On-line soft error correction in matrix–matrix multiplication

Panruo Wu\textsuperscript{b}, Chong Ding\textsuperscript{a}, Longxiang Chen\textsuperscript{b}, Teresa Davies\textsuperscript{a}, Christer Karlsson\textsuperscript{a}, Zizhong Chen\textsuperscript{b,∗}

\textsuperscript{a} Department of Electrical Engineering and Computer Science, Colorado School of Mines, Golden, CD 80401, United States
\textsuperscript{b} Department of Computer Science and Engineering, University of California, Riverside, 900 University Avenue, Riverside, CA 92521, United States

\textbf{A R T I C L E   I N F O}

Article history:
Received 30 March 2012
Received in revised form 14 February 2013
Accepted 7 May 2013
Available online 14 June 2013

Keywords:
Algorithm-based fault tolerance
Matrix multiplication
Fault tolerant linear algebra
On-line algorithm based fault tolerance

\textbf{A B S T R A C T}

Soft errors are one-time events that corrupt the state of a computing system but not its overall functionality. Soft errors normally do not interrupt the execution of the affected program, but the affected computation results cannot be trusted any more. A well known technique to correct soft errors in matrix–matrix multiplication is algorithm-based fault tolerance (ABFT). While ABFT achieves much better efficiency than triple modular redundancy (TMR) – a traditional general technique to correct soft errors, both ABFT and TMR detect errors off-line after the computation is finished. This paper extends the traditional ABFT technique from off-line to on-line so that soft errors in matrix–matrix multiplication can be detected in the middle of the computation during the program execution and higher efficiency can be achieved by correcting the corrupted computations in a timely manner. Experimental results demonstrate that the proposed technique can correct one error every ten seconds with negligible (i.e. less than 1%) performance penalty over the \texttt{ATLAS} \texttt{dgemm()}.

© 2013 Elsevier B.V. All rights reserved.

1. Introduction

As the number of components in high performance computers grows larger and larger, failures occur more and more frequently. In the meantime, driven by ever higher levels of detail and accuracy, many scientific applications today run for days or even months. To avoid re-starting computations after failures, the next generation high performance computing applications need to be able to detect and tolerate failures.

There are many types of failures in today’s computer systems. In this paper, we focus on tolerating soft errors, which are one-time events that corrupt the state of a computing system but do not interrupt the execution of the affected program. Soft errors are often fatal to scientific applications because they result in incorrect computation results [1,2]. Without error detection modules, soft errors during program execution are hard to be noticed. Soft errors have already been reported in tera-scale systems and are expected to be much more common in the next generation extreme scale platforms [3,4]. With relatively conventional architectural technology the fastest supercomputers in the world recently reached 10 petaflops barrier. However further advancement to exascale computing will require 20 times more energy efficiency than today’s most efficient systems and facing other related challenges such as fault resiliency. Large systems will be more susceptible to hardware errors due to reduced operating voltages and nanometer technology and increasing system components. To make matters worse the power consumption constraint will greatly limit the capacity of fault tolerant circuits. The traditional checkpoint / rollback and pairing / TMR techniques are becoming impractical for such extreme scale systems [5–7].

Matrix–matrix multiplication is a widely used operation in science and engineering. Soft errors in matrix–matrix multiplication can be tolerated using triple modular redundancy (TMR) – a general technique to correct soft errors in applications. TMR tolerates soft errors through voting on the final computation results of three duplicated computations and use the majority results as the correct results when there are differences between computation results. A much more efficient technique to correct soft errors in matrix–matrix multiplication is the well known algorithm-based fault tolerance (ABFT) technique [8], which detects soft errors at the end of the computation through verifying a pre-proved checksum relationship in the final computation results. ABFT has been extended to correct soft errors in the solution of system of linear equations in [9] using Sherman–Morrison formula. ABFT has been extended to correct hard errors in [10–16].

This paper presents a highly efficient on-line approach to detect, locate, and correct soft errors in the widely used matrix–matrix multiplication kernel – \texttt{ATLAS} \texttt{dgemm()} [17]. While the proposed approach can be treated as an extension of ABFT, the most prominent difference of the proposed approach is that it tolerates soft errors on-line, which means soft errors are detected, located, and corrected in the middle of the computation during the program execution.
The on-line property of our approach introduces lower fault tolerance overhead and allows better reliability and flexibility. In our on-line approach, soft errors can be detected and located before they propagate. Corrupted computations can be stopped or corrected in the middle of the program execution in a timely manner. Therefore, computation efficiency can be improved significantly. Furthermore, the frequency of the error detection can be flexibly adjusted according to the failure rate of the computing platform.

