A comparison of several fault-tolerance methods for the detection and correction of floating-point errors in matrix-matrix multiplication

Valentin Le Fèvre, Thomas Herault, Julien Langou, Yves Robert

A comparison of several fault-tolerance methods for the detection and correction of floating-point errors in matrix-matrix multiplication

Valentin Le Fèvre, Thomas Herault, Julien Langou and Yves Robert

ROMA team, Inria Grenoble Rhône-Alpes, France
LIP laboratory, ENS Lyon, France
Innovative Computing Laboratory, Knoxville, TN, USA
University of Colorado Denver, CO, USA

August 2020

Abstract

This paper compares several fault-tolerance methods for the detection and correction of floating-point errors in matrix-matrix multiplication. These methods include replication, triplication, Algorithm-Based Fault Tolerance (ABFT) and residual checking (RC). Error correction for ABFT can be achieved either by solving a small-size linear system of equations, or by recomputing corrupted coefficients. We show that both approaches can be used for RC. We provide a synthetic presentation of all methods before discussing their pros and cons. We have implemented all these methods with calls to optimized BLAS routines, and we provide performance data for a wide range of failure rates and matrix sizes.

1 Introduction

Reliable computing has become a key challenge when deploying applications on large-scale platforms. These platforms are confronted to many errors striking during execution. These errors are due to the extremely large number of floating-point operations executed by the parallel applications that are deployed on such platforms. Indeed, the probability of facing a corrupted floating-point operation is proportional to the number of such operations that are executed [8]. Even if each processor exhibits a low individual error rate, the probability of several errors striking during the execution of the parallel application becomes very high with millions of cores running in parallel for a few days, or even hours.

There are very few ways to ensure that a whole application has executed without error. The only general-purpose method is to replicate the execution

and to compare the results of both executions. If they do not coincide, an error has been detected, and the application must be executed a third time. To avoid a-posteriori re-execution, triplication can be enforced, which allows for error correction in addition to error detection, using a simple majority vote. However, triplication is even more costly than replication, which already requires half the resources to execute redundant operations. Fortunately, many scientific applications heavily rely on scientific kernels from numerical linear libraries, and much of their floating-point operations are executed within these kernels. For most linear algebra kernels, application-specific methods have been devised for error detection and correction, with a much lower cost than replication. The most prominent application-specific approaches are Algorithm-Based Fault Tolerance (ABFT) and Residual Checking (RC), which we describe in full details in Section 2. Both ABFT and RC are known to enable error detection, but ABFT has received much more attention because it is also deployed for error correction. In theory, ABFT can correct up to k errors with $2k+1$ checksums [17, 16, 13]. However, the numerical instability of floating-point ABFT currently limits its usage to correct one or two errors within a kernel.

In this paper, we revisit the Residual Checking (RC) approach, and show that it can be an efficient alternative to ABFT for error detection and correction. In particular, we focus on providing a transparent hardened version of some operation: the API, as exposed to the user, does not change, but the result is checked (and corrected if needed) before it is returned to the user. This creates a problem for ABFT, as the efficiency of the technique lies in mixing the user data and the redundant data used for failure detection and correction (see Section 2.2). RC can be implemented without modifying the API of the original computation kernel (see Section 2.3), which is a key advantage from a software engineering perspective. Another drawback of ABFT compared to RC is the lack of flexibility. By construction, ABFT uses a fixed number of checksums chosen a priori, say $2k + 1$, and will fail if more errors than k errors strike during the kernel. On the contrary, RC adapts the number of verifications on the fly, as a function of the number of errors found.

We adopt a somewhat narrow focus and only deal with protecting matrix-matrix multiplication from floating-point errors. Matrix-matrix multiplication is the archetypal linear kernel and is at the heart of several linear solvers, hence it is one of the most important kernels to study. Assessing the efficiency of residual checking for matrix-matrix multiplication will lay the foundations for the study of a full dense linear algebra library. The major contributions of this paper are the following:

- A synthetic comparison of several fault-tolerance methods for error detection and correction in matrix-matrix multiplication, with novel approaches for RC;
- A publicly-available prototype implementation of all the methods, with calls to optimized BLAS kernels;
- A comparative assessment for a wide range of failure rates and matrix sizes.

