8-2012

Hard and Soft Error Resilience for One-sided Dense Linear Algebra Algorithms

Peng Du
rick.peng.du@gmail.com

To the Graduate Council:

I am submitting herewith a dissertation written by Peng Du entitled "Hard and Soft Error Resilience for One-sided Dense Linear Algebra Algorithms." I have examined the final electronic copy of this dissertation for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Doctor of Philosophy, with a major in Computer Science.

Jack Dongarra, Major Professor

We have read this dissertation and recommend its acceptance:

Michael Berry, James Plank, Xiaobing Feng, Jack Dongarra

Accepted for the Council:
Dixie L. Thompson

Vice Provost and Dean of the Graduate School

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

8-2012

Hard and Soft Error Resilience for One-sided Dense Linear Algebra Algorithms

Peng Du
rick.peng.du@gmail.com

To the Graduate Council:

I am submitting herewith a dissertation written by Peng Du entitled "Hard and Soft Error Resilience for One-sided Dense Linear Algebra Algorithms." I have examined the final electronic copy of this dissertation for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Doctor of Philosophy, with a major in Computer Science.

Jack Dongarra, Major Professor

We have read this dissertation and recommend its acceptance:

Michael Berry, James Plank, Xiaobing Feng, Jack Dongarra

Accepted for the Council:
Carolyn R. Hodges

Vice Provost and Dean of the Graduate School

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

Hard and Soft Error Resilience for One-sided Dense Linear Algebra Algorithms

A Thesis Presented for

The Doctor of Philosophy

Degree

The University of Tennessee, Knoxville

Peng Du

August 2012

This dissertation is dedicated to my dearest parents, Wenjian Du and Xinzhi Liu.

Acknowledgements

I would like to express my deepest gratitude to my advisor, Dr. Jack Dongarra, for his guidance, motivation, and support during my graduate study at the Innovative Computing Laboratory (ICL). Dr. Dongarra has provided me extensive support, and valuable discussions throughout the process of this research. I am grateful to Dr. Dongarra for the generous financial support for this research and ample opportunities to share the research ideas in various conferences and workshops.

In addition, I would like to thank Dr. Michael Berry, Dr. James Plank, and Dr. Xiaobing Feng for agreeing to serve on my graduate committee. I greatly appreciate their time and invaluable advice to this dissertation.

I would also like to express my appreciation to Dr. Piotr Luszczek, Dr. Stanimire Tomov, Dr. Julien Langou and, especially, to Dr. George Bosilca, and the whole MPI équipe at ICL. This research could not have been accomplished without their insightful suggestions, patience, encouragement, and belief. I want to thank Sam Crawford for the help with the writing and review. And special thank also goes to my friend Teng Ma for all the challenging and inspiring discussion. This friendship help me sail through those tough times.

Last but not least, I am deeply indebted to my family and friends. I owe thanks to my parents, Wenjian Du and Xinzhi Liu for their love, sacrifices, and encouragement that are crucial to the completion of my studies. I am grateful to the friendship with Erika Parsons and Matthew Parson, Zizhong Chen, Fengguang Song, Wesley Bland, Rick Weber, and Yuanlei Zhang for the friendship and support over years.

"Because it's there" - George Mallory

Abstract

Dense matrix factorizations, such as LU, Cholesky and QR, are widely used by scientific applications that require solving systems of linear equations, eigenvalues and linear least squares problems. Such computations are normally carried out on supercomputers, whose ever-growing scale induces a fast decline of the Mean Time To Failure (MTTF). This dissertation develops fault tolerance algorithms for one-sided dense matrix factorizations, which handles Both hard and soft errors.

For hard errors, we propose methods based on diskless checkpointing and Algorithm Based Fault Tolerance (ABFT) to provide full matrix protection, including the left and right factor that are normally seen in dense matrix factorizations. A horizontal parallel diskless checkpointing scheme is devised to maintain the checkpoint data with scalable performance and low space overhead, while the ABFT checksum that is generated before the factorization constantly updates itself by the factorization operations to protect the right factor. In addition, without an available fault tolerant MPI supporting environment, we have also integrated the Checkpoint-on-Failure(CoF) mechanism into one-sided dense linear operations such as QR factorization to recover the running stack of the failed MPI process.

Soft error is more challenging because of the silent data corruption, which leads to a large area of erroneous data due to error propagation. Full matrix protection is developed where the left factor is protected by column-wise local diskless checkpointing, and the right factor is protected by a combination of a floating point weighted checksum scheme and soft error modeling technique. To allow practical use

on large scale system, we have also developed a complexity reduction scheme such that correct computing results can be recovered with low performance overhead.

Experiment results on large scale cluster system and multicore+GPGPU hybrid system have confirmed that our hard and soft error fault tolerance algorithms exhibit the expected error correcting capability, low space and performance overhead and compatibility with double precision floating point operation.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Today's high performance computers have paced into Petaflops realm, through the increase of system scale. The number of system components, such as CPU cores, memory, networking, and storage grow considerably. One of the most powerful Petaflops scale machines, Kraken [2], from National Institute for Computational Sciences and University of Tennessee, harnessed as many as 112,800 cores to reach its peak performance of 1.17 Petaflops to rank No.11 on the November 2011 Top500 list. With the increase of system scale and chip density, the reliability and availability of such systems has declined. It has been shown that, under specific circumstances, adding computing units might hamper applications completion time, as a larger node count implies a higher probability of reliability issues. This directly translates into a lower efficiency of the machine, which equates to a lower scientific throughput [128]. It is estimated that the MTTF of High Performance Computing (HPC) systems might drop to about one hour in the near future [28]. Without a drastic change at the algorithmic level, such a failure rate will certainly prevent capability applications from progressing. It is utterly important that effective fault tolerances technique is developed such that application can have the resilience to the reality of supercomputer systems with unavoidable failures.

Exploring techniques for creating a software ecosystem and programming environment capable of delivering computation at extreme scale, that are both resilient and efficient, will eliminate a major obstacle to scientific productivity on tomorrow's HPC platforms. In this dissertation, we advocate that in extreme scale environments, successful approaches to fault tolerance (e.g. those which exhibit acceptable recovery times and memory requirements) must go beyond traditional systems-oriented techniques and leverage intimate knowledge of dominant application algorithms, in order to create a middleware that is far more adapted and responsive to the application's performance and error characteristics.

In this work, we focus on one-sided dense linear algebra algorithms. Many of such algorithms, such as LU and QR factorization are at the center of computational scientific applications from solving large system of linear equations to eigenvalue problems. One famous example of such applications is the High Performance Linpack (HPL) benchmark program [48] that is used to gauge the performance of world's fastest supercomputers by the Top500 [99]. On large scale systems, it is not unusual that running the tuned HPL benchmark takes more than 24 hours [46], passing the MTTF of these systems according to [28], and to make things worse, unlike hard error which crashes part or the complete system and stops applications from further execution, soft error, normally induced by imperfect material packaging and radiation rays, could silently lead to incorrect result without leaving traces. Soft errors also cannot be detected by the widely adopted checkpointing/restart fault tolerance mechanism, which incurs excessive overhead in both storage, computing performance, and energy consumption, and therefore alternative methods must be sought. On the road to this path, we identify the error correcting capability, low overhead, and floating point operation compatibility as the indispensable components for practical fault tolerance algorithms, and these criteria guided the development of this dissertation.

1.1 Problem Statement

The goal of the dissertation is to demonstrate that one-sided dense linear algebra factorizations and solvers can be made fault tolerant to both hard error (fail-stop failure) and soft error. By combining the Algorithm Based Fault Tolerance (ABFT) scheme with strategic disk- and diskless checkpointing and encoding scheme that is resilient to round-off error from floating point operation, not only can data and execution flow be recovered from failure, but also it can be achieved efficiently in both space and run time overhead.

1.2 Contribution

Several fault tolerance techniques are developed in this dissertation such that both hard and soft error during one-sided dense linear algebra operation can be tolerated and the computation can reach correct result. The specific contribution of this research is summarized as follows:

1.2.1 Hard Error

- **Scalable Parallel-Q Checkpointing:** In a $P \times Q$ process grid, by performing the diskless checkpointing horizontally every Q iteration of the panel factorization with the checkpoint data stored in the outdated ABFT checksum area, and using snapshot of for the matrix data before the Q panel factorizations, the left factor of matrix factorization (for example, LU and QR) can be protected. Interrupted execution can be resumed by recovering data from both the ABFT checksum, diskless checksum and data snapshot

- **Checkpointing-on-Failure in QR factorization:** In Checkpointing-on-Failure (CoF) protocol, modification to MPI enables applications to regain control of MPI program after a failure occurs. We combine CoF with the Algorithm

Based Fault Tolerance (ABFT) and the parallel-Q checkpointing mechanism to protect QR factorization from hard error. Disk-based CoF checkpointing is only performed after failure strikes, allowing optimal checkpointing interval. And failure propagation effect in the trailing update is solved by delaying the recovery to the end of the current update step. In addition, through a dry-run procedure, the running stack of all processes in the grid are recovered to the same state.

1.2.2 Soft Error

- **Scalable Local Checkpointing for the Left Factor of Factorizations:** On distributed memory system, soft error in the left factor is protected by the local checkpointing scheme. At the end of each panel factorization, processes that own data in the panel perform diskless checkpointing locally, rather than an MPI_Reduce based global checkpointing. This makes the checkpointing scalable to both large problem sizes and process grid size. Soft errors in each column are mitigated separately.

- **Floating Point Number Weighted Checksum Encoding:** This encoding scheme is used to fight soft errors in both the left and the right factor. Random floating point numbers between 0 and 1 form the generator matrix G which is used to generate the ABFT checksum at the beginning of the factorization. For t soft errors in one column of the left factor, $O(N^t)$ complexity is required to locate and correct errors, while $O(N^{t+1})$ is needed for t soft errors in the right factors. Since soft errors in the right factor propagate, recovery is performed differently for the solver and factorization. The use of this encoding prevents large round-off and cancellation errors from floating point operation.

- **Complexity Reduction:** Since the number of tolerable soft errors t might cause large overhead than that of the matrix factorization, a complexity reduction scheme is devised to reduce the complexity to a practical level. The

basic idea is breaking N in $O(N^t)$ into smaller segments, for example $\sqrt{N}$, and the checksum encoding is performed on each segment separately. With a proof that checksum for each block also obeys the ABFT rule during the factorization, soft error checking and correcting can be carried out within each segment, therefore lowering the complexity.

- **Detection and Recovery Algorithm for Multiple Soft Errors in Linear System Solver on Distributed Memory System:** Fault tolerant algorithm for the soft errors in LU factorization based dense linear system solver $Ax = b$ is developed. In addition to the local checkpointing, floating point number weighted checksum encoding and complexity reduction, we proposed a technique to locate the columns of the initial soft errors in the right factor by casting multiple soft errors into one single different initial matrix. From this matrix, the same result of L and U can be reached but through a soft error free LU factorization. This manipulation allows the development of the $O(N^{t+1})$ algorithm to locate the soft errors. The solution x of the linear system is recovered by applying the ShermanMorrison formula.

- **Detection and Recovery Algorithm for Soft Errors in Matrix Factorization on Hybrid System with the GPGPU:** Fault tolerant algorithm for soft errors in QR factorization is developed for the hybrid system with both multicore CPU and GPGPU. Based on the local diskless checkpointing method for the left factor, vectors in the lower triangular matrix that are used to form the orthogonal left factor Q are protected by the floating point number weighted checksum performed by CPU in a time gap when CPU awaits GPU to finish the trailing matrix update. This gap is identified through profiling the hybrid QR factorization. The right factor R which has suffered large area of propagated errors is recovered by a combination of QR update and an innovative efficient Givens Rotation for GPGPU.

1.3 Dissertation outline

The rest of the dissertation is organized as follows: Chapter 2 gives the background of fault tolerance with a literature review, the fundamental questions this dissertation is set out to address, and the error model being used in this writing. Chapter 3 introduces the full matrix protection for hard error, including the parallel-Q protection for the left factor, ABFT for the right factor, and integration of CoF into QR with the disk-based checkpointing. Chapter 4 has soft error resilience for dense linear solver as the main topic. Local checkpointing, floating point weighted checksum encoding, multiple soft errors and complexity reduction are discussed in details. Chapter 5 extends the work in Chapter 4 to hybrid platform with GPGPU. A scalable CPU checkpointing and a QR update based recovery algorithm is described along with an innovative high performance Givens rotation on GPGPU. Chapter 6 concludes the dissertation and discussed future work.

Chapter 2

Background

In this work, from a software point of view we focus on tackling two most frequently seen types of HPC system error: hard error which interrupts program execution, and soft error in the form of bit flips in computing devices that *silently* causes erroneous computing result. The outcome of this work will allow Dense Linear Algebra (DLA) algorithms to run with high degree of resilience on large scale system and pay negligible overhead. By eliminating the high-overhead periodic checkpointing, better energy efficiency and higher computing performance can be attained.

2.1 Relate work

Due to the high complexity of modern HPC system, there is no single technique that can provide reliable fault tolerance to the entire system. Reliability is normally provided on a level by level basis [22]. Numerous methods have been developed to deal with different kind of error and failure at various levels of the architecture, ranging from the lowest level circuit hardware to the user application software at the top. To put our proposed methods into perspective, a systematic view of the related work is given in this section. This view include the system memory and cache, computing logics, software infrastructure and user application.

2.1.1 The Memory System

The memory system can be divided into main memory and cache memory. And memory errors are characterized as hard or soft. Hard errors are mostly cased by production process issues, such as defected silicon or DRAM packaging. Such errors, once emerging, usually become permanent quickly. Soft errors could originate from multiple possible sources, such as charged particles, radiation and also the production process. With the improvement of quality control over the years, nowadays the main cause of soft errors is electrical disturbance due to cosmic rays.

To combat errors, redundancy-based error checking and correcting code such as parity [120] and ECC are common methods for memory system. Such methods range from simple parity code to more complex Hamming [73] or Hsiao [78] codes which provide single-bit-error-correction and double-bit-error-detection (SEC/DED) capability. Parity checking has been replacing with ECC for main memory (DRAM) except in situation where detection of the error is sufficient and correction is not needed [129]. To deal with multiple-bit error, methods such as double-bit-error-correcting and triple-bit-error-detecting (DEC-TED) codes [86], single-nibble-error-correcting and double-nibble-error-detecting (SNC-DND) codes [30], and Reed Solomon (RS) codes [113] have been proposed. SNC-DND and RS codes are symbol based error codes, and both DEC-TED and SEC/DED are derived from BCH (Bose-Chaudhuri-Hocquenghem) code [19, 77] which detects and corrects random bit errors. Multiple-bit errors can also be treated with memory interleaving [17, 123] which distributes physically adjacent memory cells into different memory logical words such that groups of error are mapped to different word segments as single error, which can be corrected by SEC/DED. The main problem of complex ECC with multiple-bit error capability is the increased circuit, storage and computing overhead [126].

Nowadays most computer systems use commodity ECC DRAM for main memory. An ECC DIMM provides SEC/DED for each DRAM rank such that low impact to memory performance is maintained. Recently, it has been reported that memory

chip failures, possibly resulted from packaging and global circuit issues, may produce significant downtime [118]. As a result, chipkill-correct level reliability has been adopted[41, 85, 134, 144], where a DIMM is required to function even if an entire chip fails.

Cache memory is protected with similar methods to those for the main memory. Depending on cache levels and write-policy (write through or write-back), different ECC codes can been used. For example, for write through and inclusive last level cache, such as IBM Power 4 [130], ECC is only provided in the last level cache, the L2 cache for instance. Error in L1 cache line is corrected by re-fetching the cache line from L2 to overwrite the erroneous L1 cache line. For a detail survey and evaluation of different ECC codes for cache memory, please refer to [114]

The general trend in memory error [119] is that error will become more frequent as chip dimension and critical operating voltage keep shrinking and system scale keep increasing. Currently the widely used methods such as SEC/DED need to be modified to mitigate higher error rate, normally at the cost of higher overhead in storage, time and energy. Even though many efforts have been devoted to reduce such overhead [5, 80, 139, 143], this still does not guarantee perfect reliability because error could also strike other parts of the system that are not as well protected.

2.1.2 Compute Logic

In addition to the memory system, processing units can also be affected by errors in the sequential elements (latches and flip-flops) and combinational logic. As chip technology speeds into sub-65nm era, logic soft error will be a major concern HPC systems [101, 102]. Error detection and correction methods for compute logic can be summarized into two levels: circuit- and architecture-level. At circuit level, latches based on multiple flip-flops and other special logic circuits with verification functionalities are used and they normally suffer from large area and time overhead [91, 103, 111], or no complete error coverage. At architecture level,

space or execution redundancy is used to provide fault tolerance. Note that parity-based method for memory systems are not suitable for arithmetic operations. The most commonly used examples of compute logic error codes are product codes, linear residue codes, and residue-class codes [96, 112] such as AN code, which checks the result of operation such as $N_1 \oplus N_2$ by testing the equality of $A \times N_1 \oplus A \times N_2 = A \times (N_1 \oplus N_2)$, where A is a constant and $\oplus$ is operator such as plus. Such verification can be carried out with or without extra dedicated error checking circuit. Compromise is made between operation delay, design intrusiveness and circuit area overhead. Although residue code is not applicable to floating-point arithmetic directly, it has been applied to various stages of the floating-point operations independently [87].

Code-based methods are cost-effective, but custom design is required and they are relatively inflexible to cover errors in a wide range of hardware structures. A more viable option is through replicating the execution of some logic units and verifying the result, for example in several IBM systems [98, 127]. Replication can be at various level, from a single module to an entire core. In this work our focus is computing intensive application, and while replication is effective for control-intensive processors, the replicated logic units for either re-computing or error verification, and the overall effect is close to fully replicating most of the processor, causing excessive overhead especially for computing intensive applications.

2.1.3 Roll-back Recovery with Disk-based Checkpointing and Message Logging

Although error detection and correct mechanisms have been developed and integrated into hardware circuits, such design normally requires compromise between various aspects such as fault tolerance capability, chip real estate, energy. Higher error rate due to the increase of system scale and decrease of chip density requires more complex ECC, and computing intensive application like dense linear algebra operations makes replication based method only theoretically possible due to the energy expense

constrain. To complement such situation, fault tolerance method has been devised at the software level. Such methods include checkpointing (disk and diskless), message logging, compiler based technique, algorithm based fault tolerance (ABFT), to natural fault tolerance. Similar to the coding based methods in hardware, redundancy is also used in software fault tolerance methods such that lost data can be restored and application execution can be resumed. The key concept for execution recovery is consistent system states, which, for a distributed memory system using message passing interface (MPI) libraries for communication, consist of the local states on all nodes such as memory space, registers, etc., and "on-the-fly" messages. With a consistent system state available, the application execution can recover from failure by using the Checkpoint-and-Restart method, or C/R. A comprehensive survey of C/R can be found in [54].

Message logging based rollback recovery works with the assumption of piecewise deterministically [39] and performs recovery by replaying messages in the exact original order for the failed process. This way a process can be rolled back to its state right before failure even if no checkpoint is available. Message logging has flavors such as pessimistic logging, optimistic logging and casual logging, depending on the treatment to the existence of orphan process and recovery overhead [7]. In general, message logging fits applications that perform constant interact with input and output devices which cannot be rolled back using checkpointing at time of failure [56]. For automatic/transparent fault tolerance of MPI applications, message logging is recommended because of the high overhead of coordinated checkpointing which puts large stress on stable storage devices during checkpointing [24, 83]. However for dense linear algebra application, disk-based checkpointing can be replaced with diskless checkpointing, and user can relatively easily select the appropriate checkpoint location such that overhead can be largely lowered.

What further attracts application developers is checkpointing-only fault tolerance. When applicable, this requires much less development effort in contrast to complex

message logging system. Checkpointing can be performed transparently by infrastructure systems, which reportedly still remains the most popular fault tolerance mechanism on large scale system [1], or at application level explicitly.

In checkpointing, processes periodically saves their states to stable storage. These saved states should provide sufficient information to recover program execution. Two main categories of checkpointing are coordinated and uncoordinated checkpointing. In uncoordinated protocol, processes take checkpoint independently, which may prevent excessive overhead due to synchronization, but could lead to the domino effect [34] when consistent system state cannot be achieved, and all processes are forced to roll back to the initial state of the computation, losing all checkpoint and useful work performed till the failure occurs.

To avoid the disastrous result of domino effect in recovery, methods such as coordinated checkpointing [10] and communication-induced checkpointing [6], have been developed such that valid consistent state is guaranteed to exist regardless of the failure moment and location. In coordinated checkpointing, system-wide checkpointing is taken at a certain interval after all processes are synchronized. Checkpointing can be performed in system-level (for example, [64, 115]) or user/application level [108, 137].

For systems that use disk as storage media, performance overhead mostly comes from the I/O operations which save and load checkpoints data from "stable storage", other overhead including the time to restart the running environment, such as MPI, as discussed in [21].

To reduce such overhead, several methods have been developed, such as incremental checkpointing [67], forked (copy-on-write) checkpointing [84], memory exclusion [108]. With the fast increasing of number of nodes/cores, stable storage medium is still easily outnumbered and causing large overhead.

2.1.4 Diskless Checkpointing

To effectively reduce the checkpointing overhead, Plank proposed diskless checkpointing where "stable disk" is replaced by memory as the checkpoint storage media [109, 110]. In diskless checkpointing, processor redundancy, memory redundancy and failure coverage are traded off so that no stable storage is necessary to recover from failure. In applications, diskless checkpointing has been adopted to make several matrix operations fault tolerant to single hard error [81] with low overhead. Parity based checksum (XOR of bits in floating point numbers) is generated prior to computation, and is updated if data has changed in the each iterations. This checkpointing method is more suitable for applications that modify small amount of memory between iterations, such as the left-looking LU factorization, which however has lower performance than the right looking version that modifies large memory area in each iteration. For these algorithms, checksum and reverse computation methods are used to reduce memory usage.

The performance of different diskless checkpointing schemes are studied in [35, 125]. In [125], neighbour- and parity-based diskless checkpointing are implemented and evaluated on their Xplorer Parsytec machine with 8 transputers (T805) with various application benchmarks, such as NBODY, SOR and NQUEEN. Neighbour-based method (also called "checkpoint mirroring") stores checkpoints into its own physical memory and that of its neighbours', while parity-based method uses extra a processor called "parity processor" that keeps the parity checkpoint (XOR) of all local checkpoints taken by the each process. Their experiment results show that neighbour-based checkpoint has much better performance than the parity scheme at the cost of higher memory requirement. Similar result is report in [110] that neighbour based scheme has lower overhead if local checkpoints are store on disk. It is worth noting that since neighbour based method cannot tolerate total failures of the system, it should be used together with other checkpointing scheme, for example as shown in [135], where a two-level method is proposed in which diskless checkpointing

is used to tolerate the more probable single failures, while traditional disk based checkpointing is adopted for the less probable multiple failures. The objective of such design is to minimize the average fault tolerant overhead.

2.1.5 Algorithm Based Fault Tolerance

Algorithm Based Fault Tolerance (ABFT), which initially stemmed from the effort of mitigating silent error in systolic arrays [79], was introduced to further reduce the overhead of fault tolerance on modern computing systems. ABFT maintains consistency between the checksum and compute data by applying appropriate mathematical operations to both parties. Typically, for linear algebra operations, the input matrix is extended with supplementary columns and/or rows containing checksums. This initial encoding happens only once; the matrix algorithms are designed/modified to work on the encoded checksum along with matrix data, which enables invariant the checksum's relationship with the data during the course of the algorithm. Should some data be damaged by failures, it is then possible to recover the application by inverting the checksum operation to recreate missing data. The overhead of ABFT is usually low, since no periodical global checkpoint or rollback-recovery is involved during computation and the computation complexity of the checksum operations scales similarly to the related matrix operation. ABFT and diskless checkpointing have been combined to apply to basic matrix operations like matrix-matrix multiplication [20, 31, 32, 33]. ABFT has also been implemented for the High Performance Linpack (HPL) [40] and the Cholesky factorization [72] for fail-stop failure. Both Cholesky and HPL have the same factorization structure, where only half of the factorization result is required, and the update to the trailing matrix is based on the fact that the left factor result is a triangular matrix. This approach however does not necessarily apply to other factorizations, like QR where the left factor matrix is full, neither when the application requires both left and right factorization results. We have shown in [49], using the LU and QR factorization as

examples, a full matrix protection solution with low space and time overhead for hard error.

Using ABFT to mitigate single soft errors in dense matrix factorization has been explored in [89, 90]. Later, this was extended to multiple errors [9, 61, 107] by adopting techniques from finite-field based error correcting code (ECC), such as Reed-Solomon [113] and BCH [19, 77]. While mathematically these methods can detect and correct single and multiple soft errors, they suffer major limitation because of the assumption that all computation is carried out with exact arithmetics, which cannot be fulfilled by modern computers that use floating point number. For instance, in [61], j^n is used as weights for the weighted checksum generation, where j and n are integers. Such calculation could easily cause problems like overflow, cancellation and large round-off error with large computing scale and matrices. Realizing this limitation, [62] has proposed a solution that uses rational number, rather than integers, as weights to reduce the dynamic range expansion problem pointed out by [25, 26, 90], and a decoding technique that is an exact analogue of that of the BCH codes for two errors. In [62], however, all calculations are still assumed to be performed using "exact rational arithmetic". In addition, no discussion is given on how to extend to more than two errors, nor did it cover another limitation of this ABFT checksum based method, the left factor. Similar to the hard error case, ABFT only maintains relationship between the checksum and the right factor. For the concern of the left factor, backward error assertions based methods [18, 60] has been introduced to correct erroneous solution of linear system solver that is based on LU factorization. In [18] it is shown that transient errors during factorization could slip through the ABFT checksum verification, making ABFT-only method unreliable, and iterative refinement is used to correct errors in solution x. This method is effective for small error, and is extended in [60] with a large error (LE) detection and correction scheme which has $O(n^2)$ computational complexity and improved the error coverage. Errors not correctable in this algorithm is signaled. In a series of work by Du et al.[50, 52, 53], it has been shown that in the presence of round-off error on large scale and hybrid

system with accelerators like the GPGPUs, soft errors in both the left and right factors in DLA operations can be detected and corrected.

2.1.6 Other Methods For Soft Error

Recently, Fiala et al. has shown a method that uses TMR(triple modular redundancy) [58], C/R, and explicit memory locking [59] to guard memory for soft error mitigation. These methods do not suffer from round-off error but demand considerable extra resources and are difficult to extend to devices like cache and systems with GPGPU, especially for computation intensive applications like DLA operations. Also, an on-line soft error detection scheme has been devised to work on matrix-matrix multiplication using both CPU-only system and hybrid systems with GPGPU [42]. Experimental result on NVIDIA Tesla S1070 GPUs shows that the online error correction overhead is much lower than that of TMR and traditional ABFT. It is yet to show how this mechanism could be extended to more complex matrix operations.

2.2 Fundamental Questions

Facing the complexity of HPC system error, we identify the following fundamental questions that are to be addressed to fully understand and better design fault tolerant algorithms. For clarity, we use System Error (SE) to refer to both hard and soft errors.

1. How does SE affect DLA operations?
2. How to detect and locate the occurrence of SE?
3. How to recover from SE?

SE affects computation in very different ways. Time and location are two of the main focuses. A typical time issue has been discussed in [49] where the time of SE leads to different recovery method. And an example of how location of SE affect fault

tolerant algorithm design is demonstrated in [50] where soft error in the upper and lower triangular requires different protection strategies. Another important location issue is the bit flip location within floating point number. With the IEEE-754 format, even a single bit flip could lead to large quantity change to the floating point number and since most of the current ABFT algorithms uses SUM-based checkpointing, the extent to which ABFT detection and recovery are still effective will be quantified.

It is straightforward to see that the detection and locating of soft errors are more difficult than that for hard error, for which we assume that the failed process is reported by the supporting Infrastructure of MPI in the form of MPI process ID. No such assumption is to be made for soft error. To worsen the situation, since soft error is hard to spot instantaneously when it strikes, it participates the DLA operation and causes more damage than the initial bit flip error. The recovery of DLA operation result needs to reconstruct the correct solution from such erroneous result and manage a lower recovery cost than re-computing the solution from scratch.

2.3 Error Model

In this work, we use the following error model:

- **Hard Error:** Fail-stop failure causes one MPI process to stop executing and responding to communication with other MPI processes. All content in the failed MPI process's memory is lost such as the running stack and matrix data.

- **Soft Error:** Soft error appears silently and permanently changes the value of floating point number stored in the main memory. We use "soft error" and "transient error" interchangeably in this text.

Chapter 3

Hard Error Resilience on Distributed Memory System

3.1 Introduction

While many types of failures can strike a distributed system [65], the focus of this chapter is on the most common representation: the fail-stop model. In this model, a failure is defined as a process that *completely* and *definitely* stops responding, triggering the loss of a critical part of the global application state. To be more realistic, we assume a failure could occur at any moment and can affect any parts of the application's data. We introduce a new generic hybrid approach based on algorithm-based fault tolerance (ABFT) that can be applied to several ubiquitous one-sided dense linear factorizations. Using one of these factorizations, namely LU with partial pivoting, which is significantly more challenging due to pivoting, we theoretically prove that this scheme successfully applies to the three well known one-sided factorizations, Cholesky, LU and QR. To validate these claims, we implement and evaluate this generic ABFT scheme with both the LU and QR factorizations. A significant contribution of this chapter is to protect the part of the matrix below

the diagonal (referred to as "the left factor" in the rest of the text) during the factorization, which was hitherto never achieved.

The rest of the chapter is organized as follows: Section 3.2 presents background and prior work in the domain; Section 3.3 reviews the features of full factorizations. Section 3.4 discusses the protection of the right factor using the ABFT method. Section 3.5 reviews the idea of vertical checkpointing and proposes the new checkpointing method to protect the left factor. Section 3.6 shows how Checkpoint-on-Demand is integrated to provide the fault tolerance support from the MPI infrastructure. Section 3.7 evaluates the performance and overhead of the proposed algorithm using the example of LU and QR, and section 5.8 concludes the chapter.

3.2 Algorithm Based Fault Tolerance Background

The most well-known fault-tolerance technique for parallel applications, checkpoint-restart (C/R), encompasses two categories, the system and application level. At the system level, message passing middleware deals with faults automatically, without intervention from the application developer or user ([23, 27]). At the application level, the application state is dumped to a reliable storage when the application code mandates it. Even though C/R bears the disadvantage of high overhead while writing data to stable storage, it is widely used nowadays by high end systems [1]. To reduce the overhead of C/R, diskless checkpointing [88, 110] has been introduced to store checksum in memory rather than stable storage. While diskless checkpointing has shown promising performance in some applications (for instance, FFT in [55]), it exhibits large overheads for applications modifying substantial memory regions between checkpoints [110], as is the case with factorizations.

In contrast, Algorithm Based Fault Tolerance (ABFT) is based on adapting the algorithm so that the application dataset can be recovered at any moment, without involving costly checkpoints. ABFT was first introduced to deal with silent error in systolic arrays [79]. Unlike other methods that treat the recovery data and computing

data separately, ABFT approaches are based on the idea of maintaining consistency of the recovery data, by applying appropriate mathematical operations on both the original and recovery data. Typically, for linear algebra operations, the input matrix is extended with supplementary columns and/or rows containing checksums. This initial encoding happens only once; the matrix algorithms are designed to work on the encoded checksum along with matrix data, similar mathematical operations are applied to both the data and the checksum so that the checksum relationship is kept invariant during the course of the algorithm. Should some data be damaged by failures, it is then possible to recover the application by inverting the checksum operation to recreate missing data. The overhead of ABFT is usually low, since no periodical global checkpoint or rollback-recovery is involved during computation and the computation complexity of the checksum operations scales similarly to the related matrix operation. ABFT and diskless checkpointing have been combined to apply to basic matrix operations like matrix-matrix multiplication [20, 31, 32, 33] and have been implemented on algorithms similar to those of ScaLAPACK [15], which is widely used for dense matrix operations on parallel distributed memory systems.

Recently, ABFT has been applied to the High Performance Linpack (HPL) [40] and to the Cholesky factorization [72]. Both Cholesky and HPL have the same factorization structure, where only half of the factorization result is required, and the update to the trailing matrix is based on the fact that the left factor result is a triangular matrix. This approach however does not necessarily apply to other factorizations, like QR where the left factor matrix is full, nor when the application requires both the left and right factorization results. Also, LU with partial pivoting, when applied to the lower triangular L, potentially changes the checksum relation and renders basic checkpointing approaches useless.

The generic ABFT framework for matrix factorizations we introduce in this chapter can be applied not only to Cholesky and HPL, but also to LU and QR. The right factor is protected by a traditional ABFT checksum, while the left factor is protected by a novel vertical checkpointing scheme, making the resulting

approach an hybrid between ABFT and algorithm driven checkpointing. Indeed, this checkpointing algorithm harnesses some of the properties of the factorization algorithm to exchange limited amount of rollback with the ability to overlap the checkpointing of several panel operations running in parallel. Other contributions of this chapter include correctness proofs and overhead characterization for the ABFT approach on the most popular 2D-block cyclic distribution (as opposed to the 1D distributions used in previous works). These proofs consider the effect of failures during critical phases of the algorithm, and demonstrate that recovery is possible without suffering from error propagation

3.3 Full Factorizations of Matrix

In this chapter, we consider the case of factorizations where the lower triangular part of the factorization result matters, as is the case in QR and LU with pivoting. For example, the left factor Q is required when using QR to solve the least square problem, and so is L when solving $A^k x = b$ with the "LU factorization outside the loop" method [68]. In the remaining of this section, we recall the main algorithm of the most complex case of one-sided factorization, block LU with pivoting. Additionally, we highlight challenges specific to this type of algorithms, when compared to algorithms studied in previous works.

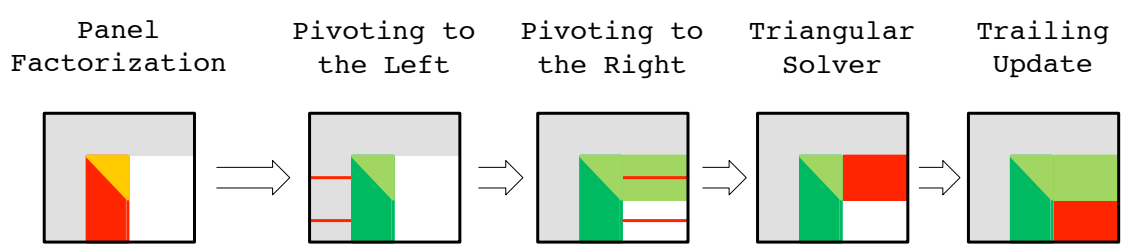


Figure 3.1: Steps applied to the input matrix in an iteration of the LU factorization; Green: Just finished; Red & Orange: being processed; Gray: Finished in previous iterations

Figure 3.1 presents the diagram of the basic operations applied to the input matrix to perform the factorization. The block LU factorization algorithm can be seen as a recursive process. At each iteration, the panel factorization is applied on a block column. This panel operation factorizes the upper square (selecting adequate pivots and applying internal row swapping as necessary to ensure numerical stability), and scales the lower polygon accordingly. The output of this panel is used to apply row swapping to the result of previous iterations, on the left, and to the trailing matrix on the right. The triangular solver is applied to the right of the factored block to scale it accordingly, and then the trailing matrix is updated by applying a matrix-matrix multiply update. Then the trailing matrix is used as the target for the next iteration of the recursive algorithm, until the trailing matrix is empty. Technically, each of these basic steps is usually performed by applying a parallel Basic Linear Algebra Subroutine (PBLAS).

The structure of the other one-sided factorizations, Cholesky and QR, are similar with minor differences. In the case of Cholesky, the trailing matrix update involves only the upper triangle, as the lower left factor is not critical. For QR, the computation of pivots and the swapping are not necessary as the QR algorithm is more stable. Moreover, there are a significant number of applications, like iterative refinement and algorithms for eigenvalue problems, where the entire factorization result, including the lower part, is needed. Therefore, a scalable and efficient protection scheme for the lower left triangular part of the factorization result is required.

3.4 Protection of the Right Factor Matrix with ABFT

In this section, we detail the ABFT approach that is used to protect the upper triangle from failures, while considering the intricacies of typical block cyclic distributions and failure detection delays.

3.4.1 Checksum Relationship

ABFT approaches are based upon the principle of keeping an invariant bijective relationship between protective supplementary blocks and the original data through the execution of the algorithm, by the application of numerical updates to the checksum. In order to use ABFT for matrix factorization, an initial checksum is generated before the actual computation starts. In future references we use G to refer to the generator matrix, and A to the original input matrix. The checksum C for A is produced by

$$C = GA \ or \ C = AG \tag{3.1}$$

When G is all-1 vector, the checksum is simply the sum of all data items from a certain row or column. Referred to as the *checksum relationship*, (3.1) can be used at any step of the computation for checking data integrity (by detecting mismatching checksum and data) and recovery (inverting the relation builds the difference between the original and the degraded dataset). With the type of failures we consider (Fail-Stop), data cannot be corrupted, so we will use this relationship to implement the recovery mechanism only. This relationship has been shown separately for Cholesky [72], and HPL [40], both sharing the property of updating the trailing matrix with a lower triangular matrix. However, in this chapter we consider the general case of matrix factorization algorithms, including those where the full matrix is used for trailing matrix updates (as is the case for QR and LU with partial pivoting). In this

context, the invariant property has not been demonstrated; we will now demonstrate that it holds for full matrix based updates algorithms as well.

