

# Algorithm Based Fault Tolerance Versus Result-Checking for Matrix Computations

Paula Prata

*Universidade da Beira Interior  
Dep. Matemática/Informática  
Rua Marquês d'Ávila e Bolama  
P- 6200 Covilhã, Portugal  
pprata@dei.uc.pt*

João Gabriel Silva

*Universidade de Coimbra/CISUC  
Dep. Eng. Informática  
Pinhal de Marrocos  
P-3030 Coimbra – Portugal  
jgabriel@dei.uc.pt*

## Abstract<sup>1</sup>

*Algorithm Based Fault Tolerance (ABFT) is the collective name of a set of techniques used to determine the correctness of some mathematical calculations. A less well known alternative is called Result Checking (RC) where, contrary to ABFT, results are checked without knowledge of the particular algorithm used to calculate them.*

*In this paper a comparison is made between the two using some practical implementations of matrix computations. The criteria are performance and memory overhead, ease of use and error coverage. For the latter extensive error injection experiments were made. To the best of our knowledge, this is the first time that RC is validated by fault injection.*

*We conclude that Result Checking has the important advantage of being independent of the underlying algorithm. It also has generally less performance overhead than ABFT, the two techniques being essentially equivalent in terms of error coverage.*

Keywords: Result-checking, ABFT, Fault injection, Error Detection, Matrix operations.

## 1. Introduction

Algorithm-Based Fault-Tolerance (ABFT) [1] [2] is a very effective error detection technique, applicable to various matrix computations that are central to many scientific and engineering programs. Essentially some additional calculations are added to (mostly linear) matrix transformations, enabling a check at the end that provides very high error coverage with low performance overhead. Besides, ABFT complements very well other low overhead error detection methods, like memory protection and control flow checking [3] [4].

A less well known set of techniques, collectively called Result-Checking (RC) [5], comprises theoretical work on developing low-redundancy checkers to verify the result of various kinds of mathematical functions, including most of those for which ABFT algorithms exist. RC has the attraction of being problem-based, that is, it does not depend on the algorithm used to solve a problem, but only on the mathematical function being calculated.

The idea behind this paper is to compare these two different approaches. We use three different criteria for that comparison.

First we examine the error coverage. Since there is no analytical way of calculating it under realistic fault models, we used our Xception fault injection tool [6] to determine that coverage experimentally. To the best of our knowledge, this is the first time that RC is validated by fault injection.

The second criterion is the overhead, both in performance and in memory usage.

Finally, we consider the ease of use, where essentially the ease of programming is taken into account.

We left out of our comparison one aspect where ABFT would do better than RC, namely fault localization and error recovery, studied extensively e.g. in [7] (RC has no such capability). We did that because fault localization and error recovery are available only for a subset of the ABFT algorithms, and because they are only possible in multiprocessor architectures like systolic arrays, where very good fault containment exists between the several processors involved in the computation. For the more general case where no such fault containment exists or is quite imperfect, these features of ABFT are of little use.

In this way our conclusions make no assumption about the underlying architecture.

The mathematical calculations that we used in our experiments were matrix multiplication and QR factorization, two widely used algorithms for which ABFT and RC techniques are available. A third case study is matrix inversion.

---

<sup>1</sup> This work was partially supported by the Portuguese Ministério da Ciência e Tecnologia, the European Union through the R&D Unit 326/94 (CISUC) and the project PRAXIS XXI 2/2.1/TIT/1625/95 (PARQUANTUM).

This small set of computations is quite representative, being nuclear in four standard problems of numerical linear algebra: linear systems of equations, least square problems, eigenvalue problems and singular value problems. These are the problems that constitute for instance the Lapack [9] and ScaLapack libraries.

To make our study as impartial as possible, for the ABFT mechanism applied to matrix multiplication and QR factorization we used the source code produced by Chowdhury [7][1], that implements ABFT assuming a single fault model. To study the RC mechanism for those cases, we have eliminated, from that code, the ABFT checkers, and introduced Freivalds' simple checker for matrix multiplication described in [8].

For matrix inversion (using LU decomposition) we used routines from the public domain library LAPACK [9], to illustrate the fact that a very important advantage of RC is its capability to use practical code. In fact, many of the "clean" mathematical algorithms used for the derivation of ABFT algorithms are seldom used in numerical analysis, for performance and stability reasons. Adapting ABFT algorithms to those practical routines is not a trivial task. For the LAPACK routines used in our test case, we only used RC, that can use them as a black box, and did not undertake the lengthy and complex task of deriving some ABFT variation for them.

The main conclusion is that the central advantage of RC is its independence of the underlying algorithm, being able to use real numerical algorithms directly. It also has generally less performance overhead than ABFT, the two techniques being essentially equivalent in terms of error coverage.

