Algorithm-based Fault Tolerance for Dense Matrix Factorizations

Peng Du
Innovative Computing Lab
UT, Knoxville
du@eecs.utk.edu

Aurelien Bouteiller
Innovative Computing Lab
UT, Knoxville
bouteill@eecs.utk.edu

George Bosilca
Innovative Computing Lab
UT, Knoxville
bosilca@eecs.utk.edu

Thomas Herault
Innovative Computing Lab
UT, Knoxville
herault@eecs.utk.edu

Jack Dongarra
Innovative Computing Lab
UT, Knoxville
dongarra@eecs.utk.edu

ABSTRACT

Dense matrix factorizations like LU, Cholesky and QR are widely used for scientific applications that require solving systems of linear equations, eigenvalues and linear least squares problems. Such computations are normally carried out on supercomputers where the ever-growing scale induces a fast decrease of the Mean Time To Failure (MTTF). This paper proposes a new algorithm-based fault tolerant (ABFT) approach, designed to survive fail-stop failures during dense matrix factorizations in extreme conditions such as the absence of any reliable components, and the possibility of loosing both data and checksum from a single failure. Both left and right factorization results are protected by ABFT algorithms, and fault-tolerant algorithms derived from this solution can be directly applied to a wide range of dense matrix factorizations, with minor modifications. Theoretical analysis shows that the overhead is sharply decreasing with the number of computing units and the problem size. We implemented the ABFT versions of LU based on ScaLAPACK as a demonstration. Experimental results on the Kraken supercomputer validate the theoretical evaluation.

Categories and Subject Descriptors

System Software [**Software approaches for fault tolerance and resilience**]: Software for high-performance computing

General Terms

Fault-tolerance

Keywords

ABFT, Fault-tolerance, ScaLAPACK

1. INTRODUCTION

As today's high performance computers paced into petaflops through the increase of system scale, the number of system component such as CPU cores, memory, networking, and storage grows enormously. One of the most powerful Petaflop scale machines, Kraken [2], from National Institute for Computational Sciences and University of Tennessee, harnessed as many as 98,928 cores to reach its peak performance of 1.02 Petaflops to rank No.7 on the November 2010 Top500 list. With the increase of system scale and chip density, the reliability and availability of such systems has not been improved at the same rate. Many types of failures can hit a distributed computing system [17], but the focus of this paper is on the most common occurrence: fail-stop model. In this type of failure, a process completely stops responding, which triggers the loss of a critical part of the computational global state. To be more realistic, we assume failure could occur at any time and can affect both checksum and matrix data. The mean-time-to-failure (MTTF) has been measured to drop as low as under 2 hours [16] on some large scale platforms. It has been shown that, at a certain critical point, adding computing units actually increases 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 [27]. It is estimated that the MTTF of High Performance Computing (HPC) systems might drop to about one hour in the near future [7]. Without a drastic change at the algorithmic level, such a failure rate will certainly prevent capability applications to progress.

Exploring techniques for creating a software system and programming environment capable of delivering computation at extreme scale that is both resilient to faults and efficient, will eliminate a major obstacle to high research productivity on tomorrow's HPC platforms. In this work 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 middleware that is far more adapted and responsive to the application's performance and error characteristics.

We introduce a generic algorithmic based fault tolerant (ABFT) scheme that can apply on several widespread dense linear

factorizations. One of these factorizations, namely LU with partial pivoting, is significantly more challenging due to the pivoting . We theoretically prove that this scheme successfully applies to the three well known factorizations, Cholesky, LU and QR, and implement and evaluate an application of this generic ABFT scheme to the LU factorization. A significant contribution of this work is to maintain the left matrix of the factorization result (referred to as "the left factor" in the rest of the text), which was hitherto never achieved.

The rest of the paper is organized as follows: Section 2 presents prior work in the domain; Section 3 develops the general framework of ABFT for matrix factorizations; Section 4 presents the algorithms for checksum generation and recovery; Section 5 discusses the protection of the left factor; Section 6 proves that all data is protected by the checksum at any moment during the factorizations; Section 7 presents the experimental evaluation of our scheme; and Section 8 concludes the work.

2. RELATED WORK

The most well-known fault-tolerance technique for parallel applications is checkpoint-restart (C/R), which 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 ([5, 6]). 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 still widely used nowadays by high end systems [1]. To reduce the overhead of C/R, diskless checkpointing [26, 24] 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 [14]), it exhibits large overheads for applications modifying substantial memory regions between checkpoints [26], as is the case with dense linear factorizations.

Algorithm-based fault tolerance (ABFT) was first introduced to deal with silent error in systolic arrays [20]. Unlike other methods that treat the checksum and data separately, ABFT only encodes data once before computation. Matrix algorithms are designed to work on the encoded checksum along with matrix data, and the correctness is checked after the matrix operation completes. 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 linearly with the related matrix operation. ABFT and diskless checkpointing have been combined to apply to basic matrix operations like matrix-matrix multiplication [8, 10, 4, 9] and have been implemented on algorithms similar to those of ScaLAPACK [3], which is widely used for dense matrix operations on parallel distributed memory systems.

