Parallel Reduction to Hessenberg Form with Algorithm-Based Fault Tolerance

Yulu Jia, George Bosilca, Piotr Luszczek
University of Tennessee
Knoxville

Jack J. Dongarra
University of Tennessee
Knoxville, Oak Ridge National Laboratory and University of Manchester

ABSTRACT

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

Categories and Subject Descriptors

G.4 [**Mathematics of Computing**]: Mathematical Software—*Algorithm design and analysis, reliability and robustness*

General Terms

Algorithms, Reliability

Keywords

Algorithm-based fault tolerance, Hessenberg reduction, ScaLAPACK, Dense linear algebra, Parallel numerical libraries

1. INTRODUCTION

Mainstream supercomputers are well into the peta-scale era, with the number of components on a sharp increase over the years. Only one year ago, Jaguar, hosted at the Oak Ridge National Laboratory, included 224,162 cores. During its 537 days of operation, an average of 2.33 failures per day [37] occured, or on average less than 10 continuous hours of operation. Already today, the new configuration of Jaguar, called Titan, has a remarkable 299,008 Opteron cores, over 18,688 compute nodes, without taking into account the number of computing units on the accelerators, which would put the count in millions. This sharp increase in the number of components is likely to continue [18], in which case even the most optimistic predictions about the failure rate of a particular component, in terms of tens of years, depict a gloomy future. A future where the Mean Time To Interrupt (MTTI) of the entire machine falls under a few hours, drastically affecting individual applications running on the system [41], with a lasting impact, not only on the scientific throughput, but directly on the cost of the scientific simulations.

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

Libraries with eigenvalue solvers are the method of choice for spectral clustering of graphs [43] and eigenvector centrality and its widely known form: the PageRank [2, 12, 13, 34]. Hessenberg form is a common intermediate representation for eigenvalue calculations.

The Hessenberg reduction [42] is an important step in calculating the eigenvalues and/or eigenvectors of a dense non-symmetric matrix or for solving the regular generalized eigenvalue problem. The orthogonal transformations are commonly used for this reduction for their guaranteed stability even though their accumulated cost becomes high in terms of both: computation and communication. One of the most common algorithms that stably computes eigenvalues of a dense matrix is the QR algorithm [42, 24]. There are two steps in the QR algorithm. In the first step, the matrix A is reduced to Hessenberg form H by a sequence of similarity transformations: $A = QHQ^\top$. A Hessenberg form, H, is a square matrix in which all the entries below the first subdiagonal are 0. The second step further reduces H to an upper triangular form T. The elements on the diagonal of T are the eigenvalues of matrix A. A Hessenberg matrix is also required for obtaining Hessenberg triangular form of the matrix pair (A,B) of the regular generalized eigenvalue problem of the form $(A-\lambda B)x = 0$ when using the QZ algorithm that originated from the implicitly shifted QR algorithm [22, 23]. More recent work involves efficient implementations of various QR iter-

SC13 November 17-21, 2013, Denver, CO, USA

ACM 978-1-4503-2378-9/13/11 ...$15.00
http://dx.doi.org/10.1145/2503210.2503249.

ation methods on modern multicore and distributed memory systems [30, 11, 26, 31, 27].

The Hessenberg reduction routine is provided in virtually all major numerical libraries, both for shared and distributed memory architectures. LAPACK [1] contains the routine **DGEHRD** for Hessenberg reduction. ScaLAPACK [4] is the open source linear algebra library providing LAPACK equivalent functionalities for distributed memory machines, its Hessenberg reduction routine is **PDGEHRD**. Commercial numerical libraries often provide optimized implementations of LAPACK and ScaLAPACK for specific architectures (such as LibSci for Cray XT architectures).

The high arithmetic complexity overall and low arithmetic intensity of its building blocks make the Hessenberg reduction a rather costly operation. In spite its high computational complexity of $O(\frac{10}{3}n^3)$, the Hessenberg reduction only achieves a fraction of the theoretical machine peak performance (unlike one-sided factorizations such as LU and QR). While its long running time makes the Hessenberg reduction routine more exposed to fail-stop failures, with few exceptions, no algorithmic solutions to tolerate fail-stop failures have been proposed. Common fault tolerant techniques such as checkpointing and algorithm based fault tolerance (ABFT) have limitations when applied to the Hessenberg reduction. Checkpointing stores application data to stable memory at certain time intervals. In Hessenberg reduction, the whole trailing matrix, which accounts for a significant portion of application data, is modified very frequently, annihilating even the potential benefits of incremental checkpointing. Moreover, the checkpointing technique introduces too much overhead due to frequent write-to-memory accesses (either hard disk or remote main memory). Similarly, the usual ABFT techniques cannot provide protection for the lower left part of the matrix during the reduction.

