of the matrix. Each area needs to be treated differently in failure recovery. When a process

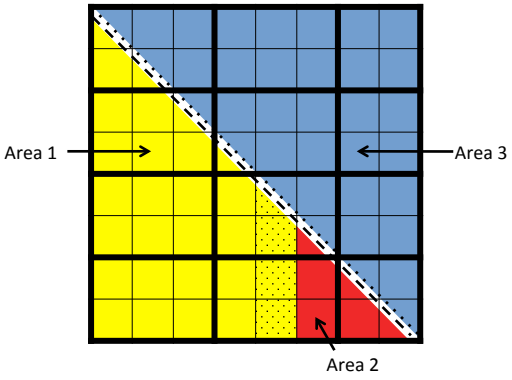


Figure 4.5: Partition for recovery

failure occurs, the **Recovery** procedure goes as follows:

1. Recover the runtime. Replace the dead process with a substitute process.

2. If any checksum block is lost, recover it using the other copy of the lost checksum block.

3. For area 1:

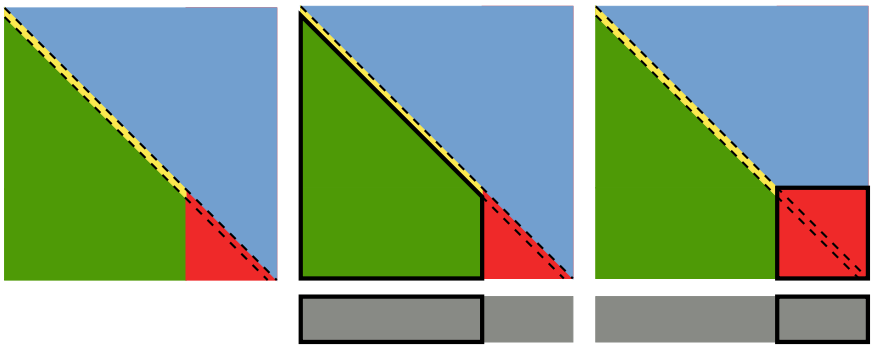
   (a) If the failure occurred during the panel factorization, all processes who own part of the current panel recover the lost blocks in the snapshot. The snapshot is checksum protected, so the lost blocks can be recovered using the checksums. Then copy the snapshot into the data matrix and re-factorize the panel. Recover the rest of area 1 using the checksums on the bottom.

   (b) If the failure did not occurred during the panel factorization, recover lost data blocks in area 1 using the checksums on the bottom.

4. For area 2, use the checksum blocks on the bottom to recover lost data blocks in area 2. The checksums on the bottom of area 2 encodes area 2 and the transpose of area 2. The algorithm retrieves data blocks from the lower triangular matrix to fill in the data blocks in the upper triangular matrix (see Figure 4.7). Then the checksums on the bottom can be used to recover the missing data block (the white block in Figure 4.7(b)).

5. Fore area 3, use the checksum blocks on the right to recover lost data blocks belonging to area 3.

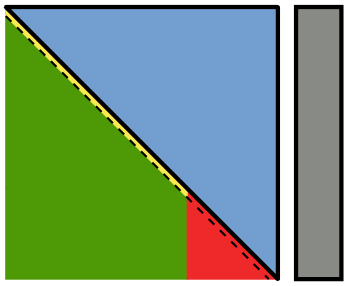
When we say recover a lost data block using the checksum block, we mean to first compute a new partial checksum of the data blocks on the live processes, then the lost data blocks can be obtained by subtracting the partial checksum from the checksum maintained during the factorization.



(a) End of iteration

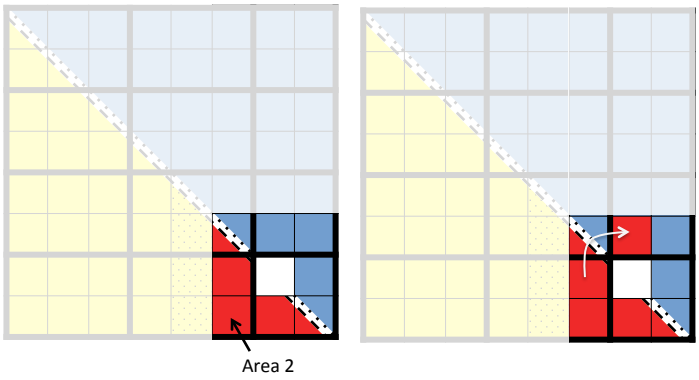
(b) Checksums for V

(c) Checksums for the traling matrix



(d) Checksums for the upper triangular part

Figure 4.6: State of the matrix after the i-th iteration



(a) One block lost in Area 2

(b) Copy blocks from the lower triangle

Figure 4.7: Partition for recovery

4.6 Complexity analysis

We analyze the complexity of the fault tolerant tridiagonal reduction algorithm in three aspects: storage overhead, communication overhead, and computation overhead.

In this section, N is the size of the input matrix. nb is the blocking factor in the 2D block cyclic data distribution. P is the number of rows in the processor grid, Q is the number of columns in the process grid.

Storage cost comes from the space for the checksum blocks, the storage for the snapshot of the current panel scope, and the storage for the buffer used in step 4 of the **Recovery** process.

The amount of storage for the checksum blocks at the bottom is

$$2\left\lceil \frac{N}{nb} \Big/ P \right\rceil \frac{N}{nb} nb^2$$

The amount of storage on the right of the input matrix is

$$2\left(\frac{N}{nb} + 2\left\lceil \frac{N}{nb} \Big/ P \right\rceil\right)\left\lceil \frac{N}{nb} / Q \right\rceil nb^2$$

The amount of storage for the snapshot is at most

$$\left(N + 2\left\lceil \frac{N}{nb} \Big/ P \right\rceil nb\right) Qnb$$

Adding them all together we have the total amount of extra storage:

$$2\frac{N^2}{P} + 2\frac{N^2}{Q} + 4\frac{N^2}{PQ} + NQnb + 2\frac{NQnb}{P}$$

If we compute the ratio between the extra storage needed by the fault tolerant

algorithm and the storage needed by the non fault tolerant algorithm, we get:

$$\frac{2}{P} + \frac{2}{Q} + \frac{4}{PQ} + \frac{Qnb}{N} + \frac{2Qnb}{NP}$$

The blocking factor nb is a relatively small number (in the hundreds). The number of rows P and the number of columns Q are close to each other, and they are smaller than N by several orders of magnitude. So we can denote the storage overhead as:

$$O\left(\frac{1}{P} + \frac{1}{Q} + \frac{1}{PQ} + \frac{Q}{N} + \frac{1}{N}\right)$$

The biggest term Q/N comes from the snapshot for the current panel scope. The total asymptotic storage overhead is very small.

The sources of computation overhead comes including generating the initial checksums, maintaining the checksums, and computation involved in the recovery procedure.

The initial checksum generation costs

$$\frac{N^2}{P}(P-1)$$

FLOPS. The additional FLOPS caused by updating the checksums during the panel factorization is

$$\begin{aligned} & \sum_{i=0}^{\lceil \frac{N}{P} \rceil - 1} 2\left(\frac{N}{P} - \lfloor i/P \rfloor \cdot nb\right)\left[2\left(N - i \cdot nb\right) - 1\right] \\ + \quad & 2 \sum_{i=0}^{\lceil \frac{N}{P} \rceil - 1} 2\left(\frac{N}{P} - \lfloor i/P \rfloor \cdot nb\right)(2nb - 1) \\ = \quad & \frac{4N^3}{3Pnb} + O(N^2) \end{aligned}$$

The additional computation needed to maintain the checksum blocks at the

bottom is

$$\begin{aligned} & 2 \cdot \sum_{i=0}^{\lceil \frac{N}{P} \rceil - 1} 2 \cdot 2 \left(\frac{N}{P} - \lfloor i/P \rfloor \cdot nb \right) \left[(N - i \cdot nb) \right] nb \\ + & \sum_{i=0}^{\lceil \frac{N}{P} \rceil - 1} 2 \left(\frac{N}{P} - \lfloor i/P \rfloor \cdot nb \right) \left[(N - i \cdot nb) \right] \\ = & \frac{(8nb+2)N^3}{3Pnb} + O(N^2) \end{aligned}$$

Adding these two costs together we have the total computation cost to maintain the checksum blocks at the bottom of the original matrix:

$$\frac{(8nb+6)N^3}{3Pnb} + O(N^2)$$

The additional FLOPS involved in recovering from a failure is N^2.

The computation complexity of the original parallel tridiagonal reduction algorithm is $4/3N^3$ FLOPS. So the computation overhead of the fault tolerant algorithm is:

$$\frac{(2nb+3)}{2Pnb} + O\left(\frac{1}{N}\right) = \frac{1}{P} + \frac{3}{2Pnb} + O\left(\frac{1}{N}\right)$$

As the number of process rows increases, the ratio of the computation overhead diminishes.

The extra communication caused by the fault tolerant algorithm only contains two components: communication during the initial checksum setup and communication during the recovery from a failure. The amount of data transferred between processes is N^2, the entire matrix. Each process owns $N/(nb \cdot P)$ block rows. Assuming the time cost to perform a reduction operation on one block row among the P processes in the same process column is T_P, the total time cost to generate the initial checksums

is

$$\frac{T_P \cdot N}{nb \cdot P}$$

Chapter 5

Hessenberg Reduction with Transient Error Resilience on GPU-Based Hybrid Architectures

5.1 Introduction

A transient error is an error in a signal or data element which is temporary, and caused by factors other than permanent component failures. Many phenomena have been blamed for transient errors, ranging from alpha particles from package decay, to cosmic rays and thermal neutrons. Cosmic rays were shown to be the most prevalent source of transient errors among these sources [96]. While transient errors may happen at different levels in the hardware hierarchy, such as communication links or digital logic, the most common situation is in the semiconductor storage.

Both GPUs and traditional CPUs, and their associated memory, are prone to transient errors. CPU designs increasingly scale the number of cores and the memory hierarchies in order to provide more processing ability. Along with increasing transistor density, newer CPU designs also adopt faster clock frequency and lower voltage. More transistors per unit area means the size of each transistor gets

smaller. A smaller feature size, combined with lower voltage to maintain transistor states, makes the transistor state easier to change, and therefore more vulnerable to external factors that might change the state. The critical charge Q_{crit}, which is the lowest electron charge needed to change the logical level, decreases as the chip feature size decreases. Higher transistor density also causes higher heat density which brings more thermal neutrons which contribute to transient errors as well. General Purpose Graphics Processing Units (GPGPUs) are gaining popularity in the scientific computing community due to the sizable acceleration they provide to computation intensive applications. A significant percentage of the acceleration is due to the large amount of data processing transistors inside the GPGPUs, where the number of transistors follow a even more drastic increase than in the CPU. As the evolution of the conventional processors and accelerators follows similar trends, the presence and frequency of transient errors have comparable progression and identical effect, becoming a disturbance to application developers.

Transient errors are also becoming a challenge for the applications. Both CPU main memory and GPU memory are DRAMs (Dynamic Random-access Memory). The measurement unit of soft error rate (SER) is Failure in time (FIT), and one FIT is one soft error in 10^9 device hours. Baumann [4] has reported that the SER of DRAM is between 1k FIT/chip - 10K FIT/chip range, and stays at the same level over 7 generations of DRAMs. Similarly, Jacob et al. [56] reported that at the 130 nm process SRAM memory exhibits a 100k FIT/chip. Michalak et al. [70] reported that the ASC Q supercomputer at Los Alamos National Laboratory experienced an average of 51.7 soft errors per week over a period of 7 weeks from September 2004 to October 2004. More recently, Haque et al. [52] assessed the probability of soft errors in NVIDIA GPUs using a benchmark called MemtestG80. They ran the test on 50000 GPUs and found that about 60% of the GPUs have a soft error probability (per test iteration) higher than 1×10^{-5}.

Useful science is based on facts, on experiments that can be replicated and results that can be trusted and verified. A single soft error can have a major impact on

the outcome of any computation as it can drastically alter the results, and thus the understanding of the analyzed phenomenon. In the extremely volatile execution environments we will encounter in the very near future, it is critical that the pillar of scientific applications, the notion of trust in the scientific outcome, is not undermined. This requires the data and the result to be carefully validated to ensure it matches the experiment, and it has not been altered during the computational phase. Ensuring this property is a difficult task if we are bound to generic methodologies. Fortunately, some of the most widely used algorithms have inherently properties that can be advantageously exploited in fulfilling this need.

In this chapter, we design and implement a soft error resilient Hessenberg reduction algorithm for GPU enabled hybrid architectures. We take advantage of diskless checkpointing, ABFT, and reverse computation techniques to achieve soft error resilience while introducing very little overhead compared to the non fault tolerant Hessenberg reduction. We further minimize the overhead by carefully overlapping workloads on the host side and the GPU side. Unlike the post-processing scheme for LU and QR in [37, 38, 39], our algorithm detects soft errors at the end of each iteration. Once detected, the errors are corrected right away, preventing the errors from propagating and contaminating other matrix elements. While the above mentioned post-processing scheme can only correct up to two soft errors total during the course of the entire LU or QR factorization, our fault tolerant Hessenberg algorithm can detect and correct more than one simultaneous soft error, assuming that the error positions in the matrix do not form a rectangle. Once the algorithm has corrected the simultaneous errors, it continues as normal and is ready to detect and correct subsequent soft errors as they occur.

