ADAPTIVE ALGORITHM-BASED FAULT TOLERANCE FOR PARALLEL COMPUTATIONS IN LINEAR SYSTEMS[1]

Javed I. Khan, W. Lin & D. Y. Y. Yun

Department of Electrical Engineering
University of Hawaii at Manoa
492 Holmes Hall, 2540 Dole Street
Honolulu, HI-96822
javed@wiliki.eng.hawaii.edu

SUMMARY

This paper presents a novel scheme for the stabilization of parallel matrix computation which is dynamically adaptive. The scheme performs automatic error **detection** and **correction** through inserting redundant but concurrent tracer computations within the folds of the regular computation. This scheme for the first time successfully eliminates the classical row/column interchange based pivoting which has been an expensive but only technique available to almost all of the parallel matrix algorithms to maintain stability. A *fault-tolerant* double wavefront algorithm for a MIMD array multi-processor with *toroidal* inter connection has been designed to demonstrate the strength of the proposed scheme. This algorithm can compute: i) matrix inverse ii) solution vector to the linear system and iii) predetermined linear combination of the solution vector from identical algorithmic framework. This tri-solution algorithm excels other known methods in parallel performance for all three problems. It can generate all three forms of solutions for a $n \times n$ system on a $\boldsymbol{p} \times p$ torus in **n steps** with $3\left\lceil \frac{(n+1)}{p} \right\rceil^2$ floating point operations per step. The proposed scheme also offers **detection** (and partial **recovery**) of various *transient* hardware failures, such as *memory faults*, and *message packet corruption* at algorithm level. Due to the adaptive pivoting and the unique dual-wavefront communication pattern, the resulting activity on the torus resembles the ripples on a pond formed by the raindrops. The paper includes performance results obtained from a 32 Node MIMD Meiko Transputer implementation.

Key Words:

Algorithmic Fault-Tolerancy, Numerical Computing, Parallel Algorithm, Adaptive Pivoting

1 This paper has been published in the proceedings of the **23rd Annual International Conference on Parallel Processing**, ICPP'94. Now we are in the process of benchmarking this algorithm on the SP2 super computer nodes at Maui High Performance Computing Center (MHPCC) and sending the final results to a journal.

ADAPTIVE ALGORITHM-BASED FAULT TOLERANCE FOR PARALLEL COMPUTING IN LINEAR SYSTEMS

Javed I. Khan, W. Lin & D. Y. Y. Yun
Department of Electrical Engineering
University of Hawaii at Manoa
Holmes 492, 2540 Dole Street, Honolulu, HI-96822
javed@wiliki.eng.hawaii.edu

ABSTRACT

This paper presents a dynamically adaptive stabilization scheme for parallel matrix computation. The scheme performs automatic error **detection** *and* **correction** *through inserting redundant, but concurrent tracer computations within the folds of the regular computation. It also eliminates the costly row interchange used in classical pivoting. A fault-tolerant double wavefront matrix algorithm for a MIMD array multi-processor with toroidal inter connection has been designed to demonstrate the strength of the proposed scheme. This algorithm can compute: i) matrix inverse ii) solution vector to the linear system and iii) predetermined linear combination of the solution vector from identical algorithmic framework. This efficient tri-solution algorithm excels most other known methods in parallel performance.*

1. INTRODUCTION

Other than speed, **stability** is the next most important computational issue in dealing with linear systems [11]. Most of the stable algorithms in linear systems are based on **triangular factorization** [2,3]. It is now known that these methods (generally based on *LU decomposition, Given's rotation*, etc.) are inefficient when parallelized [9]. Because, the involved forward and backward substitutions that follow and precede triangularization are inherently sequential. For example, in matrix inversion, one of the most stable and well-known triangular method based on Cholesky LU decomposition [10], is slower than the less stable classical *Gauss-Seidel* method by a factor of two [4]. Attempts to parallelize many linear system algorithms perplexingly revealed that the higher is the inherent stability of an algorithm, the lower is its scope of parallelization, and vice verse.

Partial pivoting can improve the stability of better parallelizable approaches [6]. Unfortunately, the very process of partial pivoting itself seriously undermines the concurrency of the target algorithm. Until now, the row (column) interchange, which is required by partial pivoting, remains prohibitively expensive given the architectural constraints of parallel processors. Very few alternative proposals exist to improve parallel pivoting. The possibility of restrained interchange based on threshold scheme has been raised by [2]. However, there is no satisfactory technique that can select appropriate threshold without incurring substantial communication cost. Some researchers have recently proposed fault tolerant approach to improve stability as a shift towards curative from preventive approach [5,8].

In this paper, we present a scheme for adaptive algorithmic fault tolerant computing in linear systems, based on two techniques. The combined scheme is capable of dynamic error detection, and correction of any single error. It has theoretical stability equivalent to that of partial pivoting. The first technique, called **adaptive pivoting** (AP), relaxes some of the artificial constraints of classical partial pivoting, and utilizes this relaxed computational model to eliminate the prohibitively expensive row column interchange of classical pivoting. The second technique, called **inter-phase checksum** (IPC), attempts to circumvent the problem of instability by deliberate insertion of redundant, but highly concurrent computational patterns inside the regular computations of an elimination based algorithm. This technique improves Huang and Abraham's [5] curative approach by incorporating a new spatio-temporal model of underlying fault propagation. Based on this new propagation model and the tracer patterns, IPC can dynamically *detect* and *correct* faults as they occur during the execution of the algorithm.

The effectiveness of the combined scheme has been demonstrated through a tri-solution algorithm. This algorithm has been derived from Faddeeva's non-triangular method for computing matrix determinant, which was first translated in English in 1959 [1]. As we will demonstrate, the derived tri-solution algorithm comes out to be faster than most other known parallel methods while solving each of the three problem forms even after combining the proposed stabilization scheme.

The following section first presents the derived algorithm and its mapping without AP and IPC. Section 3 introduces the AP technique. Section 4 presents the IPC scheme and underlying matrix error model. Finally, section 5 and 6 present the combined MIMD algorithm, its parallel complexity and scalability analysis, and the performance result from its implementation on a 32 Node MIMD Transputer System.

