FAULT-TOLERANT PARALLEL MATRIX MULTIPLICATION WITH ONE ITERATION FAULT DETECTION LATENCY†

Chul-Eui Hong and Bruce M. McMillin

CSC-91-06

September 15, 1991

Department of Computer Science
University of Missouri at Rolla
Rolla, Missouri 65401
ff@cs.umr.edu

This paper appears, in its entirety, in the Proceedings of the 15th International COMP-SAC, September, 1991, pp. 665-672

† This work was supported in part by the National Science Foundation under Grant Numbers MIP-8909749 and CDA-8820714, and in part by the AMOCO Faculty Development Program.
Abstract: The checksum technique is a low cost method to detect errors in matrix operations performed by processor arrays. The fault detection of this method is done only at problem termination, so this method is not an effective fault tolerance technique for large scale matrix multiplication.

This paper presents a new algorithm, the ID algorithm, which minimizes the fault-detection latency. In the ID algorithm, a fault is detected as soon as the fault occurs instead of at problem termination. For \( n^2 \) processors, the fault-latency time of the ID algorithm is \( \frac{1}{n} \) of that of checksum algorithm with a run-time penalty of \( O(n \log n) \) in an \( n \times n \) matrix operation. This new algorithm has better performance in terms of error coverage and expected run time in large scale matrix multiplications such as signal and image processing, weather prediction, and finite element analysis.

KEY WORDS: application-oriented fault tolerance, multicomputers.

I. Introduction

Multicomputers have gained much attention recently due to their unique scalability in providing massive computing power. Various multicomputers, such as the Ncube and the Intel iPSC, are notable examples of commercially available multicomputers [AtSe88]. Often, the core of computational algorithms running on multicomputers consists of matrix operations such as multiplication of matrices and vectors, inversion or LU decomposition, and the determination of eigenvalues and eigenvectors. Design of parallel algorithms for matrix operations on multicomputers has been a major research issue in the application domain. However, the design of fault-tolerant parallel algorithms has received little attention. As the size of multicomputers grows into thousands of processors and the corresponding scale of attempted application problems grows even faster, ensuring the correctness of a parallel algorithm’s computation becomes a necessary step.

Attempts to provide fault tolerance for this environment through traditional reliability techniques such as triple modular redundancy or self-checking processors will be prohibitively expensive in a large multicomputer due to the sheer bulk of hardware overhead. Furthermore, overall hardware confidence decreases as the number of components rises. Thus, we must turn to other techniques.

[HuAb84] introduced a different view of fault tolerance for parallel processing, algorithm-based fault tolerance. Processors involved in a computation provide fault tolerance to that application; faults that directly affect the computation of the problem will be detected using some property of the problem. Thus, in this approach, testing is performed at a peer level, i.e. processes involved in a calculation check other processors involved in different parts of the same calculations. The difficulty of such an idea involves the question of how to find the appropriate tests for a multicomputer. There are two major schools of thought.

In [HuAb84], a checksum matrix multiplication algorithm is presented as a low-cost technique to detect errors in matrix operations performed by processor arrays. The checksum method requires only small redundancy ratios. However, In the checksum algorithm, the fault detection latency time is the same as the fault-free run time. Because more errors are likely to happen during longer computations than during short computations, for long computations it is important to detect faults early and to reconfigure around the faulty unit. So lessening the expected run time by lessening the fault detection latency time is crucial in very large scale computation.

The application-oriented fault tolerance paradigm [Mcmi90] is another approach to providing this necessary fault tolerance for the multicomputer environment. The actual test is comprised of executable assertions [Stuc77] which are embedded in the program code to ensure that at each testable stage of a calculation, all tested processors conform to the program’s specification, design, and implementation. If a difference between this expected and observed behavior is detected, then a fault has occurred and a reliable communication of this diagnostic information is provided to the system so that appropriate actions may be taken.

