Preprint, final version at https://doi.org/10.1016/j.jpdc.2017.01.022

Fault Tolerant Communication-Optimal 2.5D Matrix Multiplication

Michael Moldaschl, Karl E. Prikopa, Wilfried N. Gansterer

University of Vienna, Faculty of Computer Science, Vienna, Austria

Abstract

In future computing systems, handling faults efficiently at the algorithmic level is expected to become more and more important. In this paper, we illustrate that in practice classical algorithm-based fault tolerance (ABFT) cannot protect all exponent bits of a floating-point number. Consequently, we extend the method to recover from bit-flips in all positions without additional overhead. We also derive fault detection conditions suitable for multiple checksum encoding vectors. Moreover, we show how to efficiently employ ABFT to protect communication-optimal parallel 2.5D matrix multiplication against bit-flips occurring silently during the computation. Furthermore, we show that for very low fault rates the overhead of fault tolerance in the context of the 2.5D matrix multiplication algorithms can be reduced even further. Numerical experiments on a high-performance cluster illustrate the high scalability and low overhead of our algorithms. We demonstrate the fault tolerance of our approach with randomly and asynchronously injected bit-flips and illustrate that our method can also handle bit-flips occurring at high frequencies. Like in classical ABFT, the overhead per correctable bit-flip of our approach decreases with increasing error rate.

1. Introduction

With the existence of petascale computing systems today and the objective of reaching exascale systems in the not too distant future, fault tolerance is a very important aspect of large high-performance systems which is attracting more and more interest [1, 2]. Many large applications run for days or weeks on such systems and it is important to ensure that the valuable computing time is not wasted due to faults during the computation. Nowadays it is no longer sufficient to design and implement an algorithm with the aim of high performance and good scalability, but these algorithms also need to be able to handle faults without having to re-compute the entire solution. There are a variety of fault types which have to be considered, from hardware faults to node crashes as well as soft errors like bit-flips or message loss. In this paper, we focus on silent *bit-flips* in the registers, the cache as well as in the main memory while the computation is in progress, because these are among the hardest types of faults to cope with. There are many reasons why bit-flips occur [2], e. g. cosmic rays. Measurements presented in [3] illustrate that over many ten-thousands of machines about a third of the machines exhibited at least one (soft or hard) memory error per year. Extrapolating this data to a system with millions of computing units, the mean time to failure would be less than a minute. This example already demonstrates that memory errors can occur with high frequency and therefore pose an important challenge on large-scale systems. Another example is given in [4], where 50 000 GPGPUs running on the Folding@home network were examined for their susceptibility to bit-flips. The results reveal that soft memory errors occurred on an overwhelming two-thirds of the tested GPUs. These numbers further emphasise the importance of fault tolerance on large-scale systems.

The use of error correcting codes (ECC) in memory chips is very common in high performance systems. However, higher fault rates would have to be conquered by adding redundancy in the circuits or by using

Email addresses: michael.moldaschl@univie.ac.at (Michael Moldaschl), karl.prikopa@univie.ac.at (Karl E. Prikopa), wilfried.gansterer@univie.ac.at (Wilfried N. Gansterer)

more powerful (and more costly) ECCs to protect the memory words [2]. According to [1], these improvements would lead to an increase of 20% in circuitry and energy consumption. Furthermore, the cost of the components and their development would be higher due to not having a high demand on the general market. Building exascale systems out of commodity components without ECCs would lower the costs but at the same time also lower the reliability levels of the hardware. Therefore, other techniques will be required to defend against higher fault rates.

Various approaches exist which can cope with at least some fault types. The most widely used method in high-performance computing (HPC) is checkpoint-restart (C/R) [5, 2], which saves the state of a computation at specific intervals and can recover from detected faults by rolling back to a check-pointed state. Its popularity is largely due to its general applicability. However, this approach tends to be relatively expensive in terms of overhead. Checkpoints generate a significant amount of I/O traffic and often block the progression of the application [2]. Another problem is that checkpointing only works if the error can be detected. Although several improvements have been developed to make C/R also usable on large systems [6, 7], its efficiency decreases with increasing system size [8]. Predictions on exascale systems expect a doubling of the total execution time of an application when C/R is being used [9]. Another method for handling faults is replication, where the application is executed either in parallel or sequentially multiple times. An example for this approach is triple modular redundancy (TMR) [10], which masks any single fault through majority voting. Naturally, the high resource cost in terms of either replicated hardware or multiple consecutive executions can be a major drawback in the context of HPC. Nevertheless, it has been shown that process replication strategies can outperform traditional C/R approaches for a certain range of system parameters [9]. Moreover, several benefits of *combining* C/R methods with redundancy (process replication) have been illustrated [11, 12].

An alternative approach, and the one we employ in this paper, is to design the algorithms themselves to be aware of silent faults. A prominent representative of this type of algorithms is *algorithm-based fault tolerance* (ABFT) [13], which basically adds a small amount of redundant data, the checksums, to the input to allow for detection and correction of a corresponding number of silent faults either during or at the end of the computation. Compared to C/R and replication techniques, ABFT achieves fault tolerance with much lower overhead and thus higher performance. Moreover, in contrast to C/R it can handle silent bit-flips occurring during the computation. The majority of the research on ABFT has been conducted for matrix multiplication, but the approach has also been applied to other linear algebra methods, including LU [13, 14], Cholesky [15] and QR [14] factorisation as well as Hessenberg reduction [16].

Our contributions. First, we discuss the limitations of classical ABFT, especially with respect to handling bit-flips in the exponent of a floating-point (FP) number (Section 3). As we will show, such faults can cause current ABFT methods to fail. We resolve these issues with our improved ABFT method, called *dABFT*, which protects *all* bits of a FP number without significant overhead (Section 4). The improvement even reduces the overhead compared to classical ABFT due to the efficient handling of the special floating values NaN and infinity during the fault detection step. We also derive fault detection conditions for multiple checksum encoding vectors, which up until now were not considered in the analysis of the error bound available in the literature. In Section 5, we provide a detailed analysis of the resilience of dABFT. We then combine the fault tolerance properties of dABFT with the high performance of 2.5D matrix multiplication methods [17, 18] to receive our *fault tolerant 2.5D matrix multiplication* (*2.5D FTMM*) (Section 6). For very low failure rates we show that we can further reduce the overhead of the fault tolerant method in the context of 2.5D algorithms. To demonstrate the fault tolerance of our approach, we have developed our own fault injector which asynchronously injects random bit-flips during the computation with a very low overhead of on average just 1% of the total execution time. In Section 7 we illustrate the high scalability and low overhead of our 2.5D FTMM algorithms on a high-performance cluster.

2. Related Work

For our method 2.5D FTMM, we combine two key components: a high performance parallel matrix multiplication and fault tolerance at the algorithmic level. In the following, we discuss the state-of-the-art

of both components.

2.1. Parallel Matrix Multiplication

Cannon's algorithm [19] for 2D meshes is one of the earliest parallel algorithms for matrix multiplication. However, the method is hard to generalise for rectangular grids and matrices. It has therefore since been superseded by SUMMA (scalable universal matrix multiply algorithm) [20], which has overcome the restrictions imposed by Cannon and uses blocked computations and pipelining to improve the performance. SUMMA is also the algorithm implemented in ScaLAPACK.

In recent years, communication-avoiding algorithms have been developed and a communication-optimal parallel 2.5D matrix multiplication (2.5D MM) has been presented [17]. This algorithm is based on Cannon's algorithm, but uses extra memory to store multiple replicates of the matrices to asymptotically reduce the communication cost. 2.5D MM is actually a generalisation of 2D and 3D methods, which either store a single copy (2D) or $q^{1/3}$ copies (3D) of the matrices, where q is the total number of processes. The 2.5D MM chooses a value c for the number of replications with $1 \leq c \leq q^{1/3}$. The chosen c aims to use the available memory optimally to achieve the lowest communication cost, reducing the bandwidth cost by $c^{1/2}$ and the latency cost by $c^{3/2}$ compared to the 2D version, but increases the memory cost by a factor of c. For rectangular grids and matrices, other methods have been developed, including 2.5D SUMMA [18], 3D SUMMA [21], hierarchical SUMMA [22] and communication-optimal parallel recursive rectangular matrix multiplication (CARMA) [23].

2.2. Algorithm-based Fault Tolerance

ABFT [13] handles faults at the algorithmic level by adding rows or columns to the matrices containing checksum encoded information about the data. An operation has to be checksum preserving to be able to use ABFT. Most research about ABFT focuses on node failures. In this variant, sometimes called *global ABFT*, the checksums are stored in additional processor rows and columns dedicated to the checksum encoded values. A fault tolerant MPI implementation can be used to handle node crashes (e. g. User Level Failure Mitigation [24]).

In the ABFT approach for recovering from bit-flips, the result of an operation is checked for faults by recomputing the checksums after the operation has completed and comparing the new checksums with the ones returned in the result. This indicates an important limitation: usually, only the final result of a matrix operation can be corrected. An exception has been presented in [25], where an outer product matrix multiplication uses global ABFT for fault detection and correction *during* the computation instead of only in the final result. The outer product preserves the checksum relationship at every step of the computation, unlike other matrix multiplication algorithms where the checksums are only consistent with the result after the computation is complete.

Another important aspect are numerical problems which can arise due to the growth of checksums with the problem size [26]. It has been shown that the numerical accuracy of ABFT can be improved by using checksums for blocks of data instead of global checksums (*local* ABFT [27]). Although the blocked variant improves the handling of bit-flips, it cannot recover from node failures due to the missing global checksums. However, the detection and correction of faults is confined to local operations on the computing node itself and therefore does not incur any communication overhead. In [28], the authors focus on the reduction of false positives due to numerical round-off errors and on the detection of faults in the lowest bits of the mantissa. They introduce "mantissa-preserving" checksums which are additional integer checksums of the mantissa. However, their approach can only protect multiplicative and not additive operations and therefore does not protect the complete matrix multiplication. Furthermore, the authors neither consider nor test bit-flips in the exponent.

Bosilca et. al. [29] suggested to combine ABFT and C/R. During the execution of ABFT methods, C/R would be disabled and only protects sections of the application otherwise prone to faults. Based on their performance model, the scalability is significantly better using this combined approach than only using C/R, while protecting the entire application.

In this paper, we will discuss the numerical problems arising in ABFT due to bit-flips in the exponent and how they can be handled. These important questions have not yet been discussed in the literature. We

will show how to protect 2.5D Cannon and 2.5D SUMMA against bit-flips using ABFT. Our novel method 2.5D FTMM can recover from faults after every local matrix multiplication. CARMA will not be discussed in this paper due to its non-static data layout, which results from the recursive approach. The efficient combination of ABFT with CARMA is left for future research.