The idea of our online fault tolerance is not limited to matrix multiplication. It can also be applied to other commonly used linear algebra subroutines. Matrix–matrix multiplication is of special interest because it is the simplest case and it is widely used in other matrix operations. Linear algebra packages such as LAPACK and ScaLAPACK rely heavily on matrix–matrix multiplication to achieve high performance. Many scientific applications spend majority of their execution times on such common linear algebra operations [18]. Therefore protecting such most frequently used subroutines from soft errors can provide significant degree of fault tolerance for the whole application.

Experimental results demonstrate that the proposed technique can correct one error every minute with negligible (i.e. less than 1%) performance penalty over the ATLAS dgemm().

The rest of the paper is organized as follows. Section 2 discusses related work. Section 3 discusses the failure classification. Section 4 describes our on-line fault tolerant matrix–matrix multiplication algorithm. In Section 5, we analyze the overhead of our approach theoretically, and in Section 5, we evaluate our approach experimentally. Section 7 concludes the paper.

2. Related work

Many methods have been proposed to tolerate faults of computer systems. Triple modular redundancy (TMR) is a very general approach to ensure correctness of an application. A TMR system consists of three identical systems that perform identical computation and the result is given by a two-out-of-three voting mechanism. TMR is a classic hardware technique to tolerate soft errors in a large class of systems. However it needs at least a factor of 2 or 3 additional hardware redundancy to work [8], which makes it impractical for future HPC systems because of power and performance constraints [19]. There are also redundancy methods on node, process and thread level. The rMPI library [20] is an example of node level redundancy to tolerate node failures. Redundancy at process level and thread level has been investigated in [21,22].

Another general fault tolerance technique is checkpointing. The long running scientific applications typically tolerate failures by checkpoint/restart in which all application states are saved into stable storage periodically. The major source of overhead in all stable-storage-based checkpoint systems is the time it takes to write checkpoints into stable storage [23]. Checkpointing is effective for fail-stop errors which causes system crash. If such crash occurs, the system can be recovered to the recent checkpoint saved on storage. For fail-continue soft errors, checkpointing alone is not able to detect them thus cannot effectively tolerate them. On-line algorithm based fault tolerant methods can work with checkpointing in two ways: the computing state can be checked before being written to stable storages; the actual error rate can be reduced which results in less checkpoint overhead by Daly’s formula [24]. There is a lot of literature on various checkpoint techniques (see [25,23,26,6,27]).

In matrix operations, soft errors can often be detected off-line by algorithm based fault tolerance (ABFT). In this approach, applications are modified to operate on encoded data to ensure the correctness of calculations. It has been shown that the correctness of many matrix operations can be checked using ABFT by verifying a pre-verified checksum relationship at the end of the calculation [8]. This idea has been extended to tolerate fail-stop failures in distributed environments without checkpointing [28] and to tolerate fail-continue errors on GPU [29].


3. Failure classification

When a failure occurs during an application execution, if the failed process continues working, we define the failure as fail-continue failure. Fail-continue failures include computation errors by logic circuit and bit-flips in memory. They may be caused by many reasons including alpha particles from package decay, cosmic rays, thermal neutrons, and random noises. Fail-continue failures do not interrupt the program execution. But the computation results cannot be trusted any more.

Soft errors are one-time events that corrupt the state of a computing system but not its overall functionality. When a soft error occurs, it may or may not cause the crash of the system. Therefore a soft error may or may not be a fail-continue failure.

In this paper, we restrict our scope to fail-continue soft errors.

4. Error detection, location, and correction

Data redundancy can be exploited to tolerate faults against memory errors and CPU logic errors. In this paper we implement the data redundancy through checksum matrices. For description we introduce some notions and terms of checksum matrices. A column checksum matrix of matrix A, denoted by $A^c$, is defined by $A^c := \left( \begin{array}{c} A \\ v^TA \end{array} \right)$. The checksum vector $v$ is typically set as an all-one column vector. Similarly, A row checksum matrix of matrix B is defined as $B^c := \left( B Bw \right)$, where $w$ is a column checksum vector. Finally the full checksum matrix of matrix C is $C^f := \left( \begin{array}{c} C \\ v^TCw \end{array} \right)$.

