Matrix-Matrix Multiplications and Fault Tolerance on Hypercube Multiprocessors

Yuh-Rong Leu, Ing-Yi Chen* and Sy-Yen Kuo

Department of Electrical Engineering
National Taiwan University
Taipei, Taiwan, R.O.C.

*Department of Electronic Engineering
Chung Yuan Christian University
Chungli, Taiwan, R.O.C.

Abstract

Several new algorithms for matrix-matrix multiplications on hypercube multiprocessors are presented and evaluated based on the number of multiplications, additions, and transfers. The matrices to be multiplied are uniformly distributed to all processors of a hypercube system. Each processor owns some submatrices which are derived by dividing the source matrices. Each submatrix multiplication can now be performed independently within a processor. All the partial results are then summed up and transferred to a single processor. An ***orthogonal tree*** *is used for efficient communication. The time complexity is $O(\log_2 p)$ if $p \times p$ processors are used. In addition, the UDD (Uniform Data Distribution) approach is employed when some processors do not work properly and the faulty effects have been detected. Two classes of fault patterns are considered and evaluated.*

1: Basic Matrix-Vector Multiplication Algorithm

The matrix-matrix multiplication algorithms presented in this paper are extensions of the matrix-vector multiplication algorithm in [1]. The basic matrix-vector multiplication can be described as the inner product of a vector with each row of a given matrix [3]:

$$\text{If } A \in R^{m \times n}, x \in R^n, \text{ then}$$
$$y = A \cdot x, \text{ where } y_i = \sum_{j=1}^{n} A_{ij} \cdot x_j,\ i = 1, ..., m.$$

Alternatives in viewing a matrix-matrix multiplication are listed below:

$$\text{If } A \in R^{m \times n}, B \in R^{n \times q}, C \in R^{m \times q}, \text{ then}$$
$$(1)\ C = A \cdot B, \text{ where } C_{ij} = \sum_{k=1}^{n} A_{ik} \cdot B_{kj}$$
$$(2)\ C^j = A \cdot B^j$$

In the above definitions, A^k denotes the k-th column of A and A_k denotes the k-th row of A.

A common data mapping scheme for matrices is to use a two-dimensional grid embedding for the hypercube processors. To optimize the use of each processor, it is desirable to incorporate all nodes (processors) into the grid and distribute the matrix elements as evenly as possible among the nodes. A grid may be embedded on the hypercube using all nodes by numbering the processors according to the Gray code which is a binary number representation where only one bit is different for any two given consecutive numbers [4]. Fig. 1 illustrates such a Gray code numbering on a 5-D hypercube.

Acknowledgment: This research was supported by the National Science Council, Taiwan, R.O.C., under Grant NSC 82-0404-E033-049-T and NSC 82-0408-E002-021.

0-8186-3492-8/93 $3.00 © 1993 IEEE

	00	01	11	10
000	0 (1,1)	8 (1,2)	24 (1,3)	16 (1,4)
001	1 (2,1)	9 (2,2)	25 (2,3)	17 (2,4)
011	3 (3,1)	11 (3,2)	27 (3,3)	19 (3,4)
010	2 (4,1)	10 (4,2)	26 (4,3)	18 (4,4)
110	6 (5,1)	14 (5,2)	30 (5,3)	22 (5,4)
111	7 (6,1)	15 (6,2)	31 (6,3)	23 (6,4)
101	5 (7,1)	13 (7,2)	29 (7,3)	21 (7,4)
100	4 (8,1)	12 (8,2)	28 (8,3)	20 (8,4)

Fig. 1. Grid embedding.

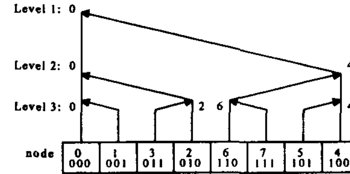


Fig. 2. Tree communication.

Suppose we need to sum up the values residing in the processors of column 1 in Fig. 1. The steps taken are shown in Fig. 2, where $\rightarrow$ indicates the data-flow direction. As soon as a processor receives data, it sums up the data with its own data.