The structure of this paper is as follows: in sections 2 and 3 the concepts of ABFT and RC are reviewed and section 4 describes the fault injection process. Sections 5 and 6 describe the experimental evaluation for the problems of matrix multiplication and QR factorization respectively. Section 7 describes the results for matrix inversion and section 8 presents the conclusions.

## 2. Algorithm based fault tolerance

ABFT was first proposed as a checksum scheme for matrix operations [2]. The input matrices are augmented with an additional checksum row and an additional checksum column. Each element of the checksum column/row is the sum of the elements of the original matrix that are in the same row/column. The augmented matrices are then multiplied using an unchanged multiplication algorithm - in the end, the additional row and column of the result matrix should still be the sum of the elements of the same row or column. If that is not the case, some error occurred. The initial ABFT schemes have been proposed for systolic architectures, [2] [10],

and required additional processors to compute the checking operations. In those systems each processor calculates a bounded number of data elements, and the fault model considered assumes that if only one processor at a time fails (single fault assumption), then only a bounded number of output data items could be wrong.

Schemes for general-purpose multiprocessors have also been developed for several numerical algorithms like matrix multiplication, LU and QR decompositions, Fast Fourier Transforms and Iterative solvers for partial differential equations ([11] [1] [12]). These works have considered the following fault-model: The processor that reads in and distributes the data, collects and interprets the results, is assumed to be fault free. It is called the host. The processors that perform the computations (the nodes) are the same that execute the checking operations. Each processor checks the results from another node and not its own results. These nodes may be faulty and may corrupt all the data they possess. Assuming a single fault model, if a faulty processor produces wrong results, its checker processor will detect the fault. If an error only affects the execution of the test, it will signal as faulty the wrong node.

This model has some weak points: A fault may affect the data before the check columns and rows are calculated, or after the results have been verified, but before they are safely stored somewhere, probably in the host's disk. Besides, it may not be realistic to assume a fault free host. To overcome these problems we have recently proposed a technique that complements the fault tolerant capabilities of ABFT, protecting that part of the program that ABFT does not cover, namely input, output and the execution of the test itself. Since the method can be applied to any calculation that is checked by an assertion (like e.g. a Result Checker), we call the general method "Robust Assertions" [13]. We have considered that extension to ABFT in the experiments described in this paper.

Another important issue related to ABFT (and RC) is the rounding error inherent in all floating-point computations, which means that the test made to determine whether there was some error in an ABFT protected computation cannot be exact. A tolerance interval has to be calculated to account for that rounding error. This means that, if a fault produces an error whose magnitude falls inside the tolerance interval, that fault won't be detected, which is probably reasonable since it can be argued that it is not significant. Then the tolerance interval will establish the threshold for what is considered an error and what is not. There are many works on deriving rounding error upper bounds [14]; a simplified error analysis to calculate the tolerance interval at run-time for some ABFT techniques is proposed in [15].

### 3. Result-checking

It is often easier to check the result of a mathematical function than to compute it. Starting from this idea, some theoretical work has been done in the RC field [5]. The concept of a simple checker for a function  $f$ , is presented as [8]: “a program which, given inputs  $x$  and  $y$ , returns the correct answer to the question, does  $f(x)=y$ ?; The checker may be randomized, in which case, for any  $x$  and  $y$ , it must give the correct answer with high probability over its internal randomization. Moreover, the checker must take asymptotically less time than any possible program for computing  $f$ ”. The initial motivation of this work was to assure software-fault tolerance by checking the result, instead of testing the program in a set of random inputs or by a formal proof of correctness. There are checkers proposed for very different areas such as sorting [5], linear transformations [16] and algebraic expressions of trigonometric functions [17].

The result checker that we will experimentally evaluate is Freivalds’ simple checker for matrix multiplication described in [18] [8], that can be applied to the three examples used in this paper: matrix multiplication, QR factorization and matrix inversion. Suppose that we want to verify if a computed matrix  $C$  is the product of the given matrices  $A$  and  $B$ . To do that we can check if the product of  $C$  with a random vector  $r$  is equal to the product of  $A$  with  $B$  times  $r$ , ( $C \times r = A \times (B \times r)$ ). When  $C=A \times B$ , the condition is obviously verified; if  $C \neq A \times B$ , the probability that a random  $r$  verifies  $C \times r = A \times (B \times r)$  is very small. If we repeat the process for several values of  $r$  the probability that it detects an error can be made arbitrarily high. Thus we have an  $O(n^3)$  computation (where  $n$  is the matrix order) verified by an  $O(n^2)$  checker.

There remains the problem of verifying the checker. In [16] it is argued that: checkers are much simpler than the code they check, therefore less likely to have bugs. If the checker fails but the program is correct, there occurs a false alarm that alerts us to an error somewhere. The situation when a checker incorrectly accepts a wrong result is the worst one. The authors claim that correlated errors that lead to such a situation are very unlikely. As to the consequences of hardware faults (or software faults with similar effects), the protection of the checker can be done by the "Robust Assertions" method referred before.

