Fail-Stop Failure ABFT for Cholesky Decomposition

Doug Hakkarinen, *Student Member, IEEE*, Panruo Wu, *Student Member, IEEE*, and Zizhong Chen, *Senior Member, IEEE*

Abstract—Modeling and analysis of large scale scientific systems often use linear least squares regression, frequently employing Cholesky factorization to solve the resulting set of linear equations. With large matrices, this often will be performed in high performance clusters containing many processors. Assuming a constant failure rate per processor, the probability of a failure occurring during the execution increases linearly with additional processors. Fault tolerant methods attempt to reduce the expected execution time by allowing recovery from failure. This paper presents an analysis and implementation of a fault tolerant Cholesky factorization algorithm that does not require checkpointing for recovery from fail-stop failures. Rather, this algorithm uses redundant data added in an additional set of processes. This differs from previous works with algorithmic methods as it addresses fail-stop failures rather than fail-continue cases. The implementation and experimentation using ScaLAPACK demonstrates that this method has decreasing overhead in relation to overall runtime as the matrix size increases, and thus shows promise to reduce the expected runtime for Cholesky factorizations on very large matrices.

Index Terms—Extreme scale systems, Linear Algebra, Checkpoint Free, Algorithm Based Fault Tolerance.

1 Introduction

In scientific computing, the manipulation of large matrices is often used in the modeling and analysis of large scale systems. One such method widely used across many applications is linear regression. Linear regression tries to find the combination of M coefficients for M regressors that best fit a model based on data of R samples, with each sample consisting of a set of independent variables (x_i) and a resulting value (y_i, with the set of y_i called **y**). Then a matrix **X** is created, where each row contains the regressor values from a sample. Regressors can be constants (for an intercept), or functions of the independent variables. Most often linear regression is used to find the best estimate in the presence of a random unknown error. Usually this estimation is done as an overdetermined system, meaning there are more samples than coefficients. Thus, **X** is of dimension $R \times M$, with R larger than M. The problem takes the form of finding the best set of coefficients for the regressors (commonly β) such that $\mathbf{X}\beta$ has the minimal difference from **y**. In ordinary least squares regression, this problem is defined as finding β such that $\mathbf{X}^T\mathbf{X}\beta=\mathbf{X}^T\mathbf{y}$. The first step is to compute the matrix product of the of $\mathbf{X}^T\mathbf{X}$. The resulting $M \times M$ matrix, **A**, is symmetric positive semi-definite and often symmetric positive definite. Factoring this matrix into a triangular form for optimization is often desirable and produces an efficient way to find optimal coefficients through substitution. As such, factoring a symmetric positive definite is focused on through several techniques.

One efficient method for factoring a symmetric positive semi-definite matrix is through Cholesky factorization. In this factorization method, the result is either the upper triangular matrix **U**, such that $\mathbf{U}^T\mathbf{U}$ is **A**, or the lower triangular matrix **X**, such that $\mathbf{X}\mathbf{X}^T$ is **A**. In this work we focus only on creating the lower factorization **L**, although all methods that are used for **X** can be done for **U** as well since it is a symmetric algorithm. The Cholesky method is favored for many reasons, the foremost being it is numerically stable and does not require pivoting. The sequential Cholesky algorithm has several forms, such as the Outer Product, Right-Looking, and bordered.

In order to use Cholesky factorization for large data sets, methods need to be used that take into account the effects of performance based on cache usage. The libraries LAPACK and BLAS aid in optimizing the usage of cache through the blocking technique. The impact of this technique on the Cholesky method is that more than one iteration is effectively handled at once. Specifically, the number of iterations handled at once is the block size, MB. Blocking reduces sequential iterations (row by row) into fewer blocked iterations. The advantage of the blocked approach that many of the steps use the same data. Taking the actions across the entire row will clear the cache of this data, causing additional cache misses. Blocking prevents taking the action across the entire row, thus improving the performance.

If using a single node is not fast enough or the matrix size exceeds the memory on a single node,

Doug Hakkarinen is with the Department of Electrical Engineering and Computer Science, Colorado School of Mines, Golden CO. Email: dhakkari@mines.edu

Panruo Wu and Zizhong Chen are with the Department of Computer Science and Engineering,University of California, Riverside. Email: pwu011,chen@cs.ucr.edu

multiple process methods have been explored, particularly in the widely known software ScaLAPACK [9]. ScaLAPACK combines both the usage of LAPACK blocked techniques and support for multiple processes. Since there are more processes and there is an advantage to distributing the work across many processes, ScaLAPACK uses block-cyclic matrix distribution of data. In block-cyclic matrix distribution, the blocks that a particular processor contains come from many parts of the global matrix. As such, additional consideration must be taken when developing algorithms that act with ScaLAPACK and also interact with the data on the processes directly.

As the number of processors employed grows large, the probability of a failure occurring on at least one processor increases. In particular, a single processor failure is commonly modeled using the exponential distribution during periods of constant failure rates. Constant failure rates appear in the "Normal life" or "random failure" period in the commonly referenced Bathtub curve [20]. Constant failure rates apply for processors that are beyond their burn-in phase and not yet to their end of life phase. Under the assumption of an exponential failure rate of each processor, the failure rate of the overall system can be shown to grow linearly with the number of processors. Thus, for systems with increasing numbers of processors, failures become a larger problem. As the high performance community looks to exa-scale processing the failure rates must be addressed.

In order to counteract this increasing failure rate as systems grow, many techniques [14], [15] have been developed in order to provide fault tolerance. The most traditional technique is the checkpoint, or the routine saving of state. Some research has suggested that checkpointing approaches may face further difficulties as the number of processes expands [13].

Another promising approach is to take advantage of additional data held within the algorithm being executed to allow the recovery of a failed process. Approaches are known as algorithm based fault tolerance (ABFT) [19], [4], which use information at the end of execution of an algorithm to detect and recover failures and incorrect calculations. ABFT has traditionally been developed for fail-continue failures (failures which do not halt execution), with the goal of detecting and correcting errors at the end of computation. ABFT algorithms have received wide investigation, including development for specific hardware architectures [3], efficient codes for use in ABFT [2], [27], comparisons to other methods [26], [25], and analysis of potential performance under more advanced failure conditions [1]. The use of result checking in contrast to ABFT post-computation has also been studied [24]. Building on earlier work [17], we investigate the use of algorithmic methods to help recover fail-stop failures, or failures that halt execution. More importantly, our work demonstrates that ABFT can be used to recover failures during computation, rather than only after computation completes. Similar ABFT methods for fail-stop failures have been studied for matrix multiplication [7], [8], [5], for LU Decomposition [10], [11], [12], and Conjugate Gradient methods [22]. This method is not an endpoint, as potential improvements can be made through hot-replace strategies [28], [29], [30], but those methods depend on the algorithm to maintaining a checksum (such as is demonstrated in this work).

The differences between using data from within an algorithm versus checkpointing are numerous. The most specific difference is that the algorithm, for the most part, runs with little modification or stoppage such as is required to perform a checkpoint. If the amount of time to recover is approximately constant relative to the overall execution time, then this greatly improves the recovery time. The disadvantage of algorithmic recovery is that it only works on the algorithm in question, whereas checkpointing can often work regardless of what computation is being performed. Furthermore, depending on the algorithm and the recovery method, it may require more intense computation time to determine and restore the state of a failed process, which may or may not exist in checkpointing systems. In this work, we develop the use of algorithmic recovery for single fail-stop failures during a Cholesky factorization. While we only examine the case of a single failed process, adaptation for recovery of multiple failures is certainly possible [6].

2 Cholesky Factorization with Checksum Matrices

In this section, we explore the impact of checksum matrix inputs (as defined by [19]) on different Cholesky factorization algorithms. Checksum matricies of this type introduce redundant data as additional columns as linear combinations of the original matrix columns. Cholesky factorization requires a positive definite matrix input. As the checksum matrix is not linearly independent, it is no longer invertible, and not positive definite. Therefore, care must be taken to ensure the Cholesky factorization result to match the result of factorization of the original matrix. We explore the effects of checksum matrices on the Bordered algorithm,

$$A^f = \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} & \sum_1^n a_{1,j} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} & \sum_1^n a_{2,j} \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} & \sum_1^n a_{n,j} \\ \sum_1^n a_{i,1} & \sum_1^n a_{i,2} & \cdots & \sum_1^n a_{i,n} & \sum_1^n \sum_1^n a_{i,j} \end{pmatrix}$$

Fig. 1. Diagram of an unblocked checksum matrix. For Cholesky, the original matrix (without the checksum) must be symmetric ($a_{ij} = a_{ji}$) and positive definite.

```
1: for $i = 1 : M$ do
2:     $A(i, 1 : i-1) \leftarrow A(i, 1 : i-1)L(1 : i-1, 1 : i-1)^{-1}$ {where $L(1 : i-1, 1 : i-1)$ is the lower triangular part of $A(1 : i-1, 1 : i-1)$ including main diagonal. }
3:     $A(i, i) \leftarrow \sqrt{A(i, i)}$
4: end for
```

Fig. 2. The unblocked bordered Cholesky algorithm factorizes a $M \times M$ symmetric matrix into $A = LL^T$ in-place, where the lower triangular matrix L overwriting the lower triangular part of A.

```
1: for $i = 1 : M$ do
2:     $A(i : M, i) \leftarrow A(i : M, i)/\sqrt{A(i, i)}$
3:     if $i < M$ then
4:         $A(i+1 : M, i+1 : M) \leftarrow A(i+1 : M, i+1 : M) - A(i+1 : M, i)A(i+1 : M, i)^T$
5:     end if
6: end for
```

Fig. 3. The unblocked outer-product Cholesky algorithm factorizes a $M \times M$ symmetric matrix into $A = LL^T$ in-place, where the lower triangular matrix L overwriting the lower triangular part of A.

the Outer Product algorithm, and the Right-Looking algorithm.