3.4.2 Checksum Invariant with Full Matrix Update

In [89], ZU is used to represent a matrix factorization (optionally with pairwise pivoting for LU), where Z is the left matrix (lower triangular in the case of Cholesky or full for LU and QR) and U is an upper triangular matrix. The factorization is then regarded as the process of applying a series of matrices Z_i to A from the left until $Z_i Z_{i-1} \cdots Z_0 A$ becomes upper triangular.

Theorem 3.4.1. *Checksum relationship established before ZU factorization is maintained during and after factorization.*

Proof. Suppose data matrix $A \in \mathbb{R}^{n \times n}$ is to be factored as $A = ZU$, where Z *and* $U \in \mathbb{R}^{n \times n}$ and U is an upper triangular matrix. A is checkpointed using generator matrix $G \in \mathbb{R}^{n \times nc}$, where nc is the width of checksum. To factor A into upper triangular form, a series of transformation matrices Z_i is applied to A (with partial pivoting in LU).

Case 1: No Pivoting

$$U = Z_n Z_{n-1} \ldots Z_1 A$$

Now the same operation is applied to $A_c = [A,\ AG]$

$$\begin{aligned} U_c &= Z_n Z_{n-1} \ldots Z_1 \, [A,\ AG] \\ &= [Z_n Z_{n-1} \ldots Z_1 A,\ \ Z_n Z_{n-1} \ldots Z_1 AG] \\ &= [U,\ \ UG] \end{aligned}$$

For any $k \leq n$, using U^k to represent the result of U at step k,

$$\begin{aligned} U_c^k &= Z_k Z_{k-1} \ldots Z_1 \left[A,\ AG\right] \\ &= \left[Z_k Z_{k-1} \ldots Z_1 A,\quad Z_k Z_{k-1} \ldots Z_1 AG\right] \\ &= \left[U^k,\quad U^k G\right] \end{aligned}$$

Case 2: With partial pivoting:

$$\begin{aligned} U_c^k &= Z_k P_k Z_{k-1} P_{k-1} \ldots Z_1 P_1 \left[A,\ AG\right] \\ &= \left[Z_k P_k Z_{k-1} P_{k-1} \ldots Z_1 P_1 A,\right. \\ &\qquad \left. Z_k P_k Z_{k-1} P_{k-1} \ldots Z_1 P_1 AG\right] \\ &= \left[U^k,\quad U^k G\right] \end{aligned}$$

Therefore the checksum relationship holds for LU with partial pivoting, Cholesky and QR factorizations. □

3.4.3 Checksum Invariant in Block Algorithms

Theorem 3.4.1 shows the mathematical checksum relationship in matrix factorizations. However, in real-world, HPC factorizations are performed in block algorithms, and execution is carried out in a recursive way. Linear algebra packages, like ScaLAPACK, consist of several function components for each factorization. For instance, LU has a panel factorization, a triangular solver and a matrix-matrix multiplication. We need to ensure that the checksum relationship also holds for block algorithms, both at the end of each iteration, and after the factorization is completed.

Theorem 3.4.2. *For ZU factorization in block algorithm, checksum at the end of each iteration only covers the upper triangular part of data that has already been factored and are still being factored in the trailing matrix.*

Proof. Input Matrix A is split into blocks of data of size $nb \times nb$ (A_{ij}, Z_{ij}, U_{ij}), and the following stands:

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \end{bmatrix} = \begin{bmatrix} Z_{11} & Z_{12} \\ Z_{21} & Z_{22} \end{bmatrix} \begin{bmatrix} U_{11} & U_{12} & U_{13} \\ 0 & U_{22} & U_{23} \end{bmatrix}, \tag{3.2}$$

where $A_{13} = A_{11} + A_{12}$, and $A_{23} = A_{21} + A_{22}$.

Since $A_{13} = Z_{11}U_{13} + Z_{12}U_{23}$, and $A_{23} = Z_{21}U_{13} + Z_{22}U_{23}$, and using the relation

$$\begin{cases} A_{11} &= Z_{11}U_{11} \\ A_{12} &= Z_{11}U_{12} + Z_{12}U_{22} \\ A_{21} &= Z_{21}U_{11} \\ A_{22} &= Z_{21}U_{12} + Z_{22}U_{22} \end{cases}$$

in (3.2), we have the following system of equations:

$$\begin{cases} Z_{21}(U_{11} + U_{12} - U_{13}) = Z_{22}(U_{23} - U_{22}) \\ Z_{11}(U_{11} + U_{12} - U_{13}) = Z_{12}(U_{23} - U_{22}) \end{cases}$$

This can be written as:

$$\begin{bmatrix} Z_{11} & Z_{12} \\ Z_{21} & Z_{22} \end{bmatrix} \begin{bmatrix} U_{11} + U_{12} - U_{13} \\ -(U_{23} - U_{22}) \end{bmatrix} = 0$$

For LU, Cholesky and QR, $\begin{bmatrix} Z_{11} & Z_{12} \\ Z_{21} & Z_{22} \end{bmatrix}$ is always nonsingular, so $\begin{bmatrix} U_{11} + U_{12} - U_{13} \\ U_{23} - U_{22} \end{bmatrix} = 0$, and $\begin{cases} U_{11} + U_{12} &= U_{13} \\ U_{23} &= U_{22} \end{cases}$.

This shows that after ZU factorization, checksum blocks cover the upper triangular matrix U only, even for the diagonal blocks. At the end of each iteration, for example the first iteration in (3.2), Z_{11}, U_{11}, Z_{21} and U_{12} are completed, and U_{13}

is already $U_{11} + U_{12}$. The trailing matrix A_{22} is updated with

$$A_{22}' = A_{22} - Z_{21}U_{12} = Z_{22}U_{22}.$$

and A_{23} is updated to

$$\begin{aligned} A_{23}' &= A_{23} - Z_{21}U_{13} \\ &= A_{21} + A_{22} - Z_{21}(U_{11} + U_{12}) \\ &= Z_{21}U_{11} + A_{22} - Z_{21}U_{11} - Z_{21}U_{12} \\ &= A_{22} - Z_{21}U_{12} = Z_{22}U_{22} \end{aligned}$$

Therefore, at the end of each iteration, data blocks that have already been and are still being factored remain covered by checksum blocks. □

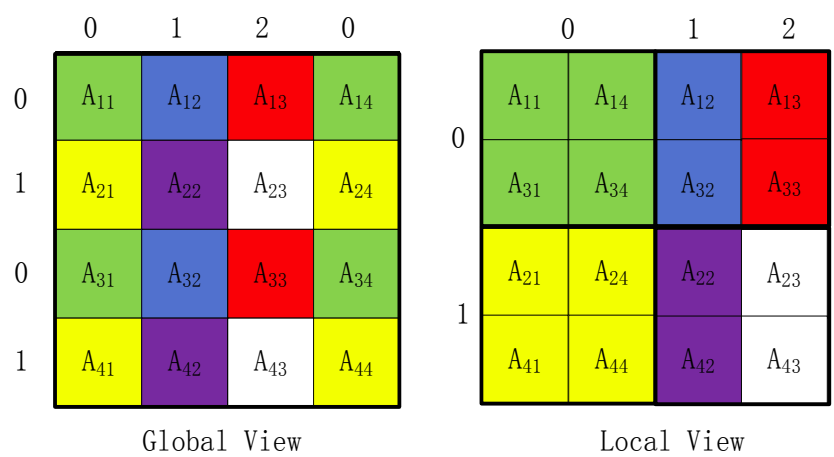


Figure 3.2: Example of a 2D block-cyclic data distribution

3.4.4 Issues with Two-Dimensional Block-cyclic Distribution

It has been well established that data layout plays an important role in the performance of parallel matrix operations on distributed memory systems [37, 82].

In 2D block-cyclic distributions, data is divided into equally sized blocks, and all computing units are organized into a virtual two-dimension grid P by Q. Each data block is distributed to computing units in round robin following the two dimensions of the virtual grid. Figure 3.2 is an example of a $P = 2, Q = 3$ grid applied to a global matrix of 4×4 blocks. The same color represents the same process while numbering in A_{ij} indicates the location in the global matrix. This layout helps with load balancing and reduces data communication frequency, because in each step of the algorithm, many computing units can be engaged in computations concurrently, and communications pertaining to blocks positioned on the same unit can be grouped. Thanks to these advantages, many prominent software libraries (like ScaLAPACK [45]) assume a 2D block-cyclic distribution.

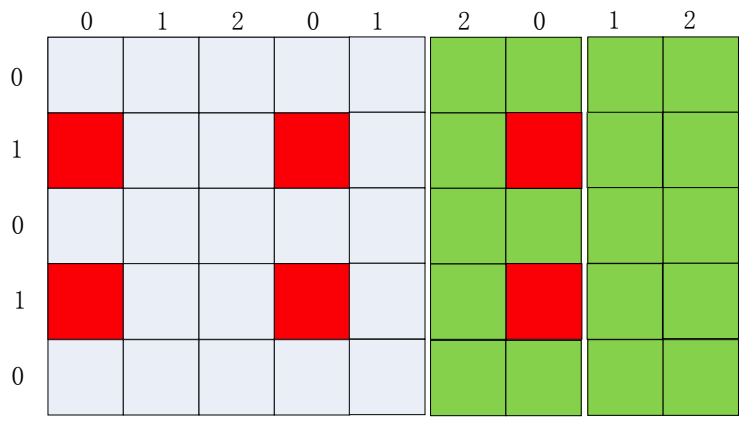


Figure 3.3: Holes in a checksum protected matrix caused by a single failure and the naive checksum duplication protection scheme (3x2 process grid)

However, with a 2D block-cyclic data distribution, the failure of a single process, usually a computing node which keeps several non-contiguous blocks of the matrix, results in holes scattered across the whole matrix. Figure 3.3 is an example of a 5×5 blocks matrix (on the left) with a 2×3 process grid. Red blocks represent holes caused by the failure of the single process (1, 0). In the general case, these holes can impact both checksum and matrix data at the same time.

3.4.5 Checksum Protection Against Failure

Our algorithm works under the assumption that any process can fail and therefore the data, including the checksum, can be lost. Rather than forcing checksum and data on different processes and assuming only one would be lost, as in [40], we put checksum and data together in the process grid and design the checksum protection algorithm accordingly.

Minimum Checksum Amount for Block Cyclic Distributions

Theoretically, the sum-based checksum C_k of a series of N blocks $A_i, 1 \leq i \leq N$, where N is the total number of blocks in one row/column of the matrix, is computed by:

$$C_k = \sum_{k=1}^{N} A_k \tag{3.3}$$

With the 2D block-cyclic distribution, a single failure punches multiple holes in the global matrix. With more than one hole per row/column, C_k in (3.3) is not sufficient to recover all lost data. A slightly more sophisticated checksum scheme is required.

Theorem 3.4.3. *Using sum-based checkpointing, for N data items distributed in block-cyclic onto Q processes, the size of the checksum to recover from the loss of one process is $\lceil \frac{N}{Q} \rceil$*

Proof. With 2D block-cyclic, each process gets $\lceil \frac{N}{Q} \rceil$ items. At the failure of one process, all data items in the group held by the process are lost. Take data item a_i, $1 \leq i \leq \lceil \frac{N}{Q} \rceil$, from group k, $1 \leq k \leq Q$. To be able to recover a_i, any data item in group k cannot be used, so at least one item from another group is required to create the checksum, and this generates one additional checksum item. Therefore for all items in group k, $\lceil \frac{N}{Q} \rceil$ checksum items are generated so that any item in group k can be recovered. □

Applying this theorem, we have the following checksum algorithm: Suppose Q processes are in a process column or row, and let each process have K blocks of data of size $nb \times nb$. Without loss of generality, let K be the largest number of blocks owned by any of the Q processes. From Theorem 3.4.3, the size of the checksum in this row is K blocks.

Let C_i be the ith checksum item, and A_i^j, be the ith data item on process j, $1 \leq i \leq \lceil \frac{N}{Q} \rceil$, $1 \leq j \leq Q$:

$$C_k = \sum_{k=1}^{Q} A_k^k \tag{3.4}$$

Under (3.4), we have the following corollary:

Corollary 3.4.4. *The ith block of checksum is calculated using the ith block of data of each process having at least i blocks.*

Checksum Duplicates

Since ABFT checksum is stored by regular processors, it has to be considered as fragile as the matrix data. From Theorem 3.4.3 and using the same N and Q, the total number of checksum blocks is $K = \lceil \frac{N}{Q} \rceil$. These checksum blocks can be appended to the bottom or to the right of the global data matrix accordingly, and since checksum is stored on computing processes, these K checksum blocks are distributed over $\min(K, Q)$ processes (see Figure 3.3). If a failure strikes any of these processes, like (1, 0) in this example, some checksum is lost and cannot be recovered. Therefore, checksum itself needs protection; in our work, duplication is used to protect checksum from failure.

A straightforward way of performing duplication is to make a copy of the entire checksum block, as illustrated by the two rightmost columns in Figure 3.3. While simple to implement, this method suffers from two major defects. First, if the checksum width K is a multiple of Q (or P for column checksum), the duplicate

of a checksum block is located on the same processors, defeating the purpose of duplication. This can be solved at the cost of introducing an extra empty column in the process grid to resolve the mapping conflict. More importantly, to maintain the checksum invariant property, it is required to apply the trailing matrix update on the checksum (and its duplicates) as well. From corollary 3.4.4, once all the ith block columns on each process have finished the panel factorization (in Q step), the ith checksum block column is no longer active in any further computation (except pivoting) and should be excluded from the computing scope to reduce the ABFT overhead. This is problematic, as splitting the PBLAS calls to avoid excluded columns has a significant impact on the trailing matrix update efficiency.

Reverse Neighboring Checksum Storage

With the observation of how checksum is maintained during factorization, we propose the following reverse neighboring checksum duplication method that allows for applying the update in a single PBLAS call without incurring extraneous computation.

Algorithm 1 Checksum Management

On a $P \times Q$ grid, matrix is $M \times N$, block size is $NB \times NB$
C_k represents the kth checksum block column
A_k represents the kth data block column
Before factorization:
Generate the initial checksum:
$C_k = \sum_{j=(k-1)\times Q+1}^{(k-1)\times Q+Q} A_j$, $k = 1, \cdots, \left\lceil \frac{N}{NB \times Q} \right\rceil$
For each of C_k, make a copy of the whole block column and put right next to its original block column
Checksum C_k and its copy are put in the kth position starting from the far right end
Begin factorization
Host algorithm starts with an initial scope of M rows and $N + \left\lceil \frac{N}{Q} \right\rceil$ columns
For each Q panel factorizations, the scope decreases M rows and $2 \times NB$ columns
End factorization

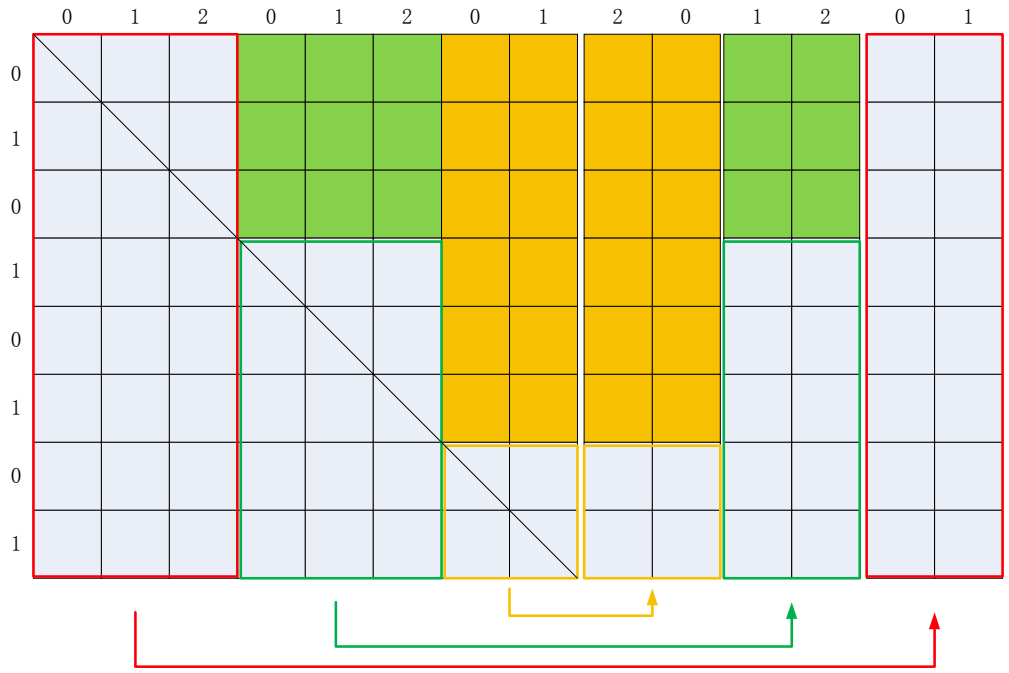


Figure 3.4: Reverse neighboring checksum storage, with two checksum duplicates per Q-wide groups

Figure 3.4 is an example of the reverse neighboring checksum method on a 2×3 grid. The data matrix has 8×8 blocks and therefore the size of checksum is 8×3 blocks with an extra 8×3 blocks copy. The arrows indicate where checksum blocks are stored on the right of the data matrix, according to the reverse storage scheme. For example, in the LU factorization, the first 3 block columns produce the checksum in the last two block columns (hence making 2 duplicate copies of the checksum). Because copies are stored in consecutive columns of the process grid, for any 2D grid with $Q > 1$, the checksum duplicates are guaranteed to be stored on different processors. The triangular solve (TRSM) and trailing matrix update (GEMM) are applied to the whole checksum area until the first three columns are factored. In following factorization steps, the two last block columns of checksum are excluded from the TRSM and GEMM scope. Since TRSM and GEMM claim most of the computation in the LU factorization, this shrinking scope greatly reduces the overhead of the ABFT mechanism. One can note that only the upper part of the checksum

is useful, we will explain in the next section how this extra storage can be used to protect the lower triangular part of the matrix.

3.4.6 Delayed Recovery and Error Propagation

In this chapter, we assume that a failure can strike at any moment during the life span of factorization operations or even the recovery process. Theorem 3.4.2 proves that at the moment where the failure happens, the checksum invariant property is satisfied, meaning that the recovery can proceed successfully. However, in large scale systems, which are asynchronous by nature, the time interval between the failure and the moment when it is detected by other processes is unknown, leading to delayed recoveries, with opportunities for error propagation.

The ZU factorization is composed of several sub-algorithms that are called on different parts of the matrix. Matrix multiplication, which is used for trailing matrix updates and claims more than 95% of the execution time, has been shown to be ABFT compatible [20] , that is to compute the correct result even with delayed recovery. One feature that has the potential to curb this compatibility is pivoting, in LU, especially when a failure occurs between the panel factorization and the row swapping updates, there is a potential for destruction of rows in otherwise unaffected blocks.

Figure 3.5 shows an example of such a case. Suppose the current panel contributes to the ith column of checksum. When panel factorization finishes, the ith column becomes intermediate data which does not cover any column of matrix. If a failure at this instant causes holes in the current panel area, then lost data can be recovered right away. Pivoting for this panel factorization has only been applied within the light green area. Panel factorization is repeated to continue on the rest of the factorization. However, if failure causes holes in other columns that also contribute to the ith column of checksum, these holes cannot be recovered until the end of the trailing matrix update. To make it worse, after the panel factorization, pivoting starts to be applied outside the panel area and can move rows in holes into healthy area or vice versa,

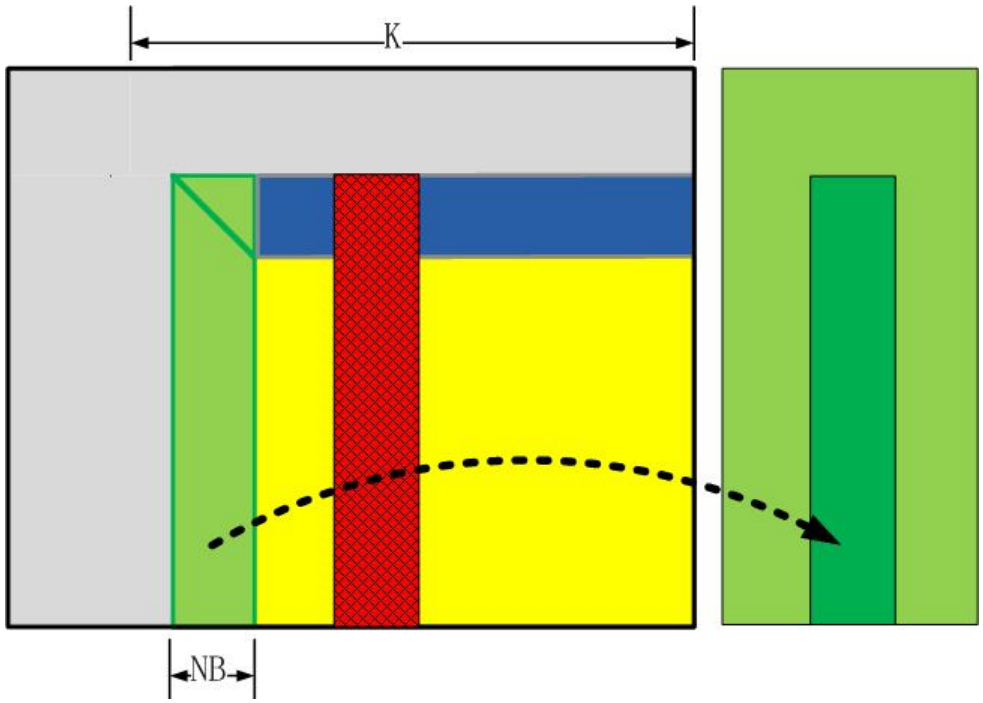


Figure 3.5: Ghost pivoting Issue
Gray: Result in previous steps
Light Green: Panel factorization result in current step
Deep Green: The checksum that protects the light green
Blue: TRSM zone Yellow: GEMM zone
Red: one of the columns affected by pivoting

extending the recovery area to the whole column, as shown in red in Figure 3.5 including triangular solving area. To recover from this case, in addition to matrix multiplication, the triangular solver is also required to be protected by ABFT.

Theorem 3.4.5. *Failure in the right-hand sides of triangular solver can recover from fail-stop failure using ABFT.*

Proof. Suppose A is the upper or lower triangular matrix produced by LU factorization (non-blocked in ScaLAPACK LU), B is the right-hand side, and the triangular solver solves the equation $Ax = B$.

Supplement B with checksum generated by $B_c = B * G_r$ to extended form $\hat{B} = [B,\ B_c]$, where G_r is the generator matrix. Solve the extended triangular equation:

$$\begin{aligned} Ax_c &= B_c = [B,\ B_c] \\ \therefore\ x_c &= A^{-1} \times [B,\ B_c] \\ &= \left[A^{-1}B,\ A^{-1}B_c\right] \\ &= \left[x,\ A^{-1}BG_r\right] \\ &= [x,\ xG_r] \end{aligned}$$

Therefore data in the right-hand sides of the triangular solver is protected by ABFT.

□

With this theorem, if failure occurs during triangular solving, lost data can be recovered when the triangular solver completes. Since matrix multiplication is also ABFT compatible, the whole red region in Figure 3.5 can be recovered after the entire trailing matrix update is done, leaving the opportunity for failure detection and recovery to be delayed at a convenient moment in the algorithm.

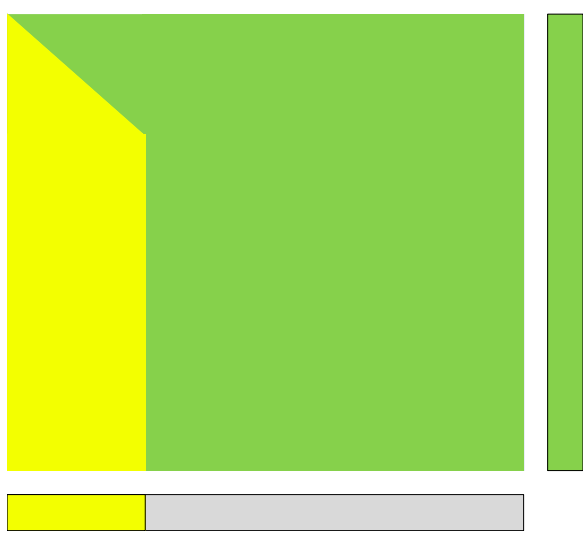


Figure 3.6: Separation of lower and upper areas protected by checksum (green) and checkpoint (yellow) during the course of the factorization algorithm

3.5 Protection of the Left Factor Matrix with Q-parallel Checkpoint

It has been proven in Theorem 3.4.2 that the checksum only covers the upper triangular part of the matrix until the current panel, and the trailing matrix is subject to future updates. This is depicted in Figure 3.6, where the green checksum on the right of the matrix protects exclusively the green part of the matrix. Another mechanism must be added for the protection of the left factor (the yellow area).

3.5.1 Impracticability of ABFT for Left Factor Protection

The most straightforward idea, when considering the need of protecting the lower triangle of the matrix, is to use an approach similar to the one described above, but column-wise. Unfortunately, such an approach is difficult, if not impossible in some cases, as proved in the remaining of this Section.

Pivoting and Vertical Checksum Validity

In LU, partial pivoting prevents the left factor from being protected through ABFT. The most immediate reason is as follow: The PBLAS kernel used to compute the panel factorization (see Figure 3.1) performs simultaneously the search for the best pivot in the column and the scaling of the column with that particular pivot. If applied directly on the matrix and the checksum blocks, similarly to what the trailing update approach does, checksum elements are at risk of being selected as pivots, which results in exchanging checksum rows into the matrix. This difficulty could be circumvented by introducing a new PBLAS kernel that does not search for pivots in the checksum.

Unfortunately, legitimate pivoting would still break the checksum invariant property, due to row swapping. In LU, for matrix A,

$$
\begin{aligned}
A &= \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} = \begin{pmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{pmatrix} \begin{pmatrix} U_{11} & U_{12} \\ 0 & U_{22} \end{pmatrix} \\
&= \begin{pmatrix} L_{11}U_{11} & L_{11}U_{12} \\ L_{21}U_{11} & L_{21}U_{12} + L_{22}U_{22} \end{pmatrix}
\end{aligned}
$$

Panel factorization is:

$$
\begin{pmatrix} A_{11} \\ A_{21} \end{pmatrix} = \begin{pmatrix} L_{11}U_{11} \\ L_{21}U_{11} \end{pmatrix} = \begin{pmatrix} L_{11} \\ L_{21} \end{pmatrix} U_{11}
$$

To protect L_{11} and L_{21}, imagine that we maintain a separate checksum, stored at the bottom of the matrix, as shown in the yellow bottom rectangle of Figure 3.6, that we plan on updating by scaling it accordingly to the panel operation. In this vertical checksum, each P tall group of blocks in the 2D block cyclic distribution is protected by a particular checksum block. Suppose rows i_1 and i_2 reside on blocks k_{i_1} and k_{j_1} of two processes. It is not unusual that $k_{i_1} \neq k_{j_1}$. By Corollary 3.4.4, block k_{i_1} and k_{j_1} contribute to column-wise checksum block k_{i_1} and k_{j_1} respectively in the column that local blocks k_{i_1} and k_{j_1} belong to. This relationship is expressed as

$$
\begin{aligned}
\textit{row } i_1 &\mapsto \textit{checksum block } k_{i_1} \\
\textit{row } j_1 &\mapsto \textit{checksum block } k_{j_1}
\end{aligned}
$$

$\mapsto$ reads 'contributes to'. After the swapping, the relationship should be updated to

$$
\begin{aligned}
\textit{row } i_1 &\mapsto \textit{checksum block } k_{j_1} \\
\textit{row } j_1 &\mapsto \textit{checksum block } k_{i_1}
\end{aligned}
$$

This requires a re-generation of checksum blocks k_{i_1} and k_{j_1} in order to maintain the checkpoint validity. Considering there are nb potential pivoting operations per panel, hence a maximum of $nb+1$ checksum blocks to discard, this operation has the potential to be as expensive as computing a complete vertical checkpoint.

QR Factorization

Although QR has no pivoting, it still cannot benefit from ABFT to cover Q, as we prove below.

Theorem 3.5.1. *Q in Householder QR factorization cannot be protected by performing factorization along with the vertical checksum.*

Proof. Append a $m \times n$ nonsingular matrix A with checksum GA of size $c \times n$ along the column direction to get matrix $A_c = \begin{bmatrix} A \\ GA \end{bmatrix}$. G is $c \times m$ generator matrix. Suppose A has a QR factorization $Q_0 R_0$.

Perform QR factorization to A_c:

$$\begin{bmatrix} A \\ GA \end{bmatrix} = Q_c R_c = \begin{bmatrix} Q_{c11} & Q_{c12} \\ Q_{c21} & Q_{c22} \end{bmatrix} \begin{bmatrix} R_{c11} \\ \varnothing \end{bmatrix}$$

Q_{c11} is $m \times m$ and Q_{c21} is $c \times m$. R_c is $m \times n$ and $\varnothing$ represents $c \times n$ zero matrix. $R_c \neq 0$ and is full rank. Because R_c is upper triangular with nonzero diagonal elements and therefore nonsingular.

$$Q_c Q_c^T = \begin{bmatrix} Q_{c11} & Q_{c12} \\ Q_{c21} & Q_{c22} \end{bmatrix} \begin{bmatrix} Q_{c11}^T & Q_{c21}^T \\ Q_{c12}^T & Q_{c22}^T \end{bmatrix} = I$$

Therefore

$$Q_{c11} Q_{c11}^T + Q_{c12} Q_{c12}^T = I. \tag{3.5}$$

Since $A = Q_{c11}R_{c11}$ and R_{c11} is nonsingular, then $Q_{c11} \neq 0$ and nonsingular.

Assume $Q_{c12} = 0$:

$Q_{c11}Q_{c21}^T + Q_{c12}Q_{c22}^T = 0$, therefore $Q_{c11}Q_{c21}^T = 0$. We have shown that Q_{c11} is nonsingular, so $Q_{c21}^T = 0$ and this conflicts with $GA = Q_{c21}R_{c11} \neq 0$, so the assumption $Q_{c12} = 0$ does not hold. From (3.5), $Q_{c11}Q_{c11}^T \neq I$. This means even though $A = Q_{c11}R_{c11}$, $Q_{c11}R_{c11}$ is not a QR factorization of A. □

3.5.2 Panel Checkpointing

Given that the ZU factorization cannot protect Z by applying ABFT in the same way as for U, separate efforts are needed. For the rest of this chapter, we use the term "checksum" to refer to the ABFT checksum, generated before the factorization, that is maintained by the application of numerical updates during the course of the algorithm, in contrast to "checkpointing" for the operation that creates a new protection block during the course of the factorization. LU factorization with partial pivoting being the most complex problem, it is used here for the discussion. The method proposed in this section can be applied to the QR and Cholesky factorizations with minimal efforts nonetheless.

In a ZU block factorization using 2D cyclic distribution, once a panel of Z is generated, it is stored into the lower triangular region of the original matrix. For example, in LU, vectors of L, except the diagonal ones, are stored in L. These lower triangular parts from the panel factorization are final results, and are not subject to further updates during the course of the algorithm, except for partial pivoting row swapping in LU. Therefore only one vertical checkpointing *"should be"* necessary to maintain each panel's safety, as is discussed in [40]. We will show how this idea, while mathematically trivial, needs to be refined to support partial pivoting. We will then propose a novel checkpointing scheme, leveraging properties of the block algorithm to checkpoint Z in parallel, that demonstrates a much lower overhead when compared to this basic approach.

3.5.3 Postponed Left Pivoting

Although once a panel is factored, it is not changed until the end of the computation, row swaps incurred by pivoting are still to be applied to the left factor as the algorithm progresses in the trailing matrix, as illustrated in Figure 3.1. The second step (pivoting to the left) swaps two rows to the left of the current panel. The same reasoning as presented in section 3.5.1 holds, meaning that the application of pivoting row swaps to the left factor has the potential to invalidate checkpoint blocks. Since pivoting to the left is carried out in every step of LU, this causes significant checkpoint maintenance overhead.

Unlike pivoting to the right, which happens during updates and inside the panel operation, whose result are reused in following steps of the algorithm, pivoting to the left can be postponed. The factored L is stored in the lower triangular part of the matrix without further usage during the algorithm. As a consequence, we delay the application of all left pivoting to the end of the computation, in order to avoid expensive checkpoint management. We keep track of all pivoting that should have been applied to the left factor, and when the algorithm has completed, all row swaps are applied just in time before returning the end-result of the routine.

3.5.4 Q-Parallel Checkpointing of Z

The vertical checkpointing of the panel result requires a set of reduction operations immediately after each panel factorization. Panel factorization is on the critical path and has lower parallelism, compared to other routines of the factorization (such as trailing matrix update). The panel factorization works only on a single block column of the matrix, hence benefits from only a P degree of parallelism, in a $P \times Q$ process grid. Checkpointing worsens this situation, because it applies to the same block column, and is bound to the same low level of exploitable parallelism. Furthermore, the checkpointing cannot be overlapped with the computation of the trailing matrix update: all processes who do not appear on the same column of the process grid are

waiting in the matrix-matrix multiply PBLAS, stalled because they require the panel column to enter the call in order for the result of the panel to be broadcasted. If the algorithm enters the checkpointing routine before going into the trailing update routine, the entire update is delayed. If the algorithm enters the trailing update before starting the checkpointing, the checksum is damaged in a way that prevents recovering that panel, leaving it vulnerable to failures.

Our proposition is then twofold: we protect the content of the blocks before the panel, which then enables starting immediately the trailing update without jeopardizing the safety of the panel result. Then, we wait until sufficient checkpointing is pending to benefit from the maximal parallelism allowed by the process grid.

Enabling Trailing Matrix Update Before Checkpointing

The major problem with enabling the trailing matrix update to proceed while the checkpointing of the panel is not finished is that the ABFT protection of the update modifies the checksum in a way that disables protection for the panel blocks. To circumvent this limitation, in a $P \times Q$ grid, processes are grouped by section of width Q, that are called a *panel scope*. When the panel operation starts applying to a new section, the processes of this panel scope make a local copy of the impending column and the associated checksum, called a *snapshot*. This operation involves no communication, and features the maximum $P \times Q$ parallelism. The memory overhead is limited, as it requires only the space for at most two extra columns to be available at all time, one for saving the state before the application of the panel to the target column, and one for the checksum column associated to these Q columns. The algorithm then proceeds as usual, without waiting for checkpoints before entering the next Q trailing updates. Because of the availability of this extra protection column, the original checksum can be modified to protect the trailing matrix without threatening the recovery of the panel scope, which can rollback to that previous dataset should a failure occur.

Q-Parallel Checkpointing

When a panel scope is completed, the $P \times Q$ group of processes undergo checkpointing simultaneously. Effectively, P simultaneous checkpointing reductions are taking place along the block rows, involving the Q processes of that row to generate a new protection block. This scheme enables the maximum parallelism for the checkpoint operation, hence decreasing its global impact on the failure free overhead. Another strong benefit is that it scales with the process grid perfectly, whereas regular checkpointing suffers from scaling with the square root of the number of processes (as it involves only one dimension of the process grid).

Optimized Checkpoint Storage

According to Corollary 3.4.4, starting from the first block column on the left, every Q block columns contribute to one block column of checksum, which means that once the factorization is done for these Q block columns, the corresponding checksum block column becomes useless (it does not protect the trailing matrix anymore, it has never protected the left factor, see Theorem 3.4.2). Therefore, this checksum storage space is available for storing the resultant checkpoint block generated to protect the panel result. Following the same policy as the checksum storage, discussed in Section 3.4.5, the checkpoint data is stored in reverse order from the right of the checksum (see Figure 3.4). As this part of the checksum is excluded from the trailing matrix update, the checkpoint blocks are not modified by the continued operation of the algorithm.

Recovery

The hybrid checkpointing approach requires a special recovery algorithm. Two cases are considered. First, when failure strikes during the trailing update, immediately after a panel scope checkpointing. For this case, the recovery is not attempted until the current step of the trailing update is done. When the recovery time comes, the checksum/checkpointing on the right of the matrix matches the matrix data as if the

initial ABFT checksum had just been performed. Therefore any lost data blocks can be recovered by the simple reverse application of the ABFT checksum relationship.

The second case is when a failure occurs during the Q panel factorization, before the checkpointing for this panel scope can successfully finish. In this situation, all processes revert the panel scope columns to the snapshot copy. Holes in the snapshot data are recreated by using the snapshot copy of the checksum, applying the usual ABFT recovery. The algorithm is resumed in the panel scope, so that panel and updates are applied again within the scope of the Q wide section; updates outside the panel scope are discarded, until the pre-failure iteration has been reached. Outside the panel scope, regular recovery mechanisms are deployed (ABFT checksum inversion for the trailing matrix, checkpoint recovery for the left factor). When the re-factorization of panels finishes, the entire matrix, including the checksum, is recovered back to the correct state. The computation then resumes from the next panel factorization, after the failing step.