The basic matrix-vector multiplication algorithm can then be expressed as follows:

1. Distribute the matrix A and the vector x to each processor according to its position (i,j) in the grid; thus, each processor acquires a portion of A (A_{ij}) and a portion of x (x_j).
2. Each processor computes sequentially the *intermediate subvector* $y_{ij} = A_{ij} \cdot x_j$.
3. Use the *tree communication* scheme row-wisely to gather and compute the resultant subvectors Y distributed on the first column-processors. The equation below describes this operation:

$$Y_i = \sum_{j=1}^{p} A_{ij} \cdot x_j = \sum_{j=1}^{p} y_{ij}.$$

4. Collect all Y_i's into processor (0,0) using the *tree communication* column-wisely.

2: Matrix-Matrix Multiplication Algorithms

The first algorithm is a simple approach. The distributing method of this algorithm is similar to the basic matrix-vector multiplication algorithm. In each computation cycle, only one column of matrix B is processed. Matrix A is distributed, and so each processor has a submatrix A_{ij}. Each B^j is distributed across the processors of a row. C^j is formed at the end of the j-th iteration. And in each iteration, a matrix-vector multiplication is performed as in the basic matrix-vector multiplication algorithm. So the entire matrix-matrix multiplication needs q computation cycles (see Fig. 3(a)).

The second one is named CE (Column-Expansion). In this algorithm, we use two augmented matrices each with m rows and nq columns. Each row of the augmented matrix consists of q copies of a row of matrix A. Associated with each copy is a column of matrix B (see Fig. 3(b)). This algorithm is based on the fact that $C_{ij} = A_i \cdot B^j$. After each processor has completed its own tasks of multiplications, additions, and transfers, the entries of each row of matrix C are distributed across the row-processors. So the tree communication scheme must be invoked to gather these data into the leftmost column-processors. In Fig. 3(b), B^{jt} stands for the transposed vector of B's j-th column.

The third is the RE (Row-Expansion) algorithm. Algorithm RE still needs two augmented matrices — one formed by matrix A and the other by B. But instead of expanding by the **column**, we expand the matrices by the **row**. Thus we have two matrices, each of size

$mq \times n$. The entries of these two matrices are then evenly distributed to all the processors. The computational mechanism is similar to that of the Simple Algorithm, but only one computation cycle is required (see Fig. 3(c)). Fig. 4 gives the comparisons between previous and proposed algorithms.

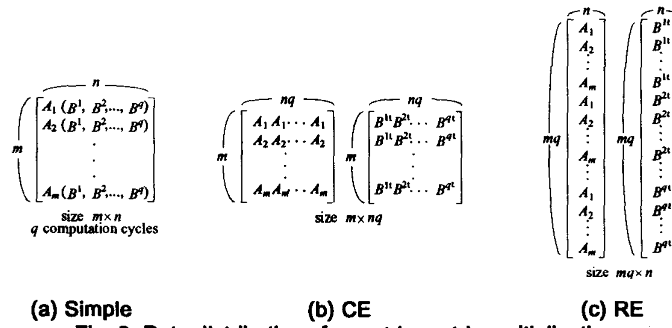


Fig. 3. Data distributions for matrix-matrix multiplications.

	Multiplications	Additions	Transfers
Sequential	$m \cdot q \cdot n$	$m \cdot q \cdot (n-1)$	0
2-D Grid	$q \cdot \lceil \frac{m}{p} \rceil \cdot \lceil \frac{n}{p} \rceil$	$q \cdot \lceil \frac{m}{p} \rceil \cdot [(\lceil \frac{n}{p} \rceil - 1) + p]$	$q \cdot \lceil \frac{m}{p} \rceil \cdot p$
Simple Algorithm	$q \cdot \lceil \frac{m}{p} \rceil \cdot \lceil \frac{n}{p} \rceil$	$q \cdot \lceil \frac{m}{p} \rceil \cdot [(\lceil \frac{n}{p} \rceil - 1) + \log_2 p]$	$q \cdot \lceil \frac{m}{p} \rceil \cdot \log_2 p$
Algorithm CE	$\lceil \frac{m}{p} \rceil \cdot \lceil \frac{nq}{p} \rceil$	$\lceil \frac{m}{p} \rceil \cdot (\lceil \frac{nq}{p} \rceil - 1)$	$\lceil \frac{m}{p} \rceil \cdot [\lceil \frac{q}{p} \rceil (p-1) + 1]$
Algorithm RE	$\lceil \frac{mq}{p} \rceil \cdot \lceil \frac{n}{p} \rceil$	$\lceil \frac{mq}{p} \rceil \cdot [(\lceil \frac{n}{p} \rceil - 1) + \log_2 p]$	$\lceil \frac{mq}{p} \rceil \cdot \log_2 p$

Fig. 4. Comparisons for matrix-matrix multiplication algorithms.