3. ABFT for Matrix Multiplication

In this section, we first review the classical ABFT method for matrix multiplication and then discuss the current limitations of the existing ABFT methods.

3.1. Review of ABFT for Matrix Multiplication

ABFT methods are verification-based and it is assumed that no faults occur during the verification process. This process consists of the following steps: *(i)* recompute the checksums, *(ii)* compare the checksums with the available checksums in the result of the operation, and *(iii)* if a fault is detected, solve a single least squares problem for all discrepancies.

The ABFT matrix multiplication to compute $C = AB$ with $A, B, C \in \mathbb{R}^{n\times n}$ is described in Algorithm 1. First, augmented matrices $A^c \in \mathbb{R}^{(n+d)\times n}$ and $B^r \in \mathbb{R}^{n\times(n+d)}$ are defined as

$$A^c = \begin{pmatrix} A \\ W^\top A \end{pmatrix} \quad \text{and} \quad B^r = \begin{pmatrix} B & BW \end{pmatrix}. \tag{1}$$

$W \in \mathbb{R}^{n\times d}$ is the weight matrix and d denotes the number of checksums and therefore also the number of rows and columns that are added to the input matrices A and B. In the original publication of ABFT [13], only a single vector (i. e. $d = 1$) with all entries equal to one is used to compute the checksums and detect errors. This choice corresponds to forming the sum of all elements in a row or column. In the literature, a second vector consisting of powers of two is often added, leading to the following weight matrix for $d = 2$:

$$W^\top = \begin{pmatrix} 1 & 1 & \dots & 1 \\ 2^0 & 2^1 & \dots & 2^{n-1} \end{pmatrix}.$$

However, as one can easily deduce, the entries of $W^\top(2,:)$ grow exponentially fast with n and in floating-point arithmetic the large coefficients will make it impossible to detect many errors [30]. In [31], the weighted checksum encoding scheme is proposed, which uses d encoding vectors to detect and correct multiple faults. With d row or column checksums ABFT can guarantee the detection of up to d errors and the correction of up to $\lfloor d/2 \rfloor$ errors per row or column.

Additionally, the correction matrix $H \in \mathbb{R}^{d\times(n+d)}$ defined by $H := \begin{pmatrix} W^\top & -I_d \end{pmatrix}$ is required, where $I_d \in \mathbb{R}^{d\times d}$ is the identity matrix of dimension d. The only condition imposed on the encoding vectors is that every possible combination of $d-1$ columns of H has to be linearly independent [31]. Multiplying the augmented matrices (1) results in the extended matrix

$$C^f = A^c B^r = \begin{pmatrix} AB & ABW \\ W^\top AB & W^\top ABW \end{pmatrix}. \tag{2}$$

The upper-left block of C^f is identical to C and augmented by row and column checksums with checksums of the checksums residing in the lower-right block. Since C^f has full checksums, which corresponds to d row *and* d column checksums, $2d$ faults can be detected and d faults can be corrected.

In classical ABFT as shown in Algorithm 1, the multiplication in line 1 is the only operation which is allowed to be unreliable. As we will show in Section 6.2, this restriction is not always necessary.

After the unreliable matrix computation, a fault detection and, if necessary, a correction step are performed. The detection process is shown in lines 3-8 in Algorithm 1. Based on the augmented result matrix C^f, the matrices S_1 and S_2 are computed for the row and column checksums of C^f, respectively. First, W is applied (from the left or the right) to a submatrix of C^f which recomputes the checksums and subsequently the subtractions in lines 5 and 6 form the difference between them and the checksums stored in

Algorithm 1 Classical ABFT for Matrix Multiplication (ABFT)

Input: $A^c \in \mathbb{R}^{(n+d)\times n}, B^r \in \mathbb{R}^{n\times(n+d)}, W \in \mathbb{R}^{n\times d}$
Output: $C^f \in \mathbb{R}^{(n+d)\times(n+d)}$

1: Unreliable $C^f = A^c B^r$
2: Set all NaN and infinity in C^f to 0
3: $C_1 = W^\top C^f(1:n, 1:n+d)$
4: $C_2 = C^f(1:n+d, 1:n)W$
5: $S_1 = C_1 - C^f(n+1:n+d, :)$
6: $S_2 = C_2 - C^f(:, n+1:n+d)$
7: $I_1 = \{j \in [1,n] | \max_{k_r=[1,d]} \frac{|S_1(k_r,j)|}{\|W^\top(k_r,:)\|_p \|A\|_p \|B(:,j)\|_p} > 2\,(2+\mu_n)\,\mu_n\} \bigcup$
$\{k_c \in [1,d] | \max_{k_r=[1,d]} \frac{|S_1(k_r,j)|}{\|W^\top(k_r,:)\|_p \|A\|_p \|B\|_p \|W(:,k_c)\|_p} > 2\mu_n(3+3\mu_n+\mu_n^2)\}$ ▷ see (8) and (10)
8: $I_2 = \{i \in [1,n] | \max_{k_c=[1,d]} \frac{|S_2(i,k_c)|}{\|A(i,:)\|_p \|B\|_p \|W(:,k_c)\|_p} > 2\,(2+\mu_n)\,\mu_n\} \bigcup$
$\{k_r \in [1,d] | \max_{k_c=[1,d]} \frac{|S_2(i,k_c)|}{\|W^\top(k_r,:)\|_p \|A\|_p \|B\|_p \|W(:,k_c)\|_p} > 2\mu_n(3+3\mu_n+\mu_n^2)\}$ ▷ see (7) and (9)
9: **if** $I_1 \neq \{\}$ **and** $I_2 \neq \{\}$ **then**
10: Solve $H(:, I_2) \cdot \Delta C = S_1(:, I_1)$ for ΔC
11: $C^f(I_2, I_1) = C^f(I_2, I_1) - \Delta C$
12: **end if**

C^f. The sets of indices I_1 and I_2 computed in lines 7 and 8 indicate the faulty rows and columns detected by evaluating the novel fault detection conditions, which will be discussed in detail in Section 3.2. If faulty values are found, the correction process (lines 9-12 in Algorithm 1) computes a correction matrix ΔC for each faulty value by solving an overdetermined linear least squares problem (line 10). In *exact* arithmetic, as long as the number of faults is less than or equal to d, the erroneous matrix C^f can be fully corrected. If more than d faults occur, the least squares problem is underdetermined and the corrupted elements in C^f cannot be recovered. Finally, the correction matrix ΔC is subtracted from the corrupted values in C^f at the intersection of the indices I_1 and I_2.

3.2. ABFT in Floating-Point Arithmetic

The distinction between bit-flips and numerical round-off errors is difficult. As long as the fault detection condition is not too strict, a limited number of false positives can be handled (depending on the number of checksums d). In [32], fault detection conditions have been defined for the case $d = 1$ based on the standard round-off error bound of the matrix product. It is stated in [32] that no bit-flips have occurred in row i of C^f and deviations in the checksum are only due to round-off errors, if

$$\left|\Sigma_{j=1}^{n} C^f(i,j) - C^f(i,n+1)\right| \leq \mu_n \left\|A^c\right\|_\infty \left\|B^r\right\|_\infty$$

and analogously, no bit-flips have occurred in column j of C^f if

$$\left|\Sigma_{i=1}^{n} C^f(i,j) - C^f(n+1,j)\right| \leq \mu_n \left\|A^c\right\|_1 \left\|B^r\right\|_1$$

where $\mu_n = \frac{nu}{1-nu}$ and u is the unit round-off error.

The fact that the computation of the checksums itself is also affected by numerical round-off errors has not been considered in the literature, mainly due to most of the time only a single weight vector being used whose elements are all equal to one. We therefore generalise the previous fault detection conditions for $d \geq 1$ encoding vectors, which requires the consideration of the weight matrix W on both sides of the inequalities. To derive this generalisation, we use the well-known error bound of the matrix-matrix product in floating-point arithmetic described in [33]. There, the computed result $\hat{C}$ is defined as $\hat{C} = [A + \Delta A]B$ and the exact result as $\bar{C} = AB$. For ΔA, it is shown that

$$\left\|\Delta A\right\|_p \leq \mu_n \left\|A\right\|_p \tag{3}$$

where $p = 1, \infty, F$. The error bound is then stated as

$$\left\|\bar{C} - \hat{C}\right\|_p = \|AB - [A + \Delta A]B\|_p = \|\Delta AB\|_p \leq \mu_n \|A\|_p \|B\|_p \ .$$

We now apply the same steps to derive the error bounds for each individual checksum block of C^f in (2).

We denote the exact result without numerical errors and unaffected by faults by $\bar{C}^f$ and the computed result with numerical errors but no faults due to bit-flips by $\hat{C}^f$. C^f denotes the computed result from the unreliable matrix multiplication (line 1 in Algorithm 1), which, in addition to numerical errors, can also be affected by bit-flips. For any row $k_r \in [1, d]$, each representing a single column-checksum entry, and any column $j \in [1, n]$, located in the lower left block $W^\top AB$ of C^f, we have

$$\begin{aligned}\hat{C}^f(n + k_r, j) = \ & \left[W^\top(k_r, :) + \Delta W^\top(k_r, :)\right][A + \Delta A]B(:, j) \\ = \ & W^\top(k_r, :) \cdot A \cdot B(:, j) + \left[W^\top(k_r, :) + \Delta W^\top(k_r, :)\right] \Delta A \cdot B(:, j) + \Delta W^\top(k_r, :) A \cdot B(:, j).\end{aligned}$$

The error bound for a single column-checksum entry is then defined as

$$\begin{aligned}\left|\bar{C}^f(n + k_r, j) - \hat{C}^f(n + k_r, j)\right| \leq \ & \left\|W^\top(k_r, :) + \Delta W^\top(k_r, :)\right\|_p \mu_n \|A\|_p \|B(:, j)\|_p + \\ & \mu_n \left\|W^\top(k_r, :)\right\|_p \|A\|_p \|B(:, j)\|_p \\ \leq \ & (2 + \mu_n)\, \mu_n \left\|W^\top(k_r, :)\right\|_p \|A\|_p \|B(:, j)\|_p \ . \end{aligned} \tag{4}$$

Analogously, for any column $k_c \in [1, d]$, each representing a single row-checksum entry, and any row $i \in [1, n]$, located in the upper right block ABW of C^f, we obtain the error bound

$$\left|\bar{C}^f(i, n + k_c) - \hat{C}^f(i, n + k_c)\right| \leq (2 + \mu_n)\, \mu_n \|A(i, :)\|_p \|B\|_p \|W(:, k_c)\|_p \, . \tag{5}$$