Executable assertion development for a program in the multicomputer environment is complicated by several multicomputer specific issues. In a multicomputer, all processors (nodes) are autonomous with their own private local memory. The nodes
are interconnected by an interconnection mechanism. There may or may not be a host processor for program downloading and data downloading/collection. Message passing is usually the only means to allow interprocessor communication, which takes non-negligible time. This limits the scope of the test that may be performed by an executable assertion to testing received messages with respect to a testing node’s local state. Thus, code or design based assertions may not be possible to implement in such an environment.

To overcome the problem of assertion development for the multicore environment, the authors proposed the constraint predicate (Φ) paradigm [Mcm90]. In the constraint predicate paradigm, assertion development begins with the specification phase of the software life cycle. Three basis metrics of progress (Φ P), feasibility (Φ F), and consistency (Φ C) are used to generate assertions which 1) preserve the global nature of the application paradigm, and compare/contrast its performance with the base algorithm. Section III surveys past work in algorithm-based fault tolerance for matrix operations. In Section IV, we develop a new fault-tolerant algorithm, ID. Section V analyzes the checksum and newly developed algorithms by run-time performance and fault coverage.

II. GENTLEMAN’S ALGORITHM

[Gent78] has shown that multiplication of two n x n matrices on the single instruction multiple data (SIMD) with wraparound mesh connections model requires lower bound Θ(n) data routing steps [Quin87]. Suppose that two n x n matrices A and B are to be multiplied, and assume that every element of A and B is stored exactly once and that no processing element contains more than one element of either matrix. In the initial allocation, only n diagonal processing elements contain a pair that needs to be multiplied. However, it is possible for n^2 processing elements to achieve Θ(n) time by staggering matrices A and B by the initial step so that at every step of algorithm an upward rotation of the elements in B and a leftward rotation of the elements of A presents each processing element with a pair of values to be multiplied (main loop in Figure 2-1). Data in Figure 2-1 represent elements after the initial step.

Both of the algorithms in this paper use Gentleman’s algorithm in the hypercube as a basis for fault-tolerant algorithm development. Gentleman’s algorithm can be easily implemented in a hypercube by embedding the wraparound mesh via binary reflected gray codes.

III. CHARACTERISTICS OF CHECKSUM ALGORITHM

In a faulty multicore, the following assumptions hold in the development of a fault-tolerant algorithm.

1) Inter-node communications and processors are subject to arbitrary faults.

foreach P_i,j SEND and RECEIVE to i/j First Iteration */
left circular shift all a_{i,j}^{(0)}’s by i – 1
up circular shift all b_{i,j}^{(0)}’s by j – 1
foreach P_i,j */Second Iteration */
c_{i,j} ← a_{i,j}^{(1)} b_{i,j}^{(1)}
for k = 1 to n-1

left circular shift a_{i,j}^{(k)} up circular shift b_{i,j}^{(k)}

Figure 2-1. Data communication patterns in Gentleman’s Algorithm and data distribution after Skewing. The superscript (0) on a_{i,j} indicates that a_{i,j} was located in processor P_{i,j} at iteration zero, i.e. at the start of the algorithm.

2) The host processor is reliable as are the links from the host processor to each node.
3) Message transmission is over point-to-point links and no atomic broadcast exists.
4) The absence of a message can be detected and constitutes an error.
5) All nodes are non-faulty at initiation of the algorithm and remain non-faulty through the first message exchange.

Since there is no central point through which all data will pass or had passed, there is no opportunity for centralized fault diagnosis; the fault detection problem must be solved in a distributed manner.

The checksum algorithm imposes an additional structure on the matrix multiplication algorithm much that same way a checksum encoding imposes an additional structure on a communications codeword.

In the encoding scheme of [HuAb84] the row, column and full checksum matrices for an n x m matrix A (denoted by A_{n,m}) with elements a_{i,j} are denoted as follows.