The remainder of the chapter is organized as follows: in Section 5.2 we survey related work, then in Section 5.3 we explain the Hessenberg reduction algorithm and its implementation in the MAGMA framework. Section 5.4 describes our soft error resilient hybrid Hessenberg reduction algorithm in detail. Section 5.5 gives a formal analysis on the performance overhead of the fault tolerant algorithm. Section 5.6

presents the experiment results of the algorithm and provides a theoretical analysis for the performance. Section 5.7 summarizes our work.

5.2 Related work

Plank et al. [60] presented a fault tolerant technique based on checksum and reverse computation for matrix computations on networks of workstations (NOWs). Their scheme tackles node failures instead of soft errors. A checksum of each processor's local matrix data is stored in main memory and regenerated periodically. When a node failure happens, the live processors reverse the computations that occurred after the failure so that the matrix data and the checksum are consistent with each other. Then the lost data on the failed processor are recovered using the checksum and the data on the live processors. Chen and Abraham [25] devised methods to detect and locate faulty processors in the computation of eigenvalues and singular values on systolic arrays. Their methods take the special properties of eigenvalue computation and singular value computation into consideration to make the detection of errors very efficient.

While the field of fault tolerance was dominated for years by solutions to address hard errors, with the increase in the number of computing components, the impact of soft errors has attracted significant attention, especially in linear algebra. Based on the ABFT idea [54, 64, 65], Du et al. [37, 38] proposed an algorithm to tolerate soft errors in the High Performance LINPACK Benchmark (HPL) [33]. Their approach can compute the correct solution vector to $Ax = b$ in the presence of one or two soft errors over the course of the factorization. Du et al. [39] also designed a scheme to tolerate soft errors in the QR factorization on hybrid systems with GPGPUs. At most, one soft error can be tolerated in this fault tolerant hybrid QR algorithm. Both the HPL fault tolerant scheme and QR fault tolerant scheme adopt a post processing approach in which the erroneous result is corrected through post processing after the regular factorization. Bronevetsky and Supinski [18] studied the impact of

soft errors on iterative linear algebra methods. They found that iterative methods are vulnerable to soft errors as well and exhibit poor soft error detection abilities. In [81], Shantharam et al. analyzed the propagation pattern of soft errors in iterative methods by modeling the iterative process with a sequence of sparse matrix-vector multiplication (SpMV) operations. In [82], Shantharam et al. proposed a soft error tolerant preconditioned conjugate gradient algorithm for sparse linear systems. Their method adapted the algorithm based fault tolerance technique to sparse linear systems and achieved an overhead of 11.3% when no soft error occurs. Chen and Abraham [25] designed a concurrent error detection scheme for transient errors in the computation of eigenvalues on systolic processor arrays using the QR algorithm [47, 85] (not to be confused with the QR factorization). Cao et al. [20] designed a soft error resilient task-based runtime with three options to achieve fault tolerance.

In [75] Plank et al. first introduced the idea of diskless checkpointing which eliminates the disk access bottleneck in the traditional checkpointing technique. In the traditional checkpointing technique, checkpoints are stored to secondary stable memory, usually in the form of hard drives. Since disk accesses are very slow compared to floating point computation, frequently writing checkpoints to disk incurs a big overhead. With diskless checkpoint, the checkpoints are stored in main memory instead of hard disk. Main memory access is much faster than hard drive access, so diskless checkpointing can greatly reduce the memory access overhead.

Jia et al. proposed a fault tolerant algorithm for the parallel Hessenberg reduction [57], dealing with fail-stop scenarios (hard errors) in the context of distributed memory machines. Process failures can be tolerated using the ABFT technique, encoding and replicating the checksums to allow for inter-process data recovery. Our work is different from [57] in at least three major aspects. First, our algorithm is designed to tolerate soft errors. Soft errors are silent, they change the content of a memory location without triggering drastic responses from the OS. Unlike hard errors, which will be reported by the execution environment and must be dealt with in a more holistic way, soft errors need to be actively detected and prevented from

propagating. Second, our algorithm works on CPU-GPU hybrid platforms instead of distributed memory machines. Third, our algorithm uses reverse computation to roll back the trailing matrix update, making the error recovery easier and faster.

The Matrix Algebra on GPU and Multicore Architectures project (MAGMA) [88] is a dense linear algebra library for hybrid architectures with GPUs. The library provides equivalent functionalities to LAPACK [1] and uses block algorithms similar to those of LAPACK. By scheduling workloads with different characteristics to CPUs and GPUs, the hybrid algorithms are able to take advantage of both computational units and gain considerable acceleration over their LAPACK counterparts. The hybrid Hessenberg reduction algorithm in MAGMA also utilizes both CPUs and GPUs in a hybrid system. This hybrid algorithm is adapted from the LAPACK algorithm in order to separate workloads which are more suitable for GPUs from workloads that are suitable for CPUs. Details of this hybrid algorithm will be explained in the next section.

5.3 Hessenberg reduction on GPU enabled hybrid architectures

In this section we describe the Hessenberg reduction algorithm and its variation as implemented in MAGMA.

5.3.1 The Unblocked Hessenberg Reduction

A square matrix H in which all entries below the first subdiagonal are zeros is said to be in upper Hessenberg matrix form. Reduction of a square matrix A to the Hessenberg form H is an important intermediate step in the Hessenberg QR algorithm which is used to compute the eigenvalues of A. Given a square matrix A, we apply a

sequence of orthogonal similarity transformations Q_i to A:

$$H = Q_n^{-1} Q_{n-1}^{-1} \cdots Q_2^{-1} Q_1^{-1} A Q_1 Q_2 \cdots Q_{n-1} Q_n$$

let $Q = Q_1 Q_2 \cdots Q_{n-1} Q_n$, we have:

$$H = Q^{-1} A Q = Q^T A Q.$$

Q_i is chosen to be the Householder reflector, which eliminates the elements below the first subdiagonal in the i-th column of $Q_{i-1}^{-1} \cdots Q_1^{-1} A Q_1 \cdots Q_{i-1}$.

5.3.2 The Blocked Hessenberg Reduction

The speed of the unblocked Hessenberg reduction algorithm on modern computers is constrained by the latency of memory accesses. The blocked Hessenberg reduction algorithm [76] greatly increased the arithmetic intensity by grouping nb Householder reflectors and applying the group to A at the same time.

$$U_1 = Q_1 Q_2 \cdots Q_{nb} = I - V T V^T$$

where I is the identity matrix, V is an $N \times nb$ matrix composed of the Householder vectors, and T is an $nb \times nb$ upper triangular matrix. This representation of U_1 is called the *compact WY representation* [79]. This representation requires less storage to store U_1 and enables the use of matrix-matrix multiplications in the factorization. Matrix-matrix multiplications are desirable because of their high arithmetic intensity and efficient implementation on modern computers with hierarchical memory systems. Algorithm 8 shows the blocked Hessenberg reduction algorithm as implemented in the LAPACK **DGEHRD** routine. $trail(A)$ means the trailing submatrix in that iteration.

Algorithm 8 Blocked Hessenberg Reduction

1: **for** i from 1 to $\lceil \frac{N}{nb} \rceil$ **do**
2: **DLAHRD**, return V, T and Y where $Y = AVT$
3: **DGEMM**: $trail(A) = trail(A) - YV^\top$
4: **DLARFB**: $trail(A) = trail(A) - VT^\top V^\top trail(A)$
5: **end for**

5.3.3 Hessenberg Reduction in MAGMA

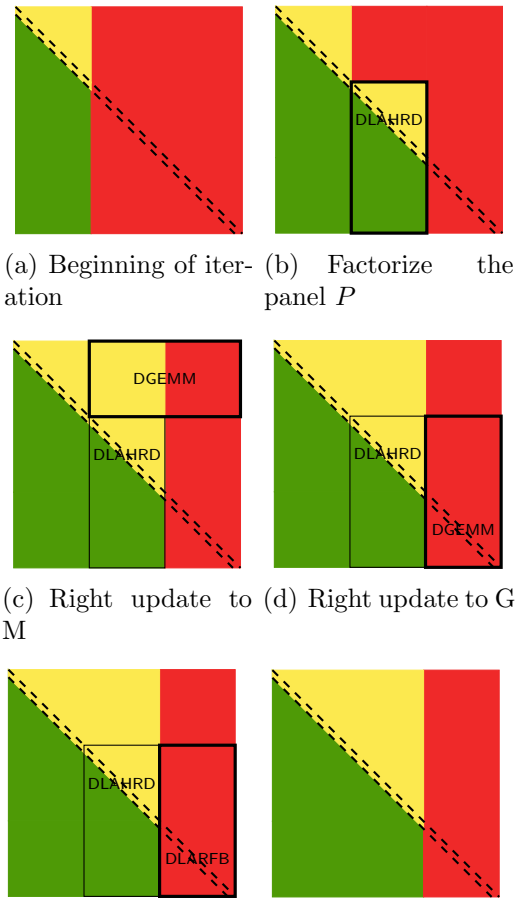
The hybrid Hessenberg reduction algorithm in MAGMA is an adapted version of Algorithm 8. Algorithm 9 shows the pseudocode for the hybrid Hessenberg reduction algorithm [89]. The input matrix A is stored in LAPACK layout, and matrix elements are stored contiguously in column major format. The matrix is logically divided into block columns, each block column is of size $N \times nb$. Upon completion, the matrix entries below the first subdiagonal are overwritten with the final Q matrix and the upper part of the matrix is overwritten with the final H matrix. The hybrid algorithm executes all the updates to the trailing matrix on the GPU. The panel factorization is assigned to the CPU, and the next panel to be factorized is transfered back to the host when both the right update and left update from the previous panel have been applied to it. Line 6 is an asynchronous data transfer, and control is returned to the CPU immediately after the data transfer is issued so that the CPU can initiate the next computation kernel. GPUs are able to do computation in parallel with communication, and using asynchronous data transfer hides the time cost to transfer the upper part of the current panel back to the CPU when it is updated and will not be modified again. The two lines in Algorithm 9, shown in red, are overlapped with each other.

Figure 5.1 visually illustrates one iteration of Algorithm 9; the computation routine called in each step and the data it operates on are pointed out with a black box. Figure 5.1(a) shows the state at the beginning of this iteration. The matrix elements in the yellow triangle and in the green trapezoid are the final results of the Q matrix and the H matrix, and they reside on the host side and will not be modified

again. The red rectangle is the trailing matrix which will be factorized and updated in this iteration. The first nb columns of the red part are called a *panel* which will be factorized next. Figure 5.1(b) shows the panel factorization **DLAHRD** which factorizes the lower part of the current panel. The yellow upper triangular matrix is updated and contains the final results of H. The green trapezoid contains the Householder vectors which are the final results in the Q matrix. Upon completion of **DLAHRD**, both the yellow triangle and the green trapezoid are on the host side. Figure 5.1(c) shows the right update on M. M is the part of the matrix marked by the black box which consists of the upper part of the current panel and the upper part of the trailing matrix. This step corresponds to line 5 of Algorithm 9. Upon completion of this step, the $nb \times nb$ square matrix in yellow contains the final results of H, and it will not be modified again. This square matrix is sent back to the host side with an asynchronous data transfer. Figure 5.1(d) shows the right update to G. The G matrix is the lower part of the trailing matrix marked by the black box. In figure 5.1(e) the left update to G is applied through the **DLARFB** call. After the **DLARFB** call, the matrix A has a smaller trailing matrix to be factorized in the next iteration. Figure 5.1(f) shows the state of the matrix at the end of this iteration. The rectangular matrix in red is the trailing matrix.

Algorithm 9 Hybrid Hessenberg Reduction

1: Transfer matrix: A on the host $\rightarrow$ d_A on the GPU
2: **for** i from 1 to $\lceil \frac{N}{nb} \rceil$ **do**
3: Send the lower part of the next panel P to the host.
4: **MAGMA_DLAHR2**, return V, T *and* Y
where $Y = [P, G]VT$
5: **DGEMM**: $M = M - MVTV^\top$
6: Send the leftmost nb columns of M to the host asynchronously.
7: **DGEMM**: $G = G - YV^\top$
8: **DLARFB**: $trail(A) = trail(A) - VT^\top V^\top trail(A)$
9: **end for**



(a) Beginning of iteration (b) Factorize the panel P

(c) Right update to G M (d) Right update to G

(e) Left update to G (f) End of iteration

Figure 5.1: One iteration of DGEHRD

5.4 Soft error resilient Hessenberg reduction algorithm

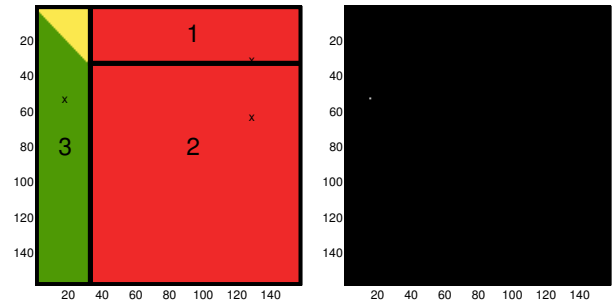
5.4.1 Failure Model

In this work we consider soft errors, which are temporary faults in the data matrix, where the factorization is oblivious to the error and continues as usual. Without loss of generality, we assume only one error happens at a single point in time.

