A PARALLEL FAULT-DETECTION SCHEME FOR MATRIX INVERSION

Woei Lin
Department of Electrical Engineering
University of Hawaii at Manoa, Honolulu, HI-96822

T. L. Sheu
IBM, Communication Systems
Research Triangle
NC 27709

Javed I. Khan
CSE, Bangladesh University
of Engineering & Technology
Dhaka-1000, Bangladesh

Published in the Proceeding of the 5th ISMM International Conference on
Parallel and Distributed Computing and Systems
Pittsburgh, October 1992, pp394-398

A PARALLEL FAULT-DETECTION SCHEME FOR MATRIX INVERSION

Woei Lin
Department of Electrical Engineering
University of Hawaii at Manoa, Honolulu, HI-96822

T. L. Sheu
IBM, Communication Systems
Research Triangle
NC 27709

Javed I. Khan
Department of Electrical Engineering, University of Hawaii at Manoa
Honolulu, HI-96822

Abstract

Matrix inversion is an important operation used for solving linear system problems. It is crucial to ensure correctness of the arithmetics for the inversion. In this abstract we summarize key ideas and approaches of a parallel matrix inversion algorithm which has capabilities of error detection and correction. We focus on algorithm enhancement for detecting and correcting both computational and communication errors. A major feature of the proposed algorithm is that we carefully insert redundant computations for error detection so as to conform with the natural communication structure of the original algorithm. These redundant computations are treated in the same way as actual computations for matrix inversion. The well match between algorithm and architecture considerably reduces overhead due to error checking redundancy. The conformation lends to small increases in hardware and response time. The proposed scheme is implemented and tested.

1 Introduction

Matrix inversion is an operation used for solving linear system problems[1-3]. The operation involves a large number of arithmetics. It is crucial to ensure the correctness of the arithmetics. In this paper we are concerned with incorporating fault-tolerance capability into a parallel matrix inversion algorithm so that errors can be detected and corrected. Our method uses hexagonally connected mesh (HCM).

The proposed error-detection algorithm is based upon an algorithm-based fault tolerance scheme, which is first presented by Hwang and Abraham [4,5]. They introduce a checksum approach that provides single-error detection for basic matrix operations. The checksum technique has shown to be considerably effective in many applications such as QR decomposition, matrix multiplication, and eigenvalue problem [4-6,9]. Most of them only consider computational aspects of the checksum scheme, while neglecting its effect on inter processor communication of the target machine. The proposed algorithm is different from previous work in three ways: (1) we consider a different matrix inversion method, called Fadeeva method [7]; (2) redundant operations are deliberately injected and arranged in conform with original communication structure; and (3) in many cases, it not only can detect single errors but can perform error corrections.

The fault model we assume here comprises two types of single errors: (1) computation errors: they are erroneous arithmetic results produced by processors; (2) communication errors; They are data messages which are corrupted or damaged during transmission between processors.

2 A Parallel Matrix Inversion Algorithm

2.1 A Computation Method for Matrix Inversion

The basis of the Faddeeva algorithm is a compact computational scheme originally used to compute matrix determinants[7-8]. The compact scheme is described as follows. Throughout this paper we assume that A is a nonsingular matrix and pivots are all nonzeros. Given the matrix:

$$A = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ . & . & . & . \\ . & . & . & . \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{pmatrix} \quad (1)$$

We compute the determinant of A. Let $a_{11} \neq 0$. Extract the element a_{11} from the first row. Thus we have:

$$|A| = a_{11} * \begin{vmatrix} a_{11.1} a_{12.1} \dots a_{1n.1} \\ a_{21.1} a_{22.1} \dots a_{2n.1} \\ \dots . \\ \dots . \\ a_{n1.1} a_{n2.1} \dots a_{nn.1} \end{vmatrix} \quad (2)$$

where $a_{ij.1}=a_{ij}-a_{i1}*a_{1j}/a_{11}$, $2 \leq i, j \leq n$. That is, the original determinant is reduced to a product of a_{11}, and a determinant of $(n-1)^{th}$ order. Now suppose $a_{22.1} \neq 0$. We carry out the same computation with the reduced determinant. It is further reduced and has the form