2. COMPUTATIONAL MODEL

Here we will only briefly describe the algorithm and its mapping on a torus. The details of the tri-solution procedure derivation can be found in [7]. Notations like $A_{xy.t}$ will be used to refer to the element at (x,y) co-ordinate of A at the t^{th} phase. In places, the co-ordinates will be dropped to refer to all the elements.

Computational Procedure: Let $AX=B$ is a linear system where X is the vector defining the n system variables, A is the coefficient matrix, and let CX be any linear combination of the system variables. Given A, B and C, we want to compute (i). A^{-1} (ii). $A^{-1}B$ and (iii). $CA^{-1}B$. In short, the scheme is equivalent to performing elimi-

nation on the following extended matrix. An *elimination* step refers to the computation of $a_{ij.k+1}=a_{ij.k}-a_{ik.k}*a_{kj.k}/a_{kk.k}$ where $n<i,j,k<n+1$.

$$\begin{bmatrix} A & IB \\ -CI & 0 \end{bmatrix}$$

For problem form (ii), the elements of vector *C*, and for problem form (i) the elements of both the vectors *B* and *C* are substituted by *1*.

Mapping of The Model: The derived procedure which solves all three forms of the problem can be summarized below in the algorithmic form.

1. If (i) Set B=C=U, if (ii) Set C=U where U is a vector with all unit elements. Enter the first phase k=1.

2. Take the original matrix $[a_{ij}]$ and create an extended matrix E^1 with $e_{ij}=a_{ij}$ for i,j=1,2..n.

3. Add a new row and a new column with all zero elements except, $e_{(n+1),1.k}=-c_k$ and $e_{1,(n+1).k}=b_k$.

4. Calculate $e_{i,j.k}=e_{i,j.k}-e_{i,1.k}*e_{1,j.k}/e_{1,1.k}$ for all i,j= 2,..(n+1).

5. Consider the elements $e_{ij.k}$ of E^k with i,j=2...(n+1) as the elements $e_{i,j.k+1}$ of the new extended matrix E^{k+1}.

6. Set k=k+1, and repeat steps 2,3 & 4 until phase k=n.

7. If (ii) Calculate column sum or, if (iii) Calculate sum of the column sums.

Steps 2 to 5 constitute the *core* part of this algorithm. Although, the size of the augmented matrix is *2nx2n*, but we have been able to map the entire computation on an *nxn* torus by considering only the active portion of the computations. The data dependency of this computational structure is satisfied by consecutive (i) row wise horizontal and (ii) a column wise vertical wave propagation between the phases. The communication speed of this compact algorithm has been further improved by allowing bi-directional waves instead of unidirectional flow, which reduces the network diameter into half. The sum-phase (7^{th} step) is required by solution forms (ii). and (iii). We will see later that this phase can be completely merged with the IPC phase.

3. ADAPTIVE PIVOTING

Classical pivoting schemes select the pivots along the principal forward diagonal (PFD). If there is a zero (or small) pivot, the row is usually interchanged with another row with non-zero (or a large) pivot. However, unlike the sequential algorithms, parallel algorithms have to incur prohibitively expensive communication cost to perform such interchange (consider the situation if the old and new rows are far away from each other). We therefore, suggest skipping the pivot position adaptively, rather than exchanging rows.

Theoretical Basis: We assume, $\boldsymbol{T}$ is the *nxn* torus with processors t_{ij}, and $\boldsymbol{A}$ is the initial matrix with elements a_{ij}, where $1 \le i, j \le n$. We also assume that initially the matrix element a_{ij} is mapped on processor t_{ij}, and let operator $\Leftarrow$ refers to 'maps on'.

***Theorem 1**: If $\boldsymbol{B}^T$ is the resultant matrix generated from **A** after performing n elimination steps and P is the set of n pivots used, then the sequence of selecting the pivots within this set in successive phases does not effect the final mapping $\boldsymbol{B}\Rightarrow\boldsymbol{T}$.*

***Theorem 2**: If **B** is the resultant matrix generated from matrix **A** after performing the Faddeeva elimination steps (as derived in section 2) using principal forward diagonal (PFD) pivot set P_{pfd}, and if P_{lm} and P_{xy} are two members of the applied pivot set P then at the end of the n^{th} phase, t_{mx} contains b_{ly} and t_{yl} contains b_{xm}.*

The formal proof of these two theorems can be found in [7]. Theorem 1 implies that the conventional PFD pivot selection strategy is over constrained. Same computational result is obtainable by other selection order in PFD. For, example, if pivot *(k,k)* is found to be zero (or too small) then, instead of exchanging the k^{th} row with $k+1^{th}$ row and then taking the new *(k,k)* element as the pivot, the algorithm can simply proceed from *(k+1,k+1)* element as the pivot, and can come back to *(k,k)* element at some later phase.

Theorem 2 further relaxes the constraint. It implies that it is not imperative to select the pivots from the PFD. Any n elements can be used as pivot so far no two of them are from the same row or column. However, this over relaxation has one important side effect; the mapping of the final elements changes.

Procedure: The crucial gap between the above relaxation and a practical pivoting scheme is the cost of retracing the convoluted mapping. A costly retracing can easily offset the gain from the avoidance of row interchange. This crucial gap has been effectively bridged by devising a distributed tracing scheme. As suggested by theorem 2, the tracing of the identity of the local element requires (i) phase sequence number, and (ii) corresponding pivot position. Both of these are available locally to the processor nodes through the basic algorithm. As a result, the entire mapping can be traced without any extra communication.

Adaptive pivoting makes prudent use of the above relaxation and, instead of interchanging rows, it adaptively skips pivot positions depending on their values (zero or too small a value suggests skipping). Every node on a torus is structurally equivalent. Therefore, all the computational phases remain uniform despite the irregular choice of pivot location during adaptive pivoting.