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

The rest of the paper is organized as follows: Section 2 presents previous research work in fault tolerance for matrix computations. Section 3 introduces the Hessenberg reduction algorithm and its implementation, and highlights the challenges in applying ABFT to the Hessenberg reduction. Sections 4 and 5, describe the encoding used to provide the redundancy on the input matrix and the algorithm to maintain it through the computation. Section 6 provides a formal analysis of the overhead and costs, while Section 7 experimentally validates them. Section 8 summarizes the results of this paper and presents future work.

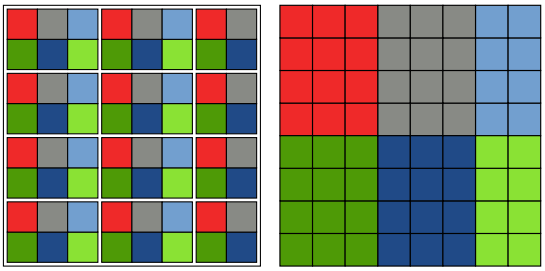
2. RELATED WORK

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

The major advantage of the C/R approach is the generality: it can be applied to a wide range of applications not only linear algebra software. In the C/R technique, consistent snapshots of program data in main memory are saved to stable storage (usually a disk drive) at certain time intervals. Once a failure happens, the entire application rolls back to the latest snapshot and computation resumes from that point on (we ignore the complexity related to the consistent view of the entire application in terms of message or file accesses). In a distributed environment the major cost of this method comes from obtaining the consistent snapshots and disk access to write the snapshots, which highlights the major drawback of such approaches, the relatively high overhead. Langou and Dongarra [33] investigated several checkpoint/recovery techniques and a checkpoint-free lossy fault tolerant technique for parallel iterative methods. Robert and Vivien [8, 10] presented a unified model for several common checkpoint/restart protocols, extended in [14] to cover process replication. Diskless checkpointing [39, 25, 35] stores checkpoints in main memory to avoid disk accesses.

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

Huang and Abraham [29] proposed a system-level method to tolerate errors in matrix computations in the context of systolic arrays. The matrix is encoded and operations are carried out on the encoded data. A single failure can be corrected during the computation. This technique has been successfully applied to matrix addition and multiplication, scalar product and the LU decomposition. Later, Luk and Park [36] extended Huang's method to make it more efficient to correct transient errors in Gaussian elimination and QR decomposition on systolic arrays. They proposed methods to compute checksums of the original matrix. Their method does not need a rollback in order to correct the error. Kim and Plank [32] presented a technique based on checksum and reverse computation to tolerate process failures in matrix operations. Chen and Dongarra [16] designed and implemented an algorithm based fault tolerance algorithm to tolerate process failures in the ScaLAPACK PDGEMM routine. Bosilca and Langou [9] also designed and implemented an algorithm based fault tolerance algorithm for the ScaLAPACK PDGEMM routine and developed performance models to predict its overhead. Hakkarinen and Chen [28] implemented an algorithm based fault tolerance algorithm for Cholesky factorization, an algorithm tolerating a single process failure at a time. Du and Dongarra et al. [21] designed algorithm based fault tolerance algorithms for LU and QR factorizations and implemented them in the ScaLAPACK framework. Their methods have a low overhead and scale well with the increase of matrix size and process grid size. Davies et al. [17] also applied the ABFT technique to HPL [38, 19] which is a highly optimized right-looking LU factorization. Yao and Wang [46] proposed a non-stop algorithm based fault tolerant



(a) Matrix View (b) Process View

Figure 1: A matrix mapped to a 2×3 process grid.

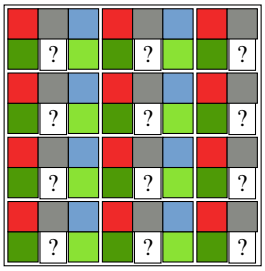


Figure 2: Global view of the matrix when a process fails.

scheme to recover the solution vector from fail-stop process failures in HPL 2.0. Bland et al. [5, 6] proposed a checkpoint-on-Failure protocol for fault recovery in dense linear algebra.