$$|A| = a_{11} \cdot a_{22.1} \cdot \begin{vmatrix} a_{11.2} a_{12.2} \dots a_{1n.2} \\ a_{21.2} a_{22.2} \dots a_{2n.2} \\ \dots . \\ \dots . \\ a_{n1.2} a_{n2.2} \dots a_{nn.2} \end{vmatrix} \quad (3)$$

where $a_{ij.2}=a_{ij.1}-a_{i1.1}*a_{1j.1}/a_{11.1}$, $3 \leq i, j \leq n$. Carrying out the computation iteratively for n times, we eventually obtain a product

$|A| = a_{11} * a_{22.1} * \dots * a_{nn.n-1}$,

provided $a_{11,}\ a_{22.1,} \dots,\ a_{nn.n-1}$ are all non-zero. During the course of reduction, each iteration k, $1 \leq k < n$, involves $(n-k)^2$ computations of $a_{ij.k,}\ k < i, j \leq n$. The key idea of inverting matrix is to compute its cofactors by using the compact scheme depicted above. To compute the cofactor of a_{ij}, we need to augment A with an additional row and an additional column in the following manner

$$A = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} & b_{1j} \\ a_{21} & a_{22} & \dots & a_{2n} & b_{2j} \\ . & . & . & . & . \\ . & . & . & . & . \\ a_{n1} & a_{n2} & \dots & a_{nn} & b_{nj} \\ b_{i1} & b_{i2} & \dots & b_{in} & c_{ij} \end{pmatrix}$$

$$b_{im} = \begin{cases} 0, & if \quad m \neq i \\ 1, & if \quad m = i, \end{cases}$$

$$b_{mj} = \begin{cases} 0, & if \quad m \neq j \\ 1, & if \quad m = j, \end{cases}$$

$$c_{ij} = 0 \tag{4}$$

We refer to such a matrix as the *augmented matrix* of the element a_{ij}. Next we show that the inverse of a matrix can be obtained by applying the compact scheme to the augmented matrix.

Theorem 1: The inverse of an *nxn* nonsingular matrix A is

$$A^{-1} = \begin{pmatrix} -c_{11.n} - c_{12.n} \ldots - c_{1n.n} \\ -c_{21.n} - c_{22.n} \ldots - c_{2n.n} \\ \ldots . \\ \ldots . \\ -c_{n1.n} - c_{n2.n} \ldots - c_{nn.n} \end{pmatrix} \tag{5}$$

where elements $c_{ij.n,}$ $1 \leq n,j \leq n$, are obtained by using the compact scheme of the augmented matrix of A, provided $a_{11,}$ $a_{22.1}$,..$a_{nn,n-1}$ are non-zero.

Proof: For an element of the matrix a_{ij}, we may obtain its augmented matrix by (4). If we delete the i^{th} row and the j^{th} column from the augmented matrix, the determinant of the resulting matrix actually is $-A_{ij}$. On the other hand, if we apply the compact scheme to the *(n+1) by (n+1)* augmented matrix, the determinant of the augmented matrix should be $-A_{ij}$. As a result of this, we have

$-\mathbf{A}_{ij}=a_{11}*a_{22.1}*...*a_{nn.n-1}*c_{ij.n}=|\mathbf{A}|*c_{ij.n}$.

By Cramer's Rule, we have

$$A^{-1} = \frac{1}{|A|} * \begin{pmatrix} A_{11} & A_{12} & \ldots & A_{1n} \\ A_{21} & A_{22} & \ldots & A_{2n} \\ . & . & . & . \\ . & . & . & . \\ A_{n1} & A_{n2} & \ldots & A_{nn} \end{pmatrix}$$

$$= \begin{pmatrix} -c_{11.n} - c_{12.n} \ldots - c_{1n.n} \\ -c_{21.n} - c_{22.n} \ldots - c_{2n.n} \\ \ldots . \\ \ldots . \\ -c_{n1.n} - c_{n2.n} \ldots - c_{nn.n} \end{pmatrix}$$

2.2 Algorithm Mapping

In the augmented matrix [4], in iteration k, $1 \leq k \leq n-1$, only computations $a_{ij.k}$, $i,j > k$, are active, while other computations $a_{ij.k}$, $i,j \leq k$ remains idle. Throughout our discussion active computations of this type are referred to as *essential computations*. The sum of essential computations in each iteration is universally equal to n^2.