4. ERROR IDENTIFICATION

4.1 Mathematical Foundation

We have incorporated a new error propagation model to perform tracing between the phases. The resulting scheme called inter-phase checksum (IPC) can dynamically detect the exact prognosis of the fault as they occur. Previously, some researchers [5,7] have used checksum as the tracer computation to perform fault detection at the end of phases.

The IPC Computing Scheme: The proposed organization of the tracer computations has the following three characteristics; these are (i) simple and concurrently executable, (ii) phase computations are same as the original computations, thus no load imbalance occurs, (iii) the IPC data-dependency uses the original communication pattern of the target algorithm.

IPC requires an additional row and column (guard) to store and process the checksum elements. We label these as s^{th} row and r^{th} column. The wrapped mapping on the torus structure and the implicit computation of the *B* and *C* vectors require a compounded update scheme to guarantee the preservation of the checksum property in the target algorithm. Equations A.1 and A.2 provide the initial value to the guard elements and equations A.3 through A.10 provides the modified update equations.

$$a_{is} = \sum_{j \neq r}^{j \in \nabla} a_{ij} \qquad a_{rj} = \sum_{i \neq s}^{i \in \nabla} a_{ij}$$

$$a_{rs} = \sum_{i \neq s}^{i \in \nabla} \sum_{j \neq r}^{j \in \nabla} a_{ij} \qquad \ldots.(A.1)$$

At any k^{th} phase, $1 \leq k \leq n$, the modified phase computation, is specified in Equations A.3 to A.10. Let us assume, that at the k^{th} phase the pivot is $a_{pq.k}$, where, $p \in \nabla, p \neq r, q \in \nabla, q \neq s$ in the element space.

On the pivot position:

$$a_{pq.k+1} = \frac{b_p . c_q}{a_{pq.k}} \qquad \ldots.(A.2)$$

On the intersections of the pivot and guard rows and columns:

$$a_{rq.k+1} = -b_p \left(1 - \frac{a_{rq.k} + c_q}{a_{pq.k}} \right) \qquad \ldots.(A.3)$$

$$a_{ps.k+1} = c_q \left(1 - \frac{a_{ps.k} - b_p}{a_{pq.k}} \right) \qquad \ldots.(A.4)$$

In all other positions of the pivot row i.e., $i \in \nabla, i \neq p, i \neq r$,

$$a_{iq.k+1} = \frac{a_{iq.k} . b_p}{a_{pq.k}} \qquad \ldots.(A.5)$$

Similarly, in all other elements of the pivot column, i.e., when, $j \in \nabla, j \neq q, j \neq s$,

$$a_{pj.k+1} = -\frac{a_{pj.k} . c_q}{a_{pq.k}} \qquad \ldots.(A.6)$$

On the intersection of guard row and column:

$$a_{rs.k+1} = (a_{rs.k} + c_q - b_p) - \frac{(a_{rq.k} + c_q)(a_{ps.k} - b_p)}{a_{pq.k}} \qquad \ldots.(A.7)$$

On the guard row, i.e., when, $i \in \nabla, i \neq p, i \neq r$,

$$a_{is.k+1} = a_{is.k} - \frac{a_{iq.k} . (a_{ps.k} - b_p)}{a_{pq.k}} \qquad \ldots.(A.8)$$

Similarly, on the guard column, i.e., when, $j \in \nabla, j \neq q, j \neq s$,

$$a_{rj.k+1} = a_{rj.k} - \frac{a_{pj.k} (a_{rq.k} + c_q)}{a_{pq.k}} \qquad \ldots.(A.9)$$

For all other elements a_{ij}, i.e., when, $i \in \nabla, j \in \nabla, i \neq p, i \neq r, j \neq q, j \neq s$ an elimination is performed on it as shown below:

$$a_{ij.k+1} = a_{ij.k} - \frac{a_{pj.k} . a_{iq.k}}{a_{pq.k}} \qquad \ldots.(A.10)$$

Preservation of Checksum: Below we show how the computations specified by (A.1) to (A.10) guarantee the preservation of checksum after each phase updates.

Theorem 3*: At the end of the phase computation specified by (A.1) to (A.10), the IPC rows and columns maintain the checksum property if the involved computations are correct.*

Proof: Let, a_{pq} be the phase pivot and r^{th} column and s^{th} row are the guards, where, $p, q, r, s \in \nabla$. The proof is in three parts. We show the preservation of the checksum property at (i). position *(p,s)*, and *(r,q)*, (ii). all other elements on the s^{th} row and r^{th} column except *(r,s)*, and (iii). at position *(r,s)*.

For part (i), from equation A.4;

$$\begin{aligned} a_{ps.k+1} &= c_q \left(1 - \frac{a_{ps.k} - b_p}{a_{pq.k}} \right) \\ &= \frac{c_q b_p}{a_{pq.k}} + \frac{c_q}{a_{pq.k}} \left(a_{pq.k} - \sum_{j \neq s}^{j \in \nabla} a_{pj.k} \right) \\ &= a_{pq.k+1} + \frac{c_q}{a_{pq.k}} \left(a_{pq.k} - \sum_{j \neq s, j \neq q}^{j \in \nabla} a_{pj.k} - a_{pq.k} \right) \\ &= a_{pq.k+1} + \sum_{j \neq s, j \neq q}^{j \in \nabla} \left(-\frac{a_{pj.k} c_q}{a_{pq.k}} \right) \\ &= a_{pq.k+1} + \sum_{j \neq s, j \neq q}^{j \in \nabla} a_{pj.k+1} = \sum_{j \neq s}^{j \in \nabla} a_{pj.k+1} \end{aligned} \qquad \varnothing$$

