Fault Tolerant QR-Decomposition Algorithm and Its Parallel Implementation

Oleg Maslennikow[1], Juri Kaniewski[1],Roman Wyrzykowski[2]

[1] Dept. of Electronics, Technical University of Koszalin,
Partyzantow 17, 75-411 Koszalin, Poland
[2] Dept. of Math. & Comp. Sci, Czestochowa Technical University,
Dabrowskiego 73, 42-200 Czestochowa, Poland

Abstract. A fault tolerant algorithm based on Givens rotations and a modified weighted checksum method is proposed for the QR-decomposition of matrices. The algorithm enables us to correct a single error in each row or column of an input $M \times N$ matrix $\mathbf{A}$ occurred at any among N steps of the algorithm. This effect is obtained at the cost of $2.5N^2+O(N)$ multiply-add operations ($M = N$). A parallel version of the proposed algorithm is designed, dedicated for a fixed-size linear processor array with fully local communications and low I/O requirements.

1 Introduction

The high complexity of most of matrix problems [1] implies the necessity of solving them on high performance computers and, in particular, on VLSI processor arrays [3]. Application areas of these computers demand a large degree of reliability of results. while a single failure may render computations useless. Hence fault tolerance should be provided on hardware or/and software levels [4].

The algorithm-based fault tolerance (ABFT) methods [5–9] are very suitable for such systems. In this case, input data are encoded using error detecting or/and correcting codes. An original algorithm is modified to operate on encoded data and produce encoded outputs, from which useful information can be recovered easily. The modified algorithm will take more time in comparison with the original one. This time overhead should not be excessive. An ABFT method called the weighted checksum (WCS) one, especially tailored for matrix algorithms and processor arrays, was proposed in Ref. [5]. However, the original WCS method is little suitable for such algorithms as Gaussian elimination, Choleski, and Faddeev algorithms, etc., since a single transient fault in a module may cause multiple output errors, which can not be located. In Refs. [8, 9], we proposed improved ABFT versions of these algorithms.

For such important matrix problems as least squares problems, singular value and eigenvalue decompositions, more complicated algorithms based on the QR-decomposition should be applied [2]. In this paper, we design a fault tolerant version of the QR-decomposition based on Givens rotations and a modified WCS method. The derived algorithm enables correcting a single error in each row or

column of an input $M \times N$ matrix $\mathbf{A}$ occurred at any among N steps of the algorithm. This effect is obtained at the cost of $2.5N^2 + O(N)$ multiply-add operations. A parallel version of the algorithm is designed, dedicated for a fixed-size linear processor array with local communications and low I/O requirements.

2 Fault Model and Weighted Checksum Method

Module-level faults are assumed [6] in algorithm-based fault tolerance. A module is allowed to produce arbitrary logical errors under physical failure mechanism. This assumption is quite general since it does not assume any technology-dependent fault model. Without loss of generality, a single module error is assumed in this paper. Communication links are supposed to be fault-free.

In the WCS method [6], redundancy is encoded at the matrix level by augmenting the original matrix with weighted checksums. Since the checksum property is preserved for various matrix operations, these checksums are able to detect and correct errors in the resultant matrix. The complexity of correction process is much smaller than that of the original computation. For example, a WCS encoded data vector $\mathbf{a}$ with the Hamming distance equal to three (which can correct a single error) is expressed as

$$\mathbf{a}^T = [a_1 \quad a_2 \ldots a_N \quad PCS \quad QCS] \tag{1}$$

$$PCS = \mathbf{p}^T \ [a_1 \quad a_2 \quad \ldots \quad a_N], \quad QCS = \mathbf{q}^T \ [a_1 \quad a_2 \quad \ldots \quad a_N] \tag{2}$$

Possible choices for encoder vectors $\mathbf{p}$, $\mathbf{q}$ are, for example, [10] :

$$\mathbf{p}^T = \left[2^0 \quad 2^1 \quad \ldots \quad 2^{N-1}\right], \quad \mathbf{q}^T = [1 \quad 2 \quad \ldots \quad N] \tag{3}$$

For the floating-point implementation, numerical properties of single-error correction codes based on different encoder vectors were considered in Refs. [7, 11].