Figure 3.7 shows an example of the recovery when the process (1,0) in a 2×3 grid failed. It presents the difference between the correct matrix dataset and the current dataset during various steps of failure recovery as a "temperature map", brighter colors meaning large differences and black insignificant differences. The matrix size is 80×80 and $NB = 10$, therefore the checksum size is 80×60. Failure occurs after the panel factorization starting at (41,41) is completed, within the $Q = 3$ panel scope. First, using a fault tolerant MPI infrastructures, like FT-MPI [57], the failed process (0,1) is replaced and reintegrates the process grid with a blank dataset, showing as evenly distributed erroneous blocks (A). Then the recovery process starts by mending the checksum using duplicates (B). The next step recovers the data which is outside the current panel scope (31:80,31:60), using the corresponding checksum for the right factor, and the checkpoints for the left factor (C). At this moment, all the erroneous blocks are repaired, except those in the panel scope (41:80, 41:50). Snapshots are applied to the three columns of the panel scope (31:80,31:60). Since these do not match the state of the matrix before the failure, but a previous state, this area

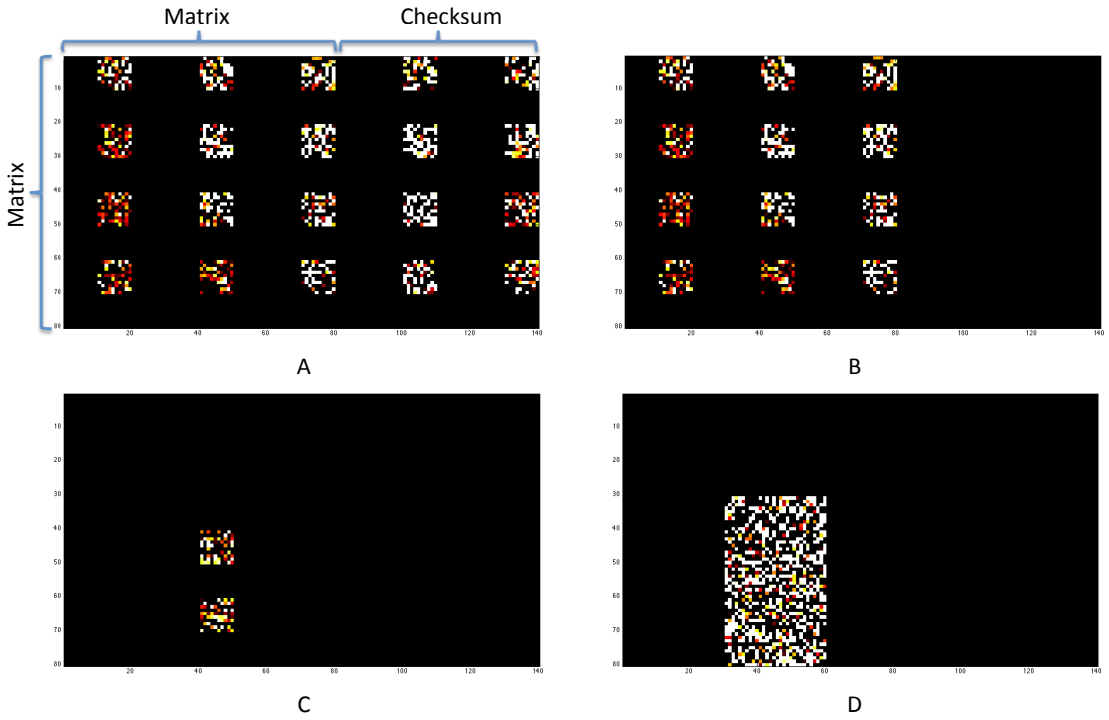


Figure 3.7: Recovery example
(matrix size 800×800, grid size 2×3, failure of process (0,1), failure step:41,
A: Failure occurs B: Checksum recovered
C: Data recovered using ABFT checksum and checkpointing output D: Three panels restored using snapshots

appears as very different (D). Panel factorization is re-launched in the panel scope, in the area (31:80,31:60), with the trailing update limited within this area. This re-factorization continues until it finishes panel (41:80,41:50) and by that time the whole matrix is recovered to the correct state (not presented, all black). The LU factorization can then proceed normally.

3.6 On-Demand Checkpointing using the Checkpoint-on-Failure Protocol

There are two critical requirements for a successful deployment of the fault tolerance algorithm described in this chapter so far. One is a supporting MPI system that allows returning the execution control to the application, and the other is the recovery of the running stack of the failed process to coordinate with the survived processes to restart the execution. The previous works by others and this chapter *assume* that a "high quality implementation" of MPI exists. At the time of failure, this MPI return the execution control to the application and provide failure information such as the rank of the failed process. Unfortunately, the current MPI-2 standard [131] addresses this as an optional features without providing significant help to deal with the required type of behavior. For the current standard, process or communication failures are to be handled as errors, and the behavior of the MPI application, after an error has been returned, is left unspecified by the standard. Most of the implementations of the MPI Standard have taken the path of considering process failures as unrecoverable errors, and the processes of the application are most often killed by the runtime system when a failure hits any of them, leaving no opportunity for the user to mitigate the impact of failures. Some efforts have been undertaken to enable ABFT support in MPI. FT-MPI [57] was an MPI-1 implementation which proposed to change the MPI semantic to enable repairing communicators, thus re-enabling communications for applications damaged by failures. This approach has proven successful and applications have been implemented using FT-MPI. However, these modifications were not adopted by the MPI standardization body, and the resulting lack of portability undermined user adoption for this fault tolerant solution. In [71], the authors discuss alternative or slightly modified interpretations of the MPI standard that enable some forms of fault tolerance. One essential idea is that process failures happening in another MPI world, connected only through an inter-communicator, should not prevent the continuation

of normal operations. The complexity of this approach, for both the implementation and users, has prevented these ideas from having a practical impact.

In [16], a Checkpoint-on-Failure (CoF) Protocol is proposed to handle the issue of MPI support and the recovery of running stack. The core idea is to only perform checkpointing at the time of failure in an on-demand fashion. In the CoF approach, the only requirement from the MPI implementation is that it does not forcibly kill the living processes without returning control. No stronger support from the MPI stack is required, and the state of the library is left undefined.

To demonstrate the effectiveness of the CoF fault-tolerance mechanism, this section integrates CoF with the QR factorization implementation described in the earlier sections of this chapter. While details of the modification to MPI can be found in [16], in this section we focus on the QR factorization, especially the situation where failure occurs during lower level routines such as PDLARFB is addressed. Such situation has been missed in most of the related work in the area for the complexity it introduces.

3.6.1 QR factorization on Distributed Memory System

For an $M \times N$ matrix A, QR factorization produces Q and R, such that $A = QR$ and Q is an $M \times M$ orthogonal matrix and R is an $M \times N$ upper triangular matrix. For simplicity of expression, we use a square matrix $M \times M$ in this chapter, but the result applies also to rectangular matrices. There are several methods for computing the QR factorization, such as the Gram-Schmidt process, the Householder transformations, and the Givens rotations. ScaLAPACK uses a block version of the QR factorization by accumulating a few steps of the Householder matrix. This method is rich in level 3 BLAS operations and therefore can achieve high performance. Q is stored under the lower diagonal of the input matrix in the form of a WY representation of the Householder transformation products[14, 116].

ScaLAPACK implements the block QR factorization as follow. At step i , an $m \times m$ submatrix A_i is partitioned and factorized as

$$A_i = \begin{bmatrix} A_1 & A_2 \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} = Q \times \begin{bmatrix} R_{11} & R_{12} \\ 0 & R_{22} \end{bmatrix}$$

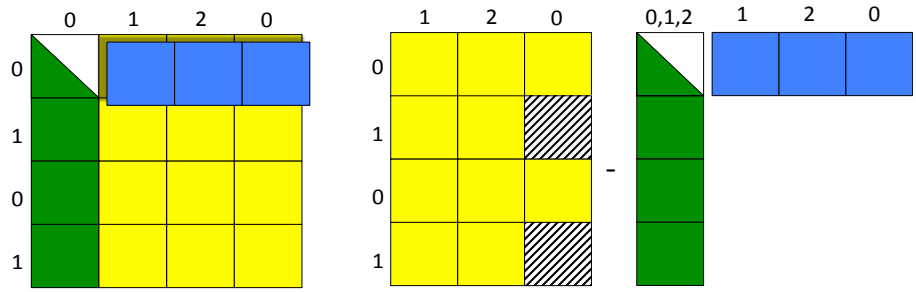
Here A_{11} is of size $nb \times nb$, where nb is called the block size. A_{21} is of size $(m - nb) \times nb$. $A_1 = [A_{11}, A_{12}]^T$ constitutes the area for the panel QR factorization. Since ScaLAPACK uses the Householder method, Q is expressed as a series of Householder transformations in the form $H_i = I - \tau_i v_i v_i^T, i = 1 \cdots nb$. v_i has 0 for the first $i - 1$ entries, 1 on the $i - th$ entry and $\tau_i = 2/v_i^T v_i$. In ScaLAPACK, v_i is stored below the diagonal of A and when Q is applied to the trailing matrix $A_2 = [A_{21}, A_{22}]^T$, Q is computed by $Q = H_1 \cdots H_{nb} = I - VTV^T$, where T is an upper triangular matrix of size $nb \times nb$ and V has v_i as its $i - th$ column. With this expression, the trailing matrix update becomes

$$\tilde{A}_2 = \begin{bmatrix} \tilde{A}_{12} \\ \tilde{A}_{22} \end{bmatrix} = Q^T A_2 = (I - VT^T V^T) A_2 \tag{3.6}$$

This finishes one iteration of the block QR factorization. This process is repeated from $\tilde{A}_{22}$ until the whole matrix is factorized.

3.6.2 Failure in PBLAS routines

An important condition for the effectiveness of ABFT is the completion of the current iteration. When a failure interrupts the program execution during an iteration, the checksum ends up in intermediate form and as a result cannot be used for recovery. This problem worsens when a failure occurs during a lower level routine, like a PBLAS, causing a partial trailing matrix update. In this case, updates have been applied to parts of the dataset, possibly without having updated accordingly the corresponding checksums. In the case of the QR algorithm, this problem is solved by saving the



(a) After the first step: $A_2^T V$ (b) $A_2 - V\tilde{W}^T$

Figure 3.8: PDLARFB

local state when a failure is detected in PDLARFB rather than in PDGEQRF. The recovery process in this case is described as follow.

As shown in (3.6), the trailing update of QR carries out operation $Q^T A_2 = (I - VT^T V^T)A_2 \rightarrow \tilde{A}_2$. The right arrow means the updated trailing matrix replaces the content of A_2. The trailing matrix update of QR is similar to PDGEMM for LU which has been shown to hold the checksum relationship only at the end. Therefore the procedure to recover from a failure in PDLARFB is:

1. Survived processes mark the progress and dump critical data to disk
2. After re-spawning, all processes dry-run to the failure point
3. All except the replacement process load checkpoint from disk
4. All processes resume computing from the failure point to the end of PDLARFB
5. At the exit of PDLARFB, recover all lost data in checksum and the whole matrix
6. Execution of PDGEQRF returns to normal

The 'dry-run' step is to re-establish the calling stack of all processes to the failing point. Therefore PBLAS and ScaLAPACK routines for computing, for example, PDGEQR2, PDLARFT, etc. are skipped over during the dry run.

The recovery is demonstrated with an example of a 4×4 blocks matrix on a 2×3 grid where failure occurs during PDLARFB.

PDLARFB implements $Q^T A_2 = (I - VT^T V^T)A_2$ in three steps:

1. $W \leftarrow V^T A_2$
2. $\tilde{W} \leftarrow T^T \times W$
3. $\tilde{A}_2 \leftarrow A_2 - V\tilde{W}$

Suppose the failure occurs right after step 1 on process (1,0). In step 1, as shown in figure 3.8(a), V is stored in the green trapezoid and A_2 is in the yellow blocks. V is first broadcast row-wise to all columns, then GEMM is called on each process that owns A_2 with the local V and A_2. Finally, the result is produced with column-wise block summation and the result is stored on the first row of processes that process the first row of A_2 (blue blocks). Offered by the MPI CoF modification presented in [16], the failure location is broadcasted to all surviving processes and matrix data are dumped to the disk, including peripheral data like the TAU array and workspace. Surviving processes also keep a record on whether they have finished the DGEMM in step 1.

After critical data is saved to disk, the program exits and is re-spawn with a replacement process in the failed process's location. The re-launched program dry-runs to the failure point in step 1 of PDLARFB. All previously surviving processes load their checkpoint from disk while the replacement process stays with its blank data. Then the program resumes execution of PDLARFB. Since failure is on process (1,0), W survives the data loss.

Step 2 of PDLARFB is $W \times T$ where W is the blue blocks in figure 3.8(a) and T is a $nb \times nb$ upper triangular matrix. Since T resides on each process in the row that owns W, the correctness of T can be always guaranteed, and therefore $\tilde{W}$ has no lost block in it after calling DTRMM. $\tilde{W}$ is broadcasted column-wise for step 3.

Step 3 of PDLARFB is shown in figure 3.8(b). In $\tilde{A}_2 \leftarrow A_2 - V\tilde{W}^T$, besides $\tilde{W}^T$, V is also correct since V has been broadcast row-wise to all process in step 1,

therefore even if blocks of V are destroyed by the failure, the result on the replacement process can be recovered from its neighbour processes in the same row. The result of step 3, also that of PDLARFB, is affected by the incorrect result in A_2, expressed in shadowed blocks in figure 3.8(b). These incorrect blocks remain in the result of DGEMM in this step. They are fixed later in the recovery process in PDGEQRF using both the ABFT and Q-parallel checksum.

For PDLARFB, both V and T can be guaranteed correct no matter when and where failure occurs, the only variable factor is W. However if the failure does punch holes in W, more shadow blocks appear in the result of PDLARFB, and they can still be fixed by the recovery in PDGEQRF.

3.7 Evaluation

In this section, we evaluate the performance of the proposed fault tolerant algorithm. For a fault tolerant algorithm, the most important consideration is the overhead added to failure free execution rate, due to various fault tolerance mechanisms such as checksum generation, checkpointing and extra flops. An efficient and scalable algorithm will incur a little overhead over the original algorithm while enabling scalable reconstruction of lost dataset in case of failure.

We use the NSF Kraken supercomputer, hosted at the National Institute for Computational Science (NICS, Oak Ridge, TN) as our testing platform. This machine features 112,896 2.6GHz AMD Opteron cores, 12 cores per node, with the Seastar interconnect. At the software level, to serve as a comparison base, we use the non fault tolerant ScaLAPACK LU and QR in double precision with block size $NB = 100$. The fault tolerance functions are implemented and inserted as drop-in replacements for ScaLAPACK routines.

In this section, we first evaluate the storage overhead in the form of extra memory usage, then show experimental result on Kraken to assess the computational overhead.

3.7.1 Storage Overhead

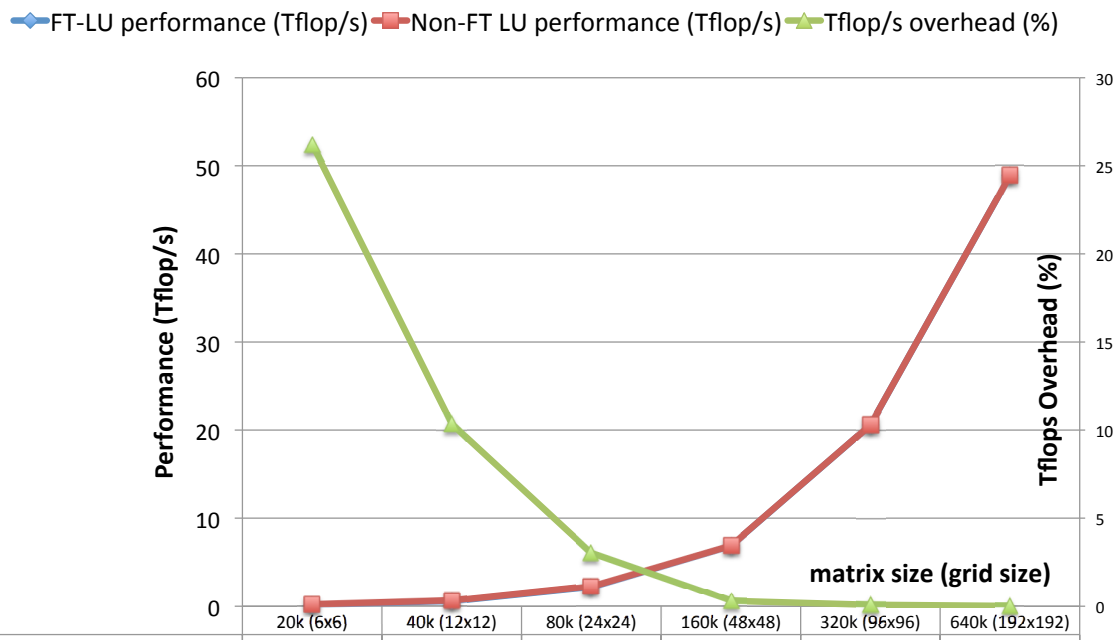
Checksum takes extra storage (memory), but on large scale systems, memory usage is usually maximized for computing tasks. Therefore, it is preferable to have a small ratio of checksum size over matrix size, in order to minimize the impact on the memory available to the application itself. For the sake of simplicity, and because of the small impact in term of memory usage, neither the pivoting vector nor the column shift are considered in this evaluation.

Different protection algorithms require different amounts of memory. In the following, we consider the duplication algorithm presented in Section 3.4.5 for computing the upper memory bound. The storage of the checksum includes the row-wise and column-wise checksums and a small portion at the bottom-right corner.

For an input matrix of size $M \times N$ on a $P \times Q$ process grid, the memory used for checksum (including duplicates) is $M \times \frac{N}{Q} \times 2$. The ratio R_{mem} of checksum memory over the memory of the input matrix, equals to $\frac{2}{Q}$, becomes negligible with the increase in the number of processes used for the computation.

3.7.2 Overhead without Failures

Figure 3.9 evaluates the completion time overhead and performance, using the LU factorization routine PDGETRF. The performance of both the original and fault tolerant version are presented, in Tflop/s (the two curves overlap due to the little performance difference). This experiment is carried out to test the weak scalability, where both the matrix and grid dimension doubles. The result outlines that as the problem size and grid size increases, the overhead drops quickly and eventually becomes negligible. At the matrix size of $640,000 \times 640,000$, on $36,864$ (192×192) cores, both versions achieved over 48Tflop/s, with an overhead of 0.016% for the ABFT algorithm. As a side experiment, we implemented the naive vertical checkpointing method discussed in section 3.5.2, and as expected the measured overhead quickly exceeds 100%.



	20k (6x6)	40k (12x12)	80k (24x24)	160k (48x48)	320k (96x96)	640k (192x192)
FT-LU performance (Tflop/s)	0.14236114	0.568567269	2.210963782	6.868980808	20.57332106	48.89869531
Non-FT LU performance (Tflop/s)	0.19290937	0.634258147	2.280367481	6.890269591	20.59102249	48.90650758
Tflop/s overhead (%)	26.20309733	10.35711999	3.043531342	0.308968803	0.085966728	0.015973871

Figure 3.9: Weak scalability of FT-LU: performance and overhead on Kraken, compared to non fault tolerant LU

FT overhead (Tflop/s)	0.051	0.066	0.070	0.021	0.018	0.008
FT overhead (%)	26.203	10.357	3.044	0.309	0.086	0.016

As the left factor is touched only once during the computation, the approach of checkpointing the result of a panel synchronously can, *a-priori*, look sound when compared to system based checkpoint, where the entire dataset is checkpointed periodically. However, as the checkpointing of a particular panel suffers from its inability to exploit the full parallelism of the platform, it is subject to a derivative of Amdahl's law, its parallel efficiency is bound by P, while the overall computation enjoys a $P \times Q$ parallel efficiency: its importance is bound to grow when the number of computing resources increases. As a consequence, in the experiments, the time to compute the naive checkpoint dominates the computation time. On the other hand, the hybrid checkpointing approach exchanges the risk of a Q-step rollback with the opportunity to benefit from a $P \times Q$ parallel efficiency for the panel checkpointing. Because of this improved parallel efficiency, the hybrid checkpointing

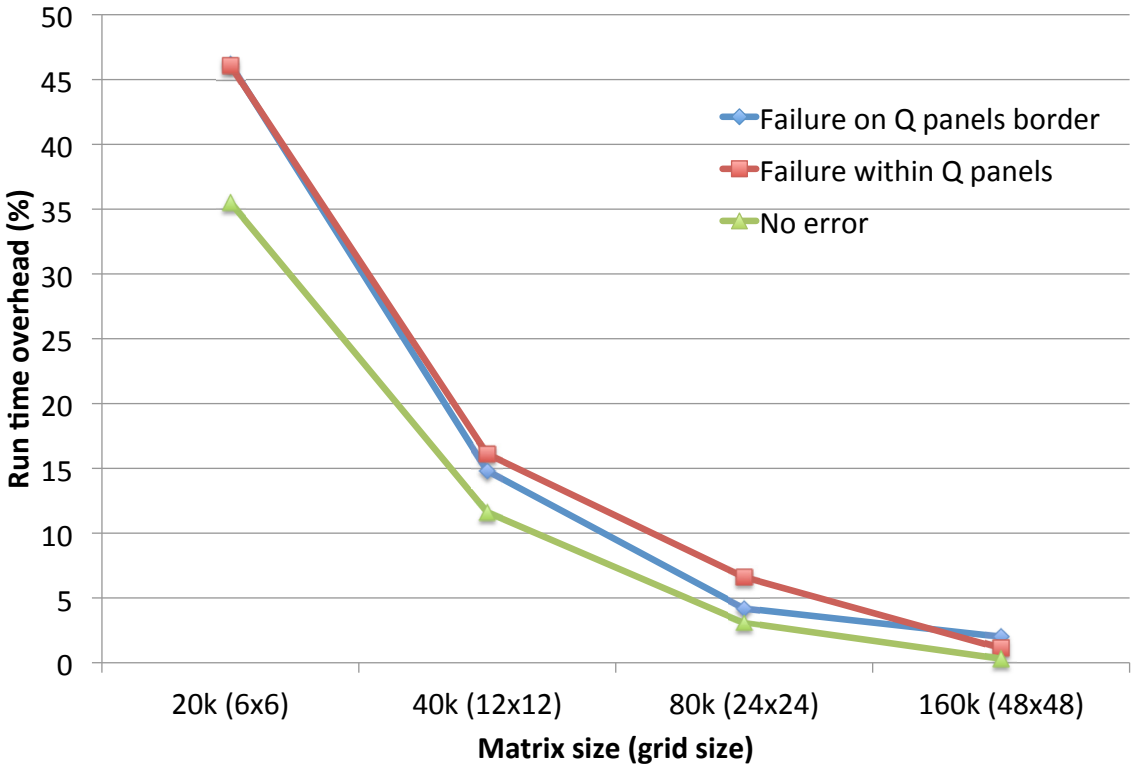


Figure 3.10: Weak scalability of FT-LU: run time overhead on Kraken when failures strike at different steps

approach benefits from a competitive level of performance, that follows the same trend as the original non fault tolerant algorithm.

3.7.3 Recovery Cost

In addition to the "curb" overhead of fault tolerance functions, the recovery from failure adds extra overhead to the host algorithm. There are two cases for the recovery. The first one is when failure occurs right after the reverse neighboring checkpointing of Q panels. At this moment the matrix is well protected by the checksum and therefore the lost data can be recovered directly from the checksum. We refer to this case as "failure on Q panels border". The second case is when the failure occurs during the reverse neighboring checkpointing and therefore local snapshots have to be used along

with re-factorization to recover the lost data and restore the matrix state. This is referred to as the "failure within Q panels".

Figure 3.10 shows the overhead from these two cases for the LU factorization, along with the no-error overhead as a reference. In the "border" case, the failure is simulated to strike when the 96th panel (which is a multiple of grid columns, $6, 12, \cdots, 48$) has just finished. In the "non-border" case, failure occurs during the $(Q+2)^{th}$ panel factorization. For example, when $Q = 12$, the failure is injected when the trailing update for the step with panel (1301,1301) finishes. From the result in Figure 3.10, the recovery procedure in both cases adds a small overhead that also decreases when scaled to large problem size and process grid. For largest setups, only 2-3 percent of the execution time is spent recovering from a failure.

3.7.4 Extension to Other factorization

The algorithm proposed in this chapter can be applied to a wide range of dense matrix factorizations other than LU. As a demonstration we have extended the fault tolerance functions to the ScaLAPACK QR factorization in double precision. Since QR uses a block algorithm similar to LU (and also similar to Cholesky), the integration of fault tolerance functions is mostly straightforward. Figure 3.11 shows the performance of QR with and without recovery. The overhead drops as the problem and grid size increase, although it remains higher than that of LU for the same problem size. This is expected: as the QR algorithm has a higher complexity than LU ($\frac{4}{3}N^3$ v.s. $\frac{2}{3}N^3$), the ABFT approach incurs more extra computation when updating checksums. Similar to the LU result, recovery adds an extra 2% overhead. At size 160,000 a failure incurs about 5.7% overhead to be recovered. This overhead becomes lower, the larger the problem or processor grid size considered.

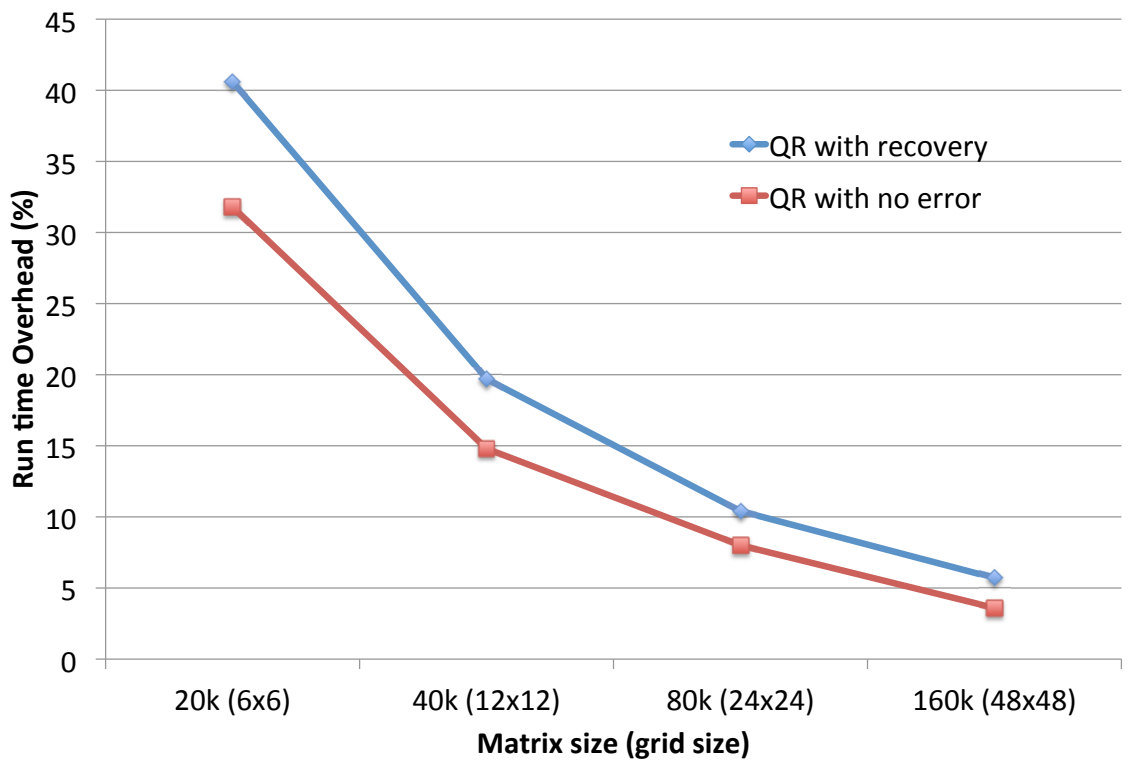


Figure 3.11: Weak scalability of FT-QR: run time overhead on Kraken when failures strike

3.7.5 Checkpointing-on-Failure for QR

The CoF QR algorithm checkpoints data from memory to disk on the living processes at the time of failure. Therefore disk I/O access time is a critical component of the performance overhead.

To evaluate the performance impact of disk access, the implementation of the CoF algorithm based on the ScaLAPACK QR is tested on two cluster systems at different scale. The first machine, "Dancer", is a 16-node cluster at the University of Tennessee, Knoxville. All nodes are equipped with two 2.27GHz quad-core Intel E5520 CPUs, connected by 20GB/s Infiniband. Solid State Drive disks are used as the checkpoint storage media. The second system is the Kraken supercomputer by Cray Inc. at the Oak Ridge National Lab. Kraken has 9,408 compute nodes. Each node has two Istanbul 2.6 GHz six-core AMD Opteron processors, 16 GB of memory,

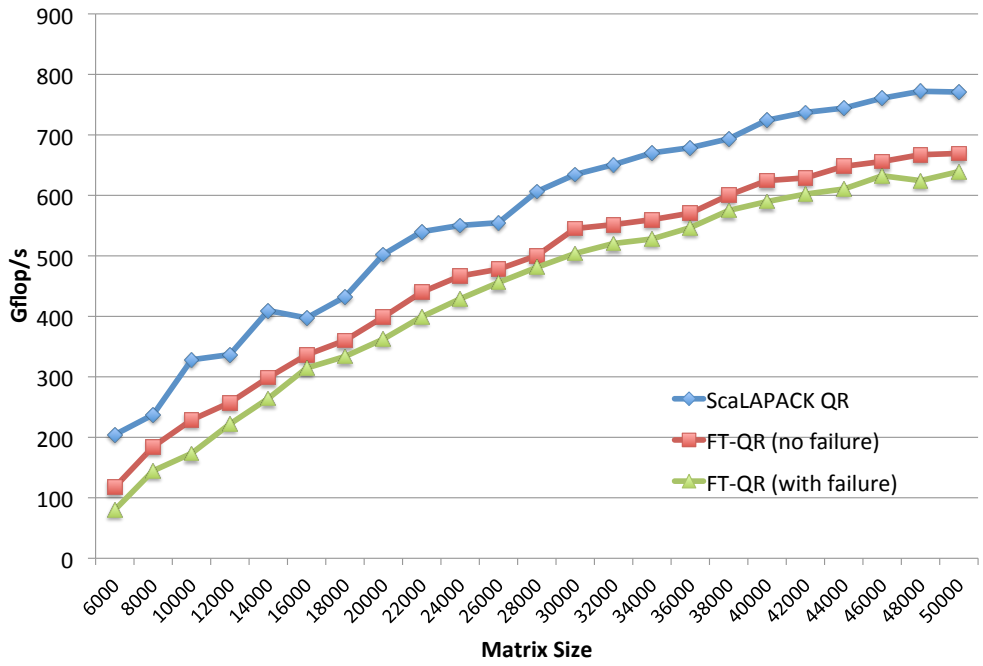


Figure 3.12: Performance on Dancer (16×8 grid)

and a highly scalable cluster file system "Lustre". All the nodes are Connected by the Cray SeaStar2+ interconnect. In all experiments, the block size is set to 100.

Figure 3.12 presents the performance of this QR implementation on the Dancer cluster with a 8×16 process grid. The FT-QR (no failure) presents the performance of the On-Demand Checkpointing implementation, in a fault-free execution, while the FT-QR (with failure) curves present the performance of the same implemenation, when the failure is injected after the first step of PDLARFB that performs $W \leftarrow V^T A_2$. The performance of the non-fault tolerant ScaLAPACK QR is also presented to serve as a reference.

The difference with the ScaLAPACK QR is caused by the parallel-Q checksum and the ABFT algorithm. This overhead has been shown to scale down with larger number of processes and matrices. In the case of a run with an error, the following overheads adds up: the times to store the checkpoint to disk, re-launch an application, re-establish the position of all processes by dry running in the application until the

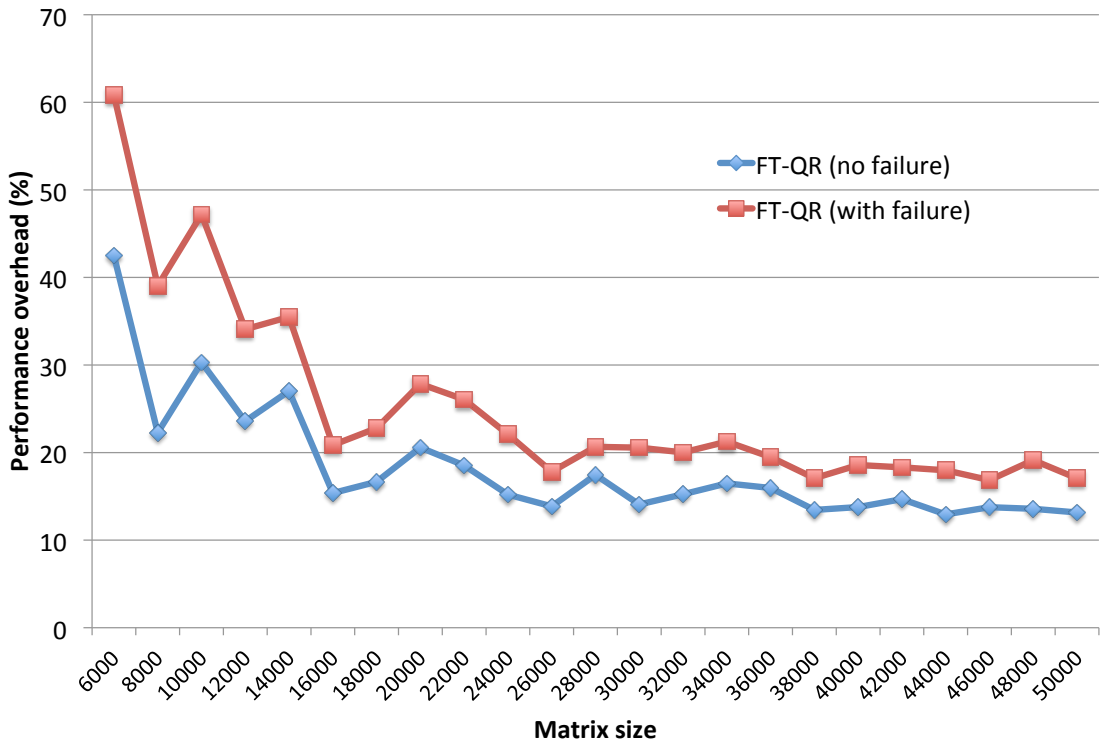


Figure 3.13: Overhead over ScaLAPACK QR on Dancer (16×8 grid)

failing point, loading checkpoint from disk and perform the ABFT recovery using the checksum found in the checkpoint of the previously living processes.

On Dancer, the performance of QR with on-demand checkpointing and recovery follows closely with the "no failure" performance. Figure 3.13 shows that as the matrix size increases, the recovery overhead falls below 5% more than the "no failure" overhead. By breaking down the run-time of each recovery elements, Figure 3.14 shows that checkpoint saving and loading only take a small percentage of the total run-time. On a problem of this size, the additional overheads are dominated by the time it takes to terminate the failing MPI application and relaunch a new one. Other than the fast solid state drive disks, the fast checkpointing can also be attributed to the disk cache provided by the OS. Since loading is performed immediately after saving, high disk cache hits can largely speed up the process. After matrix size 44,000 the memory usage on each node came close to limit and since no swap space is available on the Dancer cluster, disk cache support started to decrease and cause slight

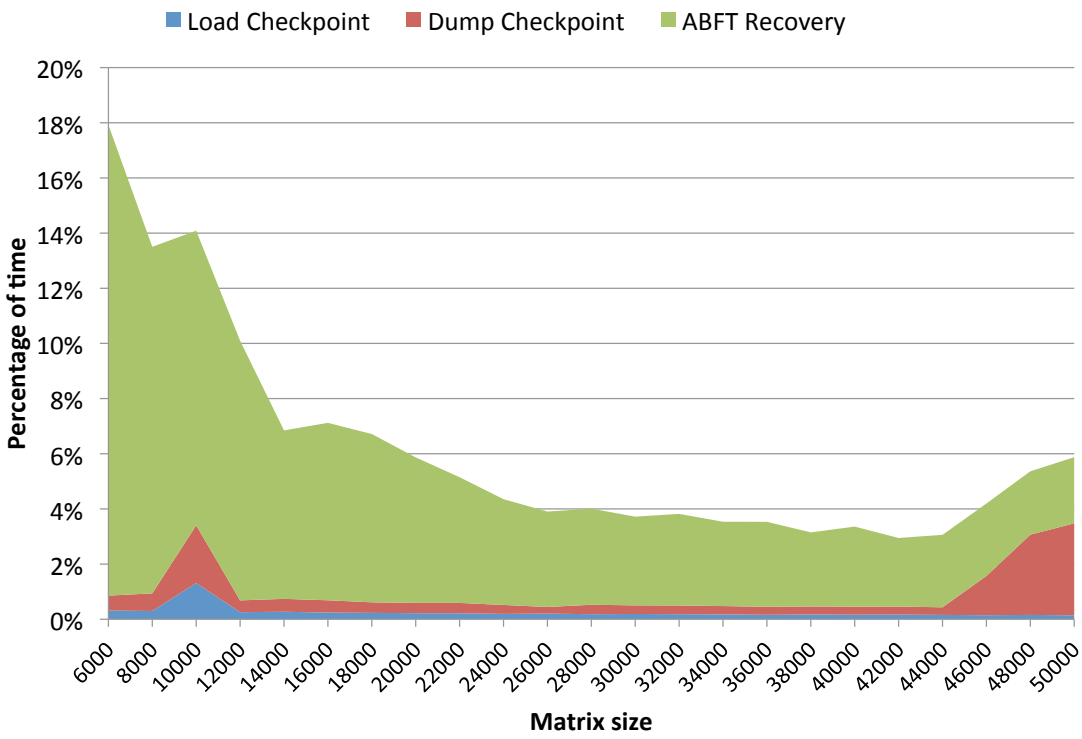


Figure 3.14: Time Breakdown of FT-QR on Dancer (16×8 grid)

increase in disk access time, which however does not affect the overhead percentage from performing recovery.