We establish which Cholesky algorithm variants have the possibility of maintaining a checksum during computation. An example unprocessed, unblocked checksum matrix is shown in Figure 1. This checksum matrix is the starting point for all the unblocked algorithms being examined. For the algorithms that the checksum is maintainable, we then establish how to do so in a 2-D block cyclic version.

2.1 The Bordered Algorithm Does Not Maintain Checksums During Computation

The sequential, unblocked inline Bordered algorithm processes the matrix from upper left corner to the lower right corner. We focus on the generation of the **L** matrix (lower triangle), but symmetry holds for the upper triangular matrix. In each iteration of the Bordered algorithm, one additional row of entries is processed on and below the diagonal.

Figure 2 shows the algorithm. If this algorithm is performed on the checksum matrix in the Figure 1, the checksum row and checksum column (the last row and column) will not be touched until the last iteration. Updating the payload matrix without touching the checksums accordingly will therefore invalidate the checksums during the whole factorization. It can be shown that the checksum becomes valid only after n iterations, at which point the desired factorization is done. The checksums are not naturally maintained during the factorization.

2.2 The Outer Product Algorithm Maintains Checksums

The sequential, unblocked Cholesky Outer Product method can be separated into iterations for which M iterations as shown in Figure 3. This method is called the Outer Product method [16].

The first modification of the Outer Product algorithm is to initially form the checksum matrix using a summing reduce operation to the checksum process row and column (see Figure 1). If the original matrix is $n \times n$, the unblocked checksum matrix is $(n+1) \times (n+1)$. The second modification is that we skip the last iteration since the checksum matrix is not positive definite anymore. Now perform the algorithm described in Figure 3 on the augmented checksum matrix. We claim that every iteration i in Figure 3 preserves the checksums in some way. The algorithm stops after n iterations, and the column checksums are maintained after every iteration of the unblocked, inline Outer Product algorithm.

To see why the outer product algorithm maintains the column checksums at the end of every iteration, we need only to look at the operations in one iteration in Figure 3. For brevity we denote the sub matrix $A(i : n, i : n)$ by $A_{i:n,i:n}$. Note that now $M = n+1$. We use induction on i. Suppose before any iteration i, the first $i-1$ columns in the lower triangular L, have column checksums available at the checksum row:

$$A_{n+1,j} = \sum_{k=j}^{n} A_{k,j}, j = 1, \ldots, i-1 \qquad (1)$$

and the trailing matrix has column checksum available:

$$A_{n+1,j} = \sum_{k=i}^{n} A_{k,j}, j = i, \ldots, n \qquad (2)$$

These conditions hold when $i = 1$ by construction of the checksum matrix. We show that upon completion of iteration i, the conditions (1, 2) hold for $i+1$. In fact, after the line 2 in Fig 3, the condition (1) holds for $j = i$ since the vector and its checksum are scaled by the same factor. After the trailing matrix update in line 3

$$A_{i+1:n+1,i+1:n+1} \leftarrow A_{i+1:n+1,i+1:n+1} - A_{i+1:n+1,i}A^T_{i+1:n+1,i}$$

we would like to prove that condition (2) holds for $i+1$. In fact, the trailing matrix update is logically as the following equation if both line 2 and 4 in Fig 3 are not done in-place in memory:

$$B_{i:n+1,i:n+1} \leftarrow A_{i:n+1,i:n+1} - \frac{1}{\sqrt{A_{i,i}}}A_{i:n+1,i}\frac{1}{\sqrt{A_{i,i}}}A^T_{i:n+1,i}$$

Note that the checksum property is closed under matrix multiplication (column checksum matrix multiply by row checksum matrix) and addition. It is

$$\left(\begin{array}{c|cccc}
\frac{a_{1,1}}{\sqrt{a_{1,1}}} & 0 & \cdots & 0 & 0 \\ \hline
\frac{a_{2,1}}{\sqrt{a_{1,1}}} & a_{2,2} - \frac{a_{2,1}a_{2,1}}{a_{1,1}} & \cdots & a_{2,n} - \frac{a_{n,1}a_{2,1}}{a_{1,1}} & \sum_1^n a_{i,2} - \frac{a_{2,1}}{a_{1,1}} \sum_1^n a_{i,1} \\
\vdots & \ddots & \ddots & \ddots & \vdots \\
\frac{a_{n,1}}{\sqrt{a_{1,1}}} & a_{n,2} - \frac{a_{2,1}a_{n,1}}{a_{1,1}} & \cdots & a_{n,n} - \frac{a_{n,1}a_{n,1}}{a_{1,1}} & \sum_1^n a_{i,n} - \frac{a_{n,1}}{a_{1,1}} \sum_1^n a_{i,1} \\
\frac{\sum_1^n a_{i,1}}{\sqrt{a_{1,1}}} & \sum_1^n a_{i,2} - \frac{a_{2,1}}{a_{1,1}} \sum_1^n a_{i,1} & \cdots & \sum_1^n a_{i,n} - \frac{a_{n,1}}{a_{1,1}} \sum_1^n a_{i,1} & \sum_1^n \sum_1^n a_{i,j} - \frac{(\sum_1^n a_{i,1})^2}{a_{1,1}}
\end{array}\right)$$

Fig. 4. Unblocked checksum matrix after one iteration of the unblocked, inline outer product method. The first column contains elements of L. Columns 2 to $n+1$ are the updated but not yet processed elements. The top row are all zero. The column checksums are preserved.

easy to see that $B_{i:n+1,i:n+1}$ is the difference between two full checksum matrix therefore must also be a full checksum matrix. Further, we note that the first row $B_{i,i+1:n+1} = 0$, which means that the non-zero sub matrix $B_{i+1:n+1,i+1:n+1}$ is a full checksum matrix. We conclude by noting that this sub matrix is the new value of $A_{i+1:n+1,i+1:n+1}$. Hence, induction shows that condition (1, 2) holds after every iteration $i = 1, 2, \ldots, n$. Figure 4 shows the preserved column checksums at the end of the first iteration.

In summary, after each iteration, both the checksums of the first i columns and the south-east $(n-i) \times (n-i)$ trailing matrix maintain checksums naturally. The equivalent of a process failure in the unblocked algorithm would be the erasure of a particular element in the matrix. In order to recover any element in the matrix, the checksum along with the other elements could be used at any step. For example, element $A_{r,c}$ is reconstructed by

$$A_{r,c} \leftarrow A_{M+1,j} - (\sum_{k=1}^{r-1} A_{k,c} + \sum_{k=r+1}^{M})A_{k,c}$$

The sum of the other rows in its column recovers the checksum row.

2.3 Right-Looking Cholesky Algorithm Maintains Column Checksums

The Right-Looking inline, unblocked algorithm is similar to the inline, unblocked Outer Product algorithm. The key difference between the Right-Looking algorithm and the Outer Product algorithm is the data that is maintained. Computationally speaking, since the trailing matrix **B** is symmetric there is no need to maintain the entire submatrix. It is possible to compute half of the **B** matrix at the same time that the **L** matrix is being filled in. The Right-Looking algorithm utilizes the symmetry and does not maintain the unneeded half of the matrix.

The result of these optimizations in the Right-Looking algorithm is that the columns that contain **B** no longer add up to the checksum. The elements above the diagonal in the **B** matrix can no longer be used to recover a missing value. For a failure in column c, the missing elements are contained in row c. Figure 5 demonstrates how to recover without additional steps in each iteration.

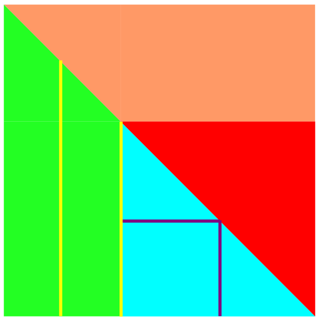


Fig. 5. Unblocked Right-Looking algorithm recovery. Yellow lines indicate what elements add to checksum for columns where the column for recovery is less than or equal to the iteration. Purple indicates what elements are needed for recovery when the column for recovery is greater than the iteration number.

The checksum row, along with the other elements of the matrix of the unblocked, inline Right-Looking method, contains sufficient information to recover any single element in the lower triangle of the matrix. We examine the block cyclic version of this algorithm in Section 3.2.

2.4 Summary of Checksum Maintenance

A summary of the checksums maintained by each algorithm is shown in Table 1. The Bordered algorithm is unable to maintain either row or column checksum. The Outer Product algorithm is able to 1) maintain both Row and Column Checksums for the as yet unprocessed matrix, 2) column checksums for the **L** matrix, and, 3) can maintain row redundancy for the **L** matrix (with modification). The Right-Looking algorithm maintains the column checksum, but requires an additional step for recovery in order to re-establish symmetry in the matrix. As both the Right-Looking algorithm and Outer Product algorithm can recover from a failure, we now look at how these algorithms behave in a 2-D block cyclic data distribution.