The target hardware is a hexagonally connected mesh (HCM) as shown in Figure 1(a). The algorithm requires an nxn HCM for inverting an nxn matrix. The essential computations of the k^{th} iterations are mapped to processor P[x,y], $1 \le x, y \le n$

$$a_{ij.k}\,(i,j > k) \rightarrow P[i-k,j-k]$$
$$b_{ij.k}\,(i \le k < j) \rightarrow P[i-k+n,j-k]$$
$$b_{ij.k}\,(j \le k < i) \rightarrow P[i-k,j-k+n]$$
$$c_{ij.k}\,(i,j \le k) \rightarrow P[i-k+n,j-k+n] \quad (6)$$

This mapping scheme shows how the interprocessor communication takes place to move operands to proper processors and makes them available to essential computations. All the essential computations are of the form

$$x_{ij.k} = x_{ij.k-1} - x_{ik.k-1} * x_{kj.k-1} / x_{kk.k-1} \quad (7)$$

where x could be a, b, or c, depending on essential computation type and iteration count. The parallelization algorithm essentially consists of three phases: shift, multiply and broadcast. Communication structure of the three phases are different.

The first phase deals with moving $x_{ij.k-1}$ of (7) to new processor locations to compute $x_{ij.k.}$ It is evident that this requires a diagonal shift of all the elements as shown in Figure 1(b).

The communication structure associated with the second phase is a 2-D mesh where each processor is connected by orthogonal connections. Peripheral processors propagate $x_{ik.k-1}$ and $x_{kj.k-1}$ through connections in x direction and in y direction, respectively. The communication pattern is shown in Figure 1(c). Each processor, upon receiving both the peripheral values, performs the computation $x_{ik.k-1} * x_{kj.k-1}$.

In the third phase, a broadcast tree is used to make $a_{kk.k-1}$ from processor P[1,1] available to the other processors. Figure 1(d) shows the communication pattern.

Now all the operands for computing (7) are present in the processors. The kth iteration completes by this evaluation. n such iterations with the three overlapped phases are required to compute the inverse. Consecutive computation and communication wavefronts corresponding to the three phases in successive iterations sweep across the HCM from the northwest corner to the southeast corner. Figure 2 shows a schematic diagram of such a wavefront notion for mapping our algorithm onto the HCM.

3 Algorithm for Error Detection

3.1 Error Detection Using Check-Sum Technique

We inject redundant computations and communications into the aforementioned algorithm for computing checksums of essential computations. By the end, results of redundant computations are used to test correctness of an inverse matrix.

The proposed algorithm necessiates a linear increment in array size and thus requires an (n+1)x(n+1) HCM to invert an nxn matrix. All the processors execute the following algorithm to compute the inverse.

In phase I, the algorithm requires additional calculations to update contents of essential computations in checksums rows and columns. The communication structure is shown in Figure 1(b). At the end of the last iteration, resulting elements of the inverse matrix are stored in processors P[i,j], $2 \le i,j \le n+1$, and checksum elements are stored in processor P[1,j] and P[i,1],$1 \le i,j \le n+1$. Figure 3 illustrates the crosses of the checksum elements that appears in the four iterations for inverting an 3x3 matrix on an 4x4 HCM.