Figure 3.15 presents the performance on Kraken with a larger grid and a different filesystem to store the checkpoint images. A similar effect of a small checkpointing saving and loading time is observed. The performance of the "with failure" case shows the same trend of closely following the "no failure" case performance. At size matrix 100,000 for instance, FT-QR successfully recovered from the failure and achieved 2.86 Tflop/s, which is 90% of the performance of the ScaLAPACK QR. This verified that the On-Demand Checkpointing QR also performs well at larger scales.

3.8 Conclusion

In this chapter, by assuming a failure model in which fail-stop failures can occur anytime on any process during a parallel execution, a general scheme of ABFT

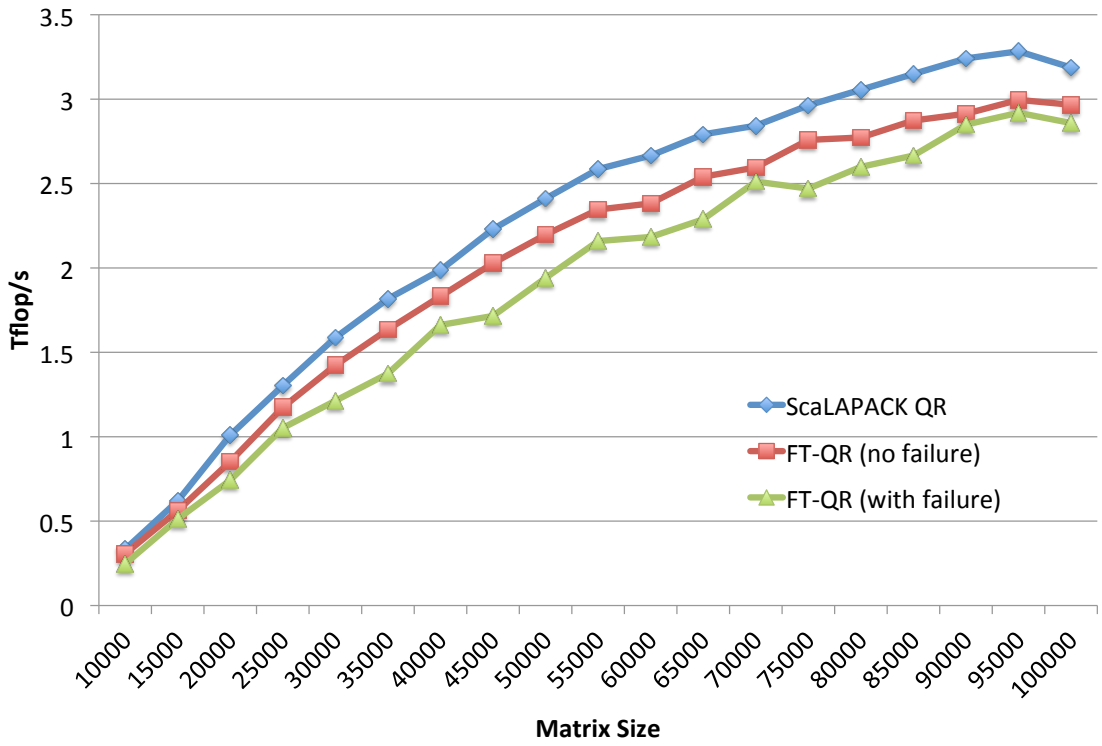


Figure 3.15: Performance on Kraken (24×24 grid)

algorithms for protecting one-sided matrix factorizations is proposed. This scheme can be applied to a wide range of dense matrix factorizations, including Cholesky, LU and QR. A significant property of the proposed algorithms is that both the left and right factorization results are protected. ABFT is used to protect the right factor with checksum generated before, and carried along during the factorizations. A highly scalable checkpointing method is proposed to protect the left factor. This method cooperatively reutilizes the memory space originally designed to store the ABFT checksum, and has minimal overhead by strategically coalescing checkpoints of many iterations. In addition, a Checkpointing-on-Failure scheme is proposed to help the recovery of the execution under the situation that no support is officially available from the MPI standard. By integrating the minimal support in the MPI system, disk-based checkpointing is only performed at the time of failure. Execution stack and matrix data are later recovered from ABFT checksum and both disk- and diskless-checkpoint. Large scale experimental results validate the design of the proposed fault

tolerance method by highlighting scalable performance and decreasing overhead for both LU and QR.

Chapter 4

Soft Error Resilience on Distrbuted Memory System

4.1 Introduction

Soft errors, normally in the form of bit flips, are events in microelectronic circuit that result in transient error without permanently damaging the device. They corrupt computed data, and produce erroneous results without leaving a trace. High-end computer systems are especially susceptible to such errors due to the ever increasing chip density and system scale. Between 2003 and 2004, the 2048-node ASC Q supercomputer for scientific computing in Los Alamos National Laboratorys experienced failure from extensive soft errors [100]. By comparing the error logs with a radiation experiment conducted in a lab, the cause was later identified to be the cosmic ray striking its parity-protected cache tag array. The Q computer is more vulnerable to soft errors because it is located at about 7500 feet above the sea level, and the neutrons from cosmic-rays are roughly 6.4 times stronger than the ones occuring at sea level. A similar incident has also appeared in a commercial computing system from Sun Microsystems that caused outages for many of its customers due to

cosmic ray soft errors [92]. These incidents signify that soft errors are a real issue that both hardware and software developers must face.

Soft error rate (SER) in memory is usually quantified using FIT (failure in time) per MB, 1 FIT is 1 failure per 10^9 operation hours per 10^6 bits. Google has reported between 778 and 25,000 FIT from errors in the DRAMs of their server fleet, an order of magnitude higher than previously expected [118]. As CMOS technology scales the feature size down with more transistors per chip and lower critical charge [75, 138], the threat of soft errors will continue to haunt the computing community.

The three main sources of soft errors are alpha particles, high energy neutrons, and thermal neutron flux. Threatening alpha particles primarily originate from memory chip packages. From alpha particles, two or three atoms of uranium or thorium in a contaminated package can already flip a bit [29]. Even though newer technology and material can to some extent mitigate the impact of alpha particles, recent studies have shown that with the scaling in CMOS circuit, soft error rate (SER) increases when the critical charge is lowered. For instance, SER at 0.3V is eight times higher than SER at 1.0V [63]. High energy neutrons from cosmic rays are the dominant cosmic ray products that cause soft errors [145]. Neutrons can penetrate most man-made construction, for example, five feet of concrete [97]. According to [75], CMOS scaling does not increase the SER from neutrons, the factor of the fast increase of system capacity in the sense of more CPU cores and memories also validate concern for the neutrons induced soft errors. Thermal neutrons are not as problematic as the other two sources and are mostly related to high energy neutron flux and materials in the neighboring environment [44].

In order to mitigate the impact of soft errors, modern HPC systems rely heavily on ECC (error correcting code). Nowadays the most commonly used ECC is SECDED (Single Error Correction, Double Error Detection). For multi-bit errors precipitated by the progress of the integrated circuit technology [132], a more powerful form of ECC has become too expensive due to the higher encoding and decoding overhead and the resulting memory performance loss. Question has been raised on whether soft

error resilience can be achieved with less cost from the application side [70]. Among HPC applications that could benefit from such fault tolerance capability, dense linear algebra applications such as the HPL benchmark for the TOP500 competition [99] and the AORSA fusion energy simulation program [11], are representative examples. These applications normally involve solving a dense system of equations of the form $Ax = b$ on large scale HPC systems with matrix sizes of A as large as 500,000. Soft errors that occur during such long running applications produce incorrect solution. This lowers productivity by wasting valuable time and energy in error tracing with little chance of locating the error.

Until now, most of the soft error resilience techniques for dense linear solvers are limited to small scale computing installations, such as on systolic arrays, assuming that the error correcting code does not seriously affect system performance and the encoding can be carried out with exact arithmetic [61, 89]. Unfortunately, none of these assumptions hold true for today's Pflop/s supercomputer systems. In [51], we have demonstrated the first attempt to take on the challenge of recovering the solution from a dense linear system solver of $Ax = b$ with a single error occurrence in both L and U of the LU factorization. This section develops a multiple soft errors resilience mechanism which could potentially be a more performance friendly alternative to the complex hardware ECC. The proposed algorithms consider the spatial and temporal multiple errors. Spatial soft errors occur at different time, whereas temporal soft errors manifest as simultaneous multiple bit flips in disparate locations. Experiments on the Kraken supercomputer from Cray at the University of Tennessee verified our design for both the error detection and correction capability as well as low performance complexity. The proposed method may also be extended to other one-sided factorizations for the recovery of linear system solution and factorization matrices.

The rest of the section is organized as follows. Section 4.2 introduces an LU based dense linear solver on distributed memory system. The impact of soft error on the linear solver is then analyzed and the general work flow of the proposed soft

error resilience algorithm is shown in Section 4.3. Sections 4.4 and 4.5 develop the protection method for both the left factor L and right factor U. Section 4.6 proposes a block protection method to reduce the computational complexity of the non-blocking protection algorithm for U. Finally, the recovery algorithm is discussed in Section 4.7 and the experimental results are shown in Section 4.8. Section 4.9 concludes the section.

4.2 High Performance Linear System Solver

For dense matrix A, the LU factorization produces $PA = LU$ (or $P = ALU$), where P is a pivoting matrix, L and U are unit lower triangular matrix and upper triangular matrix respectively. LU factorization is popular for solving systems of linear equations. With L and U, the linear system $Ax = b$ is solved by $Ly = b$ and then $Ux = y$. ScaLAPACK implements the right-looking version of LU with partial pivoting based on a block algorithm and 2D block cyclic data distribution.

For an $N \times N$ matrix (or submatrix) A. Split A into 2×2 blocks with block size NB. A_{11} has size $NB \times NB$, A_{12} is $NB \times (N - NB)$, A_{21} is $(N - NB) \times NB$, and A_{22} is $(N - NB) \times (N - NB)$, which is also known as the "trailing matrix". Decompose A as

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{bmatrix} \begin{bmatrix} U_{11} & U_{12} \\ 0 & L_{22} \end{bmatrix}$$

and therefore

$$\begin{cases} \begin{bmatrix} A_{11} \\ A_{21} \end{bmatrix} = \begin{bmatrix} L_{11} \\ L_{21} \end{bmatrix} U_{11} & \rightarrow PDGETF2 \\ A_{12} = L_{11} U_{12} & \rightarrow PDTRSM \\ L_{22} U_{22} = A_{22} - L_{21} U_{12} & \rightarrow PDGEMM \end{cases} \tag{4.1}$$

PDGETF2, PDTRSM and PDGEMM are the names of the ScaLAPACK routines that perform the corresponding operations on the left. This poses as one iteration (step) of the factorization, and pivoting is applied on the left and right of the current panel. The routines names in the ScaLAPACK LU are listed after "$\rightarrow$". For description, we use $\bar{U}$ to represent the area of U_{12} modified by PDTRSM, and $\tilde{U}$ for A_{22} in PDGEMM.

Block algorithms offer good granularity to benefit from high performance BLAS routines, while 2D block-cyclic distribution ensures scalability with load balancing.

4.3 Soft Error Resilience Framework

Since soft errors occur at times and locations unknown to the host algorithm, different methodologies are required to provide resilience to different part of the matrix. In this section, the error propagation in LU factorization is discussed and a general work flow of error detection and recovery is given. Details of each steps are explained in later sections.

4.3.1 Error Pattern in the Block LU Algorithm

During the process of LU factorization, the left factor L and right factor U have different "dynamics" with regard to the frequency of data change. For L, once a panel is factorized, the resulted data stored under the diagonal comes to the final form without undergoing any further changes. This offers an opportunity to use the traditional diskless checkpointing method to protect these data. ABFT cannot be applied to the panel factorization since otherwise checksum rows for the panel could be moved into data causing erroneous result. In LU, partial pivoting that swaps rows of both L and U is normally utilized to provide better stability, but this pivoting operation could break the static feature of the L data. For example, considering the case of one soft error, two generator matrices e_2 and w_2 can be used in addition to

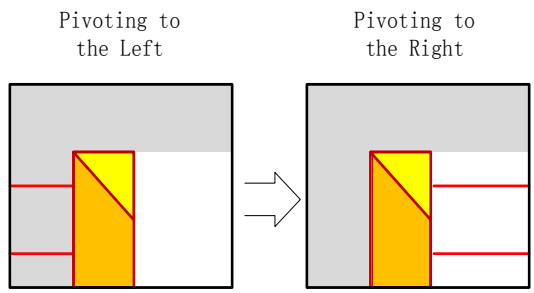


Figure 4.1: Two pivoting sweeps in LU factorization

e_1 and w_1 used for the right factor. e_i and w_i are all-1 matrix and random number matrix, respectively. For original matrix A, two rows and columns of checksum are appended as:

$$\begin{pmatrix} A & Ae_1 \; Aw_1 \\ e_2A & \dots \\ w_2A & \dots \end{pmatrix} \tag{4.2}$$

The left pivoting is depicted in Figure 4.1. Suppose pivoting requests exchanging row j and k. For e_2A:

$$e_2A = (1,1,\dots,1) \times \begin{pmatrix} a_{1\cdot} \\ \vdots \\ a_{j\cdot} \\ \vdots \\ a_{k\cdot} \\ \vdots \\ a_{m\cdot} \end{pmatrix} = (1,1,\dots,1) \times \begin{pmatrix} a_{1\cdot} \\ \vdots \\ a_{k\cdot} \\ \vdots \\ a_{j\cdot} \\ \vdots \\ a_{m\cdot} \end{pmatrix}$$

This means e_2A is immune to the left pivoting, but for w_2A this conclusion does not hold since data in w_2 are random numbers. Let $w_{2,i}$ be the ith element of w_2:

$$w_{2,1}a_1 + w_{2,2}a_2 + \cdot + w_{2,j}a_j + \cdots + W_{2,k}a_k + \cdots + w_{2,m}a_m$$
$$\neq w_{2,1}a_1 + w_{2,2}a_2 + \cdot + w_{2,j}a_k + \cdots + W_{2,k}a_j + \cdots + w_{2,m}a_m$$

and therefore each left pivoting invalidates all previous vertical checksums.

To deal with this situation, in this work the pivoting to the factorized L is delayed to the end of factorization. Since soft errors could strike at any moment, checkpointing frequency as high as once per panel factorization is necessary, but this also potentially leads to high performance overhead and therefore should be used economically. For example, even though the factorized $\bar{U}$ (result of PDTRSM) also stays static once produced, it can be protected by ABFT checksum and therefore causes less overhead.

$\tilde{U}$ differs from L and $\bar{U}$ in that it undergoes changes constantly from trailing matrix update. If soft errors alter data within $\tilde{U}$, and the erroneous data are carried along with computation to update the $\tilde{U}$, even a single-bit soft error could propagate into large area of $\tilde{U}$, let alone multiple errors at different time of the factorization.

Figure 4.2 is a demonstration of such a situation. Two LU factorizations of the same data are run. One with errors and one without error. The matrix size is 200×200 with block size 20. The two final results are subtracted and colored by the size of the absolute value of the residue. The brighter the color, the larger the residue. Using MATLAB notation, two soft errors are injected at location $(50, 120)$ and $(35, 10)$ right before the panel factorization for blocks $(41 : 200, 41 : 60)$ starts. Error at $(35, 10)$ is in the finished L area and therefore does not propagate. Error at $(50, 120)$ is in the PDTRSM area. During PDTRSM, data in column 120 gets affected and this column of errors continues into the PDGEMM area (the trailing matrix for step 40) until PDGETF2 starts on blocks $(100 : 200, 100 : 121)$ when errors spread out to the whole trailing matrix $(120 : 200, 120 : 200)$. It is worth noting that errors on the diagonals

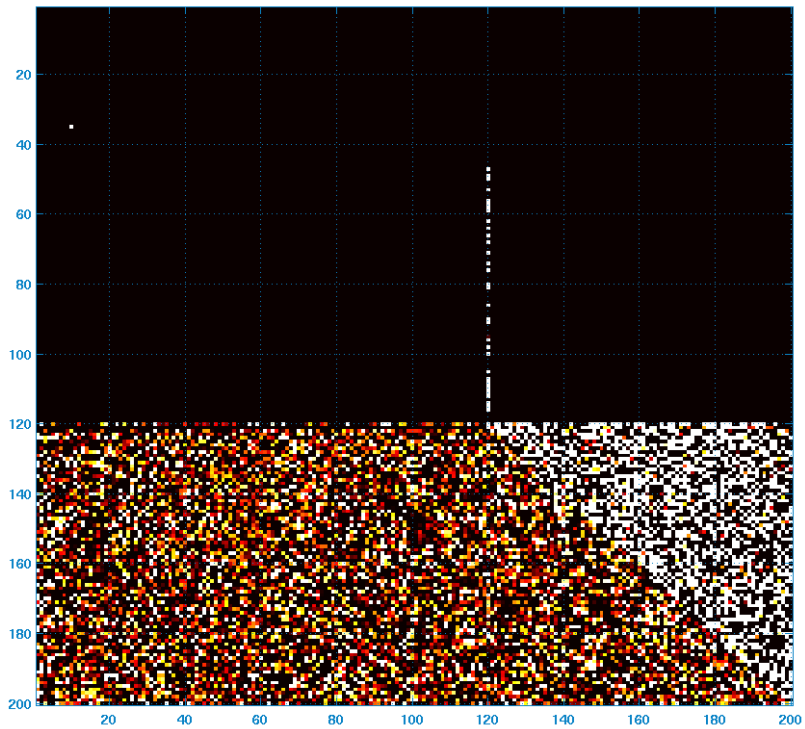


Figure 4.2: Error propagation

also cause the pivoting sequence to diverge from the correct sequence, and this affect the areas below row 120 of L.

From the example, it can be seen that large areas of the final L and U can be contaminated by a single soft error, and the affected area is a function of the soft error location and timing. Available fault tolerance, like C/R and diskless checkpointing, are not applicable because they require the location and time information of error, and by the end of the factorization the error could have propagated into their checksum and invalidated the redundancy for recovery.

Figure 4.3 shows an example of multiple-error propagation in a small matrix. Gaussian elimination is applied to a 30×30 matrix. To simplify the illustration, no pivoting nor block algorithm is used. Each step of the Gaussian elimination zeros out elements below the diagonal in one column. The color scheme is the same as Figure 4.2. During the elimination, Two soft errors are injected at step 1 and 3 at location (6,13) and (12, 18) using addition. Since both errors occur below the row 3, these

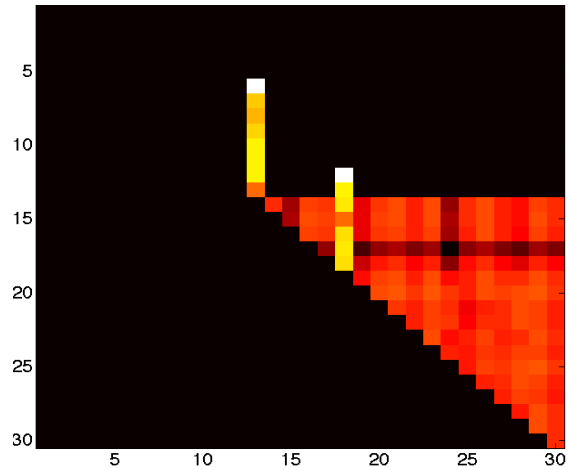


Figure 4.3: Example of error propagation in the U result of a 30×30 matrix

errors fall in the $\tilde{U}$ area of steps 1 to 3. The two white dots at (6,13) and (12, 18) are the initial injection locations. Starting from step 4, the trailing matrix update which is GEMM(matrix-matrix multiple) picks up the erroneous data for computation. As the iteration continues, the errors grow downward into the trailing matrix (in yellow). When it reaches the diagonal, the erroneous data starts to participate in the vertical scaling of zeroing out values below diagonals, and immediately the errors take over the entire trailing matrix shown in red dots. Both of the two errors follow the same propagation pattern. In the red lower right section, propagated errors from both initial errors merge. Since the propagation occurs silently, it is challenging to detect and recover from such situation without any sign of error.

4.3.2 General Work Flow

We proposed a hybrid method of ABFT and diskless checkpointing to protect LU based linear solver. This method can tolerant multiple occurrences of soft error in the whole area of factorization result and restore the correct solution x to the linear system of equations $Ax = b$. The general work flow of error detection and recovery is in Algorithm 2.

Details of Algorithm 2 will be explained in details in the coming sections.

Algorithm 2 Fault Tolerant System Work Flow

Require: $Ax = b$; Generator matrix G; Check matrix H

Step 1: Checkpointing A by $A_c = \begin{bmatrix} A & A \times G \end{bmatrix}$

Step 2: Perform LU factorization $L_c U_c = P \times A_c$ in block algorithm of block size nb with partial pivoting; Panel factorization result in each step is checkpointed immediately once produced

Step 3: Detect error occurrence by $\delta = \|U_c \times H\|$

if Found error(s) from $\delta > 0$ **then**

Step 3.1: Locate initial error(s) using δ

Step 3.2: Detect and eliminate error(s) in L

Step 3.3: Calculate $\hat{x}$ by $\hat{x} = \hat{U}(\backslash\hat{L}\backslash(P \times b))$, and

Step 3.4: Adjust $\hat{x}$ to the correct solution $x = \hat{x} + \Delta$

else

Step 4: Reach the correct solution $x = U\backslash(L\backslash(P \times b))$

end if

4.4 Detecting and Correcting Errors in L

As discussed above, diskless checkpointing is utilized to protect the left factor L from soft errors. The objective, in addition to the error correction capabilities, is having low performance overhead as the checkpointing is scaled to large computing scale.

4.4.1 Error Encoding for L: 1 Error Per Column

For any column of the computed left factor $[a_1, a_2, \cdots, a_k]^T$, the vertical checkpointing produces the following two sets of checksums:

$$\begin{cases} a_1 + a_2 + \cdots + a_k = c_1 \\ w_{2,1}a_1 + w_{2,2}a_2 + \cdots + w_{2,k}a_k = c_2 \end{cases} \tag{4.3}$$

Suppose a_j is hit by soft error to $\tilde{a_j}$, the new checksum suite becomes

$$\begin{cases} a_1 + \cdots + \tilde{a_j} + \cdots + a_k = \tilde{c_1} \\ w_{2,1}a_1 + \cdots + w_{2,j}\tilde{a_j} + \cdots + w_{2,k}a_k = \tilde{c_2} \end{cases} \tag{4.4}$$

Subtract (4.4) from (4.3), we get

$$\begin{cases} \tilde{c_1} - c_1 = \tilde{a_j} - a_j \\ \tilde{c_2} - c_2 = w_{2,j}(\tilde{a_j} - a_j) \end{cases}$$

and therefore $w_{2,j} = \frac{\tilde{c_2} - c_2}{\tilde{c_1} - c_1}$. j can be determined by looking up $w_{2,j}$ in w_2, and the erroneous data can be recovered from the first equation of (4.3).

The check matrix used for L is

$$H = \begin{bmatrix} 1, 1, \cdots, 1 & -1 & 0 \\ w_{2,1}, w_{2,2}, \cdots, w_{2,m} & 0 & -1 \end{bmatrix}$$

It is straightforward to see prove any two columns of H is independent given the random numbers in the second row do not repeat. By coding theory [104] the minimal distance of this error correcting code is 3, and therefore it can correct up to 1 error per column. In practice, the first row of H could cause large rounding errors in the recovery process due to floating pointing arithmetic. Another row of different random numbers can solve the issue as long as no two column of H are linear dependent.

4.4.2 Local Checkpointing

Since the checkpointing for L is performed in each iteration for the left factor, the scalability of such algorithm is the main concern. As already shown in Chapter 4, global vertical checkpointing does not scale because the checkpointing operation is implemented by the PDGEMM routine on the critical path of LU execution, which only engages a small amount of processes in checkpointing, and the rest are stalled.

Since the left pivoting is delayed, the left factor, once computed, is not touched any more. The communication incurred by the PDGEMM-based checkpointing can be removed by a local checkpointing scheme.

Figure 4.4 illustrates the local checkpointing mechanism. Block size is $nb \times nb$ and matrix has 5×5 blocks. The process grid is 2×3. Suppose $np_{i,j}, np_{i,j}$ are the size

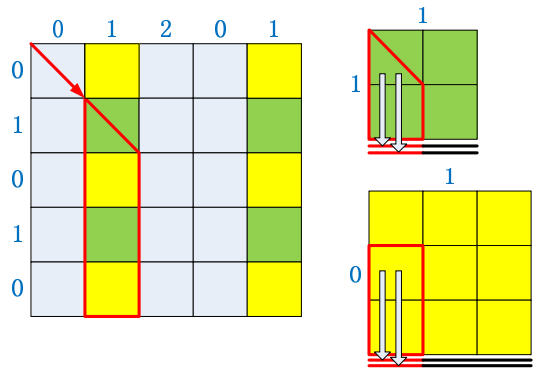


Figure 4.4: Local checkpointing algorithm

of data owned by process (i, j) (yellow and green for process (0,1) and (1,1)). Each process has a local vertical checksum space in memory of size $2 \times nq_{i,j}$.

Suppose LU factorization proceeds until the second column resulting in the left factor in the area covered in red trapezoid. Right after the panel factorization, all processes that have blocks in the current matrix column started to check if they own any blocks belonging to the current left factor. In this example, process (0,0) has 2 blocks in the red rectangle, and (0,1) has one and half blocks in the red trapezoid. Both of these two processes start to apply their local generator matrix of size $2 \times nq_{i,j}$ for the 2 blocks using DGEMM, and for process (0,1) the first DGEMM is carried out in DTRMM because only the strict lower triangular part is needed. The result is written in the corresponding local checksum location depicted in red lines in Figure 4.4.

To recover from an error, the same checkpointing scheme as in section 4.4.1 is used locally by each process. Every column of the involved processes is checked for erroneous data and therefore the local checkpointing makes the left-factor protection capable of recovering from one soft error per column of each process.

The advantage of this checkpointing is that it removes unnecessary global communication during checkpointing and breaks the checkpointing operation into $\lceil P \rceil$ embarrassingly parallelism. Further more, on a cluster where more than one

core is available on each computing node, this checkpointing can be further hidden by executing it in a separate thread so that the main thread can move on quickly to later steps. The scalability of the local checkpointing is evaluated in Section 4.8.2.

4.4.3 Error Encoding for L: Multiple Errors Per Column

In this section, the encoding scheme in Section 4.4.1 is further extended to mitigate multiple errors per column in L, implemented with the local checkpointing technique in Section 4.4.2.

For any column of the factorized panel $[l_1, l_2, \cdots, l_k]^T$ in L, the objective of checkpointing is to allow recovery from soft errors that occur to a certain number of data items in a column.

For any column of the factorized panel in L, $[l_1, l_2, \cdots, l_k]^T$, the following three checksums c_1 to c_3 are produced:

$$\begin{cases} l_1 + l_2 + \cdots + l_k = c_1 \\ w_1 l_1 + w_2 l_2 + \cdots + w_k l_k = c_2 \\ u_1 l_1 + u_2 l_2 + \cdots + u_k l_k = c_3 \end{cases} \tag{4.5}$$

Since all computation are carried out in floating point number with a fixed number of digits for exponent and fraction, the selection of w_i and u_i should avoid causing large contrast between operands during computing that encourages the accumulation of round-off errors. As an opposite example, in [61], the use of Vandermonde matrix where $w_i = j$ and $u_i = j^2$ incur fast increase of checkpointing weight magnitude and causes notable precision loss from round-off errors. When this method is used with large matrices, the resulted error locations are ambiguously in between integers.

To work with round-off errors, we propose to choose w_i and u_i from random numbers between 0 and 1. Suppose soft errors change l_i and l_j to $\hat{l_i}$ and $\hat{l_j}$ respectively, $i < j$. During the error detection step (step 3.2) in Algorithm 2, re-generating the

checksum gives:

$$\begin{cases} l_1 + \cdots + \hat{l_i} + \cdots + \hat{l_j} + \cdots + l_k = \hat{c_1} \\ w_1 l_1 + \cdots + w_i \hat{l_i} + \cdots + w_j \hat{l_j} + \cdots + w_k l_k = \hat{c_2} \\ u_1 l_1 + \cdots + u_i \hat{l_i} + \cdots + u_j \hat{l_j} + \cdots + u_k l_k = \hat{c_3} \end{cases} \tag{4.6}$$

Subtract (4.6) from (4.5), we have

$$\begin{cases} \hat{c_1} - c_1 = \hat{l_i} - l_i + \hat{l_j} - l_j \\ \hat{c_2} - c_2 = w_i(\hat{l_i} - l_i) + w_j(\hat{l_j} - l_j) \\ \hat{c_3} - c_3 = u_i(\hat{l_i} - l_i) + u_j(\hat{l_j} - l_j) \end{cases} \tag{4.7}$$

This system of equations is defined as the "symptom equations". The symptom equations establish the relationship between soft errors and checksum, however it cannot be solved "as is" since the six unknowns $\hat{l_i}$, $\hat{l_j}$, w_i, w_j and u_i, u_j outnumber the available three equations.

To reduce the number of knowns, let $u_i = w_i^2,\ i = 1, \cdots, k$. Combine the first and second equation in (4.7), we have:

$$\hat{l_j} - l_j = \frac{1}{w_j - w_i}((\hat{c_2} - c_2) - w_i(\hat{c_1} - c_1)) \tag{4.8}$$

And similarly combine the first and third equation:

$$\hat{l_j} - l_j = \frac{(\hat{c_3} - c_3) - w_i^2(\hat{c_1} - c_1)}{w_j^2 - w_i^2} \tag{4.9}$$

Eliminate $\hat{l_j} - l_j$ from (4.8) and (4.9) by connecting the right hand sides, (4.7) can be eventually reduced to

$$(\hat{c_3} - c_3) - (w_i + w_j)(\hat{c_2} - c_2) + w_i w_j(\hat{c_1} - c_1) = 0 \tag{4.10}$$

This equation is, in this work, defined as the "check equation". w_i, w_j can be determined by iterating through all possibilities in w with $O(n^2)$ complexity because $i < j$, and for each i, $n - i$ pairs of w_i w_j are tested in (4.10).

This checkpointing method also applies to one-error recovery. Suppose an error occurs to l_i only, and (4.7) becomes

$$\begin{cases} \tilde{c_1} - c_1 = \tilde{l_i} - l_i \\ \tilde{c_2} - c_2 = w_i(\tilde{l_i} - l_i) \\ \tilde{c_3} - c_3 = u_i(\tilde{l_i} - l_i) \end{cases} \tag{4.11}$$

The same method in 4.4.1 can be used to determine l_i from the first two equations of (4.11).

Using (4.7), the error detection and recovery algorithm is summarized in Algorithm 3. Note that this error protection for L applies for each column of L.

Algorithm 3 Error detection and recovery in L

Require: $\tilde{A}$, error column l and generator row w of length N, w_i, $w_j \in w$ and $w_i \neq w_j$, $i, j \in \{1 \cdots N\}$
  Calculate $\check{c}_i = \hat{c}_i - c_i$, $i = 1, 2, 3$
  if $\check{c}_i == 0$, $i = 1, 2, 3$ **then**
    No error
  else if $\check{c}_2/\check{c}_1 == \check{c}_3/\check{c}_2 == w_i$ **then**
    One error in row i, column l of the output matrix
    Recover by solving $\hat{c_1} - c_1 = \hat{l_i} - l_i$
  else
    At least two errors in column l of the output matrix
    Iterate all possible pairs w_i, $w_j \in w$
    if $(\hat{c_3} - c_3) - (w_i + w_j)(\hat{c_2} - c_2) + w_i w_j(\hat{c_1} - c_1) = 0$ **then**
      Two errors are in rows i and j, column l of the output matrix
      Recover by solving the overdetermined least square equations in (4.7) with w_i and w_j as known constants and $x = \hat{l_i} - l_i$ and $y = \hat{l_j} - l_j$ as unknowns
    else
      More than two errors occurs
    end if
  end if

The error detection and recovery algorithm can be extended to t errors with complexity $O(n^t)$ to determine the locations of errors. For example, when $t = 3$, symptom equation 4.7 becomes

$$\begin{cases} \hat{c_1} - c_1 = \hat{l_i} - l_i + \hat{l_j} - l_j + \hat{l_k} - l_k \\ \hat{c_2} - c_2 = w_i(\hat{l_i} - l_i) + w_j(\hat{l_j} - l_j) + w_k(\hat{l_k} - l_k) \\ \hat{c_3} - c_3 = u_i(\hat{l_i} - l_i) + u_j(\hat{l_j} - l_j) + u_k(\hat{l_k} - l_k) \\ \hat{c_4} - c_4 = h_i(\hat{l_i} - l_i) + h_j(\hat{l_j} - l_j) + h_k(\hat{l_k} - l_k) \end{cases} \tag{4.12}$$

Here i, j and k correspond to the three errors' locations. Similar to the double-error case, we use $u_i = w_i^2$ and $h_i = w_i^3,\ i = 1 \cdots k$. The symptom equations in (4.12) is simplified to:

$$\begin{cases} C_1 = x + y + z \\ C_2 = w_i x + w_j y + w_k z \\ C_3 = w_i^2 x + w_j^2 y + w_k^2 z \\ C_4 = w_i^3 x + w_j^3 y + w_k^3 z \end{cases} \tag{4.13}$$

where $C_i = \hat{c_i} - c_i,\ i = 1 \cdots 4$, and $x = \hat{l_i} - l_i$, $y = \hat{l_j} - l_j$, and $z = \hat{l_k} - l_k$. The task is to determine w_i, w_j and w_k.

Represent x and y as functions of z using the first two equations from (4.13):

$$\begin{cases} x = \frac{w_j C_1 C_2 - (w_j - w_k)z}{w_j - w_i} \\ y = \frac{w_i C_1 C_2 - (w_i - w_k)z}{w_i - w_j} \end{cases} \tag{4.14}$$

Replace x and y in the $3rd$ and $4th$ equations of (4.13) with (4.14) and reduce z, the check equation is formed as:

$$\begin{aligned} &\frac{C_4(w_i - w_j) + w_i^3(w_j C_1 - C_2) - w_j^3(w_i C_1 - C_2)}{(w_i - w_j)w_k^3 - (w_i - w_k)w_j^3 + (w_j - w_k)w_i^3} \\ = &\frac{C_3(w_i - w_j) + w_i^2(w_j C_1 - C_2) - w_j^2(w_i C_1 - C_2)}{(w_i - w_j)w_k^2 - (w_i - w_k)w_j^2 + (w_j - w_k)w_i^2} \end{aligned} \tag{4.15}$$

By iterating through all possible pairs of w_i, w_j and w_k using the check equation, the three error locations can be determined and the error value can be found accordingly.

4.5 Encoding for Multiple Errors in $\bar{U}$ and $\tilde{U}$

Soft errors in $\bar{U}$ and $\tilde{U}$ differ from those in L because they participate in the computation and therefore propagate to large areas. The case with two errors are discussed in detail and is then shown how to extend to $t > 2$ errors. For soft errors in matrix operation, Luk et al. has proposed to cast soft error to an initial erroneous matrix to avoid considering the difficulty of detecting error timely [89]. Fitzpatrick, et al. extended Luk's modeling to two soft errors [61]. In their works, a soft error is treated as a rank-one perturbation to the matrix. The effect of the soft error is cast back to a different initial matrix from which "conceptually" the LU factorization produces the same erroneous result but without the soft error occurring during the process. Based on this model and using a different encoding scheme (similar to the one in Section 4.4.3), this section devise methods to detect and correct multiple soft errors for LU with partial pivoting. The encoding scheme makes the soft error model suitable for floating point number operations.

4.5.1 Soft Errors Modeling

LU factorization can be viewed as multiplying a set of triangularization matrices from the left on the input matrix A to get the final triangular form. Let $A_0 = A$, and $A_t = L_{t-1}P_{t-1}A_{t-1}$. P_{t-1} is the partial pivoting matrix at step $t-1$. At the end of the factorization, $PA_0 = LU$, where U is an upper triangular matrix.