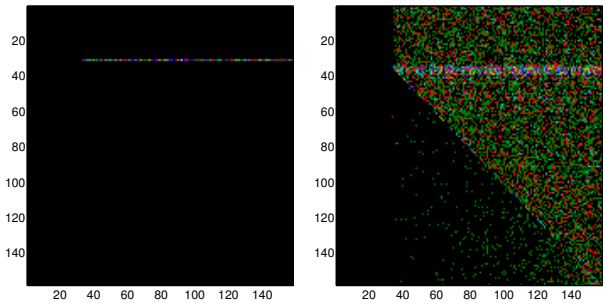
In the MAGMA Hessenberg reduction algorithm, both the CPU and GPU carry out computation. The CPU is responsible for the panel factorization, and the GPU is

responsible for the trailing matrix update. Both the CPU memory and GPU memory contain part of the final result or intermediate data that are used to compute the final result. The lower triangular matrix to the left of the current panel on the host side contains part of the final result of the Q matrix. The upper triangular matrix to the left of the current panel on the host side contains the final result of the H matrix. On the GPU, the rectangular matrix to the right of the current panel contains intermediate data that will be used to compute Q and H. Soft errors in either one of these parts will cause the factorization to give an incorrect result. We need to detect and correct soft errors in both the CPU memory and the GPU memory. The algorithm we propose in this work combines the advantage of the ABFT technique and the diskless checkpointing technique. The algorithm also uses reverse computation to roll back the program data to a previous state.

Depending on the location of the soft error, an error has different impacts on the result of the factorization. Figure 5.2 shows the impact of a soft error when it happens in three different locations. In this example, the matrix size N is set to 158, and the block size is 32. In all three figures, the soft error is injected when the first iteration has finished, and the second iteration has not yet started. Figure 5.2(a) is the partitioning of the matrix. Each of the following three figures shows the heat map of the difference matrix between the error-free result and the result when an error has happened during the factorization. Black means the difference is 0. Other colors mean the difference is bigger than 0, with each color representing a magnitude range. In Figure 5.2(b), the error occurs at location $(53, 16)$. This location is marked by an x in region 3 on the left in Figure 5.2(a). This error does not propagate as the factorization proceeds. We can see that in the final result of the factorization there is still only one incorrect element (shown as the white dot in the upper left part of the matrix). In Figure 5.2(c), the error happens at location $(31, 127)$. This location is marked by an x in region 1 shown in Figure 5.2(a). This soft error propagates row-wise, and pollutes the entire row in H when the factorization completes. In Figure 5.2(d), the error occurs at location $(63, 127)$. This location is marked by an x in region 2, shown



(a) Partition (b) Error location $(53, 16)$



(c) Error location $(31, 127)$ (d) Error location $(63, 127)$

Figure 5.2: Propagation pattern of errors at different locations

in Figure 5.2(a). An error in this region causes the most damage among the three scenarios. When the factorization completes, almost all the elements after column 32 in H are polluted, and many elements after column 32 in Q are polluted.

5.4.2 Encoding the Input Matrix

To recover from an error we need redundant information. We add redundancy to the input matrix by appending an extra column at the right side of the matrix, and an extra row at the bottom of the matrix. An element in the extra column is the summation of all the elements in the same row in the input matrix. Similarly, an element in the extra row is the summation of all the elements in the same column of the original matrix. Figure 5.3 shows the initial state of the encoded input matrix.

We define the following notations: A_{r_chk} is the column of row checksums on the right side of the original matrix; A_{c_chk} is the row of column checksums at the bottom

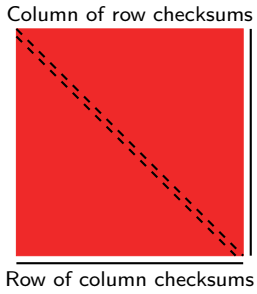


Figure 5.3: The encoded initial matrix

of the original matrix. A_{re} is the original matrix appended with A_{r_chk} on the right side (*re* for *rowwise encoded*). A_{ce} is the original matrix appended with A_{c_chk} at the bottom (*ce* for *columnwise encoded*). A_{fe} is the original matrix appended with both A_{r_chk} and A_{c_chk} (*fe* for *fully encoded*).

5.4.3 The Fault Tolerant Algorithm

In this subsection we present and explain our soft error tolerant Hessenberg reduction algorithm. e is an all one vector: $e = (1, 1, \cdots, 1, 1)^\top$. Algorithm 10 is the pseudocode for the fault tolerant algorithm.

The input matrix resides on the host side when the algorithm begins; in Algorithm 10 line 1 sends the input matrix to the GPU. Line 2 encodes the input matrix to obtain A_{fe}. Starting from line 3 the algorithm enters a **for** loop, this **for** loop iterates over the block columns of A. In each loop the algorithm first sends the lower part (the part marked by the black box in Figure 5.4(b)) of the next panel to the CPU from the GPU in line 4. In line 6 and line 7 the algorithm computes the column checksums for matrix Y and matrix V. This procedure requires two GEMV operations on the GPU. Line 8 applies the right update to matrix M_{re}. This line corresponds to Figure 5.4(c), and matrix M_{re} is the matrix marked by the black box in the figure. Line 9 and line 10 (in red text) overlap with each other. Line 10 applies the right update to matrix G. This corresponds to Figure 5.4(d). Line 11 applies the left update from the panel to matrix G, and this operation is illustrated

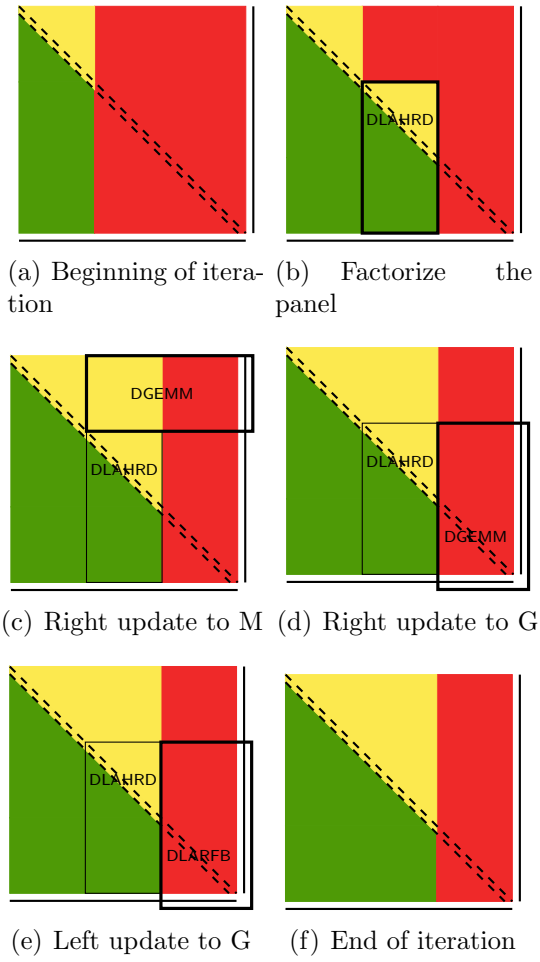
Algorithm 10 Fault Tolerant Hybrid Hessenberg Reduction

1: Transfer matrix: A on the host $\rightarrow$ d_A on the GPU
2: Encode the input matrix, expand it with a checksum column and a checksum row.
3: **for** i from 1 to $\lceil \frac{N}{nb} \rceil$ **do**
4: Send the lower part of the next panel P to the host.
5: **MAGMA_DLAHR2**, return V, T and Y where $Y = [P, G]VT$
6: Obtain Y_{ce} by computing the column checksums of Y: $Y_{chk_c} = trail(A)_{chk_c} \cdot V$
7: Obtain V_{ce} by computing the column checksums of V: $V_{chk_c} = e^\top \cdot V$
8: **DGEMM**: $M_{re} = M_{re} - MVTV_{ce}^\top$
9: Send the leftmost nb columns of M to the host asynchronously.
10: **DGEMM**: $G_{fe} = G_{fe} - Y_{ce}V_{ce}^\top$
11: **DLARFB**: $trail(A)_{fe} = trail(A)_{fe} - V_{ce}T^\top V^\top trail(A)$
12: Compute $S_{re} = \sum A_{re}(i)$ and $S_{ce} = \sum A_{ce}(i)$
13: **if** $|S_{re} - S_{ce}| > threshold$ **then**
14: Reverse the last left update and right update.
15: Correct the error.
16: **end if**
17: **end for**

in Figure 5.4(e).

We prove that, after line 11 in Algorithm 10, the column of row checksums and the row of column checksums are still valid for the yellow part and the red part in Figure 5.4(f). The proof is presented in the next subsection.

Line 12 through line 16 check for the existence of a soft error. The algorithm corrects the error if there is any. Line 12 computes the summations of the checksum row and the checksum column. Since they contain checksums of the same matrix data along different directions, the summation of each vector should equal each other. Taking rounding errors into consideration, we check the difference against a threshold. If the difference exceeds the threshold, we consider an error has happened. The threshold should be big enough to tolerate roundoff errors, at the same time it should be small enough to avoid false negatives. A proper choice of the threshold is a value larger than the machine epsilon by 2 to 3 orders of magnitude. At this point the soft error in the matrix element has propagated to both the checksum column and the



(a) Beginning of iteration (b) Factorize the panel

(c) Right update to M (d) Right update to G

(e) Left update to G (f) End of iteration

Figure 5.4: One iteration of FT_DGEHRD

checksum row, the checksums are not valid any more. Line 14 reverses the last left update and the last right update so that the checksum column and the checksum row, together with the matrix data, are restored to their states at the end of the previous iteration. The checksum relationship is made valid again. The reverse computation is possible because the intermediate data used to apply the last last left update and right update are still available at the end of the iteration. They will not be destroyed until the next panel factorization. The algorithm then enters the recovery procedure.

5.4.4 The Checksum Relationship

In this subsection we prove the following theorem:

Theorem 3. *The checksum column on the right of matrix A and the checksum row at the bottom of matrix A are valid at the end of each iteration.*

Proof. 1. The checksum column and the checksum row are valid after line 2 since they are newly computed.

2. The checksum column and the checksum row are valid after the right update to the trailing matrix.

$$
\begin{aligned}
A_{fe} &= A_{fe} - \begin{bmatrix} A \\ e^\top A \end{bmatrix} VT \begin{bmatrix} V \\ e^\top V \end{bmatrix}^\top \\
&= \begin{bmatrix} A & Ae \\ e^\top A & 0 \end{bmatrix} - \begin{bmatrix} AVTV^\top & AVTV^\top e \\ e^\top AVTV^\top & e^\top AVTV^\top e \end{bmatrix} \\
&= \begin{bmatrix} (A - AVTV^\top) & (A - AVTV^\top)e \\ e^\top (A - AVTV^\top) & * \end{bmatrix}
\end{aligned}
$$

3. The checksum column and the checksum row are valid after the left update to the checksum.

$$
\begin{aligned}
A_{fe} &= A_{fe} - \begin{bmatrix} V \\ e^\top V \end{bmatrix} T^\top V^\top \begin{bmatrix} A & Ae \end{bmatrix} \\
&= \begin{bmatrix} A & Ae \\ e^\top A & 0 \end{bmatrix} - \begin{bmatrix} VT^\top V^\top A & VT^\top V^\top Ae \\ e^\top VT^\top V^\top A & e^\top VT^\top V^\top Ae \end{bmatrix} \\
&= \begin{bmatrix} (A - VT^\top V^\top A) & (A - VT^\top V^\top A)e \\ e^\top (A - VT^\top V^\top A) & * \end{bmatrix}
\end{aligned}
$$

4. According to Mathematical Induction, the checksum row and the checksum column are valid at the end of each iteration.

□

5.4.5 Protecting Q

The Q matrix contains the Householder vectors which were used to apply the similarity transformations to A. These Householder vectors are not protected by the checksums that encode the H matrix, we should provide protection for Q through other schemes. These Householder vectors are generated on the host side and stay there until the entire factorization finishes. They are not modified after they are generated. Moreover, they are not even read after the iteration in which they were generated finishes. Hence, it suffices to maintain a checksum for each row in order to correct an error. But just like the situation in detecting a soft error in H, we need both a checksum row and a checksum column to determine both the error column index j and error row index i. We keep the checksums for Q on the host. Q_{r_chk} is the rowwise checksum vector, and Q_{c_chk} is the columnwise checksum vector.

Figure 5.5 shows the process for generating and updating the checksums for the Q matrix. The dashed line on the left of the matrix is the column of row checksums for Q. When a new panel factorization is finished as the one shown in Figure 5.5, we compute the row checksums for the newly finished panel. Then the partial checksums for the panel are applied to the dashed line on the left so that the dashed line protects the entire green part. The dashed line at the bottom of the matrix is the row of column checksums for Q. This vector is computed segment by segment. When a new panel factorization is done on an nb wide panel, an nb long segment of the column checksums is also generated. The solid line segment at the bottom of the panel in Figure 5.5 is the newly generated column checksum segment for Q. This segment is never changed once generated.