Finally we should note that, like ABFT, RC also has the problem of rounding error.

### 4. Fault injection

We have evaluated our fault-detection mechanisms with a fault injection tool, called Xception [6], which emulates hardware faults at a very low level. Xception is a

software fault injection and environment monitoring tool that can inject transient faults in specific microprocessor functional units such as Floating-point ALU, Integer ALU, Data Bus, Address Bus, General Purpose Registers, Condition Code Register(s), Memory Management Unit and Memory. Each fault affects only one machine instruction and one functional unit at a time.

The system used in this paper is a Parsytec PowerXplorer with 4 PowerPC 601, each with 8 Mbytes of memory, running Parix, a parallel OS similar to UNIX, with a Sun SparcStation as the host. We have distributed the computations over three nodes, with the fourth processor acting as the host.

In all experiments reported in this paper, the faults injected were transient bit flips of one bit, randomly chosen, applied to a randomly selected functional unit in a randomly selected processor, time triggered (i.e. randomly distributed during the program execution time). The fault definition parameters have been exactly the same in all experiments, except for the duration of the program that was adapted for each case. Since our main purpose is to compare ABFT and RC, in this work we only have considered faults in the processing nodes. The "Robust Assertion" [13] technique would handle faults in the host. To classify the outputs we have done a component-wise comparison between the result matrix (which is outputted even when an error was detected) and a reference solution, obtained in a run with no injected fault. The outputs were classified as: *correct output*, *no output* -the program terminates but does not produce any result, *timeout* - no output because of a crash and *wrong output* - at least one matrix element is different from the reference value.

### 5. Matrix multiplication

Given two matrices  $A$  and  $B \in \mathbb{R}^{n \times n}$ , the usual matrix multiplication algorithm computes each entry  $c_{ij}$  of matrix  $C=A \times B$  as the dot product of  $A$ 's  $i$ th row and  $B$ 's  $j$ th column. In Chowdhury's algorithm [7] [1], the matrix  $A$  is distributed block-wise by rows and  $B$  is replicated over all processing nodes. Each processor  $P_i$  performs the sub-matrix multiplication  $C_i=A_i \times B$  using the sequential algorithm described above. At the end, each node sends the result sub-matrices back to the host, which concatenates them.

We have computed the product of two 270-by-270 double precision matrices with random entries chosen in the range (-100, +100). The error bound for that matrix-matrix product was approximately  $10^{-8}$  (this is a component-wise error bound obtained from the expression  $|C - C'| \leq \gamma_n |A| |B|$ , where  $C'$  is the computed matrix and  $\gamma_n = nu/(1-nu)$ , with  $n=270$  and  $u=2^{-53}$  [14]). Since this rounding error bound is only an approximation, we did not use it to distinguish just two intervals (below and

above), but instead considered two additional small intervals of the size of one order of magnitude, around that central value, to enable a more precise understanding of the results. The considered intervals for the absolute difference between two corresponding elements are then: Maximum difference less than  $10^{-9}$ , maximum difference between  $10^{-9}$  and  $10^{-8}$ , maximum difference between  $10^{-8}$  and  $10^{-7}$ , and maximum difference greater than  $10^{-7}$ . This last group contains the “clearly wrong results”.

Another problem that we have faced was that some faults cause Not a Numbers (NaNs) - special codes used in the IEEE floating point format that when compared with anything, including itself, always returns false. Since the original program did not handle them, we had to introduce some additional code to detect them. This acts as an additional error detection method.

### 5.1 Matrix multiplication with ABFT

We have used the source code available from the work of Chowdhury [7] [1] as described earlier. Because of ABFT, each node additionally calculates the checksum row of their part of the result matrix, and verifies the sub-matrix calculated by its neighbor, in a directed circular cycle. It should be noted that, probably for performance and memory usage reasons, Chowdhury chose not to verify the checksum column, which can have some impact on coverage. In the comparison of the checksum vectors we have used a fixed tolerance of  $10^{-8}$ . This was the minimum value for which we did not get false alarms in the data set used. This experimentally calculated tolerance has the same magnitude of the error bound estimated for the matrix multiplication.

The outcomes of the experiments are shown in table I, following the classification described above. As can be seen only 21 faults (that is approx. 0.2%), that have produced clearly wrong results, were not detected. The study of those cases has shown that 7 faults have been injected in a low-level routine of memory copy, which is called from the communication routines used by our program. In a previous study [13], we have concluded that if we protect the results with a CRC, before sending them to the host, those faults are detected. The other 14 faults have been injected in the line of the source code that performs the dot product of each A’s row with each B’s column. These faults were mainly injected in the integer unit, and the bit that has been modified was always in the less significant bits of the word. A manual analysis of one undetected wrong result has shown that it corresponds to a situation where two errors in the same column cancel each other. Errors like these could be detected if the checksum column of C would also be verified. We have approx. 2.8% of undetected small errors.