Bordered	Outer Product	Right-Looking
No	Yes	Yes^*

TABLE 1
Column checksums maintain or not?

3 2D Block Cyclic Cholesky Algorithms with Checksums

In blocked Cholesky algorithms, including block-cyclic algorithms, a block is processed each iteration.

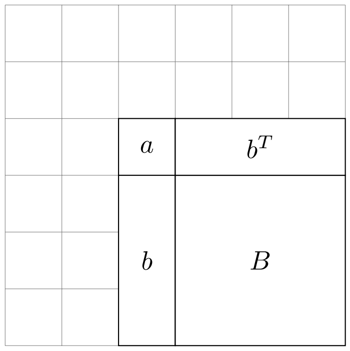


Fig. 6. Blocked Outer Product Cholesky factorization, in iteration 3

```
1: while $a$ is not empty do
2:     (POTF2) Solve $L$ from $a = LL^T$, $a \leftarrow L$
3:     (TRSM) Solve $X$ from $b = Xa^T$, $b \leftarrow X$
4:     (SYRK) $B \leftarrow B - bb^T$
5:     Move to the trailing matrix: $\begin{pmatrix} a & b^T \\ b & B \end{pmatrix} \leftarrow B$
6: end while
```

Fig. 7. Blocked Outer Produce Cholesky factorization algorithm.

For example, in the blocked Outer Product algorithm, several diagonal elements are handled on each iteration. When distributing a blocked algorithm across a process grid, each process generally manipulates only one block at a time. Blocks are usually sized to maximize the benefit of cache. In a block cyclic data distribution, each process has more than one block of data, with each block having a specific size (defined as MB rows and NB columns). Additionally, blocks from different parts of the matrix are on each process. These blocks are assigned from the global matrix to the processes in a cyclic fashion [21] to balance load. Basically the matrix is divided into many blocks, and the processes are treated as a $p_1 \times p_2$ 2D process grid. The blocks are assigned to the processes in the process grid in a round-robin manner. More specifically, block (i, j) got assigned to processer $(i \mod p_1, j \mod p_2)$ assuming all indices start from 0.

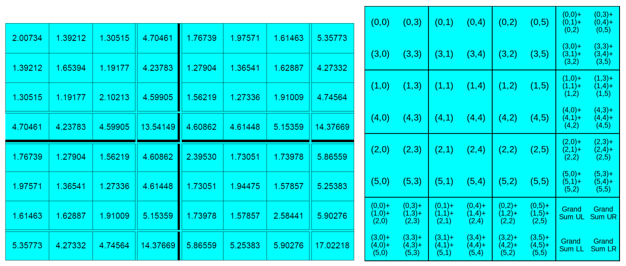
For simplicity, we cover the case where the processor grid, without the checksum row and column, is square $P \times P$. Additionally, we assume the blocking is square (i.e., $MB \times MB$). As the input matrix (without checksum) is $M \times M$, and there are $P \times P$ processors (without checksum processor), we assume each processor holds an equal number of elements,$\frac{M^2}{P^2}$, which we assume is a square number of elements with a side length ML. Finally, for ease of implementation, we assume each processor's local matrix to be evenly divisible into a square number of blocks on each processor (i.e., $(ML)^2 \mod (MB)^2 = 0$).

While these assumptions greatly ease the implementation, this method can still be used for Cholesky factorizations where these conditions do not hold. Block cyclic checksum algorithms function by reserving an extra row and/or column of processes to hold a checksum of the values, which is called the checksum process row and checksum process column, respectively. We call a matrix with a checksum process row as a row process checksum matrix. Note that in such a matrix, each process column has respective blocks that sum to the value in the checksum process row. We call a matrix with both checksum process row and column a full process checksum matrix. In the checksum processes, a sum of the respective values of the other blocks in the same row or column, respectively, are kept. To do so, each checksum process holds the same number of elements as any other process, namely $MXLLDA^2$. The total number of additional processes to add a checksum row and column is $2P + 1$. The full process checksum matrix therefore is $(M + MXLLDA) \times (M + MXLLDA)$.

The execution of the block cyclic Cholesky with checksum routine proceeds just as a normal block-cyclic Cholesky routine with the following exception: every P iterations, when a checksum block would be the next block to be used as the matrix diagonal, the algorithm jumps to the next iteration. As such, no additional iterations are performed in the running of the Cholesky routine. However, each iteration may take longer due to the additional communication and time due to the existence of the checksum row and column. Other than this, the checksum block cyclic Cholesky algorithms function similarly to a standard block cyclic Cholesky algorithm, such as is found in ScaLAPACK's PDPOTRF.

3.1 The 2D Block Cyclic Outer Product Algorithm

For a distributed 2-D block cyclic distribution, the blocked version is effectively the same. The major difference is that care must be taken in order to ensure that the data needed at any step is sent to the correct



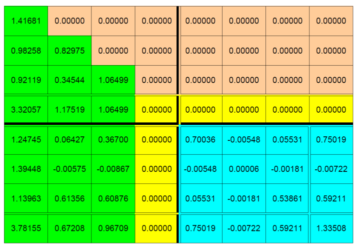
2.00734	1.39212	1.30515	4.70461	1.76739	1.97571	1.61463	5.35773
1.39212	1.65394	1.19177	4.23783	1.27904	1.36541	1.62887	4.27332
1.30515	1.19177	2.10213	4.59905	1.56219	1.27336	1.91009	4.74564
4.70461	4.23783	4.59905	13.54149	4.60862	4.61448	5.15359	14.37669
1.76739	1.27904	1.56219	4.60862	2.39530	1.73051	1.73978	5.86559
1.97571	1.36541	1.27336	4.61448	1.73051	1.94475	1.57857	5.25383
1.61463	1.62887	1.91009	5.15359	1.73978	1.57857	2.58441	5.90276
5.35773	4.27332	4.74564	14.37669	5.86559	5.25383	5.90276	17.02218

(a) Global view

(0,0)	(0,3)	(0,1)	(0,4)	(0,2)	(0,5)	(0,0)+ (0,1)+ (0,2)	(0,3)+ (0,4)+ (0,5)
(3,0)	(3,3)	(3,1)	(3,4)	(3,2)	(3,5)	(3,0)+ (3,1)+ (3,2)	(3,3)+ (3,4)+ (3,5)
(1,0)	(1,3)	(1,1)	(1,4)	(1,2)	(1,5)	(1,0)+ (1,1)+ (1,2)	(1,3)+ (1,4)+ (1,5)
(4,0)	(4,3)	(4,1)	(4,4)	(4,2)	(4,5)	(4,0)+ (4,1)+ (4,2)	(4,3)+ (4,4)+ (4,5)
(2,0)	(2,3)	(2,1)	(2,4)	(2,2)	(2,5)	(2,0)+ (2,1)+ (2,2)	(2,3)+ (2,4)+ (2,5)
(5,0)	(5,3)	(5,1)	(5,4)	(5,2)	(5,5)	(5,0)+ (5,1)+ (5,2)	(5,3)+ (5,4)+ (5,5)
(0,0)+ (1,0)+ (2,0)	(0,3)+ (1,3)+ (2,3)	(0,1)+ (1,1)+ (2,1)	(0,4)+ (1,4)+ (2,4)	(0,2)+ (1,2)+ (2,2)	(0,5)+ (1,5)+ (2,5)	Grand Sum UL	Grand Sum UR
(3,0)+ (4,0)+ (5,0)	(3,3)+ (4,3)+ (5,3)	(3,1)+ (4,1)+ (5,1)	(3,4)+ (4,4)+ (5,4)	(3,2)+ (4,2)+ (5,2)	(3,5)+ (4,5)+ (5,5)	Grand Sum LL	Grand Sum LR

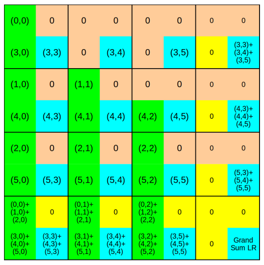
(b) Local view

Fig. 8. Global and Local views of a 6x6 input matrix (8x8 checksum matrix), symmetric blocked checksum matrix to be used in demonstration of the 2-D block cyclic Outer Product and Right-Looking algorithms. The size of each block is 1 element, the number of blocks per process is 4 ($MB = 2$).



