.

Parallel Matrix Multiplication Algorithms on Hypercube Multiprocessors[1]

PeiZong Lee[2]

Institute of Information Science, Academia Sinica

Taipei, Taiwan, R.O.C.

Abstract

In this paper, we present three parallel algorithms for matrix multiplication. The first one, which employs pipelining techniques on a mesh grid, uses only one copy of data matrices. The second one uses multiple copies of data matrices also on a mesh grid. Although data communication operations of the second algorithm are reduced, the requirement of local data memory for each processing element increases. The third one, which uses a cubic grid, shows the trade-offs between reducing the computation time and reducing the communication overhead. Performance models and feasibilities of these three algorithms are studied. We analyze the interplay among the numbers of processing elements, the communication overhead, and the requirements of local memory in each processing element. We also present experimental results of these three algorithms on a 32-node nCUBE-2 computer.

1 Introduction

This paper is concerned with the design and analysis of efficient matrix multiplication algorithms on hypercube computers. Matrix multiplications, including matrix-matrix multiplications and two degenerate cases, vector-vector multiplications and matrix-vector multiplications, are the central operations for several scientific computations, such as graph problems, signal and image processing, linear algebra, and others.

For instance, in supercomputing, two of the most important linear algebra packages, LINPACK [3] and LAPACK [5], include BLAS (basic linear algebra subprograms), Level 2 BLAS, and Level 3 BLAS to improve the performance of portable programs on high-performance computers. BLAS

[1]This work was partially supported by the NSC under Grants NSC 82-0408-E-001-016.
[2]PeiZong Lee: leepe@iis.sinica.edu.tw, TEL: +886 (2) 788-3799, FAX: +886 (2) 782-4814.

is a set of routines based on vector-vector multiplications, Level 2 BLAS is a set of routines based on matrix-vector multiplications, and Level 3 BLAS is a set of routines based on matrix-matrix multiplications. Therefore, it is of practical importance to develop good algorithms for matrix multiplications on high performance computers.

Matrix multiplications on hypercubes have been studied before. Berntsen [1], Cherkassky and Smith [2], Fox and Otto [6], and Johnsson, Harris, and Mathur [8] designed matrix multiplication algorithms based on Cannon's algorithm on mesh grids, which can be embedded into hypercubes. Dekel, Nassimi, and Sahni [4], and Tichy [11] proposed various algorithms for matrix multiplication. However, their performance models did not consider the communication overhead. McBryan and Van de Velde [10], on the other hand, showed an upper bound for computing the matrix multiplication on hypercubes. In effect, all of the above works allowed only one copy of input and output matrices during the computation.

It is our goal in this paper to present several generic algorithms for matrix multiplications $C_{M\times N} = A_{M\times K} \times B_{K\times N}$ of arbitrary size on hypercube computers. These generic algorithms can be used to solve vector-vector multiplications including inner products (when $M = N = 1$) and cross products (when $K = 1$), matrix-vector multiplications (when either $M = 1$ or $N = 1$), and matrix-matrix multiplications (when M, N, $K \neq 1$). In addition, their performance models should include both the computation time and the communication overhead.

The rest of this paper is organized as follows. In Section 2, we review some properties of hypercubes. In Section 3, we introduce three parallel algorithms for matrix multiplications. In Section 4, we analyze the performance of these three algorithms using several different parameters. We also present experimental results of these three algorithms on a 32-node nCUBE-2 computer. Finally, some concluding remarks are given in Section 5.

2 Terminology and Definitions

In this paper, we are concerned with implementing the matrix multiplication $C_{M\times N} = A_{M\times K} \times B_{K\times N}$ on distributed memory supercomputers. The abstract target machine we adopt is a ω-D grid of $N_1 \times N_2 \times \cdots \times N_\omega$ processors, where D stands for dimensional, $\omega \leq 3$, $N_i = 2^{d_i}$, and $d_i \geq 0$. If ω is equal to 1, it is a processor ring. If ω is equal to 2, it is a mesh grid (torus). If ω is equal to 3, it is a cubic grid. Such a topology can be easily embedded into a hypercube computer using a binary reflected Gray code [7].