To improve the detection capabilities of ABFT we have implemented a new version of the program adding to

the algorithm the checker that verifies the checksum column of C, and protecting the results with a CRC after they were produced, as suggested in [13]. Table II presents the results for this enhanced version of matrix multiplication with ABFT. The percentage of undetected wrong results is now approx. 2.3%. But we have an important qualitative difference. All the clearly wrong results have been detected. This means that the probability of undetected “clearly wrong results” is very low for the fault/error model used. In section 5.3 we present an upper bound confidence limit for the probability of the undetected errors. As can also be seen from the table the number of wrong outputs decreases while the number of correct outputs grows. This is a natural consequence of adding test code.

Matrix mult. with ABFT (checksum row only)		no. of faults	% of total	Undetected	
				no.	%
Correct Output		3944	44.5	-	-
No output		4	0.1	-	-
Timeout		1959	22.1	-	-
Wrong outputs	NaNs	4	0.1	0	0.0
	diff. > $10^{-7}$	2605	29.4	21	0.2
	$10^{-8}$ < diff. $\leq 10^{-7}$	82	0.9	0	0.0
	$10^{-9}$ < diff. $\leq 10^{-8}$	66	0.7	52	0.6
	diff. $\leq 10^{-9}$	194	2.2	194	2.2
Total		8858	100.0	267	3.0

Table I - Results of fault injection for matrix multiplication with ABFT (with checksum row only).

Matrix mult. with ABFT (and CRC)		no. of faults	% of total	Undetected	
				no.	%
Correct Output		4029	47.7	-	-
No output		7	0.1	-	-
Timeout		1813	21.5	-	-
Wrong outputs	NaNs	6	0.1	0	0.0
	diff. > $10^{-7}$	2303	27.3	0	0.0
	$10^{-8}$ < diff. $\leq 10^{-7}$	60	0.7	0	0.0
	$10^{-9}$ < diff. $\leq 10^{-8}$	54	0.6	30	0.3
	diff. $\leq 10^{-9}$	169	2.0	168	2.0
Total		8441	100.0	198	2.3

Table II - Results for matrix multiplication with ABFT (version with row and column checksums and CRC)

### 5.2 Matrix multiplication with result-checking

To evaluate the RC mechanism for matrix multiplication we have used the initial program, without fault-detection mechanisms, and added Freivalds’ checker, described in section 3, to the code that is executed in the host. After reading the matrices A and B, the host processor also reads a random vector r (with values in the range (-1,1)) that was generated first. Then it computes the two matrix-vector products,  $B \times r$  and  $A \times (B \times r)$ . Matrices A and B are sent to the processing nodes and the

program follows as before. The host, after receiving the result matrix C, computes the vector C×r. Finally the component-wise comparison between the two check-vectors, A×(B×r) and C×r, is performed. The tolerance value was 10<sup>-8</sup>. Again it was the minimum value for which we did not get false alarms in the data set used. First we have implemented the checker using just one random vector. Table III shows the outcomes of the experiments.

As can be seen all the clearly wrong results have been detected. The value of undetected errors is slightly higher than in ABFT cases (approx. 3.1%), including some results with a maximum difference from the reference result, between 10<sup>-8</sup> and 10<sup>-7</sup>, (0.3%). We tried to improve this mechanism using two vectors, that is, we checked the result twice with different random vectors.

Table IV shows the outcomes of the experiments, with this second RC version. One random vector r1, with values in the range (-1,1), as the previous one, and a second vector r2, with values in the range (0,5). The undetected wrong results are now approx. 2.7% but there were 4 “clearly wrong results“ that were detected only with the second random vector. This shows that small variations in the random vector (we only changed the seed of vector r1 between the two experiments) can have some impact in the coverage of errors close to the rounding error.

Matrix mult. with RC (one random vector)		no. of faults	% of total	Undetected	
				no.	%
Correct Output		3921	44.2	-	-
No output		13	0.2	-	-
Timeout		1983	22.4	-	-
Wrong outputs	NaNs	11	0.1	0	0.0
	diff. > 10 <sup>-7</sup>	2600	29.3	0	0.0
	10 <sup>-8</sup> <diff. ≤10 <sup>-7</sup>	82	0.9	27	0.3
	10 <sup>-9</sup> <diff. ≤10 <sup>-8</sup>	69	0.8	59	0.7
diff. ≤10 <sup>-9</sup>		187	2.1	187	2.1
Total		8866	100.0	273	3.1

Table III - Results of fault injection for matrix multiplication with RC (using one random vector).