Suppose two soft errors occur in the $\bar{U}$ or $\tilde{U}$ area at locations (i_1, j_1) and (i_2, j_2) in step s_1 and s_2. In the most general case, $s_1 \neq s_2$, $i_1 \neq i_2$ and $j_1 \neq j_2$. Without loss of generality, let $s_1 < s_2$.

At step s_2, express the soft error as a perturbation to the matrix at location (i_2, j_2):

$$\hat{A}_{s_2} = A_{s_2} - \delta e_{i_2} e_{j_2}^T$$

A_{s_2} is the state of the matrix at step s_2 right before the soft error occurs, and $\hat{A}_{s_2}$ is outcome of A_{s_2} modified by a soft error of magnitude δ at location (i_2, j_2). e_{i_2} and e_{j_2} are zero column vectors with 1s at rows i_2 and j_2 respectively.

The error at step s_2 is cast back as a perturbation to the matrix at step s_1,

$$\begin{aligned}
& \hat{A}_{s_2} = A_{s_2} - \delta e_{i_2} e_{j_2}^T \\
= \quad & L_{s_2-1} P_{s_2-1} L_{s_2-2} P_{s_2-2} \cdots L_{s_1} P_{s_1} \hat{A}_{s_1} - \delta e_{i_2} e_{j_2}^T \\
\therefore \quad & (L_{s_2-1} P_{s_2-1} L_{s_2-2} P_{s_2-2} \cdots L_{s_1} P_{s_1})^{-1} \hat{A}_{s_2} \\
= \quad & \hat{A}_{s_1} - (L_{s_2-1} P_{s_2-1} L_{s_2-2} P_{s_2-2} \cdots L_{s_1} P_{s_1})^{-1} \delta e_{i_2} e_{i_2}^T
\end{aligned}$$

Let

$$\begin{aligned}
f = (L_{s_2-1} P_{s_2-1} L_{s_2-2} P_{s_2-2} \cdots L_{s_1} P_{s_1})^{-1} \delta e_{i_2}, \\
(L_{s_2-1} P_{s_2-1} L_{s_2-2} P_{s_2-2} \cdots L_{s_1} P_{s_1})^{-1} \hat{A}_{s_2} = \hat{A}_{\bar{s_2}}
\end{aligned}$$

Therefore,

$$\hat{A}_{\bar{s_2}} = \hat{A}_{s_1} - f e_{j_2}^T \tag{4.16}$$

Continue casting (4.16) to the soft error at step s_1:

$$\begin{aligned}
& \hat{A}_{\bar{s_2}} = \hat{A}_{s_1} - f e_{j_2}^T \\
= \quad & A_{s_1} - \lambda e_{i_1} e_{j_1}^T - f e_{j_2}^T \\
= \quad & L_{s_1-1} P_{s_1-1} L_{s_1-2} P_{s_1-2} \cdots L_0 P_0 A_0 - \lambda e_{i_1} e_{j_1}^T - f e_{j_2}^T
\end{aligned}$$

Let

$$d = (L_{s_1-1}P_{s_1-1}L_{s_1-2}P_{s_1-2}\cdots L_0P_0)^{-1}\lambda e_{i_1},$$
$$(L_{s_1-1}P_{s_1-1}L_{s_1-2}P_{s_1-2}\cdots L_0P_0)^{-1}\hat{A}_{\bar{s_2}} = \hat{A}_{\bar{s_1}}$$

And notice that

$$\begin{aligned}&(L_{s_1-1}P_{s_1-1}L_{s_1-2}P_{s_1-2}\cdots L_0P_0)^{-1}\times\\&(L_{s_2-1}P_{s_2-1}L_{s_2-2}P_{s_2-2}\cdots L_{s_2}P_{s_2})^{-1}\\&=(L_{s_2-1}P_{s_2-1}L_{s_2-2}P_{s_2-2}\cdots L_0P_0)^{-1}\end{aligned}$$

Let

$$\begin{aligned}g &= (L_{s_1-1}P_{s_1-1}L_{s_1-2}P_{s_1-2}\cdots L_0P_0)^{-1}\times f\\&=(L_{s_2-1}P_{s_2-1}L_{s_2-2}P_{s_2-2}\cdots L_0P_0)^{-1}\delta e_{i_2}\end{aligned}$$

And we have

$$\hat{A}_{\bar{s_1}} = A_0 - de_{j_1}^T - ge_{j_2}^T$$

Through this modeling process, the two soft errors are cast back to the input matrix A_0 as perturbation to the columns of j_1 and j_2. For more than 2 errors, the same process can be repeated and the general model for t errors is

$$\hat{A}_0 = A_0 - \sum_{j=1}^{t} d_{j_i}e_{j_i}^T$$

4.5.2 Errors Detection

The soft error model can be used to determined the errors' locations, as will be shown in this section. While this model is for the case where soft errors occur only in matrix

A, in fact checksum and the right hand sides b of $Ax = b$ are equally susceptible to soft errors. For these errors, b can be protected by duplication and cross check, and the protection method for L in section 4.4 can be directly applied to protect right hand sides.

In [61], four columns of checksum are used to locate two soft errors. Instead, we show that for N errors, $N + 1$ columns are enough for error detection and data recovery.

For the input matrix $A \in \mathbb{R}^{N \times N}$, checksum is generated before the factorization using generator matrix

$$G = \begin{bmatrix} e^T \\ w^T \\ (w^2)^T \end{bmatrix} = \begin{bmatrix} 1 & \cdots & 1 \\ w_1 & \cdots & w_N \\ w_1^2 & \cdots & w_N^2 \end{bmatrix} \tag{4.17}$$

and A is encoded as

$$[A, A \times G^T] = [A, Ae, Aw, Aw^2]$$

Note that the square operation is elementwise.

LU factorization is applied with the three additional checksum columns on the right as

$$P[A,\ Ae,\ Aw,\ Aw^2] = L[U,\ c,\ v,\ s] \tag{4.18}$$

$c, v, s \in \mathbb{R}^{N \times 1}$ are the checksum after factorization.

As shown in the error model, the LU factorization infected with errors is equal to an error-free LU factorization to a different initial (erroneous) matrix $\hat{A}_0$. Using A to represent the original correct initial matrix and $\hat{A}$ for the erroneous initial matrix, (4.18) becomes:

$$\hat{P}[\hat{A},\ Ae,\ Aw,\ Aw^2] = \hat{L}[\hat{U},\ \hat{c},\ \hat{v}, \hat{s}]$$

And using relationship between $\hat{c}$ and Ae:

$$\begin{aligned}\hat{c} &= \hat{L}^{-1}\hat{P}Ae = \hat{L}^{-1}\hat{P}(\hat{A} + de_{j_1}^T + ge_{j_2}^T)e \\ &= \hat{L}^{-1}(\hat{L}\hat{U} + \hat{P}de_{j_1}^T + \hat{P}ge_{j_2}^T)e \\ &= \hat{U}e + \hat{L}^{-1}\hat{P}d + \hat{L}^{-1}\hat{P}g\end{aligned}$$

Therefore

$$\hat{c} - \hat{U}e = \hat{L}^{-1}\hat{P}d + \hat{L}^{-1}\hat{P}g$$

By the same token,

$$\begin{aligned}\hat{v} - \hat{U}w &= w_{j_1}\hat{L}^{-1}\hat{P}d + w_{j_2}\hat{L}^{-1}\hat{P}g \\ \hat{s} - \hat{U}w^2 &= w_{j_1}^2\hat{L}^{-1}\hat{P}d + w_{j_2}^2\hat{L}^{-1}\hat{P}g\end{aligned}$$

Let $x = \hat{L}^{-1}\hat{P}d \in \mathbb{R}^{\ N\times 1}$, and $y = \hat{L}^{-1}\hat{P}g \in \mathbb{R}^{\ N\times 1}$, we have

$$\begin{cases} \hat{c} - \hat{U}e = x + y \\ \hat{v} - \hat{U}w = w_{j_1}x + w_{j_2}y \\ \hat{s} - \hat{U}w^2 = w_{j_1}^2 x + w_{j_2}^2 y \end{cases}$$

This system of equations is the vector form of (4.7), and similarly can be reduced to the check equation:

$$\begin{aligned}(\hat{s} - \hat{U}w^2) - (w_{j_1} + w_{j_2})(\hat{v} - \hat{U}w) + \\ w_{j_1}w_{j_2}(\hat{c} - \hat{U}e) = 0\end{aligned} \tag{4.19}$$

w_{j_1} and w_{j_2} can be determined by iterating through all possible $N \times (N-1)$ combinations in w for a pair that makes (4.19) hold. As a result, the error columns

j_1 and j_2 are determined. Later, these error columns are used to recover the solution of $Ax = b$.

For t soft errors, with the error model in (4.17), the check equation is:

$$\begin{cases} c_0 - \hat{U}w^0 & = & w^0_{j_1}x_1 + \cdots + w^0_{j_t}x_t \\ c_1 - \hat{U}w^1 & = & w^1_{j_1}x_1 + \cdots + w^1_{j_t}x_t \\ & \vdots & \\ c_{t-1} - \hat{U}w^{t-1} & = & w^{t-1}_{j_1}x_1 + \cdots + w^{t-1}_{j_t}x_t \end{cases} \tag{4.20}$$

All powers in (4.20) are elementwise. This general case of check equation in vector form for t errors exhibits the same structure as in the scalar form. For $t = 3$ it has been shown that check equation (4.15) can be used to determine error locations except the scalar residues C_i is replaced with vector residues $c_i - \hat{U}w^i$.

For two errors, the complexity of locating w_{j_1} and w_{j_2} is $O(N^3)$ because for each pair of w_{j_1} and w_{j_2} a vector norm is calculated to test for zero vector in (4.19) which takes $O(N)$ operations. For $t > 2$, the complexity of determining the error columns exceeds the complexity of LU factorization itself, making this method computationally impractical for real use. The same problem exists for L protection too when $t > 3$. The next section provides solution to this issue.

Since errors in $\bar{U}$ and $\tilde{U}$ propagate, the solution to (4.20) alone is insufficient for recovering the right factor U as only the columns of the initial errors can be determined. However for system of linear equations, by using Sherman-Morrison-Woodbury formula, the solution can be recovered.

4.6 Complexity Reduction

As the number of tolerable errors t increases, the complexity of locating the initial error columns grows exponentially. To resolve this issue and provide multiple error

resilience capability with practical overhead, this section proposes the complexity reduction methods for L and U.

4.6.1 Reduction for L

As shown in (4.15), to tolerate three errors in a column of L of length N, $O(N^3)$ operations are required. Even though the search can be embarrassingly parallelized since each search path is independent of others, the overall complexity is still high when large t is desired.

In the complexity $O(N^{t+1})$, N is the factor that determines the range of search. By breaking the search range into smaller segments, thereby reducing N, the complexity can be decreased to an affordable level.

There exist many ways of segmenting N but since each segment requires storage space for checksum, the segmenting method should minimize the overall storage requirement. Use N_k to represent the segment size. The k_{th} root of the vector length is chosen in this work as the segment size where k is integer and , $k \geq 1$.

Split N into equally sized segments of length $N_k = N^{\frac{1}{k}}$. Apply the encoding method in (4.5) to each of the $N^{1-\frac{1}{k}}$ segments. For each vector to tolerate t errors, $t+1$ checksum items are required. Therefore for a vector of length N, the total amount of space required to store checksums is

$$N^{1-\frac{1}{k}} \times (t+1) \times N$$

And the storage overhead over that for the data vector has the trend

$$\lim_{N\to\infty} (N^{1-\frac{1}{k}} \times (t+1) \times N \times \frac{1}{N^2}) = \lim_{N\to\infty} \frac{t+1}{\sqrt[k]{N}} = 0$$

Based on the k_{th} root segmenting method, the error detection and recovery are performed following Algorithm 4 (Using $t = 2$ as an example). Since the expensive error locating procedure is now carried out within a smaller range, the complexity

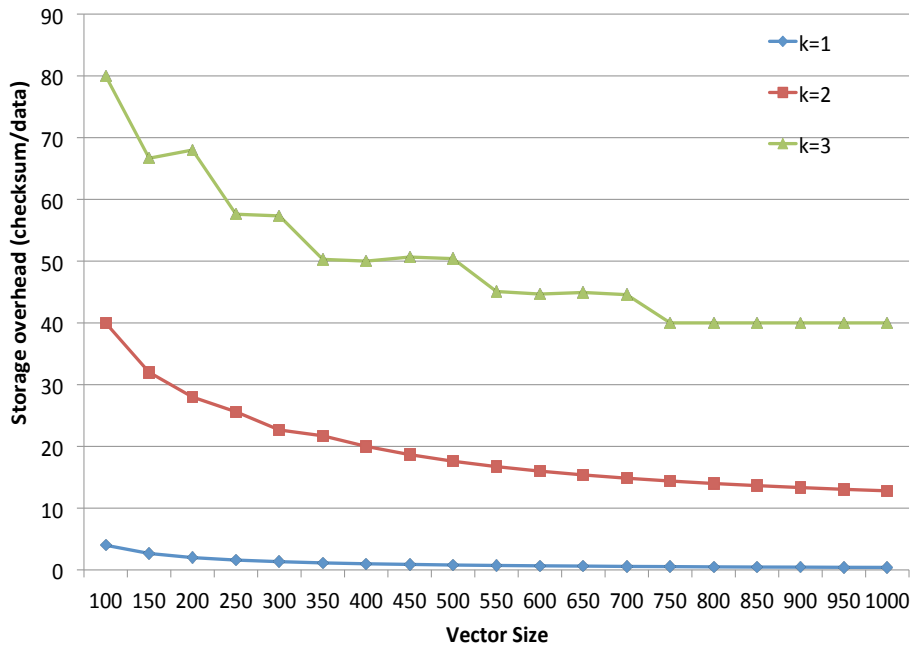


Figure 4.5: Storage overhead ($t = 3$)

of error detection is largely reduced. The operation count for Algorithm 4 includes $N^{1-\frac{1}{K}}$ vector norms of length $\sqrt[k]{N}$, and iterating in $\sqrt[k]{N}$ for the correct pair of w_i and w_j. The total overhead of locating t errors in one segment is

$$
\begin{aligned}
& O(N^{\frac{1}{k}} \times N^{1-\frac{1}{k}}) + O((N^{\frac{1}{k}})^t) \\
& = O(N) + O(N^{\frac{t}{k}}) = \begin{cases} O(N) & \text{if } t \leq k \\ O(N^{\frac{t}{k}}) & \text{if } t > k \end{cases}
\end{aligned}
$$

Algorithm 4 Error detection and recovery for one column l of L

Require: Vector l of length N; Segment length $nb = \sqrt[k]{N}$.
  for $i = 1 \rightarrow N^{1-\frac{1}{k}}$ **do**
    Using notation in (4.5),
    In the i_{th} segment with elements $l_1, \cdots, l_{nb}$
    if $\|(l_1 + l_2 + \cdots + l_{nb}) - c_1\| > 0$ **then**
      Locate errors using (4.10)
      Fix errors by solving the symptom equations in (4.7)
    end if
  end for

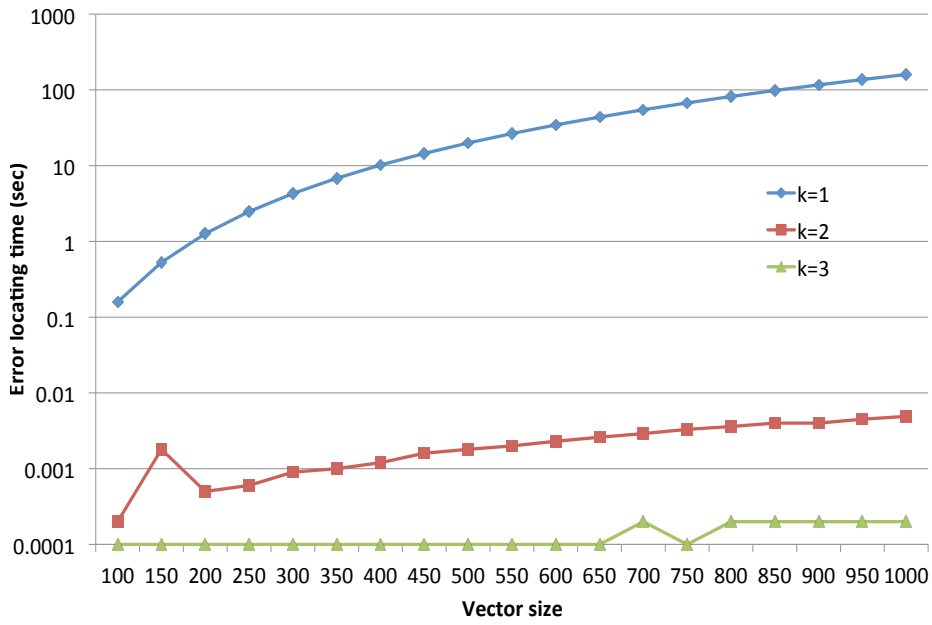


Figure 4.6: Error locating time ($t = 3$)

Note that the number of tolerable soft error t is for each segment. Therefore for large total number of tolerable errors per vector, each segment can select a smaller t, hence demanding less error locating overhead. For a fixed t, increasing k has the same effect by reducing the range of search, but comes at the cost of more extra storage according to (4.21). To evaluate the effect that different k plays on storage overhead and error locating time, an simulation is performed for $t = 3$ in MATLAB. Figure 4.5 and 4.6 show the result. In the vector case, $t = 3$ is the smallest "forbidden case" since the complexity to locate errors is $O(N^3)$, already the same as that of LU factorization. In this simulation, three errors are injected to the farthest end of input vectors, making it the worst case for error locating since all combinations of w_i, w_j and w_k have to be tried against (4.15) before a match can be found.

Compare the storage overhead and error locating time, when $k = 3$, checksum uses the most (nearly 40%) extra storage while finds error in less than 0.001 seconds, while $k = 2$ only requires slightly over 10% extra storage but still achieves over 10^4 speed up to locate errors at large sizes. When $k > 3$ the storage overhead becomes

unaffordable with little improvement in error locating speed. Therefore $k = 2$ is a fair choice compromising both storage overhead and error locating speed, and the complexity when $t = 3$ and $k = 2$ is $O(N^{\frac{3}{2}}) < O(N^3)$.

4.6.2 Reduction for U

For $\bar{U}$ and $\tilde{U}$, without any complexity management, locating t soft errors requires $O(N^{t+1})$ operations, one order higher than the original complexity for L protection. To reduce the complexity to an affordable level, the segmenting method in section 4.6.1 is extended to apply on blocked LU algorithm for $\bar{U}$ and $\tilde{U}$ protection.

Block Encoding of Matrix

In blocked LU algorithm, panel factorization itself is an LU factorization of a tall and skinny panel, therefore the encoding technique in 4.5 can be used to protect a panel or several panels too if the encoding is performed accordingly.

Theorem 4.6.1. *Block Encoding protects the trailing matrix at the end of each iteration of LU factorization*

Proof. Given a matrix A of size $N \times N$ and generator matrix G. Split A into equally sized block $N_k \times N_k$ and let G have size $N_k \times (t+1)$, where t is the number of errors tolerable by using G. Matrix A is encoded as:

$$\left[\begin{array}{c||ccc|ccc} A_{11} & A_{12} & \cdots & A_{1n} & A_{11}G & \cdots & A_{1n}G \\ \hline\hline A_{21} & A_{22} & \cdots & A_{2n} & A_{21}G & \cdots & A_{2n}G \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ A_{n1} & A_{n2} & \cdots & A_{nn} & A_{n1}G & \cdots & A_{nn}G \end{array}\right]$$

Start by performing one iteration of LU factorization for the first panel of block of size $N \times N_k$, generating U_{11} and $L_{i,1}$ where $2 < i < n$. Then perform triangular

solving and trailing matrix update making the encoded matrix into state:

$$\left[\begin{array}{c||ccc|ccc} L_{11}\backslash U_{11} & U_{12} & \cdots & U_{1n} & C_{11} & \cdots & C_{1n} \\ \hline\hline L_{21} & \tilde{A}_{22} & \cdots & \tilde{A}_{2n} & C_{21} & \cdots & C_{2n} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ L_{n1} & \tilde{A}_{n2} & \cdots & \tilde{A}_{nn} & C_{n1} & \cdots & C_{nn} \end{array}\right] \tag{4.21}$$

Note that similar to ScaLAPACK storage format, the lower triangular blocks $L_{i,1}, 2 < i < n$ are stored in the zeroed out area in the first panel.

According to (4.1), we have

$$\begin{cases} U_{1j} = L_{11}^{-1} A_{1j} & \\ C_{1j} = L_{11}^{-1} A_{1j} G & i = \{2, \cdots, n\} \\ \tilde{A}_{ij} = A_{ij} - L_{i1} \times U_{1j} & j = \{1, \cdots, n\} \\ C_{ij} = C_{ij} - L_{i1} \times C_{1j} & \end{cases}$$

$$\therefore\ \tilde{C}_{ij} = A_{ij} G - L_{i1} L_{11}^{-1} A_{ij} G$$

$$= (A_{ij} - L_{i1} U_{1j}) G = \tilde{A}_{ij} G$$

Similar method can be used in the rest of the iterations.

As an example, take a matrix A of 2×2 blocks encoded using the generator in (4.17):

$$A_c = \begin{bmatrix} A_{11} & A_{12} & A_{11}G & A_{12}G \\ A_{21} & A_{22} & A_{21}G & A_{22}G \end{bmatrix}$$

Carry out LU factorization to A_c and we have:

$$A_c = \begin{bmatrix} L_{11} & 0 \\ L_{21} & L22 \end{bmatrix} \begin{bmatrix} U_{11} & U_{12} & C_{11} & C_{12} \\ 0 & U_{22} & C_{21} & C_{22} \end{bmatrix}$$

And C_1 to C_4 can be calculated as:

$$\begin{cases} C_{11} & = U_{11}G, \ C_{12} & = U_{12}G \\ C_{21} & = \varnothing, \ C_{22} & = U_{22}G \end{cases} \tag{4.22}$$

This shows that after LU factorization, the added four checksum blocks offer protection to the three data blocks U_{11}, U_{12} and U_{22} independently. Since G in (4.17) offers t errors protection capability, the three data blocks in U ***each*** can tolerate up to t soft errors.

In ScaLAPACK, matrix A is split into blocks of size $NB \times NB$, therefore when $k = 2$, the encoding block size N_k is $N \times \sqrt{N}$ rounded to multiple of NB.

Error detection is performed on each $\sqrt{N} \times \sqrt{N}$ blocks. For U_{11}, first $\|U_{11} \times G(:,1) - C_{11}(:,1)\|$ is checked and if the norm is sufficiently large, the error detection procedure in 4.5.2 is then activated for this $\sqrt{N} \times \sqrt{N}$ block.

The complexity of performing blocked error detection and locating includes the error check that is either full or upper triangular matrix-vector multiplication and the error locating operation within the block. Suppose $N_k = N^{\frac{1}{k}}$ rounded to a multiple of NB, and the generator matrix G has size $N_k \times t$ for t error resilience capability. Since error checking is only carried out in the upper triangular blocks of A, there are in total $1+2+\cdots+N^{1-\frac{1}{k}}$ number of blocks. Therefore the error checking complexity is

$$\begin{aligned} &(1+2+\cdots+N^{1-\frac{1}{k}}) \times O((N^{\frac{1}{k}})^2) \\ &= \frac{N^{1-\frac{1}{k}}(N^{1-\frac{1}{k}}+1)}{2} \times O(N^{\frac{2}{k}}) \end{aligned}$$

And the error locating complexity is $O(N^{\frac{t}{k}} \times N^{\frac{1}{k}})$. For instance, when $k = 2$ and $t = 2$, the total overhead of error detection and locating is

$$\frac{\sqrt{N}(\sqrt{N}+1)}{2} \times O(N) + O(N^{\frac{3}{2}}) = O(N^2) < O(N^3)$$

Therefore the overhead is affordable for LU factorization.

The total amount of extra storage for storing checksum columns is

$$N \times N^{1-\frac{1}{k}} \times (t+1)$$

And the storage overhead over that for the data vector has the trend

$$\lim_{N\to\infty} (N \times N^{1-\frac{1}{k}} \times (t+1) \times \frac{1}{N^2}) = \lim_{N\to\infty} \frac{t+1}{\sqrt[k]{N}} = 0$$

Similar to the scalar case in 4.6.1, compromise has to be made between t and k for number of error tolerated and storage overhead for checksum. Following the evaluation in Figure 4.6 and 4.5, $t = k = 2$ is chosen for the experiments in this work.

Reduction of ABFT Extra Flops

The additional ABFT checksum columns to protect U participate in the trailing matrix update ($L_{22}U_{22} = A_{22} - L_{21}U_{12}$ in (4.1)) of LU factorization, and since trailing matrix update takes up a majority of the floating point operations (FLOPS) of LU, extra FLOPS from the additional checksum columns could cause significant overhead even if no errors occur at all. Block encoding in section 4.6.2 helps reduce the error locating overhead, but in fact it also offers an insight to lower the error-free overhead.

In (4.22), encoding is performed within blocks $[A_{11},\ A_{21}]^T$ and $[A_{12},\ A_{22}]^T$ separately. For block A_{11}, the checksum C_{11}'s relationship with U_{11} by $C_{11} = U_{11} \times G$ is established when panel $[A_{11},\ A_{21}]^T$ is factorized. After this point, C_{11} are not subject to any further change and C_{21} remains zero even though further operations (triangular solve with C_{21} as right hand sides) are applied. The invariance of C_{11} and C_{21} after the first panel factorization indicates that $[C_{11},\ C_{21}]^T$ can be excluded from any later operations.

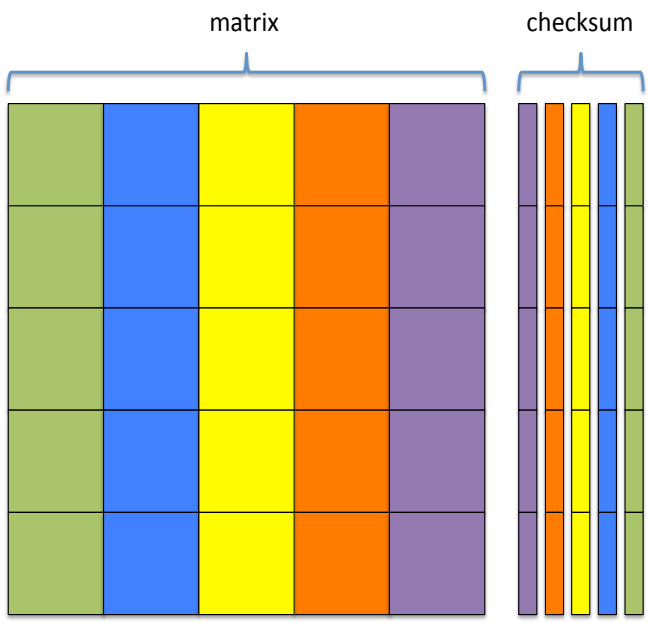


Figure 4.7: Checksum layout example of a 5×5 blocks matrix

Corollary 4.6.2. *After each step of LU factorization, one panel of checksum columns corresponding to the panel being factorized in this step can be excluded from further operation.*

Proof. In (4.21), U_{12} does not participate in any further operation because the next iteration starts from $\tilde{A}_{22}$ to the bottom-right corner of the encoded matrix.

$$\begin{aligned}
\because \quad & L_{i1} = A_{i1} \times U_{11}^{-1},\ i = [2, \cdots, n] \\
\therefore \quad & C_{i1} = A_{i1}G - L_{i1}C_{11} = A_{i1}G - L_{i1}U_{11}G \\
& = L_{i1}U_{11}G - L_{i1}U_{11}G = \emptyset
\end{aligned}$$

Since C_{21} is used as the right hand sides of the triangular solve in the next iteration, which produces $\emptyset$ as result too, after the trailing matrix update of the next iteration, $C_{21}, \cdots, C_{n1}$ are all still $\emptyset$. Therefore the panel $[C_{11}, \cdots, C_{n1}]$ are not subject to further change nor does it contribute to any factorization result, hence this panel can

be excluded. In the next iteration, the actively participating data is

$$\left[\begin{array}{c||ccc|ccc} L_{11}\backslash U_{11} & U_{12} & \cdots & U_{1n} & C_{12} & \cdots & C_{1n} \\ \hline\hline L_{21} & \tilde{A}_{22} & \cdots & \tilde{A}_{2n} & C_{22} & \cdots & C_{2n} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ L_{n1} & \tilde{A}_{n2} & \cdots & \tilde{A}_{nn} & C_{n2} & \cdots & C_{nn} \end{array}\right]$$

By the same process, in each iteration the panel of checksum columns that corresponds to the just factorized matrix can be left out of further operation.

In order to benefit from the complexity reduction of Corollary 4.6.2, the layout of checksum columns is reversed horizontally. Figure 4.7 shows an example of such design. The block size is the N_k. Each $N_k \times N_k$ block has a $N_k \times t$ block of checksum. The checksum blocks are labelled with the same color as the data they serve, for example the green blocks on the right end protects the green data blocks on the left end. This layout makes it easy to implement the complexity reduction in ScaLAPACK PDGESV by simply reducing the scope of PDTRSM and PDGEMM. When panel factorization finishes the green data blocks, the green checksum blocks are no longer touched in coming iterations and therefore the extra FLOPS of updating the green blocks are eliminated. The same process continues with each checksum panels till the end of the factorization.

4.7 Recovery Algorithm

After soft errors are detected and located by their columns, the correct solution to the system of equations $Ax = b$ can be recovered. [61] suggested using Sherman-Morrison-Woodbury formula for the case of two soft errors. In this section we first review the recovery procedure and then analyze the computational complexity for correcting t soft errors.

4.7.1 Correction for x

As shown in Algorithm 2, factorization result $\hat{L}$ and $\hat{U}$ are used to compute the solution $\hat{x}$ even if the factorization has been subject to soft errors. The solution x is corrected later from $\hat{x}$ when errors are detected.

From $Ax = b$, we have

$$\begin{aligned} x &= A^{-1}b = A^{-1}(\hat{P}^{-1}\hat{P})b \\ &= (\hat{P}A)^{-1}\hat{P}b \end{aligned}$$

Both $\hat{P}$ and b are known, so $(\hat{P}A)^{-1}$ is needed for x.

From (4.17), the erroneous initial matrix $\hat{A}_{\bar{s_1}}$ differs from the real initial matrix A_0 by column j_1 and j_2, therefore

$$\begin{aligned} \hat{P}A - \hat{P}\hat{A} &= (\hat{P}a_{\cdot j_1} - \hat{L}\hat{U}_{\cdot j_1})e_{j_1}^T + (\hat{P}a_{\cdot j_2} - \hat{L}\hat{U}_{\cdot j_2})e_{j_2}^T \\ \therefore\ \hat{P}A &= \hat{L}\hat{U} + (\hat{P}a_{\cdot j_1} - \hat{L}\hat{U}_{\cdot j_1})e_{j_1}^T + (\hat{P}a_{\cdot j_2} - \hat{L}\hat{U}_{\cdot j_2})e_{j_2}^T \\ &= \hat{L}\hat{U} + \hat{L}(\hat{L}^{-1}\hat{P}a_{\cdot j_1} - \hat{U}_{\cdot j_1})e_{j_1}^T + \hat{L}(\hat{L}^{-1}\hat{P}a_{\cdot j_2} - \hat{U}_{\cdot j_2})e_{j_2}^T \end{aligned}$$

Let $t_{j_1} = \hat{L}^{-1}\hat{P}a_{\cdot j_1} - \hat{U}_{\cdot j_1}$, and $t_{j_2} = \hat{L}^{-1}\hat{P}a_{\cdot j_2} - \hat{U}_{\cdot j_2}$,

$$\therefore\ \hat{P}A = \hat{L}\hat{U}(I + \hat{U}^{-1}t_{j_1}e_{j_1}^T + \hat{U}^{-1}t_{j_2}e_{j_2}^T)$$

Let $v_{j_1} = \hat{U}^{-1}t_{j_1}$ and $v_{j_2} = \hat{U}^{-1}t_{j_2}$,

$$\begin{aligned} \therefore\ \hat{P}A &= \hat{L}\hat{U}(I + v_{j_1}e_{j_1}^T + v_{j_2}e_{j_2}^T) \\ &= \hat{L}\hat{U}(I + \begin{bmatrix} v_{j_1} & v_{j_2} \end{bmatrix} \begin{bmatrix} e_{j_1} & e_{j_2} \end{bmatrix}^T) \end{aligned}$$

Let $U_x = \begin{bmatrix} v_{j_1} & v_{j_2} \end{bmatrix}$, $V_x = \begin{bmatrix} e_{j_1} & e_{j_2} \end{bmatrix}$),

$$\therefore\ (\hat{P}A)^{-1} = (I + U_x V_x^T)^{-1}(\hat{L}\hat{U})^{-1} \tag{4.23}$$

Apply the Sherman-Morrison-Woodbury formula [140, 141] to (4.23):

$$\begin{aligned} x &= (\hat{P}A)^{-1}\hat{P}b \\ &= (I - U_x(I + V_x U_x^{-1} V_x^T)(\hat{L}\hat{U})^{-1}\hat{P}b \\ &= (I - U_x(I + V_x^T U_x)^{-1} V_x^T)\hat{x} \end{aligned} \tag{4.24}$$

Hence the correct solution x can be corrected from $\hat{x}$.

4.7.2 Computation Complexity

At the center of computing the correct solution is

$$U_x(I + V_x^T U_x)^{-1} V_x^T$$

For t errors, $V_x^T U_x$ produces a $t \times t$ matrix. t is normally selected as small integers such as 2 or 3 for the protection from flips in 2× or 3× 64 bits, hence the inverse of $t \times t$ can be solved directly. For example, when $t = 2$

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}^{-1} = \frac{1}{ad - bc} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix}$$

resulting in eight FLOPS. And since V_x is filled with 0s except the two 1s at row j_1 and j_2 of column one and two respectively, four FLOPS are needed to generate $V_x^T U_x$ and I plus the result of $V_x^T U_x$, each. Let $Y = (I + V_x^T U_x)^{-1}$, similarly due to the sparsity of V_x, $Y \times V_x^T$ also requires four FLOPS. Let $Z = Y \times V_x^T$, compute $Z \times \hat{x}$ yields a 2×1 matrix costing six FLOPS, and at last $4 \times N$ FLOPS are paid to update the solution on $\hat{x}$. In summary, $O(N)$ overhead is required to calculate (4.24).

Another part of operation overhead comes from computing U_x, namely $v_{j_1} \cdots v_{j_t}$. Each of these vectors takes $O(N^2)$ to compute by PDTRSM with $\hat{U}$, and also $O(N^2)$ to generate the t right hand side vectors from $t_{j_k} = \hat{L}^{-1}\hat{P}a_{\cdot j_k} - \hat{U}_{\cdot j_k},\ k \in [1 \cdots\ t]$ for

PDTRSM. Therefore, to tolerate up to t soft errors, with t being a constant, $O(N^2)$ is the computation complexity for the recovery of x from $\hat{x}$.

4.8 Performance Evaluation

This section evaluates the performance of our algorithm in scalability, checkpointing overhead and performance. The scalability and overhead tests are carried out on a small cluster at the University of Tennessee, Knoxville (UTK) named "Dancer", which is an 8-node based on two quad Intel 2.27GHz Xeon cores per node, with an Infiniband 20G interconnect. For the performance experiment, we use another cluster at UTK called "Newton", which has 72 Dell C6100 computing nodes connected by QDR Infiniband for MPI application. Each node has two 6-core Intel Xeon CPUs. We use OpenMPI on both clusters, and our algorithm implementation is based on ScaLAPACK 1.8.0 from the Netlib using double precision, and on each node GotoBLAS2-1.13 is used. Last, the experiments are run on a Cray XT5 named "Kraken" in large scale. In all the experiments, block size NB for ScaLAPACK is set to 100. The column of original matrix that is required for recovery is re-generated by PDMATGEN of ScaLAPACK. We first evaluate the case of one soft error occurs in the left and right factor, then this is extended to the multiple-error case.

4.8.1 Performance Model for the Right Factor

For the right factor, $t+1$ columns of checksum are appended at the beginning of the factorization to protect against t errors, therefore the overhead consists of this one-time checkpointing and extra FLOPS of carrying out LU factorization with the checksum.

According to [15], the execution time of LU driver (PDGESV) in ScaLAPACK is

$$T(N,P) = C_f \frac{N^3}{P} t_f + C_v \frac{N^2}{\sqrt{P}} t_v + C_m \frac{N}{NB} t_m \tag{4.25}$$

Here N and NB are matrix size and block size (supposed square matrix with square blocks), and P is the total number of processes. $C_f = \frac{2}{3}$, $C_v = 3 + \frac{1}{4}\log_2 P$ and $C_m = NB(6 + log_2 P)$. Because in our implementation, checksum resides in-site with computing processes, all three constants remain unchanged in (4.25).