Our algorithm overlaps the checksum generation for Q with the update to the

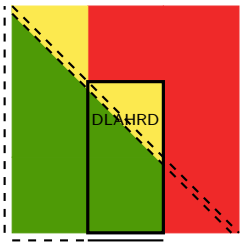


Figure 5.5: Maintaining the checksums for Q

trailing matrix on the GPU. The checksum generation involves two GEMV operations. GEMV is a level 2 BLAS operation which is a memory bound operation. We choose to perform the checksum generation on the CPU while the GPU is updating the trailing matrix. The CPU is idle in the non-fault tolerant MAGMA Hessenberg reduction algorithm, and our arrangement hides the time cost of the checksum generation.

5.4.6 Recovery

Once we have detected a soft error, we first determine the row index and the column index of the soft error before we can correct the error. We recalculate a checksum column A'_{r_chk} and a checksum row A'_{c_chk} of the current matrix (the yellow part and the red part in Figure 5.4(f)). Then we compare A'_{r_chk} and A_{r_chk}, and the error row index i can be determined if $A'_{r_chk}(i) \neq A_{r_chk}(i)$. Similarly, the error column index j can be determined by comparing A'_{c_chk} and A_{c_chk}.

The erroneous element can be corrected using the formula $A(i,j) = A_{r_chk}(i) - \sum_{k=1}^{k\leq n, k\neq j} A(i,k)$ or the formula $A(i,j) = A_{c_chk}(j) - \sum_{k=1}^{k\leq n, k\neq i} A(k,j)$.

Since a soft error in the Q matrix does not propagate, we only examine the checksum relationship once, at the end of the factorization. The error detection and correction scheme is similar to those for the H matrix, except that it is carried out once at the end of the entire factorization instead of once per iteration.

5.5 Performance Evaluation

In this section we give a formal analysis for the overhead of our fault tolerant Hessenberg reduction algorithm. The fault tolerant Hessenberg reduction algorithm performs extra floating point operations and extra data transfers between the host and the GPU in addition to those in the original MAGMA Hessenberg reduction. The fault tolerant algorithm also consumes extra storage to keep data redundancy. So, we evaluate the overhead in terms of extra FLOPS, extra communication, and extra storage. We denote the matrix dimension as N, the block size as nb, and the amount of floating point operations as *FLOP*.

After the algorithm transfers the input matrix to the GPU, the algorithm computes the global row checksums and the column checksums for the input matrix. This involves two DGEMV operations on the GPU: $A_{r_chk} = Ae$ and $A_{c_chk} = e^\top A$. The amount of floating operations:

$$FLOP_{init} = 2N(N + N - 1) = 4N^2 - 2N.$$

In every iteration, the algorithm computes column checksums for matrix V. In the i-th iteration, the dimension of matrix V is $(N - nb \cdot i) \cdot nb$. The accumulated FLOP count over the course of the factorization is:

$$\begin{aligned} FLOP_{chkV} &= \sum_{i=0}^{N/nb-1} nb \cdot (N - nb \cdot i + N - nb \cdot i - 1) \\ &= O(N^2). \end{aligned}$$

The amount of floating point operations applied on the right hand side checksums is:

$$\begin{aligned} FLOP_{r_chk} = & \sum_{i=0}^{N/nb-1} \{(N - nb \cdot i) \cdot (nb + nb - 1) \\ & + N \cdot (nb + nb - 1) + nb \cdot [(N - nb \cdot i) + (N - nb \cdot i) - 1]\} \\ & = O(N^2). \end{aligned}$$

The amount of floating point operations applied on the bottom checksums is:

$$\begin{aligned} FLOP_{c_chk} = & \sum_{i=0}^{N/nb-1} [(N - nb \cdot i)(nb + nb - 1) \\ & + (N - nb \cdot i)(nb + nb - 1)] = O(N^2). \end{aligned}$$

The amount of floating point operations spent on intermediate results used by both row and column checksums is:

$$FLOP_{common} = \sum_{i=0}^{N/nb-1} nb \cdot (nb + nb - 1) = O(N).$$

The computation cost to detect the error in Algorithm 10 requires two dot product operations, one for the summation of the row checksums, and one for the summation of the column checksums. The total cost is given by:

$$FLOP_D = \sum_{i=0}^{N/nb-1} 2(N + N - 1) = O(N^2).$$

Adding all these together we get the total amount of extra floating point operations performed by the fault tolerant algorithm:

$$\begin{aligned} FLOP_{extra} = & FLOP_{init} + FLOP_{chkV} + FLOP_{r_chk} \\ & + FLOP_{c_chk} + FLOP_{common} + FLOP_D = O(N^2). \end{aligned}$$

The computation complexity of the Hessenberg reduction is $FLOP_{orig} \sim 10/3N^3$, so when there is no errors, the overhead of the fault tolerant Hessenberg reduction in terms of FLOP percentage is:

$$Overhead = \frac{FLOP_{orig}}{FLOP_{extra}} = \frac{O(N^2)}{10/3N^3} = \frac{3}{10}O(N^{-1}).$$

When N increases the overhead tends to: 0.

In order to locate the error, a vector of new row checksums and a vector of new column checksums need to be computed on the matrix consisting of the yellow part and the red part in Figure 5.2(a). The cost is given by:

$$FLOP_L = 2N(N + N - 1) = 4N^2 - 2N.$$

To correct the error requires a dot product and a subtraction:

$$FLOP_C = N - 2 + 1 = N - 1.$$

After an error has been detected, the algorithm performs a roll back by reverse update, which includes a reverse left update and a reverse right update. Then the pre-factorized panel is retrieved from the buffer, and the entire iteration is repeated after the error correction. The amount of overhead is a function of the size of the trailing matrix. Assume the error occurred in the j-th iteration, and we have:

$$\begin{aligned}
&FLOP_{redo} = FLOP_{repeat} + FLOP_{panel} \\
&\approx N \cdot (N - j \cdot nb)(2nb - 1) + \\
&\quad (N - j \cdot nb) \cdot (N - j \cdot nb)(2nb - 1) \\
&\quad + (N - j \cdot nb) \cdot nb \cdot [(N - j \cdot nb) + (N - j \cdot nb) - 1] \\
&\quad + (N - j \cdot nb) \cdot nb \cdot (nb + nb - 1) \\
&= O(N^2).
\end{aligned}$$

Table 5.1: Detailed specification of the test platform.

	CPU	GPU
Processor model	Intel Xeon E5-2670	NVIDIA Tesla K40c
Clock frequency	2.6 GHz	745 MHz
Memory	62 GB	11519 MiB
Peak DP	10.4 Gflop/s	1.43 Tflop/s
BLAS/LAPACK	Intel MKL 11.2	CUBLAS 7.0.28
OS	CentOS 6.4	-
Compiler	gcc 4.4.7	nvcc 7.0 V7.0.27

Compared with the computation cost of the original Hessenberg reduction, the extra FLOP introduced by the fault tolerant algorithm is very low. It tends to 0 when n increases.

The storage requirement of the fault tolerant Hessenberg reduction algorithm consists of a panel worth of work space for the intermediate result to update the trailing matrix, and four columns worth of space for the checksums:

$$S = nb \cdot N + 4 \cdot N$$

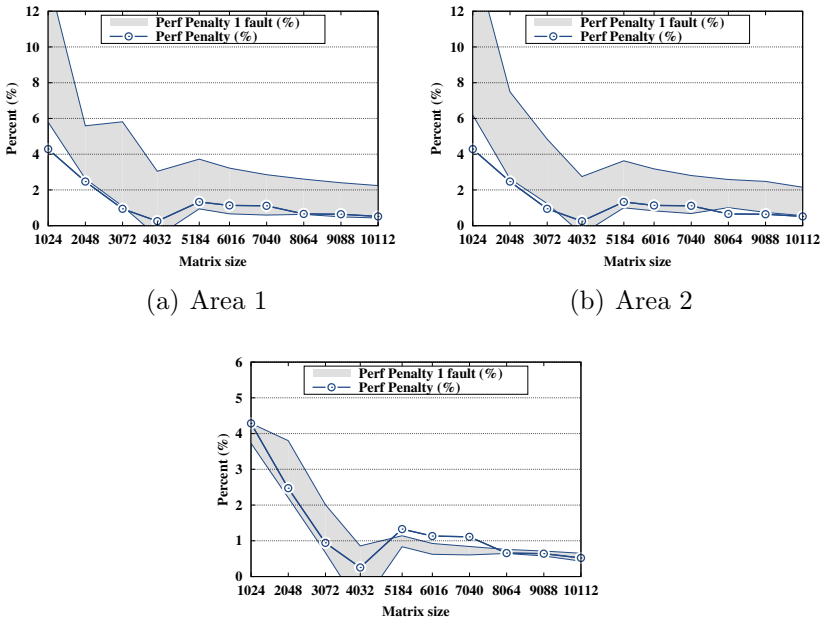
5.6 Experiments

In this section we present the performance of our fault tolerant algorithm. Our testbed consists of an Intel Sandy Bridge-EP CPU and an NVIDIA Kepler GPU. The detailed specifications of the test platform are listed in Table 5.1.

5.6.1 Performance Study

As shown in Figure 5.2(a), during the factorization the matrix is partitioned in three areas. We analyze the performance of our algorithm when the soft error occurs in each of the different areas, at different moments of the factorization.

Figure 5.6(a) shows the performance overhead in the case where the soft error occurs in area 1 (see Figure 5.2(a)). This overhead includes setting up and maintaining the checksums, the reverse update to the trailing matrix, and the re-execution of



(a) Area 1

(b) Area 2

(c) Area 3

Figure 5.6: Overhead of FT-Hess. The blue line is the overhead without failures, while the gray area is the uncertainty interval when one single error is introduced in a specific area (as described in Figure 5.2(a)).

the faulty iteration. Among all these costs, the most expensive step is the panel factorization when re-executing the faulty iteration. When the error occurs early in the factorization, the size of the panel which the algorithm re-factorizes is larger, and the performance overhead is also larger. The gray area in the figure indicates the range of the overhead depending on the moment when the single fault is introduced in Area 1. We can see that the overhead range remains small for all matrix sizes while the overhead exhibits a decreasing trend as the matrix size grows; at matrix size 10112×10112 the overhead is less than 4% when one error occurs in Area 1.

Figure 5.6(b) shows the performance overhead of the fault tolerant algorithm when the soft error occurs in area 2 (see Figure 5.2(a)). Similar to Figure 5.6(a), the overhead is dependent on the moment when the error occurs. It maintains the same constant range and it exhibits the same decreasing trend as the matrix size grows. The performance overhead is less than 4% at matrix size 10112×10112.

Table 5.2: Numerical Stability A1, A2, A3

Matrix Size	MAGMA Hess	Area 1 FT-Hess B	Area 1 FT-Hess M	Area 1 FT-Hess E	Area 2 FT-Hess B	Area 2 FT-Hess M	Area 2 FT-Hess E	Area 3 FT-Hess B/M/E
1024	6.2529×10^{-18}	6.2764×10^{-18}	6.2520×10^{-18}	6.2540×10^{-18}	6.2764×10^{-18}	6.2520×10^{-18}	6.2540×10^{-18}	3.9780×10^{-16}
2048	2.6291×10^{-18}	2.6552×10^{-18}	2.6502×10^{-18}	2.6276×10^{-18}	2.6552×10^{-18}	2.6502×10^{-18}	2.6276×10^{-18}	1.6047×10^{-15}
3072	8.0088×10^{-18}	8.0023×10^{-18}	7.9987×10^{-18}	8.0066×10^{-18}	8.0023×10^{-18}	7.9987×10^{-18}	8.0066×10^{-18}	1.9576×10^{-15}
4032	8.4784×10^{-18}	8.4697×10^{-18}	8.4747×10^{-18}	8.4790×10^{-18}	8.4697×10^{-18}	8.4747×10^{-18}	8.4790×10^{-18}	1.9473×10^{-14}
5184	1.2012×10^{-17}	1.2024×10^{-17}	1.2008×10^{-17}	1.2011×10^{-17}	1.2024×10^{-17}	1.2008×10^{-17}	1.2011×10^{-17}	2.5166×10^{-15}
6016	1.5892×10^{-17}	1.5881×10^{-17}	1.5891×10^{-17}	1.5892×10^{-17}	1.5881×10^{-17}	1.5891×10^{-17}	1.5892×10^{-17}	4.3368×10^{-15}
7040	1.9573×10^{-17}	1.9580×10^{-17}	1.9571×10^{-17}	1.9571×10^{-17}	1.9580×10^{-17}	1.9571×10^{-17}	1.9571×10^{-17}	2.6158×10^{-14}
8064	3.7656×10^{-18}	3.7575×10^{-18}	3.7690×10^{-18}	3.7656×10^{-18}	3.7575×10^{-18}	3.7690×10^{-18}	3.7656×10^{-18}	8.9874×10^{-15}
9088	6.3745×10^{-18}	6.3814×10^{-18}	6.3736×10^{-18}	6.3746×10^{-18}	6.3814×10^{-18}	6.3736×10^{-18}	6.3746×10^{-18}	2.2618×10^{-14}
10112	1.7536×10^{-17}	1.7531×10^{-17}	1.7535×10^{-17}	1.7536×10^{-17}	1.7531×10^{-17}	1.7535×10^{-17}	1.7536×10^{-17}	2.4302×10^{-14}

Figure 5.6(c) shows the performance overhead of the fault tolerant algorithm when the soft error occurs in Area 3 (see Figure 5.2(a)). In this case we can see that the performance overhead is smaller, closely following the overhead of the case without failures. There are two reasons for this phenomenon: the error detection and correction are only carried out once at the end of the factorization, and after an error is detected, only a dot product is necessary to correct the error. In contrast, an error in either area 1 or 2 requires a reverse update, a repeated panel factorization, and a trailing matrix update. We also observe that the uncertainty interval of the performance overhead is very small at all matrix sizes. No matter when the error occurred during the factorization, they are treated at the end with the same procedure, with the same minimalistic approach. Therefore they incur the same amount of overhead. Overall, these results indicate that our approach is a practical solution to ensure the correctness of the Hessenberg reduction with minimal overhead, and that this overhead consistently decreases as the size of the matrix increases. Also, these results are consistent with the results reported in [57].