Definition 3-1: The row checksum matrix A_r of the matrix A is an n by m+1 matrix consisting of the matrix A in the first m columns and a row check vector in the (m+1)st column; the elements of the check vector are generated as
\[ a_{i,j}^{n+1} = \sum_{j=1}^{n} a_{i,j} \quad \text{for } 1 \leq i \leq n. \]

**Definition 3-2:** The column checksum matrix \( A_i \) of the matrix \( A \) is an \( n+1 \) by \( m \) matrix which consists of the matrix \( A \) in the first \( n \) rows and a column check vector in the \((n+1)\)st row; the elements of the check vector are generated as
\[ a_{m+1,j} = \sum_{i=1}^{n} a_{i,j} \quad \text{for } 1 \leq j \leq m. \]

**Definition 3-3:** The full checksum matrix \( A_f \) of the matrix \( A \) is an \( n+1 \) by \( m+1 \) matrix which is the column checksum matrix of the row checksum matrix \( A_r \).

There are five matrix operations which preserve the checksum property, matrix addition, multiplication, scalar product, LU-decomposition and transpose. Figure 3-1 depicts the column and row checksum matrix relationship.

**Theorem 3-1** [HuAb84]: i)The result of a column checksum matrix \( A_i \) multiplied by a row checksum matrix \( B_j \) is a full checksum matrix \( C_{ij} \). ii)Their corresponding information matrices \( A, B \) and \( C \) have the following relation (Figure 3-1):
\[
A \times B = C
\]

**Theorem 3-2:** The checksum encoding scheme is not valid during the matrix multiplication step \( i \) where \( i < n \) in \( n \times n \) square matrix.

**Proof:** We can prove Theorem 3-2 by counter example. In Figure 2-1, node \( p_{0,3} \) is the column checksum element. After the first iteration, the sum of the result matrix of row 0,
\[
\sum_{j=0}^{n-1} c_{0,j}^{(1)} = a_{0,0}b_{0,0} + a_{0,1}b_{1,1} + a_{0,2}b_{2,2}
\]
must be equal to the Checksum element,
\[
c_{0,3}^{(1)} = a_{0,0}b_{0,0} + a_{0,1}b_{1,1} + a_{0,2}b_{2,2} + c_{0,3}^{(0)}
\]

But this Checksum encoding is not valid. \( \square \)

**3.2 Run-Time Costs**

The expected run time for the checksum algorithm grows with the number of processors in a long matrix computation. Let \( T \) be the fault-free run time of the algorithm (the time to run the algorithm if no errors occur). If the system does not fail before time \( T \), then the job runs in time \( T \). If the system fails before time \( T \), then the job must be restarted and the expected run time is the time spent before the failure plus another expected run time.

The expected run time,
\[
E(R) = TP_f[Y > T] + (E(R) + T)(1 - P_f[Y > T]) = TE^{(n/\mu)}T
\]
where cdf of $P_i[Y>T]$ is $G(t) = e^{-(\mu_0 t^2)}$ and $n$ is the number of processors and $\mu_0$ is the individual MTTF of a processor.

The derivation of (3-1) is in [HoMc91]. The general shape of (3-1) is given in Figure 3-3.

From the above analysis, the checksum algorithm is same as the unreliable algorithm in terms of the expected performance evaluation because a single processor failure cause a job to restart.

IV. A Better Fault-Tolerant Algorithm

In this section, we re-examine providing fault tolerance for the problem of matrix multiplication in an attempt to (1) improve the fault detection latency, (2) locate the fault, and (3) use the problem specification itself to generate a fault-tolerant algorithm. The tool we use is the application-oriented fault tolerance paradigm [McMi90].

4.1. General application-oriented fault tolerant constraint predicates

The application-oriented fault tolerance paradigm relies on properties of expected algorithm behavior to form a test for faulty behavior thus also detecting software failures. The application-oriented fault tolerance paradigm is a promising approach to providing this necessary fault tolerance for the multicomputer environment [McNi89].