Recently, ABFT has been applied to the High Performance Linpack (HPL) [12] and to the Cholesky factorization [19]. 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. Also, LU with partial pivoting, when applied to the lower triangular L, potentially changes the checksum relation and renders basic checksum approaches useless.

The generic ABFT framework for matrix factorizations that we introduce in this work can be applied to Cholesky and HPL, but also to LU and QR. The right factor is protected by an ABFT checksum, while we evaluate the commonly accepted vertical checkpointing scheme, and propose two approaches for the left factor protection. One is based on a novel hybrid checkpointing scheme, that harnesses some insights from the algorithm structure to reduce the overhead on checkpointing. Another is a mathematical method to rebuild the left factor once the factorization completes, even if it has been damaged by failures. We investigate the consequences of failures hitting at critical phases of the algorithm and demonstrate that recovery is possible without suffering from error propagation and extensive synchronization.

3. ABFT FOR FAIL-STOP FAILURE IN MATRIX FACTORIZATIONS

In order to use ABFT for matrix factorization, an initial checksum is generated before the actual computation starts. Throughout the text we use matrix G to represent the generator matrix, and A for the original input matrix. Checksum for A is produced by

$$C = GA \text{ or } C = AG \tag{1}$$

When G is all-1 vector, checksum is simply the sum of all data items in a certain row or column. Referred to as "checksum relationship", Equation 1 is used after the computation for failure detection and recovery. This relationship has been shown separately for Cholesky[19], and HPL[12], which share the propriety of updating the trailing matrix with lower triangular matrix. In [25], ZU is used to represent LU (optionally with pairwise pivoting) and QR factorizations where Z is the left matrix (lower triangular or full) and U is an upper triangular matrix, and the factorization is 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. This scheme can in fact be applied to LU with partial pivoting, Cholesky and QR factorization:

THEOREM 3.1. *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

$$U = Z_n Z_{n-1} \ldots Z_1 A \tag{2}$$

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

$$\begin{aligned} U_c &= Z_n Z_{n-1} \dots Z_1 [A,\ AG] \\ &= [Z_n Z_{n-1} \dots Z_1 A,\quad Z_n Z_{n-1} \dots Z_1 AG] \\ &= [U,\quad UG] \end{aligned} \tag{3}$$

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

$$\begin{aligned} U_c^k &= Z_k Z_{k-1} \dots Z_1 [A,\ AG] \\ &= [Z_k Z_{k-1} \dots Z_1 A,\quad Z_k Z_{k-1} \dots Z_1 AG] \\ &= \left[U^k,\quad U^k G\right] \end{aligned} \tag{4}$$

Case 2: With partial pivoting:

$$\begin{aligned} U_c^k &= Z_k P_k Z_{k-1} P_{k-1} \dots Z_1 P_1 [A,\ AG] \\ &= [Z_k P_k Z_{k-1} P_{k-1} \dots Z_1 P_1 A, \\ &\qquad Z_k P_k Z_{k-1} P_{k-1} \dots Z_1 P_1 AG] \\ &= \left[U^k,\quad U^k G\right] \end{aligned} \tag{5}$$

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

Theorem 3.1 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 show that the checksum relationship also holds for block algorithms, both at the end of each iteration, and after the factorization is completed.

THEOREM 3.2. *For ZU factorization in block algorithm, checksum at the end of each iteration only covers data blocks that have already been factored and are still being factored in the trailing matrix.*

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

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \end{bmatrix} = \begin{bmatrix} Z_{11} & Z_{12} \\ Z_{21} & Z_{22} \end{bmatrix} \begin{bmatrix} U_{11} & U_{12} & U_{13} \\ 0 & U_{22} & U_{23} \end{bmatrix}, \tag{6}$$

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 Equation 6, 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 Equation 6, 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 □

4. CHECKSUM GENERATION AND PROTECTION

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 [12], we put checksum and data together in the process grid and design the checksum protection algorithm accordingly.

4.1 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 [11, 22]. 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. This layout helps with load balancing and reduces data communication frequency since in each step of the algorithm as many computing units can be engaged in computations and most of the time, each computing unit only works on its local data. Figure 1 is an example of $P = 2, Q = 3$ and a global matrix of 4×4 blocks. The same color represents the same process and numbering in A_{ij} indicates the location in the global matrix. Mapping between local blocks and their global locations can be found in [13].

With 2D block-cyclic data distribution, failure of a single process, usually a computing node which keeps several noncontiguous blocks of the matrix, causes holes distributed across the whole matrix. Figure 2 is an example of holes (red blocks) caused by the failure of process (1,0) in a 2×3 grid, and these holes are distributed into both checksum and matrix data.