3. ScaLAPACK HESSENBERG REDUCTION

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

3.1 2D Block Cyclic Data Distribution

There are several possible ways to distribute a matrix across distributed memory machines. Among them, the 2D block cyclic distribution was chosen for ScaLAPACK based on its good scalability properties and the ability to use Level 3 BLAS routines. Figure 1 illustrates the 2D block cyclic data distribution with an example. A matrix is partitioned into small $nb \times nb$ square blocks. nb is called the blocking factor. These blocks are mapped to a 2×3 process grid. If a data block is mapped to a process it means the data block is physically stored in the local memory associated with that process. All the data blocks assigned to the same process are stored contiguously. Figure 1(a) shows the global matrix from a logical point of view. Each of the six colors represents a process. Data blocks are assigned to the processes in a round-robin fashion in both horizontal and vertical directions. Figure 1(b) is the processes' view of the distribution. Same as in Figure 1(a), each color represents a process. Each process's own part of the matrix is stored contiguously in its local memory in column major. Each process is assigned roughly the same amount of data, which means they are responsible for roughly the same amount of total floating point operations. Block algorithms in ScaLAPACK proceed from left to right. As the algorithm continues, each process has roughly the same amount of work load left. This avoids prolonged idle time and keeps all the processes busy most of the time.

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

3.2 Failure Model Under 2D Block Cyclic Data Distribution

In this work, we consider process failures. When a process fails in the process grid the data resident on that process will be all gone. Figure 2 shows the status of the matrix when a process failure happens. The colored squares are the data blocks owned by the live processes. The blank squares with question marks are the data blocks owned by the failed process. After we have recovered the process grid, the replacement process contains invalid data. These invalid data blocks need to be recovered to their state before the failure happened. If we continue the Hessenberg reduction without recovering the lost data the final result will be completely wrong.

3.3 Non-blocked Hessenberg Reduction

The Hessenberg reduction takes a general nonsymmetric square matrix $A \in \mathbb{R}^{n \times n}$ and decomposes it: $A = UHU^\top$. U is an orthogonal matrix, H is a Hessenberg matrix. The non-blocked Hessenberg reduction is an iterative process, $n-1$ Householder transformations are applied to the matrix A from left and right $H_{n-1}H_{n-2}\ldots H_2H_1AH_1^\top H_2^\top \ldots H_{n-2}^\top H_{n-1}^\top = H$. The orthogonal matrix U is $U = H_1H_2\ldots H_{n-2}H_{n-1}$. A Householder transformation H_i can be generated efficiently [42, page 83]. The non-blocked version uses Level 2 BLAS operations which have a low flop/transfer ratio and are slow. ScaLAPACK uses a blocked Hessenberg reduction algorithm which has a larger number of efficient Level 3 BLAS operations.

3.4 Blocked Hessenberg Reduction

In the blocked Hessenberg reduction [20] nb (the blocking factor in the 2D block cyclic distribution) Householder reflectors are accumulated and applied to the trailing matrix together using Level 3 BLAS. Using the WY representation [3, 40] the reduction can be written as:

$$H_{nb}^\top \cdots H_1^\top A H_1 \cdots H_{nb} \quad = \quad A - VW - YV^\top \tag{1}$$

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

Algorithm 1 PDGEHRD

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

Algorithm 1 is the pseudo code for **PDGEHRD**. The function call **PDLAHRD** reduces a panel with a sequence of Householder transformations. It takes the trailing submatrix, reduces the first panel of nb columns, and overwrites the bottom part of the panel with the Householder reflectors. Although this panel factorization routine only modifies the panel, it has a data dependency on the trailing matrix. In other words, once the trailing matrix is modified and we lose data inside the panel, the panel factorization cannot be repeated. This poses a challenge for our fault tolerant algorithm design as explained in later sections.

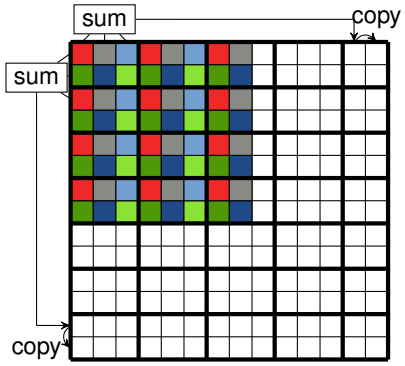
Figure 3 illustrates one iteration of the ScaLAPACK Hessenberg reduction algorithm. In Figure 3(a) the yellow part is part of the final result of the Hessenberg matrix. This part will not be touched once they have been computed. Columns in the green part are the Householder reflectors used to transform the matrix. The red part is the trailing matrix which will be reduced in future iterations. As other factorizations in ScaLAPACK, **PDGEHRD** is an iterative algorithm. In each iteration, **PDLAHRD** reduces the first block column which is called the *panel*. This call produces the final result of the desired Hessenberg matrix (the yellow upper trapezoid in Figure 3(b)) and *nb* Householder reflectors (the green lower trapezoid in Figure 3(b)). This call also generates intermediate matrices V and Y which are used by the **PDGEMM** and **PDLARFB** immediately following the panel reduction to update the trailing submatrix. When this iteration finishes, we get a smaller trailing submatrix: the red part on the right in Figure 3(e). In the next iteration, the same process is repeated on the shrunk trailing submatrix which further reduces it to a smaller size. This algorithm is a right-looking algorithm, in that, the updates only access data to the right of the current panel. Matrix entries to the left of the current panel are never touched again after the panel computation proceeded to the right.