Instead of multiplying matrices $A$ by $B$ we use their checksum versions, and instead of obtaining $C := AB$ we get its full checksum version $C^f := (AB)^f$.

$$A^c \times B^c = \left( \begin{array}{c} A \\ v^TA \end{array} \right) \times \left( \begin{array}{c} B \\ Bw \end{array} \right) = \left( \begin{array}{c} AB \\ v^TABw \end{array} \right) = (AB)^f = C^f$$

This extra information of the multiplication result can be used to detect, locate and possibly recover faults occurred during computation. If a soft error occurred, either because of memory fault or CPU computation fault, the faulty part will violate the checksum relationship of the result. If only one error occurred, exactly one row and one column of the result will not satisfy the checksum matrix definition. This relationship can be used to detect and locate the error. Furthermore, by solving a linear equation(s) it is possible to recover the faulty entry in C.

Our improvement over the traditional ABFT matrix multiplication is, by using the outer product version of matrix multiplication algorithm, we revealed that there are enormous opportunities during the operation to do the above checking and recovering. Instead of checking only at the end of whole matrix computation we can
do the checking many times as the computation progresses, which greatly improves the ability to detect and correct errors. Moreover, the frequency of checking during computations is flexible; it’s possible to adjust it in accordance with the expected error rates of a computer.

The detailed method will be discussed in the following subsections.

4.1. Fault tolerance for single error

To tolerate single error we assume checksum vectors v, w to be all-one vectors. Thus in a full checksum matrix $C^f$ the sum of each row of C is stored in the extra column, and the sum of each column of C is stored in the extra row of $C^f$. For brevity we assume $A, B$ are $n$ by $n$ square matrices. The checksum relationship can be mathematically expressed as:

$$c_{i, n+1} = \sum_{j=1}^{n} c_{i,j}$$

$$c_{n+1,j} = \sum_{i=1}^{n} c_{i,j}$$

$$c_{n+1,n+1} = \sum_{i,j=1}^{n} c_{i,j}$$

If one entry of matrix C, say $c_{i,j}$, is corrupted, it is easy to locate the fault by examining the above checksum relationship, in which case exactly one row $i$ and one column $j$ will fail the examination above. Once the fault is detected we may use either the row sum or column sum to correct the faulty entry $c_{i,j}$ by:

$$c_{i,j}^{\text{correct}} = c_{i,n+1} - \sum_{j=1, j \neq i}^{n} c_{i,j}$$

$$c_{n+1,j}^{\text{correct}} = c_{n+1,j} - \sum_{i=1, i \neq j}^{n} c_{i,j}$$

To tolerate multiple errors in C more sophisticated checksum vectors should be used. Readers are referred to [8] for details. However in our approach we are able to tolerate errors during computation so we do not need tolerating multiple errors in a single phase. We thus stick to the simple scheme outlined above.

This scheme of encoding $A, B, C$ into checksum matrices $A^f, B^f, C^f$ can only tolerate faults in C. If all $A, B, C$ are to be protected we may encode both $A, B$ into its full checksum matrix, and only use their partial checksum matrix (row checksum matrix for $B$ and column checksum matrix for $A$) in multiplication. Then all $A^f, B^f, C^f$ should go through the checksum relationship examination.

4.2. Checksum relationship maintained during block outer product matrix multiplication algorithm

Traditional ABFT matrix multiplication algorithms only check the checksum relationship (detecting, locating and correcting faults) at the end of the whole multiplication process. By employing block outer product matrix multiplication algorithm we could do as much as $n$ times fault tolerance during the multiplication process. The point of our scheme is that the checksum relationship is maintained during block outer product matrix multiplication, or more specifically at the end of each outer iteration of the block outer product algorithm.

We first define outer product version of matrix multiplication algorithm.