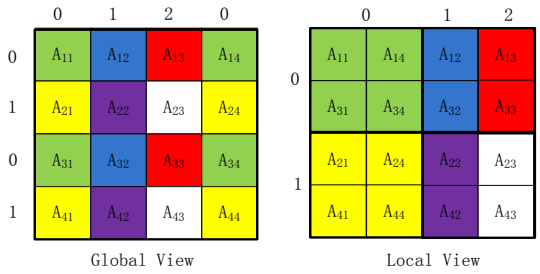

Figure 1: Example of 2D block-cyclic data distribution

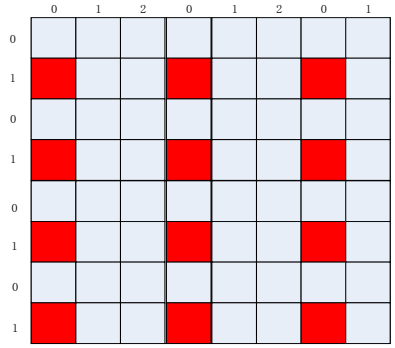

Figure 2: Holes in global matrix caused by failure

4.2 Checksum Generation

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:

$$C_k = \sum_{k=1}^{N} A_k \tag{7}$$

ScaLAPACK works with 2D block-cyclic distributed data, and a failure punches multiple holes in global matrix. With more than one hole per row/column, C_k in Equation 7 is not sufficient to recover all holes. A slightly more sophisticated checkpointing scheme is necessary.

THEOREM 4.1. *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 checkpointing 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 4.1, the size of the checksum in this row is K blocks.

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

$$C_k = \sum_{k=1}^{Q} A_k^k \tag{8}$$

Under Equation 8, we have the following corollary:

COROLLARY 4.2. *The i^{th} block of checksum is calculated using the i^{th} block of data of each process having at least i blocks.*

4.3 Protection for Checksum

Since ABFT checksum is considered as fragile as matrix data, from Theorem 4.1 and using the same N and Q, the total number of checksum blocks is $K = \lceil \frac{N}{Q} \rceil$. These checksum blocks are 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. If failure strikes any of these processes, part or all of checksum is lost. We propose two algorithms to protect the checksum, one compute intensive the other memory consuming.

4.3.1 *Duplication based checksum recovery*

In this algorithm, for both column- and row-wise checksums, the checksum is duplicated after creation. This duplication is treated in the same way as the original checksum under the condition that no checksum block and its duplicate maps to the same process.

Therefore the valid condition of the duplication based checksum recovery is that the checksum width K is not a multiple of process number in this column/row. If this condition is not respected, a simple solution consists of inserting an extra block row/column between the checksum blocks and their duplicate blocks. This extra block row/column shifts the mapping and ensures the condition.

Figure 3 is an example of 5×5 blocks matrix (on the top-left) with 2×3 process grid. Red blocks represent holes caused by the failure of process (1, 0). Green blocks on the right are row-wise checksum of width of 2 blocks with duplicates, and green blocks on the bottom are column-wise checksum of height of 2 blocks with shift column between duplicates.

Note that shift columns or rows are just padding, and they are not recovered. Gray blocks on the bottom-right is not marked since this part is not used at all. It is easy to verify that all holes in the checksum can be recovered from their duplicates. For instance, red blocks in the 6^{th} row are recovered by the corresponding (same column) green blocks in row 9.

4.3.2 *Multi-level checkpointing based checksum recovery*

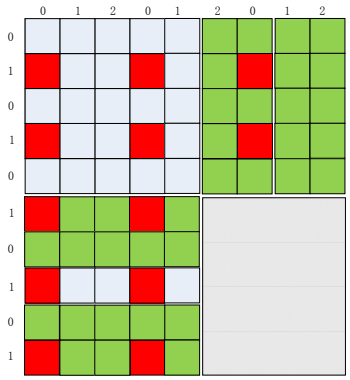

Figure 3: Checksum duplication for recovery

Another way of protecting checksum is from the observation that except for the mathematical meaning of checksum, there is no difference between checksum blocks and data blocks in the sense of storage. This means checksum blocks can be viewed as extra portion of data that needs to be covered, and therefore extra checksum can be generated for checksum blocks. These two levels of checksum are named *tier 1* for checksum from matrix data and *tier 2* for checksum from checksum.

Using the variables in section 4.3.1, let K be the number of tier 1 checksum column/row. Since ABFT factorization algorithm is more suitable for large number of processes for minimum checkpointing overhead impact, assume Q is large enough that each tier 1 checksum row/column maps to a different process. Tier 2 checksum is calculated as:

$$C'' = \sum_{i=1}^{K} C_i' \tag{9}$$

where the number of $'$ (apostrophe) means tier level. At the time of losing any C_i', it can be recovered by C'' and all C's that survive. To prevent C'' from being lost, duplication can be used to map C'' to a different node.

Compared to the checksum recovery method in Section 4.3.1, this method has the advantage of using less storage and less FLOPs from factorization for checksum protection. The extra computations that are required for generating the higher tier checksums could be overcome by the saving in factorization FLOPs on checksum. On the other hand, the storage savings might also become an asset when dealing with multiple-process failures.