Matrix mult. with RC (two random vectors)		no. of faults	% of total	Undetected	
				no.	%
Correct Output		4027	45.17	-	-
No output		14	0.16	-	-
Timeout		1945	21.82	-	-
Wrong outputs	NaNs	5	0.06	0	0.0
	diff. >10 <sup>-7</sup>	2574	28.87	0	0.0
	10 <sup>-8</sup> <diff. ≤10 <sup>-7</sup>	67	0.75	2	0.02
	10 <sup>-9</sup> <diff. ≤10 <sup>-8</sup>	80	0.89	43	0.48
diff. ≤10 <sup>-9</sup>		203	2.28	198	2.22
Total		8915	100.0	243	2.72

Table IV - Results of fault injection for matrix multiplication with RC (using two random vectors)

### 5.3 ABFT Vs result-checking

The error coverage of both mechanisms for matrix multiplication looks very similar. The values of undetected wrong results are summarized in table V. We assume that the injected faults are representative faults that is, the selection probability of each injected fault is equal to its occurrence probability. Then  $f/n$  is an unbiased estimator for the non-coverage  $\bar{C}$  of each mechanism studied ( $n$  is the number of injected faults and  $f$  the number of undetected faults). An upper 100 $\gamma$ % confidence limit estimator for the non-coverage is given by 
$$\frac{(f+1)F_{2(f+1),2(n-f),\gamma}}{(n+f)+(f+1)F_{2(f+1),2(n-f),\gamma}} [19],$$
 where  $F_{\nu_1, \nu_2, \gamma}$  is the 100 $\gamma$ % percentile point of the  $F$  distribution with  $\nu_1, \nu_2$  degrees of freedom. The last column of table V presents the upper 99% confidence limit estimates for the non-coverage of each mechanism studied.

Matrix mult. – undetected wrong outputs					
mechanism	injected faults	diff> 10 <sup>-7</sup>	diff≤ 10 <sup>-7</sup>	$\bar{C}$	$\bar{C}_{99\%}^{\uparrow}$
ABFT row ch. only	8858	21	246	3.01%	3.27%
whole ABFT+CRC	<b>8441</b>	<b>0</b>	<b>198</b>	<b>2.35%</b>	<b>2.63%</b>
RC-1 rand. vect.	8866	0	273	3.08%	3.33%
RC-2 rand. vect.	<b>8915</b>	<b>0</b>	<b>243</b>	<b>2.72%</b>	<b>2.99%</b>

Table V – Undetected wrong outputs, ABFT Vs RC, for matrix multiplication.

In summary we can say that the final version of ABFT has a fault/error coverage greater than 97.37% and the final version of RC has a fault/error coverage greater than 97.01% with a confidence level of 99%.

ABFT has a slightly better coverage for small errors. This is explained by the fact of that, in the ABFT scheme, each partial test verifies a fraction of the data smaller than in the result-checking test. But then those are almost certainly non-significant errors. For both mechanisms the percentage of undetected small errors will depend on how accurate is the tolerance value used to compare the check vectors. After all we can consider that when a fault detection mechanism does not detect errors of the same magnitude as the rounding error this is not a “real failure” of the fault-detection mechanism. In that case the coverage lower bounds estimates became 99.95% for both mechanisms with a confidence level of 99%.

Comparing the execution time of both fault-detection mechanisms, both are checkers of  $O(n^2)$  for a calculation of  $O(n^3)$ . Then we can expect that the overhead will be negligible for large matrix sizes. But in the ABFT case when we add the test for the second matrix dimension, we are introducing additional communication. Moreover, with the first test (using only the checksum row) we are

taking advantage of the row distribution of matrix A, but this is not the case in the second test.

Table VI shows the calculation time overheads for some matrix dimensions. We have excluded the input/output operations from the time measured. We can see that ABFT is superior to RC only in the first version (the one where the second matrix dimension was not tested and the results were not protected). Comparing the versions that have equivalent fault coverage (whole ABFT with CRC and RC with two random vectors) result checking has a clear advantage.

Comparing the memory overheads, initially both checkers have an  $O(n)$  overhead. That corresponds to the check vectors. But, in the ABFT case, when we introduce the test for the second matrix dimension, we need to send to each node a copy of the block of A corresponding to the next processor. That means a duplication of the matrix A. In that case the memory overhead for ABFT is  $O(n^2/p)$ .

Finally, ABFT is much more complex to implement than RC. We have compared the number of lines of code for the unprotected initial program, and the final versions of both ABFT and RC. The overheads are 67% for RC and 104% for the ABFT version. But the main advantage of RC is that it can be used, without modification, with any algorithm of matrix multiplication. ABFT, by design, depends on the particular algorithm used.

Matrix multiplication – calculation overheads						
matrix dim.	base time: second	ABFT-row check.	whole ABFT	whole ABFT+ CRC	RC - 1 rand. vector	RC - 2 rand. vectors
210	2.80	1.4%	14.3%	<b>18.2%</b>	2.5%	<b>4.6%</b>
300	6.78	1.2%	11.9%	<b>14.9%</b>	2.2%	<b>3.7%</b>
360	10.8	0.9%	10.8%	<b>13.1%</b>	2.0%	<b>3.5%</b>

Table VI – Calculation time overheads, ABFT Vs RC, for matrix multiplication.