2 Methods

2.1 Replication

The first approach to detect computational errors is also the only systemic approach that can apply to any algorithm: it consists in replicating computations, and checking that both executions produce the same result. In the context of mutable data, this also implies to work on a copy of the data to compute, in order to enable the replicated computation [12]. There are multiple ways to implement replication: the computations can be executed sequentially, one after the other, at any level of granularity, or in parallel. Ultimately, the replication process provides two copies of the output of the computation and these copies are compared bit-to-bit, to detect errors.

Any error detected can then be resolved with a voting process: more replicas are computed, and if (at least) two output results converge on a same result, this result is considered valid. The probability that two computation errors produce the same result is considered negligible, since errors are supposed to be independent and identically distributed random variables.

2.2 ABFT

ABFT is an approach introduced in [10], that leverages mathematical properties of the algorithm to introduce redundancy in the data and thus allows to detect, and sometimes locate and correct errors during a computation. Applied to the matrix-matrix multiplication of the $C \leftarrow AB$ as an example, where A is n-by-n and B is n-by-n, the main idea of ABFT is to extend the matrix on which the operation is applied with checksum vectors that are pre-computed before the matrix-matrix multiplication. This gives

$$A \text{ extended as } \begin{pmatrix} A \\ A_c \end{pmatrix} \text{ with } A_c = v^T A, \quad B \text{ extended as } \begin{pmatrix} B & B_r \end{pmatrix} \text{ with } B_r = Bw$$

where w and v are checksum generator vectors. Once A and B have been augmented, we perform the matrix multiplication $\begin{pmatrix} C & C^{(r)} \\ C^{(c)} & C^{(\alpha)} \end{pmatrix} \leftarrow \begin{pmatrix} A \\ A^{(c)} \end{pmatrix} \begin{pmatrix} B & B^{(r)} \end{pmatrix}$, and we see that we must have the following relations

$$C^{(r)} = Cw \quad \text{and} \quad C^{(c)} = v^T C \quad \text{and} \quad C^{(\alpha)} = v^T C w. \tag{1}$$

Therefore, a way to check that the entries of C have been correctly computed is to check that the equalities in Equation (1) hold. With this scheme, we can, for example, guarantee to detect any single error in C. (In other words, if no more than one entry of C is corrupted, then this scheme will detect the error.) Note that w and v does not have to be vectors, but they can also be block of vectors,

The whole realm of error correction codes (e.g. Reed Solomon error correction code) is now at our doorstep since for each row C_i of C, we have computed

C_i and its checksum with respect to w, $C_i w$, and so not only can we detect errors, but we can also locate and recover errors. Using Reed Solomon error correction code, for example, we can detect, locate, and recover k errors with $2k+1$ checksums (provided that we use an appropriate encoding block of vectors w). However, the Reed Solomon algorithm is notoriously unstable in finite precision arithmetic [6] and does not enable one to recover from many errors or to handle very long vectors.

For detection, in practice, one row checksum of the form $C_i w$ is often enough to detect errors in any row of C, C_i. We simply check whether $C_i w = C_i^{(r)}$. This check can fail if the error vector introduced in C is orthogonal to w. However, this is unlikely. Tolerance of the order of machine precision has to be added to the check. Indeed, we only intend to detect errors that are larger than the errors made by the round-off errors of the numerical computation. So we check, for example, that $\|C^{(r)} - Cw\|_2 \leq 10u\|A\|_{\text{fro}}\|B\|_{\text{fro}}\|w\|_{\text{fro}}$, where u is the machine roundoff and the number "10" is taken arbitrarily [9]. A standard way to locate errors is to use "*coordinate checkpointing*". So if the row checksum $C_i^{(r)}$ is not $C_i w$ and the column checksum $C_j^{(c)}$ is not $v^T C_j$ then we conclude that the entry c_{ij} is false. Once an error is located, we can either recover the c_{ij} through the redundancy introduced by the checksum and therefore solving a system of linear equations with unknown c_{ij}, this leads to the method ABFT-SOLVE, or we can, in the case of matrix-matrix multiplication, simply recompute the entry c_{ij} from the ith row of A and the jth row of B, this leads to the method ABFT-RECOMP.