4. ENCODING THE INPUT MATRIX

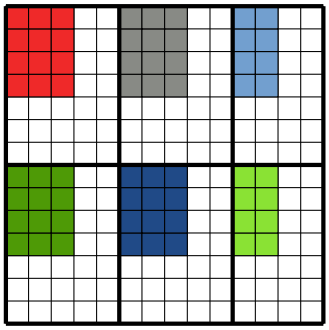
The essential part of ABFT technique is to expand the original matrix data with redundant data and maintain the relationship between the original matrix and the redundant data through computation. In our fault tolerant Hessenberg reduction algorithm, we chose to append the matrix with row checksums to the right of the original matrix. We show the checksum scheme with an example in Figure 4. A matrix of $N \times N$ blocks is mapped to a $P \times Q$ process grid in the 2D block cyclic fashion (here $N = 8$, $P = 2$, $Q = 3$). Each process will be assigned at most $\lceil N/P \rceil \times \lceil N/Q \rceil$ data blocks. We add $\lceil N/Q \rceil \times 2$ block columns to the right as checksum blocks. Data blocks in the same position of different processes of the same process row are added together element-wise to form a checksum block. This checksum block is duplicated and stored next to itself. The details are shown in Figure 4(a).

We also expand the original matrix with checksum blocks at the bottom. Only the storage is allocated, the actual checksums are not actually calculated in the beginning. The extra storage at the bottom will be used for pseudo checksums of the V matrix which contains the block Householder reflectors. The number of pseudo checksum block rows at the bottom is the same as the number of checksum block columns to the right of the matrix. And each pseudo checksum block is calculated in this way: pretend the matrix is distributed over a $Q \times Q$ process grid (despite that it is actually distributed over a $P \times Q$ grid), then we sum corresponding data blocks in different processes in the same process column element-wise and obtain a pseudo checksum block. The summing relationship is also shown in Figure 4(a). In this figure, the pseudo checksum block is the sum of the first three blocks, because had we distributed the matrix over a 3×3 process grid, the first three data blocks would be the first blocks in the three processes in the their respective local matrices.

These checksum blocks are treated as normal matrix data and distributed across the process grid. Figure 4(b) shows each process's local matrix containing the checksum blocks. Each black box represents a process. The white blocks are the checksum blocks. Note that the example in Figure 4 uses a small process grid, the checksum data are relatively large compared to the original input matrix. But in practice the process grid is rarely this small. The checksum data only accounts for a small portion of the input matrix when the size of the process grid increases.



(a) Matrix View



(b) Process View

Figure 4: An encoded matrix mapped to a 2×3 process grid.

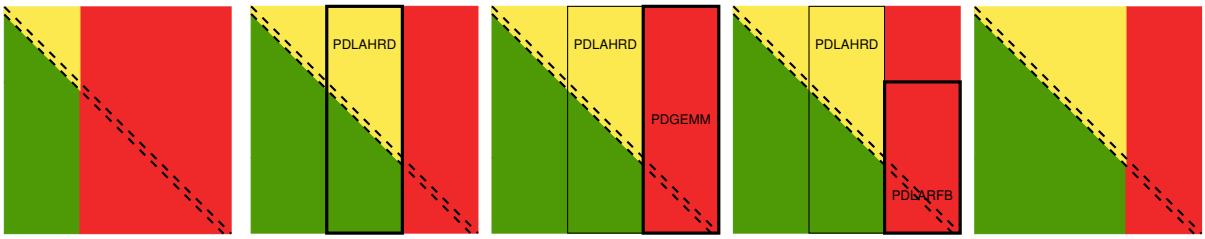
5. THE ALGORITHM

Algorithm 2 ABFT Hessenberg Reduction (non-delayed)