In a similar way it can be proved that,

$$a_{rq.k+1} = -b_p \left(1 - \frac{a_{rq.k} + c_q}{a_{pq.k}} \right)$$

$$= \sum_{i \neq r}^{i \in \nabla} a_{iq.k+1} \qquad \varnothing$$

For part (ii), we will first show that an element on the guard column a_{rj}, when, $j \neq q, j \neq s$, from equation (A.9),

$$\begin{aligned} a_{rj.k+1} &= a_{rj.k} - \frac{a_{pj.k}}{a_{pq.k}} (a_{rq.k} + c_q) \\ &= \sum_{i \neq r}^{i \in \nabla} a_{ij.k} - \frac{a_{pj.k} a_{rq.k}}{a_{pq.k}} - \frac{a_{pj.k} c_q}{a_{pq.k}} \\ &= \sum_{i \neq r, i \neq s}^{i \in \nabla} \left(a_{ij.k} - \frac{a_{pj.k} a_{iq.k}}{a_{pq.k}} \right) + 0 + a_{pj.k+1} \\ &= \sum_{i \neq r}^{i \in \nabla} a_{ij.k+1} \end{aligned} \qquad \varnothing$$

In a similar way it can be shown that the elements on the guard row a_{ir} is, when $i \neq p, i \neq r$ preserves the checking property or,

$$a_{is.k+1} = \sum_{j \neq s, j \neq q}^{j \in \nabla} a_{ij.k+1} \qquad \varnothing$$

For part (iii) we will use the result of part (i) and (ii). Starting from equation (A.7),

$$a_{rs.k+1} = a_{rs.k} + c_q - b_p - \frac{(a_{rq.k} + c_q)(a_{ps.k} - b_p)}{a_{pq.k}}$$

$$= \sum_{i \neq r}^{i \in \nabla} a_{is.k} - \sum_{i \neq r}^{i \in \nabla} \frac{a_{iq.k}}{a_{pq.k}}(a_{ps.k} - b_p) - b_p + c_q\left(1 - \frac{a_{ps.k} - b_p}{a_{pq.k}}\right)$$

$$= \sum_{i \neq r, i \neq p}^{i \in \nabla} a_{is.k+1} + a_{ps.k+1}$$

$$= \sum_{i \neq r}^{i \in \nabla} a_{is.k+1} = \sum_{i \neq r}^{i \in \nabla} \sum_{j \neq s}^{j \in \nabla} a_{ij.k+1} \qquad \varnothing$$

The following three corollaries of theorem 2 and 3 provides validity of the resultant computation.

Corollary 1*: Given a linear system AX=B of order n, and, $c_1 = c_2 = \ldots c_n = 1$, and $b_1 = b_2 = \ldots b_n = 1$ the computational scheme specified by equations (A.1) to (A.10), computes the inverse of A of the linear system at the regular computational space $(\nabla - \nabla_s)$ with mapping specified by Theorem 2.*

The similarity of phase computation in the regular row and column with the computation scheme of section 2.2 provides the proof for this corollary.

Corollary 2*: Given a linear system AX=B of order n, and, $c_1 = c_2 = \ldots c_n = 1$, the computational scheme specified by equations (A.1) to (A.10), computes the solution of the linear system at the guard row a_{is}, when, $i \neq p, i \neq r$.*

Proof: From part (i) of theorem 3, we know;

$$a_{is.k+1} = \sum_{j \neq s}^{j \in \nabla} a_{ij.k+1}$$

By induction at the end of n^{th} phase,

$$a_{is.n} = \sum_{j \neq s}^{j \in \nabla} a_{ij.n}$$

At the end of n^{th} phase, we have already shown that the elements $a_{ij.n}$, where, $i \in \nabla, j \in \nabla, i \neq r, j \neq s$, represent the elements of the transpose of the inverse matrix, appropriately multiplied by the elements of the row matrix *B*. Thus, the guard row is equivalent to the 7th phase of the original computational procedure. Thus, this row provides the solution of the linear system.

Corollary 3*: Given a linear system AX=B, and a row matrix C, the computational procedure specified in the equations (A.1) to (A.10), at the end of n^{th} phase, computes the linear combination of the system variables CX at a_{rs}.*

Proof: From part (iii) of the theorem 3, we know,

$$a_{rs.k+1} = \sum_{j \neq s}^{j \in \nabla} \sum_{i \neq r}^{i \in \nabla} a_{ij.k+1}$$

From induction as before, we can say at the end of n^{th} phase,

$$a_{rs.n} = \sum_{j \neq s}^{j \in \nabla} \sum_{i \neq r}^{i \in \nabla} a_{ij.n}$$

Corollary 2, and 3 demonstrate the well-knitted compactness of the combined computational scheme. For the problem forms (ii) and (iii) the 7th of the original scheme (section 2.2) is no longer required. The results are automatically computed at the guard elements.

4.2 Error Detection Vector and Matrix

At the end of each phase, row and column wise aggregation provides two vectors called as *Error Detecting Vectors* (EDVx for row and EDVy for column). If (p,q) is the phase pivot then at the end of dual wavefront sum phase these vectors respectively appear as a row and column intersected at $\left(\left\lceil \frac{(p+n)}{2} \bmod (n+1) \right\rceil, \left\lceil \frac{(q+n)}{2} \bmod (n+1) \right\rceil\right)$ position. The collection of EDVs augmented over all the phases provide two matrices called as *Error Detecting Matrices* (EDMx and EDMy). No two EDV(i), where i is the phase index, appears on the same row and column. Thus, two floating variables per element can distributedly store the EDMs. EDMs contain the complete trace of error. EDM, based on the following spatio-temporal state model of matrix error propagation, is used to dynamically detect, analyze and correct the possible occurrences of errors inside the operating algorithm.