One advantage of Reed Solomon is that it enables locating and correcting via checksum only on the rows or only the columns, while coordinate checkpointing would need both row and column checksums. For matrix-matrix multiplication, it is convenient to maintain both checksums, while for other linear algebra operations, this is not always natural. Now, how to choose v and w? In the case ABFT-SOLVE, Chen and Dongarra [5, 6] showed that taking random matrices enable to recover the solution with high probability during the linear solve to recover the corrupted entries. While less critical, it does seem a good idea to also take random vectors v and w for ABFT-RECOMP.

As for the overhead, we see that to encode and compute with k checksums with $k \ll n$ is $\mathcal{O}(n^3)$ flops, the cost to detect, locate and recover ℓ errors is $\mathcal{O}(n^2\ell)$ flops. Therefore the cost (in term of flops) of recovery is theoretically negligible compared to the cost of computation.

2.3 Residual Checking (RC)

A closely related method is RC, which exploits the fact that checking the correctness of the result of a computation is usually easier than computing it. In short, one more time using the $C \leftarrow AB$ matrix-matrix multiplication as an example, if one wants to check at low cost whether C is correctly computed, one can compute, on the one hand, Cw and, on the other hand, $A(Bw)$ and check whether these two vectors are similar. And, not surprisingly, the two methods ABFT and RC share similar characteristics: (1) Low cost, (2) if w is

in the nullspace of $C - AB$, the error matrix, then we will not detect the errors, however this is unlikely, etc. Hence RC is very similar to ABFT. Historically RC was introduced with "error detection" in mind only. So you would perform the computation, use RC to detect errors, and then redo the computation if any error is detected [14, 15]. RC has long be thought to only be able to detect errors, and not able to locate and correct errors. For example, Prata and Silva [14] writes: "*We left out of our comparison one aspect where ABFT would do better than RC, namely fault localization and error recovery, (RC has no such capability).*" Actually, in very much the same way as ABFT, RC is able to detect, locate and correct errors. The two methods (ABFT and RC) are essentially similar and have the same capabilities.

2.4 Differences between ABFT and RC

There is a fundamental principle difference between RC and ABFT. Given some input, an algorithm computes some output such that a relation is true. For example, given A, (1) LU factorization: compute P, L, and U such that $PA = LU$, (2) QR factorization: compute Q, R such that $A = QR$, (3) SVD decomposition: compute U, Σ, and V^T such that $A = U\Sigma V^T$. RC finds a quick way to check whether this final relation holds. For example, given a vector x, (1) check that $P(Ax) = L(Ux)$, (2) check that $Ax = Q(Rx)$, (3) check that $Ax = U(\Sigma(V^T x))$. If the relation does not hold, then RC has succeeded in detecting an error. If the relation holds, then RC has succeeded in assessing (with high probability) the correctness of the result.

On the contrary, ABFT starts with checksums on the initial data, and maintains the consistency of the checksums along the algorithm. So the checksums are being modified as the data is being modified so that current data is consistent with current checksum. As a side comment, the difference above explains why that it is easier to derive RC for many more algorithms than for ABFT. (In a few lines, we gave RC for three algorithms, and for ABFT, we barely explained how this concretely worked.) However, in the case of matrix-matrix multiplication and linear algebra in general, once RC and ABFT algorithms are implemented, the differences are not so clear any longer, and we find that the algorithms are often very close. We describe the design space as having three dimensions. These three dimensions are essentially orthogonal in the sense that it is possible to make choices in any dimension independently of the others.

Dimension 1: appending checksums or leaving checksums separate. The checksums (for example A_c) can either (case `1ab`) be appended to the main matrix (e.g. as extra rows to A) or (case `1rc`) left as separate independent blocks of vectors. On the one hand, for RC, the checksums are naturally separate from the matrices. On the other hand, ABFT has been presented with both possibilities. RC is always `1rc`. ABFT can be `1ab` (e.g., [2, 10]) or `1rc` (e.g., [18]).

One advantage of leaving the checksums separate from the matrices is to not change the data structures of the original (non fault-tolerant) code. This is much

easier to accomplish from a software engineering point of view. One advantage of appending the checksum is to call kernels only once (on the extended data structure). The computation on the checksums is then processed at the same time as the computation on the main matrix. This can be much faster.