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

5.1 Maintaining Data Redundancy in the Factorization

Two versions of ABFT Hessenberg reduction algorithms are shown in Algorithm 2 and Algorithm 3. These two versions are mathematically equivalent, but their actual implementations have different performance characteristics due to the behavior of PBLAS routines. We use Algorithm 2 to explain how the method works. In iteration i, we refer to the Q block columns starting from $\lfloor i/Q \rfloor$ to $\lceil i/Q \rceil$



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

Figure 3: One iteration of PDGEHRD

(inclusive) as the *panel scope*. N is the dimension of the original matrix, nb is the blocking factor.

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

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

Lines 8 and 9 record the state of the panel scope after each panel factorization. These two lines also record the state of Y and T which are the results of panel factorization. Y and T are stored in a separate workspace apart from A. The newly calculated Householder reflectors are stored in-place in the lower portion of A. These reflectors do not have any protection mechanism. Unlike a recent implementation of QR factorization [21], the panel factorization in the Hessenberg reduction has a data dependence on the trailing submatrix. The **PDLAHRD** routine needs the unmodified trailing submatrix to factorize the panel. This means that the panel factorization result has to be protected right away after it is obtained. We do not delay the recording of the state of the panel result (V, Y and T) till either the **PDGEMM** call or the **PDLARFB** call. This is because in the case of a process failure, data loss would occur in the panel result. This is possible for a failure that happens after the panel factorization and after the start of the trailing matrix update. The panel result cannot be recovered in that case by a rollback of the panel and re-factorizing it despite the fact that we can manage to recover the panel data right before its factorization. It is possible to reverse the effect of the trailing matrix update if we store the V, Y and T matrices which were used to update the trailing matrix. But they are not available since these three matrices are exactly what are supposed to be recovered.

Line 12 recovers data that were lost due to a process failure. The details of the recovery procedure are explained in section 5.3.

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

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

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

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

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

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

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

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

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

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

Algorithm 3 ABFT Hessenberg Reduction (delayed)

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

5.2 Checksum Duplication

We protect the row checksums appended to the right of the matrix by maintaining two copies of exactly the same checksums. Because the checksums are distributed as normal matrix data over the process grid, any process failure will also cause loss of the checksums resident on the failed process. To solve this problem we maintain two copies of the checksums as in [21]. Both are kept valid through updating them independently. These two copies are stored next to each other so they are distributed to different process columns. Since only one process could fail, we always have one valid copy and can use this copy to recover the other copy. This approach does not need dedicated checksum processes, and does not have to assume that the checksum processes never fail. This approach also has good load balancing property. These traits are preferable because it does not require users of the ScaLAPACK library to change their application or the way they run their application. It is also easier to implement since the code has clear logic. The update of the checksum data does not need special treatment, the only thing needed is to change the dimensions of the trailing matrix during the update step of the original ScaLAPACK code.

5.3 Recovery

When the Hessenberg factorization is in progress, the matrix can be divided into different areas based on the status of the data as shown in Figure 5. Different areas of the matrix data need different methods to recover.

The recovery process:

1. Recover the runtime system. Replace the lost process and restore the process grid.
2. Recover lost checksums using the duplication.

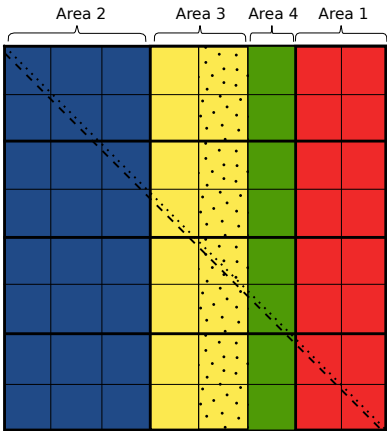


Figure 5: Partitions of the matrix. The dotted block column in area 3 has just been factorized. Area 1 (red) is the trailing matrix after the current panel scope. Area 2 (blue) is the finished part of the matrix. Area 3 (yellow) is the block columns in the current panel scope that have been factorized. Area 4 (green) is part of the current panel scope which belongs to the trailing matrix.

3. Recover lost data in area 1 and 2 using the row checksum on the right and the data on the live processes. First calculate the sum of data blocks on different processes in the same process row, then subtract this partial sum from the checksum to get the lost data blocks. Send the recovered data blocks to the replacement process.
4. Recover the lost data in area 3 using the checkpoint.
5. Recover the lost data in area 4. First retrieve the backup data from the snapshot, then apply all the left updates and right updates since the last snapshot.
6. Resume computation as usual. Ready to recover from the next failure.