1.41681	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000
0.98258	0.82975	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000
0.92119	0.34544	1.06499	0.00000	0.00000	0.00000	0.00000	0.00000
3.32057	1.17519	1.06499	0.00000	0.00000	0.00000	0.00000	0.00000
1.24745	0.06427	0.36700	0.00000	0.70036	-0.00548	0.05531	0.75019
1.39448	-0.00575	-0.00867	0.00000	-0.00548	0.00006	-0.00181	-0.00722
1.13963	0.61356	0.60876	0.00000	0.05531	-0.00181	0.53861	0.59211
3.78155	0.67208	0.96709	0.00000	0.75019	-0.00722	0.59211	1.33508

(a) Global view



(0,0)	0	0	0	0	0	0	0
(3,0)	(3,3)	0	(3,4)	0	(3,5)	0	(3,3)+ (3,4)+ (3,5)
(1,0)	0	(1,1)	0	0	0	0	0
(4,0)	(4,3)	(4,1)	(4,4)	(4,2)	(4,5)	0	(4,3)+ (4,4)+ (4,5)
(2,0)	0	(2,1)	0	(2,2)	0	0	0
(5,0)	(5,3)	(5,1)	(5,4)	(5,2)	(5,5)	0	(5,3)+ (5,4)+ (5,5)
(0,0)+ (1,0)+ (2,0)	0	(0,1)+ (1,1)+ (2,1)	0	(0,2)+ (1,2)+ (2,2)	0	0	0
(3,0)+ (4,0)+ (5,0)	(3,3)+ (4,3)+ (5,3)	(3,1)+ (4,1)+ (5,1)	(3,4)+ (4,4)+ (5,4)	(3,2)+ (4,2)+ (5,2)	(3,5)+ (4,5)+ (5,5)	0	Grand Sum LR

(b) Local view

Fig. 9. Global and local views after three iterations of the Outer Product algorithm. Blue is the unprocessed matrix. Green are parts of **L**. Orange are parts that should be treated as 0s for ease of using column checksums. The next step would have a checksum block as the diagonal block, so it is skipped. The now zero checksum blocks are noted in yellow.

processes. The snapshot of iteration $j = 3$ is shown in Figure 6 and the blocked right looking Choleskys algorithm is shown in Figure 7.

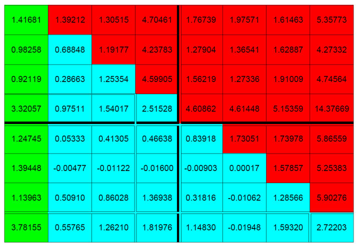
For clarity, we present a numerical example of the blocked version. The initial checksum matrix is shown in Figure 8. Figure 9 illustrates the state after the first three steps. From this view, it looks similar to the unblocked method as the block sizes are 1; however, note that the checksums are maintained within each section of the matrix. After step three, the next step would have a checksum block as the diagonal block and is skipped. As the data of the original matrix is in the non-checksum blocks, the algorithm produces the same result as the non-checksummed algorithm.

3.2 Handling Failure in a 2D Block Cyclic Right-Looking Algorithm

We next look at the 2-D block cyclic Right-Looking algorithm. This algorithm is similar to the corresponding Outer Product algorithm. Only the lower triangle of **B** matrix is maintained.

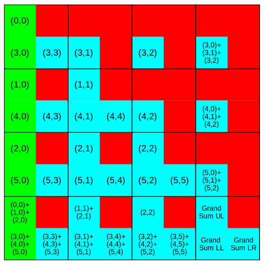
After any given iteration of the inline algorithm, the global matrix contains three different parts: 1) the partial result matrix **L**, 2) the unfactorized elements **B**, and 3) data that is no longer needed. Figure 10 shows the breakdown of a matrix after one iteration. The **L** portion of the matrix holds the partial result. The **B** matrix contains the yet to be factorized elements.

For blocks that are in the **L** section of the matrix, the checksum is the sum of the entire column of corresponding local entries of **L**. For blocs in the **B** section of the matrix, the checksum is the sum of the elements in the column of the **B** matrix, including those that would be past the diagonal if they were maintained. To use these checksum blocks, symmetric elements must be used to recover elements in **B** similar to the unblocked Right-Looking algorithm. These are transposed from the equivalent row of the failure column into the failure column, thus reconstructing the column checksum relationship.



1.41681	1.39212	1.30515	4.70461	1.76739	1.97571	1.61463	5.35773
0.98258	0.68848	1.19177	4.23783	1.27904	1.36541	1.62887	4.27332
0.92119	0.28663	1.25354	4.59905	1.56219	1.27336	1.91009	4.74564
3.32057	0.97511	1.54017	2.51528	4.60862	4.61448	5.15359	14.37669
1.24745	0.05333	0.41305	0.46638	0.83918	1.73051	1.73978	5.86559
1.39448	-0.00477	-0.01122	-0.01600	-0.00903	0.00017	1.57857	5.25383
1.13963	0.50910	0.86028	1.36938	0.31816	-0.01062	1.28566	5.90276
3.78155	0.55765	1.26210	1.81976	1.14830	-0.01948	1.59320	2.72203

(a) Global view



(0,0)							
(3,0)	(3,3)	(3,1)		(3,2)		(3,0)+ (3,1)+ (3,2)	
(1,0)		(1,1)					
(4,0)	(4,3)	(4,1)	(4,4)	(4,2)		(4,0)+ (4,1)+ (4,2)	
(2,0)		(2,1)		(2,2)			
(5,0)	(5,3)	(5,1)	(5,4)	(5,2)	(5,5)	(5,0)+ (5,1)+ (5,2)	
(0,0)+ (1,0)+ (2,0)		(1,1)+ (2,1)		(2,2)		Grand Sum UL	
(3,0)+ (4,0)+ (5,0)	(3,3)+ (4,3)+ (5,3)	(3,1)+ (4,1)+ (5,1)	(3,4)+ (4,4)+ (5,4)	(3,2)+ (4,2)+ (5,2)	(3,5)+ (4,5)+ (5,5)	Grand Sum LL	Grand Sum LR

(b) Local view

Fig. 10. Global and Local views of Matrix from Figure 8 after one iteration of the Right-Looking algorithm. The block size is 1 element. Green shows parts of **L**. Blue shows unprocessed data in **B**. Red shows parts of **B** that are not maintained.

Recovery Routine

```
1: Restore Matrix Symmetry for Column $f_c$
2: $A(f_r, f_c, *) \leftarrow \mathbf{0}$
3: if Column = $f_c$ then
4:     for all Blocks $b \in LocalMatrix$ do
5:         if $f_r$ = $checksum_r$ then
6:             $Reduce_{col}$ $\sum_{i=1}^{P} A(i, f_c)$ to $A(f_r, f_c)$
7:         else
8:             $Reduce_{col}$ $A(P + 1, f_c) + \sum_{i=1}^{P} -A(i, f_c)$ to $A(f_r, f_c)$
9:         end if
10:    end for
11: end if
```

Fig. 11. The steps to recover from a failure in the 2-D block cyclic Right-Looking parallel algorithm, where A(r,c,b) is the local matrix block for row r, column c, block b, the process failure row is f_r and the process failure column is f_c.

In a block cyclic distribution, one process may contain data from all three sections. Thus, upon a failure, each element must be examined individually to determine how it should be recovered and restore symmetry as necessary. Once in this form, the block cyclic equivalent of the unblocked recovery algorithm is used (see Figure 11).

When a failure occurs, the algorithm must restore the values of the failed data to the failed process. The following preconditions must be met: 1) the row and column of the failed process must be known, 2) no additional processes may fail during recovery, 3) the failed step of the iteration must be known and detected before the next step begins, and 4) the iteration must be known. Section 4.3 examines specific recovery procedures based on the step in an iteration when a process fails.

4 Overhead Scalability Analysis

To examine the overhead scalability of ABFT Cholesky, we first examine the runtime of the (non-fault tolerant) blocked Right-Looking Cholesky factorization on a $P \times P$ process grid. We then examine the scalability of the overhead and for recovery in a square $((P+1) \times (P+1))$ process grid using a full process checksum matrix.

4.1 Blocked Right-Looking Cholesky Runtime

The runtime of inline blocked Outer Product Cholesky routines is dominated by the iterative matrix updates. We examine the contributions to runtime of these iterations to characterize the general runtime of the inline block-cyclic Right-Looking Cholesky factorization in the lower triangular case. Each iteration consists of a local factorization of the next diagonal block, a panel (elements below the diagonal block) update, and an update of the remaining matrix. The size of the panel and the remaining matrix depends on the iteration, going from largest to smallest. We first examine the size of the panel and remaining matrix. We then examine the runtime of each step of an iteration. We then summarize the total runtime.

4.1.1 Size Of Data Affected in An Iteration

In general, there are a total of $\lceil \frac{M}{MB} \rceil$ iterations. For simplicity of illustration, we assume that the M is evenly divisible by MB, though the analysis does not significantly change in the case that there is a remainder. In each iteration, the panel to be updated consists of all rows below the diagonal in the columns of the diagonal block. In the first iteration, the panel would have dimensions $MB \times (M - MB)$. In the i^{th} iteration the panel size is $MB \times (M - (i \cdot MB))$. The total number of elements updated during the panel update over all iterations is:

$$\sum_{i=1}^{\frac{M}{MB}} MB \cdot (M - i \cdot MB) = \frac{M^2 + M \cdot MB}{2} \tag{3}$$

We analyze the average size of an iteration. The average size of the panel on a given iteration is the total divided by the number of iterations:

$$\frac{\frac{M^2+M\cdot MB}{2}}{\frac{M}{MB}} = \frac{M \cdot MB + MB^2}{2} = O(M \cdot MB) \tag{4}$$

As the remaining matrix update is the scaled subtraction of the the outer product of the panel with its transpose from the existing elements, the number of elements is the square of the number of row elements of the panel. Therefore, the average size of the trailing matrix is $O(M^2)$.

4.1.2 Diagonal Block Factorization

The diagonal block is held on a single process. Therefore any serial unblocked factorization can be used. The block is a matrix of size $(MB \times MB)$. Several serial algorithms are capable of performing this factorization in $O(MB^3 \cdot \gamma)$ where γ is the time to perform an arithmetic operation. We consider the common case that $MB << M$. The block factorization is not the rate limiting step so we do not explore this step further.