4.3 Error Propagation Model

Definition of Space: A space is defined as a tuple of *x* and *y* coordinates. The total computational space ∇ has all the rows and all the columns used in the above computation, which has a size of *(n+1)x(n+1)*. At any phase *k*, where $1 \leq k \leq n$, this space is divided into the following 4 tuple sub-spaces: (i) *guard space* ∇_s, (ii) the *pivot space* $\nabla_p(k)$, (iii) the *computed space* $\nabla_c(k)$, and (iv) the *uncomputed space* $\nabla_u(k)$. These are defined as follows:

Definition 1: *The guard space ∇_s is defined by the guard row and column, i.e., $a_{ij.k} \in \nabla_s$, if $i = r$ or $j = s$. ∇_s remains static through all the phases.*

Definition 2: *The pivot space $\nabla_p(k)$ is defined by the pivot row and column at k^{th} phase.*

Definition 3: *The computed space $\nabla_c(k)$ represents all the rows and columns, which has been selected as pivots. It is an incremental space which increases with phase. It is defined as, $\nabla(1) = \varnothing$, and $\nabla_c(k+1) = \nabla_c(k) \cup \nabla_p(k) \nabla_c(k)$*

Definition 4: *The uncomputed space $\nabla_u(k)$ represents the rows and columns (except the checksums) which has not been used as pivot row and column in past and present phases. It is an decremental space. It is defined as, $\nabla_u(1) = \nabla - \nabla_s$, and $\nabla_u(k+1) = \nabla_u(k) - \nabla_p(k)$.*

The total computational space is the intersection of all these 4 sub-spaces, or,

$$\nabla = \nabla_s \cup \nabla_p(k) \cup \nabla_c(k) \cup \nabla_u(k)$$

SPATIAL ERROR CLASSIFICATION

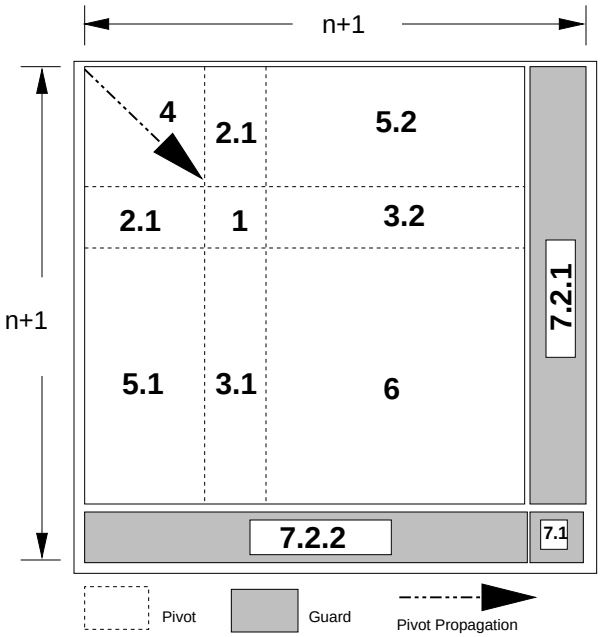

Fig-1

ERROR STATE TRANSITION MODEL

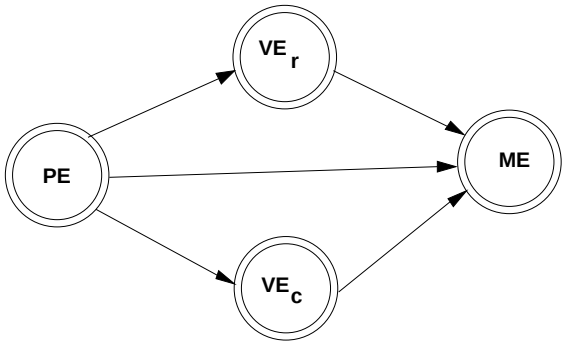

Fig-2

In the definition above, and the formalism below, $\nabla_c(k)$, or $\nabla_u(k)$ is not necessarily geometrically connected and contiguous. However, for simplicity of illustration Fig-1 assumes a principal forward diagonal pivot order (PFD) which makes them contiguous at any phase.

Error Classification: There can be 7 basic category of errors depending on their location of occurrence in the computational space defined above. Any possible single computational error falls into one of these classes. Below we describe these errors. Fig-1 shows their location.

Type 1: Error at $a_{ij.k}$, where $i \in \nabla_p(k)$, and $j \in \nabla_p(k)$. Such error results in non-zero error pattern at the intersection of the $\nabla_p(k)$.

Type 2: There are two symmetrical situations. Case 2.1 is where error occurs at $a_{ij.k}$, where $i \in \nabla_p(k)$, and $j \in \nabla_c(k)$. Case 2.2 is its symmetric situation.

FAULTY EDV PATTERNS

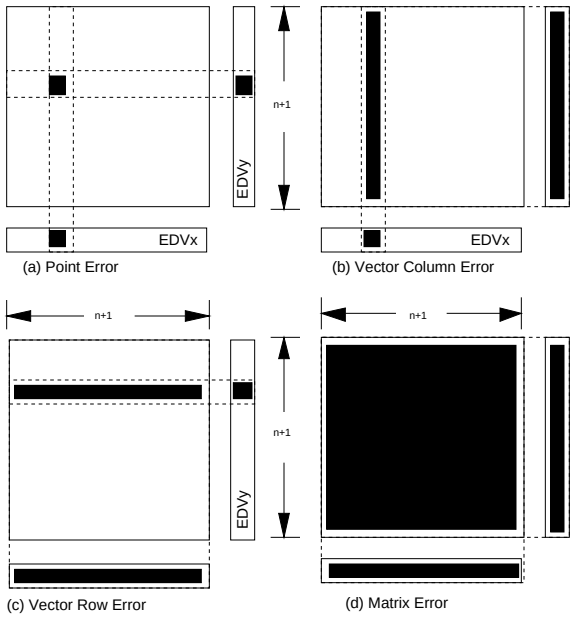

Fig-3

Type 3: This case also has two symmetrical situations. Case 3.1 is where error occurs at $a_{ij.k}$, where $i \in \nabla_p(k)$, and $j \in \nabla_u(k)$. Case 3.2 is its symmetric situation.

Type 4: When the error occurs at $a_{ij.k}$, where $i \in \nabla_c(k)$ and $j \in \nabla_c(k)$.

Type 5: Case 5 also has two symmetrical situations. Case 5.1 is where error occurs at $a_{ij.k}$, where $i \in \nabla_c(k)$, and $j \in \nabla_u(k)$. Case 5.2 is its symmetric situation.