Dimension 2: computing checksums on input data before computation or after. If we compute the initial checksums before the matrix-matrix multiplication, we call this `2ab`. If we compute the initial checksums after the matrix-matrix multiplication, we call this `2rc`. The main distinction between `2ab` and `2rc` is not really when we compute checksums, but more whether we "can" recompute initial checksums after the main operation. Recomputing the initial checksums after the computation means that we are storing the input data, and we are not overwriting in the initial data with computation. In Numerical Linear Algebra, this is a significant constraints since we often have one operand that is in/out. If we perform `2rc`, we must use backup (copy) of all in-out operands.

It seems that, in the literature, ABFT always compute the initial checksums before the computation. If one wants to append the checksums to the matrix, then one will in general compute the checksums before the computation. Therefore, often, `1ab` $\Rightarrow$ `2ab`. (And its contrapositive: `2rc` $\Rightarrow$ `1rc`: if we compute the checksums after, then the checksums will be separate.) One advantage to compute the checksums after is to compute as many initial checksums as needed by the number of errors, which is useful to lower the overhead, and to avoid making any assumption on the maximum number of errors that will be encountered.

Dimension 3: detect+recompute or detect+locate+lazy-recompute or detect+locate+solve. Case `3rc`: detect errors, and recompute the whole computation if some errors are detected `3rc`. Case `3lo`: detect errors, locate errors and recompute only the corrupted entries (also called *lazy recomputation* in [18].) Case `3ab`: detect errors, locate errors and recover the corrupted entries from the redundant information in the checksum, we call this `3ab`. A long-held misconception is that computing initial checksum after does not enable to recover corrupted entries from the checksum. In other words the misconception is `2rc` $\Rightarrow$ `3rc`. As already explained this is false.

For `3lo` and `3ab`, in this paper, the localization is done through "coordinate checkpointing". `3lo` assumes that entries can be recomputed somewhat easily from only the input data, and maybe some non-corrupted entries. It is not obvious that there are many kernels for which this is possible. Matrix-matrix multiplications is one such kernel. For `3ab`, assuming that we can locate the errors, (through coordinate checkpointing, for example,) Chen and Dongarra [5, 6] showed that taking random matrices enable to recover the solution with high probability during the linear solve to recover the corrupted entries.

Reed-Solomon encoding enables `3ab` with either a row checksum or a column checksum, it does not require both row and column checksum. This is very useful for some operations. (Not matrix-matrix mutiplication though.) However the

checksum block of vectors v and w are extremely ill-conditioned and leads to numerically unstable codes. We note that `2ab` +`3ab` is the only way (in this design space) to overwrite in/out operands during the computation and recover from errors. All other methods needs to copy and store in/out operands to extra memory space to be able to recompute from the input in case an error occurs.

Which dimension distinguishes ABFT vs RC. Dimension 1: we can distinguish ABFT and RC by defining ABFT as appending checksums to matrices, and RC as having checksum separate from matrices. Dimension 2: we can distinguish ABFT and RC by defining ABFT as computing the initial checksums before computation, and RC as computing the initial checksums after computation. Dimension 3: we can distinguish ABFT and RC by defining RC as detecting and maybe locating errors, and following a detection by recomputation, and defining ABFT as recovering the corrupted entries, after detection and location, from the redundant information contained in the checksum.

3 Related work

Multitudinous papers have been published on replication, ABFT and RC. A surveys on ABFT is provided in [3]. Due to lack of space, we refer to the extended version [11] for a more comprehensive overview. We have selected below a small set of closely related works, which we classify in Table 1 according to the criteria given in Section 2.

Reference	`1ab`	`2ab`	`3ab`	`1rc`	`2rc`	`3rc`	`3lo`
[10]	✓	✓	✓				
[14]				✓	✓	✓	
[7]				✓	✓	✓	
[4]*	✓	✓	✓				
[2]*	✓	✓	✓				
[1]			✓	✓	✓		
[18]		✓		✓			✓

*errors are "failures" and therefore the detection and localization of the error is known

Table 1: Taxonomy of related work

4 Experiments

4.1 Implementations

We implemented variants of all the techniques discussed above. The implementation is in C, relying on the BLAS kernels for all linear algebra operations (namely GEMM and GEMV), and each hardened routine provides the same API as the GEMM routine defined by BLAS, but implements a different error detection and correction strategy. Here is the list of the six routines that we implemented, and that we compare in Section 4.3:

• NoFT is a reference point, and is a direct call to the GEMM routine provided by the BLAS library, without any error checking nor correction strategy.