**Algorithm 1.** Outer product matrix multiplication

<table>
<thead>
<tr>
<th>Require:</th>
<th>$A, B$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ensure:</td>
<td>$C = A \times B$</td>
</tr>
<tr>
<td>for $s = 1 \rightarrow n$ do</td>
<td></td>
</tr>
<tr>
<td>$C_{s} = C + A([1:n, s] \times B([s, 1:n])$</td>
<td></td>
</tr>
<tr>
<td>end for</td>
<td></td>
</tr>
<tr>
<td>output $C$</td>
<td></td>
</tr>
</tbody>
</table>

The validity of this matrix multiplication algorithm can be easily verified by decomposing $A, B$ into column and row blocks.

$$AB = \begin{bmatrix} A_{1}, \ldots, A_{n} \end{bmatrix} \times \begin{bmatrix} B_{1}, \ldots, B_{n} \end{bmatrix}^T = A_{1}B_{1} + \cdots + A_{n}B_{n}$$

In the above outer product algorithm if we input checksum matrices $A^f, B^f$, we prove that at the end of every iteration (after each rank 1 update) the partial result $C$ which we denote by $C_s(s = 1, 2, \ldots, n)$ is a full checksum matrix. Let $A_i := A([1:n, 1:s])$ and $B_i := B([1:s, 1:n])$, then

$$C_{i} = A_i^f \times B_i^f = \begin{bmatrix} A_i \end{bmatrix} \times \begin{bmatrix} B_i \end{bmatrix} \times \begin{bmatrix} v^T A_i \end{bmatrix} = (A_iB_i)^f$$

The equation above shows that the partial product $C_i$ is a full checksum matrix, which makes it possible to do fault tolerating at the end of every iteration $s$.

In practice implementations of matrix multiplication using outer product multiplication are unlikely to be efficient. In order to make better use of cache system of a computer we need to use block algorithms. The modified version of outer product multiplication (the block outer product algorithm) is to do a rank $k$ update each iteration instead of a rank 1 update. For interface compatibility we use the same convention as in BLAS in which $\text{op}(A)$ means $A$ or $A^T$ depending on parameters passed into the subroutine.

**Algorithm 2.** Block outer product matrix multiplication

<table>
<thead>
<tr>
<th>Require:</th>
<th>$A, B, C, \alpha, \beta$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ensure:</td>
<td>$C = \alpha \times \text{op}(A) \times \text{op}(B)$</td>
</tr>
<tr>
<td>for $s = 1, s \leq \lfloor \frac{n}{k} \rfloor$ do</td>
<td></td>
</tr>
<tr>
<td>$C_{s} = C + \text{op}(A)([1:n, s], [s:k-1] \times \text{op}(B)(s:s+k-1, 1:n)$</td>
<td></td>
</tr>
<tr>
<td>end for</td>
<td></td>
</tr>
<tr>
<td>if $s \leq n$ else</td>
<td></td>
</tr>
<tr>
<td>$C_{s} = C + \text{op}(A)([1:n, s] \times \text{op}(B)(s:n, 1:n)$</td>
<td></td>
</tr>
<tr>
<td>end if</td>
<td></td>
</tr>
<tr>
<td>output $C$</td>
<td></td>
</tr>
</tbody>
</table>

Recalling that checksum relationship is closed under addition, similar to the outer product algorithm the partial result after each iteration in the corresponding block version above is also a full checksum matrix. Therefore we have opportunities to insert checking procedures into the algorithm, and check the checksum relationship regularly at the frequency we see fit. This results in the online version of ABFT matrix multiplication algorithm:

**Algorithm 3.** On-line ABFT matrix multiplication

<table>
<thead>
<tr>
<th>Require:</th>
<th>$A, B, C, \alpha, \beta$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ensure:</td>
<td>$C = \alpha \times \text{op}(A) \times \text{op}(B)$</td>
</tr>
<tr>
<td>encode $\text{op}(A), \text{op}(B), C$ as $\text{op}(A)^T, \text{op}(B)^T, C^T$</td>
<td></td>
</tr>
<tr>
<td>for $s = 1, s \leq \lfloor \frac{n}{k} \rfloor$ do</td>
<td></td>
</tr>
<tr>
<td>if $s$ reaches the point we need a check then</td>
<td></td>
</tr>
<tr>
<td>verify the checksum relationship of $C^T$</td>
<td></td>
</tr>
<tr>
<td>if checksum relation does not hold then</td>
<td></td>
</tr>
<tr>
<td>if only one line(i) and one column(j) fails the checksum verification then</td>
<td></td>
</tr>
<tr>
<td>recover the faulty entry $\frac{C_{i,j}^{T}}{C_{i,k}^{T}}$</td>
<td></td>
</tr>
<tr>
<td>end if</td>
<td></td>
</tr>
<tr>
<td>end if</td>
<td></td>
</tr>
<tr>
<td>end for</td>
<td></td>
</tr>
</tbody>
</table>


Our algorithm introduces much better reliability and flexibility than the original ABFT matrix multiplication algorithms.