The application-oriented paradigm utilizes the following three basis metrics of progress, feasibility and consistency to generate assertions which (1) preserve the goal nature of the problem and 2) can be implemented locally in each node. The collection of such assertions forms the constraint predicate.

1) Progress Test($\Phi_P$): The progress component ensures that algorithm termination will occur and that at each testable step of the solution, the state of the solution advances to the goal or final solution of the problem. This test guarantees that the algorithm works well and detects software and hardware faults.

2) Feasibility Test($\Phi_F$): Each testable result must remain within the defined solution space of the problem. In general this solution space corresponds to a problem’s Natural Constraints such as boundary conditions or, to a limited extent, range checking.

3) Consistency Test($\Phi_C$): Many intermediate calculations contain additional properties that are indirectly obtained from the problem’s natural constraints. To achieve consistency, we require that each processor “hears” the same version of message using vertex disjoint paths so that a few faulty processors or links cannot destroy the intended content of a message. A processor receiving a message makes a consistency test based on the content of the message compared with the processor’s perceived current state of the entire system.

4.2. The ID algorithm

We introduce several new data structures into Gentleman’s algorithm [Gent78] to implement application-oriented fault tolerance.

Definition 4-1: Each node $P_{i,j}$ maintains two vectors $LMA_i$ and $LMB_j$. They are the row and column elements of matrices $A$ and $B$ respectively. $LMA_i$ and $LMB_j$ are formed in the initial stage to ensure that all elements of matrices $A$ and $B$ remain correct and consistent during run time.

$LMA_i = \{a_{i,m}^{(1)} | 0 \leq m \leq n - 1\}$ in node $P_{i,j}$

$LMB_j = \{b_{m,j}^{(1)} | 0 \leq m \leq n - 1\}$ in node $P_{i,j}$

The superscript $(\cdot)$ denotes the current iteration count in matrix multiplication, that is, the element $a_{i,j}^{(k)}$ is in node $P_{i,j}$ in the $k^{th}$ iteration for $k = 1, \ldots, n$ and $k = 0$ is before the initial staggering step.

Definition 4-2: Each node $P_{i,j}$ maintains $Sa_{i,j}$ and $Sb_{i,j}$ which are the sum of row or column elements of matrices $A$ and $B$ respectively. $Sa_{i,j}$ and $Sb_{i,j}$ shift upward and leftward at every iteration.

$Sa_{i,j}^{(k)} = \sum_{m=1}^{k} a_{(i,k-m)\%n,j}\%(m-1)\%n$

$Sb_{i,j}^{(k)} = \sum_{m=1}^{k} b_{(j,k-m)\%n}^{(1)\%m-1}\%n$

Example: $Sa_{1,1}^{(3)} = a_{1,1}^{(1)} + a_{1,2}^{(1)} + a_{2,2}^{(1)} + a_{3,2}^{(1)}$

$Sb_{1,1}^{(3)} = b_{1,1}^{(1)} + b_{1,2}^{(1)} + b_{2,2}^{(1)} + b_{1,4}^{(1)}$

We assume, in this paper, $LMA_i$ and $LMB_j$ are considered reliable vectors in each $P_{i,j}$. This is not an overly restrictive assumption, however, as a fault-tolerant all-partial-sums calculation is easy to implement [McNi89].

![Figure 3-1. Wrap-around Data Movement in Fault-Tolerant Algorithm ID for processor $P_{i,j}$. In the ID algorithm, Sa, Tb, and $a_{(i,k-m)\%n,j}^{(1)}$ are shifting upward and Sh,Ta, and $b_{(j,k-m)\%n}^{(1)}$ shift leftward at every $k$ iteration.](image)
Example: \( T_{a(1)}^{(k)} = a_{i,j}^{(k)} \) ( \( b_{i,j}^{(1)} + b_{i,j}^{(2)} + b_{i,j}^{(3)} + b_{i,j}^{(4)} \) )

\( T_{b(1)}^{(k)} = b_{i,j}^{(1)} \) ( \( a_{i,j}^{(k)} + a_{i,j}^{(2)} + a_{i,j}^{(3)} + a_{i,j}^{(4)} \) )

### 4.3. Predicate Generation

In this sub-section, we will present application-oriented fault tolerance predicates of the ID algorithm. These predicates form the assertions to provide concurrent fault tolerance in the ID algorithm.