For the lower right block $W^\top ABW$ of C^f, which contains the checksums of the checksums, we have

$$\begin{aligned}\hat{C}^f(n + k_r, n + k_c) = \ & \left[W^\top(k_r, :) + \Delta W^\top(k_r, :)\right][A + \Delta A]\, B\, [W(:, k_c) + \Delta W(:, k_c)] \\ = \ & W^\top(k_r, :) \cdot A \cdot B \cdot W(:, k_c) + \Delta W^\top(k_r, :)\, [A + \Delta A] \cdot B\, [W(:, k_c) + \Delta W(:, k_c)] + \\ & W^\top(k_r, :) \Delta AB\, [W(:, k_c) + \Delta W(:, k_c)] + W^\top(k_r, :) AB \Delta W(:, k_c)\end{aligned}$$

where $k_r, k_c \in [1, d]$. The corresponding error bound is expressed as follows:

$$\begin{aligned}\left|\bar{C}^f(n + k_r, n + k_c) - \hat{C}^f(n + k_r, n + k_c)\right| \leq \ & \mu_n \left\|W^\top(k_r, :)\right\|_p (1 + \mu_n) \|A\|_p \|B\|_p (1 + \mu_n) \|W(:, k_c)\|_p + \\ & \mu_n \left\|W^\top(k_r, :)\right\|_p \|A\|_p \|B\|_p (1 + \mu_n) \|W(:, k_c)\|_p + \\ & \mu_n \left\|W^\top(k_r, :)\right\|_p \|A\|_p \|B\|_p \|W(:, k_c)\|_p \\ = \ & \mu_n(3 + 3\mu_n + \mu_n^2) \left\|W^\top(k_r, :)\right\|_p \|A\|_p \|B\|_p \|W(:, k_c)\|_p \ . \end{aligned} \tag{6}$$

Transforming inequalities (4), (5) and (6) to fault detection conditions leads to the following four conditions for the different blocks of C^f. Naturally, we do not have access to the exact result $\bar{C}^f$. Therefore, we use the re-computed checksums from lines 3 and 4 in Algorithm 1 as the values of $\bar{C}^f$. The checksums from the unreliable matrix multiplication in line 1 of Algorithm 1 are used for $\hat{C}^f$. Both checksums are computed in finite precision. They use the same operations but only differ in their order. In the worst case, the second computation of the checksums could double the numerical error. Therefore, the numerical error for both checksums is considered in the four fault detection conditions by multiplying the right-hand side by 2. Based on (5), for the row checksums, i. e. the upper right block of C^f, a fault is detected in row $i \in [1, n]$ if

$$\max_{k_c \in [1,d]} \frac{\left|\Sigma_{j=1}^n C^f(i, j) W(j, k_c) - C^f(i, n + k_c)\right|}{\|A(i, :)\|_p \|B\|_p \|W(:, k_c)\|_p} > 2\,(2 + \mu_n)\, \mu_n \ . \tag{7}$$

The lower left block of C^f contains the column checksums and based on (4), a fault is detected in column $j \in [1, n]$ if

$$\max_{k_r \in [1,d]} \frac{\left|\Sigma_{i=1}^{n} W^\top(k_r, i) C^f(i, j) - C^f(n + k_r, j)\right|}{\|W^\top(k_r, :)\|_p \, \|A\|_p \, \|B(:, j)\|_p} > 2\,(2 + \mu_n)\,\mu_n \ . \tag{8}$$

To detect faults in the checksums of the checksums in the lower right block of C^f the following two different conditions are formulated, depending on which checksums are used in the comparison. Based on (6), for the column checksum $k_r \in [1, d]$ a fault is detected if

$$\max_{k_c \in [1,d]} \frac{\left|\Sigma_{j=1}^{n} C^f(n + k_r, j) W(j, k_c) - C^f(n + k_r, n + k_c)\right|}{\|W^\top(k_r, :)\|_p \, \|A\|_p \, \|B\|_p \, \|W(:, k_c)\|_p} > 2\mu_n(3 + 3\mu_n + \mu_n^2) \tag{9}$$

and for the row checksum $k_c \in [1, d]$ a fault is detected if

$$\max_{k_r \in [1,d]} \frac{\left|\Sigma_{i=1}^{n} W^\top(k_r, i) C^f(i, n + k_c) - C^f(n + k_r, n + k_c)\right|}{\|W^\top(k_r, :)\|_p \, \|A\|_p \, \|B\|_p \, \|W(:, k_c)\|_p} > 2\mu_n(3 + 3\mu_n + \mu_n^2) \ . \tag{10}$$

Both conditions have to be evaluated to ensure the detection of all faults in the checksums of the checksums. The numerator of (7) and (9) corresponds to $|S_2(i, k_c)|$ (see line 6 in Algorithm 1) and of (8) and (10) to $|S_1(k_r, j)|$ (see line 5 in Algorithm 1).

The conditions only have to hold for one of the checksums in k_c or k_r to indicate the existence of a faulty value in the corresponding row or column, respectively. For example, a fault is detected in row i of C^f if condition (7) holds for at least one $k_c \in [1, d]$. Both conditions (7) and (8) (or (9) and (10) for the checksums of the checksums) have to be fulfilled to retrieve the column indices I_1 and the row indices I_2. The locations of the faulty values are given by the Cartesian product of I_1 and I_2 (see lines 7-8 in Algorithm 1). If only a column or only a row index is found, then this fault is treated as a numerical error.

In Section 5.2, we will illustrate experimentally that Conditions (7)-(10) guarantee highly accurate fault correction despite bit-flips.

3.3. Limitations of Existing ABFT Methods

Unfortunately, in all exiting ABFT methods the ability to correct faults is limited. More specifically, the limitations can be distinguished along two main dimensions – temporal and spatial limitations.

Temporal limitations. In the literature, faults are generally only allowed to occur during the matrix multiplication (line 1 in Algorithm 1) and not during the detection and correction process. More precisely, after an element of the matrix C^f has been checked for faults and is considered to be correct, no faults are allowed to occur in that element. This implies that faults are also allowed to occur in lines 3 and 4 of Algorithm 1. In floating-point arithmetic, this is only possible as long as the fault does not change the value to NaN. Any faults occurring during or after the correction process would not be detected or corrected and lead to an incorrect result in C. This limitation cannot be completely eliminated, but in the context of the 2.5D matrix multiplication it can be postponed until after the last distributed computation, as we will show in Section 6.2.

Spatial limitations. In floating-point arithmetic, classical ABFT can only handle faults in the mantissa or sign bits, but in practice faults can obviously occur at any position of the floating-point representation. Bit-flips in the exponent bits are much harder to correct due to numerical aspects. For example, consider the special case where all exponents of the original data are zero and a bit-flip changes one exponent to $\beta > 0$, then in the worst case (in double precision) the error caused by the fault can be reduced at most to $O(10^{-16+\beta})$ due to the cancellation of the least significant bits during the computation of S_1 and S_2. For $\beta \geq 16$ this leads to all bits being incorrect. This also implies that classical ABFT does not work if the magnitudes of the elements of the input data differ widely.

In the following, we provide a simple example in more detail for demonstrating the spatial limitation of the fault correction of classical ABFT. Without loss of generality, we use the decimal system for easier

illustration, with a maximum of 16 decimal digits. Consider the matrix C as a scalar with $C = 1.23999$ being the correct value before a fault has occurred. Using $W = 1$, the checksum is then also 1.23999. Due to a fault, the value of C is changed to $1.1 \cdot 10^{14}$. To correct the fault, this erroneous value is subtracted from the checksum. During this subtraction, the checksum value is normalised to the same exponent as the new value of C and therefore truncated to 1.23. The truncation error is $O(10^{-2})$ (corresponding to the case of $\beta = 14$ in the previous paragraph) and results in a corrected C of 1.23.

4. Improving the Resilience of ABFT

In the literature, bit-flips are implicitly modeled to only affect the mantissa of a floating-point number. Our experiments demonstrate that, due to the spatial limitations mentioned in Section 3.3, classical ABFT cannot handle all bit-flips occurring in the exponent (see Section 5.2). In this section, we improve the existing ABFT methods for matrix multiplication to be able to detect and correct bit-flips in all parts of a floating-point number, in particular allowing bit-flips also to occur in the exponent.

A large change in an exponent dominates the checksum of a faulty row or column. The correction can only reduce the fault by the maximum accuracy of the floating-point representation (about 16 decimal digits in double precision). Our novel idea is to compute the correction matrix ΔC *without* using the corrupted values. Therefore, we set all faulty elements in C^f to zero (line 9 in Algorithm 2) and recompute the corresponding part of the matrices S_1 and S_2 (lines 10-13). This enables our improved ABFT method to handle any bit-flips in the result, including all bits of the exponent, without significant overhead compared to classical ABFT (see Section 5). We call our novel method *direct ABFT* (*dABFT*), motivated by the way the correction matrix is computed. The resulting algorithm for dABFT matrix multiplication is shown in Algorithm 2.

The formal proof of correctness of ABFT is not affected by the modifications introduced here for dABFT. Each detected faulty value is just replaced with zero, which is in itself just a different faulty value. Instead of computing the difference between the faulty value and the checksums (like in classical ABFT), dABFT computes the correct value directly. By setting the faulty value to zero, we reduce all possible fault scenarios to a single fault scenario, one that can already be handled by classical ABFT.

We return to the simple example discussed in Section 3.3 to illustrate that classical ABFT can only guarantee a correction up to $O(10^{-16+\beta})$, and now show the effect of our modifications to ABFT. In dABFT, the faulty value $C = 1.1 \cdot 10^{14}$ is set to zero, which leads to the checksum value not being truncated and therefore the faulty value being correctly retrieved using the checksum value. Thus, $C = 1.23999$, having full accuracy after the correction.

Aside from the correction of bit-flips in the exponent, dABFT can also handle the special floating-point values NaN and infinity in C^f implicitly. The conditions for I_1 and I_2 in lines 6 and 7 of Algorithm 2 can be implemented to be true automatically for any of these special values, due to the definition of the comparison operators for these values in the IEEE-754 specification [34] (by reformulating the conditions to check whether the left side is "$\not\leq$" the right side). This strategy is useless when applied to classical ABFT since NaN and infinity already influence the computation of S_1 and S_2. Therefore, classical ABFT has to check for the existence of these special values in C^f explicitly before starting the fault detection process (see line 2 in Algorithm 1). dABFT can combine both steps through the proper implementation of the conditions and therefore reduces the overhead compared to classical ABFT. This is possible because S_1 and S_2 are recomputed after setting all faulty values (including NaN and infinity) to zero (see lines 12 and 13 in Algorithm 2). The performance increase due to this advantage is shown in Section 5.3, where we conduct experiments for both variants of ABFT.