4.3. Threshold to distinguish between roundoff error and soft error

In our algorithms we need to verify the checksum relationship, which requires adding a row or a column of entries and comparing their sums against the entries in the last row or column in C. Because of roundoff errors in floating point computations we cannot expect the checksum relationship to hold exactly. Then how to distinguish roundoff errors from soft errors becomes an issue. Apparently manually enforcing a threshold and assuming computation correct if differences are less than that threshold is unreliable. Too large a threshold may hide soft errors while too small one may interrupt correct computations. Therefore it’s helpful to develop a reasonable threshold based on concrete analysis.

A well known bound of matrix product roundoff error [30] is

$$|\|AB\| - AB\|_\infty \leq \gamma_n |A|_\infty |B|_\infty$$

(1)

where $\gamma_n = nu(1 - nu)$, and $u$ is the unit roundoff error of the target machine, and $n$ is the common dimension of the matrix $A$ and $B$. We begin by assuming the computations are correct, i.e. $\hat{C} - A \cdot B'$. As a convention the floating point version of a variable has a hat over the corresponding name. Then we have

$$|\sum_{j=1}^{n} \hat{C}_{ij} - \hat{C}_{i,n+1}| = |\sum_{j=1}^{n} (\hat{C}_{ij} - c_{ij}) - (\hat{C}_{i,n+1} - c_{i,n+1})| \leq |\sum_{j=1}^{n} (\hat{c}_{ij} - c_{ij})|

+ |(\hat{c}_{i,n+1} - c_{i,n+1})| \leq |\|C\|_1 - \|C\|_\infty |A||B|_\infty := \lambda$$

Therefore if the difference of the computed result satisfies $|\sum_{j=1}^{n} \hat{C}_{ij} - \hat{C}_{i,n+1}| \leq \lambda$ it is reasonable to claim that no errors other than roundoff errors have occurred. Otherwise we should regard it as a failure of checksum relationship examination and do the fault recovery procedure.

Similarly for the column sum there is a corresponding constant that serves as the threshold which is $\mu := \gamma_n |A||B|_1$.

5. Performance analysis

To quantify the efficiency of our approach, in this section, we analyze the overhead introduced by our fault tolerance approach theoretically. Let $1/\gamma$ denote the number of floating-point arithmetic operation per second (FLOPS) and $N$ denote size of the matrix (i.e. the matrix is of size $N \times N$). We assume $1/\gamma$ as average FLOPS. Let $T_{\text{overall}}$ denote the time for matrix multiplication, then it is well known that $T_{\text{overall}} = O(N^3)$.

5.1. Overhead for encoding

The time complexity of generating row or column checksum for input matrices can be expressed as follows

$$T_{\text{encode}} = N^2 \gamma$$

(2)

The overhead can be calculated by

$$\frac{T_{\text{encode}}}{T_{\text{overall}}} = O \left( \frac{1}{N} \right)$$

The time and overhead to construct a full checksum matrix are

$$T_{\text{encode,full checksum}} = 2N^2 \gamma$$

(3)

$$\frac{T_{\text{encode,full checksum}}}{T_{\text{overall}}} = O \left( \frac{1}{N} \right)$$

5.2. Overhead for computation

Encoded information which helps detect and recover errors will introduce overhead to compute $A' \times B'$ instead of just computing $A \times B$. The additional computation time due to the increase of the matrix size is

$$T_{\text{comp}} = 2(N + 1) \times N \times (N + 1) - 2N^3 = (4N^2 + 2N)\gamma \approx 4N^2 \gamma$$

(4)

The overhead, which has nothing to do with the number of errors, is

$$\frac{T_{\text{comp}}}{T_{\text{overall}}} = O \left( \frac{1}{N} \right)$$

5.3. Overhead for detecting errors

The process which scans a whole $(N + 1) \times (N + 1)$ matrix with full checksum once needs $2 \times N^2$ addition operations and $2 \times N$ branch operations. If the program is to tolerate $m$ errors, the time and overhead to detect a matrix is:

$$T_{\text{detect}} = 2mN^2 \gamma$$

(5)

$$\frac{T_{\text{detect}}}{T_{\text{overall}}} = O \left( \frac{1}{N} \right)$$

5.4. Overhead for recovery

For the simple case that matrix $C$ has only one encoded column and row, corrupted data can be recovered from the checksum relationship by just solving a linear equation. The overhead recovering data depends on the number $N$ and how many errors the fault tolerant matrix multiplication recovers as well. Assuming the program is designed to tolerate at most $m$ errors, the time complexity of recovery is

$$T_{\text{recovery}} = mN^2 \gamma$$

(6)