Following the vertex disjoint communication paths of Sa and Sb (upward and leftward respectively), the following relation holds on Sa’s and Sb’s at every iteration.

**Lemma 4-1: \( \Phi_{P} \)** The following assertions hold at each iteration \( k \):

To verify \( S_{a(1)}^{(k)} \):

- at node \( P_{i,j}(1-n)/n,j \):
  \( \text{Sa}_{i,j}^{(k)} + b_{i,j}^{(1)} = \text{Sa}_{i}^{(k)}(1-n)/j+1)\)

- at node \( P_{i,j}(n)/n,j \):
  \( \text{Sa}_{i,j}^{(k)} + b_{i,j}^{(1)} = \text{Sa}_{i}^{(k)}(1-n)/j+1)\)

To verify \( S_{b(1)}^{(k)} \):

- at node \( P_{i,j}(1-n)/n,j \):
  \( \text{Sa}_{i,j}^{(k)} + a_{i,j}^{(1)} = \text{Sa}_{i}^{(k)}(1-n)/j+1)\)

- at node \( P_{i,j}(n)/n,j \):
  \( \text{Sa}_{i,j}^{(k)} + a_{i,j}^{(1)} = \text{Sa}_{i}^{(k)}(1-n)/j+1)\)

**Proof:** The proof is in [HoMc91].

The example is for an interior node instead of one at the wraparound boundaries for simplicity.

**ex) To verify \( S_{b(1)}^{(k)} \) to be correct, at the processor \( P_{0,1} \): \( \text{Sa}_{1,1}^{(k)} + b_{0,1}^{(1)} = \text{Sa}_{1}^{(k)}(1-n)/j+1)\)**

so, \( b_{0,1}^{(1)} + b_{1,1}^{(1)} + b_{2,1}^{(1)} + b_{3,1}^{(1)} = b_{0,1}^{(1)} + b_{1,1}^{(1)} + b_{2,1}^{(1)} \)

at the processor \( P_{1,0} \): \( \text{Sa}_{1,0}^{(k)} + b_{1,0}^{(1)} = \text{Sa}_{1}^{(k)}(1-n)/j+1)\)

so, \( b_{1,0}^{(1)} + b_{1,0}^{(1)} + b_{1,0}^{(1)} = b_{1,0}^{(1)} + b_{1,0}^{(1)} + b_{1,0}^{(1)} \)

By the feasibility test, we can guarantee that result matrix C is always in the correct range at every iteration.

**Lemma 4-2: \( \Phi_{P} \)** At every iteration, \( k \), the sum of \( Ta^{(k)} \) and the result matrix \( C \) in the same row are equal and the sum of \( Tb^{(k)} \) and the result matrix \( C \) in the same column are equal.

\[
\sum_{j=0}^{n-1} T_{a(1)}^{(k)} = \sum_{j=0}^{n-1} c_{1,j}^{(k)} \quad \text{at every iteration } k
\]

\[
\sum_{j=0}^{n-1} T_{b(1)}^{(k)} = \sum_{j=0}^{n-1} c_{j,1}^{(k)} \quad \text{at every iteration } k
\]

**Proof:** From Definition 4-3, it is obvious that Lemma 4-2 is preserved at every iteration because Ta and Tb is shifting left and upward at every iteration respectively and result elements \( c_{i,j} \) remain in the local node \( P_{i,j} \).

By the progress test \( \Phi_{P} \), we can guarantee that correct elements of matrix A and B are multiplied from LMA and LMB. The number of iterations is known \textit{a priori} to all processors and all elements of LMA and LMB must be multiplied sequentially only once.