• Replication uses the most simple (and systematic approach): replication, as described in Section 2.1: the GEMM operation is computed twice, then resulting elements are compared one by one, and if an error is detected, the entire

operation is computed a third time. Elements are then selected by a simple majority vote, and if no majority can be obtained for some element, the operation is applied again, until a pair of matching results can be found.
• ABFT-SOLVE (=`1ab` +`2ab` +`3ab`) is the traditional ABFT method: the input matrices are copied into larger matrices, that are extensions of the inputs with a fixed number of column and row checksums. These checksums are computed from the initial data, and the GEMM operation is applied on the extended matrix. After it completes, we check the checksums to detect errors. If errors are detected, a linear system of equations is solved [2, 4, 17, 16, 13] to compute the corrected values, and the resulting matrix is copied in the output parameter.
• ABFT-RECOMP (=`1ab` +`2ab` +`3lo`) follows the same strategy as ABFT-SOLVE to detect errors, but the matrix is extended with a single column and row as checksums. By crossing the columns in which the row-checksum is incorrect and the rows in which the column-checksum is incorrect, we extract a number of suspected wrong results, and we recompute only these elements from the input data. The result is checked (iterating another step of re-computation if needed), and copied back into the output parameter.
• RC-SOLVE (=`1rc` +`2rc` +`3ab`) uses the RC to compute the checksums (see Section 2.3): the GEMM operation is computed, and once it is computed, a single column checksum is generated randomly, and the routine compares how applying the output of GEMM on it differs from applying the two input matrices. If the result differs in any element, there is at least an error on the corresponding row(s). Additional checksums are then generated, until a system of linearly independent equations can be formed. That system is solved to correct the errors.
• RC-RECOMP (=`1rc` +`2rc` +`3lo`) uses the same approach as RC-SOLVE, until the correction phase is reached. When this is the case (there is at least one row with errors), a row-checksum is computed (as the column checksum was), and by crossing the row-checksum errors and the column-checksum errors, we can approximately locate suspected error locations. These elements of the output matrix are recomputed from the initial data to patch the result matrix which is returned by the routine.

4.2 Setup

For introducing errors in the operations, we use a parameter r which is the error rate of one floating-point operation. We compute the probability for an element to be erroneous, knowing it is the result of m operations: $P = 1 - (1 - r)^m$ and we modify each element that has been drawn to be corrupted by multiplying the element by a factor randomly chosen between 0.5 and 1.5, after doing the computation. We first apply this modification on all the elements of the matrix after the GEMM operation, with $m = 2n - 1$, because there are n multiplications and $n - 1$ additions per element when multiplying square matrices of size n. Then, for the recomputed elements of RC-RECOMP and ABFT-RECOMP implementations, we set $m = 2n - 1$ for each element that is recomputed from scratch and we check again the result. For RC-SOLVE and ABFT-SOLVE, $m = \mathtt{c}^2$ where `c`

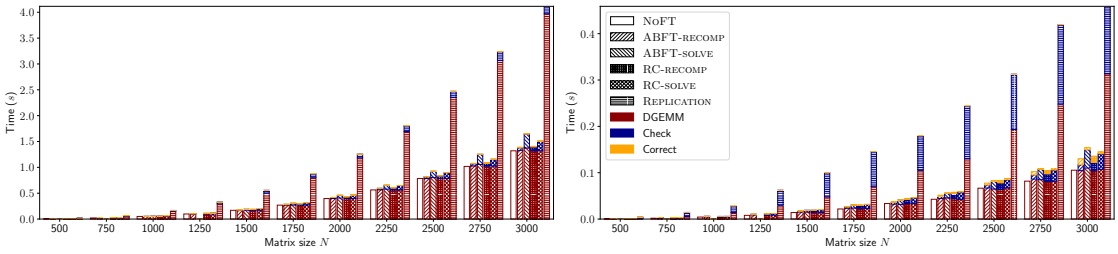

Figure 1: Sequential (left) and multi-threaded (right) algorithms, error rate of 10^{-9}.

is the number of corrupted columns in the matrix. Finally for Replication, $m = 2n-1$ for each element of every new matrix computed. In each experiment, the maximum duration of the hardened operation is bounded by 4 iterations of the applied check / correct procedure, and if the matrix is still corrupted at this point, the operation is considered failed. ABFT-solve needs one additional parameter which is the number of checksums to add to the matrix: we set it to $2 \times 2N^3r$ as $2N^3r$ is the expected number of failures during the computation and we want a margin to tolerate more errors in bad scenarios. If ABFT-solve cannot solve the system of equations, the operation is considered as failed.