When $t = 1$, the two extra columns of checksum cause

$$T_{extra} = C_f \frac{6N^2 + 12N + 8}{P} t_f + C_v \frac{4N + 4}{\sqrt{P}} t_v + C_m \frac{1}{NB} t_m$$

extra run time, which is $O(N^2)$ and is negligible to $T(N, P)$ when problem size and machine size scale up. In fact, this result also applies to the case of multiple errors. Since the number of checksum columns is a function of the number of tolerable errors t, which is independent of N, T_{extra} is still $O(N^2)$.

The initial checkpointing for the right factor is dominated by a matrix-matrix operation with matrices of size $N \times N$ and $N \times (t+1)$ for t errors in $\tilde{U}$ and $\bar{U}$. Using a similar model in (4.25), this overhead is also $O(N^2)$.

4.8.2 Scalability

Since checkpointing is performed in each iteration for the left factor, the scalability of this algorithm is the main concern. The operation counts of checkpointing a panel of height N_i using PDGEMM is $2 \times NB \times N_i$.

Figure 4.8 is the overhead experiment under weak scaling on the Dancer cluster. The overhead is calculated by

$$\frac{T_{ft\_pdgesv} - T_{netlib\_pdgesv}}{T_{netlib\_pdgesv}} \times 100\%$$

And T_{ft_pdgesv} is the run time of the soft resilient version of PDGESV, whereas T_{netlib_pdgesv} is the run time of the Netlib PDGESV, which is what the fault tolerance version is built upon, and serves as a performed baseline.

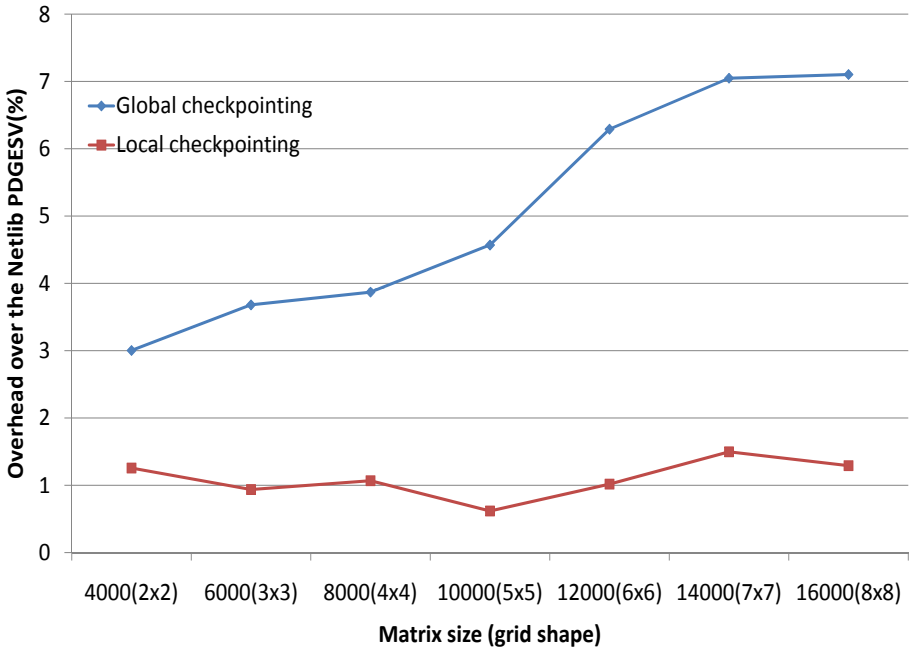


Figure 4.8: Weak scalability of global and local checkpointing for the left factor on the Dancer cluster

The result shows that the overhead of vertical checkpointing increases as computing scale and problem size scales up. Since vertical checkpointing is implemented by PDGEMM with $M = 2$, $K = N_i$ and $N = NB$, the checkpointing performance is limited by the performance of PDGEMM. Figure 4.9 is PDGEMM performance under such shape comparing to the $M = N = K$ case. The colors of lines are coordinated with the color of vertical axis titles. Clearly PDGEMM does not scale in this matrix shape. In fact, PDGEMM in PBLAS (part of ScaLAPACK) is implemented based on the DIMMA [36] algorithm, which is an enhanced version of SUMMA [136]. SUMMA is designed to work with outer product shape for high parallelism along with sophisticated broadcasting scheme, therefore the inner product shape used by the vertical checkpointing cannot benefit from such a design. In contrast, the local checkpointing scales well because checkpointing is performed in parallel by all involved processes and global collective operation is avoided. This

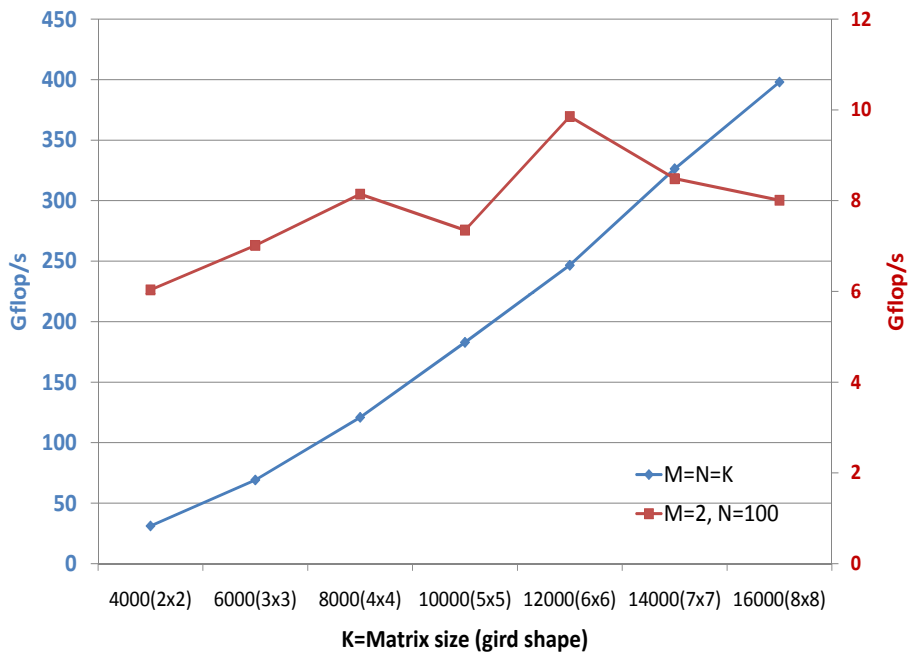


Figure 4.9: Weak scalability test of PDGEMM on the Dancer cluster

scalability ensures that the overhead caused by the left factor checkpointing will not grow into a performance drag when moving to a larger scale.

With the local checkpointing, the overall overhead of the fault tolerant PDGESV is shown in Figure 4.10, where 64 processes are arranged in a 8×8 grid. For the case marked with "one error in L and U", two data items are modified as error injection at location (400,150) and (300,500) right before the panel factorization for blocks (501:end,501:600) starts. The "one error" case includes the checkpointing overhead and the time to recover from the two errors. Same setup applied to performance experiments with alike marks.

This experiment shows that the overhead decreases with larger problems. At 32000, the overhead of the initial checkpointing for the right factor, local checkpointing for the left factor and the extra FLOPS from doing PDGESV with two extra columns is below 1%.

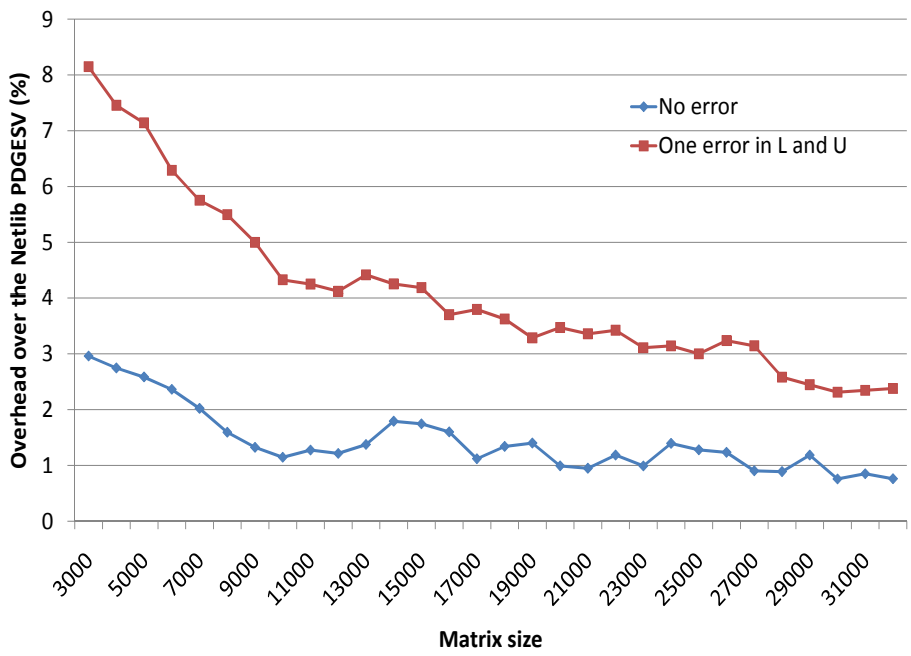


Figure 4.10: The checkpointing and recovery overhead on the Dancer cluster

4.8.3 Recovery Performance

To test the recovery performance, experiments are carried out on the Dancer, Newton and Kraken clusters.

Single Error

Figure 4.11 is the performance in Gflop/s of the same experiment in Figure 4.10. PDGEMM performance is included as the achievable machine peak to show that ScaLAPACK PDGESV runs at a reasonable speed. Figure 4.12 is the result on Newton with 256 processes in a 16×16 grid. Both Gflop/s performance results show that the soft error resilience functionality demands little overhead, and moving to a larger grid does not cause overhead increase.

For LU, algorithm stability is an important issue and it is critical that the recovered solution is numerically close to the original solution. Since in all our experiments the recovered residue $r = \frac{\|Ax-b\|}{\|A\|\|b\|M}$ is in the same magnitude as that of the original solution, this comparison is skipped.

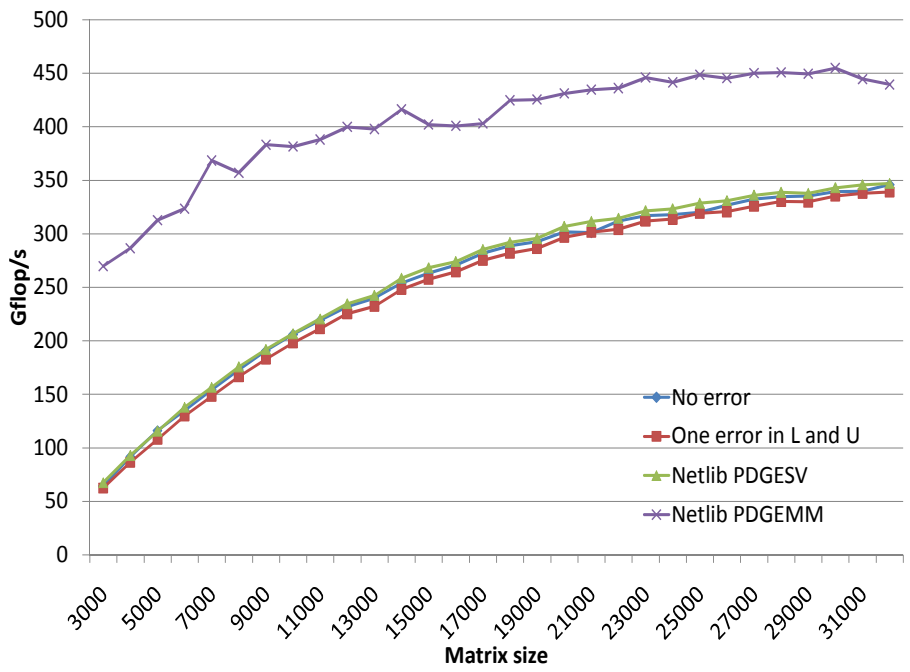


Figure 4.11: PDGESV performance with and without soft error resilience on the Dancer cluster

Figure 4.13 and 4.14 show experiments on a larger installation: the Kraken supercomputer. For this two runs, 6144 and 24576 cores were used respectively. The MPI processes were arranged in 32×32 and 64×64 grid, and each MPI process resides on a six-core AMD 2.6 GHz Istanbul CPU running 6 threads for local BLAS operation. Both results on Kraken show negligible overhead of error recovery.

Multiple Errors

Soft errors in the left factor are static and the detection and recovery in this area has been evaluated in Section 4.8.3 showing the scalability and small performance impact to the host algorithm. The algorithms for multiple soft errors in the right factor, on the other hand, have higher complexity and are most effectively affected by the proposed encoding and complexity reduction method. This section therefore focuses on the evaluation to this part.

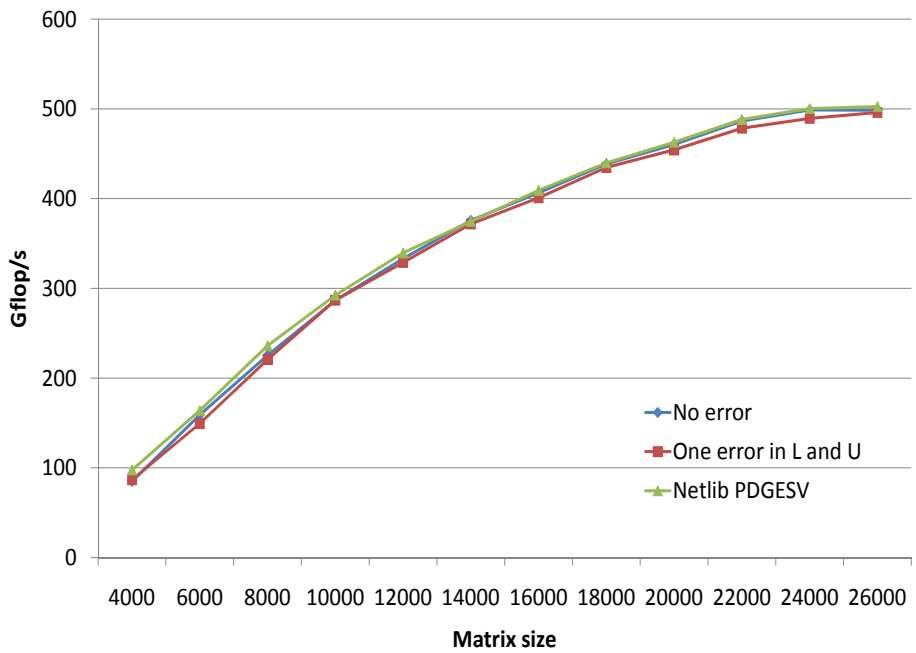


Figure 4.12: PDGESV performance with and without soft error resilience on the Newton cluster

The experiments in this section are carried on the Kraken supercomputer. In the experiments, two soft errors are injected into randomly selected locations $(336, 361)$ and $(347, 359)$ at the beginning of the randomly selected $2nd$ and $3rd$ panel factorization, respectively. Data values are incremented with random magnitudes to simulate the results of bit flips in the memory slots that hold these data. The block size for encoding is $\sqrt{N}$.

Figure 4.15 shows the effectiveness of the complexity reduction method for U with a 16×16 process grid on Kraken, and $t = 2$. The overhead is calculated by

$$\frac{FLOPS_{non-FT} - FLOPS_{FT}}{FLOPS_{non-FT}}\%$$

When $N_k = N$, block encoding for soft errors in U is not in effect. The whole matrix is encoded with a generator matrix of size $N \times 3$. In this case the overhead is close to 100% (the blue line), which means the error detection and recovery combined take as much time as solving the linear system of equation. This is consistent with the

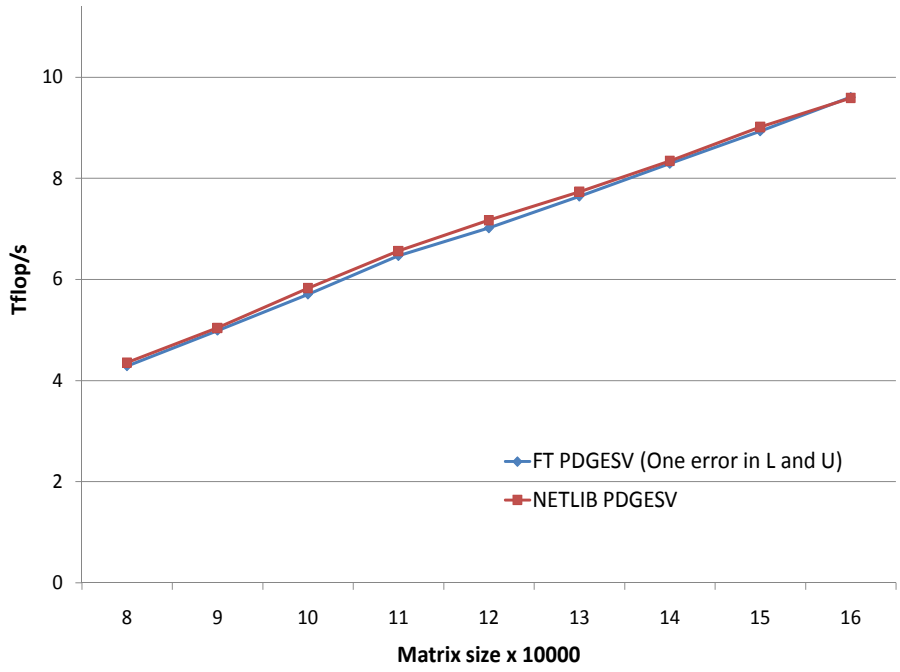


Figure 4.13: PDGESV performance with and without soft error resilience on 6144 cores of Cray XT5.

theoretical complexity of $O(N^{t+1}) = O(N^3)$. The red line, on the other hand, is the result when $N_k = \sqrt{N}$. The overhead drops quickly from a little less than 40% to 2%, which verifies that block encoding largely reduces the error detection overhead. The cost of this improvement is the extra space for storing checksum which is roughly 1% of the input matrix for size 50,000.

Figure 4.16 shows the performance of different matrix sizes on Kraken using 16,384 cores in a 128×128 grid. As the matrix becomes larger, both the original ScaLAPACK PDGESV and fault tolerant PDGESV with and without errors exhibit close performance. At the largest size 1000,000, the non-error case adds roughly 1.1% overhead, and with error correction the overhead increases to 1.3%.

Figure 4.17 is the weak scalability experiment result where both matrix size and grid dimension are doubled. Throughout all the testing sizes from 64 to 16,384 cores, FT-PDGESV declares around 1% overhead for both with and without errors cases.

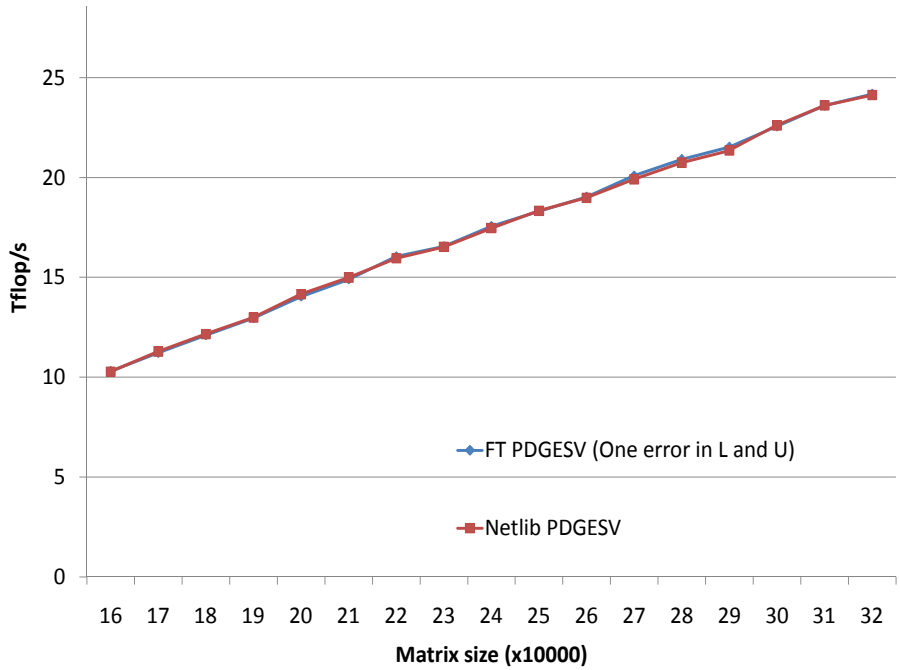


Figure 4.14: PDGESV performance with and without soft error resilience on 24576 cores of Cray XT5.

From the result in experiments, it can be confirmed that the complexity of recovering the solution to $Ax = b$ from double soft errors in the right factor has been effectively managed by the complexity reduction method, and soft errors can be precisely detected and located with the presence of round-off error. The fault tolerance functionalities can recover the solution of the dense linear system with trivial performance impact.

4.9 Conclusion

Resilience to soft error will become a critical task when computer system paces into the Petaflops age. This section proposes application-level fault tolerance algorithm that could tolerate multiple soft errors during the execution of linear system solver on large scale system. The core algorithm is based on diskless checkpointing where an efficient local checkpointing scheme is devised, and algorithm based fault tolerance where the effect of soft error in the right factor is casted to the beginning into

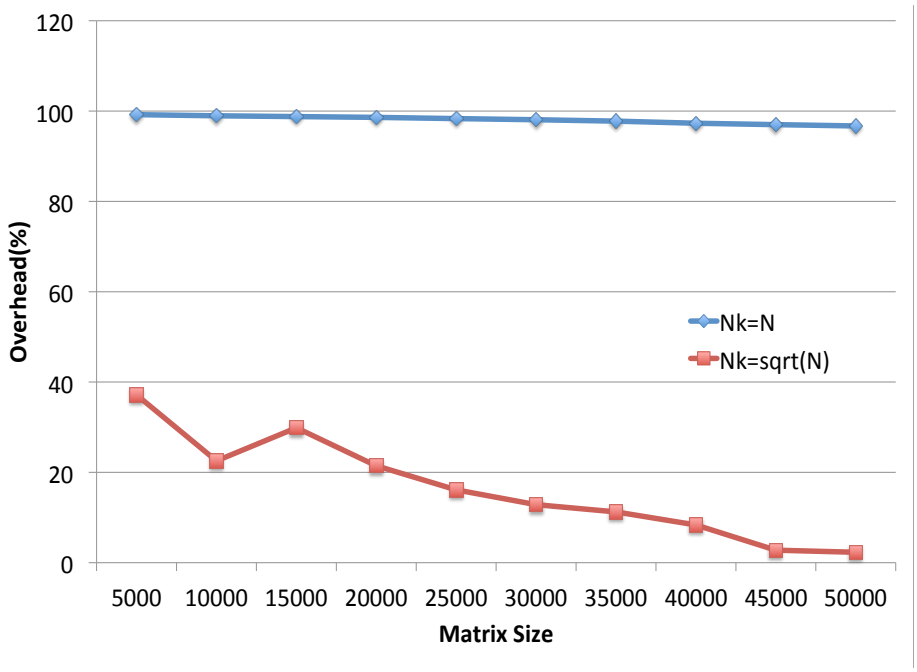
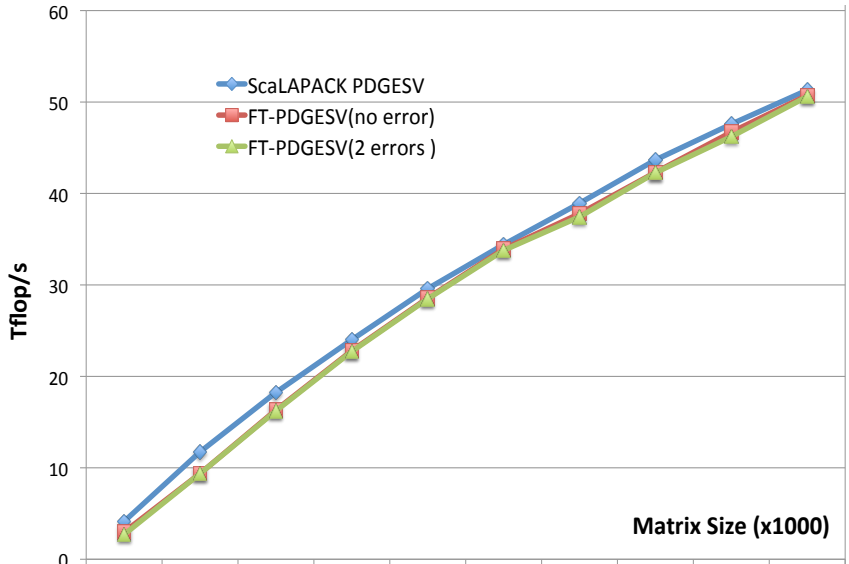


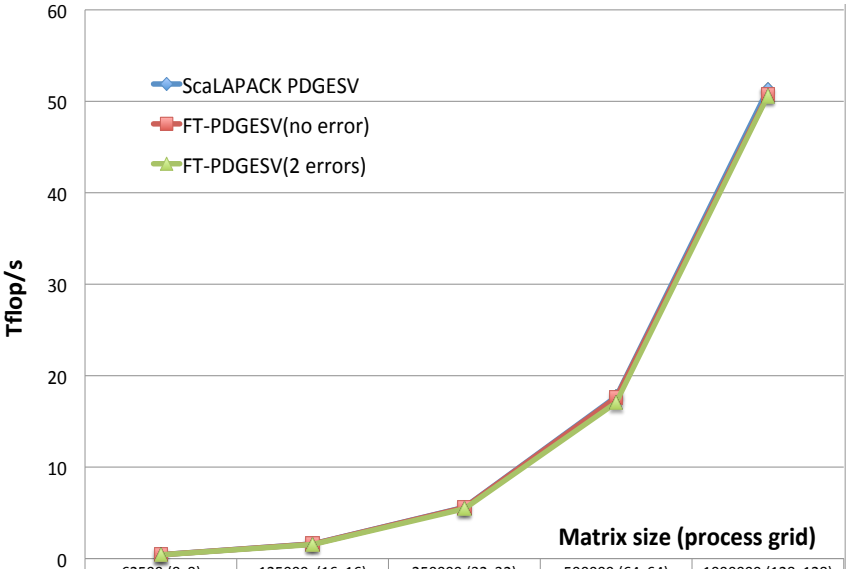
Figure 4.15: Overhead comparison result on Kraken (16×16 grid)

a different input matrix. A checksum encoding scheme is proposed to work with floating point operation, and a complexity reduction method is designed to make the soft error detection and recovery algorithm practical with low performance overhead. Experimental results on various clusters, including the Kraken supercomputer confirm the soft error mitigation capability, scalability, the effectiveness of the complexity reduction scheme, and the negligible performance overhead. Multiple soft errors in both the left and right factor can be detected and corrected, and the solution to system of linear equations $Ax = b$ can be recovered even if soft errors have caused severe error propagation, leading to large area of errors in the right factor U. The implementation is based on ScaLAPACK, but can be easily extended to other platforms, and application users can benefit from such implementation directly without getting involved into the details of error correction. Further research consist of hardening the implementation for the case where soft errors strike during the detection and recovery process, and correcting NaN caused by bit flips.



	100	200	300	400	500	600	700	800	900	1000
ScaLAPACK PDGESV	4.14	11.74	18.24	24.01	29.59	34.41	38.93	43.67	47.58	51.3
FT-PDGESV(no error)	2.98	9.38	16.32	22.81	28.56	33.92	37.79	42.31	46.7	50.72
FT-PDGESV(2 errors)	2.71	9.37	16.22	22.72	28.48	33.78	37.45	42.27	46.22	50.6

Figure 4.16: Result on Kraken with 16,384 (128×128) cores



	62500 (8x8)	125000 (16x16)	250000 (32x32)	500000 (64x64)	1000000 (128x128)
ScaLAPACK PDGESV	0.4187	1.5925	5.5959	17.7025	51.295
FT-PDGESV(no error)	0.4155	1.585	5.5731	17.5969	50.7183
FT-PDGESV(2 errors)	0.4086	1.5409	5.4418	17.0516	50.5969

Figure 4.17: Weak scalability result on Kraken

Chapter 5

Soft Error Resilience on Hybrid System with GPGPU

5.1 Introduction

Since the introduction of general-purpose computing on graphics processing units (GPGPU), GPUs have quickly become the backbone of the modern high performance computing systems. For instance, China's Tianhe-1A that ranked number one on the November 2010 TOP500 list [99] uses $7,168$ NVIDIA Tesla M2050 GPGPUs to achieve 2.57 Pflop/s in the High-Performance LINPACK (HPL) benchmark. While GPUs provide extremely high floating-point processing power, when combined with a conventional multi-core CPU in a hybrid fashion, it has been shown to be capable of further boosting the performance of scientific applications [4] by executing tasks with less parallelism on CPUs, concurrently with tasks that have high parallelism on the GPUs.

As the deployment of the GPGPUs grows rapidly, the issue of fault tolerance that has been only affecting CPU-based computing systems [65, 117] starts to emerge on GPU-based platforms. Traditionally, fault tolerance had been ignored in systems utilizing the GPUs because they were originally developed mainly for

graphics applications, such as 3D games which favor performance over reliability at bit-wise accuracy. Therefore, transient errors can be tolerated in a vast majority of rendering situations. As technology brings the GPUs into the scientific computing arena, transient errors during computing are no longer acceptable, and, to worsen the situation, in hybrid systems such errors could propagate between the CPUs and the GPUs making the hybrid systems even more fragile. Unlike fail-stop failure which brings down the whole system and halts the application execution, transient errors occur silently causing a "silent data corruption" due to various sources, mostly from cosmic radiation [76]. The errors leave no trace in system logs for system administrators to react at the time of failure. The consequences of the transient errors include incorrect application results, unpredictable code paths taken as a result of errors, and propagation of the initial failure which, all together, make the task of error detection and recovery so much more daunting.

In this chapter, we set out to provide fault tolerance and soft error resilience to the algorithms featured in the Matrix Algebra on the GPU and Multicore Architect (MAGMA) project [133]. In Chapter 4, methods to recover solution of linear system $Ax = b$ is discussed. Since only the solution x is required, no recovery is performed to the factorization result. However there are cases where the factorization result, both the left and right factor, are equally important, such as QR factorization. This chapter proposed the methods to detect and recover soft errors in QR factorization, and the implementation is based on the hybrid MAGMA code using both CPU and GPGPU.

The rest of this chapter is organized as the follows: Section 5.2 gives a lists the related work in the field of soft error protection on the GPGPU platforms. Section 5.3 introduces the target QR algorithm and its implementation in MAGMA. Section 5.4 models soft error in the QR algorithm, and Section 5.5 details the recovery algorithm including the optimization of primitives for Givens rotations on the GPU. Section 5.6 proposes a multiple-error protection algorithm for the left factor Q through tracing the MAGMA QR. Section 5.7 shows experimental results that evaluate various aspects of

our fault tolerant algorithm and, finally, Section 5.8 concludes the work and outlines possible future directions.

5.2 Related Work

Soft error in the GPU has been exploited [74], and methods have been developed to detect [122, 142] and recover from error [93, 94, 121]. Recently, soft error in matrix multiplication on a GPU has also been studied [43].

Since the introduction of the 'Fermi' architecture [105], Error Correcting Code (ECC) has been integrated to protect from errors in the GPU global memory, however this adds overhead to communication and reduces overall computing performance.

In the realm of fault tolerant QR factorization, Givens rotation based QR has been studied in [95]. However, since Householder QR is widely used in most modern math libraries, in our work we consider a right-looking Householder based QR for a hybrid CPU/GPU system. Our method is based on the error model by Luk et al. in [90]. We extended this model by adding protection to the left factor Q and provided optimized recovery algorithm on the GPU.

5.3 Hybrid QR

In linear algebra, a QR factorization decomposes a matrix A into a product $A = QR$, where Q is an orthogonal matrix and R is an upper triangular matrix. QR factorization is often used to solve the linear least squares problem, and also in QR algorithm which is at the center of a special version of eigenvalue algorithm.

Several methods exist for computing the QR factorization, such as the Gram-Schmidt process, Householder transformations, and Givens rotations. In today's high performance math libraries, for instance, LAPACK [8], ScaLAPACK [37], and MAGMA, a block version of the Householder transformations is adopted to achieve

high performance with the memory hierarchy in modern systems. For example, given an input matrix A, a Householder matrix Q_1 is multiplied to A such that

$$Q_1 A = \begin{bmatrix} r_{11} & r_{12} \cdots r_{1n} \\ 0 & \\ \vdots & A' \\ 0 & \end{bmatrix}$$

This zeros out the elements under the diagonal in the first column. The next step is carried out on the trailing matrix A' with

$$Q_2{}' = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & & & \\ \vdots & & Q_2 & \\ 0 & & & \end{bmatrix}$$

In practice, MAGMA uses a block version of the QR factorization by accumulating a few steps of the Householder matrix. This version is rich in level 3 BLAS operations and therefore achieves high performance. The result of Q is stored under the lower diagonal of the input matrix in the form of WY representation of Householder transformation products[14, 116].

Implementation-wise, the algorithm used by MAGMA is close to the LAPACK QR, except the MAGMA QR is designed and optimized for heterogeneous architectures, in particular, consisting of a CPU and a GPU. The way to accomplish this is described as follows.

The hybrid QR that we consider has the input matrix and the result on the GPU memory. The computational pattern is similar to the LAPACK's QR – a sequence of panel factorization followed by a corresponding trailing matrix update. The current panel to be factored is sent to the CPU and factored using LAPACK. The result is copied back to the GPU memory and used on the GPU for the trailing matrix update. The update is split into two – first is an update for the columns that will

form the"next" panel, followed by the update for the rest of the trailing matrix. This splitting, known as *lookahead* technique, is done so that the factorization of the next panel can start before finishing the entire update for which the next panel is part of. This allows overlapping the large update of the trailing matrix and sending the panel to the CPU, its factorization and copy back to the GPU. As a result, for large enough matrices, the overall performance of the algorithm is dictated by the performance of the matrix-matrix multiplications on the GPU. Note that communication is minimized (and overlapped with computation) as on each step the algorithm communicates a panel of size $O(NB \times N)$ and performs operations of size $O(NB \times N^2)$. For further detail on the implementation, one can see the sources available through the MAGMA site.

5.4 Soft Error Modeling

MAGMA algorithms run with both the GPU and CPU, therefore soft errors on both platforms are considered a source of contamination. Also since the result of panel factorization and lookahead trailing panel commutes between the CPU and GPU frequently, soft error could propagate between the GPU and CPU as well, depending on when and where error occurs. To ease the error analysis and avoid dealing with the timing of errors, we adopt the error modeling technique proposed in [90].

5.4.1 Error Model

Luk et al. derived their model for both LU and QR using the "ZU" notation where Z represents the left factor and U represents the right factor that is upper triangular. We return to the "QR" notation for clarity, and have in mind the right-looking Householder QR algorithm as the implementation method.

Having the initial matrix,

$$A_0 = A,$$

Householder QR is carried out by introducing Householder transforms from the left to get the final triangular form. Let

$$A_t = Q_{t-1}A_{t-1}$$

Q_{t-1} is the Householder transform matrix at step $t-1$. At step $t-1$, error occurs at random location (i, j) in matrix A as

$$\begin{aligned} \tilde{A}_t &= Q_{t-1}A_{t-1} - \lambda e_i e_j^T \\ &= Q_{t-1}(Q_{t-2}\dots Q_0)A_0 - \lambda e_i e_j^T \end{aligned} \tag{5.1}$$

e_i is a column vector with all 0 elements except 1 as the ith element. Since no error warning is raised, the factorization continues from step t till the end. If the soft error at step t is viewed as the result of perturbation to an erroneous initial matrix

$$\tilde{A} = A - de_j^T \tag{5.2}$$

where $d = \lambda(Q_{t-1}\dots Q_0)^{-1}e_i$, then the erroneous process of QR factorization equals to an error-free QR factorization from a erroneous initial matrix $\tilde{A}$.

In essence, this model treats the effect of soft error as if the factorization starts from a matrix that is a perturbation to the initial matrix. This is the same idea used in Chapter 4, where the original idea by Luk, et al. has been extended to multiple soft errors. In this chapter, therefore, we focus on the recovery of left and right factor of QR factorization.

5.4.2 Checksum for R

In MAGMA, the right-looking Householder QR algorithm follows LAPACK QR storage, where the right factor R overwrites the upper triangular part of the input

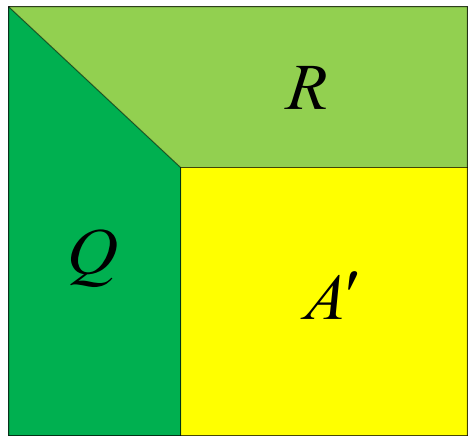


Figure 5.1: Different regions of A during factorization

matrix, including the diagonals, while the lower triangular part is replaced by Q in the form of vectors that defines elementary reflectors.