Type 6: This type of error occurs at $a_{ij.k}$, where $i \in \nabla_u(k)$, and $j \in \nabla_u(k)$. This, has three sub-classes. In case 6.1, at some later phase, location *(i,j)* becomes the pivot position. In case 6.2.1, later on first *i* becomes the pivot row in subclass before *j* becomes the pivot column. Case 6.2.2 is the symmetric situation of 6.2.1 when only *j* becomes the pivot row before *i* becomes the pivot column.

Type 7: This type of error occurs at $a_{ij.k}$, where $i \in \nabla_s(k)$ and $j \in \nabla_s(k)$. This is an error on the guard row and column. This has three subclasses, Case 7.1 is where the error is on a_{rs}. Case 7.2.1 is where the error is on the guard column. Case 7.2.2 is where the error occurs on the guard row.

Out of these 7 classes of errors, catastrophic cancellation in finite precision arithmetic can cause errors of class 4, 5, 6 and 7. Because, only the updates in these classes involve addition or subtraction.

Metamorphosis of Error: A single computational error goes through the three distinct states: (i). *Point* (PE), (ii). *Vector* (VE) and (iii). *Matrix* (ME), with phase. Fig-2 shows the state transition diagram.

A single error originates as a PE. If the error occurs at $a_{ij.k}$, at k^{th} phase, then two of the checksum elements become non-zero like figure 5(a). A Point Error may propagate and expand into a VE if at some later phase *l*, the row (or column) corresponding to the element is selected as the pivot row (or column). Fig 5(b) and 5(c) show the EDV pattern corresponding to a VE. A PE can change into a ME in two ways, first, (i) if a point element is chosen as the pivot, or (ii). if a row (or column) of VE is chosen as pivot row (or column). Figure 5(d) shows the error pattern corresponding to the ME.

4.4 Error Identification

Each of the error states generates a distinct pattern in corresponding EDVs by which they can be precisely identified. Errors, of the first six classes can be identified at the originating phase in the PE. If the originating phase is k, then the EDV(k) will look like the Point Error Pattern.

An error of class 1, 2 and 4 remains in the point state for the rest of the computational phases. Because, both of its coordinates are in $\nabla_c(i)$, where $i > k$, for all future phases. Thus, for these classes of errors, all EDV(i), where $k \le i \le n$ resemble Fig-3(a).

An error of class 3 and 5 at phase *k* remains in PE state until a later phase *l*, when the element in error appears on the pivot row (or column). At this phase, it transforms into VE state. However, since after that $i, j \in \nabla_c(l+1)$. Therefore, for the remaining phases EDV(i), $i > l$ looks like Fig-3(b) or (c).

An error of Class 6 remains in the PE state until phase *l*, when it appears on the pivot row (or column). Just as before, it changes to VE state and remains in that state until a later phase *m*, $l \le m$, when the original point in error appears on the pivot column (or row). However, if $m = l$, then error of class 4 directly transforms to ME from PE state.

An error of Class 7 also follows similar transformations. However, since, an error element in guard column (or row) can only be part of a pivot row (or column), but not both, row and column, therefore, an error in this class never transforms into ME state. Such error is reflected as a single element error either on EDVx or on EDVy.

4.5 Error Correction

If an error is in PE state, it can be corrected at any phase i by adding the non-zero element of the EDVx(i) or EDVy(i) to the element. The element can be traced by the orthogonal propagation of two error markers initiated by the non-zero EDVx(i) and EDVy(i) nodes. Similarly, if an error is in VE state, if it is a row error, then it can be corrected by adding EDVx(i) to it. If it is a column error, then it can be corrected by adding EDVy(i) to it. However, once an error reaches ME state, it can not be corrected efficiently. In such a case, it is wise to recompute.

Errors of type 1, 2, 3, 5 and 7 never expands to ME state. Therefore, no in-phase correction is required for these classes. In the overlapped 7^{th} step these errors can be corrected directly. Only errors of class 6 require in-phase correction. However, the urgency of error correction, in such case depends on the phases *k, l,* and *m* during which it one by one transforms to PE, VE and finally into ME states. The error must be corrected before m^{th} phase. If we intend to correct class 6 errors, then at the detection of class 6 error an error correction wave should be initiated. This is generally straightforward, but costly.

The processors can partially detect the class of any error independently. (Complete class detection requires consultation between the EDVx EDVy nodes). Therefore, the processor(s) possessing non-zero checksum(s), can adopt a lazy policy of restraining in-phase error correction, unless the purity of computation is really threatened. Since, at any phase *k*, all the processors algorithmically come to know $\nabla_p(i)$ and $\nabla_c(i)$, where $i \le k$, therefore, they can detect whether it is in class 2, 5 or 6 by maintaining only two flags locally. Each of the processors, sets own *x* and *y* flags when it is selected as the pivot row or pivot column. If any of the flags is unset on the non-zero checksum element, then the corresponding error may fall into class 2, 5 or 6. In that case, it may initiate correction wave as a precautionary measure if rigorous error correction is intended.

4.6 Communication Structure

The incorporation of this fault-detection scheme potentially adds three types of communication costs. Below we show, how each of these is optimized.

(a) Data dependency: the data dependency of the equations A1..A10 can be satisfied using exactly the communication pattern of the original computation scheme, thus, it does not add any communication cost to the original algorithm.

(b) Detection wave: the dynamic IPC scheme verifies the checking property at the end of each computational phase by initiating a *checksum wave* after each phase This wave can be completely merged with the regular computational wave. In this technique, the vertical phase of the regular wave carries the column checksums and the subsequent horizontal phase carries the row checksums, along with the regular data elements. In double wave front communication, two partial sums propagate in the two directions. Thus, resulting IPC detection scheme remains almost transparent.

Conventional but less rigorous *end-of-phase* checking scheme requiring only one checksum wave at the last phase can also be used where *error correction* is not prime. This scheme squeezes the *phase width* by one floating point operation besides the bandwidth saving. This scheme, as shown by corollaries 2 and 3, fully overlaps the detection wave with the step 7 of the original procedure (section 2.2).