5. PROTECTION OF THE LEFT FACTOR MATRIX

For the rest of this paper, we use the term "checksum" to refer to the ABFT checksum generated before factorization and put on the right of matrix, and "checkpointing" for the operation that protects the left matrix Z in ZU factorization. It was proven in Theorem 3.2 that checksum only covers the part of the matrix that has been and still is being factored, as shown in Figure 4 where the green checksum on the right of the matrix protects the green part of the matrix. For applications like iterative refinement and QR algorithm for eigenvalue problem, the whole factorization result (both Z and U) is needed, and therefore requires protection for the entire matrix. In LU, partial pivoting prevents the vertical checksum from being protected through ABFT, because data in the checksum should not be considered for pivoting. QR on the other hand, has no pivoting but still cannot use ABFT to cover Q as we prove below.

THEOREM 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

$$Q_{c11} Q_{c11}^T + Q_{c12} Q_{c12}^T = I. \tag{10}$$

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 Equation 10, $Q_{c11} Q_{c11}^T \neq I$. This means even though $A = Q_{c11} R_{c11}$, $Q_{c11} R_{c11}$ is not a QR factorization of A. □

Given that the ZU factorization cannot protect Z by applying ABFT in the same way as for U, separate efforts are needed. In ScaLAPACK where block algorithms are used for performance reasons, once a panel of Z in a ZU factorization is generated, it is stored into the lower triangular region of the original matrix and does not change until the end of the factorization. For example, in LU, vectors of L except the diagonal ones are stored, and in QR, vectors v that are used to generate the elementary reflectors are stored. These lower triangular parts are subject to no further change except partial pivoting in LU, and therefore only a unique vertical checkpointing is necessary to maintain each panel's safety, as discussed in [12]. We'll show that this idea, while mathematically trivial and being no different from diskless checkpointing with a frequency of once-per-panel factorization, suffers from a scalability issue . Then we will propose

an alternative ABFT algorithm to protect Z. LU factorization with partial pivoting is used for discussion but the result can be extended to QR.

Beside the two checkpointing based methods, we devise an ABFT recovery algorithm for L in LU factorization, which has no checkpointing involved.

5.1 ABFT for the Lower Triangular Matrix L

For a certain class of scientific applications in which the original data matrix after factorization is accessible with lower complexity, we devised an ABFT recovery algorithm for the lower triangular matrix. LU factorization is written as:

$$\begin{bmatrix} A_1 \\ A_2 \end{bmatrix} = PLU = P \begin{bmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{bmatrix} U$$

Due to hardware failure, data in L is lost when the failed process is recovered with holes in matrix. Pivoting matrix P moves rows in L that own lost data but can be tracked trivially by applying P onto a column vector of zeros except rows that contain lost data. Without loss of generality, suppose L_{11} has all the holes in L, and the corresponding rows of A are in $A_1{}'$. Both checksum and U can be recovered as stated previously, and since $A_1{}' = \begin{bmatrix} L_1 & 0 \end{bmatrix} U$, L_{11} can be recovered by solving Y in $A_1{}'Y = U$ using triangular solver.

The overhead of this algorithm includes: a) Assembly of the triangular system of equations; b) Solving the system of equations; c) Restoring L_1 in result matrix.

Let A be of size $M \times M$ on a $P \times Q$ grid, and the size of L_1 and $A_1{}'$ is $\lceil \frac{M}{P} \rceil \times M$. In strong scaling $\lceil \frac{M}{P} \rceil$ decreases as P increases while in weak scaling $\lceil \frac{M}{P} \rceil$ is a constant. Also Q processes can work in parallel and therefore the overhead of assembling the triangular system of equations for recovery and restoring is $O(\frac{M}{Q})$ communications.

The triangular solving step requires accessing a $\lceil \frac{M}{P} \rceil \times M$ part of matrix A, therefore this algorithm only applies to those applications whose complexity of "filling in" the original matrix is lower than the complexity of solving the linear system of equations using LU factorization [21, 23], hence, do not require storing the matrix in full form through disk I/O. In fact, only $\frac{1}{P}$ of the original data matrix is required. The computational overhead of triangular linear system solving is $O(\lceil \frac{M}{P} \rceil M^2)$. Similarly, in both weak and strong scaling, this overhead approaches $O(M^2)$ as P increases and remains lower than the $O(M^3)$ computational overhead of LU factorization and the $O(M^3)$ communication overhead of checkpointing for L.

Many applications spend more time doing factorization than generating the original data matrix, for those that cannot guarantee this requirement, a variation of the algorithm can be adopted: Using the same generator matrix G and factorization $PA = LU$

$$\begin{aligned} GA &= GP^{-1}LU \\ &= YU \end{aligned}$$

Y can be solved similarly and this only requires an initial checksum of A at the beginning. Then similar recovery process can be applied with some extra book-keeping scheme about pivoting when restoring L.

Figure 4: Checkpointing factorization

This method does not apply to QR since in ScaLAPACK, elementary reflectors, rather than Q itself, are stored in the lower triangular matrix, and re-constructing elementary reflectors from the recovered Q using the method proposed in this section requires higher complexity[18] than QR. For QR, other checkpointing methods are recommended.