Based on an encoder vector, a matrix $\mathbf{A}$ can be encoded as either a row encoded matrix $\mathbf{A}_R$, a column encoded matrix $\mathbf{A}_C$, or a full encoded matrix $\mathbf{A}_{RC}$ [11]. For example, for the matrix multiplication $\mathbf{A}\,\mathbf{B} = \mathbf{D}$, the column encoded matrix $\mathbf{A}_C$ is exploited [6]. Then choosing the linear weighted vector (3), the equation $\mathbf{A}_C * \mathbf{B} = \mathbf{D}_C$ is computed. To have the possibility of verifying computations and correcting a single error, syndromes S_1 and S_2 for the j-th column of $\mathbf{D}$-matrix should be calculated, where

$$S_1 = \sum_{i=1}^{M} c_{ij} - PCS_j, \quad S_2 = \sum_{i=1}^{M} i * c_{ij} - QCS_j \tag{4}$$

3 Design of the ABFT QR-Decomposition Algorithm

The complexity of the Givens algorithm [2] is determined by $4N^3/3$ multiplications and $2N^3/3$ additions, for a real $N \times N$ matrix $\mathbf{A}$. Based on equivalent matrix transformations, this algorithm preserves the Euclidean norm for columns of $\mathbf{A}$ during computations. This property is important for error detection and enable us to save computations.

In the course of the Givens algorithm, an $M \times N$ input matrix $\mathbf{A} = \mathbf{A}^1 = \{a_{ij}\}$ is recursively modified in K steps to obtain the upper triangular matrix $\mathbf{R} = \mathbf{A}^{K+1}$, where $K = M-1$ for $M \leq N$, and $K = N$ for $M > N$. The i-th step consists in eliminating elements a^i_{ji} in the i-th column of $\mathbf{A}^i$ by multiplications on rotation matrices $\mathbf{P}_{ji}$, $j = i+1, \ldots, M$, which correspond to rotation coefficients

$$c_{ji} = a^{j-1}_{ii} / \sqrt{(a^{j-1}_{ii})^2 + (a^i_{ji})^2}, \qquad s_{ji} = a^i_{ji} / \sqrt{(a^{j-1}_{ii})^2 + (a^i_{ji})^2}$$

Each step of the algorithm includes two phases. The first phase consists in recomputing $M - i$ times the first element a_{ii} of the pivot (i.e. i-th) row, and computing the rotation coefficients. The second phase includes computation of a^{i+1}_{jk}, and resulting elements r_{ik} in the i-th row of $\mathbf{R}$. This phase includes also recomputing $M - i$ times the rest of elements in the pivot row.

Consequently, if during the i-th step, $i = 1, \ldots, K$, an element a^j_{ii} is wrongly calculated, then errors firstly appear in coefficients c_{ji} and s_{ji}, $j = i+1, \ldots, M$, and then in all the elements of $\mathbf{A}^{i+1}$. Moreover, if at the i-th step, any coefficient c_{ji} or s_{ji} is wrongly calculated, then errors firstly appear in all the elements of the pivot row, and then in all the elements of $\mathbf{A}^{i+1}$. All these errors can not be located and corrected by the original WCS method. To remove these drawbacks, the following lemmas are proved. It is assumed that a single transient error may appear at each row or column of $\mathbf{A}^i$ at any step of the algorithm.

Lemma 1. *If at the i-th step, an element a^{i+1}_{jk} $(i < j,\ i < k)$ is wrongly calculated, then errors will not appear among other elements of $\mathbf{A}^{i+1}$.*

However, if a^i_{jk} is erroneous, then error appears while computing either the element a^{j+1}_{jk} in the pivot row at the j-th step of the algorithm, for $j \leq k$, or values of a^j_{kk}, c_{jk} and s_{jk} $(j = i+1, \ldots, M)$ at the k-th step, for $j > k$. Hence in these cases, we should check and possibly correct elements of the i-th and j-th rows of $\mathbf{A}^i$, each time after their recomputing.