5.6.2 Numerical Stability

In this subsection we investigate the numerical behavior of our fault tolerant Hessenberg algorithm compared with the non-fault tolerant algorithm.

The block Hessenberg reduction algorithm implemented in MAGMA is backward

stable. The following residual is used to verify the factorization result:

$$r = \frac{\|A - QHQ^\top\|_1}{N\|A\|_1}$$

where A is the input matrix, and N is the matrix dimension. Table 5.2 shows the comparison of the residuals as obtained from the original MAGMA non-fault tolerant algorithm and our fault tolerant algorithm with one soft error.

The three main sections of the table indicate the location of the error, Area 1, Area 2, or Area 3. In each section the letter appended to the name of the column indicates the moment when the error occurs, B for the beginning of the factorization, M for the middle, and finally, E for the end of the factorization. Finally, in the case of Area 3, all columns were collapsed into a single column as the residuals were identical. We can see that for every matrix size the residuals from Area 1 and Area 2 are on the same order of magnitude, with minimal variations, as the original MAGMA algorithm. In some cases the fault tolerant algorithm even has a smaller residual than the fault free original algorithm. When the error was introduced on the left part of the matrix (i.e., Q, in Area 3) the final residuals are higher than their counterparts in the MAGMA routine, but they are still within the acceptable range. The extra amount of error compared with the classic algorithm is introduced by the encoding/recovery process. In the encoding phase, N elements (in a row or column) are added together to form one checksum element. In the recovery phase, $N-1$ elements are subtracted from the checksum element. Both phases are implemented as dot products. We refer interested readers to [23] for a detailed discussion of rounding errors in dot products. Overall, these results are evidence that our fault tolerant Hessenberg reduction algorithm can successfully correct soft errors without degrading the stability of the original algorithm.

Table 5.3: Orthogonality of Q A1, A2, A3

Matrix Size	MAGMA Hess	Area 1 FT-Hess B	Area 1 FT-Hess M	Area 1 FT-Hess E	Area 2 FT-Hess B	Area 2 FT-Hess M	Area 2 FT-Hess E	Area 3 FT-Hess
1024	3.65×10^{-17}	3.80×10^{-17}	3.41×10^{-17}	3.36×10^{-17}	3.64×10^{-17}	3.46×10^{-17}	3.35×10^{-17}	6.84×10^{-16}
2048	3.72×10^{-17}	3.61×10^{-17}	3.71×10^{-17}	3.65×10^{-17}	3.53×10^{-17}	3.64×10^{-17}	3.64×10^{-17}	2.76×10^{-15}
3072	3.62×10^{-17}	3.40×10^{-17}	3.57×10^{-17}	3.63×10^{-17}	4.61×10^{-17}	3.63×10^{-17}	3.65×10^{-17}	3.31×10^{-15}
4032	3.75×10^{-17}	3.40×10^{-17}	3.75×10^{-17}	3.75×10^{-17}	3.98×10^{-17}	3.81×10^{-17}	3.77×10^{-17}	3.28×10^{-14}
5184	4.59×10^{-17}	3.78×10^{-17}	3.63×10^{-17}	3.61×10^{-17}	3.92×10^{-17}	3.62×10^{-17}	3.62×10^{-17}	4.19×10^{-15}
6016	3.74×10^{-17}	3.71×10^{-17}	3.63×10^{-17}	3.62×10^{-17}	3.89×10^{-17}	3.60×10^{-17}	3.62×10^{-17}	7.19×10^{-15}
7040	4.10×10^{-17}	4.44×10^{-17}	4.51×10^{-17}	4.50×10^{-17}	4.00×10^{-17}	4.52×10^{-17}	4.51×10^{-17}	4.35×10^{-14}
8064	3.64×10^{-17}	3.31×10^{-17}	3.74×10^{-17}	3.74×10^{-17}	3.58×10^{-17}	3.77×10^{-17}	3.74×10^{-17}	1.49×10^{-14}
9088	3.64×10^{-17}	3.75×10^{-17}	4.22×10^{-17}	4.22×10^{-17}	4.08×10^{-17}	4.18×10^{-17}	4.22×10^{-17}	3.71×10^{-14}
10112	4.36×10^{-17}	4.20×10^{-17}	4.32×10^{-17}	4.29×10^{-17}	4.15×10^{-17}	4.30×10^{-17}	4.29×10^{-17}	4.05×10^{-14}

5.6.3 Orthogonality of Q

In this subsection we verify the orthogonality of matrix Q generated by our fault tolerant algorithm. As explained in Subsection 5.3.1, we have $H = Q^T A Q$. Q is an orthogonal matrix. We use the following residual to examine the orthogonality of Q:

$$r = \frac{\|QQ^\top - I\|_1}{N}.$$

I is the identity matrix, N is the matrix dimension. Table 5.3 shows the residuals from the non-fault tolerant MAGMA algorithm and residuals from our fault tolerant algorithm when one error occurs in different areas and different stages of the matrix. When the soft-error occurs in Area 1 and Area 2, all residuals are on the order of 10^{-17}, which is the same as the residuals from the MAGMA algorithm. When the soft-error occurs in Area 3, the residual is higher but still comparable to the residuals from MAGMA. So the orthogonality of Q is not damaged after the recovery from an error.

5.7 Conclusion

In this chapter we presented the design and analysis of a soft error resilient hybrid Hessenberg reduction algorithm, an algorithm capable of taking advantage of current

and future hybrid architectures to ensure data correctness during an entire two-sided factorization. This goal is achieved by an attentive combination of the strengths of ABFT and diskless checkpointing to maintain data redundancy during the factorization. From an algorithmic perspective, our algorithm detects the soft errors on-line and corrects them before they have the opportunity to propagate to the rest of the matrix data, minimizing the cost of the recovery process. In the case of a soft error, our algorithm carries out a reverse computation to roll the program data back to a consistent state and then correct the soft error. The overhead of our approach is very low since it mainly utilizes extra computation to detect and correct the error, and the amount of extra memory necessary for the checksum is minimal. The performance overhead of our fault tolerant algorithm compared to the non-fault tolerant MAGMA Hessenberg reduction reaches 1.45% when no errors occur, and reaches 3.29% when one error occurs. Another important capability of our fault tolerant algorithm is that it can detect and correct more than one consecutive error, making it a potential candidate for highly volatile environments. Moreover, the methodology highlighted in this chapter is generic enough to be applicable to the entire spectrum of two-sided factorizations, as well as other similar algorithms. This applicability is on our list of things to explore in the near term as we plan to provide soft error resilience for the rest of the hybrid two-sided factorizations in MAGMA.

Chapter 6

CPU-GPU Hybrid Bidiagonal Reduction With Soft Error Resilience

6.1 Introduction

Bidiagonalization of a general $M \times N$ matrix A is prerequisite to computing the singular value decomposition (SVD) of A. The execution time of numerical bidiagonalization on modern computers dominates the computation of SVD. Given an $M \times N$ real matrix A, the SVD decomposition computes $A = U\Sigma V^\top$ where U is an $M \times M$ orthogonal matrix, Σ is an $M \times N$ diagonal matrix, and $V^\top$ is an $N \times N$ orthogonal matrix. The diagonal entries of matrix Σ are called the singular values of A. The numerical SVD decomposition of a matrix is usually performed in two steps. In the first step, matrix A is reduced to bidiagonal form: $A = QBP^\top$ where Q is an $M \times M$ orthogonal matrix, P is an $N \times N$ orthogonal matrix, and B is an $M \times N$ bidiagonal matrix. In the second step, matrix B is further reduced to diagonal form. The implementations of the steps are slow because they contain a lot of matrix-vector multiplies (GEMV), which are Level 1 BLAS operations and have a low computation

intensity. As a result of the above reasons, it is time consuming to calculate either of the two stages.

Advances in Integrated Circuits (IC) manufacturing technology, described below, bring forward higher probability of soft errors in computer systems due to decreased feature size and increased complexity. A soft error is a temporary malfunction of a chip element which causes a change in the program state without any notification other than an incorrect result, but the chip element continues to function normally, and the change in program state is unnoticed by either the hardware or the software. Moore's Law states that the number of transistors on integrated circuits doubles every two years [72]. This increased transistor density on a unit silicon area provides every more prominent possibility for soft error. Also, higher transistor density requires smaller transistor feature size (the minimum size of a transistor or a wire on an IC), and causes increased heat dissipation as the circuit consumes higher power. Smaller transistors require lower voltage to operate at increasing frequencies, which makes it easier to change the transistor state unpredictably. High heat dissipation generates more thermal neutrons, which in turn cause more soft errors in the chip [94].

The computation needs to be protected so that there is no need to repeat the computation in the presence of soft errors. There are a few challenges in tolerating soft errors. First, it is difficult to detect them since a soft error changes the application state without the hardware or software noticing it. There is no permanent physical damage to the hardware, so the program proceeds normally in the presence of a soft error (assuming that the soft error does not alter the control logic). Second, it is difficult to pinpoint the error even when given the knowledge of the existence of an error. There are a large number of transistors involved in a single computation, so locating the error is analogous to finding a needle in a haystack. Third, suppose we know an error exists, and we know the exact location of the error, it is still difficult to restore the data to the correct value.

In this chapter, we propose an effective and efficient algorithm to detect, locate, and correct soft errors in the numerical bidiagonal reduction of a real matrix.

We employ the Algorithm Based Fault Tolerance (ABFT) technique and reverse computation to achieve fault tolerance. Our fault tolerant bidiagonal reduction algorithm is very efficient in that it introduces very low performance overhead compared with the non-fault tolerant counterpart. The overhead tends asymptotically to 0 as the matrix size scales up. We show the effectiveness and efficiency of our algorithm through an implementation based on the MAGMA library [88, 90]. Experiments show that our algorithm has a low overhead of 0.354% at matrix size about 10000. The overhead exhibits a decreasing trend as the matrix size increases.

The rest of this chapter is organized as follows: Section 6.2 reviews related work. Section 6.3 introduces the block bidiagonal reduction algorithm as implemented in the MAGMA project. Section 6.4 discusses the soft error propagation pattern. Section 6.5 explains our fault tolerant algorithm. Section 6.6 reports the experimental results. Finally, section 6.7 presents our conclusions.

6.2 Related Work

Research and reports about the existence and impact of soft errors on GPUs show that soft errors are a real problem for scientific applications [52, 84, 86, 77]. There are efforts to tolerate these errors using both software-based approaches [67, 66, 30] and hardware-based approaches [83, 87].

Du et al. [40] proposed a soft error resilient QR factorization algorithm using a post processing approach. In their scheme, the input matrix is encoded with two extra checksum columns. These two extra columns are maintained as the regular QR factorization proceeds. After the factorization has finished, the two extra columns are used to detect the existence of a soft error and locate the column index where the error occurred. The error is then projected to a rank-1 perturbation of the original input matrix. Then the correct factorization result is obtained using the QR update technique [47]. This post-processing scheme can successfully tolerate, at most, one soft error, no matter what point in time the error has occurred.

Kim et al. [60] designed a general scheme for fault tolerant matrix operations including matrix multiplication, Cholesky factorization, LU factorization, QR factorization, and Hessenberg reduction. This scheme tackles hard errors (process failures). The method uses a checksum to encode the input matrix, and the checksum is generated at certain intervals and serves as a checkpoint of the application state. In the case of a hard error, a roll back is performed to bring the program data back to the state at which the last checksum was generated.

The MAGMA project [88] is an effort to take advantage of the latest development of GPU accelerators to boost the performance of linear algebra operations. The project redesigned the block algorithms in LAPACK [1] to better suit GPU-enabled hybrid platforms. The methodology is called hybridization, by which computational tasks are split according to their characteristics and scheduled to the CPUs and GPUs accordingly. The `magma_dgebrd` routine in MAGMA implements the hybrid block bidiagonal reduction algorithm.