The performance of classical ABFT and dABFT are dominated by the matrix multiplication of order $O\left(n^3 + n^2 d\right)$ in line 1 of both algorithms. The detection and correction steps of classical ABFT and lines 2-7 in Algorithm 2 for dABFT are of order $O\left(n^2 d\right)$, an order of magnitude lower than the dominating operation (as d is normally much smaller than n). The additional detection and correction steps in dABFT (lines 9-13) are only of order $O\left(nd^2\right)$ because they only recompute the parts of the factors S_1 and S_2 which used the faulty values in lines 4 and 5. Moreover, as already mentioned, dABFT does not require the explicit checking

Algorithm 2 Direct ABFT for Matrix Multiplication (dABFT)

Input: $A^c \in \mathbb{R}^{(n+d)\times n}, B^r \in \mathbb{R}^{n\times(n+d)}, W \in \mathbb{R}^{n\times d}$
Output: $C^f \in \mathbb{R}^{(n+d)\times(n+d)}$

1: Unreliable $C^f = A^c B^r$
2: $C_1 = W^\top C^f(1:n, 1:n+d)$
3: $C_2 = C^f(1:n+d, 1:n)W$
4: $S_1 = C_1 - C^f(n+1:n+d, :)$
5: $S_2 = C_2 - C^f(:, n+1:n+d)$
6: $I_1 = \{j \in [1,n] | \max_{k_r=[1,d]} \frac{|S_1(k_r,j)|}{\|W^\top(k_r,:)\|_p \|A\|_p \|B(:,j)\|_p} > 2\,(2+\mu_n)\,\mu_n\} \bigcup$
$\{k_c \in [1,d] | \max_{k_r=[1,d]} \frac{|S_1(k_r,j)|}{\|W^\top(k_r,:)\|_p \|A\|_p \|B\|_p \|W(:,k_c)\|_p} > 2\mu_n(3+3\mu_n+\mu_n^2)\}$ ▷ see (8) and (10)
7: $I_2 = \{i \in [1,n] | \max_{k_c=[1,d]} \frac{|S_2(i,k_c)|}{\|A(i,:)\|_p \|B\|_p \|W(:,k_c)\|_p} > 2\,(2+\mu_n)\,\mu_n\} \bigcup$
$\{k_r \in [1,d] | \max_{k_c=[1,d]} \frac{|S_2(i,k_c)|}{\|W^\top(k_r,:)\|_p \|A\|_p \|B\|_p \|W(:,k_c)\|_p} > 2\mu_n(3+3\mu_n+\mu_n^2)\}$ ▷ see (7) and (9)
8: **if** $I_1 \neq \{\}$ **and** $I_2 \neq \{\}$ **then**
9: $C^f(I_2, I_1) = 0$
10: $C_1(:, I_1) = W^\top C^f(:, I_1)$
11: $C_2(I_2, :) = C^f(I_2, :)W$
12: $S_1(:, I_1) = C_1 - C^f(n+1:n+d, I_1)$
13: $S_2(I_2, :) = C_2 - C^f(I_2, n+1:n+d)$
14: Solve $H(:, I_2) \cdot C^f(I_2, I_1) = -S_1(:, I_1)$ for $C^f(I_2, I_1)$
15: **end if**

for NaN and infinity (see line 2 in Algorithm 1), which saves $O\left(n^2\right)$ comparison operations. Overall, the performance benefits from the implicit handling of the special floating-point values.

Compared to classical ABFT, dABFT does not introduce additional numerical errors but actually improves the numerical accuracy. Any numerical floating-point errors caused by lines 9-13 in Algorithm 2 are guaranteed to be lower than in lines 2-7 due to the faulty values being removed from the matrix. The values of S_1 and S_2 are overwritten with the new values in lines 12 and 13 and are therefore more accurate than the ones computed with the faulty values in lines 4 and 5.

As mentioned in Section 3.3 (cf. temporal limitations), faults are also allowed to occur in lines 2 and 3 of Algorithm 2, as long as they occur before the value of C^f has been checked for faults. In contrast to classical ABFT, dABFT can also handle the case of a fault changing a value to NaN, as the value would subsequently be set to zero during the detection phase and not influence the computation of the least squares problem in line 14.

5. Experimental Evaluation of dABFT

In this section, we experimentally demonstrate the fault resilience of dABFT. All experiments were run on a single core of an AMD Opteron Processor 6174 and OpenBLAS (version 0.2.14) was used as the optimised BLAS library. For all experiments, the checksum encoding matrix W is generated using random, uniformly distributed values between 0 and 1. Faults are injected randomly using our approach summarised in Section 5.1. The resilience of classical ABFT and dABFT is compared in Section 5.2 and the low overhead of both ABFT variants is shown in Section 5.3.

5.1. Fault Injection

For the experiments in Section 5.2, we designed our own method for injecting faults randomly in time and space. We use an additional thread per process that has access to all data structures where the user allows bit-flips to occur. The time interval between two fault injections is exponentially distributed with

a specified mean time to failure (MTTF) per byte. The position at which a fault is injected is uniformly distributed, ensuring that all bits have the same chance of being flipped. Furthermore, the injection range can be specified to cover any bit of a floating-point number or can be restricted to the mantissa, the exponent or the sign bit. For the comparison of classical ABFT and dABFT, the specification of the injection range is very important. Due to the injector threads running asynchronously, the injections can also take place in library calls, e. g. BLAS `dgemm`, without any source code modifications or recompilation.

By design, our thread-based injection approach is very close to a realistic setup where faults can occur at any time during the execution. In order to limit the influence of the fault injection on the computation as much as possible, the fault injector thread runs asynchronously to the main algorithm and does not halt the main thread during the modification of the value. A bit-flip in the matrix is injected into the register of the core and then automatically synchronised to all other memory hierarchies. The injector thread reads a floating-point value from the main memory into a local memory variable, modifies the value by flipping a bit and then writes the modified value back to the main memory. In very rare situations, this can lead to a race-condition with the main thread, which may be operating on the same value selected by the fault injector. The main thread would then write the correct result from the multiplication back to the main memory, where it would subsequently be overwritten by the fault injector thread with the original value containing a single bit-flip. However, from the point of view of the main thread, the value would have been exposed to more than one bit-flip even though the fault injector only changed a single bit. Nevertheless, for the ABFT algorithms it is irrelevant how many faults occur in the same value. The faults would still only be detected as a single faulty value.

Our fault injector, has a very low overhead and hardly influences the performance results of the tested algorithms. On average, the overhead is only about 1% of the total execution time, independent of the number of cores used. This compares favourably to other published fault injection strategies, e. g. FlipIt [35], where significantly higher overhead has been reported (slow-down factors up to 123).

5.2. Resilience of ABFT Variants

In this section, we compare dABFT as presented in Section 4 to classical ABFT in terms of resilience. It suffices to demonstrate the resilience properties of the ABFT variants on a single core, because the 2.5D algorithms presented in Section 6 only use ABFT locally.

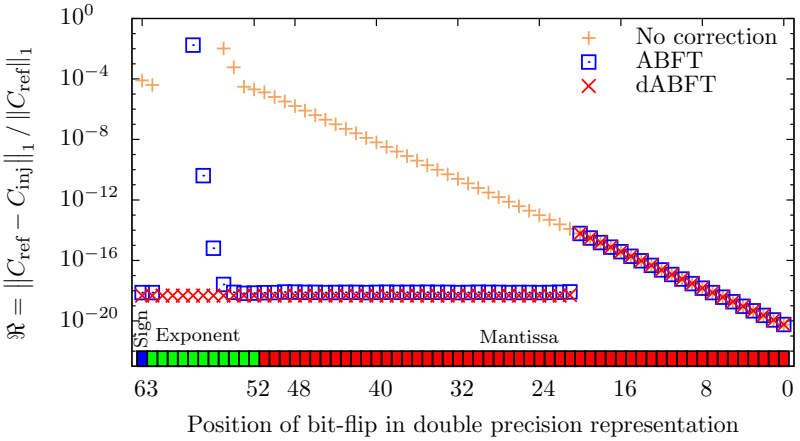


Figure 1: Error of the matrix product $C = AB$ after correcting a single bit-flip for different positions of the bit-flip.

We begin with injecting a single bit-flip deterministically into the element (1,1) of the result of the matrix multiplication $C = AB$. Subsequently, classical ABFT and dABFT recover from the fault. Figure 1 shows an average over 100 experiments with the position of the bit-flip on the x-axis and the resulting relative error $\Re := \|C_{\text{ref}} - C_{\text{inj}}\|_1 \,/\, \|C_{\text{ref}}\|_1$ on the y-axis, where the result matrix without any bit-flips is denoted by C_{ref} and the result matrix after injecting and correcting a bit-flip by C_{inj}.

As one can see in Figure 1, when flipping one of the least significant bits of the mantissa, the error increases with the position of the bit-flip for all methods. The effects of these faults are so small that they are below the thresholds of the fault detection conditions (7)-(10). Starting with the 21^{st} bit of the mantissa, the conditions are satisfied and both methods actually correct the injected bit-flip. The quality of the correction of classical ABFT and dABFT are equal up to the 4^{th} bit of the exponent. Then, the error of classical ABFT increases, in the worst case, exponentially. For a bit-flip in the 7^{th} to 10^{th} bit of the exponent all digits of the result produced by classical ABFT are incorrect. This is due to the fact that it tries to correct the faulty data by subtracting a correction value. This correction value can only be correct up to 16 digits due to the limited numerical representation of double precision. For the 11^{th} bit in the exponent, the result of classical ABFT is again correct because in all our experiments the bit-flip *decreased* the value (the 11^{th} bit of the exponent was originally always 1). Consequently, for classical ABFT, the quality of the correction strongly depends on the location of the bit-flip. In contrast, dABFT sets the faulty value to zero and computes the correct value directly, which results in an accurate result in all cases, regardless of the position of the bit-flip. In these experiments, only a bit-flip in the most significant bit of the exponent decreases the value of the floating-point number. In general, a bit-flip in the exponent can either decrease or increase the value (depending on whether the flipped bit is 1 or 0) and therefore it can cause small as well as large faults. Thus, the effect of the fault does not only depend on the position of the flipped bit but also on the floating-point number in which the bit-flip occurred (for details see [36]).