In a ω-D processor grid, there are two methods to represent a processing element (PE). First, the Cartesian coordinate representation: a processor on the ω-D grid is represented by the tuple $(x_1, x_2, \ldots, x_\omega)$, where $0 \leq x_i \leq N_i - 1$ for $1 \leq i \leq \omega$. Second, the binary code representation: let P, which is $p_{d_1+d_2\cdots+d_\omega-1}p_{d_1+d_2\cdots+d_\omega-2} \cdots p_{d_2\cdots+d_\omega}p_{d_2\cdots+d_\omega-1} \cdots p_{d_\omega}\ p_{d_\omega-1}\cdots p_1p_0$, be the binary code representation of the tuple $(x_1, x_2, \ldots, x_\omega)$. That is, x_1 is $p_{d_1+d_2\cdots+d_\omega-1}p_{d_1+d_2\cdots+d_\omega-2} \cdots p_{d_2\cdots+d_\omega}$, x_2 is $p_{d_2\cdots+d_\omega-1}p_{d_2\cdots+d_\omega-2}\cdots p_{d_3\cdots+d_\omega}$, …, and x_ω is $p_{d_\omega-1}p_{d_\omega-2}\cdots p_1p_0$. Therefore, $PE(P)$ is the binary code representation of $PE(x_1, x_2, \ldots, x_\omega)$. For instance, $PE(011011)$ is the binary code representation of $PE(1, 2, 3)$ on a $2^2 \times 2^2 \times 2^2$ cubic grid of processors.

In a tightly-coupled distributed memory computer, data arrays (or data matrices) must be distributed across processors before any computation. Data arrays can be distributed across processors or be replicated; they can be partitioned in a contiguous (block) manner, a cyclic manner, or a block cyclic manner [9]. In this paper, as we consider the general values of coefficients M, N, K, and N_i for $1 \leq i \leq \omega$, therefore, data arrays are distributed across processors in a cyclic manner to achieve a better load balance among processors. Let $A(P)$ (or $A(x_1, x_2, \ldots, x_w)$) represent the set of all entries of matrix A in $PE(P)$ (or $PE(x_1, x_2, \ldots, x_\omega)$), $B(P)$ represent the set of all entries of matrix B in $PE(P)$, and $C(P)$ represent the set of all entries of matrix C in $PE(P)$.

Let $P^{(q)}$ be the binary code representation such that $P^{(q)} = P$ except that the qth bit of $P^{(q)}$ is different from the qth bit of P. Therefore, if $P = p_{d_1+d_2\cdots+d_\omega-1}\cdots p_q \cdots p_1p_0$, then $P^{(q)} = p_{d_1+d_2\cdots+d_\omega-1}\cdots \bar{p}_q \cdots p_1p_0$, where $\bar{p}_q$ is the complement of p_q. We now define two notations, which will be used in Algorithms 2 and 3. Notation $A(P^{(q)}) \Longleftarrow A(P^{(q)}) \cup A(P)$ represents two operations. For all processing elements, first, send a copy of $A(P)$ from $PE(P)$ to $PE(P^{(q)})$; however, the original $A(P)$ is still in $PE(P)$. Second, in $PE(P^{(q)})$, union $A(P^{(q)})$ and $A(P)$ to $A(P^{(q)})$. Therefore, $A(P^{(q)})$ becomes $A(P^{(q)}) \cup A(P)$. This operation is known to be the multicast operation. For example, suppose that $A(P) = \{a_a,\ a_b,\ a_c,\ a_d\}$ and $A(P^{(q)}) = \{a_e,\ a_f,\ a_g,\ a_h\}$, then, $A(P^{(q)}) \cup A(P) = \{a_a,\ a_b,\ a_c,\ a_d,\ a_e,\ a_f,\ a_g,\ a_h\}$. Notation $C(P^{(q)}) \longleftarrow C(P^{(q)}) + C(P)$ represents two operations. For all processing elements, first, send a copy of $C(P)$ from $PE(P)$ to $PE(P^{(q)})$; however, the original $C(P)$ is still in $PE(P)$. Second, in $PE(P^{(q)})$, add $C(P)$ to $C(P^{(q)})$ pairwisely. Therefore, $C(P^{(q)})$ becomes $C(P^{(q)}) + C(P)$. This operation is known to be the cascade sum operation. For example, suppose that $C(P) = \{c_a,\ c_b,\ c_c,\ c_d\}$ and $C(P^{(q)}) = \{c_e,\ c_f,\ c_g,\ c_h\}$, then, $C(P^{(q)}) + C(P) = \{c_a + c_e,\ c_b + c_f,\ c_c + c_g,\ c_d + c_h\}$.