5.2 Static vertical checkpointing for Z

In LU, for matrix A,

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

Panel factorization is:

$$\begin{pmatrix} A_{11} \\ A_{21} \end{pmatrix} = \begin{pmatrix} L_{11}U_{11} \\ L_{21}U_{11} \end{pmatrix} = \begin{pmatrix} L_{11} \\ L_{21} \end{pmatrix} U_{11} \quad (12)$$

In each step a panel of width nb is factored. To protect L_{11} and L_{21}, a separate checksum is put on the bottom of the matrix (referred to as "bottom checkpoint"), as shown in the yellow bottom rectangle of Figure 4. Once a panel is factored, it is not changed until the end of the computation with the exception of pivoting. When the generator matrix for checksum, G, is an all-one vector, checksum is not affected by row-exchanging pivoting. However this does not apply to 2D block cyclic distribution.

Figure 5 is a diagram of the components of LU factorization. The 2nd step (pivoting to the left) swaps two rows to the left of the current panel. Suppose rows i_1 and i_2 resides 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 4.2, 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

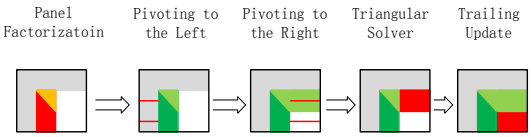

Figure 5: LU factorization diagram; Green: Just finished; Red & Orange: being processed; Gray: Finished in previous steps

relationship is expressed as

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

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

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

This requires a re-generation of checksum blocks k_{i_1} and k_{j_1} in order to maintain the checkpoint validity. Since pivoting to the left is carried out in every step of LU, this causes significant checksum maintenance overhead. To remedy this problem, pivoting to the left is accumulated and delayed to the very end of the LU factorization. Because the factored L is stored in the lower triangular part of the matrix without further usage, LU factorization with delayed left-pivoting still produces the correct result, as long as delayed pivoting is applied at the end of the computation.

Figure 6 shows an example of the recovery when process (1,0) in a 2×3 grid failed during LU factorization, right after pivoting-to-right is done during the 5th iteration.

Through a fault tolerant MPI infrastructures, like FT-MPI[15], failed process (1,0) is brought back to life and reintegrates the 2×3 grid. With the help of vertical static checkpointing and ABFT checksum, the lost data is recovered, and computation can proceed.

Suppose $T_{reduction}(x)$ is the time to do reduction on x processes, and suppose matrix size is $M \times N$ with block size NB. The overhead of vertical checkpointing is

$$\frac{M}{P} \times T_{reduction}(P) \times \frac{N}{NB}$$

5.3 Hybrid checkpointing for Z

The vertical checkpointing requires a set of reduction operations immediately after each panel factorization. Since panel factorization is on the critical path and has lower parallelism comparing to other components of the factorization (trailing matrix update for example), vertical checkpointing further worsens the situation by keeping processes in only one process column busy.

Since vertical checkpointing operation by nature is diskless checkpointing with checkpointing frequency once per iteration along the column direction, by increasing the checkpointing frequency and perform checkpointing along the row direction, less checkpointing with higher parallelism can be

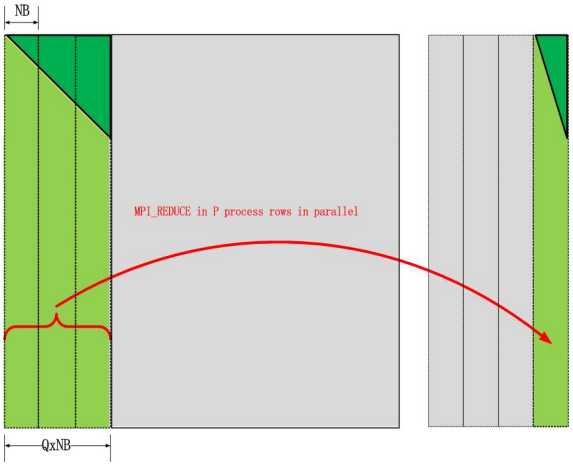

Figure 7: Hybrid checkpointing

achieved. Supposed ZU factorization on a $P \times Q$ grid. According to Corollary 4.2, starting from the first column on the left, every Q columns contributes to one column of checksum, which means once factorization is done for this Q column, the corresponding checksum column is fixed with the checksum of the upper triangular part of U. Utilizing this feature, we increase the checkpointing frequency to every Q iteration. During checkpointing, checksum is generated along row direction overwriting the corresponding checksum column, which makes all P process involved in the reduction operation in parallel. To comply with this checkpointing and prevent checkpointing result being mistakenly updated by trailing matrix update, the order of initial checksum on the right of the matrix is reversed so that during factorization, once a checkpointing is done, the generated checksum column is moved out of the range of trailing matrix update. Figure 7 is an illustration of the checkpoitning. $Q = 3$. Once the first 3 columns of data matrix completes factorization, they are checkpointed into the first checksum column reversely put on the right end of the checksum.