**Lemma 4-3: \( \Phi_{P} \)** each element of matrix A, \( a_{i,j}^{(k)} = \frac{T_{a(1)}^{(k)}}{Sa_{i,j}^{(k)}} \) and each element of matrix B, \( b_{i,j}^{(k)} = \frac{T_{b(1)}^{(k)}}{Sa_{i,j}^{(k)}} \) at every iteration at each node.

**Proof:** From Definition 4-3, Lemma 4-3 is obvious.

### 4.4. Development of the ID algorithm

Since \( Ta \) and \( Tb \) are circuited in a row and a column at every iteration respectively, unlike the checksum algorithm (Theorem 3-2), in the ID algorithm the partial portion of the checksum vectors such as Ta and Tb can be summed and compared with the sum of the result matrix C at every iteration by a row and a column respectively.

**Theorem 4-1:** Some processor \( P_{i,j} \) running an algorithm using the developed constraint predicate \( \Phi \) can detect an error in the result matrix C immediately if an undetectable syndrome does not arise.

**Proof:** By Lemma 4-1, Sa’s and Sb’s in each node are guaranteed to be correct at every iteration except the undetectable syndrome which will be explained in Section V. If Sa’s and Sb’s are correct in each node at every iteration, then by Lemma 4-3: \( \text{Fa}_{i,j} = a*Sa \) and \( Tb = b*Sa \). Ta and Tb are correct in each node at every iteration since we can guarantee that elements of matrix A and B, \( a_{i,j} \), and \( b_{i,j} \), are correct in each node \( P_{i,j} \) at every iteration from LMA and LMB. And if Ta’s and Tb’s are correct in each node at every iteration, then by Lemma 4-2: \( \sum_{j=0}^{n-1} \sum_{j=0}^{n-1} C = \sum_{j=0}^{n-1} \sum_{j=0}^{n-1} \text{Ta} \) and \( \sum_{j=0}^{n-1} \sum_{j=0}^{n-1} C = \sum_{j=0}^{n-1} \sum_{j=0}^{n-1} \text{Ta} \) and \( \sum_{j=0}^{n-1} \sum_{j=0}^{n-1} \text{Ta} \) and \( \sum_{j=0}^{n-1} \sum_{j=0}^{n-1} \text{Ta} \) and \( \sum_{j=0}^{n-1} \sum_{j=0}^{n-1} \sum_{j=0}^{n-1} Tb \), the fault of the result matrix C is detected at every iteration.

However, the ID algorithm is not complete in fault detection under an arbitrary number of faults. This undetectable syndrome will be discussed in Section V.

### V. Performance of the ID Algorithm

The effectiveness of any fault-tolerant algorithm is measured by (1) fault coverage, (2) comparison with sequentially executed performance, and (3) comparison with other fault-tolerant methods. In the following discussion, denotes the fault-free run time of the checksum algorithm, denotes the fault-free run time of the ID algorithm, denotes the fault-free run time of a single iteration of the ID algorithm (\( T_{1} = T_{1}/n \)).

#### 5.1. Undetectable Syndromes of the ID algorithm

The ID algorithm is not complete in fault detection. In this subsection, we will analyze the probabilities of undetectable syndromes.

**Definition 5-1**: Let the event \( TRI \) be that all faulty nodes form a right triangle and the vertices are adjacent. All right triangle nodes are faulty at the same iteration (Figure 5-1).

**Theorem 5-1**: If the event \( TRI \) occurs, we may not detect a fault in Sa’s and Sb’s.

**Proof:** Immediate from Lemma 4-1 and Figure 4-1. If at least one of 3 vertices in \( TRI \) is fault-free, we can detect a fault because one faulty node is tested by the other two nodes.
The proof is obvious from Lemma 5-1.