Lemma 2. *Let an element a^j_{ik} of the pivot row $(j = i+1, \ldots, M;\ k = 1, \ldots, N)$ or an element a^{i+1}_{jk} of a non-pivot row $(k = i+1, \ldots, N)$ was wrongly calculated when executing phase 2 of the i-th step of the Givens algorithm. Then it is possible to correct its value while executing this phase, using the WCS method for the row encoded matrix $\mathbf{A}_R$, where*

$$\mathbf{A}_R = [\mathbf{A}\ \ \mathbf{Ap}\ \ \mathbf{Aq}] = [\mathbf{A}\ \mathbf{PCS}\ \mathbf{QCS}] \tag{5}$$

$$\begin{aligned} PCS^{i+1}_j &= a^{i+1}_{j,i+1} + a^{i+1}_{j,i+2} + \ldots a^{i+1}_{j,N} = (-s_{ji}a^i_{i,i+1} + c_{ji}a^i_{j,i+1}) + \ldots \\ &+ (-s_{ji}a^i_{i,N} + c_{ji}a^i_{j,N}) = -s_{ji}(a^i_{i,i+1} + a^i_{i,N}) + c_{ji}(a^i_{j,i+1} + a^i_{j,N}) \end{aligned} \tag{6}$$

So before executing phase 2 of the i-th step we should be certain that c_{ji}, s_{ji} and a^j_{ii} were calculated correctly at phase 1 of this step (we assume that the remaining elements were checked and corrected at the previous step). For this aim, the following properties of the Givens algorithm may be used:

- $$(c_{ji})^2 + (s_{ji})^2 = 1 \tag{7}$$
- preserving the Euclidean norm for column of $\mathbf{A}$ during computation

$$\sqrt{(a^i_{i,i})^2 + (a^i_{i+1,i})^2 + \ldots + (a^i_{M,i})^2} = a^M_{i,i} \ (\text{where } a^M_{ii} = r_{ii}) \tag{8}$$

The triple time redundancy (TTR) method [4] may also be used for its modest time overhead. In this case, values of c_{ji}, s_{ji} or/and a^j_{ii} are calculated three times.

Hence, for c_{ji} and s_{ji}, the procedure of error detection and correction consists in computing the left part of expression (7), and its recomputing if equality (7) is not fulfilled, taking into account a given tolerance τ [6]. For elements a^j_{ii}, $i = 1, \ldots, K$, this procedure consists in computing the both parts of expression (8), and their recomputing if they are not equal. The correctness of this procedure is based on the assumption that only one transient error may appear at each row or column of $\mathbf{A}^i$ at any step. Moreover, instead of using the correction procedure based on formulae (4), we recompute all elements in the row with an erroneous element detected.

The resulting ABFT Givens algorithm is as follows:

1. The original matrix $\mathbf{A} = \{a_{jk}\}$ is represented as the row encoded matrix $\mathbf{A}_R = \{a^1_{jk}\}$ with $a^1_{jk} = a_{jk}$, $a^1_{j,N+1} = PCS^1_j$, for $j = 1, \ldots, M$; $k = 1, \ldots, N$.
2. For $i = 1, 2, \ldots, K$, stages 3-9 are repeated.
3. The values of $a^j_{ii} = \sqrt{(a^{i-1}_{ii})^2 + (a^i_{ji})^2}$, are calculated, $j = i+1, \ldots, M$.
4. The norm $\mid \mathbf{a}_i \mid$ for the i-th column of $\mathbf{A}$ is calculated. This stage needs approximately $M-i$ multiply-add operations. The value of $\mid \mathbf{a}_i \mid$ is compared with the value of a^M_{ii}. If $a^M_{ii} \neq \mid \mathbf{a}_i \mid$, then stages 3,4 are repeated.
5. The coefficients c_{ji} and s_{ji} are computed, and correctness of equation (7) is checked, for $j = i+1, \ldots, M$. In case of non-equality, stage 5 is repeated.
6. For $j = i+1, \ldots, M$, stages 7-10 are repeated.
7. The elements a^j_{ik} of the i-th row of $\mathbf{A}^{i+1}$ are computed, $k = 1, \ldots, N+1$.
8. The value of PCS^j_i is calculated according to (6). This stage needs approximately $N-i$ additions. The obtained value is compared with that of $a^j_{i,N+1}$. In case of the negative answer, stages 7,8 are performed again.
9. The elements a^{i+1}_{jk} in the j-th row of $\mathbf{A}^{i+1}$ are calculated, k = i+1, ...,N+1.
10. The value of PCS^{i+1}_j is computed according to expression (6). This stage also needs approximately $N-i$ additions. The computed value is compared with that of $a^{i+1}_{j,N+1}$. In case of the negative case, stages 9,10 are repeated.