The mathematical treatment in the following theorem shows the correctness of our algorithm. It shows that at the end of the execution, the top row and the leftmost column should contain the summation vectors of the resulting inverse matrix. Let A be an nxn matrix to be inverted and A be the augmented matrix of A with summation column and summation row, that is,

$$A' = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} & a_{1(n+1)} \\ a_{21} & a_{22} & \dots & a_{2n} & a_{2(n+1)} \\ . & . & . & . & . \\ . & . & . & . & . \\ a_{n1} & a_{n2} & \dots & a_{nn} & a_{n(n+1)} \\ a_{(n+1)1} & a_{(n+1)2} & \dots & a_{(n+1)n} & a_{(n+1)(n+1)} \end{pmatrix} \tag{1}$$

where

$a_{i(n+1)} = \sum_{i=1}^{n} a_{ij}$, $a_{(n+1)j} = \sum_{j=1}^{n} a_{ij}$, and the

$$a_{(n+1)(n+1)} = \sum_{i=1}^{n} \sum_{j=1}^{n} a_{ij} \tag{8}$$

Theorem 2: By applying the algorithm to A, we have,

$$A^{-1} = \begin{pmatrix} -c_{11.n} & -c_{12.n} & \dots & -c_{1n.n} & \\ -c_{21.n} & -c_{22.n} & \dots & -c_{2n.n} & \\ . & . & . & . & . \\ . & . & . & . & . \\ -c_{n1.n} & -c_{n2.n} & \dots & -c_{nn.n} & \end{pmatrix}$$

where the elements c_{ij} are the inverse of matrix A as equation (5) and,

$b_{1(n+1)} = \sum_{i=1}^{n} c_{ij}$, $b_{(n+1)j} = \sum_{j=1}^{n} c_{ij}$,

and $b_{(n+1)(n+1)} = \sum_{i=1}^{n} \sum_{j=1}^{n} c_{ij}$.

Proof: The first part can be proved in a similar way as Theorem 1. It is omitted here. The second part can be proved by induction. In the first iteration, by (7) and (8) we have:

$$b_{(n+1)1.1} = b_{(n+1)1.0} - a_{(n+1)1.0} * b_{11.0}/a_{11.0}$$

$$= 1.0 - \left(a_{11.0} + \sum_{i=2}^{n} a_{i1.0} + 1.0 \right) * 1.0/a_{11.0}$$

$$= \sum_{i=2}^{n} b_{i1.1} + c_{11.1}$$

$$a_{(n+1)j.1} = a_{(n+1)j.0} - a_{(n+1)1.0} * a_{1j.0}/a_{11.0}$$

$$= \sum_{i=2}^{n} a_{ij.1} + b_{ij.1},$$

where $1 < j \le n$. Now suppose in the k^{th} iteration, we have

$$b_{(n+1)j.k} = \sum_{i=k+1}^{n} b_{ij.k} + \sum_{i=1}^{k} c_{ij.k}, \quad 1 \le j \le k, \text{and}$$

$$a_{(n+1)j.k} = \sum_{i=k+1}^{n} a_{ij.k} + \sum_{i=1}^{k} b_{ij.k}, \quad k < j \le n \cdot$$

In the k^{th} iteration, we should have

$$b_{(n+1)j.(k+1)}$$

$$= b_{(n+1)j.k} - a_{(n+1)(k+1).k} \cdot \frac{b_{(k+1)j.k}}{a_{kk.k}}$$

$$= \sum_{i=k+1}^{n} \left(b_{ij.k} - a_{i(k+1).k} \cdot \frac{b_{(k+1)j.k}}{a_{(k+1)(k+1).k}} \right)$$

$$= \sum_{i=k+2}^{n} b_{ij.k+1} + \sum_{i=1}^{K+1} c_{ij.k+1},$$

where $1 \le j \le k+1$. Likewise we should have

$$a_{(n+1)j.(k+1)} = \sum_{i=k+2}^{n} a_{ij.k+1} + \sum_{i=1}^{K+1} b_{ij.k+1}$$

where $k+1 < j \le n$. As a consequence of these, in the nth iteration, we should have,

$$b_{(n+1)j.n} = \sum_{i=1}^{n} c_{ij.n}$$

where $1 \le j \le n$. Similarly, we can prove that

$$b_{i(n+1).n} = \sum_{j=1}^{n} c_{ij.n} \quad \text{and} \quad a_{(n+1)(n+1).n} = \sum_{i=1}^{n} \sum_{j=1}^{n} c_{ij.n}.$$