3 Three Parallel Algorithms for Matrix Multiplications

A sequential matrix multiplication algorithm for computing $C_{M\times N} = A_{M\times K} \times B_{K\times N}$ can be as follows:

for $i = 0$ **to** $M - 1$, **for** $j = 0$ **to** $N - 1$, **for** $k = 0$ **to** $K - 1$, **do**

$c[i,j] = c[i,j] + a[i,k] * b[k,j]$.

The iteration space of the above algorithm is a three dimensional lattice space. For a specific i and k, $a[i,k]$ will be used in all indices $(i,j,k)^t$, for $0 \le j < N$. For a specific k and j, $b[k,j]$ will be used in all indices $(i,j,k)^t$, for $0 \le i < M$. For a specific i and j, $c[i,j]$ will be used in all indices $(i,j,k)^t$, for $0 \le k < K$. That is, the data dependence vectors with variables $a[i,k]$, $b[k,j]$, and $c[i,j]$ are $(0,1,0)^t$, $(1,0,0)^t$, and $(0,0,1)^t$, respectively. In addition, these three data dependence vectors can span the whole three dimensional lattice space. Therefore, under this algorithm structure, if only one copy of matrices A and B are distributed across processors, then communication operations are necessary for computing $C = A \times B$.

Since matrices A and B are only used to calculate matrix C, they will not be modified during the computation. Therefore, if multiple copies of matrices A and B are distributed across processors, communication overhead may be reduced. However, each processing element must contain a sufficiently large amount of data memory for storing replicated data. On the other hand, each entry of matrix C is the value of an inner product by a row of matrix A and a column of matrix B. This computation can be accomplished by using the divide-and-conquer technique. In effect, in a massively parallel computer, each processing element is designed to compute only a reasonable grain size of iterations, therefore, computation time can be small. However, communication overhead must be paid for integrating the final result. To sum up, there are several trade-offs when designing an efficient parallel algorithm, including: using only a single copy of data matrices versus using multiple copies of data matrices; requiring data communication operations versus requiring a sufficiently large amount of local data memory; and, reducing computation time versus reducing data communication time.

In this section, we describe three parallel algorithms for computing matrix multiplications. The first one uses only one copy of data matrices A and B, and therefore, data communication operations are required. We will introduce a variant version of Cannon's algorithm using a mesh grid [1]. The second one uses multiple copies of data matrices also on a mesh grid. Although data communication operations of the second algorithm reduces, the requirement of local memory for each processing element increases. The third algorithm, which uses a cubic grid, shows the trade-offs between reducing the computation time and reducing the communication time. In effect, Algorithm 1 is a special case of Algorithm 2 and Algorithm 2 is a special case of Algorithm 3.

3.1 A Parallel Algorithm Based on the Pipelining Technique

Algorithm 1: Given a hypercube with 2^{DIM} PEs. Compute matrix multiplications $C_{M\times N} = A_{M\times K} \times B_{K\times N}$ on a subcube with $2^{d_1} \times 2^{d_2}$ PEs, where $d_1 + d_2 \leq DIM$, $M \geq 2^{d_1}$, $N \geq 2^{d_2}$, DIM, M, K, and N are positive integers, and d_1 and d_2 are non-negative integers.