The next step is an exhaustive investigation of the resilience of both ABFT methods against any location of bit-flips occurring during the matrix multiplication using our non-deterministic fault injection method described in Section 5.1. We ran more than 12 000 matrix multiplications with problem size $n = 1000$ for each ABFT variant. The experiments covered different numbers of checksums $d \in \{1, 3, 5, 10, 15, 20, 25, 50, 100\}$ and different MTTFs, always ensuring that at most d faults were injected. Figure 2 illustrates the achieved correction quality by a cumulative distribution of the resulting relative errors. For both ABFT variants, we use exactly the same set of random injection points in time and position to make the results as comparable as possible. However, due to the asynchronicity of the fault injection thread it cannot be guaranteed that different individual runs are identical. Nevertheless, the huge number of experiments leads to a high confidence in the comparison of the two methods.

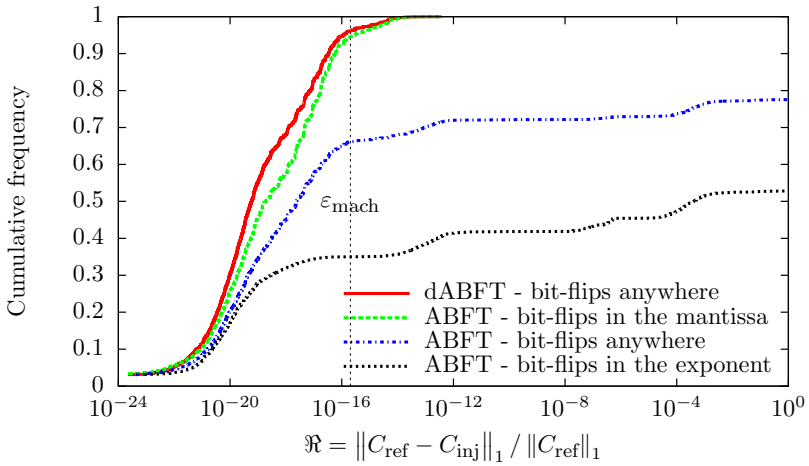


Figure 2: Cumulative distribution of $\Re$ for classical ABFT and dABFT with $n = 1000$. "bit-flips anywhere" means that bit-flips were injected in all bits of the floating-point numbers, including the sign bit. These results include single as well as multiple bit-flips per matrix multiplication.

We have seen in Figure 1 that the quality of classical ABFT strongly depends on the position of the bit-flip. Therefore, the results for the accuracy of classical ABFT in Figure 2 are split into three groups: faults in any bit of the floating-point number, faults only in the mantissa or faults only in the exponent. Focusing on the fault detection condition, Figure 2 shows the error of dABFT being less than 10^{-13} in all

tested cases. In comparison, for classical ABFT this is only true for bit-flips occurring in the mantissa. If faults are also injected into the exponent, the error can become unacceptably large. For example, a bit-flip in the exponent can result in the faulty value being smaller than the original number, a case which can also be handled by classical ABFT. However, in the worst case, if a bit-flip increases the affected matrix element, classical ABFT can reduce the error by a factor of $O(10^{-16})$ and will fail to produce a correct result, as discussed in Section 3.3 (cf. spatial limitations). For bit-flips at any position in the floating-point number, the result of classical ABFT was incorrect in all digits in more than 20% of all tested cases (see Figure 2). Therefore, in a realistic system, where bit-flips can occur in any bit, the classical ABFT method cannot be used for fault tolerant matrix multiplications. The step-like pattern in Figure 2 for classical ABFT (e. g., at 10^{-13}) illustrates that the error varies strongly due to the influence of different bit-flips in the exponent. From one bit to the next, a single fault can potentially double the exponent of the error. Moreover, as discussed at the end of Section 4, dABFT handles bit-flips in the mantissa numerically more accurately than classical ABFT because setting the faulty value to zero basically reduces the number of floating-point operations (an addition or multiplication in floating-point arithmetic with one operand being zero always has an error of zero). However, this effect can only really be seen if the residual is smaller than the machine epsilon ε_{mach}.

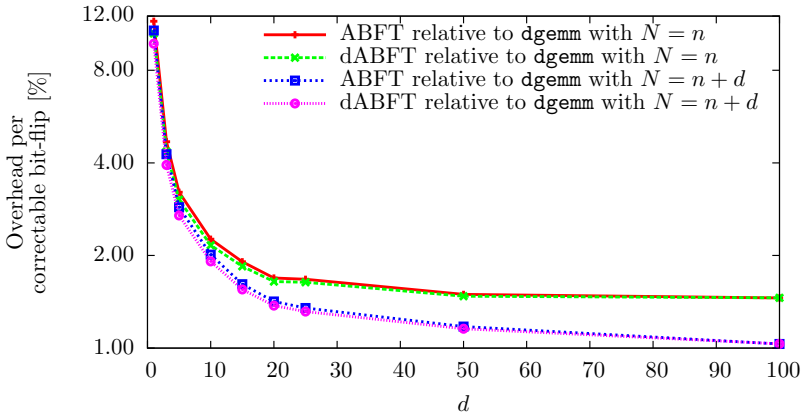
5.3. Runtime Performance of ABFT Variants

In the previous section, we demonstrated the superiority of dABFT over classical ABFT in terms of resilience. Now, we compare both methods in terms of runtime performance. In Figure 3, the runtime overhead of classical ABFT and dABFT is compared to a standard (non-fault tolerant) matrix multiplication of two $N \times N$ matrices (using `dgemm` from OpenBLAS). On the x-axis, the number of checksums d per row or column is shown. The overhead per correctable bit-flip is depicted on the y-axis. The first set of lines refer to the overhead caused by ABFT compared to `dgemm` with the original matrix dimension $N = n$. The second set of lines shows the overhead of the correction and detection steps of ABFT by comparing it to `dgemm` with $N = n + d$, the same size as the augmented matrices A^c and B^r. In this comparison, the overhead caused by the larger matrices is ignored and only the additional steps of ABFT are considered.

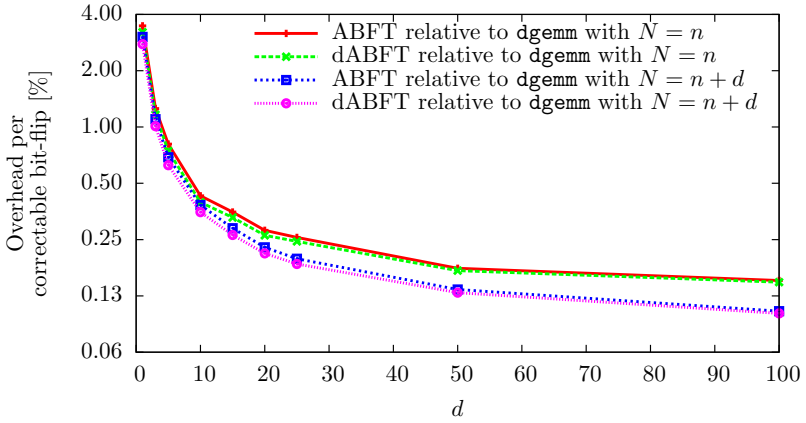
dABFT is always faster than classical ABFT due to the fact that it does not need to check for NaNs or infinity explicitly, as explained in Section 4. For $n = 1000$ (Figure 3a), the overhead per correctable bit-flip decreases in all cases and reaches a value of about 1.45% for the entire overhead ($N = n$) and 1.03% for the detection and correction process ($N = n + d$). About half of the overhead is caused by the increased matrix size and the other half by the detection and correction process. A more detailed analysis shows that the correction itself is only responsible for up to 10% of the total overhead because the least squares problems to be solved are at most of size d.

Figure 3b shows the same analysis of the overhead for larger matrices with $n = 5000$. As expected, the larger matrix size significantly decreases the overhead due to the impact of the number of checksums being smaller relative to the matrix size. In this case, the entire overhead for classical ABFT and dABFT is only about 0.15%. For the detection and correction process it is only about 0.1%, again about half of the total overhead. These results highlight the extremely low overhead, especially for large matrices and high fault rates.

The total overhead for all correctable bit-flips (shown in Figure 4) naturally increases with an increasing failure rate for a fixed n, but the overhead does not grow as fast as the number of checksums. Furthermore, for the same failure rates, the overhead caused by the increase in checksums decreases with the increase of the matrix size. For dABFT and $n = 1000$ (see Figure 4a), the total overhead is up to 7.23 times lower than the increase of d from 1 to 100, with 10.5% for $d = 1$ and 145% for $d = 100$. Increasing the matrix size to $n = 5000$ (shown in Figure 4b), the total overhead is even up to 21.73 times lower than the increase of d, with 3.22% for $d = 1$ and 14.8% for $d = 100$. This is already a significant decrease compared to $n = 1000$ and for the same fault rates the overhead will decrease even more for larger matrices.



(a) Overhead for $n = 1000$



(b) Overhead for $n = 5000$

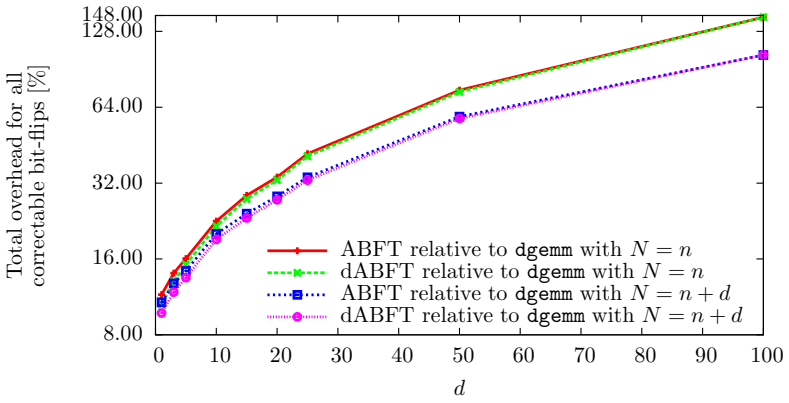
Figure 3: Overhead per correctable bit-flip of ABFT and dABFT relative to BLAS `dgemm`.