Case 2 > \( P_r[M - LOOP]\text{only 5 nodes are faulty}] = 0 \)

Case 3 > \( P_r[M - LOOP]\text{only 6 nodes are faulty}] (Figure 5-1) \)

\[ = \frac{1}{3} \binom{n}{2} \cdot n(n-1)^2(n-2) \cdot p^6 q^r^2 - 6 \]

Again, the probability of more than 6 nodes becoming faulty during the same iteration is negligible.

\[ P_r[M - LOOP]\text{Y < T_1}] \]

\[ = \binom{n}{2} p^3 q^r^2 + \frac{1}{2} \frac{n(n-1)^2}{n} (n-2) \cdot p^6 q^r^2 - 6 \]

The probability of an undetectable syndrome occurring during 1 iteration(\( T_1 \)) is obtained by adding the probabilities of event TRI and event M-LOOP.

\[ P_r[Undetectable syndrome]\text{Y < T_1}] \]

\[ = \binom{n}{2} p^3 q^r^2 + \frac{1}{2} \frac{n(n-1)^2}{n} (n-2) \cdot p^6 q^r^2 - 6 \]

\<note> We ignore the common probability of event TRI and event M-LOOP for simplicity which, in the case of 6 faulty nodes, is negligible. \</note>

5.2. Expected Error Coverage

Let \( p \) be the probability of fault in any individual processor during 1 iteration, and assume \( p \) is constant between the two candidate algorithms so we can have the same number of faulty processors in the ID algorithm and the checksum algorithm. With above assumption, we can calculate the expected error coverage.

Lemma 5-1: The ID Algorithm can detect faults in a single iteration \( T_1 \) with probability

\[ P_r[\text{detectable}\text{ Y < T_1}] = 1 - P_r[Y > T] + P_r[\text{undetectable}\text{ Y < T_1}] \]

\textbf{Proof:} The probability of a fault occurring in the system during one iteration(\( T_1 \)) is

\[ P_r[\text{fault - free for } T_1] = P_r[Y > T_1] = (1-p)^n = q^r \]

\text{(5-1)}

The proof is obvious. \( \square \)

\textbf{Theorem 5-3:} The probability of detecting a fault in the system over all iterations taking fault-free run-time \( T_i \) is

\[ P_r[\text{detectable}\text{ Y < T_i}] \]

\[ = 1 - (P_r[\text{fault - free for } T_i] + P_r[\text{undetectable}\text{ Y < T_i}]) \]

\textbf{Proof:} The proof is obvious from Lemma 5-1. \( \square \)

\textbf{Theorem 5-4:} The Checksum Algorithm can detect faults with probability

\[ P_r[\text{detectable}\text{ Y < T_i}] \]

\[ = 1 - (P_r[Y > T_i] + P_r[\text{undetectable}\text{ Y < T_i}]) \]

\textbf{Proof:} In \( n \times n \) matrix multiplication, we need \( (n+1) \times (n+1) \) processors and \( n+1 \) iterations because extra checksum vectors are added. Any fault in any processor during any iteration can affect the result. Let \( p \), the probability of fault in any individual processor during 1 iteration, be constant, so the checksum algorithm can have a same number of faults as the ID algorithm.
Let $P_{ch}$ be the probability of a fault at an individual processor over all iterations ($T_c$)

$$P_{ch} = 1 - \prod_{i=1}^{n+1}(1-p) = 1 - (1-p)^{n+1}$$

There are $(n+1)^2$ processors. We apply r-n system and assume that checksum algorithm can detect up to 3 faults.

$$P_{ID}[\text{undetectable} \mid Y < T_i] = \frac{(n+1)^2n^2}{4} P_{ch}q_{ch}^{(n+1)^2-4} + \frac{(n+1)^2n^2(n-1)}{6} P_{ch}q_{ch}^{(n+1)^2-6}$$

$$P_{ID}[\text{detectable} \mid Y < T_i] = 1 - (P_{ID}[Y > T_i] + P_{ID}[\text{undetectable} \mid Y < T_i]) \quad \square$$

As shown in the Figures 5-3 and 5-4, the ID algorithm is better than the checksum algorithm in the expected error coverage when the number of processors are large and the failure rate of any individual processor is high.