Initial data distributions of $a[r,s]$, $b[u,v]$, and $c[i,j]$ on $PE(x,y)$ are given in Table 1, where $0 \leq r,\ i \leq M-1$, $0 \leq s,\ u \leq K-1$, $0 \leq v,\ j \leq N-1$, $0 \leq x \leq 2^{d_1}-1$, and $0 \leq y \leq 2^{d_2}-1$. $C(x,y) = 0$ (the initial condition).

Main part: In each $PE(x, y)$,

for iteration $= 0$ **to** $\max\{2^{d_1}, 2^{d_2}\} - 1$ **do**

  $C(x, y) = C(x, y) + A(x, y) \times B(x, y)$;

  send $A(x, y)$ **to** $PE(x,\ (y + 1) \bmod 2^{d_2})$;

  send $B(x, y)$ **to** $PE((x + 1) \bmod 2^{d_1},\ y)$;

  receive $A(x, y)$ **from** $PE(x,\ (y - 1) \bmod 2^{d_2})$;

  receive $B(x, y)$ **from** $PE((x - 1) \bmod 2^{d_1},\ y)$;

end_for. $\square$

Entries	Initially in $PE(x, y)$
$a[r, s]$	$(r \bmod 2^{d_1},\ (-r - s) \bmod 2^{d_2})$
$b[u, v]$	$((-u - v) \bmod 2^{d_1},\ v \bmod 2^{d_2})$
$c[i, j]$	$(i \bmod 2^{d_1},\ j \bmod 2^{d_2})$

Table 1: Initial data distributions of matrices A, B, and C on $PE(x, y)$ in Algorithm 1.

Fig. 1 shows the initial data distributions of matrices $A_{4\times4}$, $B_{4\times4}$, and $C_{4\times4}$ on a 4×4 processor grid. After computing a submatrix multiplication, in each PE, the submatrix of A will be sent to the right-neighbor (eastern) PE; the submatrix of B will be sent to the below-neighbor (southern) PE. In this algorithm, only at most $\lceil \frac{M}{2^{d_1}} \rceil \cdot \lceil \frac{K}{2^{d_2}} \rceil$ and $\lceil \frac{K}{2^{d_1}} \rceil \cdot \lceil \frac{N}{2^{d_2}} \rceil$ units of data locations in each PE are required for storing temporary submatrices A and B, respectively.

x \ y	0	1	2	3
0	A00	A03	A02	A01
1	A13	A12	A11	A10
2	A22	A21	A20	A23
3	A31	A30	A33	A32

B00	B31	B22	B13
B30	B21	B12	B03
B20	B11	B02	B33
B10	B01	B32	B23

C00	C01	C02	C03
C10	C11	C12	C13
C20	C21	C22	C23
C30	C31	C32	C33

Figure 1: Initial data distributions of matrices $A_{4\times4}$, $B_{4\times4}$, and $C_{4\times4}$ on a 4×4 processor grid.

3.2 A Parallel Algorithm Employing Multiple Copies of Data Arrays

Algorithm 1 computes $C_{M\times N} = A_{M\times K} \times B_{K\times N}$ using a mesh grid with $2^{d_1} \times 2^{d_2}$ PEs. In Algorithm 2, $2^{\delta_1} \times 2^{\delta_2}$ mesh grids are used, where $0 \le \delta_1 \le d_1$, $0 \le \delta_2 \le d_2$, and each mesh grid contains $2^{d_1-\delta_1} \times 2^{d_2-\delta_2}$ PEs. Specifically, in Algorithm 2, $2^{d_1} \times 2^{d_2}$ PEs are virtually divided into $2^{\delta_1} \times 2^{\delta_2}$ blocks. Each block is a mesh grid containing $2^{d_1-\delta_1} \times 2^{d_2-\delta_2}$ PEs. In addition, each block will compute a subproblem $C_{M'\times N'} = A_{M'\times K} \times B_{K\times N'}$ using Algorithm 1, where $M' = \lceil \frac{M}{2^{\delta_1}} \rceil$ or $\lfloor \frac{M}{2^{\delta_1}} \rfloor$, and $N' = \lceil \frac{N}{2^{\delta_2}} \rceil$ or $\lfloor \frac{N}{2^{\delta_2}} \rfloor$.