4.1.3 Panel Update

During the panel update, the factorized diagonal block is sent to the processes holding the panel elements. Each of the elements in the panel must be updated. The initial distribution of the factorized diagonal block requires a broadcast down the process column. The optimal algorithm for this broadcast depends on the system's latency (α), bandwidth (β), number of nodes in the process column (P), and the number of elements to broadcast (i.e., MB^2) (see Table 3). As there are several methods to broadcast, we generically denote the runtime for broadcast as $BCast(\eta, P_b)$ where η is the number of elements to be broadcasted, and P_b is the number of processes involved in the broadcast. As there is one block that is broadcasted, the broadcast step requires $BCast(MB^2, P)$. The runtime of this step of each iteration is $O(BCast(MB^2, P) + \frac{MB^2 \cdot M}{P} \cdot \gamma)$.

4.1.4 Remaining Matrix Update

The remaining matrix update requires the outer product of the panel to be computed on the processes holding the remaining matrix elements. Each block in the remaining matrix requires at maximum two different blocks from the panel (the two blocks used to perform the outer product). One of these blocks is the corresponding block in the same row in the panel. Therefore a row broadcast of the panel results in this block being received by the appropriate processes. Again, the optimal broadcast method depends on several factors (see Table 3). This sub-step therefore requires $BCast(\frac{M \cdot MB}{P}, P)$.

The elements of the transpose of the panel are distributed to the columns that correspond to the equivalent rows of the panel. On a square process grid, this panel transpose can be performed pairwise between processes holding the panel and those holding the first row of the remaining matrix blocks. $M \cdot MB$ elements must be transmitted, and are distributed over P process rows, and must be sent to P process columns. We note the time of one send as $Snd(\eta)$ where η is the amount of data to send. In this case, the runtime would be $Snd(\frac{M \cdot MB}{P})$.

After the exchange, the process row with the transposed panel then broadcasts the transposed panel down the process column to complete the data distribution. This sub-step requires $BCast(\frac{MB \cdot M}{P}, P)$. Once the data is distributed, there are $O(M^2)$ elements to update using $P\times P$ processes. As each element update requires MB multiplications, MB additions and one subtraction, the element update is $O(\frac{MB\cdot M^2}{P^2}\cdot\gamma)$ each iteration. The update step therefore has a runtime of:

$$O(BCast(\frac{M\cdot MB}{P},P)+Snd(\frac{M\cdot MB}{P})+\frac{M^2\cdot MB\gamma}{P^2}) \tag{5}$$

4.1.5 Runtime Over Iterations

The overall runtime can be approximated as the average time of each step multiplied by the number of iterations. The maximum in this case is determined either by the time to broadcast, or the arithmetic update of any given iteration. In both cases, the remaining matrix update dominates the iteration. There are $\frac{M}{MB}$ iterations. The overall runtime for the distributed block-cyclic Cholesky factorization through this variant is therefore:

$$O(\frac{M^3\gamma}{P^2}+\frac{M\cdot BCast(\frac{MB\cdot M}{P},P)+BCast(\frac{MB\cdot M}{P},P)}{MB}) \tag{6}$$

4.2 Overhead of ABFT Cholesky on a Square Grid with Full Processor Checksum Matrix

The overhead of ABFT Cholesky consists of the overhead for setting up the initial checksum process row and checksum process column (i.e., full checksum matrix), the overhead due to increased matrix size, and the overhead due to algorithm modification. The overhead of the algorithm modification is negligible as it consists only of skipping iterations where a checksum block would be the diagonal block. This check would only occur once per iteration. The other two overheads, however, deserve closer analysis. In this section, we assume that the original matrix is distributed over a $P\times P$ process grid, with the full checksum matrix process grid being $(P+1)\times(P+1)$.

4.2.1 Overhead of Checksum Matrix Setup

The checksum process column contains a block-wise summation of corresponding blocks over the blocks of each row. Therefore, each block needs to be transmitted and summed into the checksum process column. This operation is the equivalent of an summation MPI Reduce call. Depending on the characteristics of the matrix and system, the best way to perform this reduction varies. As such, we note the time for reduction as $Reduce(\eta, P_r)$, where η is the size of the data to be reduced by process, and P_r is the number of processes involved. We assume that the destination does not significantly impact the runtime.

Each process, including checksum processes, holds $\frac{M^2}{P^2}$ elements. Each process row has P processes, plus the checksum column process, making $P+1$ processes. Therefore each reduction occurs over $P+1$ processes. As such, the reduce across rows takes $Reduce(\frac{M^2}{P^2},P+1)$. There are a total of P reductions needed, but these can be done in parallel.

After construction the checksum process column, we construct the checksum process row. This construction is also the summation of corresponding blocks, but this time over the process columns into the checksum process row. There are P process rows, plus the checksum row, making $P+1$ rows. Again, each process holds $\frac{M^2}{P^2}$ elements. The summation into the checksum process row requires $Reduce(\frac{M^2}{P^2},P+1)$. Once the second reduction step is complete, the checksum process row and column setup is complete. Therefore, the overall runtime for the checksum setup step is $O(Reduce(\frac{M^2}{P^2},P+1))$.

4.2.2 Overhead of Increased Matrix Size

To analyze the overhead of the increased matrix size, we must analyze both any increase in program wall-clock runtime and the overhead that results from using an increased number of processes. We first look at the sources of increased wall-clock runtime.

The checksum matrix increases the size of the matrix in each dimension by one process. The modified algorithm skips any iteration where a checksum block is the diagonal block. Therefore, the number of iterations remains $\lceil\frac{M}{MB}\rceil$. Within each iteration the amount of data has increased.In the diagonal block factorization step, no additional data has been introduced so the runtime of this step remains unchanged. During the panel update step, the diagonal block must be communicated to $(P+1)$ processes instead of P processes and perform additional computation on the blocks held in the checksum process. For the computation, as an additional process is available, the amount of wall-clock time remains unchanged. The increase in the broadcast results in a runtime of $BCast(MB^2,P+1)+\frac{MB\cdot M}{P}$, or overhead of:

$$O(BCast(MB^2,P+1)-BCast(MB^2,P)) \tag{7}$$

During the remaining matrix update, the broadcast across process rows and the broadcast across process columns both broadcast over one additional process. Therefore these steps both now take $BCast(\frac{MB\cdot M}{P},P{+}1)$. The panel transpose must also be communicated from and to the respective checksum row process of the panel and the checksum column process. In a $P\times P$ grid, there is a one to one correspondence of these processes, and therefore the time is only to send the elements. This is also true for the original processes, and as such this step presents no additional overhead. The number of matrix elements that must be computed during the panel update also increases. However, the number of elements increases proportionally to the number of checksum processes, and reside on those checksum processes. Therefore, the runtime increase for this step consists of the increase in the additional time for broadcast, or $O(BCast(\frac{MB \cdot M}{P}, P+1) - BCast(\frac{MB \cdot M}{P}, P))$.

Looking at the iteration steps, the remaining matrix update dominates the runtime as in the non-fault tolerant case. As the number of iterations has not changed, the wall-clock overhead for all the steps is:

$$O(\frac{M(BCast(\frac{MB \cdot M}{P}, P+1) - BCast(\frac{MB \cdot M}{P}, P))}{MB}) \quad (8)$$

The remainder of the overhead can be classified as the time to use the additional processors that could be used for some other job in a cluster. The number of processors has increased by one processor row and one processor column. Therefore, the number of processors has increased from P^2 to $(P+1)^2$. To determine the overall overhead, we therefore scale the wall-clock overhead by $\frac{(P+1)^2}{P^2}$.

Multiplying this overhead factor by the dominant overhead in Equation 8 yields the approximate overhead for the fault tolerant Cholesky routine due to the increased matrix size. Thus the overall overhead from the increased matrix size is:

$$O(\frac{(P+1)^2}{P^2} \cdot \frac{M}{MB} \cdot BCast(\frac{MB \cdot M}{P}, P+1)) \quad (9)$$

4.3 Recovery on a P x P Processor Grid

When a process fails, the contents of its memory related to the matrix need to be reconstructed when it is restarted, or on a replacement process. We refer to the process taking the place of the failed process as p_f. We also assume a recovery routine is initiated on every process that indicates which process has failed. With knowledge of which process failed, the other processes place p_f in the equivalent point in their communications, and that p_f assumes the role in terms of memory and processing tasks that would have been performed by the failed process. Therefore, the objective of recovery is to restore the memory of p_f to a point where the factorization can continue.

Immediately after p_f comes online, an arbitrary other process, say process 0, should transmit information concerning which iteration was being performed, and furthermore during what step of the iteration the failure occurred. Additional information such as the characteristics and decomposition of the matrix should also be sent. Based on the iteration and the decomposition, p_f determines what block positions in the matrix it should contain and determine whether those blocks contain the result matrix **L**, are no longer needed (i.e., are above the diagonal in a row that has already been passed by the iterations), the diagonal block $A_{i,i}$, panel blocks b_i, or remaining matrix blocks B_i. The other processes should likewise identify that type of data contained in p_f's blocks.

```
1: if This Node Failed then
2:    $LocalMatrix \leftarrow 0$
3: else
4:    for all Blocks ∈ Local Matrix do
5:       if Block Above Diag AND Column = $F_{col}$ then
6:          Receive Block into $LocalMatrix$
7:          Transpose $LocalMatrix$
8:       end if
9:       if Block Below Diag AND Row = $F_{col}$ then
10:         Send $LocalMatrix$ to transposed position
11:      end if
12:   end for
13: end if
14: if Column = $F_{Col}$ then
15:   if Row = $Chk_{Row}$ then
16:      ColumnReduce $-ONE \cdot LocalMatrix$ To $F_{Row}$
17:   else
18:      ColumnReduce $LocalMatrix$ to $F_{Row}$
19:   end if
20: end if
```

Fig. 12. The steps performed by each node to recover from a failure needing the Hybrid row + column data. Data is the local data matrix, Row is the processor row, Column is the processor column, and F_{Row} and F_{Co} are the row and column of the failed node. Send, Receive, and ColumnReduce represent calling the respective MPI functions. It is assumed that a common ordering of blocks is available across processors.