We run the experiments with 16 cores out of a 20-core Intel Xeon CPU E5-2650 v3 at 2.30GHz, with 64GB of memory hosted at the University of Tennessee. The code is compiled with GCC 9.2.0, and the BLAS kernels where provided by Intel MKL version 2019.3.199. We evaluate both the sequential and multi-threaded versions of the algorithms. We run 100 iterations of each combination of implementations and parameters (the matrix size N and the error rate r) and we average the execution times of the different parts of the algorithm. *DGEMM* is the time spent doing the main operation (and subsequent DGEMMs for Replication); *Check* is the time spent computing the checksums and finding the location of the errors; *Correct* is the time spent recomputing or solving the systems depending on the chosen implementation. We report the execution times when each of the 100 iterations succeeds; otherwise, we report the number of failed iterations. As a reference, we show the time to execute a GEMM on a $N \times N$ matrix without fault tolerance nor failure injection under the name NoFT. The source code of the implementations used for the experiments is available at `https://github.com/vlefevre/abft-rescheck`.

4.3 Results

Figure 1 describes the detailed execution of our 6 implementations for an error rate $r = 10^{-9}$ and a varying matrix size N. The first thing to notice is that replication is always the less efficient technique. Indeed, even without failures, two full DGEMM operations need to be executed to detect failures. Moreover, every time there is at least one error during the computation, we need to compute the resulting matrix a third times to correct it. It is enough to correct in most cases but the cost of a DGEMM operation, especially in sequential, is much bigger than the cost of a detection and the ensuing correction at this error

Implementation	ABFT-SOLVE							RC-SOLVE	
Error rate r	10^{-10}	10^{-9}				8×10^{-9}	10^{-8}	8×10^{-9}	10^{-8}
Matrix size N	3000	500	750	1000	1250	3000	3000	3000	3000
Sequential	4	2	23	0	7	1	3	11	78
Multi-threaded	3	2	24	4	3	0	4	15	81

Table 2: Number of failed iterations (over 100) for parameters used in Fig. 1–2.

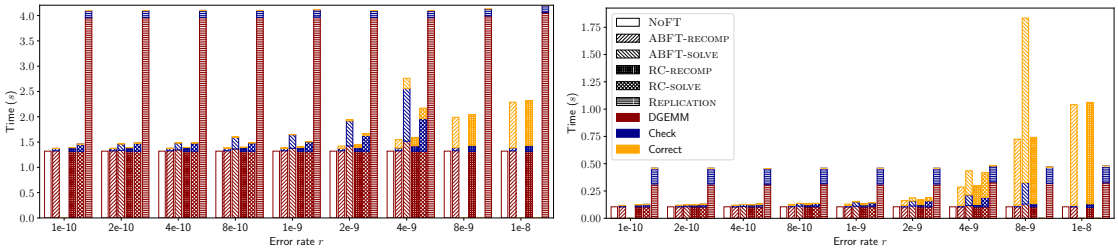

Figure 2: Sequential (left) and multi-threaded (right) algorithms, matrix size 3000.

rate.

The overheads of detecting and correcting errors for all methods but REPLICATION remain small, even when the matrix size (thus the number of errors) increases: there is only a small proportion of the output matrix that is corrupted, and thus the amount of recomputation or the size of the linear problem to solve to correct are small. Recomputation-based approaches, however, outperform significantly system-solving approaches.

The multi-threaded case shows the same characteristics overall, except the check time of REPLICATION is significantly increased, relative to the duration of the GEMMs. As checking for REPLICATION is a memory-bound problem, when all the cores access the memory simultaneously, the memory bus becomes the bottleneck and limit parallel efficiency.