To utilize the hybrid checkpointing for recovery, at the beginning of each Q panel factorization, all processes make a copy of their local data into a temporary buffer (reusable throughout factorization). At this moment, their checksum column still follows the ABFT checksum. Therefore for any failure during the Q factorization, a failed process uses checksum to return at the state of the beginning of the Q iteration, and survived processes uses their local copies to return to the same state. Then the factorization for this Q columns can be restarted. Once rolled back, until the end of this Q iteration trailing matrix update is not applied outside this Q columns to avoid repetition. If failure occurs outside the Q iteration, then lost data can be recovered from the corresponding checksum column.

Suppose we use the same symbols as in the static vertical checkpointing model, the checkpointing performance for hy-

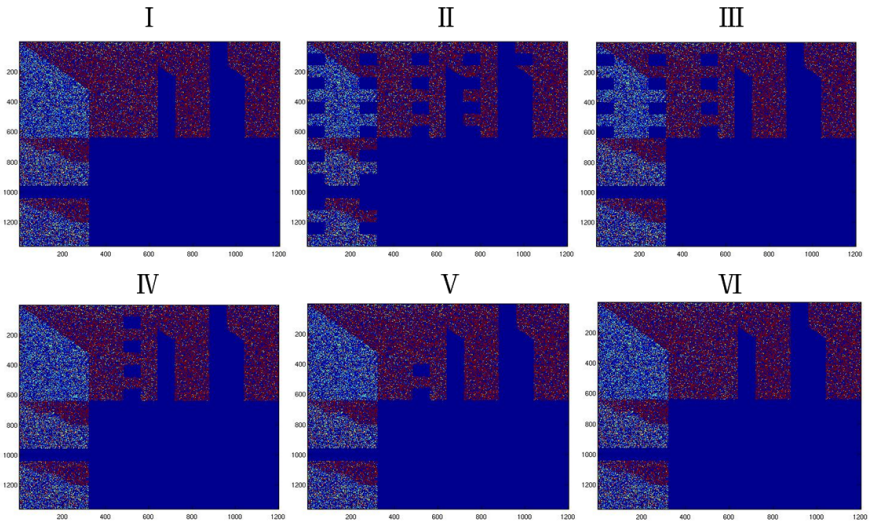

Figure 6: Recovery example (matrix size 640×640, grid size 2×3, failure of process (1,0), i=4, failure occurred after pivoting-to-right is done)
I: Right before failure II: After failure is fixed with holes (blue squares)
III: Checksum recovered IV-VI: data matrix recovered

brid checkpointing is

$$\frac{N}{Q \times NB} \times T_{reduction}(P)$$

6. RECOVERY FROM FAILURE WITH ERROR PROPAGATION

In a fail-stop failure, failure strikes at random moment during the life span of factorization operations or recovery process, and the interval between the exact time when the failed process stopped functioning and the time all survived processes are notified leaves opportunity for error propagation.

Most of the components of ZU factorization can be protected by ABFT. Matrix multiplication, which is used for trailing matrix updates and claims more than 95% of LU and QR execution time, has been shown in previous work[4] to be ABFT compatible. One feature that breaks this compatibility is pivoting in LU, especially when failure occurs between panel factorization and pivoting.

Figure 8 shows an example of such a case. Suppose the current panel contributes to the i^{th} column of checksum. When panel factorization finishes, the i^{th} 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 i_{th} 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, expending the recovery area to the whole column, as shown in red in Figure 8 including triangular solving area.

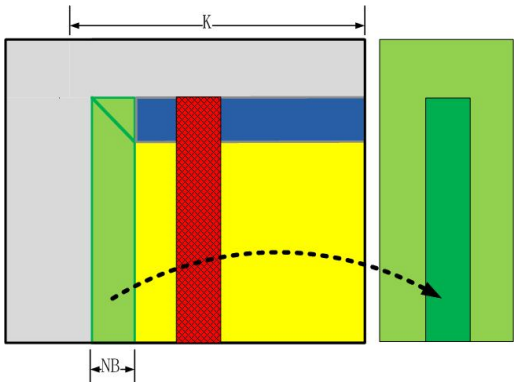

Figure 8: Ghost pivoting Issue
Gray: Result in previous steps
Light Green: Panel factorization result in current step
Deep Green: The checksum that protects the light green
Blue: TRSM zone Yellow: GEMM zone
Red: one of the columns affected by pivoting

To Recover from this case, in addition to matrix multiplication, the triangular solver is also required to be protected by ABFT.

THEOREM 6.1. *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 8 can be recovered when trailing matrix update is done.

7. EVALUATION

In this section, we evaluate the memory overhead of the proposed algorithm, as well as the computational efficiency on the *LU* factorization. The overhead comes from two different sources; extra flops generated by maintaining the right checksum up to date, and the extra communications involved in maintaining the L checkpoint. Should one consider the input matrix is easy to re-obtain, the latter can be ignored, and rely on the post-factorization recovery instead. To serve as a comparison base we use the non fault tolerant ScaLAPACK *LU*. For this purpose we used the NICS Kraken supercomputer hosted at the Oak Ridge National Laboratory, which features 112,896 2.6GHz AMD opteron cores with the Seastar interconnect. For small size experiments, we used a small cluster at the University of Tennessee, Knoxville named "dancer", which is an 8-node based on two quad Intel 2.27GHz Xeon cores per node, with a Infiniband 20G interconnect.