(c) Correction wave: Only type 6 error should be corrected within a deadline. Other classes of errors can be corrected at the end of computation using a lazy scheme. The cost of error correction is not generally critical to the overall performance of the algorithm. Because, such correction is occasional. As we have already shown, the cost of error detection, which must be incurred regularly irrespective of the occurrence of error, is almost negligible to our advantage.

The Double Wave Front Algorithm

```
int k=0, s=n, p_count=n;
int xphase[],yphase[],xx,yy;
getpid(i,j);

/* Loop for n phases*/
while(p_count) {
    px= xphase[k];
    py= yphase[k++];
    rr= .5*(px+n) mod (n+1);
    ss= .5*(py+n) mod (n+1);

    set_orientation(i, px,
      &vert_dst, &vert_src);
    set_orientation(j, py,
      &horz_dst, &horz_src);

/* If the Node Phase Pivot*/
    if(px=i and py=j) {
      if(v <= thres) {
        v= ABORT;
        xphase[k]=px;
        yphase[k++]=yx;
        p_count--;
        sum=v;
      }
      send(v,sum:south);
      send(v,sum:east);
      send(v,0:north);
      send(v,0:west);
      v=cb/v;
    }

/* If the Node is Pivot Row*/
    elseif(py==j) {
      recvb(p,sum:horz_src);
      sum=sum+v;
      if(p==ABORT) {
        xphase[k]=px;
        yphase[k++]=yx;
        v=ABORT;
        p_count--;
      }
      if(i==r) then v=v+c[py];
      v=v/p;
      send(v,sum:east);
      send(v,sum:west);
      send(p,sum:horz_dst);
      xx=k;
      if(i==r) v=v-b[py];
    }

/* If the Node is Pivot Column*/
    elseif(px==i) {
    sum=v;
    if(s==j) v=v-b[px];

    send(v,sum:south);
    send(v,0:north);
    recvb(p,sum:vert_src);
    sum=sum+v;
    if(p==ABORT) {
      xphase[k]=px;
      yphase[k++]=yx;
      p_count--;
    }
    send(p,sum:vert_dst);
    v=-v*c/p;
    if(s==j) v=v+c[px];
    yy=k;
  }

/*If the Node is on EDV*/
  else of (i==rr ||j==ss) {
    if (i==rr ) {
        recvb(l,s1:horz_src);
        recvb(l,s2:horz_src);
        x_sum[k]=s1+s1+v;
        if(l==ABORT) {
          xphase[k]=px;
          yphase[k++]=yx;
          p_count--;
        }
      if (j==ss ) {
          recvb(u,s1:vert_src);
          recvb(u,s2:vert_src);
        y_sum[k]=s1+s2+v;
      }
      if(y_sum[k]||x_sum[k])
        error();
      v=v-u*l;
      }

/* For Other Non EDV Nodes*/
    else {
      recvb(l,s:horz_src);
      s=s+v;
      if(u==ABORT) {
        xphase[s]=px;
        yphase[s++]=yx;
        p_count--;
      }
      send(l,s:horz_dst);
      recvb(u,s:vert_src);
      s=s+v;
      s]end(u,s:vert_dst);
     if(i==r & j==s)
      v=v-u*l;
    }
    if(i==r || j==s) v=-v;
}
```

Fig-4

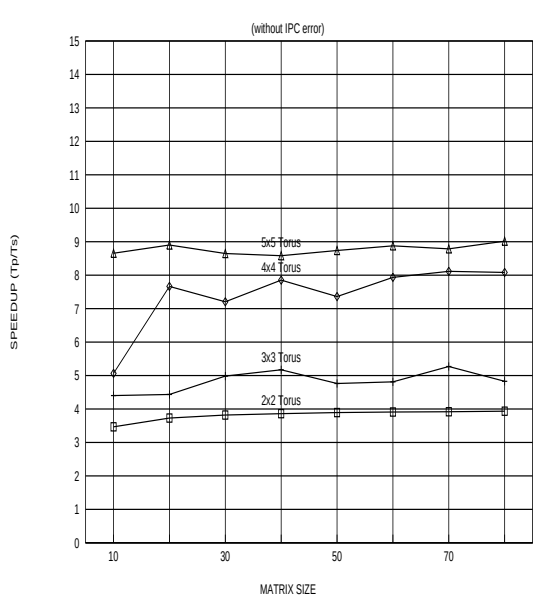

Fig-5

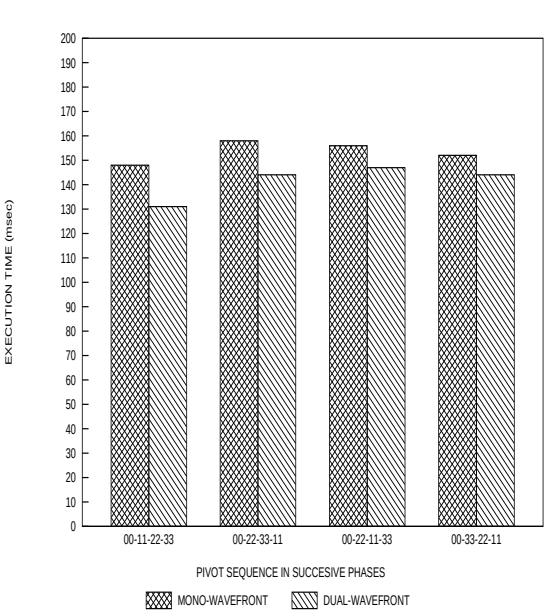

Fig-6

5. THE ALGORITHM

The algorithm generates a series of successive wavefronts. Each originates at a phase pivot and then propagates in all four directions. As shown in Fig-2, within each phase, each of the nodes transmits message packets containing **U/V** and **L**, and **S** (checksum) to its neighbors and performs update specified in equations A1..A10.

Adaptive Pivoting: If there is any zero pivot, it is skipped and the phase computation proceeds from the next pivot position chosen according to a *default order*. Generally, any *diagonal order* is recommended as default, because it has the minimum *inter phase* initiation gap (which is more critical than the *phase-width* for the performance of wave algorithms).