After elements $c_{ij.n}$ are computed, we propagate checksums through columns and rows for verifying computations of $c_{ij.n,}$ $1 \le i,j \le n$. Each processor P[i,j], $2 \le i, j \le n+1$, subtracts its $c_{ij.n}$ from a checksum received from a neighboring processor and then passes the resulting checksum to the succeeding processor in the same direction. Resulting checksums at peripheral processors of the $(n+1)^{th}$ column and the $(n+1)^{th}$ row are referred to as *Error Detecting* Vector (EDV). EDV consists of two major components EDV_x and EDV_y. If no error occurs during the execution, EDV_x and EDV_y should be zero.

3.2 Error Identification

There are 5 major classes of single errors depending on the EDV patterns as shown in Figure 4. These patterns can be used to identify errors. These are the following:

(1) Class A: Errors of $a_{ij.k}$, $1 \le k \le n$. This class makes all the elements of EDV non-zero as shown in Figure 4(a). There are two subtypes: (i). A.1: when $k<i \le n$. In this situation an erroneous $a_{ii.k}$ (pivot) contaminates all the essential computations, and (ii) A.2: when $k<i$, $j \le n$ and $i \ne j$. By (7), $a_{ij.i-1}$ is used for evaluating $a_{mj.i}$, $i<m \le n+1$, and $b_{mj.i}$, $1 \le m \le i$. Therefore, such errors also contaminate all the subsequent essential computations.

(2) Class B: Errors of $b_{ij.k}$ $1 \le k \le n$. There are again two subtypes; (i) B.1: when $i \le k<j$. Unlike errors of class A, an erroneous $b_{ij.k}$ causes local damage to essential computations $b_{ij.m,}$ $k<m<j \le n$ and $i \le m$. At the end of the n^{th} iteration, errors are confined to $b_{i(n+1).n}$ and $c_{ij.n}$ $1 \le j \le n$. (ii) B.2: when $j \le k<i$. This is the symmetric situation of (i). Figures 4(b) and 4(c) show these two subtypes of class B errors.

(3) Class C: Errors of $c_{ij.k}$, $1 \le k \le n$ and $1 \le i, j \le k$. Such errors can only effect subsequent essential computations of $c_{ij.m}$, $k<m \le n$. Thus, the EDV pattern at the end is shown in Figure 4(d).

(4) Class X: Errors of (n+1) column and row. This is further divided into three subtypes. There corresponding EDV patterns are shown in Figure 4(e)-(g). A Common characteristic among EDV patterns of the three subclasses is that at least one of $EDV_x(n+1)$ and $EDV_y(n+1)$ contains a non-zero element. These subclasses are: (i) X.1: errors in $a_{i(n+1).k}$, $1 \le k \le n$. Here errors propagate like that of B.1, except that the error is indicated at $EDV_x(n+1)$ instead. (ii) X.2: errors in $a_{(n+1)j.k,}$ $1 \le k \le n$, which is symmetric to X.1, (iii) X.3: errors in $a_{(n+1)(n+1).k}$, $b_{i(n+1).k}$ and $b_{(n+1)j.k}$, $1 \le k \le n$, $1 \le i, j \le k$. This is similar to class C.

(5) Class Z: A single $EDV_x(i)$ or $EDV_y(j)$ becomes zero. Because checksum elements flow straight through columns or rows, only one element of EDV will be affected in the presence of a single error.

4 Error Correction Using EDV

Errors of class A is considered incurable. Errors of class B and C can be corrected in the following way. Errors of class X are not propagated to the essential computations. Thus, even in the presence of class X error, inverse is correctly obtained. Errors of class Z refer to incorrect computations performed for checksum testing. Thus they do not cause damage to an inverse matrix either. Below we present an error correction method for class B and C errors.