7.1 Storage Overhead

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

The storage of the checksum includes the row-wise and column-wise checksums and a small portion at the bottom-right corner. Different checksum protection algorithms requires different amounts of memory. In the following, we consider the duplication algorithm presented in the Section 4.3.1 for computing the upper memory bound.

For input matrix of size $M \times N$ on a $P \times Q$ process grid, the memory used for checksum (including duplicates) is

$$\frac{2MN}{P} + \frac{2MN}{Q} + \frac{4MN}{PQ}$$

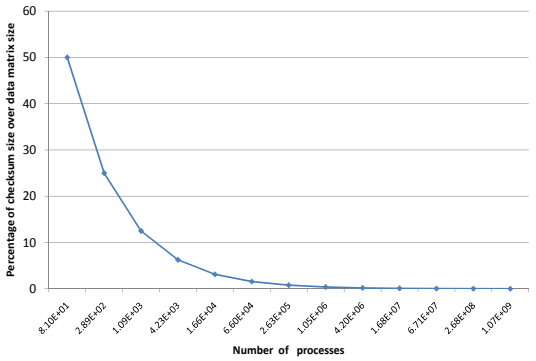

Figure 9: Checksum storage overhead

The ratio R_{mem} of checksum memory over the memory of the input matrix is

$$R_{mem} = \frac{2}{P} + \frac{2}{Q} + \frac{4}{PQ}$$

When $P = Q$ (as suggested for better performance in [13])

$$R_{mem} = 4 \times \left(\frac{1}{Q} + \frac{1}{Q^2}\right)$$

And therefore the storage overhead model is

$$Q = \frac{2 \times \left(1 + \sqrt{1 + R_{mem}}\right)}{R_{mem}} \tag{13}$$

Figure 9 shows an estimation according to Equation 13. For the current top ten machines on the Top500 list, the storage overhead is less than 1% of the data matrix.

7.2 Checksum and Checkpoint Costs

Figures 10 and 12 present the execution time in seconds (and the overhead in the zoomed-in graphs) for the algorithms presented above. In both cases, we used weak-scaling maintaining a ratio of 4000×4000 data per core while increasing the number of cores involved in the *LU* factorization. In the case of the overhead, we compute it based on the ScaLAPACK version of the *LU* factorization, with no fault tolerance features.

In Figure 10 the overhead of the checksum is under 10% starting with matrix sizes of 300×300, and as expected it decreases with the size of the factorization. As the size of the matrix increases, the number of cores involved in the factorization increases and the ratio of checksum memory to data memory decreases. As a consequence, the overhead of the trailing matrix update, which is the most time consuming operation when maintaining the checksum, decreases.

In Figure 12 we present the overhead of the fully protected ABFT factorization. In addition to the checksum presented previously, we added the checkpoint algorithms described in Section 5.2 and 5.3. As expected the static vertical checkpointing introduces a high overhead due to its sequential nature (checkpoint and panel are done in sequence blocking the progression of the updates). In contrast, the hybrid approach shows promising scalability capabilities.

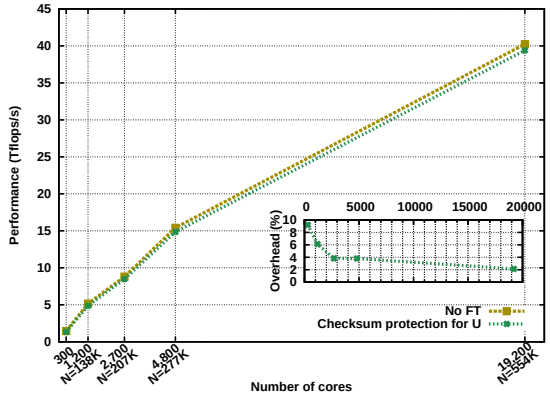

Figure 10: Weak Scalability of ABFT LU on the Kraken platform. Scaled from N=4000 for 1 core

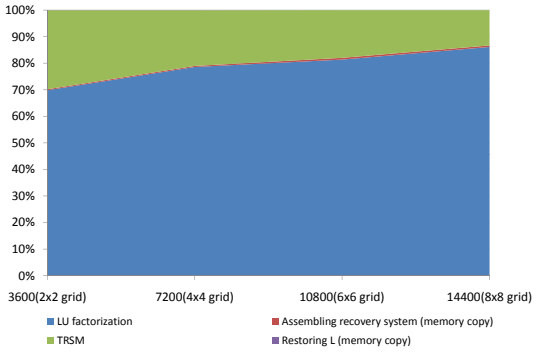

Figure 11: ABFT recovery overhead for L on Dancer

As L is touched only once during the computation, the approach of checkpointing the result of a panel synchronously can look sound, a-priori (when compared to system based checkpoint, where the entire dataset is checkpoint 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, where its importance is bound to grow when the number of computing resources increases. Its parallel efficiency is bound by P, while the overall computation enjoys a PxQ parallel efficiency. 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 PxQ parallel efficiency for the panel checkpointing. Because of this improved parallel efficiency, the hybrid checkpointing approach benefits from a competitive level of performance, that follows the same trend as the original non fault tolerant algorithm.