## 6. QR factorization

The problem of QR factorization is to decompose an initial matrix A into an upper triangular matrix R and an orthogonal matrix Q, i. e,  $A=QR$ .

Chowdhury's parallel algorithm that we use [7], initializes Q as the identity matrix and does a column cyclic data distribution of A and Q. These matrices are successively transformed to obtain R and  $Q^T$  respectively, using orthogonal (Householder) transformations. To perform the  $k$ th update each processor needs the  $k$ th column of the previously updated matrix A. This column is sent to all processors by its owner, in each iteration.

A component-wise error bound for QR factorization is given by  $|A - QR| \leq nr\gamma_c G|A|$ , where  $\gamma_c = cnu/(1 - cnu)$  with c a small integer constant, n is the matrix dimension, r is the number of Householder transformations, and  $G = n^{-1}ee^T$ , with  $e=(1,1, \dots, 1)^T$  [14].

We have got a bound of approx.  $10^{-9}$  for our data, which is quite high because computing the real difference for the data we used  $|A - QR|$  we got a maximum error of approx.  $10^{-13}$ . For this case we then decided to use three intervals to classify the errors: the case when the maximum absolute difference to the reference results is less than  $10^{-12}$ , the case when that difference is between  $10^{-12}$  and  $10^{-9}$  and the case when that difference is greater than  $10^{-9}$ .

We used as our test case a 210-by-210 double precision matrix with random entries chosen from (-100, +100). The output is a file with the matrices Q and R.

### 6.1 QR factorization with ABFT

The ABFT used for QR factorization [7] checks both dimensions of Q and R, and also tests the column that is broadcast in each iteration. As before the nodes are logically configured in a directed cycle. Before sending the final results to the host, the working processors protect their results with a CRC.

The tolerance values used in each test were the minimum values for which we did not get false alarms for the data set used. We also test if any value of the result matrices, Q and R, are NaNs. The results of fault injection experiments are shown in table VII. As can be seen all the faults that lead to results with an error greater than  $10^{-12}$  have been detected.

QR with ABFT (and CRC)	no. of faults	% of total	Undetected		
			no.	%	
Correct Output	4306	48.8	-	-	
No output	70	0.8	-	-	
Timeout	2550	28.9	-	-	
Wrong outputs	NaNs	21	0.2	0	0.0
	diff. $>10^{-9}$	1608	18.2	0	0.0
	$10^{-12} < \text{diff.} \leq 10^{-9}$	117	1.3	0	0.0
	diff. $\leq 10^{-12}$	160	1.8	140	1.6
Total	8832	100.0	140	1.6	

Table VII - Results for QR factorization with ABFT.

### 6.2 QR factorization with result-checking

For the RC version, we eliminated from the previous program all the code corresponding to the ABFT, and added Freivalds' checker to the QR factorization. That is, we verify  $A=Q \times R$  by checking the equality  $A \times r = Q \times (R \times r)$ , where r is a random vector. We used two random vectors with entries in the range (-1,1) and (0,1). The tolerance value was  $10^{-12}$ , again calculated in order to eliminate false alarms for the matrix used. Table VIII presents the outcomes for this experiment

The results are similar to those of the ABFT mechanism. As can be seen only one wrong output

between  $10^{-12}$  and  $10^{-9}$  and a number of errors smaller than  $10^{-12}$  were not detected.

QR with RC (2 random vectors)		no. of faults	% of total	Undetected	
				no.	%
Correct Output		4332	45.4	-	-
No output		16	0.2	-	-
Timeout		2604	27.3	-	-
Wrong outputs	NaNs	25	0.3	0	0.0
	diff. $>10^{-9}$	2226	23.3	0	0.0
	$10^{-12} < \text{diff.} \leq 10^{-9}$	139	1.5	1	0.01
	diff. $\leq 10^{-12}$	196	2.0	149	1.56
Total		9538	100.0	150	1.57

Table VIII - Results for QR factorization with RC.

### 6.3 ABFT Vs result-checking

Comparing the error coverage of both mechanisms for QR factorization, we find similar results. Table IX shows the estimates for the non-coverage of both mechanisms and the corresponding upper 99% confidence limit estimates, calculated as in the matrix multiplication case.

Then ABFT has a fault/error coverage greater than 98.14% and RC has a fault/error coverage greater than 98.16% with a confidence level of 99%. If we consider only errors of magnitude greater than  $10^{-12}$ , which is probably more realistic, we get 99.99% for the coverage lower bound of ABFT and 99.93% for RC, with the same confidence level of 99%.

It is when comparing the execution time of both mechanisms that significant differences begin to appear, because ABFT has a much higher overhead than RC. Table X shows the calculation time overheads for some matrix dimensions. This large difference mainly results from the extra communication introduced by ABFT.