6. Fault Tolerant 2.5D Matrix Multiplication

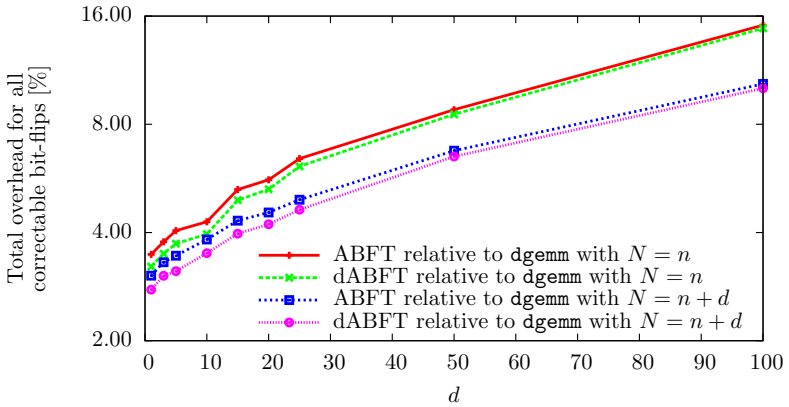
In this section, we present and discuss the *2.5D FTMM* algorithm, which results from integrating ABFT into 2.5D MM. As discussed in Section 2, 2.5D Cannon [17] is restricted to square matrices, whereas 2.5D SUMMA [18] can handle rectangular grids and matrices. For simplicity, we demonstrate our approach for the 2.5D SUMMA algorithm. However, all modifications can also be applied to 2.5D Cannon.

6.1. 2.5D Matrix Multiplication

For the 2.5D MM we define a three dimensional process grid $\Pi \in \mathbb{N}^{\sqrt{q/c} \times \sqrt{q/c} \times c}$, where q is the number of processes and c the size of the third dimension. In 2.5D SUMMA (see Algorithm 3), c is the number of copies of the input matrices A and B that are distributed along the third dimension of the grid in the first step of the algorithm. Initially, process $\Pi(i, j, 1)$ stores blocks $A_{i,j} := A((i-1)b+1 : ib, (j-1)b+1 : jb)$ and $B_{i,j} := B((i-1)b+1 : ib, (j-1)b+1 : jb)$, where $b = n/\sqrt{q/c}$ is the local block size. First, these blocks are broadcast along the third dimension. Then, iteratively, along each column of $\Pi(:, :, \kappa)$ one block of A and along each row of $\Pi(:, :, \kappa)$ one block of B is broadcast. The received blocks A_{local} and B_{local} are multiplied locally and added to C_{local}. Finally, the local results are summed along the third dimension $\Pi(i, j, :)$, so that $\Pi(i, j, 1)$ receives the sum $C_{i,j}$, which corresponds to the block of the result matrix C.



(a) Total overhead for $n = 1000$



(b) Total overhead for $n = 5000$

Figure 4: Total overhead for all correctable bit-flips of ABFT and dABFT relative to BLAS `dgemm`.

6.2. Combining dABFT and 2.5D SUMMA

In 2.5D SUMMA, the local matrix multiplication in line 6 of Algorithm 3 can be replaced by dABFT to allow bit-flips to be detected and corrected in the parallel algorithm without requiring any additional messages. To achieve this, the checksums are not added to the global matrix but to each local block A^c_{local} and B^r_{local}. This also leads to the detection and correction process being performed locally on each node. Therefore, no additional communication is incurred, thus preserving the communication-avoiding properties of the 2.5D algorithms. Naturally, the local computation time is slightly increased due to the augmented system size, but the overhead is negligible for a small number of checksums d. Slightly more data has to be sent due to the additional rows and columns in the augmented blocks A^c_{local} and B^r_{local}. For small d, the additional data is very small in relation to the protected data and becomes negligible for growing matrix dimensions.

Applying the checksums to local blocks instead of the global matrix also improves the numerical properties of the fault detection. Rexford and Jha [30] showed that, depending on the choice of the encoding vectors, numerical problems become more severe with the size of the data. They therefore proposed the partitioned encoding scheme to reduce the numerical inaccuracies occurring in the computation and comparison of the checksums. Naturally, from a global perspective, more memory is required for adding $2d \cdot n/b$ elements to the local matrix block. However, each block of the matrix is protected by d checksums which also allows more faults to be tolerated compared to the global approach. If the same fault rate is considered for both approaches, faults are expected less often in smaller amounts of data. Therefore, the number of

Algorithm 3 2.5D SUMMA

Input: $A \in \mathbb{R}^{n \times n}, B \in \mathbb{R}^{n \times n}$
Output: $C \in \mathbb{R}^{n \times n}$

1: **for each** process $(i, j, \kappa) \in \Pi$ **do**
2: Broadcast $A_{i,j}$, $B_{i,j}$ to $\Pi(i, j, :)$
3: **for** $t = (\kappa - 1)\sqrt{q/c^3} + 1 : \kappa\sqrt{q/c^3}$ **do**
4: A_{local}: Broadcast $A_{t,j}$ to $\Pi(i, :, \kappa)$
5: B_{local}: Broadcast $B_{i,t}$ to $\Pi(:, j, \kappa)$
6: $C_{\text{local}} = C_{\text{local}} + A_{\text{local}} B_{\text{local}}$
7: **end for**
8: $C_{i,j} \leftarrow$ Sum-reduction of C_{local} over $\Pi(i, j, :)$
9: **end for**

Algorithm 4 Fault-tolerant 2.5D matrix multiplication (2.5D FTMM)

Input: $A \in \mathbb{R}^{n \times n}, B \in \mathbb{R}^{n \times n}, W \in \mathbb{R}^{b \times d}$, frequency F
Output: $C \in \mathbb{R}^{n \times n}$

1: **for each** process $(i, j, \kappa) \in \Pi$ **do**
2: Broadcast $A^c_{i,j}$, $B^r_{i,j}$ to $\Pi(i, j, :)$
3: **for** $t = (\kappa - 1)\sqrt{q/c^3} + 1 : \kappa\sqrt{q/c^3}$ **do**
4: A^c_{local}: Broadcast $A^c_{t,j}$ to $\Pi(i, :, \kappa)$
5: B^r_{local}: Broadcast $B^r_{i,t}$ to $\Pi(:, j, \kappa)$
6: **if** $(t - (\kappa - 1)\sqrt{q/c^3}) \bmod F = 0$ **then**
7: $C^f_{\text{local}} = C^f_{\text{local}} + dABFT(A^c_{\text{local}}, B^r_{\text{local}})$
8: **else**
9: $C^f_{\text{local}} = C^f_{\text{local}} + A^c_{\text{local}} B^r_{\text{local}}$
10: **end if**
11: **end for**
12: $C_{i,j} \leftarrow$ Sum-reduction of C_{local} over $\Pi(i, j, :)$
13: dABFT fault detection and correction in $C^f_{i,j}$
14: **end for**

checksums per block can be reduced, which decreases the overhead per block. The authors in [30] also showed that all matrix operations discussed in [13] preserve the partitioned checksum property. Therefore, existing ABFT algorithms can directly benefit from the use of block-level checksums without having to be modified. The partitioned encoding scheme improves the upper bound on the round-off error by a factor of $O(\left\|W^{(n)}\right\|_2 / \left\|W^{(b)}\right\|_2)$, where $W^{(n)} \in \mathbb{R}^{n \times d}$ is the encoding matrix for global checksums and $W^{(b)} \in \mathbb{R}^{b \times d}$ for block-level checksums. The ability to detect faults is also improved, because the maximum tolerated error is decreased by this factor. Consequently, the blocked encoding scheme does not increase the number of false positives and at the same time allows for more faults at lower magnitudes to be detected than the global approach.

Our fault-tolerant 2.5D matrix multiplication (2.5D FTMM) is shown in Algorithm 4. The data distribution is the same as described in Section 6.1 for 2.5D SUMMA. The only difference is that each local block of A and B is augmented by the block-level checksums. The extended blocks are broadcast and used for the local ABFT matrix multiplication (lines 6-10 in Algorithm 4). Furthermore, the resulting local block of C is also extended by full checksums (rows and columns) at the block-level. Replacing the local matrix multiplication by an ABFT-based matrix multiplication ensures that bit-flips occurring during the multiplication can be corrected. A final detection and correction step consisting of lines 2-15 in Algorithm 2 has to be inserted at the end (line 13 in Algorithm 4) to also cover faults that occur during the reduction process.

As mentioned in Section 3.3 (cf. temporal limitations), the detection and correction process in classical

ABFT has to run fault free because it runs at the end of the computation and the returned result matrix C^f has to be guaranteed to be correct. In contrast, in 2.5D FTMM this constraint can be lifted for the multiple local matrix multiplications which are performed throughout the method. Therefore, except for the final detection and correction process in line 13, 2.5D FTMM is *completely fault tolerant*. There are no further restrictions on the time when or place where faults can be tolerated. This includes all detection and correction steps taking place during the local matrix multiplications. Although the final detection and correction step in line 13 of Algorithm 4 has to be reliable, this additional step has very little influence on the total execution time and, like all other ABFT steps, is performed locally at each node without any additional communication. As we will see in Section 7, this computation is very fast and therefore could be performed on reliable hardware at low cost.

Improvements for very low fault rates. The choice of the number of checksums d depends on the expected fault rate. For very low fault rates, 2.5D FTMM does not necessarily have to perform a detection and correction step after each local matrix multiplication. Very low fault rates would theoretically only require $d < 1$ checksum encoding vectors. Naturally, to protect the result from faults, d has to be at least 1, but the 2.5D matrix multiplication can be improved by not wasting valuable resources continuously searching for faults that are highly unlikely to occur in each consecutive local matrix multiplication. We introduce the parameter $F \geq 1$ which defines the frequency at which the ABFT steps are performed. For $F = \nu$ the ABFT steps are executed only after every ν^{th} local matrix multiplication. If F is larger than the total number of local matrix multiplications, only the last detection and correction process in line 13 would be performed. In Section 7, we will demonstrate the performance benefits that can be achieved for different values of F. As long as the number of faults during F subsequent local matrix multiplications does not exceed d, the accuracy of the result is not influenced at all by this parameter.

The best choice for the parameters d and F strongly depends on the system, the expected fault rate and whether 2.5D FTMM is combined with other fault tolerance techniques. This could include C/R, which is necessary for fail-stop errors and can also help to recover from faults that can be detected but not corrected by ABFT. However, in terms of floating-point operations, it is less efficient to increase d and F than to reduce both parameters. The number of floating-point operations in the detection and correction step is about $dn(n+d)/F$. If both parameters d and F are increased by a factor of ν, the number of operations is increased to $\nu dn(n+\nu d)/(\nu F) = dn(n+\nu d)/F$.

7. Experimental Evaluation of 2.5D FTMM

In this section, we present performance results for our 2.5D FTMM algorithm for matrices up to $n = 64\,000$ on 4096 cores. All experiments were run on the Vienna Scientific Cluster VSC-2 [1] consisting of 1314 nodes. Every node has two AMD Opteron 6132HE processors with eight cores each and 32 GB of main memory. The nodes are connected via QDR InfiniBand using a fat tree topology.