When N increases, both RC-SOLVE and ABFT-SOLVE are likely not to correct everything within 4 re-executions as the correction is done by solving linear systems of size c, hence with $O(\mathsf{c}^3)$ flops, where c is the number of corrupted columns. For a given error rate, increasing N will increase both the number of columns and the probability that it is corrupted at the beginning. Thus the number of operations involved in the solve phase (c^2 compared to $2n-1$) can quickly grow and we need more iterations to finish. ABFT-SOLVE also does not always correct for small error rates or small matrix sizes (see Table 2). As the margin on the number of checksums to add is smaller, it becomes easy to have more errors than what we estimated even if we already added a factor 2 to the expected number of failed operations. This risk is managed by the RC-SOLVE implementation as the checksums are computed after failures hit the initial DGEMM operation, and thus the exact minimal number of checksums is used.

Figure 2 shows the same measurements, but with a fixed problem size ($N = 3000$) and a varying error rate. The Solve-based approaches do not produce

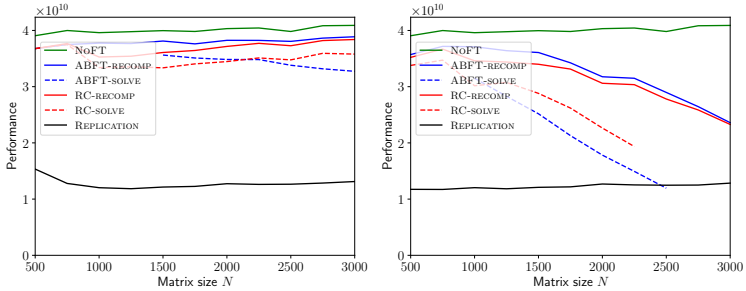

Figure 3: Overall performance of the 6 algorithms for $r = 10^{-9}$ (left) and $r = 10^{-8}$ (right).

results at 8×10^{-9} and 10^{-8} error rates in the sequential case, and ABFT-SOLVE only produce an output in a very long time in the multithreaded case with an error rate of 8×10^{-9}. As the number of columns including errors gets closer to N, the size of the system to solve becomes closer to the size of the original matrix. Since errors can also impact these computations, with a higher probability, the solve-based approaches fail, leading to repeated iterations of the correction process.

For low error rates, RC-RECOMP and ABFT-RECOMP are the two best performing algorithms and behave very similarly. The main difference between the two algorithms is that RC-RECOMP is easier to (1) set up since the check is done after the main computation and does not depend on the algorithm (for detection) and (2) to use as a blackbox for the user with no conversion of data needed. This last point is important as a user-friendly library would take as input $N \times N$ matrices and ABFT needs to add some extra steps to compute a bigger matrix with the checksums in it. This can quickly increase the execution time (and the memory footprint) of the algorithm if only a few DGEMM operations are done in a row because of the memory allocations and copies.

However, as the error rate increases, the recomputation-based approaches start to show slower corrections. This is particularly visible in the multithreaded case: REPLICATION eventually outperforms RC-RECOMP and ABFT-RECOMP. To explained this: first, REPLICATION's efficiency is independent from the error rate, because errors hit independent elements in the 3 computed matrices; second, as the number of errors in the matrix gets closer to N^2, the recomputation algorithm is less efficient than re-doing a fully optimized GEMM: it implements a parallel loop over the failed elements of sequential dot products. In the multi-threaded case, this is less efficient than recomputing the entire GEMM.

We sum up these results in Figure 3. We represent here the performance of the operations, as the ratio between $2N^3$ (the number of floating point operations in a GEMM) and the execution time of the sequential algorithms. It is clearly visible that the error rate has no influence on REPLICATION while ABFT-RECOMP and RC-RECOMP are the two best performing algorithms and their performance is equivalent. We also see that their performance stays close

to that of NoFT as long as both r and N do not become too big. See the extended version [11] for a similar figure where the matrix size is fixed and the error rate is varied.