Algorithm 2: Given a hypercube with 2^{DIM} PEs. Compute matrix multiplications $C_{M\times N} = A_{M\times K} \times B_{K\times N}$ on a subcube with $2^{d_1} \times 2^{d_2}$ PEs, which are virtually divided into $2^{\delta_1} \times 2^{\delta_2}$ blocks, each block contains $2^{d_1-\delta_1} \times 2^{d_2-\delta_2}$ PEs, where $0 \le \delta_1 \le d_1$, $0 \le \delta_2 \le d_2$, $d_1 + d_2 \le DIM$, $M \ge 2^{d_1}$, $N \ge 2^{d_2}$, DIM, M, K, and N are positive integers, and d_1, d_2, δ_1, and δ_2 are non-negative integers.

Initial data distributions of entries A, B, and C are the same as those in Algorithm 1.

Main part: In each $PE(P)$,

/* Phase 1: Multicast phase. */

1. **for** $q = d_2 - 1$ **down to** $d_2 - \delta_2$ **do**
2. $\quad A(P^{(q)}) \Longleftarrow A(P^{(q)}) \cup A(P)$

   /* Multicast $A(P)$ to 2^{δ_2} PEs,

   which are in the same row and each PE is in a distinct block. */
3. **end_for**;
4. **for** $q = d_1 + d_2 - 1$ **down to** $d_1 - \delta_1 + d_2$ **do**
5. $\quad B(P^{(q)}) \Longleftarrow B(P^{(q)}) \cup B(P)$

   /* Multicast $B(P)$ to 2^{δ_1} PEs,

   which are in the same column and each PE is in a distinct block. */
6. **end_for**;

/* Phase 2: Computation phase.

$2^{d_1} \times 2^{d_2}$ PEs are virtually divided into $2^{\delta_1} \times 2^{\delta_2}$ blocks,

each block contains $2^{d_1-\delta_1} \times 2^{d_2-\delta_2}$ PEs, and all $2^{\delta_1} \times 2^{\delta_2}$ blocks of PEs execute Algorithm 1 simultaneously. */

7. **forall** $2^{\delta_1} \times 2^{\delta_2}$ blocks of PEs **do** Algorithm 1. □

Fig. 2 shows the data distributions of two copies each of matrices $A_{4\times4}$ and $B_{4\times4}$ and one copy of matrix $C_{4\times4}$ on a 4×4 processor grid after executing Step 6 in Algorithm 2. The 4×4 processor grid is virtually divided into 2×2 blocks. Each block, which contains 2×2 PEs, will then compute a subproblem $C_{2\times2} = A_{2\times4} \times B_{4\times2}$ using Algorithm 1. Note that Algorithm 2 is a generalization of Algorithm 1. If $\delta_1 = \delta_2 = 0$, then there is only one block, and Algorithm 2 is equal to Algorithm 1. However, in the other extreme case when $\delta_1 = d_1$ and $\delta_2 = d_2$, there are $2^{d_1} \times 2^{d_2}$ blocks and each block contains only one PE. In this case, all necessary data entries of submatrices A and B must be stored in the local memory of each PE before any computation. It requires at least $\lfloor \frac{M}{2^{d_1}} \rfloor \cdot K$ and $K \cdot \lfloor \frac{N}{2^{d_2}} \rfloor$ units of data locations in each PE for storing submatrices A and B, respectively. Yet, no communication operations are required for the further computation.