The recovery overhead is

$$\frac{T_{\text{recovery}}}{T_{\text{overall}}} = O \left( \frac{1}{N^2} \right)$$

The formula derived above shows the complexity of matrix multiplication dominates the complexity of the recovery overhead as the matrix size $N$ approaches infinity.

6. Experimental evaluation

In this section, we experimentally evaluate the performance of our fault tolerance approach. Three sets of experiments are performed to quantify the
• Performance and overhead of our on-line \texttt{FT-DGEMM} (Fault Tolerant General Matrix Matrix Multiplication).
• Performance comparison between our on-line \texttt{FT-DGEMM} and ABFT as well as TMR.
• Performance comparison between our on-line \texttt{FT-DGEMM} and ATLAS \texttt{DGEMM}.

All tests are performed on Alamode and Mio provided by CCIT and GECO in the Colorado School of Mines. The CPU on Alamode is Intel(R) Core(TM) i7-2600 CPU with 3.40 GHz. The CPU on Mio is Intel(R) Xeon(R) CPU E5530 with 2.40 GHz. The number of failures in the x-axis of all figures and texts in this section refers to the number of soft errors we can tolerate during one execution of our matrix matrix multiplication.

6.1. The overhead of our on-line \texttt{FT-DGEMM}

6.1.1. Overall overhead

The first set of experimental results report the overall overhead over the ATLAS \texttt{DGEMM} for our on-line fault tolerant matrix matrix multiplication with different number of failures (i.e. different failure rates) but fixed matrix size (i.e. 10,000). Fig. 1 shows the percentage of overhead our approach introduces when different number of failures are injected into the program execution. Two failures during the program execution equal to 0.8 failures per minute and twenty failures during the program execution equal to 7.5 failures per minute. Fig. 1 demonstrates that the proposed technique can correct one error every ten seconds (i.e. six errors during the whole program execution) with negligible (i.e. less than 1%) performance penalty over the ATLAS \texttt{dgemm}.

6.1.2. Overhead in detail

In the experiments, overhead of each part is timed to verify the derivations in Section 5. Result matches the analysis in Section 5. Overhead is shown in a stack in each independent run. Runtime of overhead of each part is clearly displayed to show the portion they take in the overall overhead. Figs. 2–5 show the execution time of overhead in stack bar figure. The runtime increases linearly as the failure rate grows. As we observe that the main growth is the overhead of detection whose frequency is determined by the failure rate.
According to Figs. 2–5, the execution time for generating column checksum for matrix \( A \) and row checksum for matrix \( B \) is unchanged for fixed matrix size which is a \( 4N^2 \) operation. Detecting errors is a process to calculate summation of each row and column of matrix \( C \) to test whether they match the value on row and column checksum. It is a \( O(mN^2) \) FLOPs computation. From Figs. 2–5, we can see that this portion of overhead increases linearly with the number of failures per execution, or failure rate the parameter of our implementation.

Recovery routines are implemented to try to recover corrupted entry in partial result \( C \). If faults are found in detection phase, the row and column index would be flagged and errors in the intersection of problem row and problem column would be corrected by our mechanism, which is a \( O(N) \) operation that is negligible.

The overhead of computation is introduced due to the increase of the matrix size from \( N \times N \) to \( (N+1) \times (N+1) \). As shown in Eq. (4), it is \( O(N^2) \) FLOP computation, which should be unchanged with fixed matrix size.