For simplicity of implementation, it may also be advantageous for any process that contains unneeded blocks to ensure these blocks contain zeros as it enables column summation reduce calls on these blocks without special handling of the memory for these blocks. On the other hand, this may cause an overhead increase and can be avoided through careful implementation. p_f should also ensure that the contents of its local matrix initially are zeros.

The actions to recover any given block depends on the type of data that should be contained in that block and on what part of an iteration the failure occurred. Of these, there is no need to recover elements that are no longer needed, as these blocks are not part of either the result nor are still needed for computation.

The most straightforward blocks to recover are those of the resulting **L** matrix, as these do not depend on which step of an iteration the failure occurred. If a p_f is a checksum row process, any blocks in **L** can be recovered by performing a summation reduce call on p_f's process column. For any other process, the blocks in **L** are recovered by transmitting the contents of the checksum process of its column, and then subtracting the result of a summation reduce of processes 1 to P (where p_f contains zeros). This pattern of summation

Block	Step - sub-step	Recovery Point	Data Used
Unneeded	Any	Not recovered	None
L	Any	Final	Column
$A_{i,i}$,b_i	Diagonal factorization	Iteration start	Column
B_i	Diagonal factorization	Iteration start	Hybrid row + column
$A_{i,i}$, b_i	Panel update - Broadcast	Iteration start	Column
$A_{i,i}$, b_i	Panel update - Computation	End of update	Column
B_i	Panel update - Any	Iteration start	Column
$A_{i,i}$, b_i	Remaining matrix update	End of iteration	Column
B_i	Remaining matrix update - b_i broadcast	Start of update	Hybrid row + column
B_i	Remaining matrix update - b_i^T creation	Start of update	Hybrid row + column
B_i	Remaining matrix update - b_i^T broadcast	Start of update	Hybrid row + column
B_i	Remaining matrix update - computation	End of iteration	Hybrid row + column

TABLE 2
Recovery by block location and iteration sub-step summary for recovery on a lower triangular matrix.

reduce to reconstruct a checksum process, or subtracting the summation of all other processes' elements from the checksum process's elements is the basic strategy for recovery, and we refer to it as the *recovery strategy*.

At the beginning of each iteration, conceptually $A_{i,i}$, b_i, and B_i are partitioned. After the diagonal block factorization, $A_{i,i}$ contains $L_{i,i}$. Until the completion of the panel update, the column checksum for the panel column is not maintained. If p_f contains $L_{i,i}$, that block is reconstructed by recovering from the processes column and then restarting the iteration. If p_f contains any of the blocks in b_i, recovery occurs to the end of the panel update (i.e., all other processes complete the panel update and *then* recover the lost data in p_f). In the panel update, $L_{i,i}$ is broadcast to the processes of the rest of the panel b_i. If a failure occurs during the broadcast, it may be uncertain which of the unfailed processes have received the factorized diagonal block and a re-broadcast is necessary.

As the checksum block of the panel is also updated, the column checksum is again maintained for this column. In fact, after this point, the panel and is part of the **L** matrix. Recovering to the end of the panel update removes any need to recompute the result on any other process, or even to perform the explicit computations on p_f. That the recovery of p_f can actually occur to a point where the failed process had not yet reached reduces the recovery runtime.

To review, in the remaining matrix update step there is a row broadcast, the creation of b_i^T, a column broadcast, and computation. For the remaining matrix update steps, each block has a block from b_i and from b_i^T. As these are shared between rows and columns respectively of blocks of B_i on a process, we assume that the process holds these separately until the computation step. Therefore, if a failure occurs during the row broadcast, the creation of b_i^T, or the column broadcast, none of the elements of B_i have yet been affected. If the full Outer Product method were used (not just lower triangular), the blocks of B_i on p_f could be recovered through the *recovery strategy* on the process column. Since the upper triangular half of B_i is not maintained in the Right-Looking algorithm, the direct usage of the blocks above the diagonal is not possible. Fortunately, the needed data is held symmetrically in the lower triangular matrix. In a $P \times P$ process grid (with square block sizes), there is a one to one correspondence of processes in the row and column for these blocks As such, a transpose from the process row to the process column can be performed. The *recovery strategy* over the process column is used. We refer to this pattern as recovery over a hybrid row + column (See 12).

p_f also has to receive another copy of the needed blocks of b_i and b_i^T. These blocks are obtained through direct communication from any process on its process row and column respectively. Once the elements of b_i and b_i^T are distributed, all of the processes perform the update of B_i as $B_i \Leftarrow B_i$ - $b_i b_i^T$. If any process fails during this update, the remaining processes should complete their computation of the iteration, and then use the hybrid row + column to recover p_f to the end of the iteration.

As such, it is possible to recover any given block at any step of the computation without keeping information of any earlier iteration. We summarize the recovery steps by block type and step in Table 2. **L** is the inline result matrix. $A_{i,i}$ is the diagonal block of the iteration. b_i is the panel of the iteration. B_i is the remaining matrix. Unneeded are blocks above the diagonal.

4.4 Scalability of ABFT Cholesky

To determine the scalability of the ABFT Cholesky overhead and recovery times, we must know the runtime of the broadcast and reduce algorithms being used. As the best algorithms vary depending on several factors, we choose two families of collective operations to evaluate against: the Binomial family and the Pipeline family. The Binomial family broadcast and reduce work through distance doubling and reduce the total number of communications required to the order of $\log_2(P)$. The Pipeline family broadcast and

Family	$BCast(\eta, P_b)$
Binomial	$\lceil \log_2(P_b) \rceil \cdot \frac{\eta}{\eta_s}(\alpha + \eta_s \cdot \beta)$
Pipeline	$(P_b + \frac{\eta}{\eta_s} - 2)(\alpha + \eta_s \cdot \beta)$
	$Reduce(\eta, P_r)$
Binomial	$\lceil \log_2(P_b) \rceil \cdot \frac{\eta}{\eta_s}(\alpha + \eta_s \cdot \beta + \eta_s \cdot \gamma)$
Pipeline	$(P_r + \frac{\eta}{\eta_s} - 2)(\alpha + \eta_s \cdot \beta + \eta_s \cdot \gamma)$

TABLE 3
Runtimes for Binomial and Pipeline families [23]

Family	Cholesky Runtime Approximation
Binomial	$O(\frac{M^3\gamma}{P^2} + \frac{M^2 \log_2 P}{P \cdot MB} \cdot (\alpha + MB \cdot \beta))$
Pipeline	$O(\frac{M^3\gamma}{P^2} + \frac{M}{MB} \cdot (P + \frac{M}{P} - 2)(\alpha + MB \cdot \beta))$

TABLE 4
Runtimes for Cholesky

Family	**Checksum Setup**
Binomial	$O(\lceil \log_2 P + 1 \rceil \cdot \frac{M^2(\alpha + MB \cdot \beta + MB \cdot \gamma)}{MB \cdot P^2})$
Pipeline	$O((P + \frac{M^2}{MB \cdot P^2} - 1)(\alpha + MB \cdot \beta + MB \cdot \gamma))$
	Increased Matrix Size
Binomial	$O(\frac{(P+1)^2}{P^2} \frac{M^2}{MB \cdot P}(\alpha + MB\beta))$
Pipeline	$O(\frac{(P+1)^2}{P^2} \frac{M}{MB}(\alpha + MB\beta))$

TABLE 5
Overheads using the Binomial and Pipeline families

Family	**Checksum Setup**
Binomial	$O(\frac{\lceil \log_2(P+1) \rceil(\alpha + MB\beta + MB\gamma)}{M \cdot MB\gamma + P\lceil \log_2 P \rceil(\alpha + MB\beta)})$
Pipeline	$O(\frac{(P + \frac{M^2}{MB \cdot P^2} - 1)(\alpha + MB \cdot \beta + MB \cdot \gamma)}{\frac{M^3}{P^2} + \frac{M}{MB}(P + \frac{M}{P} - 2)(\alpha + MB \cdot \beta)})$
	Increased Matrix Size
Binomial	$O(\frac{M \cdot (\alpha + MB\beta)(P^2 + 2P + 1)}{P^2 \cdot MB(\frac{M^2\gamma}{P} + \lceil \log_2 P \rceil(\alpha + MB\beta))})$
Pipeline	$O(\frac{(P^2 + 2P + 1)(\alpha + MB\beta)}{M^2 \cdot MB + (P^3 + M \cdot P - 2P^2)(\alpha + MB\beta)})$

TABLE 6
Relative overheads for Cholesky runtime

reduce break up the data into pieces and stream the data through the communicating processes such that there is high utilization of every processes communication resources. The Pipeline family incurs a startup cost related to the number of processes involved so these algorithms tend to perform best when data is able to be broken up into many more pieces than there are processes. Table 3 shows the respective runtimes for $BCast$ and $Reduce$ operations assuming a message size of η_s under the Hockney model [18] with the data size η and message size η_s. We assume that the latency (α) and bandwidth (β) are not functions of message sizes.