Memory overhead is of  $O(n)$  and is similar for both checkers.

Finally, ABFT is again much more complex to implement than RC, and RC is again applicable to a black box routine, while ABFT depends heavily on the particular algorithm and decomposition used.

QR – undetected wrong outputs					
mechanism	injected faults	diff $> 10^{-12}$	diff $\leq 10^{-12}$	$\bar{C}$	$\bar{C}_{99\%}$
ABFT+CRC	8832	0	140	1.59%	1.86%
RC-2 rand. vec.	9538	1	149	1.57%	1.84%

Table IX – Undetected wrong outputs, ABFT Vs RC, for QR.

QR factorization – calculation overheads			
matrix dim.	base time (second)	ABFT + CRC	RC 2 random vectors
210	5.05	41.58%	1.58%
300	11.83	39.98%	1.44%
360	19.08	34.75%	1.36%

Table X – Calculation time overheads, ABFT Vs RC, for QR.

## 7. Matrix inversion

Computing the inverse of a matrix A is equivalent to solving the system of linear equations  $AX=I$  (where I is the identity matrix). This is usually done by first computing the LU factorization of matrix A where L is a unit lower triangular matrix and U is upper triangular. It is known that the Gaussian elimination used to perform the LU decomposition is unstable unless we use partial or complete pivoting [20]. Introducing ABFT in that algorithm, as is proposed in [21] for systolic architectures, is complex, and the overhead due to the extra communication, needed to check the pivoting operations, will probably be even higher than in the QR case. Moreover, [22] shows, with an example, that the ABFT may not be able to detect faults in the pivoting operations. Besides, since we used here LAPACK routines, the changes needed to introduce ABFT would affect essentially all the routines' code, and would not be trivial at all to do (and thus quite error prone). We did not attempt to do it.

On the other hand, we did implement RC for that particular LAPACK routine, again using Freivalds' checker. This is intended as a very clear demonstration of the most important advantage of RC - its adaptability to already available, highly complex, numerical routines. We verified  $A \times A^{-1} = I$  by checking the equality  $A \times (A^{-1} \times r) = I \times r$ , with r a random vector. We should note that computing the inverse is an  $O(n^3)$  algorithm. Testing directly  $A \times A^{-1} = I$ , is an approx.  $O(n^3)$  test, while Freivalds' checker is an  $O(n^2)$  calculation.

We have used the LAPACK routines DGETRF and DGETRI to compute the inverse of a 200-by-200 double precision matrix with random entries from (-10, +10). The random vector had entries in the range (-1, +1) and the tolerance value was experimentally determined to be  $10^{-11}$  for the data set used. Table XI presents the outcomes for this experiment where we used only one processing node.

The results are quite good even with just one random vector. As can be seen all the wrong results with an error greater than  $10^{-11}$  have been detected and just 2.2% of the faults produced undetected wrong results with an error smaller than  $10^{-11}$ . That means an upper bound for the non-coverage of 2.51% (with 99% of confidence level).

This case shows that RC has a very high coverage even for Gauss elimination with pivoting. In that algorithm, faults that affect some intermediate results used for determining the order of the rows in pivoting, can not be detected by ABFT because these errors do not affect the checksum vectors. This shows that RC, due to its end-to-end nature, is able in this case to verify calculation steps that ABFT does not cover. This means that, although we did not perform that test, we would expect for this algorithm a coverage advantage for RC.

Matrix inversion with RC		no. of faults	% of total	Undetected	
				no.	%
Correct Output		3657	42.8	-	-
No output		12	0.2	-	-
Timeout		2039	23.9	-	-
Wrong outputs	NaNs	46	0.5	0	0.0
	diff. $>10^{-11}$	2488	29.1	0	0.0
	diff. $\leq 10^{-11}$	301	3.5	190	2.2
Total		8543	100.0	190	2.2

Table XI - Results for matrix inversion with RC.

## 8. Conclusions

In this work we compared RC with ABFT for some common matrix calculations. We tried to establish which has better coverage, lower overhead, greater ease of use and wider applicability. To the best of our knowledge, this is the first time that RC is validated by fault injection.

The main advantage of RC over ABFT is its independence of the underlying algorithm used to calculate some mathematical function. In fact, a complicated practical hurdle to use ABFT is the fact that many of the "clean" mathematical algorithms used for the derivation of ABFT algorithms are seldom used in numerical analysis, for performance and stability reasons. Adapting ABFT algorithms to those practical routines is not a trivial task. Result Checking can use those practical routines unchanged, as we have shown in the Matrix Inversion example. Besides, at least for the cases studied, the number of lines of code needed to implement RC is much smaller than for ABFT.

In terms of performance overhead, the advantage of RC is also clear for the more complex algorithms studied, where the difference can become quite large. In the more simple examples there is no significant difference between the two. No relevant difference was found in memory overhead.