**6.2. Performance comparison: TMR vs ABFT vs on-line ABFT**

The set of data in Figs. 6–9 demonstrate performance comparison between on-line FT-DGEMM, ABFT and TMR with the same matrix size 10,000 and under three different actual failure rates. ABFT is a very famous technique to check the correctness of most matrix operation and recover the corrupted data which can tolerate fail-continue failures. TMR is a fault-tolerant mechanism in which three systems perform an identical process and the result is processed by a voting system to produce a single correct output. In our emulated TMR, we run the same program three times to tolerate faults during computation which results in incorrect data. To tolerate two errors the on-line FT-DGEMM simply examines the checksum relationship twice during the calculation. However traditional ABFT can only detect one error (in the worst case) at the end of computation given the same checksum matrices. If two errors occur, traditional ABFT has to re-run the program in hope to get the correct result. If two errors occur again in the re-run of the program, then a third run has to be performed. So the execution time of traditional ABFT is at least twice as much as the execution time of our approach, in the case two failures occurred during one matrix multiplication execution. In TMR, the same program has to be executed three times to produce three computation results for voting. Therefore, the total computation time to obtain correct result is at least three times the execution of the ATLAS DGEMM. Figs. 6–9 indicate that our on-line FT-DGEMM has much higher efficiency than ABFT and TMR.
Fig. 11. Performance for different failure rate (matrix size: 15,000).

Fig. 12. Performance for different failure rate (matrix size: 10,000).

6.3. Performance comparison: ATLAS DGEMM vs our on-line FT-DGEMM

In this section, we show the comparison of performance between non fault tolerant ATLAS DGEMM and our on-line FT-DGEMM. Two sets of tests are performed on both Alamode and Mio. Figs. 10–13 indicate that our online FT-DGEMM increases the reliability dramatically by only introducing a very low overhead. As we can see on the figures, the performance drops by no more than 5% when tolerating up to twenty soft errors during the program execution if at most one error occurs in each iteration. The flexibility and high reliability of our approach make it possible to it to various situations with low overhead.

Fig. 13. Performance for different failure rate (matrix size: 15,000).

7. Conclusion

In this paper, we extended the traditional ABFT technique for fault tolerant matrix–matrix multiplication from off-line to on-line. The most prominent difference of the proposed approach is that it tolerates soft errors on-line. In our on-line fault tolerance approach, soft errors are detected, located, and corrected in the middle of the computation during the program execution. Because soft errors can be detected and located before they propagate and corrupted computations can be stopped and corrected in the middle of the program execution in a timely manner, computation efficiency can be improved significantly. Experimental results demonstrate that the proposed technique can correct one error every ten seconds with negligible (i.e. less than 1%) performance penalty over the ATLAS dgemm( ).

Acknowledgments

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 US National Science Foundation, under grants #CNS-1304969, #CCF-1305622, and #OCI-1305624.

References


Panruo Wu received his Bachelor's degree in Mathematics from the University of Science and Technology of China in 2011. He is currently a PhD student at the University of California, Riverside. His research interests are in high performance computing, fault tolerance, and power-aware algorithms and software.

Chong Ding received his Bachelor's degree in Measurement & Control Technology and Instruments from Beijing University of Chemical Technology in 2009. He is currently a PhD student at the Colorado School of Mines. He is interested in high performance computing on GPUs with CUDA and OpenCL, and parallel programming with MPI.

Longxiang Chen received his Bachelor's degree in Automation from the University of Science and Technology of China in 2011. He is currently a PhD student at the University of California, Riverside. His research interests are in high performance computing, fault tolerance, and power-aware algorithms and software.

Teresa Davies received her Bachelor's degree in Electrical Engineering and Computer Science with Summa Cum Laude from the Colorado School of Mines in 2008. She is currently a PhD student at the Colorado School of Mines. She received a three-year NSF fellowship for her Ph.D. study at the Colorado School of Mines. Her research interests are in high performance computing and fault tolerance.

Christer Karlsson received his Bachelor's degree in Mathematics and Computer Science from the University of Wyoming in 2006. He received his Master's degree in Computer Science from the Colorado School of Mines in 2009. He is currently a PhD student at the Colorado School of Mines. His research interests are in topology aware MPI communications and fault tolerant high performance computing.

Zizhong Chen received a B.S. degree in mathematics from Beijing Normal University in China, in 1997, and M.S. and PhD degrees in Computer Science from the University of Tennessee, in 2003 and 2006, respectively. He is currently an assistant professor of computer science at the University of California, Riverside. His research interests include high performance computing, fault tolerance and checkpointing, power-aware algorithms and software, real number error/erasure correcting codes, numerical linear algebra algorithms and software, and computational science and engineering. He received a Distinguished Research Award from Jacksonville State University in 2008, an Outstanding Faculty Award from the Colorado School of Mines in 2010, and a CAREER Award from National Science Foundation in 2012. He is currently a senior member of the IEEE.