We compare the performance of 2.5D FTMM to state-of-the-art parallel matrix multiplication routines from DPLASMA [37] (version 2.0.0) and ScaLAPACK (version 2.0.0) and show the benefits of reducing the overhead of ABFT in the context of 2.5D MM. For all methods, MVAPICH2 (version 1.9) was used as the MPI library and OpenBLAS (version 0.2.14) as the optimised BLAS library.

In Figure 5, we compare our implementations of the *non-fault tolerant* 2.5D algorithms, 2.5D SUMMA and 2.5D Cannon, with the parallel matrix multiplication routines available in ScaLAPACK and DPLASMA for a matrix size $n = 50\,000$. To determine the optimal factor c of the 2.5D algorithms, we ran all possible values of c for each number of processes q and only report the results with the best performance. The local block size b is determined by q and c and was between 782 and 3125 (for details on c and b see Table 1). For ScaLAPACK we tested a wide range of block sizes and show the results with the best performance for each number of processor in Figure 5. For DPLASMA, we chose the parameters to be optimal for large processor counts and therefore do not show the runtimes of DPLASMA for smaller processor counts

[1] http://vsc.ac.at/systems/vsc-2/

in Figure 5. DPLASMA is executed using one process per node, with each node on the VSC-2 having 16 cores available. DPLASMA provides many different parameters to tune its routines for high performance, including various block sizes (tile, supertile and inner blocking), parameters for defining the process grid and the type and size of the high and low-level reduction trees. All variants of the reduction trees were tested using various tree sizes, but the best results were achieved using the default greedy tree settings. We also tested various block sizes and setting the tile size to 400 returned the best results. For all other block sizes, the default settings achieved the best performance on the VSC-2.

Our implementations of the 2.5D algorithms outperform both libraries, DPLASMA and ScaLAPACK. This demonstrates the high efficiency of our implementations of the communication-avoiding matrix multiplications. Furthermore, a lower bound on the runtime of triple modular redundancy (TMR) is also included in Figure 5. We use a fictitious TMR by assuming optimal scalability and absolutely no overhead. We set the runtime of this fictitious TMR with q processes equal to the runtime of the best 2.5D MM with $q/3$ processes. This guarantees a more than fair comparison with our approach by avoiding inefficient implementations of TMR or simply multiplying the runtime by three.

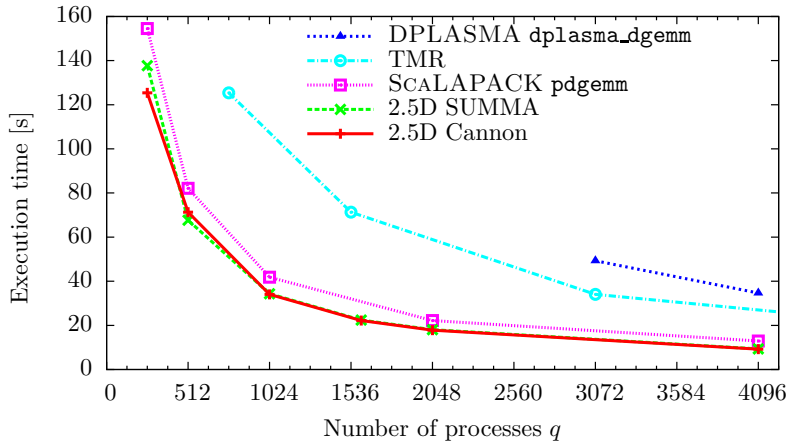


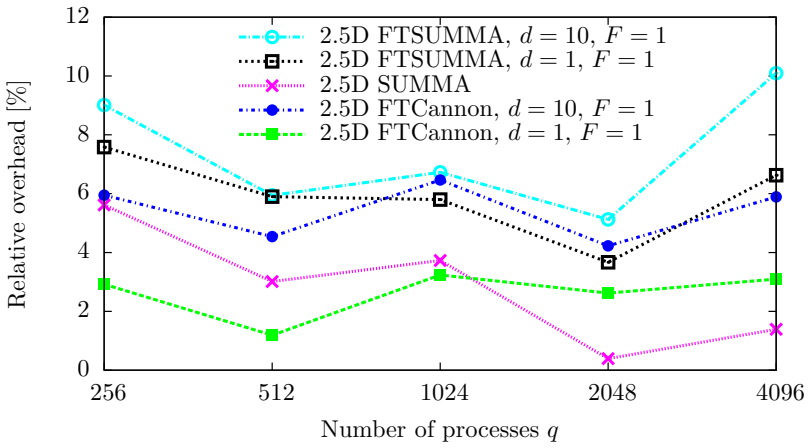
Figure 5: Runtime of our implementations of 2.5D SUMMA and 2.5D Cannon compared to state-of-the-art libraries and an optimal (fictitious) triple modular redundancy (TMR) for $n = 50\,000$.

Number of processes q	**Possible values for c**	**Best performance 2.5D Canon**	**Best performance 2.5D SUMMA**
256	1,4	$c = 1$, $b = 3125$	$c = 1$, $b = 3125$
512	2,8	$c = 2$, $b = 3125$	$c = 2$, $b = 3125$
1024	1,4	$c = 1$, $b = 1563$	$c = 4$, $b = 3125$
1600	1,4	$c = 1$, $b = 1250$	$c = 4$, $b = 2500$
2048	2,8	$c = 2$, $b = 1563$	$c = 8$, $b = 3125$
4096	1,4,16	$c = 1$, $b = 782$	$c = 4$, $b = 1563$

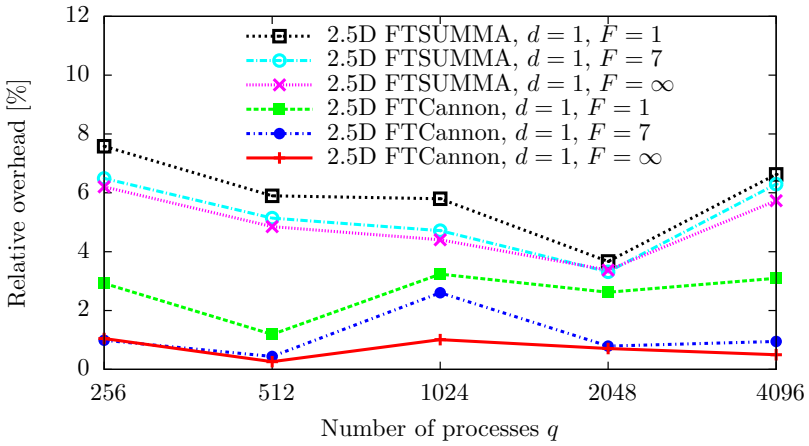
Table 1: All possible values of c for the given number of processes q and the values of c which achieved the best performance shown in Figure 5 for both 2.5D algorithms and $n = 50\,000$. Additionally, the corresponding block sizes b are also reported.

As discussed in Section 6.2, integrating dABFT into 2.5D MM incurs a small overhead due to the local operations becoming more expensive and slightly larger messages having to be sent. However, the number of messages stays the same. In Figure 6a, both 2.5D FTMM variants, 2.5D FTSUMMA and 2.5D FTCannon, are compared for different numbers of checksums d per local block. The overhead (in %) is shown relative to non-fault tolerant 2.5D Cannon, since this was identified as the non-fault tolerant reference implementation with the highest performance in Figure 5. For all methods, the matrix dimension n increases with the

number of processes q, from $n = 8\,000$ for $q = 256$ to $n = 64\,000$ for $q = 4096$. The local block size b for the best performance of the 2.5D algorithms varies with q due to the parameter c. In Figure 6, b was between 1000 and 4000. In our experiments, 2.5D SUMMA is on average 4% slower than 2.5D Cannon. For $d = 1$, the overhead for protecting 2.5D Cannon against a single bit-flip per local block is about 3%. The overhead naturally increases with d and reaches about 7% for 2.5D FTCannon with $d = 10$ and is slightly higher for 2.5D FTSUMMA.



(a) Varying number of checksums d



(b) Varying frequencies F of fault detection and correction

Figure 6: Overhead relative to non-fault tolerant 2.5D Cannon for $n = 8\,000$ ($q = 256$) to $n = 64\,000$ ($q = 4096$).

As mentioned in Section 6.2, we can significantly reduce the overhead caused by ABFT in the context of the 2.5D algorithms by reducing the number of detection and correction steps during the computation. The resulting improvements are shown in Figure 6b for different fault detection and correction frequencies F (see Algorithm 4), again relative to 2.5D Cannon. The best results are achieved for $F = \infty$, where each local matrix multiplication does not check for faults and the detection and correction only takes place in the last step of 2.5D FTMM after the reduction operation. In this case, the overhead has been reduced to less than 1% for 2.5D FTCannon. As already discussed in Section 6.2, $F = \infty$ can only be used if the fault rate is very low. For $F = 7$, which checks for faults after every 7^{th} local matrix multiplication, the overhead still remains very low while providing resilience against higher fault rates. Even though Figure 6 shows 2.5D FTSUMMA having a higher overhead due to also including the overhead compared to 2.5D Cannon, similar

effects can be observed.

The last detection and correction step of 2.5D FTMM has to be performed fault-free to guarantee a correct result, as mentioned in Section 6.2. For this step, reliable hardware or other strategies like TMR could be used. However, in our experiments this single step costs on average only 0.5% of the entire execution time of the matrix multiplication and is only performed locally. Therefore, employing more expensive strategies for this step would not influence the total runtime significantly.

8. Conclusions

We illustrated that classical ABFT as described in the literature is not able to correct all possible bit-flips and therefore cannot guarantee a correct result. Bit-flips can only be corrected if they are restricted to the mantissa and some of the least significant bits of the exponent, an unrealistic limitation in real-world applications. We proposed a novel improved ABFT method for matrix multiplication called dABFT, which removes this constraint and can handle bit-flips at any position, a necessity for fault tolerant matrix multiplications on real-world systems. We also derived novel fault detection conditions which, for the first time in the literature, are suitable for multiple checksum encoding vectors. Aside from the increased resilience, the performance of ABFT has also been improved due to the efficient handling of the special floating-point values NaN and infinity. We experimentally confirmed that our new method handles completely random bit-flips using our non-deterministic fault injector and conducting a large number of experiments, covering a wide range of fault rates, where an accurate result was always returned. Furthermore, we showed that the relative overhead of classical ABFT and dABFT per correctable bit-flip decreases with the number of checksums d, which makes these approaches very efficient for increasing matrix sizes and on systems with high fault rates.