6. PERFORMANCE ANALYSIS

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

These extra floating point operations are all in matrix matrix multiplies which are efficiently implemented, so the overhead in terms of floating point operation count can also be interpreted as overhead in terms of running time. From the formula above, we observe that as the size of the matrix is big enough, N tends to infinity, and the terms containing N in the denominator tend to 0:

$$\lim_{N\to\infty} Overhead = \frac{1}{5Q} \tag{2}$$

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

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

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

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

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

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

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

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

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

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

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

$$N\,(N/Q) + nb\,(N/Q)$$

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

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

7. EXPERIMENTS

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

Titan is a hybrid supercomputing system located at Oak Ridge National Laboratory. It is the fastest parallel computer on the current TOP500 list (Nov., 2012). Since we are only using the traditional CPU section of the machine, information about NVIDIA GPUs on Titan is not reported. Titan is composed of 18,688 nodes with 299,008 cores, for a CPU peak performance in double precision of 2.63 PFlop/s.

7.1 Overhead Without Failure

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

Figure 7 shows the overhead of Algorithm 3 on Titan. We see that the performance overhead keeps dropping in the beginning, but it starts to go up again at grid size 96×96. There are three main reasons which cause the overhead increase. Firstly, when we delay the updates of the global checksums at the end of each panel scope, these updates resulting from each panel factorization are applied sequentially. When the process grid size increases, the number of panels in the panel scope also becomes larger. The sequence of updates to the global checksums takes longer to finish. Secondly, when updating the checksums separately from the trailing matrix, the updates (**PDLARFB** and **PDGEMM**) are applied to a tall and skinny matrix. These two routines perform best when applied to more rectangular matrices. Also, splitting the calls to these routines disrupts their internal communication pipeline that hides latency and creates additional synchronization points upon exit and then entry into these routines. Thirdly, updating the checksums separately causes extra communication between processes owning V and processes owning the checksums. These overheads are critical in the context of an already communication-rich operation such as the Hessenberg reduction, and they inhibit scalability as the Figure 7 indicates.

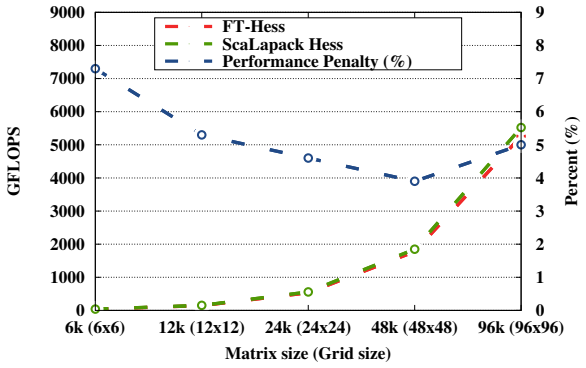


Figure 7: Overhead of FT-Hess without failures. Platform: Titan, $NB = 80$, Algorithm 3

7.2 Overhead With Failure

Figure 6(b) shows the performance and performance overhead of our fault tolerant Hessenberg reduction algorithm on Titan when one failure happens in the factorization. Compared with Figure 6(a) the performance overhead shown in the Figure 6(b) includes one more factor: the recovery overhead. The recovery process involves a global row-wise reduction operation on the entire global matrix. Before this global reduction the data on the replacement process are set to zero. This global reduction operation calculates a new global checksum. The lost data belonging to Area 1 and Area 2 in Figure 5 are recovered using the new checksum and the old checksum that we have been maintaining along with the factorization. The cost of this global reduction depends on the bandwidth of the link between the processes. This cost accounts for a small portion of the total running time of the Hessenberg reduction. Figure 6(b) shows that, even with the recovery cost included, the total overhead of our fault tolerant Hessenberg reduction algorithm is still very low and it decreases as the problem increases. It is down to 4.03% for the matrix of size 96000 (process grid dimension: 96×96).