5 Conclusion

In this paper, we have reviewed and compared ABFT and Residual Checking (RC) for detecting and correcting floating-point errors in matrix multiplication. On the theoretical side, we have detailed both methods, their variants, their common characteristics and their differences. On the practical side, we have implemented two variants for error correction in each method, one based on solving a small linear system, and one based on recomputing only corrupted elements, using coordinate checksumming to locate them. An extensive experimental comparison reveals similar execution times for the core of each method, but ABFT requires to embed the checksum in the user data in order to benefit from the high performance kernel implementation, while RC does not. Also, the flexibility of RC becomes very important when error rates are high, because RC can adapt a posteriori to the number of errors encountered within each particular execution. On the contrary, ABFT protection is constructed in a rigid way, with a fixed number of checksums which will rarely match the exact number of errors striking in a given run. This represents an acceptable overhead when the number of errors is smaller than expected, but it leads to the failing of the method when the number of errors is higher than the maximum number of errors that can be tolerated. To summarize, we point out that RC can be extended to correct silent errors in addition to detecting them, in a flexible and adaptive way, and without the burden of the extra memory allocation required by ABFT. Future work will be devoted to extending the approaches to other linear algebra kernels, and to protect from memory corruptions in addition to floating-point errors.

References

[1] Argyrides, C., Lisboa, C.A.L., Pradhan, D.K., Carro, L.: A fast error correction technique for matrix multiplication algorithms. In: 15th Int. On-Line Testing Symposium. pp. 133–137. IEEE (2009)

[2] Bosilca, G., Delmas, R., Dongarra, J., Langou, J.: Algorithm-based fault tolerance applied to high performance computing. J. Par. Dist. Comput. **69**, 410–416 (2009)

[3] Bouteiller, A., Herault, T., Bosilca, G., Du, P., Dongarra, J.J.: Algorithm-based fault tolerance for dense matrix factorizations, multiple failures and accuracy. ACM Trans. Parallel Comput. **1**(2), 10:1–10:28 (2015)

[4] Chen, Z., Dongarra, J.: Algorithm-based checkpoint-free fault tolerance for parallel matrix multiplications on volatile resources. In: Proc. IPDPS. IEEE (2006)

[5] Chen, Z., Dongarra, J.J.: Condition numbers of gaussian random matrices. SIAM J. Matrix Analysis Appli. **27**(3), 603–620 (2005)

[6] Chen, Z., Dongarra, J.J.: Numerically stable real number codes based on random matrices. In: ICCS 2005. LNCS vol 3514. Springer (2005)

[7] Gunnels, J., Katz, D., Quintana-Ortí, E., Van de Geijn, R.: Fault-tolerant high-performance matrix multiplication: Theory and practice. In: Proc. Dependable Systems and Networks (DSN). pp. 47–56 (2001)

[8] Herault, T., Robert, Y. (eds.): Fault-Tolerance Techniques for High-Performance Computing. Computer Communications and Networks, Springer Verlag (2015)

[9] Higham, N.J., Mary, T.: A new approach to probabilistic rounding error analysis. SIAM J. Scientific Computing **41**(5), A2815–A2835 (2019)

[10] Huang, K., Abraham, J.: Algorithm-based fault tolerance for matrix operations. IEEE Trans. Computers **33**, 518–528 (1984)

[11] Le Fèvre, V., Herault, T., Langou, J., Robert, Y.: A comparison of several fault-tolerance methods for the detection and correction of floating-point errors in matrix-matrix multiplication. Research report RR-9351, INRIA (June 2020)

[12] Lyons, R.E., Vanderkulk, W.: The use of triple-modular redundancy to improve computer reliability. IBM J. Res. Dev. **6**(2), 200–209 (1962)

[13] Plank, J.S.: A tutorial on Reed-Solomon coding for fault-tolerance in RAID-like systems. Software – Practice & Experience **27**(9), 995–1012 (1997)

[14] Prata, P., Silva, J.G.: Algorithm based fault tolerance versus result-checking for matrix computations. In: Digest of Papers. 29th Int. Symp. Fault-Tolerant Computing. pp. 4–11 (1999)

[15] Prata, P., Silva, J.G.: Fault-detection by result-checking for the eigenproblem. In: Dependable Computing — EDCC-3. Springer (1999)

[16] Reed, I.S., Solomon, G.: Polynomial codes over certain finite fields. J. oSociety for Industrial and Applied Mathematics **8**(2), 300–304 (1960)

[17] Roy-Chowdhury, A., Banerjee, P.: Algorithm-based fault location and recovery for matrix computations on multiprocessor systems. IEEE Trans. Comput. **45**(11) (1996)

[18] Smith, T.M., van de Geijn, R.A., Smelyanskiy, M., Quintana-Ortí, E.S.: Towards ABFT for BLIS GEMM. Tech. Rep. 76, FLAME Working Note (June 2015)