During QR factorization, once a panel of Q is produced, its values do not change till the end. Theorem 3.5.1 has shown that Q cannot be protected by appending rows of checksum at the bottom of the input matrix and having QR factorization along with the checksum rows.

The part of the matrix other than Q is divided into two regions, the already formed R and the trailing matrix A', as shown in Figure 5.1. Each iteration of the trailing update moves a few rows from A' to R, and therefore both A' and R undergo constant changes during the factorization, and cannot be protected by static checkpointing as for Q. For R, we adopt the ABFT technique from [3, 79], which was also used in Luk's work [89, 90] for soft error in systolic arrays.

To capture one error, for input matrix $A \in \mathbb{R}^{m \times n}$, two generator matrices are used, $e = (1, 1, \ldots, 1)$ and a random matrix w. $e, w \in \mathbb{R}^{m \times 1}$.

Before factorization, two columns of checksum $(Ae\ Aw)$ are calculated and appended on the right of the input matrix as $A_c = (A\ Ae\ Aw)$. Then QR factorization is applied to A_c:

$$(A\ Ae\ Aw) = Q(R\ c\ v)$$

$c, v \in \mathbb{R}^{m \times 1}$ are checksum columns after factorization.

Due to soft error, A becomes the erroneous matrix $\tilde{A}$, and the checkpointed matrix becomes

$$(\tilde{A}\ Ae\ Aw)$$

And the QR factorization becomes:

$$(\tilde{A}\ Ae\ Aw) = \tilde{Q}(\tilde{R}\ \tilde{c}\ \tilde{v}) \tag{5.3}$$

From (5.3)

$$\begin{aligned} \tilde{c} &= \tilde{Q}^{-1}Ae = \tilde{Q}^{-1}(\tilde{A} + de_j^T)e \\ &= \tilde{Q}^{-1}(\tilde{Q}\tilde{R} + de_j^T)e \\ &= \tilde{R}e + \tilde{Q}^{-1}de_j^T e = \tilde{R}e + \tilde{Q}^{-1}d \end{aligned}$$

By the same token,

$$\tilde{v} = \tilde{R}w + w_j\tilde{Q}^{-1}d$$

Assume residual vectors $r, s \in \mathbb{R}^{m \times 1}$

$$\tilde{r} = \tilde{c} - \tilde{R}e = \tilde{Q}^{-1}d \tag{5.4}$$

and

$$\tilde{s} = \tilde{v} - \tilde{R}w = w_j\tilde{Q}^{-1}d \tag{5.5}$$

Combining (5.4) and (5.5),

$$\tilde{s} = w_j\tilde{r}. \tag{5.6}$$

$\tilde{r}$ can be used to check for error, and in case an error occurs, the column in which the error initially strikes can be determined by (5.6).

5.5 Recovery Algorithm

With the knowledge of error column j, Luk et al. [90] recommended a spike-reducing technique to recover the left and right factors of ZU factorization without giving the actual algorithm. In this section we continue this work on a slightly different path due to the storage format of MAGMA QR.

5.5.1 Spike-Eliminating Technique

Using the QR notation, the spike reducing technique in [90] starts with the difference of the true initial matrix A and the erroneous initial matrix $\tilde{A}$, obtained in Equation 5.2.

$$\begin{aligned} A - \tilde{A} &= (a_{\cdot j} - \tilde{Q}\tilde{R}_{\cdot j})e_j^T \\ A &= \tilde{Q}\tilde{R} + (a_{\cdot j} - \tilde{Q}\tilde{R}_{\cdot j})e_j^T \\ A &= \tilde{Q}\tilde{R} + \tilde{Q}(\tilde{Q}^T a_{\cdot j} - \tilde{R}_{\cdot j})e_j^T \\ A &= \tilde{Q}(\tilde{R} + pe_j^T) \\ A &= \tilde{Q}\tilde{C},\ C = \tilde{R} + pe_j^T, p = \tilde{Q}^T a_{\cdot j} - \tilde{R}_{\cdot j} \end{aligned} \tag{5.7}$$

C in (5.7) is an upper triangular matrix with a spike in column j. Since QR requires Q to be an orthogonal matrix, orthogonal transformations are needed to remove non-zeros related to the spike.

There are a few choices of algorithm such as Householder transformation and Givens rotation. Householder is more computing intensive and has higher parallelism which is more suitable for the GPU, but it also requires higher amount of extra memory because, while the first Householder transformation removes the spike in column j, the triangular submatrix $(j+1 : end, j+1 : end)$ becomes a full matrix, and if j is small, this requires an extra buffer almost as large as the data matrix A and since in MAGMA QR the lower triangular is used to store Q, data matrix

space cannot be borrowed. Given that the global memory on the GPU is normally used to the limit for matrix data , Householder transformation does not qualify for this high memory demand and we choose Givens rotation as the non-zero elimination algorithm. In [90], Luk et al. also suggested a few methods including Givens rotation to eliminate this spike with matrix factorization modifying method [66] in $O(k^2)$ steps. Since Givens rotation is memory-bound, implementation on the GPU requires careful design for the best performance. This will be covered in section 5.5.3.

5.5.2 QR Update as the Recovery Algorithm

From (5.2), it can be seen that the recovery algorithm is in essence a QR update problem. Since QR update is also widely used in applications where repeated updating is required [124], this work implements the QR update algorithm for the GPU and applies it to the soft error recovery problem at hand.

The rank-1 update to QR factorization has been described in [69]. We show the algorithm in the context of QR recovery.

Given the erroneous initial matrix and its QR factorization $\tilde{A} = \tilde{Q}\tilde{R}$, the objective is to find the QR factorization of the true initial matrix $A = QR$.

Let $u = a_{\cdot j} - \tilde{Q}\tilde{R}_{\cdot j}$, and $v = e_j$,

$$\begin{aligned} A &= \tilde{A} + uv^T \\ &= \tilde{Q}\tilde{R} + uv^T \\ &= \tilde{Q}(\tilde{R} + \tilde{Q}^T uv^T) \\ \therefore\ A &= \tilde{Q}(\tilde{R} + wv^T),\ w = \tilde{Q}^T u = \tilde{Q}^T a_{\cdot j} - \tilde{R}_{\cdot j} \end{aligned}$$

First, a series of Givens rotations $J^T = J_1^T \cdots J_{n-1}^T$ is used such that

$$J^T \times w = \pm \|w\|_2 e_1$$

The sequence $1 \cdots n-1$ applied from left to w means the elimination is from bottom up. It can be shown that $H = J^T \times R$ is an upper Hessenberg matrix, and therefore

$$J^T \times (\tilde{R} + wv^T) = H \pm \|w\|_2 e_1 v^T = \hat{H}$$

is also upper Hessenberg.

To get R from $\hat{H}$, another series of Givens rotations $G^T = G_{n-1}^T \cdots G_1^T$ is used such that

$$G^T \times \hat{H} = R$$

The sequence $n-1 \cdots 1$ means the elimination is from top down.

Combining J and G,

$$Q = \tilde{Q}JG = \tilde{Q}(J_{n-1} \cdots J_1)(G_1 \cdots G_{n-1})$$

Algorithm 5 describes the above recovery procedure.

Algorithm 5 QR Recovery Algorithm based on QR-update

Require: $\tilde{A}$, $\tilde{Q}$, and $\tilde{R}$
Obtain $a_{\cdot j}$ and $\tilde{R}_{\cdot j}$
Calculate $w = \tilde{Q}^T u = \tilde{Q}^T a_{\cdot j} - \tilde{R}_{\cdot j}$
Zero out w using Givens Rotations as $k_1 = J^T \times w = \pm \|w\|_2 e_1$
Apply J^T to $\tilde{R}$ as $k_2 = J^T \tilde{R}$, and store the subdiagonals of k_2 into extra storage Y

Perform $\hat{H} = k_2 + k_1 e_j^T$
Zero out subdiagonals of $\hat{H}$ by Givens rotations $G^T \times \hat{H} = R$

Along with Algorithm 5, there are some implementation details worth noticing. First, the column j of the original matrix A is required for recovery. For scientific

applications that expect soft error with high probability, a mechanism to recover some part of the original matrix is required. Some applications can generate any column of A easily, others need to store the whole matrix A. In our implementation, at the beginning of QR factorization, matrix A on the GPU memory is asynchronously copied to the CPU memory during the first panel factorization for this purpose.

Second, recovery can be performed using the GPU in place or the CPU with two data transfers, one to load data from the GPU to the CPU and one to store result back. This solution is easier in implementation since LAPACK is equipped with Givens rotation utilities like DLARTG and DLASR, but it suffers from performance impact of data transfer and much lower parallelism of the CPU compared to the GPU. Therefore, we choose to perform the QR recovery on the GPU in place with the matrix data. Since R can only overwrite the upper triangular of A, subdiagonals of k_2 and $\hat{H}$ are kept in a separate 1D buffer Y.

5.5.3 Givens Rotation Utilities for the GPU

Givens rotation is at the center of the recovery procedure. Two operations involved are DROTG and DLASR. While these operations are readily available for the CPU, on the GPU they pose a significant challenge to be implemented with good performance especially in a fused fashion. We'll first discuss the two major challenges and then our solution.

Memory Access Pattern

DROTG generates a plane rotation such that

$$\begin{bmatrix} c & s \\ -s & c \end{bmatrix} \begin{bmatrix} f \\ g \end{bmatrix} = \begin{bmatrix} r \\ 0 \end{bmatrix}$$

In this work we use an improved version of DROTG called DLARTG, which is more numerically reliable [13].

DLASR applies a set of plane rotations to a matrix in a certain order, for example one set of plane rotation is applied to a $2 \times N$ matrix,

$$\begin{bmatrix} c & s \\ -s & c \end{bmatrix} \begin{bmatrix} x_{11} & \cdots & x_{1N} \\ x_{21} & \cdots & x_{2N} \end{bmatrix} = \begin{bmatrix} y_{11} & \cdots & y_{1N} \\ y_{21} & \cdots & y_{2N} \end{bmatrix} \tag{5.8}$$

The FLOP count is $12N$ and the memory operation is $4N + 4$, making it a memory-bound operation. While each column of the right hand side $\begin{bmatrix} y_{1j}, & y_{2j} \end{bmatrix}^T$ can be fully parallelized, without data reuse, on the GPU the performance of DLASR is still limited by the memory bandwidth between the GPU global memory and the registers. To make this situation worse, since MAGMA QR uses column-major storage, if each thread calculated one column of the right hand side, the fetching of $[x_{i1}, \cdots, x_{iN}]$ and $[y_{i1}, \cdots, y_{iN}], i = 1, 2$ by each thread does not fit the condition of global memory coalescing on the GPU, and each column has to be accessed one at a time.

Data Caching

In Algorithm 5, DLARTG and DROTG are fused together to firstly create the upper Hessenberg matrix H, and then reduce it to upper triangular. This common operation has two steps:

1. Generate a plane rotation $\begin{bmatrix} c & s \\ -s & c \end{bmatrix}$ using DLARTG for a vector $\begin{bmatrix} x_1 \\ y_1 \end{bmatrix}$
2. Apply $\begin{bmatrix} c & s \\ -s & c \end{bmatrix}$ to a $2 \times N$ matrix as in (5.8) (DLASR)

Both of these steps are carried out on the GPU. These two steps are consecutive. Figure 5.2 is an example in the last step of Algorithm 5. The plus signs on the subdiagonal are those elements to be zeroed out, and the red plus signs are the values being eliminated in the current step. Green and red are the elements that participate in the current step. This operation sweeps from top to bottom until an upper triangular matrix is produced.

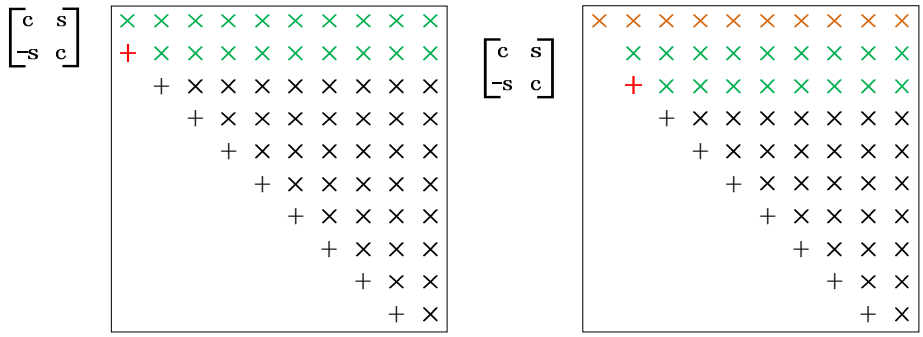


Figure 5.2: Reduction from upper Hessenberg to upper triangular

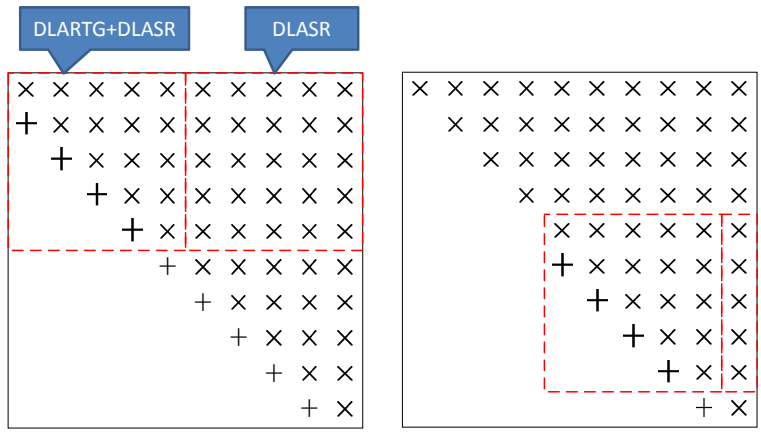


Figure 5.3: Reduction from upper Hessenberg to upper triangular (block algorithm)

Take the first two steps for example, the second row of the matrix is updated by the DLASR in the first step and then used as input for the second step. To reduce global memory access that is far more expensive than that of registers and shared memory on the GPU, this row should be cached for the next step rather than read from global memory after being just written there. Naturally we use one thread to handle each column of H, and given the size of H, more than one thread blocks is needed for each step. In addition, one thread blocks (one thread per se) performs the DLRTG before all the DLARTG thread blocks could start, hence a synchronization is needed to hold DLASR threads while waiting for the one thread that does DLARTG to finish. To achieve the aforementioned caching using registers, both DLARTG

and DLASR functionalities need to reside in one GPU kernel, otherwise the DLASR kernel calls are separated from each other by DLARTG kernel calls, and caching can only be done through shared memory, which is less efficient. The dilemma here is that CUDA offers no lightweight mechanism to synchronize all thread blocks from within threads. Available synchronization mechanisms include global synchronization initiated by host, and synchronization of all threads within a thread block. The atomic operation provides some possibilities but threads that participate in an atomic operation through a variable in global memory are serialized, and therefore suffers a large performance penalty.

Algorithm for fused DLARTG and DLASR operation

For dense linear algebra, blocked algorithms have been widely used to achieve high performance on modern computer systems with complex cache hierarchy [47]. To bridge the requirement of caching intermediate rows to reduce global memory access and the difficulty of no lightweight synchronization from within threads, we devised the following algorithm for the fused DLARTG and DLASR operation by having each step work with a block of data rather than only 2 rows.

Two types of kernels are designed. The first kernel generates a set of plane rotations and use these rotations to reduce an $NB \times NB$ upper Hessenberg submatrix on the diagonal to upper triangular. NB is selected as the maximum number of threads per thread block allowed by the GPU in use except for edge cases. In our experiment, with a Tesla T20, aka 'Fermi', $NB = 1024$.

The second kernel applies this set of plane rotations to all the data on the right of the diagonal $NB \times NB$. Global synchronization on the host is used between these two kernels. This algorithm moves down along the diagonal with a step size of NB until an upper triangular matrix is produced. Figure 5.3 is an example of this algorithm with $NB = 5$. During each iteration, only one thread block is spawned for the first type of kernel and as many thread blocks as needed are spawned for the second kernel.

Within the first kernel, steps proceed as in the unblocked version of fused DLARTG and DLASR. Intermediate rows that are produced by step $i-1$ and will be used in step i are cached in registers to avoid loading from global memory. Thread-block level synchronization is used to separate DLARTG and DLASR functionalies. Within the second kernel, steps proceed from the top down, one row each step. Similarly, intermediate rows are cached in registers. The plane rotations are stored in two vectors, respectively, in global memory to pass between the two kernels. In the second kernel, the fetching of current plane rotation pair c and s that is on the critical path of execution is moved to the beginning of kernel execution where NB threads are used to fetch NB plane rotation pairs in a coalesced fashion.

Efficient Memory Access Scheme

Figure 5.4 is a modified memory access scheme to remedy the problem discussed in section 5.5.3 for the type II kernel in section 5.5.3.

In the original kernel, all threads are lined up in a row, and during each step each thread fetches two values in a column along with a Given rotation pair from global memory. For double precision (8-byte word) memory access within half warp to be coalesced, CUDA requires all 16 words to fall in the same 16-word segment [106] but since each element in consecutive columns of this row are separated by the leading dimension, the coalescing rule does not hold.

In order to benefit from the throughput advantage provided by coalescing, a level of inner blocking is added to the kernel. Take Tesla T20 for example where the maximum number of thread per thread block is 1024. Rather than striding one row down at each step, a 4×64 block of data (yellow) are fetched together from global memory to the corresponding 64×4 piece in a shared memory buffer of size 1024×4 using a 16×64 layout of the 1024 threads such that all 16 threads in each column of the grid have consecutive thread IDs. Therefore the $4 \times 64 = 256$ elements in the yellow zone are fetched by 64 coalesced accesses. These fetching loops continue from left to right until the four rows are completely loaded. After the loading, thread layout

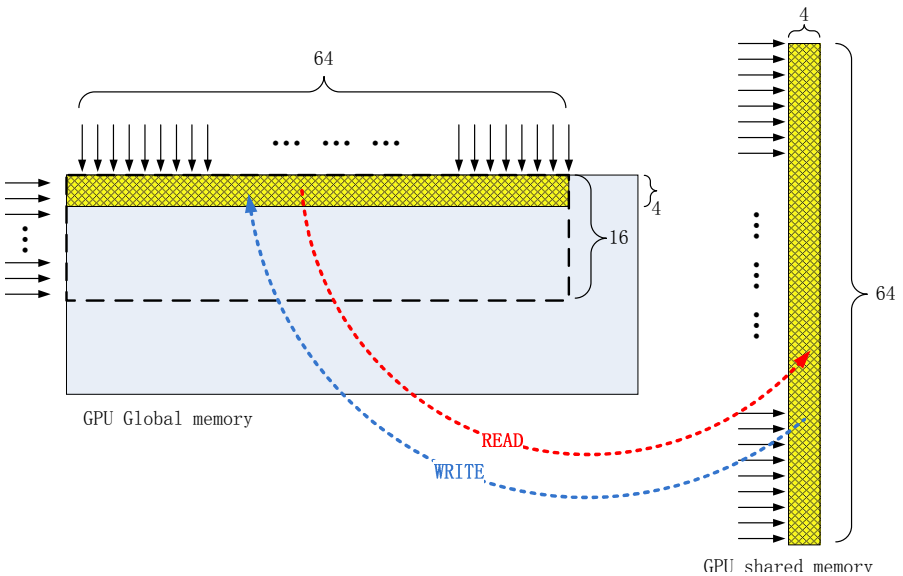


Figure 5.4: Global memory accesses in the blocked DLASR kernel

is re-arranged to 1024×1 in the inner blocking. Each thread loads two consecutive elements in a row from the shared memory. The layout of the shared memory buffer lowers the bank conflict to minimum. The inner blocking loop consumes the four rows of data (four columns in shared memory) to apply the corresponding Givens rotations, and once this four rows are finished, results are written back in the same coalesced manner as loading.

The scheme described in this section can also be used for other similar kernels in this work.

Improvement Experiment

Figure 5.5 is an experiment result of the run time for the reduction of H from upper Hessenberg to upper triangular. The matrix size derives from actual recovery experiment in section 5.7.3 where the impact of the new reduction algorithm on recovery performance is shown in Figure 5.9. By using a more efficient memory access pattern and the blocked algorithm for fused DLARTG and DLASR operation, 5x speedup is achieved.

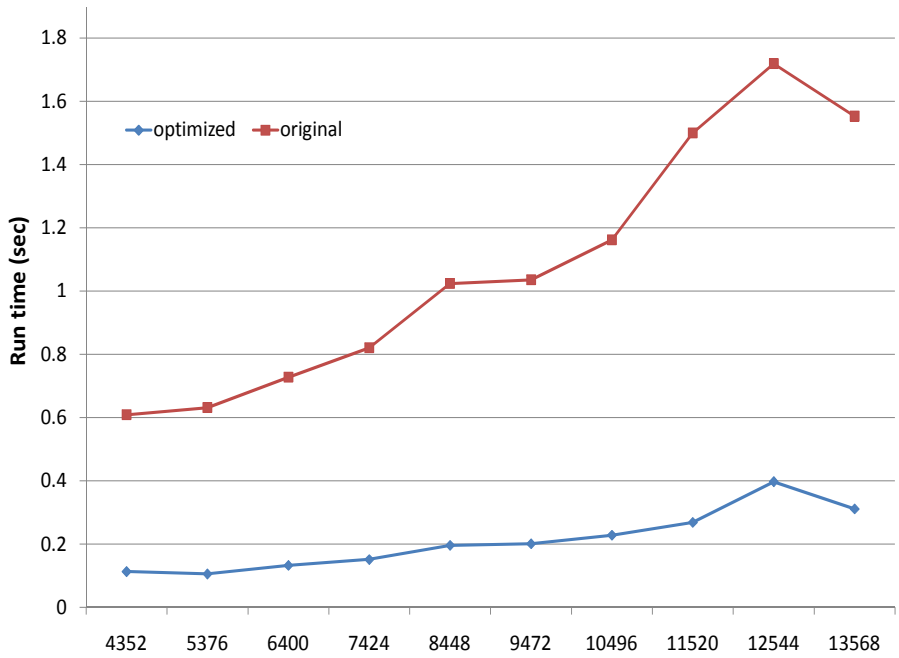


Figure 5.5: Run time comparison of the blocked DLASR (optimized) kernel and the original version

5.6 Protection for Q

Theorem 3.5.1 has shown that Q cannot be protected by ABFT as R, and the spike-eliminating algorithm 5 inherited from work by Luk et al. [90] function under the assumption that no soft error strikes $\tilde{Q}$, which is the erroneous Q caused by soft error in R or A'. In MAGMA QR, since Q occupies half of the matrix, it is as eligible to be soft error victim as other section of the matrix and therefore has to be protected.

5.6.1 Static Checkpointing for Q

In order to provide soft error resilience to Q, we propose to use diskless checkpointing algorithm because once a panel is factorized on the CPU, the result remains unchanged until the end of factorization.

For any column of the factorized panel $v_i = [v_{i1}, v_{i2}, \cdots, v_{ik}]^T$, the objective of the checkpointing scheme is to allow recovery from errors that occur to random items in the column. It has been shown in Section 4.4 how soft errors in a column of L in

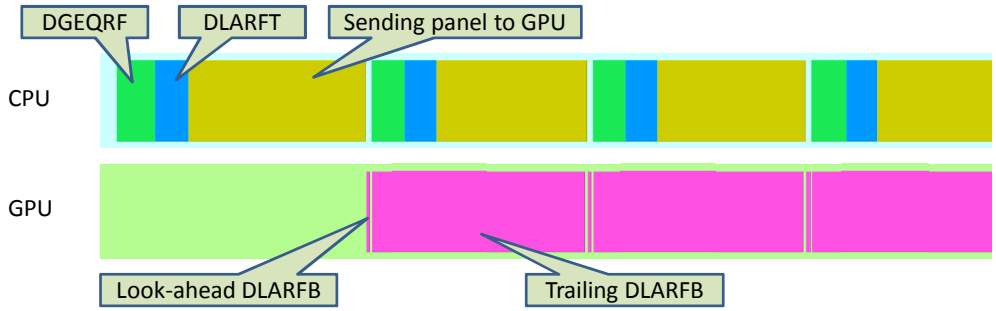


Figure 5.6: MAGMA QR tracing

LU can be protected with trivial overhead. Q can be protected by the same method, including encoding, error detection and recovery, and dimension reduction, except that the vector v_i being encoded is not a column of Q but the vector that is used to generate the Householder transformation of the form $H_i = I - \tau_i v_i v_i^T$ [38].

5.6.2 Timing of Checkpointing

The checkponting for Q is performed once per iteration of the QR factorization. Therefore the placement of this procedure requires careful consideration to avoid large performance penalty.

As described in section 5.3, The GPU onsite version of MAGMA QR produces Q using the CPU implementation DGEQRF and during step i, an $M_i \times NB$ block of the trailing matrix is sent from the GPU to the CPU memory to be factorized by DGEQRF. Then the triangular factor T of a real block reflector H is constructed by DLARFT on the CPU and both the panel factorization and T are sent to the GPU to update the trailing matrix using a GPU version DLARFB. This process is illustrated by the trace of an actual MAGMA QR run on a 48-core CPU + NVIDIA T20 GPU machine shown in Figure 5.6 generated by TAU (Tuning and Analysis Utilities) [12]. The size of this run is 17408×17408, and only the first few iterations are shown.

The best way to place the checkponting for Q such that performance overhead is minimized is finding a time slot where CPU and GPU tasks are fully overlapped and

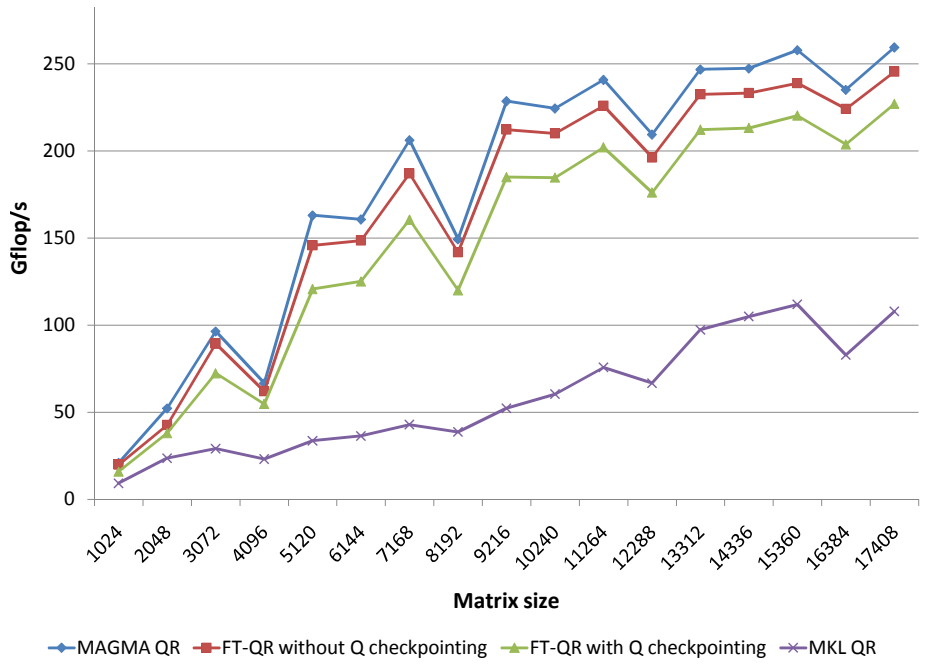


Figure 5.7: Performance of FT-QR with/without checkpointing for Q

CPU idles waiting for the GPU tasks to finish. Even though the DLARFB on the GPU takes a long time to finish, through using lookahead it keeps the CPU busy most of the time, leaving very little room for extra operation. By closely examining the tracing, we notice that the yellow section that represents cublasSetMatrix(), which sends panel factorization result from the CPU to the GPU, actually takes longer than the actual communication, and the reason is that cublasSetMatrix() is a blocking call on the GPU and it does not start the data transfer until all activities on the GPU started previously are finished. From Figure 5.6, clearly cublasSetMatrix() is always called on the CPU during the trailing matrix update (DLARFB) on the GPU and this accordingly not only blocks both the data transferring to the GPU, but also put the CPU in a busy wait and therefore cannot perform other tasks. This does not affect the performance of MAGMA QR since MAGMA QR uses 1-depth lookahead and therefore the next trailing matrix update cannot start anyway without the previous one finished.

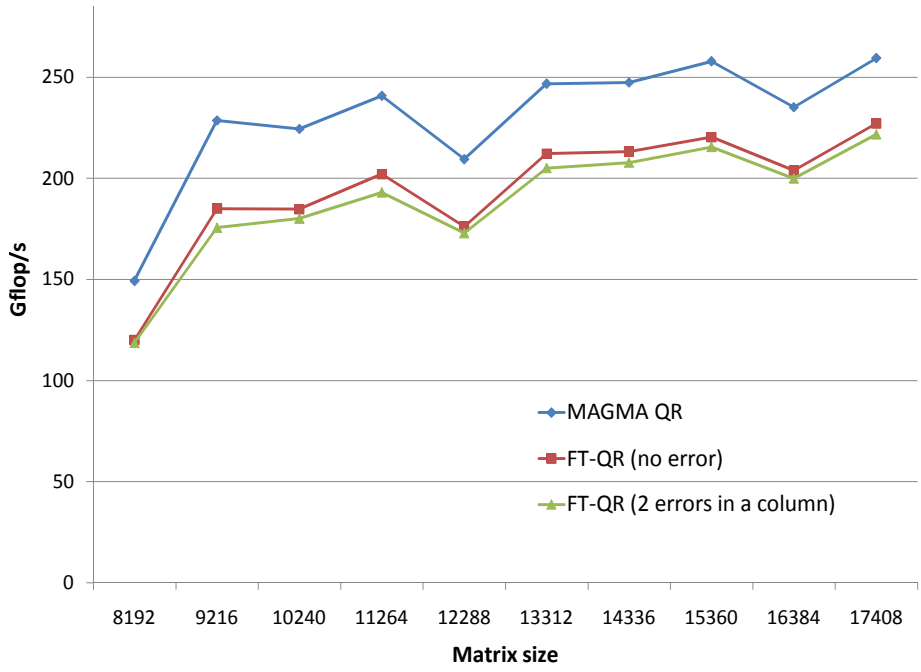


Figure 5.8: Performance of recovery for errors in Q

To release the CPU from the busy wait, cublasSetMatrix() is replaced with an asynchronous data transferring function cudaMemcpy2DAsync(). This function initiates the data transferring and returns control immediately to the CPU. The time gap between this initiation time and when the GPU DLARFB is finished is large enough to hide the checkpointing Q from the critical path. As the trailing matrix becomes smaller, there is a certain threshold of time when the GPU DLARFB finishes before the initiation of cublasSetMatrix(), and this could expose the checkpointing and cause performance impact, but this only accounts for a small portion of the execution. For such a situation, the checkpointing could be moved to run on the GPU between the time GPU DLARFB finishes and the initiation of cublasSetMatrix on the CPU.

5.7 Performance Evaluation

In this section we evaluate the performance of the fault tolerant QR algorithm on two hybrid systems. The configuration of the first experiment platform is in table 5.1:

MKL with 48 threads is used on the CPU and CUDA 4.0 is driving the GPU. All computing is in double precision and based on MAGMA version 1.0. The maximal matrix size is limited by the GPU global memory.

As discussed in section 5.5.2, the recovery algorithm requires a column of the original matrix. While this column may be re-generated cheaply, in our experiment we want to simulate the worst case where this convenience is not available, and therefore the original matrix is duplicated for the recovery process. Since the GPU memory is relatively small compared to that of the host, and is normally fully utilized for computing, the copy of the original matrix is put on the host memory. To avoid performance impact, the data transferring is performed asynchronously during the first panel factorization. The panel data is copied first so that DGEQRF on the CPU could start as soon as possible, and while the CPU is busy with the panel factorization, the rest of data is copied through DMA to the host memory. All the performance results shown in this section include this overhead.

5.7.1 Overhead Analysis

The overhead of fault tolerance comes from the following sources:

1. Duplicating the original matrix from the GPU to the CPU
2. Generating checksum on the GPU
3. Performing QR with two checksum columns on the GPU

	Brand	Frequency	# cores	Memory
CPU	AMD Opteron 6180 SE	2.5 GHz	48	256 Gb
GPU	NVIDIA C2050	1.1 GHz	14	2.7 Gb

Table 5.1: Experiment configuration

4. checkpointing Q on the CPU

5. Check for error in R and A' on the GPU

6. Check for error in Q on the GPU

7. Recovery from error in Q on the CPU and GPU

8. Recovery from error in R and A' on the GPU

Each item of the overhead sources, except the memory copy, requires $O(n^2)$ extra FLOPS. And comparing to the $\frac{4}{3}n^3$ FLOPS of QR factorization, the overhead fades away when matrix size is large enough.

5.7.2 Checkpointing of Q

Figure 5.7 is an experiment to show the overhead caused by checkpointing Q. The red line shows the performance without checkpointing Q and the performance between the red line and blue line is the overhead caused by (1)-(3) and (5)-(6) in the overhead source list. With the checkpointing Q switched on, the green line performance dips by another 5% at large matrix sizes. The green line represents the case of our fault tolerant QR runs without any error. To compare the performance with the CPU implementation, the result of MKL QR running with 48 threads is also shown. It can be seen that even with the overhead of fault tolerance, our FT-QR is still showing 2-3x speedup over the CPU implementation.

5.7.3 Recovery

Since our algorithm can deal with errors in the full matrix, the recovery performance are divided into the left factor and the right factor.

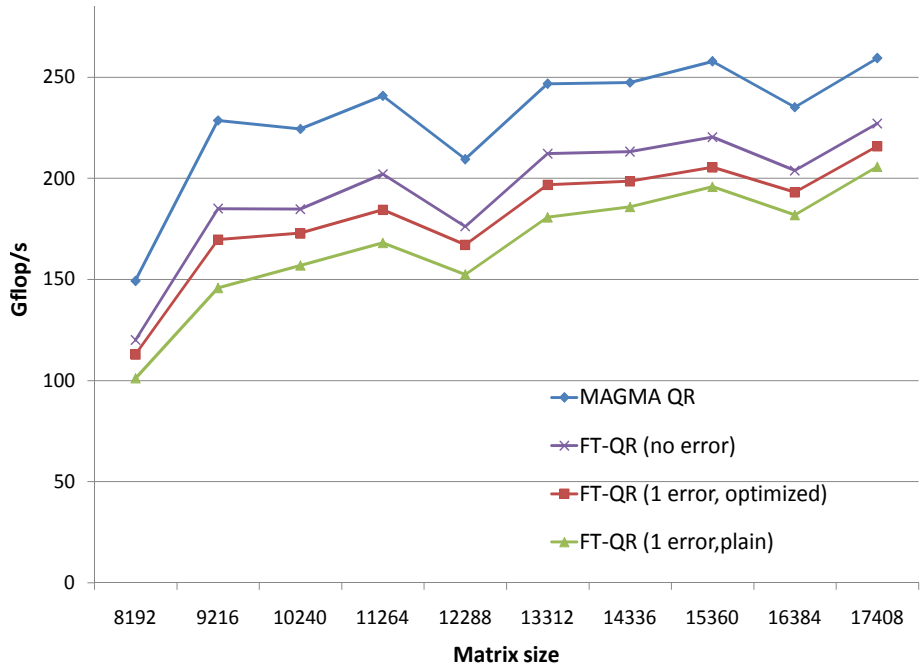


Figure 5.9: Performance of recovery for error in R

The Right Factor

Figure 5.8 is the performance of recovery from errors in Q. Two errors are injected to column 312 in rows 612 and 729 respectively. This experiment is simulating random double errors in a column of Q and therefore the error locations are not informed to the recovery algorithm. Performance result shows a small overhead from the no-error case of the fault tolerant QR, and about 15% decrease from the original MAGMA QR. This percentage will continue to drop as matrix sizes grows larger permitted by GPU with larger global memory.

The Left Factor

Figure 5.9 is the performance of recovery from error initially in R or A'. For all matrix sizes, error is injected to a random location (7681,7682) in A' on the GPU right before the 31st step of panel factorization. The purple line is the performance of FT-QR with checkpointing Q and no error.

Two recovery performances are shown. The green line is the plain implementation of Givens rotation utilities on the GPU. This implementation is limited by the GPU global memory access speed without the help of coalescing and shared memory. The red line is the optimized recovery performance where a blocked and fused DLARTG and DLASR with better memory access mechanism is in place. At the largest problem size available to this GPU, the optimization improves 5% of the recovery performance. The recovery from one soft error in A', using the optimized algorithm, reduces 15% of the overall performance of QR. This percentage will also continue to drop with larger matrix sizes.

5.7.4 Result on Keeneland

The NVIDIA C2050 has relative small on-chip global memory which limits the size of matrix in the first experiment. The second testing platform is the Keeneland Initial Delivery system which features a cluster of NVIDIA M2070 with 6GB memory, and each host runs two Intel Westmere hex-core CPUs. Figure 5.10 is the performance of both the original and soft error resilient MAGMA QR on a single node of Keeneland. Error recovery experiments use the same setup as in the test on C2050, and as a comparison, MKL QR performance is also shown running with 12 threads on a single node. The extended matrix size range shows similar overhead to the result on C2050, verifying that with the small overhead of fault resilience functionalities, the hybrid QR still outperforms multi-threaded QR on the multicore CPUs by almost 100%, and errors can be recovered with little performance impact.