A00 A02	A01 A03	A00 A02	A01 A03
A11 A13	A10 A12	A11 A13	A10 A12
A20 A22	A21 A23	A20 A22	A21 A23
A31 A33	A30 A32	A31 A33	A30 A32

B00 B20	B11 B31	B02 B22	B13 B33
B10 B30	B01 B21	B12 B32	B03 B23
B00 B20	B11 B31	B02 B22	B13 B33
B10 B30	B01 B21	B12 B32	B03 B23

C00	C01	C02	C03
C10	C11	C12	C13
C20	C21	C22	C23
C30	C31	C32	C33

Figure 2: Data distributions of matrices $A_{4\times4}$, $B_{4\times4}$, and $C_{4\times4}$ on a 4×4 processor grid, which is virtually divided into 2×2 blocks, after executing Step 6 in Algorithm 2.

3.3 A Parallel Algorithm Using a Cubic Grid of Processors

In Algorithm 3, the problem $C_{M\times N} = A_{M\times K} \times B_{K\times N}$ is first divided into 2^{d_3} subproblems $C_{M\times N} = A_{M\times K'} \times B_{K'\times N}$, where K' is $\lceil \frac{K}{2^{d_3}} \rceil$ or $\lfloor \frac{K}{2^{d_3}} \rfloor$. In effect, the division of the original problem is accomplished by assigning initial entries of A, B, and C to 2^{d_3} groups of PEs, where each group contains $2^{d_1} \times 2^{d_2}$ PEs. Then, each subproblem is computed by a group of $2^{d_1} \times 2^{d_2}$ PEs using Algorithm 2. Finally, these 2^{d_3} different copies of $C_{M\times N}$ are summed up to obtain the result.

Therefore, in this algorithm, $2^{d_1} \times 2^{d_2} \times 2^{d_3}$ PEs are used. PEs can be indexed by $PE(x, y, z)$ or $PE(P)$, where $0 \le x \le 2^{d_1} - 1$, $0 \le y \le 2^{d_2} - 1$, $0 \le z \le 2^{d_3} - 1$, and $P = p_{d_1+d_2+d_3-1} p_{d_1+d_2+d_3-2} \cdots p_{d_2+d_3} p_{d_2+d_3-1} \cdots p_{d_3} p_{d_3-1} \cdots p_1 p_0$. That is, x is $p_{d_1+d_2+d_3-1} p_{d_1+d_2+d_3-2} \cdots p_{d_2+d_3}$, y is $p_{d_2+d_3-1} \cdots p_{d_3}$, and z is $p_{d_3-1} \cdots p_1 p_0$. In addition, PEs can be virtually divided into 2^{d_3} groups, where each group contains $2^{d_1} \times 2^{d_2}$ PEs.

Algorithm 3: Given a hypercube with 2^{DIM} PEs. Compute matrix multiplications $C_{M \times N} = A_{M \times K} \times B_{K \times N}$ on a subcube with $2^{d_1} \times 2^{d_2} \times 2^{d_3}$ PEs, which are virtually divided into 2^{d_3} groups. In addition, each group is virtually divided into $2^{\delta_1} \times 2^{\delta_2}$ blocks, and each block contains $2^{d_1-\delta_1} \times 2^{d_2-\delta_2}$ PEs, where $d_1 + d_2 + d_3 \le DIM$, $0 \le \delta_1 \le d_1$, $0 \le \delta_2 \le d_2$, $M \ge 2^{d_1}$, $N \ge 2^{d_2}$, $K \ge 2^{d_3}$, DIM, M, K, and N are positive integers, and d_1, d_2, d_3, δ_1, and δ_2 are non-negative integers.

Initial data distributions of $a[r, s]$, $b[u, v]$, and $c[i, j]$ on $PE(x, y, z)$ are given in Table 2, where $0 \le r, i \le M - 1$, $0 \le s, u \le K - 1$, $0 \le v, j \le N - 1$, $0 \le x \le 2^{d_1} - 1$, $0 \le y \le 2^{d_2} - 1$, and $0 \le z \le 2^{d_3} - 1$. $C(x, y, z) = 0$ (the initial condition).