6.3 Block Bidiagonal Reduction in MAGMA

In order to understand our bidiagonal reduction algorithm with fault tolerant features, it is essential to understand the non-fault tolerant algorithm first. In this section, we describe the standard block bidiagonal reduction algorithm.

Suppose A is an $M \times N$ matrix, the block algorithm logically partitions A into $M \times nb$ block columns and $nb \times N$ block rows. The reduction is an iterative process, whereby every iteration reduces the leftmost nb matrix columns and the uppermost nb matrix rows into the bidiagonal form. In every iteration, the following operation is performed on the unreduced trailing matrix [34, 27]:

$$A^{(i+1)} = A^{(i)} - VY^{\top} - XU^{\top}$$

where $A^{(i)}$ is the unreduced trailing matrix at the beginning of the i-th iteration,

and $A^{(i+1)}$ is the resulting matrix of the i-th iteration (also the input for the $(i+1)$-th iteration). V is an $M \times nb$ matrix which consists of the Householder vectors to annihilate columns of A; this matrix is used to update the trailing matrix from the left. U is the matrix used to transform the input matrix from the right; this matrix contains the Householder vectors used to annihilate rows of A. Y and X are intermediate matrices, and each of them is generated through an iterative process. Denoting the k-th column of Y as y_k, the k-th column of X as x_k, the calculation of Y and X are given by [34, 27]:

$$y_{k+1} = \tau_{v_{k+1}}(A^{(i)^\top} v_{k+1} - Y_k V_k^\top v_{k+1} - U_k X_k^\top v_{k+1})$$
$$x_{k+1} = \tau_{u_{k+1}}(A^{(i)} u_{k+1} - X_k U_k^\top u_{k+1} - V_{k+1} Y_{k+1}^\top u_{k+1})$$

In MAGMA, the bidiagonal reduction routine for a double precision real matrix is `magma_dgehrd`. In every iteration, the algorithm performs the following operations:

1. Call `magma_dlahrd_gpu`. This call reduces the i-th block column and the i-th block row of A to bidiagonal form, and generates V, U, X, and Y. The two largest tasks in this routine (two GEMVs) are offloaded to the GPU.

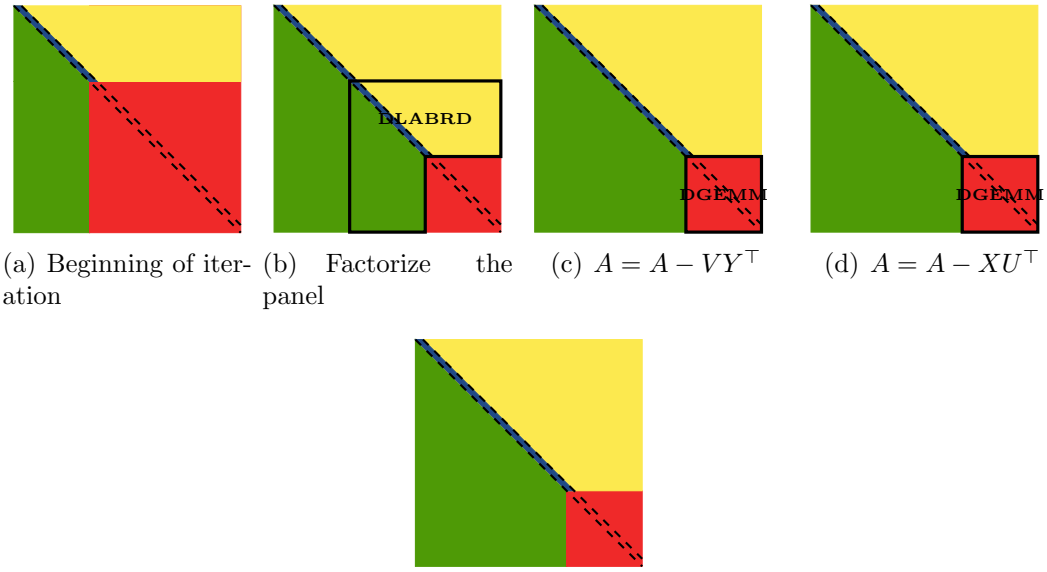
2. Call `magma_dgemm` to compute $A = A - VY^\top$. This matrix-matrix multiply and the following one are offloaded to the GPU.

3. Call `magma_dgemm` to compute $A = A - XU^\top$

Figure 6.1 shows one iteration of the `magma_dgehrd` routine.

6.4 Error Propagation

In this work we target soft errors specifically (as opposed to hard errors). A single bit flip is sufficient to completely invalidate the factorization result. Figure 6.2 shows the impact of a soft error in the course of the factorization. This example uses a 158×158

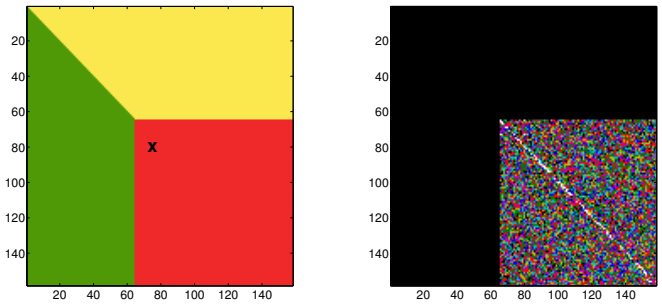


(a) Beginning of iteration (b) Factorize the panel (c) $A = A - VY^\top$ (d) $A = A - XU^\top$

(e) End of iteration

Figure 6.1: One iteration of `magma_dgebrd`

matrix with the block size $nb = 32$. The soft error occurs in the second iteration at location $(72, 79)$, which is marked by a cross in Figure 6.2(a). Figure 6.2(b) shows the heat map of the difference matrix between the correct factorization result and the factorization result affected by one soft error. Black color indicates that the difference is 0, and any other color indicates a difference of a magnitude proportionate to the color. We can observe that the rectangular matrix at the bottom right corner contains wrong values.



(a) Error location $(72, 79)$. (b) Partition

Figure 6.2: Propagation pattern of an error.

6.5 Fault Tolerant DGEBRD

In this section, we describe the fault tolerant bidiagonal reduction algorithm. The algorithm is inspired by the ABFT concept [54]. The basic idea is to add redundant information to the original data. A soft error means that some information in the original matrix is corrupted. After the detection of the soft error, the algorithm uses the redundancy to recover the corrupted information. The redundant information of the input matrix is provided by a checksum column and a checksum row. Algorithm 11 shows the details of our approach.

Algorithm 11 Fault Tolerant Hybrid Bidiagonal Reduction

1: Transfer matrix: A on the host $\rightarrow$ d_A on the GPU
2: Encode the input matrix, expand it with a checksum column and a checksum row.
3: **for** i from 1 to $\lceil \frac{N}{nb} \rceil$ **do**
4: Transfer the leftmost nb columns and uppermost nb rows of the trailing matrix to the host.
5: **FT_MAGMA_DLABRD_GPU**, return V, U, X and Y
6: Compute $X_{ce}, Y_{ce}, V_{ce}, U_{ce}$
7: **DGEMM**: $A_{fe} = A_{fe} - V_{ce} Y_{ce}^{\top}$
8: **DGEMM**: $A_{fe} = A_{fe} - X_{ce} U_{ce}^{\top}$
9: Compute $S_{re} = \sum A_{re}(i)$ and $S_{ce} = \sum A_{ce}(i)$
10: **if** $|S_{re} - S_{ce}| > threshold$ **then**
11: Reverse the last left update and right update.
$A_{fe} = A_{fe} + V_{ce} Y_{ce}^{\top}$
$A_{fe} = A_{fe} + X_{ce} U_{ce}^{\top}$
12: Correct the error
13: **end if**
14: **end for**

Algorithm 12 Locate(i, j, k)

1: **DGEMV**: $\hat{A}_{chk_r} = A_{trail} \cdot e$
2: IF
3: **DGEMV**: $\hat{A}_{chk_r} = A_{trail} \cdot e^{\top}$
4: IF

Algorithm 13 Recover(i, j, k)

1: Set $A(i, j)$ to 0.
2: Compute $S_{new} = \sum_{i=k+1}^{M} A(i, j)$
3: $\hat{A}(i, j) = A_{ce}(j) - S_{new}$

6.5.1 Data Redundancy

The algorithm first encodes the input matrix with both row checksums and column checksums. The row checksums form a column vector which is appended to the right of the matrix, the column checksums form a row vector which is appended to the bottom of the input matrix. This task is accomplished in line 2. The algorithm enters the main loop in line 3. In every iteration, the next panel is transferred to the CPU to be factorized there (in Line 5). The original `magma_dlabrd_gpu` routine only computes part of X and part of Y. Assuming the trailing matrix is of size $m \times n$, we only need the lower $n \times nb$ part of Y and the lower $m \times nb$ part of X are needed to update the trailing matrix. In our fault tolerant algorithm, we need the entire X and Y to calculate their respective column checksums, so we modified the `magma_dlabrd_gpu` routine to compute the complete X and Y. The new routine is named `ft_magma_dlabrd_gpu`. Line 7 and line 8 update the trailing matrix. The row checksums and column checksums are also updated together with the trailing matrix. After the update, the row checksums remain to be the row checksums of their corresponding rows. The column checksums remain to be the column checksums of their corresponding columns. In other words, the checksum relationship is preserved throughout the algorithm.

6.5.2 Error Detection

Line 9 and line 10 carry out error detection. Error detection is achieved by comparing the sum of the row checksums of the trailing matrix and the sum of the column checksums of the trailing matrix. Because both checksum vectors protect the same matrix data (the trailing matrix), their sums should be equal to each other. If the

difference is higher than a certain threshold, we consider an error has occurred. The comparison of the two sums is performed in line 10.

6.5.3 Error Location and Correction

If an error is detected at line 10, the algorithm initiates the procedure to locate and correct the error. To achieve this, the algorithm first performs a reverse update on the trailing matrix. This is accomplished in line 11. The reverse update brings the trailing matrix back to the state at the beginning of the erroneous iteration. At this point, the error only exists in one matrix entry, the contamination to other matrix entries is reversed, and now we have the correct row checksums and column checksums. The error location works as follows. We compute the new row checksums and column checksums of the actual trailing matrix, and these new checksums will encode the erroneous value. Moreover, there will be exactly one row checksum which differs from its corresponding old row checksum, and there will be exactly one column checksum which differs from its corresponding column checksum. The row index and column index of the error can be identified by comparing the new checksums and the old checksums.

Once the error location (i, j) has been determined, we can use the row checksums and the column checksums as devices to recover the lost matrix element. First we set $A(i, j)$ to zero, then we compute the checksum chk_r for the i-th row. The lost matrix element can be recovered by $A(i, j) = chk_r - old_chk_r$. old_chk_r is the row checksum of the i-th row which the algorithm maintains since the beginning of the factorization. The algorithm resumes its normal operations after the recovery. It continues to detect, locate, and correct errors in subsequent iteration until the factorization completes.

6.5.4 Multiple Concurrent Errors

In previous subsections, we only considered the case in which only one soft error happens in an iteration. In fact the fault tolerant algorithm can deal with more than

one soft errors in one iteration. When more than one soft error occurs, the entire trailing matrix will be contaminated as in the one-error case, so the existence of errors can always be detected. Similar to the analysis by Huang et al. [54], when the faulty elements form a rectangle, these four errors cannot be located. Other than such a situation, multiple errors can be located and then corrected.

6.5.5 Range of Application

The fault tolerant algorithm stated above is for bit-flips in the data matrix. The fault tolerant algorithm does not deal with soft errors in the control logic. It does not consider persistent errors either. Persistent errors are usually caused by malfunctioning hardware, this type of errors are outside of the range range covered by this work.

6.6 Experiment Results

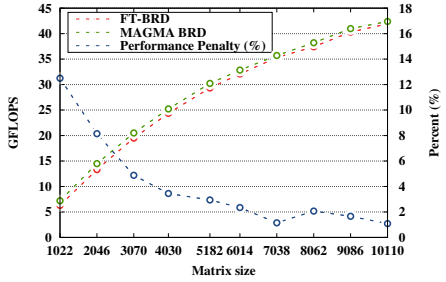
In this section, we present performance results of our fault tolerant bidiagonal reduction algorithm.

The test platform we use is a machine at the University of Tennessee. This machine has an Intel Xeon E5-2670 processor with a clock frequency of 2.6 GHz. It features an NVIDIA Tesla K20c GPU (known also as Kepler) with the clock frequency of the GPU at 705.5 MHz and the on-board memory of the GPU of 4799.6 MB. Te machine has 62 GB of main memory. The operating system is Red Hat 4.4.6-4 and the compiler is GCC 4.4.6 and NVCC 5.0 V0.2.1221.