5.8 Conclusion

In this chapter we developed a soft error resilient QR algorithm for hybrid architecture where the CPU and GPU are utilized together. This work enables the high

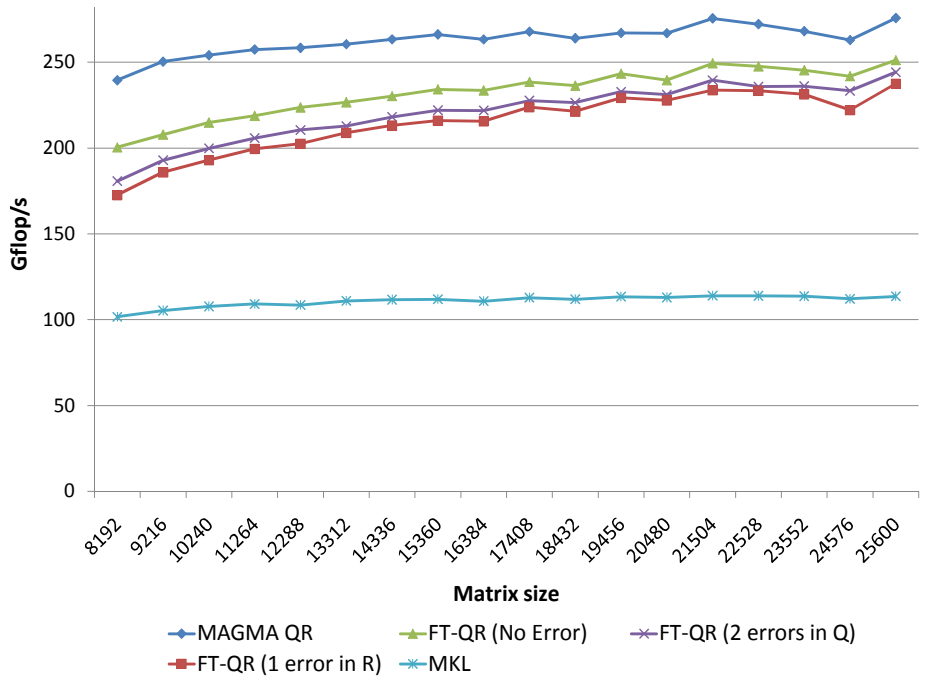


Figure 5.10: Performance on Keeneland

performance implementation of MAGMA QR to be tolerant to soft errors caused by radiation-based interference.

In the ABFT algorithm by Luk et al., the FT-QR algorithm can tolerate up to one soft error in data section R and A'. Since the recovery algorithm requires an error-free left factor Q, which is not guaranteed by Luk's algorithm, a stable and scalable multiple-error checkpointing/recovery mechanism is devised and placed in the computing environment based on the execution feature of MAGMA QR such that the checkpointing is hidden away from the critical path and therefore prevents severe performance impact. In addition, a more efficient recovery algorithm based on Givens rotation is designed. This fast Givens rotation utilities can also be used in other applications to reduce an upper Hessenberg matrix to upper triangular on the GPU.

Chapter 6

Conclusions and Future Work

6.1 Conclusion

In this dissertation, fault tolerant algorithms for both hard and soft error are developed for dense linear algebra operations on HPC systems, including large scale cluster system and hybrid system with multicore CPU and the GPGPU. For both kinds of error, we focus on full matrix protection and practical issue in real-world computing systems, such as scalability, performance impact, recovery of execution (program stack) without full fault tolerance MPI system support.

For hard errors, we developed the scalable parallel-Q checkpointing scheme, which allows the left factor to be efficiently protected with low performance impact and storage requirement. For the right factor we adopted the ABFT checksum and by using the Checkpointing-on-Failure technique, the dense matrix operations such as LU and QR factorization can recover both the lost data and running stack on the failed process. In the case where failure occurs during the trailing matrix update, immediate recovery might not be plausible due to the inconsistency of matrix states on all the processes. For such case we have shown by both proof and implementation that a delayed recovery could resolve the problem because trailing update is composed of operations that obey the rule of ABFT.

Soft error is more challenging to tolerate than hard error because transient errors normally do not cause system to crash and therefore leave no trace of existence. The effect of this silent errors is that erroneous data are carried into further computation and eventually causes large area of errors due to propagation. To combat soft error, we developed full matrix protection technique such that multiple soft errors in both the left and right factor can be detected, and both the factorization based linear system solver and the factorization itself can successfully reach correct result by recovery. We developed a floating point number encoding scheme that is used in both the scalable local checkpointing for the left factor and the ABFT based method for the right factor. This encoding scheme is resilient to the nature errors of floating point operation such as round-off and cancellation, and enables the determination of soft error locations. Soft errors in the left factor are corrected in a column-by-column fashion, while soft errors in the right factor are corrected by using the combination of soft error modeling and recovery schemes such as QR update and Sherman-Morrison-Woodbury. One particular practical issue is the computation complexity of such detection and recovery scheme. To this need, a complexity reduction method is devised where encoding is performed in segments such as $\sqrt{N}$. This drops the complexity to be lower than that of the matrix factorization and solver and thus leads to negligible overhead on real-world HPC systems.

Since more and more HPC systems nowadays are equipped with the GPGPU, and the GPGPU has historically been under-protected of ECC scheme, we have applied the proposed soft error resilience to hybrid systems with the GPGPU. Based on the hybrid QR factorization from MAGMA, vectors in the lower triangular matrix that are used to form the orthogonal left factor Q are protected by the floating point number weighted checksum performed by CPU in a time gap when CPU awaits GPU to finish the trailing matrix update, and the right factor R which normally suffers large area of propagated errors is recovered by a combination of QR update and an efficient fast Givens Rotation for the GPGPU.

Experiments on large scale cluster and hybrid HPC system have confirmed that the developed algorithms all meet the design criteria in terms of error correction and performance/storage overhead. This altogether offers very promising alternatives to the currently widely used checkpointing/restart method with much less overhead and energy consumption.

6.2 Future Work

This work uses the soft error model where the soft errors present in the form of changing the value of floating point number, and in our implementation such changing does not involve the extreme cases where bit flips in the IEEE 754 format representation cause "horrendous" modification, for example to NaN. Considering the practical possibility of such event happening, either an uncorrectable error notification should be raise to applications or such errors could be corrected in an online fashion because neither ABFT nor any of the checkpointing scheme in the field of soft error fault tolerance is effective by performing correction off-line. This will be addressed as part of the future work.

With the quick development of the GPGPU in both performance and application, more and more HPC cluster systems are being equipped with the GPGPU. With this added complexity and large scale, fault tolerance will become a critical issue that threatens productivity. To meet the goal of providing high reliability on such systems, we are in development of fault tolerance algorithm that could support multiple hard errors and soft errors in large scale distributed memory cluster system with the GPGPU.

Another interesting area is the protection of dense linear algebra computation on multicore nodes with tile algorithms. Our current implementation is based on LAPACK and ScaLAPACK which use an "old-fashion" fork-join parallelism model. However on multicore CPU system, tile algorithm such as in [4] produces better performance. We will extended the fault tolerance coverage to such system.

Bibliography

Bibliography

[1] (2009). Fault tolerance for extreme-scale computing workshop report. 12, 19

[2] (2011). http://www.top500.org/. 1

[3] Abraham, J. (1986). Fault tolerance techniques for highly parallel signal processing architectures. *Highly parallel signal processing architectures*, pages 49–65. 111

[4] Agullo, E., Demmel, J., Dongarra, J., Hadri, B., Kurzak, J., Langou, J., Ltaief, H., Luszczek, P., and Tomov, S. (2009). Numerical linear algebra on emerging architectures: The plasma and magma projects. In *Journal of Physics: Conference Series*, volume 180, page 012037. IOP Publishing. 105, 133

[5] Alameldeen, A., Wagner, I., Chishti, Z., Wu, W., Wilkerson, C., and Lu, S. (2011). Energy-efficient cache design using variable-strength error-correcting codes. In *Proceeding of the 38th annual international symposium on Computer architecture*, pages 461–472. ACM. 9

[6] Alvisi, L., Elnozahy, E., Rao, S., Husain, S., and De Mel, A. (1999). An analysis of communication induced checkpointing. In *Fault-Tolerant Computing, 1999. Digest of Papers. Twenty-Ninth Annual International Symposium on*, pages 242–249. IEEE. 12

[7] Alvisi, L. and Marzullo, K. (1995). Message logging: Pessimistic, optimistic, and causal. In *Distributed Computing Systems, 1995., Proceedings of the 15th International Conference on*, pages 229–236. IEEE. 11

[8] Anderson, E., Bai, Z., Bischof, C., Blackford, S. L., Demmel, J. W., Dongarra, J. J., Croz, J. D., Greenbaum, A., Hammarling, S., McKenney, A., and Sorensen, D. C. (1999). *LAPACK User's Guide*. Society for Industrial and Applied Mathematics, Philadelphia, Third edition. 107

[9] Anfinson, C. and Luk, F. (1988). A linear algebraic model of algorithm-based fault tolerance. *Computers, IEEE Transactions on*, 37(12):1599–1604. 15

[10] Babaoglu, O. and Marzullo, K. (1993). Consistent global states of distributed systems: Fundamental concepts and mechanisms. *Distributed Systems*, 2. 12

[11] Barrett, R., Chan, T., D'Azevedo, E., Jaeger, E., Wong, K., and Wong, R. (2010). Complex version of high performance computing linpack benchmark (hpl). *Concurrency and Computation: Practice and Experience*, 22(5):573–587. 63

[12] Biersdorff, A., Spear, W., and Mayanglambam, S. (2010). An experimental approach to performance measurement of heterogeneous parallel applications using cuda. 123

[13] Bindel, D., Demmel, J., Kahan, W., and Marques, O. (2002). On computing givens rotations reliably and efficiently. *ACM Transactions on Mathematical Software (TOMS)*, 28(2):206–238. 116

[14] Bischof, C. and Van Loan, C. (1985). The wy representation for products of householder matrices. In *Selected Papers from the Second Conference on Parallel Processing for Scientific Computing*, pages 2–13. Society for Industrial and Applied Mathematics. 46, 108

[15] Blackford, L., Cleary, A., Choi, J., D'Azevedo, E., Demmel, J., Dhillon, I., Dongarra, J., Hammarling, S., Henry, G., Petitet, A., et al. (1997). *ScaLAPACK users' guide*. Society for Industrial Mathematics. 20, 94

[16] Bland, W., Du, P., Bouteiller, A., Herault, T., Bosilca, G., and Dongarra, J. (2012). A checkpoint-on-failure protocol for algorithm-based recovery in standard mpi. *18th Euro-Par. p. to appear. LNCS, Springer (August 2012)*. 46, 49

[17] Blaum, M., Bruck, J., and Vardy, A. (1998). Interleaving schemes for multidimensional cluster errors. *Information Theory, IEEE Transactions on*, 44(2):730–743. 8

[18] Boley, D., Golub, G., Makar, S., Saxena, N., and McCluskey, E. (1995). Floating point fault tolerance with backward error assertions. *Computers, IEEE Transactions on*, 44(2):302–311. 15

[19] Bose, R. and Ray-Chaudhuri, D. (1960). On a class of error correcting binary group codes*. *Information and control*, 3(1):68–79. 8, 15

[20] Bosilca, G., Delmas, R., Dongarra, J., and Langou, J. (2009). Algorithm-based fault tolerance applied to high performance computing. *Journal of Parallel and Distributed Computing*, 69(4):410–416. 14, 20, 33

[21] Bosilca, G., Herault, T., Rezmerita, A., and Dongarra, J. (2011). On scalability for mpi runtime systems. In *Cluster Computing (CLUSTER), 2011 IEEE International Conference on*, pages 187–195. IEEE. 12

[22] Bossen, D., Tendler, J., and Reick, K. (2002). Power4 system design for high reliability. *Micro, IEEE*, 22(2):16–24. 7

[23] Bouteiller, A., Bosilca, G., and Dongarra, J. (2010). Redesigning the message logging model for high performance. *Concurrency and Computation: Practice and Experience*, 22(16):2196–2211. 19

[24] Bouteiller, A., Cappello, F., Herault, T., Krawezik, G., Lemarinier, P., and Magniette, F. (2003). Mpich-v2: a fault tolerant mpi for volatile nodes based on pessimistic sender based message logging. In *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 25. ACM. 11

[25] Brent, R., Luk, F., and Anfinson, C. (1989). Choosing small weights for multiple error detection. In *Proceedings SPIE*, volume 1058, pages 130–136. 15

[26] Brent, R., Luk, F., and Anfinson, C. (1990). Checksum schemes for fault tolerant systolic computing. *Mathematics in Signal Processing II (edited by JG McWhirter), Clarendon Press, Oxford*, pages 791–804. 15

[27] Burns, G., Daoud, R., and Vaigl, J. (1994). LAM: An open cluster environment for MPI. In *Proceedings of SC'94*, volume 94, pages 379–386. 19

[28] Cappello, F. (2009). Fault tolerance in petascale/exascale systems: Current knowledge, challenges and research opportunities. *International Journal of High Performance Computing Applications*, 23(3):212. 1, 2

[29] Cataldo, A. (2001). Sram soft errors cause hard network problems. *Electronic Engineering Times*, pages 1–2. 62

[30] Chen, C. and Hsiao, M. (1984). Error-correcting codes for semiconductor memory applications: A state-of-the-art review. *IBM Journal of Research and Development*, 28(2):124–134. 8

[31] Chen, Z. and Dongarra, J. (2006a). Algorithm-based checkpoint-free fault tolerance for parallel matrix computations on volatile resources. In *IPDPS'06*, pages 10–pp. IEEE. 14, 20

[32] Chen, Z. and Dongarra, J. (2006b). *Scalable techniques for fault tolerant high performance computing*. PhD thesis, University of Tennessee, Knoxville, TN. 14, 20

[33] Chen, Z. and Dongarra, J. (2008). Algorithm-based fault tolerance for fail-stop failures. *IEEE TPDS*, 19(12):1628–1641. 14, 20

[34] Chiu, G. and Young, C. (1996). Efficient rollback-recovery technique in distributed computing systems. *Parallel and Distributed Systems, IEEE Transactions on*, 7(6):565–577. 12

[35] Chiueh, T. and Deng, P. (1996). Evaluation of checkpoint mechanisms for massively parallel machines. In *Fault Tolerant Computing, 1996., Proceedings of Annual Symposium on*, pages 370–379. IEEE. 13

[36] Choi, J. (1997). A new parallel matrix multiplication algorithm on distributed-memory concurrent computers. In *hpc-asia*, page 224. Published by the IEEE Computer Society. 96

[37] Choi, J., Demmel, J., Dhillon, I., Dongarra, J., Ostrouchov, S., Petitet, A., Stanley, K., Walker, D., and Whaley, R. (1996a). ScaLAPACK: a portable linear algebra library for distributed memory computers–design issues and performance. *Computer Physics Communications*, 97(1-2):1–15. 27, 107

[38] Choi, J., Dongarra, J. J., Ostrouchov, S., Petitet, A., Walker, D. W., and Whaley, R. C. (1996b). The design and implementation of the ScaLAPACK LU, QR, and Cholesky factorization routines. *Scientific Programming*, 5:173–184. 123

[39] Damani, O. and Garg, V. (1996). How to recover efficiently and asynchronously when optimism fails. In *Distributed Computing Systems, 1996., Proceedings of the 16th International Conference on*, pages 108–115. IEEE. 11

[40] Davies, T., Karlsson, C., Liu, H., Ding, C., and Chen, Z. (2011). High performance linpack benchmark: a fault tolerant implementation without checkpointing. In *Proceedings of the 25th ACM International Conference on Supercomputing (ICS 2011). ACM*. 14, 20, 23, 29, 39

[41] Dell, T. (1997). A white paper on the benefits of chipkill-correct ecc for pc server main memory. *IBM Microelectronics Division*, pages 1–23. 9

[42] Ding, C., Karlsson, C., Liu, H., Davies, T., and Chen, Z. (2011a). Matrix multiplication on gpus with on-line fault tolerance. In *Parallel and Distributed Processing with Applications (ISPA), 2011 IEEE 9th International Symposium on*, pages 311–317. IEEE. 16

[43] Ding, C., Karlsson, C., Liu, H., Davies, T., and Chen, Z. (2011b). Matrix multiplication on gpus with on-line fault tolerance. In *Proceedings of the 9th IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA 2011)i*. IEEE Computer Society Press. 107

[44] Dirk, J., Nelson, M., Ziegler, J., Thompson, A., and Zabel, T. (2003). Terrestrial thermal neutrons. *Nuclear Science, IEEE Transactions on*, 50(6):2060–2064. 62

[45] Dongarra, J., Blackford, L., Choi, J., Cleary, A., D'Azevedo, E., Demmel, J., Dhillon, I., Hammarling, S., Henry, G., Petitet, A., et al. (1997). ScaLAPACK user's guide. *Society for Industrial and Applied Mathematics, Philadelphia, PA*. 28

[46] Dongarra, J. and Luszczek, P. (2010). Reducing the time to tune parallel dense linear algebra routines with partial execution and performance modelling. *niversity of Tennessee Computer Science Technical Report, Tech. Rep.* 2

[47] Dongarra, J. and Walker, D. (1993). The design of linear algebra libraries for high performance computers. 119

[48] Dongarra, J. J., Luszczek, P., and Petitet, A. (2003). The LINPACK benchmark: Past, present, and future. *Concurrency and Computation: Practice and Experience*, 15:1–18. 2

[49] Du, P., Bouteiller, A., Bosilca, G., Herault, T., and Dongarra, J. (2012a). Algorithm-based fault tolerance for dense matrix factorizations. In *Proceedings*

of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming, pages 225–234. ACM. 14, 16

[50] Du, P., Luszczek, P., and Dongarra, J. (2011a). High performance dense linear system solver with soft error resilience. In *Cluster Computing (CLUSTER), 2011 IEEE International Conference on*, pages 272–280. IEEE. 15, 17

[51] Du, P., Luszczek, P., and Dongarra, J. (2011b). High performance dense linear system solver with soft error resilience. In *Proceedings of the IEEE Cluster 2011*. IEEE Computer Society Press. 63

[52] Du, P., Luszczek, P., and Dongarra, J. (2012b). High performance dense linear system solver with resilience to multiple soft error. In *The International Conference on Computational Science*. 15

[53] Du, P., Luszczek, P., Tomov, S., and Dongarra, J. (2011c). Soft error resilient qr factorization for hybrid system with gpgpu. In *Proceedings of the second workshop on Scalable algorithms for large-scale systems*, pages 11–14. ACM. 15

[54] Elnozahy, E., Alvisi, L., Wang, Y., and Johnson, D. (2002). A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys (CSUR)*, 34(3):375–408. 11

[55] Elnozahy, E., Johnson, D., and Zwaenepoel, W. (1991). The performance of consistent checkpointing. In *Reliable Distributed Systems, 1992. Proceedings., 11th Symposium on*, pages 39–47. IEEE. 19

[56] Elnozahy, E. and Zwaenepoel, W. (1994). On the use and implementation of message logging. In *Fault-Tolerant Computing, 1994. FTCS-24. Digest of Papers., Twenty-Fourth International Symposium on*, pages 298–307. IEEE. 11

[57] Fagg, G. and Dongarra, J. (2000). FT-MPI: Fault tolerant MPI, supporting dynamic applications in a dynamic world. *EuroPVM/MPI*. 43, 45

[58] Fiala, D. (2011). 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. IEEE. 16

[59] Fiala, D., Ferreira, K., Mueller, F., and Engelmann, C. (2011). Poster: a tunable, software-based dram error detection and correction library for hpc. In *Proceedings of the 2011 companion on High Performance Computing Networking, Storage and Analysis Companion*, pages 49–50. ACM. 16

[60] Fitzpatrick, P. (1997). Extending backward error assertions to tolerance of large errors in floating point computations. *Computers, IEEE Transactions on*, 46(4):505–510. 15

[61] Fitzpatrick, P. and Murphy, C. (1992). Fault tolerant matrix triangularization and solution of linear systems of equations. In *Application Specific Array Processors, 1992. Proceedings of the International Conference on*, pages 469–480. IEEE. 15, 63, 73, 77, 80, 91

[62] Fitzpatrick, P. and Murphy, C. (1993). Solution of linear systems of equations in the presence of two transient hardware faults. In *Computers and Digital Techniques, IEE Proceedings-*, volume 140, pages 247–254. IET. 15

[63] Fuketa, H., Hashimoto, M., Mitsuyama, Y., and Onoye, T. (2010). Alpha-particle-induced soft errors and multiple cell upsets in 65-nm 10t subthreshold sram. In *Reliability Physics Symposium (IRPS), 2010 IEEE International*, pages 213–217. IEEE. 62

[64] Gao, Q., Yu, W., Huang, W., and Panda, D. (2006). Application-transparent checkpoint/restart for mpi programs over infiniband. In *Parallel Processing, 2006. ICPP 2006. International Conference on*, pages 471–478. IEEE. 12

[65] Gibson, G. (2007). Failure tolerance in petascale computers. In *Journal of Physics: Conference Series*, volume 78, page 012022. 18, 105

[66] Gill, P., Golub, G., Murray, W., and Saunders, M. (1974). Methods for modifying matrix factorizations. *Mathematics of Computation*, 28(126):505–535. 114

[67] Gioiosa, R., Sancho, J., Jiang, S., and Petrini, F. (2005). Transparent, incremental checkpointing at kernel level: a foundation for fault tolerance for parallel computers. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 9. IEEE Computer Society. 12

[68] Golub, G. and Van Loan, C. (1996a). *Matrix computations*. Johns Hopkins Univ Pr. 21

[69] Golub, G. and Van Loan, C. (1996b). *Matrix Computations*. Johns Hopkins University Press, Baltimore, MD, 3rd edition. 114

[70] Gonzalez, A., Mahlke, S., Mukherjee, S., Sendag, R., Chiou, D., and Yi, J. (2007). Reliability: Fallacy or reality? *Micro, IEEE*, 27(6):36–45. 63

[71] Gropp, W. and Lusk, E. (2004). Fault tolerance in message passing interface programs. *Int. J. High Perform. Comput. Appl.*, 18:363–372. 45

[72] Hakkarinen, D. and Chen, Z. (2010). Algorithmic Cholesky factorization fault recovery. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–10. IEEE. 14, 20, 23

[73] Hamming, R. (1950). Error detecting and error correcting codes. *Bell System technical journal*, 29(2):147–160. 8

[74] Haque, I. and Pande, V. (2010). Hard data on soft errors: A large-scale assessment of real-world error rates in gpgpu. In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pages 691–696. IEEE Computer Society. 107

[75] Hazucha, P. and Svensson, C. (2000). Impact of cmos technology scaling on the atmospheric neutron soft error rate. *Nuclear Science, IEEE Transactions on*, 47(6):2586–2594. 62

[76] Heijmen, T. (2002). Radiation-induced soft errors in digital circuits-a literature survey. 106

[77] Hocquenghem, A. (1959). Codes correcteurs derreurs. *Chiffres*, 2(2):147–56. 8, 15

[78] Hsiao, M. (1970). A class of optimal minimum odd-weight-column sec-ded codes. *IBM Journal of Research and Development*, 14(4):395–401. 8

[79] Huang, K. and Abraham, J. (1984). Algorithm-based fault tolerance for matrix operations. *Computers, IEEE Transactions on*, 100(6):518–528. 14, 19, 111

[80] Kim, S. (2009). Reducing area overhead for error-protecting large l2/l3 caches. *Computers, IEEE Transactions on*, 58(3):300–310. 9

[81] Kim, Y. and Dongarra, J. (1996). *Fault tolerant matrix operations for parallel and distributed systems*. PhD thesis, Citeseer. 13

[82] Kumar, V., Grama, A., Gupta, A., and Karypis, G. (1994). *Introduction to parallel computing: design and analysis of algorithms*, volume 400. Benjamin/Cummings. 27

[83] Lemarinier, P., Bouteiller, A., Herault, T., Krawezik, G., and Cappello, F. (2004). Improved message logging versus improved coordinated checkpointing for fault tolerant mpi. In *Cluster Computing, 2004 IEEE International Conference on*, pages 115–124. IEEE. 11

[84] Li, K., Naughton, J., and Plank, J. (1994). Low-latency, concurrent checkpointing for parallel programs. *Parallel and Distributed Systems, IEEE Transactions on*, 5(8):874–879. 12

[85] Li, S., Chen, K., Hsieh, M., Muralimanohar, N., Kersey, C., Brockman, J., Rodrigues, A., and Jouppi, N. (2011). System implications of memory reliability in exascale computing. In *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*, pages 1–12. IEEE. 9

[86] Lin, S. and Costello Jr, D. (1983). Error control coding: Fundamentals and applications, 1983. *Prentice-Hall, Inc. Englewood Cliffs*, 3:2–3. 8

[87] Lo, J. (1994). Reliable floating-point arithmetic algorithms for error-coded operands. *Computers, IEEE Transactions on*, 43(4):400–412. 10

[88] Lu, C. (2005). *Scalable diskless checkpointing for large parallel systems*. PhD thesis, Citeseer. 19

[89] Luk, F. and Park, H. (1988a). An analysis of algorithm-based fault tolerance techniques* 1. *Journal of Parallel and Distributed Computing*, 5(2):172–184. 15, 24, 63, 77, 111

[90] Luk, F. and Park, H. (1988b). Fault-tolerant matrix triangularizations on systolic arrays. *Computers, IEEE Transactions on*, 37(11):1434–1438. 15, 107, 109, 111, 113, 114, 122

[91] Lunardini, D., Narasimham, B., Ramachandran, V., Srinivasan, V., Schrimpf, R., and Robinson, W. (2004). A performance comparison between hardened-by-design and conventional-design standard cells. In *2004 Workshop on Radiation Effects on Components and Systems, Radiation Hardening Techniques and New Developments*. 9

[92] Lyons, D. (2000). Sun screen. Available at http://www.forbes.com/forbes/2000/1113/6613068a.html. 62

[93] Maruyama, N., Nukada, A., and Matsuoka, S. (2009). Software-based ecc for gpus. In *2009 Symposium on Application Accelerators in High Performance Computing (SAAHPC'09)*. 107

[94] Maruyama, N., Nukada, A., and Matsuoka, S. (2010). A high-performance fault-tolerant software framework for memory on commodity gpus. In *Parallel and Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12. IEEE. 107

[95] Maslennikow, O., Kaniewski, J., and Wyrzykowski, R. (1998). Fault tolerant qr-decomposition algorithm and its parallel implementation. In *Euro-Par'98 Parallel Processing*, pages 1–1. Springer. 107

[96] Massey, J. and Garcia, O. (1972). Error-correcting codes in computer arithmetic. 10

[97] Mastipuram, R. and Wee, E. (2004). Soft errors impact on system reliability. *EDN, Sept*, 30. 62

[98] Meaney, P., Swaney, S., Sanda, P., and Spainhower, L. (2005). Ibm z990 soft error detection and recovery. *Device and Materials Reliability, IEEE Transactions on*, 5(3):419–427. 10

[99] Meuer, H. W., Strohmaier, E., Dongarra, J. J., and Simon, H. D. (2010). *TOP500 Supercomputer Sites*, 36th edition. (The report can be downloaded from http://www.netlib.org/benchmark/top500.html). 2, 63, 105

[100] Michalak, S., Harris, K., Hengartner, N., Takala, B., and Wender, S. (2005). 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. 61

[101] Mitra, S., Karnik, T., Seifert, N., and Zhang, M. (2005a). Logic soft errors in sub-65nm technologies design and cad challenges. In *Design Automation Conference, 2005. Proceedings. 42nd*, pages 2–4. IEEE. 9

[102] Mitra, S., Zhang, M., Mak, T., Seifert, N., Zia, V., and Kim, K. (2005b). Logic soft errors: a major barrier to robust platform design. In *Test Conference, 2005. Proceedings. ITC 2005. IEEE International*, pages 10–pp. IEEE. 9

[103] Mohanram, K. and Touba, N. (2003). Cost-effective approach for reducing soft error failure rate in logic circuits. In *Test Conference, 2003. Proceedings. ITC 2003. International*, volume 1, pages 893–901. IEEE. 9

[104] Neubauer, A., Freudenberger, J., and Kühn, V. (2007). *Coding theory: algorithms, architectures, and applications*. Wiley-Interscience. 71

[105] NVIDIA (2009). Nvidia's next generation cuda compute architecture: Fermi v1.1. Technical report, NVIDIA Corporation. 107

[106] Nvidia, C. (2011). Nvidia cuda c programming guide. *NVIDIA Corporation*. 120

[107] Park, H. (1992). On multiple error detection in matrix triangularizations using checksum methods. *Journal of Parallel and Distributed Computing*, 14(1):90–97. 15

[108] Plank, J., Beck, M., Kingsley, G., and Li, K. (1994). *Libckpt: Transparent checkpointing under unix*. University of Tennessee], Computer Science Department. 12

[109] Plank, J. and Li, K. (1994). Faster checkpointing with n+ 1 parity. In *Fault-Tolerant Computing, 1994. FTCS-24. Digest of Papers., Twenty-Fourth International Symposium on*, pages 288–297. IEEE. 13

[110] Plank, J., Li, K., and Puening, M. (1998). Diskless checkpointing. *Parallel and Distributed Systems, IEEE Transactions on*, 9(10):972–986. 13, 19

[111] Rao, R., Blaauw, D., and Sylvester, D. (2006). Soft error reduction in combinational logic using gate resizing and flipflop selection. In *Computer-Aided*

Design, 2006. ICCAD'06. IEEE/ACM International Conference on, pages 502–509. IEEE. 9

[112] Rao, T. (1974). *Error coding for arithmetic processors*. Academic Press, Inc. 10

[113] Reed, I. and Solomon, G. (1960). Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2):300–304. 8, 15

[114] Rossi, D., Timoncini, N., Spica, M., and Metra, C. (2011). Error correcting code analysis for cache memory high reliability and performance. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011*, pages 1–6. IEEE. 9

[115] Sankaran, S., Squyres, J., Barrett, B., Sahay, V., Lumsdaine, A., Duell, J., Hargrove, P., and Roman, E. (2005). The lam/mpi checkpoint/restart framework: System-initiated checkpointing. *International Journal of High Performance Computing Applications*, 19(4):479–493. 12

[116] Schreiber, R. and Van Loan, C. (1989). A storage-efficient wy representation for products of householder transformations. *SIAM J. Sci. Stat. Comput.*, 10(1):53–57. 46, 108

[117] Schroeder, B. and Gibson, G. (2007). Understanding failures in petascale computers. In *Journal of Physics: Conference Series*, volume 78, page 012022. IOP Publishing. 105

[118] Schroeder, B., Pinheiro, E., and Weber, W. (2009). DRAM errors in the wild: a large-scale field study. In *Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems*, pages 193–204. ACM. 9, 62

[119] Semiconductor, T. (2004). Soft errors in electronic memory-a white paper. *URL: http://www. tezzaron. com/about/papers/Papers. htm*. 9

[120] Sharangpani, H. and Arora, H. (2000). Itanium processor microarchitecture. *Micro, IEEE*, 20(5):24–43. 8

[121] Sheaffer, J., Luebke, D., and Skadron, K. (2007). 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*, pages 55–64. Eurographics Association. 107

[122] Shi, G., Enos, J., Showerman, M., and Kindratenko, V. (2009). On testing gpu memory for hard and soft errors. In *Proc. Symposium on Application Accelerators in High-Performance Computing*. 107

[123] Shi, Y., Zhang, X., Ni, Z., and Ansari, N. (2004). Interleaving for combating bursts of errors. *Circuits and Systems Magazine, IEEE*, 4(1):29–42. 8

[124] Shroff, G. and Bischof, C. (1992). Adaptive condition estimation for rank-one updates of qr factorizations. *SIAM journal on matrix analysis and applications*, 13:1264. 114

[125] Silva, L. and Silva, J. (1998). An experimental study about diskless checkpointing. In *Euromicro Conference, 1998. Proceedings. 24th*, volume 1, pages 395–402. IEEE. 13

[126] Slayman, C. (2005). Cache and memory error detection, correction, and reduction techniques for terrestrial servers and workstations. *Device and Materials Reliability, IEEE Transactions on*, 5(3):397–404. 8

[127] Slegel, T., Averill III, R., Check, M., Giamei, B., Krumm, B., Krygowski, C., Li, W., Liptay, J., MacDougall, J., McPherson, T., et al. (1999). Ibm's s/390 g5 microprocessor design. *Micro, IEEE*, 19(2):12–23. 10

[128] Streitz, F., Glosli, J., Patel, M., Chan, B., Yates, R., Supinski, B., Sexton, J., and Gunnels, J. (2006). Simulating solidification in metals at high pressure: The

drive to petascale computing. In *Journal of Physics: Conference Series*, volume 46, page 254. IOP Publishing. 1

[129] Team, I. (2008). Overview of the ibm blue gene/p project. *IBM Journal of Research and Development*, 52(1/2):199–220. 8

[130] Tendler, J., Dodson, J., Fields, J., Le, H., and Sinharoy, B. (2002). Power4 system microarchitecture. *IBM Journal of Research and Development*, 46(1):5–25. 9

[131] The MPI Forum (2099). MPI: A Message-Passing Interface Standard, Version 2.2. Technical report. 45

[132] Tipton, A., Pellish, J., Reed, R., Schrimpf, R., Weller, R., Mendenhall, M., Sierawski, B., Sutton, A., Diestelhorst, R., Espinel, G., et al. (2006). Multiple-bit upset in 130 nm cmos technology. *Nuclear Science, IEEE Transactions on*, 53(6):3259–3264. 62

[133] Tomov, S., Dongarra, J., and Baboulin, M. (2010). Towards dense linear algebra for hybrid gpu accelerated manycore systems. *Parallel Computing*, 36(5-6):232–240. 106

[134] Udipi, A., Muralimanohar, N., Chatterjee, N., Balasubramonian, R., Davis, A., and Jouppi, N. (2010). Rethinking dram design and organization for energy-constrained multi-cores. In *ACM SIGARCH Computer Architecture News*, volume 38, pages 175–186. ACM. 9

[135] Vaidya, N. (1998). A case for two-level recovery schemes. *Computers, IEEE Transactions on*, 47(6):656–666. 13

[136] Van De Geijn, R. and Watts, J. (1997). SUMMA: Scalable universal matrix multiplication algorithm. *Concurrency Practice and Experience*, 9(4):255–274. 96

[137] Walters, J. and Chaudhary, V. (2006). Application-level checkpointing techniques for parallel programs. *Distributed Computing and Internet Technology*, pages 221–234. 12

[138] White, M., Qin, J., and Bernstein, J. (2011). A study of scaling effects on dram reliability. In *Reliability and Maintainability Symposium (RAMS), 2011 Proceedings-Annual*, pages 1–6. IEEE. 62

[139] Wilkerson, C., Alameldeen, A., Chishti, Z., Wu, W., Somasekhar, D., and Lu, S. (2010). Reducing cache power with low-cost, multi-bit error-correcting codes. In *ACM SIGARCH Computer Architecture News*, volume 38, pages 83–93. ACM. 9

[140] Woodbury, M. (1949). The stability of out-input matrices. *Chicago, IL*. 93

[141] Woodbury, M. (1950). Inverting modified matrices. *Memorandum report*, 42:106. 93

[142] Yim, K. and Iyer, R. (2011). Hauberk: Lightweight silent data corruption error detectors for gpgpu. *In Proceedings of the 17th Humantech Thesis Prize (Also in IPDPS 2011)*. 107

[143] Yoon, D. and Erez, M. (2009). Memory mapped ecc: low-cost error protection for last level caches. In *ACM SIGARCH Computer Architecture News*, volume 37, pages 116–127. ACM. 9

[144] Yoon, D. and Erez, M. (2010). Virtualized and flexible ecc for main memory. *ACM SIGARCH Computer Architecture News*, 38(1):397–408. 9

[145] Ziegler, J. (1996). Terrestrial cosmic rays. *IBM Journal of Research and Development*, 40(1):19–39. 62

Vita

Peng Du was born in Dalian, P. R. China. He received his high-school education from the No.24 High School, Dalian, P.R.China from 1995 to 1998. He obtained a Bachelor degree in Measuring and Control, and a master degree in Optical Engineering from Beijing University of Aeronautics and Astronautics, Beijing, P.R.China in 2002 and 2005, respectively.

He came to the University of Tennessee, Knoxville, to pursue a doctoral degree in August 2005. During his graduate studies, he worked as a graduate research assistant in Innovative Computing Laboratory (ICL) under the guidance of Dr. Jack Dongarra. He was involved in the following federally funded projects related to high performance computing: LAPACK/ScaLAPACK, CLAPACK, Fault Tolerance Linear Algebra (FT-LA) and MAGMA. His current research interests include high performance computing, parallel and distributed computing. Peng Du is expected to receive a Doctor of Philosophy degree in Computer Science in August 2012.