Regarding coverage, RC and ABFT proved to be essentially equivalent, at least for the examples studied. It would be interesting to see RC and ABFT compared for other mathematical calculations.

A general conclusion of this work is that RC is a very promising line of research. It would be very nice to have a kind of library of result checkers for most of the existing numerical algorithms.

## References

[1] Chowdhury, A. R. and P. Banerjee, "Algorithm-Based Fault Location and Recovery for Matrix Computations" in FTCS-24, Austin, Texas, 1994, pp. 38-47.  
[2] Huang, K.-H. and J. A. Abraham, "Algorithm-Based Fault Tolerance for Matrix Operations", in IEEE Trans. on Comp., vol. C-33 no. 6, 1984, pp. 518-528.

[3] Rela, M., H. Madeira, and J. G. Silva. "Experimental Evaluation of the Fail-Silent Behavior of Programs with Consistency Checks" in FTCS-26, Sendai, Japan, 1996, pp. 394-403.  
[4] Silva, J. G., J. Carreira, H. Madeira, D. Costa, and F. Moreira, "Experimental Assessment of Parallel Systems" in FTCS-26, Sendai, Japan, 1996, pp. 415-424.  
[5] Blum, M. and S. Kannan, "Designing Programs that Check Their Work", Journal of the Association for Computing Machinery, 42(1), 1995, pp. 269-291.  
[6] Carreira, J., H. Madeira, and J. G. Silva, "Xception: A Technique for the Experimental Evaluation of Dependability in Modern Computers", in IEEE Trans. on Software Eng., vol. 24, no. 2, Feb. 1998, pp. 125-135.  
[7] Chowdhury, A. R., "Manual and Compiler Assisted Methods for Generating Fault-Tolerant Parallel Programs", Ph.D. Thesis. University of Illinois at Urbana-Champaign, 1995, 127 pages.  
[8] Blum, M. and H. Wasserman, "Reflections on The Pentium Division Bug", in IEEE Trans. on Comp., vol. 45, no. 4, April 1996, pp. 385-393.  
[9] Anderson, E., Z. Bai, C. Bischof, and e. al., "LAPACK Users' Guide", 1995, SIAM.  
[10] Jou, J.-Y. and J. A. Abraham, "Fault-Tolerant Matrix Arithmetic and Signal Processing on Highly Concurrent Computing Structures", in Proc. of the IEEE, vol. 74, no. 5, 1986, pp. 732-741.  
[11] Banerjee, P., et al., "Algorithm-Based Fault Tolerance on a Hypercube Multiprocessor", in IEEE Trans. on Comp., vol. 39, no. 9, Sep. 1990, pp. 1132-1144.  
[12] Chowdhury, A. R., N. Bellas, and P. Banerjee, "Algorithm-Based Error-Detection Schemes for Iterative Solution of Partial Differential Equations", in IEEE Trans. on Comp., vol. 45, no. 4, April 1996, pp. 394-407.  
[13] Silva, J. G., P. Prata, M. Rela, and H. Madeira. "Practical Issues in the Use of ABFT and a New Failure Model" in FTCS-28, Munich, Germany, 1998, pp. 26-35.  
[14] Higham, N., "Accuracy and Stability of Numerical Algorithms", SIAM, 1996.  
[15] Chowdhury, A. R. and P. Banerjee, "Tolerance Determination for Algorithm-Based Checks Using Simplified Error Analysis Techniques" in FTCS-23, 1993, pp. 290-298.  
[16] Wasserman, H. and M. Blum, "Software Reliability via Run-Time Result-Checking", Journal of the ACM, 44(6), 1997, pp. 826-849.  
[17] Rubinfeld, R. "Robust functional equations with applications to self-testing / correcting" in 35th IEEE Conference on Foundations of Computer Science, 1994, pp. 288-299.  
[18] Rubinfeld, R., "A Mathematical Theory of Self-Checking, Self-Testing and Self-Correcting Programs", PhD Thesis. University of California at Berkeley, 1990, 103 pages.  
[19] Powell, D., M. Cukier, and J. Arlat, "On Stratified Sampling for High Coverage Estimations", in 2nd European Dependable Computing Conference, Taormina, Italy, 1996, pp. 37-54.  
[20] Golub, G. H. and C. F. V. Loan, "Matrix Computations", Johns Hopkins University Press, Second edition 1989, 642 pages.  
[21] Yeh, Y.-M. and T.-Y. Feng, "Algorithm-Based Fault Tolerance for Matrix Inversion with Maximum Pivoting" Journal of Parallel and Distributed Computing, 14, 1992, pp. 373-389.  
[22] Boley, D., G. E. Golub, S. Makar, N. Saxena, and E. J. McCluskey, "Floating Point Fault Tolerance with Backward Error Assertions", in IEEE Trans. on Comp., vol. 44, no. 2, Feb. 1995, pp. 302-311.