Error Tracking: In case of finite precision floating point error, if a node with non-zero EDV detects an error, then it starts tracking the error and its state transition. This tracking is completely local because eventually every node algorithmically receives the location of the pivot. If the row (or column) of the error sensing element has been

selected as a pivot row (or column) then it is in ∇_c. Therefore, it waits till the end phase. On, the other hand, if it is in ∇_u, then it initiates necessary communication for error correction. However, the error correction wave lags the computation wave by one phase.(It is possible to rearrange the phase computations so that it will lag by half phase). Therefore, when the error is of class 6.1, and *k+1=l=m*, then error cannot be corrected. (A modified scheme can correct such errors, but it requires rollback).

Pseudo-Code: Fig-4 presents a pseudo code (like C) version of the algorithm. At the beginning of each phase, the **set_orientation()** routine finds the vertical and horizontal source and destination neighbors for each node depending on their own relative position w.r.t. the current pivot. Since, several pivot selection strategies can be used for *adaptive pivoting*, we have assumed **xphase[]** and **yphase[]** arrays are supplying the pivot positions. For solving problems (ii) and (iii) a sum phase should be added at the end. The final identities of the elements are in local variables *(xx,yy)*. If a non-zero sum is detected, then the error correction scheme is initiated by the abstract routine **error()**.

6. PERFORMANCE

6.1 Complexity Analysis

If the matrix size in nxn and the torus space is pxp, then, using block decomposition each of the processors gets a mxm block matrix, and assuming the total communication time to be $t_{comm} = t_{send} + t_{channel} + t_{recv}$, i.e., it is the sum of packing, channel and unpacking time, then the total parallel time becomes:

$$T_{parl} = (p-1)\{m(m^2 t_{\text{comp}} + 4t_{send}) + 2(t_{comm} - t_{send})\} + (p-1)t_{comm}$$

In the maximally parallel decomposition with 1:1 *folding ratio*, with p=n+1, and m=1, on (n+1)x(n+1) torus,

$$T_{parl} = n(t_{\text{comp}} + 2t_{send} + 3t_{comm})$$

This performance is better than the algorithms based on i. Gauss-Jordan, ii. Gauss-Seidel and iii. LU decomposition for each of the three problem forms [9]. On the basis of above equation, when $p \gg 1$, the optimum *folding ratio* 1: m_{opt} is given by:

$$m_{opt} = \left\{ \frac{3t_{comm} - 2t_{send}}{2t_{\text{comp}}} \right\}^{\frac{1}{3}}$$

6.2 Performance on Meiko

We have implemented the dual wavefront algorithm with flexible order pivoting and tested its performance on a 32 node Meiko Transputer. Fig-5 shows the relative speedup with the variation of the matrix size for four different torus configurations. We varied the matrix size from 10x10 to 80x80 and used block decomposition. As shown by the overall flat nature of the curves, the implementation demonstrated sustained scalability.

Fig-6 shows the effect of *adaptive pivoting* on execution cost. According to the AP scheme, the pivots are selected at run time. However, for demonstration purpose, in this experiment, we have used 4 different pre-defined pivot sequences (marked at x-axis) and observed the computation time for both mono and dual wavefront versions of the algorithm. The experiment indicates that the occasional disruption from the PFD, due to adaptive pivoting will have negligible effect on the overall system performance. Fig-6 also shows the improvement due to double wavefront communication over usual single wavefront communication.

7. CONCLUSION

Adaptive pivoting replaces the classical row interchange for improving numerical stability. This new pivoting can effectively guard against zero pivot. However, guarding against small pivots still requires additional O(*n*log*n*) parallel comparison steps similar to other classical pivoting schemes. Nevertheless, our *adaptive pivoting* saves the traditional interchange required after the sorting. The theoretical effectiveness of AP is equivalent to that of *full or partial pivoting* [11,3], but is more parallel.

IPC is a corrective approach to stability. One of the key unresolved issue is whether such corrective measure can outweigh the preventive methods (such as LU decomposition). The dynamic nature of IPC makes the direct comparison extremely hard.

8. REFERENCE

[1] Faddeeva, V. N., *Computational Methods of Linear Algebra*, Chapter 2, Translated by C. D. Benster, Dover Pub., New York 1959.

[2] Gill, P. E., W. Murray & M. H. Wright, *Numerical Linear Algebra and Optimization*, v. 1, Chapter 3 & 4, Addison-Wesley Publishing Company, California 1991.

[3] Golub G. H. & C. F. V. Loan, *Matrix Computations*, The Johns Hopkins University Press, Maryland, 1985.

[4] Heller, D., "A Survey of Parallel Algorithms In Numerical Linear Algebra", *SIAM Review*, Vol 20, no 4, October 1978.

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

[6] Kant, R. M. & T. Kimura, "Decentralized Parallel Algorithms for Matrix Computation", *Proc. 5th Annual Symposium of Com. Arch.*, pp96-100, 1978.

[7] Khan, J. I., W. Lin & D. Y. Y. Yun, "A Parallel Matrix Inversion Algorithm on Torus with Adaptive Pivoting", *Proc. 21st International Conference on Parallel Processing*, Chicago, August 1992, v-III, pp69-72.

[8] Lin W.,T. L. Sheu & J. I. Khan, "A Parallel Fault-Detection Scheme for Matrix Inversion", *Proc. ISMM Int. Conf. on Parallel and Distributed Computing and Systems*, Pittsburgh, October 1992, pp394-398.

[9] Modi, J.J., *Parallel Algorithms and Matrix Computation*, Oxford University Press, Oxford, 1988.

[10] O'Leary, D. W. & G. W. Stewart, "Data Flow Algorithms for Parallel Matrix Computations", *Communications of the ACM*, V. 28, no.8, pp.840-853, August 1985.

[11] Wilkinson, J. H., "Error Analysis of Direct Methods of Matrix Inversion", *J. Assoc. Comp*. Mach. 8, p281-330, 1961.