7.3 ABFT based post-factorization recovery for L

Figure 11 illustrates the time to recover the L part of the result matrix, using the ABFT based recovery algorithm proposed in section 5.1. The figure presents a percentage break down of the time required to perform the original factorization, to obtain U, and the time it takes to recompute L from the resultant U matrix and the original A matrix. The ratio

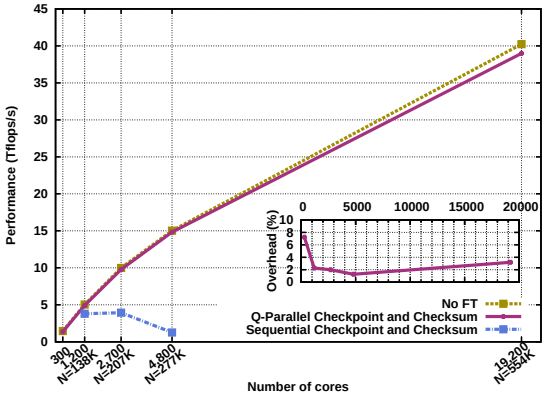

Figure 12: Weak Scalability of ABFT and L checkpointing on the Kraken platform. Scaled from N=4000 for 1 core

of time spent for the recovery of L over the time to compute U decreases when the number of processor increases, which is expected as the TRSM kernel, involved in this reconstruction, has a lower polynomial complexity that the LU factorization. Moreover, the recovery of L, which is shown to be a small part of the entire computation, is necessary only if and where failures occurred. However, this approach requires to access a small part of the input matrix whose location is unknown before the factorization. Therefore this method only applies to applications which can re-generate the input matrix with less overhead than the factorization.

8. CONCLUSION

In this paper, by assuming a failure model in which fail-stop failures can occur randomly during execution, a general scheme of ABFT algorithms for protecting matrix factorizations has been proposed. This framework 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 left and right factorization results are protected. For the left result several strategies have been proposed to ensure that it is always available. The experiments show the performance overhead of the checksum generation decreases with the increasing number of computing resources and the problem size. Moreover, the hybrid Q-parallel checkpointing overhead for the left factorization result maintains the scalability of the original factorization. As future work, multi-process failures and optimization of checkpointing strategies will be further investigated.

9. REFERENCES

[1] Fault tolerance for extreme-scale computing workshop report, 2009.

[2] http://www.top500.org/, 2011.

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

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

[5] A. Bouteiller, G. Bosilca, and J. Dongarra. Redesigning the message logging model for high performance. In *International Supercomputer Conference (ISC 2008), Dresden, Germany (June 2008)*. Citeseer.

[6] G. Burns, R. Daoud, and J. Vaigl. LAM: An open cluster environment for MPI. In *Proceedings of supercomputing symposium*, volume 94, pages 379–386, 1994.

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

[8] Z. Chen and J. Dongarra. Algorithm-based checkpoint-free fault tolerance for parallel matrix computations on volatile resources. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pages 10–pp. IEEE, 2006.

[9] Z. Chen and J. Dongarra. *Scalable techniques for fault tolerant high performance computing*. PhD thesis, University of Tennessee, Knoxville, TN, 2006.

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

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

[12] 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 25th ACM International Conference on Supercomputing (ICS 2011)*. ACM.

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

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

[15] G. Fagg and J. Dongarra. FT-MPI: Fault tolerant MPI, supporting dynamic applications in a dynamic world. *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 346–353, 2000.

[16] S. Ghemawat, H. Gobioff, and S. Leung. The Google file system. *ACM SIGOPS Operating Systems Review*, 37(5):29–43, 2003.

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

[18] G. Golub and C. Van Loan. *Matrix computations*. Johns Hopkins Univ Pr, 1996.

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

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

[21] K. Klimkowski and H. Ling. Performance evaluation of moment-method codes on an intel iPSC/860 hypercube computer. *Microwave and Optical Technology Letters*, 6(12):692–694, 1993.

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

[23] E. Lezar and D. Davidson. GPU-Accelerated Method of Moments by Example: Monostatic Scattering. *Antennas and Propagation Magazine, IEEE*, 52(6):120–135, 2010.

[24] C. Lu. *Scalable diskless checkpointing for large parallel systems*. PhD thesis, Citeseer, 2005.

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

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

[27] F. Streitz, J. Glosli, M. Patel, B. Chan, R. Yates, B. Supinski, J. Sexton, and J. Gunnels. Simulating solidification in metals at high pressure: The drive to petascale computing. In *Journal of Physics: Conference Series*, volume 46, page 254. IOP Publishing, 2006.