One important consideration is the choice of message size. While the optimum message size on a system depends on topology, latency, and bandwidth among other factors, we choose the size MB as it allows more direct comparison and is a reasonable value in many cases. Table 4 shows the non-fault tolerant blocked Cholesky runtimes using these families under the Hockney model [18]. These values are derived by substituting the values from Table 3 into Equation 6.

We evaluate the checksum setup and increased matrix size overheads using the Binomial and Pipeline families for $BCast$ and $Reduce$ operations in Table 5. A key observation for the Binomial family is that $\lceil \log_2 P + 1 \rceil - \lceil \log_2 P \rceil$ has a maximum value of one enabling the simplification shown. We then divide the overheads by the runtime of the non-fault tolerant Right-Looking Cholesky routine in Table 6.

We now show that each of the four scaled overheads in Table 6 is scalable as M and P become large. Table 6 shows overheads for Cholesky runtime divided by the runtime of the non-fault tolerant routine using the Binomial and Pipeline families under the Hockney model [18] assuming a message size of MB. In the ideal case, the overhead diminishes as both M and P get large, but we note that the Pipeline family checksum setup overhead does not diminish, but does not grow as a fraction of runtime either. We start with the binomial checksum setup overhead. The denominator dominates the numerator in terms of both M (only in denominator) and in P ($\lceil \log_2 P + 1 \rceil < P \cdot \lceil \log_2 P \rceil$ as P becomes large). Therefore, as M, P, or a combination of the two get large, this overhead diminishes as a fraction of the runtime. For the binomial increased Matrix size overhead, the denominator dominates in P ($P^2 + 2P < P^2 \lceil \log_2 P \rceil$ as P becomes large) and in M ($M < M^2$). The differing coefficients are of concern if latency, bandwidth, or processing speed change drastically relative to each other (e.g., slowing a processor to save energy). We leave that analysis for future work.

For the Pipeline checksum setup overhead, the overhead diminishes with a larger M ($M^2 < M^3$), though this case too suffers from the differing coefficients mentioned before. Additionally, as P becomes large, the overhead as a fraction of runtime converges to $1 + \frac{MB\gamma}{\alpha + MB\beta}$. While this fraction does not grow, it unfortunately does not shrink as P becomes large. Fortunately, in common cases, P only grows when M grows, and the overhead fraction tends to reduce, though it does not converge to zero. The increased matrix size overhead for the Pipeline family diminishes as both P and M become large.

As the overhead under both families has been shown to be generally scalable as P and M increase, we now consider the scalability of the recovery routine. The dominating factor of the recovery routine is the $Reduce$ call to reconstruct the local matrix of p_f. Fortunately, the amount of data and number of processes involved is the same as for the checksum setup. As such, the recovery too is scalable as P and M become large.

For the performance of the system overall, it should also be considered that as the number of components grows, the expected number of failures grow similarly. As we are using additional checkpoint processes, the failure rate has increased by the number of checkpoint processes divided by the processes without checkpoints, or $\frac{2P+1}{P^2}$. As the number of processes grows, this failure rate converge to the failure rate of the calculation without fault tolerance added and therefore can be ignored for large scale systems.

Another concern is that the number of failures overall will increase as the process grid grows large, leading to longer runtimes. We note that this is not a function of overhead as we have considered it, but is a concern for minimizing the runtime of the application and that this increased failure rate occurs regardless of fault tolerance method.

The Binomial and Pipeline families cover two of the major behaviors for collective operations and are frequently used in MPI implementations. The scalability of ABFT Cholesky factorization under these two families shows promise for providing generically scalable fault tolerance. We next perform an evaluation of an implementation to verify the scalability.

5 Evaluation

To verify the correctness of recovery and runtime analysis, we compare two functions for doing the Right-Looking Cholesky factorization. The first, PDPOTRF, is the ScaLAPACK function for doing Cholesky factorization. In order to simulate a failure, a second Cholesky factorization routine, FTPDPOTRF, was written that assumes the full matrix with checksum row and column are given as parameters. The method implements the $P \times P$ with full process checksum matrix described in Section 4. As previously described, this method skips any data blocks along the diagonal that belong to the checksum processes. At the end of the method, the contents of the first $P \times P$ process contents are examined to verify they match the result from a standard call to PDPOTRF on the matrix without the checksum processes.

The recovery test method has three additional parameters that specify where and when a failure occurs. Specifically, it takes the row and column of the process to fail and the iteration failure occurs. In particular, it simulates a failure during either the diagonal factorization or during the panel broadcast update when the iteration may have to be restarted. For the $Reduce$ implementation, we allow the MPI environment to select an algorithm as our individual choice of algorithm would not be reflected in PDPOTRF either. Forcing a particular algorithm in PDPOTRF would also deviate its runtime from its more realistic use. While this testing is not adequate to do a full scale exponential distribution failure simulation, this method successfully tests that the recovery method and calculates the time required to recover.

The recovery function first sets up the checksum recovery to make a column reduction operation to the failed process provide all the data needed to recover. The recovery function then reduces to the failed process. Finally, the function uses the communicated data to set the values in the global matrix.

The first phase requires that each process go through its blocks and determine which is necessary for recovery. These blocks are maintained in a temporary matrix, which is used to perform the reduction. Additionally, since the **B** section of the matrix is not symmetric in ScaLAPACK, it is first necessary to get the transpose of this section of the matrix, find its transpose (minus the diagonal), and add it back to the temporary matrix. In the second stage, a column-wise by process grid communicator is established. If the column contains the failed process, then a summation reduction is performed. Upon completion of this step, the original matrix is recovered.

For the scalability testing, trials on the Kraken supercomputer were performed. Kraken is a Cray XT5 system featuring 12 core AMD "Istanbul" processors (2.6GHz). Specifically, tests using process grids of 35×35 (36×36 for FTPDPOTRF) and 107×107(108×108 for FTPDPOTRF) were performed. In these tests, the matrix was sized to 4000×4000 double precision elements per process. The runtime for PDPOTRF and FTPDPOTRF were measured, as well as the runtime for one call of the recovery routine. The block size was selected based upon the best of several smaller scale runs to be 200×200 elements. The runtimes for PDPOTRF were averaged over two runs, and FTPDPOTRF was averaged over four runs.

Figure 13 compares the wall-clock runtimes of PDPOTRF and FTPDPOTRF. FTPDPOTRF is slower than PDPOTRF on both the $P = 35$ and $P = 107$ sizes. Despite this, the difference between the two methods is a small fraction of the total runtime. The fraction of the runtime is shown in Figure 14. The wall-time fraction of total runtime is decreasing as was predicted in Section 4. Both Figure 13 and Figure 14 do not account for the additional overhead from the additional processors being used.

To account for the additional processors used, we scale the results by the number of processors used to produce a direct comparison of processor-time. We represent the data in processor-hours due to the scaled values. We show in Figure 15 the runtime used by PDPOTRF and FTPDPOTRF on the process grids with $P = 35$ and $P = 107$. Again, the difference does not appear to be significant in comparison with the overall runtime of either function. Figure 16 shows the explicit percentage overhead in reference to the runtime of PDPOTRF. As can be seen in the figure,

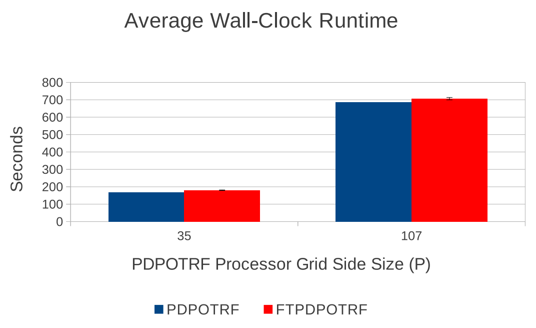


Fig. 13. Average wall-clock runtime of FTPDPOTRF ($(P{+}1) \times (P{+}1)$) and PDPOTRF (on $P \times P$) by in the global matrix size. Left has $P{=}35$ and right has $P{=}107$.

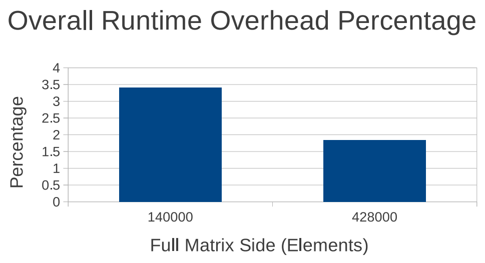


Fig. 14. Percent wall-clock runtime of overhead of FTPDPOTRF ($(P{+}1) \times (P{+}1)$) compared with PDPOTRF ($P \times P$). Left has $P{=}35$ and right has $P{=}107$.

the overhead is less than 10% on the $P = 35$ processor grid, and decreases to less than 4% on the $P = 107$ processor grid size. The reduction of overhead as a fraction of runtime as the processor grid and matrix grow is consistent with the analysis in Section 4.