7.3 Numerical Stability After Recovery From a Failure

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

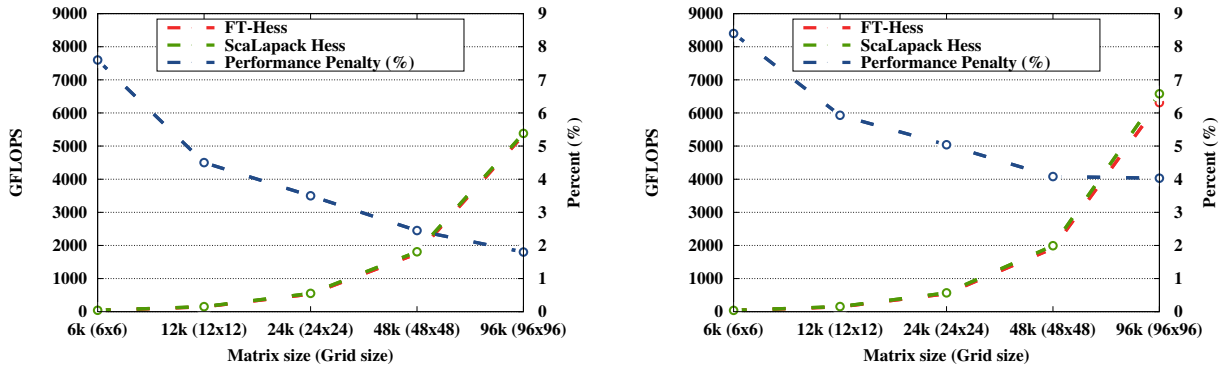
Floating point numbers are represented in IEEE 754 format in modern computers, floating point operations are not carried out in exact arithmetic. Standard error analysis for the reduction of a general matrix A to Hessenberg form H by means of similarity transformations shows the process to be backward stable [44, page 363]. In particular, the process reduces a nearby problem $\hat{A} = A + E$ into $\hat{H}$ with a set of similarity transformations U and at the end we get:

$$\hat{H} = U^{\top} \hat{A} U \tag{3}$$

The bound on the residual error E [44, page 351] is

$$\|E\|_F / \|A\|_F \leq \phi(N)\varepsilon \tag{4}$$

where ϕ is a low degree polynomial [44, page 351, Table 1] and ε is the *unit roundoff* (machine precision). This is an expected result since the transformation only employs orthogonal transformations and therefore does not introduce rounding errors larger than those already existing in the data. In fact, its backward error analysis



(a) Without failures

(b) With failures

Figure 6: Overhead of FT-Hess without failures and with one failure. Platform: Titan, $NB = 80$, Algorithm 2

has been used in a scheme that detects soft errors in linear algebra operations at runtime [7].

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

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

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

To show backward stability of the recovery process, we use the technique of projecting the error (resulting from a fault) back into the original matrix A [36]. We then exploit the fact that the backward error analysis already involves a perturbation to A and the reduction is shown to provide a solution to a nearby problem $\hat{A}$ with a satisfactory bound on the perturbing error. Then, using a standard dot-product error analysis [15], we show that the numerical stability is not affected by the recovery from the fault. The dot-product analysis applies to our checksum procedure with only a slight modification.

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

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

Errors from encoding the input matrix. The initial checksums are calculated through a simple summation operation. On a $P \times Q$ process grid, each checksum element involves at most $Q-1$ addition operations . The rounding error (denoted by E_1) introduced by encoding the input matrix is bounded by

$$E_1 \leq (Q-1)\,\varepsilon \tag{5}$$

This upper bound is reached in the worst case scenario when rounding errors happen in every element and all have the same sign. In reality, rounding errors do not happen for every operation and/or do not all have the same sign. A pair of rounding errors with opposite signs will cancel each other out. The actual error is much smaller than the upper bound – the suggested approximation is the square root of quantities dependent on the problem size [45].

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

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

$$E_2 \leq (Q-1)\,\varepsilon \tag{6}$$

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

$$E_3 = E_1 + E_2 \leq 2\,(Q-1)\,\varepsilon \tag{7}$$

If the failure happens in the i-th iteration, denote the accumulated transformations so far by $U_{(i)}$, we have

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

where

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

Again, this is the theoretical upper bound assuming the worst possible cases. In reality rounding errors are mostly likely random, so

Table 1: Residual Comparison

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

they will cancel each other out. The recovery process will not cause observable extra backward errors.

Table 1 shows a comparison of the residual r obtained in our fault tolerant algorithm when a failure happens and the residual obtained in the fault-free ScaLAPACK routine. We can see that our fault tolerant algorithm computes answers on the same order of magnitude as the original ScaLAPACK algorithms, with minor differences due to the randomness of the initial matrix and the lack of bitwise reproducibility of the algorithm. Overall, our fault tolerant Hessenberg reduction algorithm is as backward stable as the ScaLAPACK version.