3: Redistribution under Multiple Faults

When some processors are faulty and the errors are detected, the jobs they are performing must be redistributed to other fault-free processors. This is the basic concept of load redistribution. We use the row/column UDD (Uniform Data Distribution) strategy, i.e. the work load of the faulty processors is first redistributed row-wisely and then column-wisely [2]. Two cases of multiple faults are considered in this section — either all faults are in one row or in one column.

Assume there are r faulty processors in a row and totally $p \times p$ processors in the hypercube.

Let: w be the width of the original largest submatrix,

h be the height of the original largest submatrix,

w_{1f} be the width of submatrices distributed after row-wise UDD,

h_{1f} be the reduced height of submatrices in the row with faulty nodes,

h_1 be the increased height of submatrices distributed to the remaining rows.

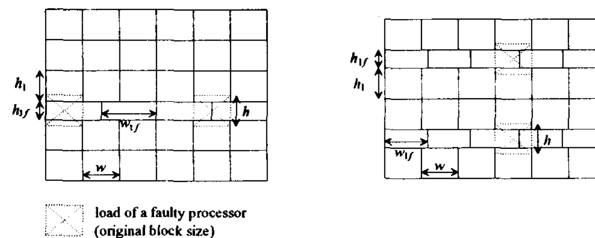
Therefore, $w = \lceil n/p \rceil$ and $h = \lceil m/p \rceil$ in the Simple Algorithm ; $w = \lceil nq/p \rceil$ and $h = \lceil m/p \rceil$ in Algorithm CE; $w = \lceil n/p \rceil$ and $h = \lceil mq/p \rceil$ in Algorithm RE. The following equations must hold in order for each node to have an equal number of data elements after the UDD (see Fig. 5(a)):

$w \cdot h_1 = w_{1f} \cdot h_{1f}$
$h_{1f} + (p-1) \cdot h_1 = p \cdot h$
$w_{1f} = w + rw/(p-r)$.

After a sequence of operations on the above equations, we get:
$h_{1f} = hp(p-r)/(p^2-r)$ and $w_{1f} = wp/(p-r)$.

Assume there are c faulty processors in a column, and again, $p \times p$ processors in the hypercube. Similarly, from Fig. 5(b), we have:
$w \cdot h_1 = w_{1f} \cdot h_{1f}$
$c \cdot h_{1f} + (p-c) \cdot h_1 = p \cdot h$
$w_{1f} = w + w/(p-1)$.

After manipulating these equations, the final results are:
$h_{1f} = hp(p-1)/(p^2-c)$; $w_{1f} = wp/(p-1)$.



(a) Faults in a row (b) Faults in a column
Fig. 5. Matrix partitioning after row/column-UDD.

4: Conclusions

In this paper, three algorithms for matrix-matrix multiplications are proposed and compared in terms of the number of multiplications, additions, and transfers. Furthermore, fault-tolerance issues for these algorithms are also discussed. These algorithms explore the connectivity of the hypercube and therefore are communication efficient. The transfer complexity of these algorithms is $O(\log_2 p)$ if $p \times p$ processors are used. Overall, Algorithm RE behaves better than the other two algorithms with better load balance, and lower communication overhead.

References

[1] A. C. Elster and A. P. Reeves, "Block-matrix Operations Using Orthogonal Trees," *Proceeding of the Third International Conference on Hypercube Multiprocessors*, SIAM, Pasadena, CA (January, 1988).

[2] A. C. Elster, M. Umit Uyar, and Anthony P. Reeves, "Fault-Tolerant Matrix Operations on Hypercube Multiprocessors", *International Conference on Parallel Processing* (1989).

[3] G. H. Golub and C. F. Van Loan, *Matrix Computations*, Johns Hopkins, Baltimore, MD (1983).

[4] J. Salmon, "Binary Gray Codes and the Mapping of Physical Lattice into a Hypercube," *Caltech Concurrent Processor Report (CCP)* Hm-51, (1983).

[5] S. L. Johnson, "Communication Efficient Basic Linear Algebra Computations on Hypercube Architecture," *Journal of Parallel and Distributed Computing*, 4, pp. 133-172 (1987).

[6] O. A. Mcbryan and E. F. Van de Velde, "Matrix and Vector Operations on Hypercube Parallel Processors," *Parallel Computing*, 5,(1&2), pp. 117-125, North-Holland, (July 1987).

[7] Angel L. Decegama, *The technology of Parallel Processing (Parallel Processing Architecture and VLSI Hardware Volume 1)*, Prentice Hall International Editions (1989).