The procedures of error detection and correction increase the complexity of the Givens algorithm on $N^2/2 + O(N)$ multiply-add operations and $N^2 + O(N)$ additions, for $M = N$. Due to increased sizes of the input matrix, the additional overhead of the proposed algorithm is $2N^2 + O(N)$ multiplications and $N^2 + O(N)$ additions ($M = N$). As a result, the complexity of the whole algorithm is increased approximately on $2.5N^2 + O(N)$ multiply-add operations and $2N^2 + O(N)$ additions. At this cost, the proposed algorithm enables us to correct one single transient error occured in each row or column of $\mathbf{A}$ at any among K steps of computations. Consequently, for $M = N$, it is possible to correct up to N^2 single errors when solving the whole problem.

4 Parallel Implementation

The dependence graph $\mathbf{G}_1$ of the proposed algorithm is shown in Fig.1 for $M = 4$, $N = 3$. Nodes of $\mathbf{G}_1$ are located in vertices of the integer lattice $\mathbf{Q} = \{\mathbf{K} =$

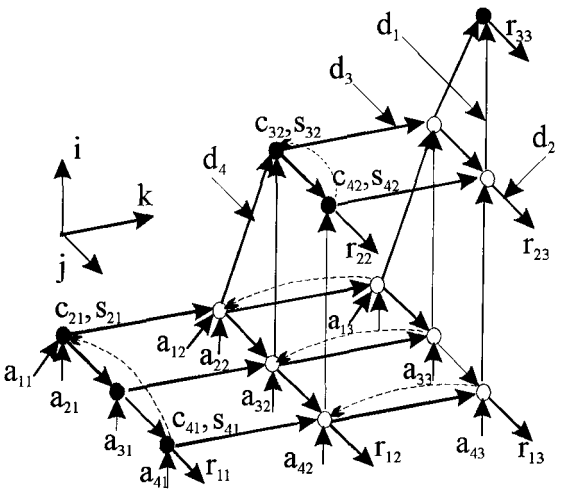

Fig. 1. Graph of the algorithm

$(i,j,k) \ : \ 1 \leq i \leq K;\ i+1 \leq j \leq M;\ i \leq k \leq N\}$. There are two kind of nodes in $\mathbf{G}_1$. Note that non-local arcs marked with broken lines, and given by vectors $\mathbf{d}_5 = (0, i+1-M, 0)$, $\mathbf{d}_6 = (0, 0, i-N)$ are result of introducing ABFT properties into the original algorithm.

To run the algorithm in parallel on a processor array with local links, all the vectors $\mathbf{d}_5$ are excluded using the TTR technique for computing values of a_{ii}^{j}, c_{ji} and sji. This needs to execute additionally $6N^2 + O(N)$ multiplications and additions. Then all the non-local vectors $\mathbf{d}_6$ are eliminated by projecting $\mathbf{G}_1$ along k-axis. As a result, a 2-D graph $\mathbf{G}_2$ is derived (see Fig.2).