Based on dABFT, we investigated how to integrate ABFT concepts into state-of-the-art high performance matrix multiplication algorithms. We introduced the fault tolerant matrix multiplication 2.5D FTMM, which combines the fault tolerance properties of dABFT with the high performance of 2.5D algorithms. In 2.5D FTMM, faults are allowed to occur throughout the entire algorithm, only requiring the last step – the final detection and correction step after the reduction operation – to be computed fault-free to ensure a correct result, drastically reducing the temporal limitations imposed by ABFT. This means that the vast majority of the computation – every local matrix multiplication and all detection and correction steps up until the reduction operation – can be computed on unreliable hardware. Only a very small part of the algorithm, which only requires on average 0.5% of the total execution time, has to be made fault tolerant by other approaches. However, the cost of the fault protection of this part hardly influences the performance of the overall fault tolerant algorithm. Therefore, even methods which tend to be rather expensive in general, e. g. triple modular redundancy, can be used for this part.

Additionally, for very low fault rates we were able to reduce the overhead of ABFT in the 2.5D algorithms by reducing the frequency of the detection and correction steps. Without any loss of accuracy, this reduced the overhead to less than 1% compared to a non-fault tolerant computation, a very small price to pay for making high performance matrix multiplication resilient against silent bit-flips.

Acknowledgments

This work has been partially funded by the Vienna Science and Technology Fund (WWTF) through project ICT15-113. The computational results presented have been achieved using the Vienna Scientific Cluster (VSC) [2].

[2] http://vsc.ac.at/

References

[1] M. Snir, R. W. Wisniewski, J. A. Abraham, S. V. Adve, S. Bagchi, P. Balaji, J. Belak, P. Bose, F. Cappello, B. Carlson, A. A. Chien, P. Coteus, N. A. Debardeleben, P. C. Diniz, C. Engelmann, M. Erez, S. Fazzari, A. Geist, R. Gupta, F. Johnson, S. Krishnamoorthy, S. Leyffer, D. Liberty, S. Mitra, T. Munson, R. Schreiber, J. Stearley, and E. V. Hensbergen, "Addressing Failures in Exascale Computing," *Int. J. High Perform. C.*, vol. 28, no. 2, pp. 129–173, 2014.

[2] F. Cappello, A. Geist, W. D. Gropp, S. Kale, B. Kramer, and M. Snir, "Toward Exascale Resilience: 2014 Update," *Supercomputing Frontiers and Innovations*, vol. 1, pp. 1–28, 2014.

[3] B. Schroeder, E. Pinheiro, and W.-D. Weber, "DRAM Errors in the Wild: A Large-scale Field Study," *SIGMETRICS Perform. Eval. Rev.*, vol. 37, no. 1, pp. 193–204, 2009.

[4] I. S. Haque and V. S. Pande, "Hard Data on Soft Errors: A Large-Scale Assessment of Real-World Error Rates in GPGPU," in *Proc. CCGRID*, 2010, pp. 691–696.

[5] J. C. Sancho, F. Petrini, K. Davis, R. Gioiosa, and S. Jiang, "Current Practice and a Direction Forward in Checkpoint/Restart Implementations for Fault Tolerance," in *Proc. IEEE IPDPS*, 2005, CD-ROM.

[6] C. Wang, F. Mueller, C. Engelmann, and S. Scott, "Hybrid Checkpointing for MPI Jobs in HPC Environments," in *ICPADS*, 2010, pp. 524–533.

[7] G. Zheng, X. Ni, and L. Kale, "A Scalable Double In-memory Checkpoint and Restart Scheme Towards Exascale," in *IEEE/IFIP 42nd Int. Conf. DSN-W*, 2012, pp. 1–6.

[8] D. Fiala, F. Mueller, C. Engelmann, R. Riesen, K. Ferreira, and R. Brightwell, "Detection and Correction of Silent Data Corruption for Large-scale High-performance Computing," in *Proc. Int. Conf. SC*, 2012, pp. 78:1–78:12.

[9] K. Ferreira, J. Stearley, J. H. Laros, III, R. Oldfield, K. Pedretti, R. Brightwell, R. Riesen, P. G. Bridges, and D. Arnold, "Evaluating the Viability of Process Replication Reliability for Exascale Systems," in *Proc. Int. Conf. SC*, 2011, pp. 44:1–44:12.

[10] R. Lyons and W. Vanderkulk, "The Use of Triple-Modular Redundancy to Improve Computer Reliability," *IBM J. Res. Dev.*, vol. 6, no. 2, pp. 200–209, 1962.

[11] J. Elliott, K. Kharbas, D. Fiala, F. Mueller, K. Ferreira, and C. Engelmann, "Combining Partial Redundancy and Checkpointing for HPC," in *IEEE 32nd ICDCS*, 2012, pp. 615–626.

[12] H. Casanova, Y. Robert, F. Vivien, and D. Zaidouni, "On the Impact of Process Replication on Executions of Large-scale Parallel Applications with Coordinated Checkpointing," *Future Gener. Comp. Sy.*, vol. 51, pp. 7–19, 2015.

[13] K. Huang and J. Abraham, "Algorithm-Based Fault Tolerance for Matrix Operations," *IEEE T. Comput.*, vol. C-33, no. 6, pp. 518–528, 1984.

[14] P. Du, P. Luszczek, and J. Dongarra, "High Performance Dense Linear System Solver with Soft Error Resilience," in *Proc. IEEE CLUSTER*, 2011, pp. 272–280.

[15] D. Hakkarinen, P. Wu, and Z. Chen, "Fail-Stop Failure Algorithm-Based Fault Tolerance for Cholesky Decomposition," *IEEE T. Parall. Distr.*, vol. 26, no. 5, pp. 1323–1335, 2014.

[16] Y. Jia, G. Bosilca, P. Luszczek, and J. Dongarra, "Parallel Reduction to Hessenberg Form with Algorithm-based Fault Tolerance," in *Proc. Int. Conf. SC*, 2013, pp. 88:1–88:11.

[17] E. Solomonik and J. Demmel, "Communication-optimal Parallel 2.5D Matrix Multiplication and LU Factorization Algorithms," in *Proc. Euro-Par*, 2011, pp. 90–109.

[18] E. Georganas, J. González-Domínguez, E. Solomonik, Y. Zheng, J. Touriño, and K. Yelick, "Communication Avoiding and Overlapping for Numerical Linear Algebra," in *Proc. Int. Conf. SC*, 2012, pp. 1–11.

[19] L. E. Cannon, "A Cellular Computer to Implement the Kalman Filter Algorithm," Ph.D. dissertation, Montana State University, 1969.

[20] R. A. Van De Geijn and J. Watts, "SUMMA: Scalable Universal Matrix Multiplication Algorithm," *Concurrency: Practice and Experience*, vol. 9, no. 4, pp. 255–274, 1997.

[21] M. Schatz, J. Poulson, and R. van de Geijn, "Parallel Matrix Multiplication: 2D and 3D," The University of Texas at Austin, Dep. of Computer Sciences, Technical Report TR-12-13, 2012.

[22] J.-N. Quintin, K. Hasanov, and A. Lastovetsky, "Hierarchical Parallel Matrix Multiplication on Large-Scale Distributed Memory Platforms," in *Proc. ICPP*, 2013, pp. 754–762.

[23] J. Demmel, D. Eliahu, A. Fox, S. Kamil, B. Lipshitz, O. Schwartz, and O. Spillinger, "Communication-optimal Parallel Recursive Rectangular Matrix Multiplication," in *Proc. IEEE IPDPS*, 2013, pp. 261–272.

[24] W. Bland, A. Bouteiller, T. Herault, G. Bosilca, and J. Dongarra, "Post-failure recovery of MPI communication capability: Design and rationale," *Int. J. High Perform. C.*, vol. 27, no. 3, pp. 244–254, 2013.

[25] G. Bosilca, R. Delmas, J. Dongarra, and J. Langou, "Algorithm-based Fault Tolerance Applied to High Performance Computing," *J. Parallel Distr. Com.*, vol. 69, no. 4, pp. 410–416, 2009.

[26] V. Nair and J. Abraham, "General Linear Codes for Fault-tolerant Matrix Operations on Processor Arrays," in *18th Int. Symp. Fault-Tolerant Computing*, 1988, pp. 180–185.

[27] J. Rexford and N. Jha, "Algorithm-based Fault Tolerance for Floating-point Operations in Massively Parallel Systems," in *Proc. IEEE ISCAS*, vol. 2, 1992, pp. 649–652.

[28] S. Dutt and F. Assaad, "Mantissa-preserving Operations and Robust Algorithm Based Fault Tolerance for Matrix Computations," *IEEE T. Comput.*, vol. 45, no. 4, pp. 408–424, 1996.

[29] G. Bosilca, A. Bouteiller, T. Herault, Y. Robert, and J. Dongarra, "Assessing the Impact of ABFT and Checkpoint Composite Strategies," in *Proc. IEEE IPDPS Workshops*, 2014, pp. 679–688.

[30] J. Rexford and N. K. Jha, "Partitioned Encoding Schemes for Algorithm-based Fault Tolerance in Massively Parallel Systems," *IEEE T. Parall. Distr.*, vol. 5, no. 6, pp. 649–653, 1994.

[31] J.-Y. Jou and J. Abraham, "Fault-tolerant Matrix Arithmetic and Signal Processing on Highly Concurrent Computing Structures," *Proc. IEEE*, vol. 74, no. 5, pp. 732–741, 1986.
[32] P. Wu, C. Ding, L. Chen, F. Gao, T. Davies, C. Karlsson, and Z. Chen, "Fault Tolerant Matrix-matrix Multiplication: Correcting Soft Errors On-line," in *Proc. 2nd ScalA Workshop*, 2011, pp. 25–28.
[33] N. J. Higham, *Accuracy and Stability of Numerical Algorithms*, 2nd ed. SIAM, 2002.
[34] *IEEE Standard for Floating-Point Arithmetic*, 2008, IEEE Std 754-2008.
[35] J. Calhoun, L. Olson, and M. Snir, "FlipIt: An LLVM Based Fault Injector for HPC," in *Proc. Euro-Par*, 2014, pp. 547–558.
[36] J. Elliott, M. Hoemmen, and F. Mueller, "Exploiting Data Representation for Fault Tolerance," *J. Comput. Science*, vol. 14, pp. 51–60, 2016.
[37] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, H. Haidar, T. Herault, J. Kurzak, J. Langou, P. Lemariner, H. Ltaief, P. Luszczek, A. YarKhan, and J. Dongarra, "Distributed Dense Numerical Linear Algebra Algorithms on Massively Parallel Architectures: DPLASMA," University of Tennessee, Tech. Rep. UT-CS-10-660, 2010.