(1) Correction of Class B: In case of class B.1, add $EDV_y(j)$ to $-c_{ij.n}$ for $1 \le j \le n$. In case of class B.2, add $EDV_x(i)$ to $-c_{ij.n}$ for $1 \le i \le n$. Note, this action also restores the checksum condition.

(2) Error Correction for classes C, X and Z: In the case of erroneous $c_{ij.k}$, $1 \le k \le n$, $1 \le i, j \le k$, exactly $EDV_x(i)$ and $EDV_y(j)$ will become nonzero in n iterations. The intersection of the i^{th} row and j^{th} column of the inverse matrix locates the erroneous matrix element. Due to the confinement, errors of this class are considerably easy to correct. The procedure is: add $EDV_x(i)$ or $EDV_y(j)$ to $c_{ij.n}$.

Although incorrect essential computations of class X also incur non-zero elements present in EDV_x and EDV_y, their damages are not as critical as those of A, B and C. From (7) we know that errors of this class are never propagated to other essential computations of classes A, B and C, which have a direct impact on an inverse matrix. Hence, even in the presence of errors of class X, we still obtain a correct inverse. Errors of class Z refer to those incorrect computations performed in the last phase of the algorithm. Neither do they cause damages to an inverse matrix. Therefore, errors of the class can be ignored.

5 Summary

We have presented a parallel matrix inversion algorithm with the capability of error checking and correction. The strength of this algorithm lies in the fact that the redundant computations are integrated and overlapped with the essential computations. Five major classes of errors and their error patterns are identified. Procedures are provided to correct critical errors.

References

[1] H. T. Kung & C. E. Leiserson, Systolic Array (for VLSI), Technical report CS79103, Carnegie-Mellon University, Apr 1979.

[2] R. M. Kant & T. Kimura, Decentralized Parallel Algorithm for Matrix Computation, Proc. 5th Ann. Symp. of Comp. Arch., 1978, pp-100.

[3] G. A. Giest & M. T. Heath, Matrix Factorization on a Hypercube Multiprocessor, SIAM Proc. 1st Conf. on Hypercube Multiprocessor, Aug 1985, pp. 161-180.

[4] K. H. Huang & J. A. Abraham, Algorithm Based Fault Tolerance for Matrix Operations, IEEE Trans. on Comput., Vol c-33, No.6, June 1984. pp-518-528.

[5] K. H. Huang & J. A. Abraham, Fault Tolerant Algorithms and Their Applications to Solving Laplace Equations, Proc. 1984 Int'l Cong. of Parallel Processing, pp.117-122.

[6] F. T. Luk & H. Park, An Analysis of Algorithm-Based Fault-Tolerance Technique, Proc. SPIE, Vol. 696, pp. 222-227, 1986.

[7] V. N. Faddeeva, Computational methods of Linear Algebra, Chap. 2, trans. by C. D. Benster, Dover Pub., N.Y., 1959.

[8] B. F. V. Wanugh & P. S. Dwyer, Compact Computation of the Inverse of a Matrix, Annals of Mathematical Statistics, 16, pp 259-271, 1945.

[9] C. Y. Chen & J. A. Abraham, Fault-Tolerance Systems for Computation of Eigenvalues and Singular Values, Proc. SPIE, Vol. 696, pp. 228-237, 1986.

[10] W. Lin, Parallel programming with PAPA-A User Guide, Computer Engineering Technical report, The Penn State University, TR-88-063.

[11] J. G. Nash & S. Hausen, Modified Faddeeva Algorithm for Concurrent Execution of Linear Algebraic operations, IEEE Trans. on Computers, Vol. 37, No 2, Feb 1988, pp 129-139.

[ORIENTATION]A PARALLEL FAULT-DETECTION SCHEME FOR MATRIX INVERSION

Woei Lin
Department of Electrical Engineering
University of Hawaii at Manoa,
Honolulu, HI-96822

T. L. Sheu
IBM, Communication Systems Research Triangle
NC 27709

Javed I. Khan
CSE, Bangladesh University of Engineering & Technology
Dhaka-1000, Bangladesh