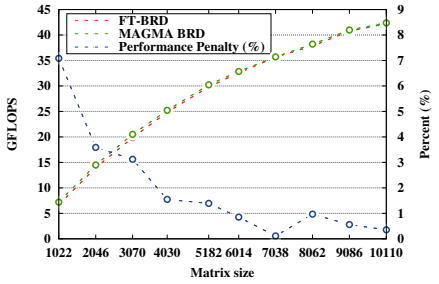
Figure 6.3 shows the comparison of the performance of the fault tolerant bidiagonal reduction and the performance of the MAGMA bidiagonal reduction. Figure 6.3(a) shows the performance comparison when the fault tolerant bidiagonal reduction suffers from one soft error. The error is injected in the third iteration in the panel area. This is nearly the worst case scenario. The earlier the error occurs, the higher

the cost of locating and correcting the error. The reason is that if the error occurs in the early iterations of the factorization, we need to reverse the update of the trailing matrix once we detect an error, and the trailing matrix is large in early iterations. To locate the error, we need two DGEMV operations on the trailing matrix. In early iterations the large trailing matrix also incurs higher costs in these two DGEMVs.

Figure 6.3(b) shows the performance comparison when the FT bidiagonal reduction does not experience any errors. We can see that the performance overhead also drops when the matrix size increases.



(a) With error



(b) Without error

Figure 6.3: Performance of the FT-BRD

6.7 Conclusion

In this chapter, we showed a design, implementation, and a performance evaluation of a hybrid bidiagonal reduction algorithm based on the MAGMA framework equipped with fault tolerant features. Our fault tolerant bidiagonal reduction algorithm employs reverse computation and algorithm-based fault tolerance to detect, locate,

and correct soft errors in the bidiagonal reduction on CPU-GPU hybrid architectures. Experimental results show that the performance overhead of our fault tolerant algorithm is very low when the matrix size is small, and the performance overhead as fraction of the overall computation time continues to drop as the matrix size increases. At matrix sizes of about 10000, the overhead decreases to 1.085% when one soft error occurs, and to 0.354% when no errors occur.

Chapter 7

Conclusions and Future Directions

7.1 Conclusion

This research studied the algorithmic properties of two-sided dense matrix factorizations — namely the Hessenberg reduction, the tridiagonal reduction, the bidiagonal reduction — and designed algorithm based fault tolerance algorithms for these dense matrix reduction algorithms to provide resilience against hard errors and soft errors. Theoretical analysis and experimental results both prove that our fault tolerant algorithms are able to protect the factorizations against errors, are low-cost and are scalable. We also showed that our fault tolerant algorithms do not degrade the numerical stability of the original non-fault tolerant algorithms.

The major difficulty in designing fault tolerant algorithms for two-sided dense matrix factorizations is finding a way to properly maintain the checksums. In the one-sided dense matrix factorization algorithms, the ABFT version of those algorithms only needs to extend the normal trailing matrix update operations to include the checksums. Then the checksums will be automatically updated to the correct checksums of the updated trailing matrix. Whereas in two-sided dense matrix factorizations, the trailing matrix is updated from both the right side and the left side. The left side update destroys the checksums that are appended to the right side

of the original matrix data. In order to perform the left side update to the trailing matrix without destroying the checksums, we calculate a set of column checksums for the block column resulting from the panel factorization. On distributed memory machines, this checksum calculation involves an MPI reduction operation on one process column in the process grid, and is carried out once in every iterative cycle of the algorithm. Since MPI reduction operations are costly, a big portion of the time overhead of our ABFT algorithm is spent on this repetitive checksum calculation. On shared memory machines, we modified the panel factorization routine so that the column checksum calculation is fused with the panel factorization. No observable overhead is shown due to the checksum calculation for the block column resulting from the panel factorization.

For the fault tolerant parallel tridiagonal reduction algorithm, we take advantage of the symmetry of the matrix to retrieve data blocks from the lower triangular part of the matrix when needed. In the ScaLAPACK implementation of the tridiagonal reduction algorithm, only the lower triangular part of the matrix is accessed and updated. This is done to save floating point operations. But in order to be able to update the checksums in our ABFT algorithm, we choose to let the checksum blocks on the right side of the matrix encode the full matrix instead of only encoding the lower triangular matrix. When recovering from a failure, we need all matrix blocks on the block row where the failure strikes, but the matrix blocks residing in the upper triangular part are invalid. Since the matrix is symmetric, we can find the locations of the corresponding data blocks in the lower triangular part and retrieve data from there.

For the fault tolerant algorithms on CPU-GPU hybrid platforms, we use a row of column checksums and a column of row checksums to protect the matrix. At the end of each iterative cycle of the algorithm, we compute the sum of the row of checksums and the sum of the column of checksums. By comparing the equality of the two sums, we can detect if errors have occurred. If errors are detected, we reverse the trailing matrix updates performed in this iterative cycle, and the state of the matrix is rolled

back to the end of the previous iterative cycle. Now, the errors can be located and corrected using the row of checksums and the column of checksums.

7.2 Future Directions

This research addresses hard errors in the parallel Hessenberg reduction algorithm and the parallel tridiagonal reduction algorithm. This work also addresses soft errors in the Hessenberg reduction algorithm and the bidiagonal reduction algorithms on CPU-GPU hybrid algorithms. If supercomputer development continues according to current trends, then future exascale machines will be composed of compute nodes with both general purpose CPUs and special purpose accelerators. That is, each compute node will have a number of CPUs and a number of accelerators. Fault tolerant algorithms for dense matrix factorizations which utilize both CPUs and accelerators are a necessary future research topic.

In this work, every fault tolerant algorithm deals with a single type of error, either hard errors or soft errors. When errors strike, there is no guarantee that only one type of error will occur. So our future plan includes integrating protection against both hard errors and soft errors for two sided dense matrix factorization algorithms running on distributed memory machines.

In our fault tolerant algorithms, the checksums are generated using a simple addition operation. When the matrix values are too big, the checksums may overflow. When the difference between the values of matrix elements is large, the numerical stability of the factorization algorithms may be severely degraded. A more proper approach is to examine the matrix elements before generating the checksums. Choosing a properly crafted generation matrix to generate the checksums will obtain better numerical stability after the fault tolerant algorithms recover from errors. This will be one other direction of our future work.

Bibliography

[1] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. SIAM, Philadelphia, PA, Third edition, 1999. 18, 49, 76, 101

[2] ARM Cortex-R7 MPCore Technical Reference Manual, Retrieved in September 2015. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0458c/DDI0458C_cortex_r7_r0p1_trm.pdf. 11

[3] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *Dependable and Secure Computing, IEEE Transactions on*, 1(1):11–33, Jan 2004. 7

[4] R.C. Baumann. Radiation-induced soft errors in advanced semiconductor technologies. *Device and Materials Reliability, IEEE Transactions on*, 5(3):305–316, 2005. 72

[5] Leonardo Bautista-Gomez, Seiji Tsuboi, Dimitri Komatitsch, Franck Cappello, Naoya Maruyama, and Satoshi Matsuoka. Fti: High performance fault tolerance interface for hybrid systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 32:1–32:32, New York, NY, USA, 2011. ACM. 14

[6] Michael Berry and Murray Browne. *Understanding Search Engines: Mathematical Modeling and Text Retrieval*. SIAM, Philadelphia, Second edition, 2005. 17

[7] Christian H. Bischof and Charles Van Loan. The WY Representation for Products of Householder Matrices. In *Parallel Processing for Scientific Computing*, pages 2–13, 1985. 24

[8] L. S. Blackford, J. Choi, A. Cleary, E. D'Azeuedo, J. Demmel, I. Dhillon, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. SIAM, Philadelphia, PA, 1997. 18, 49, 50, 53

[9] W. Bland, P. Du, A. Bouteiller, T. Herault, G. Bosilca, and J. Dongarra. Extending the scope of the checkpoint-on-failure protocol for forward recovery in standard MPI. Technical Report UT-CS-12-702, University of Tennessee, 2012. 22

[10] Wesley Bland, Peng Du, Aurelien Bouteiller, Thomas Herault, George Bosilca, and Jack Dongarra. A checkpoint-on-failure protocol for algorithm-based recovery in standard MPI. In *Proceedings of the 18th international conference on Parallel Processing*, Euro-Par'12, pages 477–488, 2012. 22

[11] Daniel Boley, Gene H. Golub, Samy Makar, Nirmal Saxena, and Edward J. McCluskey. Floating point fault tolerance with backward error assertions. *IEEE Transactions on Computers*, 44(2):302–311, February 1995. 43

[12] S. Börm and C. Mehl. *Numerical Methods for Eigenvalue Problems*. De Gruyter Textbook. De Gruyter, 2012. 49

[13] George Bosilca, Aurélien Bouteiller, Élisabeth Brunet, Franck Cappello, Jack Dongarra, Amina Guermouche, Thomas Hérault, Yves Robert, Frédéric Vivien, and Dounia Zaidouni. Unified Model for Assessing Checkpointing Protocols at Extreme-Scale. Technical Report RR-7950, INRIA, October 2012. 20

[14] George Bosilca, Rémi Delmas, Jack Dongarra, and Julien Langou. Algorithm-based fault tolerance applied to high performance computing. *J. Parallel Distrib. Comput.*, 69(4):410–416, April 2009. 21

[15] Marin Bougeret, Henri Casanova, Yves Robert, Frédéric Vivien, and Dounia Zaidouni. Using Group Replication for Resilience on Exascale Systems. Technical Report RR-7876, INRIA, February 2012. 20

[16] K. Braman, R. Byers, and R. Mathias. The multishift QR algorithm. ii. aggressive early deflation. *SIAM J. Matrix Anal. Appl.*, 23:948–973, 2002. 18

[17] Sergey Brin and Lawrence Page. The antaomy of a large-scale hypertextual Web search engine. *Computer Networks and ISDN Systems*, 33:107–17, 1998. Also available online at http://infolab.stanford.edu/pub/papers/google.pdf. 17

[18] Greg Bronevetsky and Bronis de Supinski. Soft error vulnerability of iterative linear algebra methods. In *Proceedings of the 22nd annual international conference on Supercomputing*, ICS '08, pages 155–164, New York, NY, USA, 2008. ACM. 74

[19] Kurt Bryan and Tanya Leise. The $25,000,000,000 eigenvector. the linear algebra behind google. *SIAM Review*, 48(3):569–81, 2006. Also avaiable at http://www.rose-hulman.edu/~bryan/google.html. 17

[20] Chongxiao Cao, George Bosilca, Thomas Herault, and Jack Dongarra. Design for a Soft Error Resilient Dynamic Task-based Runtime. In *29th IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, Hyderabad, India, 05/2015 2015. IEEE, IEEE. 75

[21] Henri Casanova, Yves Robert, Frédéric Vivien, and Dounia Zaidouni. Combining Process Replication and Checkpointing for Resilience on Exascale Systems. Technical Report RR-7951, INRIA, May 2012. 20

[22] Anthony M. Castaldo, R. Clint Whaley, and Anthony T. Chronopoulos. Reducing floating point error in dot product using the superblock family of algorithms. *SIAM J. Sci. Comput.*, 31(2):1156–1174, 2008. 43

[23] Anthony M. Castaldo, R. Clint Whaley, and Anthony T. Chronopoulos. Reducing floating point error in dot product using the superblock family of algorithms. *SIAM J. Scientific Computing*, 31(2):1156–1174, 2008. 95

[24] F. Chatelin, M. Ahues, and W. Lederman. *Eigenvalues of Matrices: Revised Edition*. Classics in Applied Mathematics. Society for Industrial and Applied Mathematics, 2012. 49

[25] Chien-Yi Chen and Jacob A. Abraham. Fault-tolerant systems for the computation of eigenvalues and singular values. In *Advanced Algorithms and Architectures for Signal Processing I*, volume 0696, pages 228–237, April 1986. 15, 51, 74, 75

[26] Zizhong Chen. *Scalable techniques for fault tolerant high performance computing*. PhD thesis, University of Tennessee, Knoxville, TN, USA, 2006. 15, 21

[27] Jaeyoung Choi, Jack J. Dongarra, and David W. Walker. The design of a parallel dense linear algebra software library: Reduction to hessenberg, tridiagonal, and bidiagonal form. *Numerical Algorithms*, 10(2):379–399, 1995. 101, 102

[28] Teresa Davies, Christer Karlsson, Hui Liu, Chong Ding, and Zizhong Chen. High Performance Linpack Benchmark: A Fault Tolerant Implementation Without Checkpointing. In *Proceedings of the International Conference on Supercomputing*, ICS '11, pages 162–171, New York, NY, USA, 2011. ACM. 15, 22

[29] Timothy Dell. A White Paper on the Benefits of Chipkill-Correct ECC for PC Server Main Memory, Retrieved in September 2015. http://www.ece.umd.edu/courses/enee759h.S2003/references/ibm_chipkill.pdf. 10

[30] Martin Dimitrov, Mike Mantor, and Huiyang Zhou. Understanding software approaches for gpgpu reliability. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-2, pages 94–104, New York, NY, USA, 2009. ACM. 100

[31] Jack Dongarra, Pete Beckman, and Terry Moore. The international Exascale software project roadmap. *Int. J. High Perform. Comput. Appl.*, 25(1):3–60, February 2011. 1, 17