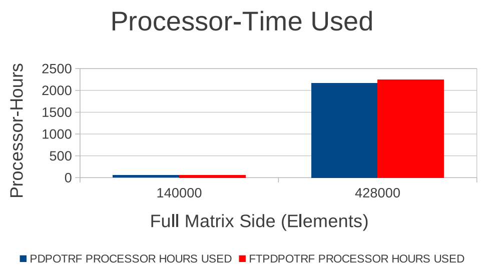


Fig. 15. Comparison of processor-hours used by FTPDPOTRF ($(P{+}1) \times (P{+}1)$) compared with PDPOTRF ($P \times P$) by number of elements in the global matrix. Left has $P{=}35$ and right has $P{=}107$.

Beyond the overhead, we also consider the scalability of recovery as the process grid and matrix size jointly grow. A scalable overhead is unfortunately of little use if the recovery time is not scalable as well. Figure 17 shows the average recovery in seconds of the PDPOTRF runtime for the data communication needed to recover a single failure. This recovery time

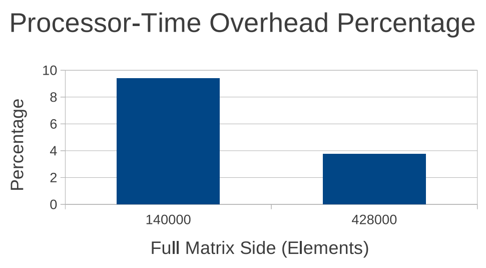


Fig. 16. Overhead viewed as percent of processor-hours used for FTPDPOTRF ($(P{+}1) \times (P{+}1)$) and PDPOTRF ($P \times P$) by number of elements in the global matrix. Left has $P = 35$ and right has $P = 107$.

is on the order of seconds even for the larger matrix size. As such, this supports the analysis in Section 4.

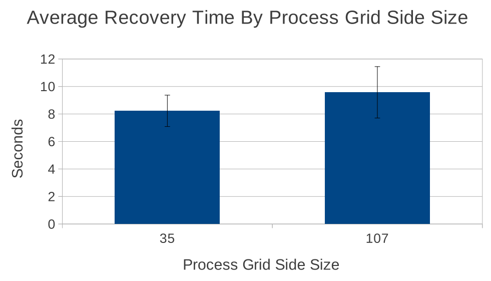


Fig. 17. Average wall-clock for restoring one processes data by process grid side size.

6 Conclusions

As matrices and process grids grow, fault tolerance becomes an increasing concern due to increased failure rates. We have presented a scalable method for adding Algorithm Based Fault Tolerance to the Cholesky Factorization through the use of a full checksum matrix on square process grids. Through this work we have made the following contributions:

- Shown that column checksums are maintained for the Outer Product Algorithm
- Shown that the row+column checksums can be used for the Right-Looking Algorithm
- Shown that checksums are not maintained for the Border Algorithm
- Developed an ABFT method that is compatible with 2D Block Cyclic Decomposition
- Showed the method is scalable as the process grid and matrix become large
- Developed a proof of concept implementation (FTPDPOTRF)
- Evaluated the overhead and recovery scalability on a process grid on the order of 100×100

ACKNOWLEDGEMENT

The authors would like to thank the anonymous reviewers for their insightful comments and valuable suggestions to improve the quality of this paper. This research is partly supported by National Science Foundation, under grants #CCF-1118039, #OCI-1150273, and #CNS-1118037.

REFERENCES

[1] A. Al-Yamani, N. Oh, and E. McCluskey. Performance Evaluation of Checksum-Based ABFT. In *IEEE International Symposium on Defect and Fault-Tolerance in VLSI Systems*, page 461, 2001.

[2] J. Anfinson and F. T. Luk. A Linear Algebraic Model of Algorithm-Based Fault Tolerance. *IEEE Transactions on Computing*, 37(12):1599–1604, 1988.

[3] P. Banerjee, J. Rahmeh, C. Stunkel, V. S. Nair, K. Roy, V. Balasubramanian, and J. Abraham. Algorithm-Based Fault Tolerance on a Hypercube Multiprocessor. *IEEE Transactions on Computers*, 39(9):1132–1145, 1990.

[4] R. Banerjee and J. Abraham. Bounds on Algorithm-Based Fault Tolerance in Multiple Processor Systems. *IEEE Transactions on Computers*, 35(4):296–306, 1986.

[5] 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.

[6] Z. Chen. Optimal Real Number Codes for Fault Tolerant Matrix Operations. In *Proceedings of the 2009 ACM/IEEE SuperComputing Conference on Supercomputing*, pages 29:1–29:10, 2009.

[7] Z. Chen and J. Dongarra. Algorithm-based Checkpoint-free Fault Tolerance for Parallel Matrix Computations on Volatile Resources. In *Proceedings of the International Parallel and Distributed Processing Symposium*, page 76, 2006.

[8] Z. Chen and J. Dongarra. Algorithm-Based Fault Tolerance for Fail-Stop Failures. *IEEE Transactions on Parallel and Distributed Systems*, 19(12):1628–1641, 2008.

[9] J. Choi, J. Dongarra, S. Ostrouchov, A. Petitet, D. Walker, and C. Whaley. Design and Implementation of the ScaLAPACK LU, QR, and Cholesky Factorization Routines. *Scientific Programming*, 5(3):173–184, 1996.

[10] T. Davies, Z. Chen, C. Karlsson, and H. Liu. Algorithm-based Recovery for HPL. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, pages 303–304, 2011.

[11] 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*, pages 162–171, 2011.

[12] P. Du, A. Bouteiller, G. Bosilca, T. Herault, and J. Dongarra. Algorithm-based Fault Tolerance for Dense Matrix Factorizations. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 225–234, 2012.

[13] E. Elnozahy and J. S. Plank. Checkpointing for Peta-Scale Systems: A Look into the Future of Practical Rollback-Recovery. *IEEE Transactions on Dependable Secure Computing*, 1(2):97–108, 2004.

[14] G. Fagg, E. Gabriel, Z. Chen, T. Angskun, G. Bosilca, J. Pjesivac-grbovic, and J. Dongarra. Process Fault Tolerance: Semantics, Design and Applications for High Performance Computing. *International Journal for High Performance Applications and Supercomputing*, 2004.

[15] E. Gabriel, G. Fagg, G. Bosilca, T. Angskun, J. Dongarra, J. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. Castain, D. Daniel, R. Graham, and T. Woodall. Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. In *Proceedings of the 11th European PVM/MPI Users' Group Meeting*, pages 97–104, 2004.

[16] G. Golub and C. Van Loan. *Matrix Computations (Johns Hopkins Studies in Mathematical Sciences)*. The Johns Hopkins University Press, 3rd edition, Oct 1996.

[17] D. Hakkarinen and Z. Chen. Algorithmic Cholesky Factorization Fault Recovery. In *Proceedings of the 2010 IEEE International Symposium on Parallel Distributed Processing*, 2010. ©IEEE 2010.

[18] R. Hockney. The Communication Challenge for MPP: Intel Paragon and Meiko CS-2. *Parallel Computing*, 20(3):389–398, 1994.

[19] K. Huang and J. Abraham. Algorithm-Based Fault Tolerance for Matrix Operations. *IEEE Transactions on Computers*, C-33(6):518–528, Jun 1984.

[20] G.-A. Klutke, P. C. Kiessler, and M.A. Wortman. A Critical Look at the Bathtub Curve. *IEEE Transactions on Reliability*, 52(1):125–129, 2003.

[21] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to parallel computing*, volume 110. Benjamin/Cummings Redwood City, 1994.

[22] F. Oboril, M.B. Tahoori, V. Heuveline, D. Lukarski, and J.-P. Weiss. Numerical Defect Correction as an Algorithm-Based Fault Tolerance Technique for Iterative Solvers. In *Proceedings of the 17th Pacific Rim International Symposium on Dependable Computing (PRDC)*, pages 144–153, Dec 2011.

[23] J. Pjeivac-Grbovi, T. Angskun, G. Bosilca, G. Fagg, E. Gabriel, and J. Dongarra. Performance Analysis of MPI Collective Operations. *Cluster Computing*, 10(2):127–143, 2007.

[24] P. Prata and J. G. Silva. Algorithm Based Fault Tolerance Versus Result-Checking for Matrix Computations. In *Proceedings of the Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing*, page 4, 1999.

[25] V. Stefanidis and K. Margaritis. Algorithm Based Fault Tolerant Matrix Operations for Parallel and Distributed Systems: Block Checksum Methods. In *Proceedings of the 6th Hellenic-European Conference on Computer Mathematics and its Applications*, pages 767–773, 2003.

[26] V. Stefanidis and K. Margaritis. Algorithm Based Fault Tolerance: Review and Experimental Study. In *Proceedings of the International Conference of Numerical Analysis and Applied Mathematics*, 2004.

[27] D.L. Tao, C.R.P. Hartmann, and Y. Han. New Encoding/Decoding Methods for Designing Fault-Tolerant Matrix Operations. *IEEE Transactions on Parallel and Distributed Systems*, 7(9):931–938, 1996.

[28] R. Wang, E. Yao, M. Chen, G. Tan, P. Balaji, and D. Buntinas. Building Algorithmically Nonstop Fault Tolerant MPI Programs. In *Proceedings of the IEEE International Conference on High Performance Computing*, 2011.

[29] E. Yao, M. Chen, R. Wang, W. Zhang, and G. Tan. A New and Efficient Algorithm-Based Fault Tolerance Scheme for A Million Way Parallelism. *CoRR*, abs/1106.4213, 2011.

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