![Figure 5-3. Undetectable Syndrome Crossover Points for the ID algorithm by Fixed Probability of node failure $p = 10^{-8}$](image)

![Figure 5-4. Undetectability by Probability of Processor Failure.](image)

### 5.3. Run time overhead estimation

Following the discussion in Section III, we consider the expected run time of both the ID and checksum algorithms.

For this analysis, we relax the assumption the probability of a fault of an individual processor is constant between the two algorithms.

**Theorem 5-5:** The Checksum Algorithm has an expected run-time,

$$E(R) = T_c \cdot e^{-\frac{\lambda}{\mu}}$$

**Proof:** Since fault detection is only done at problem termination, the run time is unaffected by the number of fault occurrences. Thus, when a failure is detected, it is always detected at time $T_c$.

$$E(R) = T_c \cdot P_{ID}[Y > T_i] + (E(R) + T_c) \cdot P_{ID}[Y < T_i] \quad (5-3)$$

So the expected run time for all iterations will be

$$E(R) = \frac{nE_1(R)}{1 - P_{ID}[\text{undetectable} \mid Y < T_i]} \quad (5-4) \square$$

### 5.4. Comparison of two algorithms

The checksum and ID algorithms were implemented on an Intel iPSC/2. Figure 5-5 presents the fault-free running time $T_c$, $T_i$, a sequential algorithm, and Gentleman’s algorithm for 16 processors. As a worst case, we assume that the execution of a sequential algorithm is fault-free. The key performance figure, however, is not this fault-free run time, but the expected run time in the presence of errors.

To estimate the expected run time in a massively parallel machine, the estimated fault-free run time of the two algorithms are calculated with constants based on the iPSC/2 determined by regression analysis. Let $n$ be a dimension of square matrix and $p$ is a dimension of processors.

For the checksum algorithm, [HoMc91]

$$T_c = n^3 \left( \frac{0.0194 \times \frac{1}{p}}{1 - 0.0465 \times \frac{1}{p} + 6.25E - 03 \times \frac{1}{\sqrt{p}}} \right) \quad (5-5)$$
The ID algorithm is better than sequential, but worse than the Checksum algorithm.

For the ID algorithm,
\[
T_i = n^3 \left( 0.03241 \times \frac{1}{p} \right) + n \left( 0.0117 \times \frac{\log_2 \sqrt{p}}{\sqrt{p}} + 0.05205 \times \frac{1}{\sqrt{p}} + 0.0112 \times \frac{1}{p} \right) + n \left( 2.57 \times \log_2 \sqrt{p} - 0.11 \right) + 1.66 \times \log_2 \sqrt{p}
\]

From the above estimated fault-free run time, Figure 5-6 presents the crossover point where dimension of processors is 1024 and exponential parameter (MTTF of any one processor) is 1000. Figure 5-7 presents the crossover point where dimension of processors is 32768 and parameter is 5000. From two figures, we can know that ID algorithm is best suited to massive parallel processing.

**VI. CONCLUSION**

Matrix multiplication is one of the classic model problems for application-oriented and algorithm-based fault tolerance techniques. We have presented fault-tolerant algorithm (ID) for matrix multiplication which was derived using the application-oriented fault tolerance paradigm. The ID algorithm detects faults with a single iteration fault latency as opposed to a full problem run-time fault detection latency for the checksum algorithm. The penalty for this increased fault coverage in the case of no faults occurring is an extra \(\log n\) factor in the size of the matrix. However, the expected running time of the ID in the presence of faults improves quickly for moderate to large problems over the checksum algorithm. While the error coverage of the ID algorithm is not complete, it is better than the checksum algorithm.

**VII. BIBLIOGRAPHY**