[32] Jack J. Dongarra, Piotr Luszczek, and Antoine Petitet. The LINPACK benchmark: Past, present, and future. *Concurrency and Computation: Practice and Experience*, 15:1–18, 2003. 22

[33] Jack J. Dongarra, Piotr Luszczek, and Antoine Petitet. The LINPACK benchmark: Past, present, and future. *Concurrency and Computation: Practice and Experience*, 15(9):803–820, August 10 2003. DOI: 10.1002/cpe.728. 74

[34] 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, 1989. 101, 102

[35] Jack J. Dongarra and Robert A. van de Geijn. Reduction to condensed form for the eigenvalue problem on distributed memory architectures. *Parallel Computing*, 18(9):973–982, 1992. 24

[36] Peng Du, Aurelien Bouteiller, George Bosilca, Thomas Herault, and Jack Dongarra. Algorithm-based fault tolerance for dense matrix factorizations. *SIGPLAN Not.*, 47(8):225–234, February 2012. 15, 21, 29, 31, 52

[37] Peng Du, P. Luszczek, and J. Dongarra. High performance dense linear system solver with soft error resilience. In *Cluster Computing (CLUSTER), 2011 IEEE International Conference on*, pages 272–280, 2011. 73, 74

[38] Peng Du, Piotr Luszczek, and Jack Dongarra. High performance dense linear system solver with resilience to multiple soft errors. *Procedia Computer Science*, 9(0):216 – 225, 2012. 73, 74

[39] Peng Du, Piotr Luszczek, Stan Tomov, and Jack Dongarra. Soft error resilient QR factorization for hybrid system with GPGPU. In *Proceedings of the second*

workshop on Scalable algorithms for large-scale systems, ScalA '11, pages 11–14, New York, NY, USA, 2011. ACM. 73, 74

[40] Peng Du, Piotr Luszczek, Stan Tomov, and Jack Dongarra. Soft error resilient QR factorization for hybrid system with GPGPU. *Journal of Computational Science*, 2013. 100

[41] Jason Duell, Paul Hargrove, and Eric Roman. The design and implementation of berkeley lab's linux checkpoint/restart, December 2002. 48

[42] 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, September 2002. 12

[43] Christian Engelmann. Resilience challenges and solutions for extreme-scale supercomputing. Invited talk at the Technical University of Dresden, Dresden, Germany, September 3, 2013. 47

[44] D. Fiala. Detection and correction of silent data corruption for large-scale high-performance computing. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pages 2069–2072, May 2011. 12

[45] J. G. F. Francis. The QR transformation a unitary analogue to the LR transformation. I. *Comput. J.*, 4:265–271, 1961. 18

[46] J. G. F. Francis. The QR transformation II. *Comput. J.*, 4:332–345, 1962. 18

[47] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. The John Hopkins University Press, 4th edition, December 27 2012. ISBN-10: 1421407949, ISBN-13: 978-1421407944. 18, 75, 100

[48] Leonardo Arturo Bautista Gomez, Naoya Maruyama, Franck Cappello, and Satoshi Matsuoka. Distributed Diskless Checkpoint for Large Scale Systems.

In *Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on*, pages 63–72, 2010. 20

[49] R. Granat, B. Kågström, and D. Kressner. A novel parallel QR algorithm for hybrid distributed memory HPC systems. *SIAM J. Sci. Comput.*, 32:2345–2378, 2010. 18

[50] R. Granat, B. Kågström, D. Kressner, and M. Shao. Parallel library software for the multishift QR algorithm with aggressive early deflation. Technical Report UMINF-12.06, Dept. of Computing Science, Umeøa University, Sweden, 2012. 18

[51] D. Hakkarinen and Zizhong Chen. Algorithmic Cholesky Factorization Fault Recovery. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–10, April 2010. 21, 52

[52] I.S. Haque and V.S. Pande. Hard Data on Soft Errors: A Large-Scale Assessment of Real-World Error Rates in GPGPU. In *Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on*, pages 691–696, 2010. 72, 100

[53] M. Y. Hsiao. A class of optimal minimum odd-weight-column sec-ded codes. *IBM J. Res. Dev.*, 14(4):395–401, July 1970. 10

[54] Kuang-Hua Huang and J. A. Abraham. Algorithm-based fault tolerance for matrix operations. *IEEE Trans. Comput.*, 33(6):518–528, June 1984. 14, 21, 74, 104, 107

[55] 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. 48, 51

[56] B. Jacob, S. Ng, and D. Wang. *Memory Systems: Cache, DRAM, Disk*. Elsevier Science, 2010. 72

[57] 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 on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 88:1–88:11, New York, NY, USA, 2013. ACM. 75, 94

[58] R. Mathias K. Braman, R. Byers. The multishift QR algorithm. I. maintaining well-focused shifts and level 3 performance. *SIAM J. Matrix Anal. Appl.*, 23:929–947, 2002. 18

[59] B. Kågström, D. Kressner, and M. Shao. On aggressive early deflation in parallel variants of the QR algorithm. In *PARA 2010, Applied Parallel and Scientific Computing, LNCS*, volume 7134, pages 1–10. Springer, 2012. 18

[60] Y. Kim, J. S. Plank, and J. Dongarra. Fault Tolerant Matrix Operations Using Checksum and Reverse Computation. In *6th Symposium on the Frontiers of Massively Parallel Computation*, pages 70–77, Annapolis, MD, October 1996. 21, 51, 74, 101

[61] J. Langou, Z. Chen, G. Bosilca, and J. Dongarra. Recovery patterns for iterative methods in a parallel unstable environment. *SIAM J. Sci. Comput.*, 30(1):102–116, November 2007. 20

[62] Amy Langville and Carl Meyer. *Google's PageRank and Beyond: The Science of Search Engine Rankings*. Princeton University Press, 2006. 17

[63] Charng-Da Lu. *Scalable Diskless Checkpointing for Large Parallel Systems*. PhD thesis, University of Illinois, Urbana, Illinois, USA, 2005. 20

[64] F. T. Luk and H. Park. Fault-tolerant matrix triangularizations on systolic arrays. *IEEE Trans. Comput.*, 37(11):1434–1438, November 1988. 21, 43, 74

[65] Franklin T. Luk and Haesun Park. An analysis of algorithm-based fault tolerance techniques. *J. Parallel Distrib. Comput.*, 5(2):172–184, April 1988. 74

[66] N. Maruyama, A. Nukada, and S. Matsuoka. A high-performance fault-tolerant software framework for memory on commodity gpus. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12, 2010. 100

[67] Naoya Maruyama, Akira Nukada, and Satoshi Matsuoka. Software-based ECC for GPUs. In *SAAHPC'09*, July 2009. 100

[68] Esteban Meneses. Clustering Parallel Applications to Enhance Message Logging Protocol. https://wiki.ncsa.illinois.edu/download/attachments/17630761/INRIA-UIUC-WS4-emenese.pdf?version=1&modificationDate=1290466786000. 17

[69] Hans Meuer, Jack Dongarra, Erich Strohmaier, and Horst Simon. The TOP500 List. http://www.top500.org/. 1

[70] S.E. Michalak, K.W. Harris, N.W. Hengartner, B.E. Takala, and S.A. Wender. 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. 72

[71] Adam Moody. *The Scalable Checkpoint / Restart (SCR) Library Version 1.1-6 User Manual*. Livermore Computing Center, Lawrence Livermore National Laboratory, Livermore, CA, April 2010. 14

[72] G. E. Moore. Cramming More Components onto Integrated Circuits. *Electronics*, 38(8):114–117, April 1965. 99

[73] Beresford N. Parlett. *The Symmetric Eigenvalue Problem*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1998. 54

[74] A. Petitet, C. Whaley, J. Dongarra, and A. Cleary. HPL - a Portable Implementation of the High-Performance Linpack Benchmark for Distributed-memory Computers, September 2008. http://www.netlib.org/benchmark/hpl/. 22

[75] J. S. Plank, K. Li, and M. A. Puening. Diskless checkpointing. *IEEE Trans. Parallel Distrib. Syst.*, 9(10):972–986, October 1998. 20, 75

[76] Gregorio Quintana-Ortí and Robert van de Geijn. Improving the performance of reduction to Hessenberg form. *ACM Trans. Math. Softw.*, 32(2):180–194, June 2006. 77

[77] P. Rech, C. Aguiar, R. Ferreira, M. Silvestri, A. Griffoni, C. Frost, and L. Carro. Neutron-induced soft errors in graphic processing units. In *Radiation Effects Data Workshop (REDW), 2012 IEEE*, pages 1–6, 2012. 100

[78] Edwin D. Reilly. *Milestones in Computer Science and Information Technology*. Greenwood Publishing Group Inc., Westport, CT, USA, 2003. 9

[79] Robert Schreiber and Charles Van Loan. A storage efficient WY representation for products of householder transformations. *SIAM Journal on Scientific and Statistical Computing*, 10, 1989. 24, 77

[80] Bianca Schroeder and Garth A. Gibson. Understanding failures in petascale computers. *Journal of Physics: Conference Series*, 78, 2007. 1, 17, 47

[81] Manu Shantharam, Sowmyalatha Srinivasmurthy, and Padma Raghavan. Characterizing the impact of soft errors on iterative methods in scientific computing. In *Proceedings of the international conference on Supercomputing*, ICS '11, pages 152–161, New York, NY, USA, 2011. ACM. 75

[82] Manu Shantharam, Sowmyalatha Srinivasmurthy, and Padma Raghavan. Fault tolerant preconditioned conjugate gradient for sparse linear system solution. In

Proceedings of the 26th ACM international conference on Supercomputing, ICS '12, pages 69–78, New York, NY, USA, 2012. ACM. 75

[83] Jeremy W. Sheaffer, David P. Luebke, and Kevin Skadron. A hardware redundancy and recovery mechanism for reliable scientific computation on graphics processors. In *Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, GH '07, pages 55–64, 2007. 100

[84] G. Shi, J. Enos, M. Showerman, and V. Kindratenko. On testing gpu memory for hard and soft errors. In *Proc. Symposium on Application Accelerators in High-Performance Computing*, 2009. 100

[85] G. W. Stewart. *Matrix Algorithms, Volume II: Eigensystems*. SIAM: Society for Industrial and Applied Mathematics, First edition, August 2001. 2, 18, 24, 75

[86] Jingweijia Tan, Nilanjan Goswami, Tao Li, and Xin Fu. Analyzing soft-error vulnerability on GPGPU microarchitecture. In *Proceedings of the 2011 IEEE International Symposium on Workload Characterization*, IISWC '11, pages 226–235, Washington, DC, USA, 2011. IEEE Computer Society. 100

[87] Jingweijia Tan, Zhi Li, and Xin Fu. Cost-effective soft-error protection for sram-based structures in gpgpus. In *Proceedings of the ACM International Conference on Computing Frontiers*, page 29. ACM, 2013. 100

[88] S. Tomov, R. Nath, P. Du, and J. Dongarra. *MAGMA Users' Guide*. ICL, UTK, November 2009. 76, 100, 101

[89] Stanimire Tomov and Jack Dongarra. Accelerating the reduction to upper Hessenberg form through hybrid GPU-based computing. Technical Report UT-CS-09-642, University of Tennessee Knoxville, 2009. 78

[90] Stanimire Tomov, Jack Dongarra, and Marc Baboulin. Towards dense linear algebra for hybrid GPU accelerated manycore systems. *Parallel Computing*, 36(5-6):232–240, 2010. 100

[91] U. von Luxburg. A tutorial on spectral clustering. *Stat. Comput.*, 17:395–416, 2007. 17

[92] J. H. Wilkinson. *The Algebraic Eigenvalue Problem*. Oxford University Press, Inc., New York, NY, USA, 1988. ISBN-10: 0198534183, ISBN-13: 978-0198534181. 42

[93] J. H. Wilkinson. *Rounding Errors in Algebraic Processes*. Dover, New York, 1994. 44

[94] J.D. Wilkinson, C. Bounds, T. Brown, B.J. Gerbi, and J. Peltier. Cancer-radiotherapy equipment as a cause of soft errors in electronic equipment. *Device and Materials Reliability, IEEE Transactions on*, 5(3):449–451, 2005. 99

[95] Erlin Yao, Rui Wang, Mingyu Chen, Guangming Tan, and Ninghui Sun. A Case Study of Designing Efficient Algorithm-based Fault Tolerant Application for Exascale Parallelism. In *Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 438–448, May 2012. 15, 22

[96] J. F. Ziegler and W. A. Lanford. Effect of Cosmic Rays on Computer Memories. *Science*, 206(4420):776–788, 1979. 71

Vita

Yulu Jia was born in Baoding, Bebei Province, P.R. China. He graduated from Fuping High School in 1999. He obtained his Bachelor's degree in Mathematics and Applied Mathematics and his Master's degree in Applied Computer Technologies in China University of Geosciences, Beijing, P.R. China in the year of 2006 and 2009 respectively. In 2009 he enrolled in the Ph.D. program in Computer Science at the University of Tennessee, Knoxville. He joined the Innovative Computing Laboratory as a graduate research assistant in January, 2011. He is currently conducting research under the supervision of Professor Jack Dongarra. Yulu Jia is expected to receive his Doctor of Philosophy degree in December, 2015.