8. CONCLUSIONS AND FUTURE WORK

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

9. ACKNOWLEDGMENTS

The authors would like to thank the NSF for funding through grants 0904952 and 1063019, JST Japan, and DOE INCITE through the Performance End Station PEAC Project – this research used resources of the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725. The authors thank Peng Du, Aurelien Bouteiller and Thomas Herault for their valuable contribution and challenging discussions.

10. REFERENCES

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

[2] M. Berry and M. Browne. *Understanding Search Engines: Mathematical Modeling and Text Retrieval*. SIAM, Philadelphia, Second edition, 2005.

[3] C. H. Bischof and C. V. Loan. The WY Representation for Products of Householder Matrices. In *Parallel Processing for Scientific Computing*, pages 2–13, 1985.

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

[5] W. Bland, P. Du, A. Bouteiller, T. Herault, G. Bosilca, and J. Dongarra. A checkpoint-on-failure protocol for algorithm-based recovery in standard MPI. In *Proceedings of the 18th international conference on Parallel Processing*, Euro-Par'12, pages 477–488, 2012.

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

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

[8] G. Bosilca, A. Bouteiller, É. Brunet, F. Cappello, J. Dongarra, A. Guermouche, T. Hérault, Y. Robert, F. Vivien, and D. Zaidouni. Unified Model for Assessing Checkpointing Protocols at Extreme-Scale. Technical Report RR-7950, INRIA, October 2012.

[9] G. Bosilca, R. Delmas, J. Dongarra, and J. Langou. Algorithm-based fault tolerance applied to high performance computing. *J. Parallel Distrib. Comput.*, 69(4):410–416, April 2009.

[10] M. Bougeret, H. Casanova, Y. Robert, F. Vivien, and D. Zaidouni. Using Group Replication for Resilience on Exascale Systems. Technical Report RR-7876, INRIA, February 2012.

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

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

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

[14] H. Casanova, Y. Robert, F. Vivien, and D. Zaidouni. Combining Process Replication and Checkpointing for Resilience on Exascale Systems. Technical Report RR-7951, INRIA, May 2012.

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

[16] Z. Chen. *Scalable techniques for fault tolerant high performance computing*. PhD thesis, University of Tennessee, Knoxville, TN, USA, 2006.

[17] T. Davies, C. Karlsson, H. Liu, C. Ding, and Z. Chen. High Performance Linpack Benchmark: A Fault Tolerant Implementation Without Checkpointing. In *Proceedings of the International Conference on Supercomputing*, ICS '11, pages 162–171, New York, NY, USA, 2011. ACM.

[18] J. Dongarra, P. Beckman, and T. Moore. The international Exascale software project roadmap. *Int. J. High Perform. Comput. Appl.*, 25(1):3–60, February 2011.

[19] J. J. Dongarra, P. Luszczek, and A. Petitet. The LINPACK

benchmark: Past, present, and future. *Concurrency and Computation: Practice and Experience*, 15:1–18, 2003.

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

[21] P. Du, A. Bouteiller, G. Bosilca, T. Herault, and J. Dongarra. Algorithm-based fault tolerance for dense matrix factorizations. *SIGPLAN Not.*, 47(8):225–234, February 2012.

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

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

[24] G. H. Golub and C. F. V. Loan. *Matrix Computations*. The John Hopkins University Press, 4th edition, December 27 2012. ISBN-10: 1421407949, ISBN-13: 978-1421407944.

[25] L. A. B. Gomez, N. Maruyama, F. Cappello, and S. Matsuoka. Distributed Diskless Checkpoint for Large Scale Systems. In *Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on*, pages 63–72, 2010.

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

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

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

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

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

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

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

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

[34] A. Langville and C. Meyer. *Google's PageRank and Beyond: The Science of Search Engine Rankings*. Princeton University Press, 2006.

[35] C.-D. Lu. *Scalable Diskless Checkpointing for Large Parallel Systems*. PhD thesis, University of Illinois, Urbana, Illinois, USA, 2005.

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

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

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

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

[40] R. Schreiber and C. V. Loan. A storage efficient WY representation for products of householder transformations. *SIAM Journal on Scientific and Statistical Computing*, 10, 1989.

[41] B. Schroeder and G. A. Gibson. Understanding failures in petascale computers. *Journal of Physics: Conference Series*, 78, 2007.

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

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

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

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

[46] E. Yao, R. Wang, M. Chen, G. Tan, and N. Sun. A Case Study of Designing Efficient Algorithm-based Fault Tolerant Application for Exascale Parallelism. In *Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 438–448, May 2012.