To run $\mathbf{G}_2$ on a linear array with a fixed number n of processors, $\mathbf{G}_2$ should be decomposed into a set of $s =]N/n[$ subgraphs with the "same" topology and without bidirectional data dependencies. Such a decomposition is done by cutting $\mathbf{G}_2$ by a set of straight lines parallel to j-axis. These subgraphs are then mapped into an array with n processors by projecting each subgraph onto i-axis [12]. The resulting architecture, which is provided with an external RAM module, features a simple scheme of local communications and a small number of I/O channels. The proposed ABFT Givens algorithm is executed on this array in

$$T = \sum_{i=1}^{s} [(\ N + 3 - n(i-1)) \ + \ (M - n(i-1)\ )]$$

time steps. For $M = N$, we have

$$T = N^3/n \ - \ (N-n)N^2/2 \ + \ (2N-n)N^2/(6n)$$

The processor utilization is $E_n = W/(T * n)$, where W is the computational complexity of the proposed algorithm.

Using these formulae, for example, in case of $s = 10$, we obtain

$$E_n = 0.86$$

Note that with increasing in parameter s the value of E_n also increases.

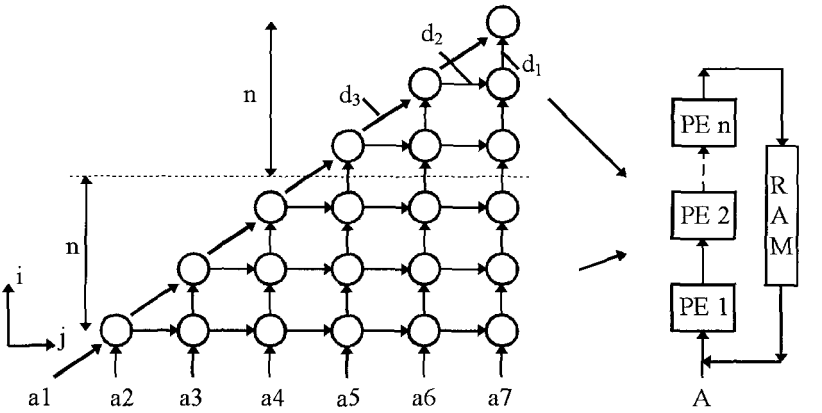

Fig. 2. Fixed-sized linear array

References

1. G.H. Golub, C.F. Van Loan, *Matrix computations*, John Hopkins Univ. Press, Baltimore, Maryland, 1996.
2. Å. Björck, *Numerical methods for least squares problems*, SIAM, Philadelphia, 1996.
3. S.Y. Kung, *VLSI array processors*, Prentice-Hall, Englewood Cliffs, N.J., 1988.
4. S.E. Butner, Triple time redundancy, fault-masking in byte-sliced systems. Tech.Rep. CSL TR-211, Dept. of Elec.Eng., Stanford Univ., CA, 1981.
5. K.H. Huang, J.A. Abraham, Algorithm-based fault tolerance for matrix operations. IEEE Trans. Comput., C-35 (1984) 518-528.
6. V.S. Nair, J.A. Abraham, Real-number codes for fault-tolerant matrix operations on processor arrays. IEEE Trans. Comp., C-39 (1990) 426-435.
7. J.-Y. Han, D.C. Krishna, Linear arithmetic code and its application in fault tolerant systolic arrays, in *Proc. IEEE Southeastcon*, 1989, 1015-1020.
8. J.S. Kaniewski, O.V. Maslennikow, R. Wyrzykowski, Algorithm-based fault tolerant matrix triangularization on VLSI processor arrays, in *Proc. Int. Workshop "Parallel Numerics'95"*, Sorrento, Italy, 1995, 281-295.
9. J.S. Kaniewski, O.V. Maslennikow, R. Wyrzykowski, Algorithm-based fault tolerant solution of linear systems on processor arrays, in *Proc. 7-th Int. Workshop "PARCELLA '96"*, Berlin, Germany, 1996, 165 - 173.
10. D.E. Schimmel, F.T. Luk, A practical real-time SVD machine with multi-level fault tolerance. Proc. SPIE, 698 (1986) 142-148.
11. F.T. Luk, H. Park, An analysis of algorithm-based fault tolerance techniques. Proc. SPIE, 696 (1986) 222-227.
12. R. Wyrzykowski, J. Kanevski, O. Maslennikov, Mapping recursive algorithms into processor arrays, in *Proc. Int. Workshop Parallel Numerics'94*, M. Vajtersic and P. Zinterhof eds., Bratislava, 1994, 169-191.