Main part: In each $PE(P)$,

/* Phase 1: Multicast phase. */

1. **for** $q = d_2 + d_3 - 1$ **down to** $d_2 - \delta_2 + d_3$ **do**
2. $\quad A(P^{(q)}) \Longleftarrow A(P^{(q)}) \cup A(P)$

   /* Multicast $A(P)$ to 2^{δ_2} PEs, which are in the same row of a specific group and each PE is in a distinct block. */
3. **end_for**;
4. **for** $q = d_1 + d_2 + d_3 - 1$ **down to** $d_1 - \delta_1 + d_2 + d_3$ **do**
5. $\quad B(P^{(q)}) \Longleftarrow B(P^{(q)}) \cup B(P)$

   /* Multicast $B(P)$ to 2^{δ_1} PEs, which are in the same column of a specific group and each PE is in a distinct block. */
6. **end_for**;

/* Phase 2: Computation phase.

$2^{d_1} \times 2^{d_2} \times 2^{d_3}$ PEs are virtually divided into 2^{d_3} groups,

each group is virtually divided into $2^{\delta_1} \times 2^{\delta_2}$ blocks,

each block contains $2^{d_1-\delta_1} \times 2^{d_2-\delta_2}$ PEs, and

all $2^{\delta_1} \times 2^{\delta_2} \times 2^{d_3}$ blocks of PEs execute Algorithm 1 simultaneously. */

7. **forall** $2^{\delta_1} \times 2^{\delta_2} \times 2^{d_3}$ blocks of PEs **do** Algorithm 1;

/* Phase 3: Reduction phase.

Add terms based on the binary tree summation.

The final results of matrix C are replicated on all 2^{d_3} groups of PEs. */

8. **for** $q = d_3 - 1$ **down to** 0 **do**

9. $C(P^{(q)}) \longleftarrow C(P^{(q)}) + C(P)$

10. **end_for**.

Entries	Initially in $PE(x, y, z)$
$a[r, s]$	$(r \bmod 2^{d_1},\ (-r - \lfloor s/2^{d_3} \rfloor) \bmod 2^{d_2},\ s \bmod 2^{d_3})$
$b[u, v]$	$((-\lfloor u/2^{d_3} \rfloor - v) \bmod 2^{d_1},\ v \bmod 2^{d_2},\ u \bmod 2^{d_3})$
$c[i, j]$	$(i \bmod 2^{d_1},\ j \bmod 2^{d_2},\ X)$

Table 2: Initial data distributions of matrices A, B, and C on $PE(x, y, z)$ in Algorithm 3. X means that data will be replicated on all PEs along a specific grid dimension.

Fig. 3-(a) shows initial data distributions of matrices $A_{4\times 8}$, $B_{8\times 4}$, and $C_{4\times 4}$ on a $4 \times 4 \times 2$ cubic grid of processors in Algorithm 3. The cubic grid is virtually divided into 2 groups, each group contains 2×2 blocks, and each block contains 2×2 PEs. Fig. 3-(b) shows data distributions of two copies each of matrices A, B, and C across processors after executing Step 6 in Algorithm 3. Each block of PEs will then compute a subproblem $C_{2\times 2} = A_{2\times 4} \times B_{4\times 2}$ using Algorithm 1. Finally, two different copies of matrix C are summed up to obtain the final result. Note that Algorithm 3 is a generalization of Algorithm 2. If $d_3 = 0$, then there is only one group, and Algorithm 3 is equal to Algorithm 2. On the other hand, if $d_3 \neq 0$, then the cubic grid of PEs is divided into 2^{d_3} groups of PEs, where each group contains $2^{d_1} \times 2^{d_2}$ PEs, and in addition, each group of PEs will apply Algorithm 2 to solve a subproblem. The final result can then be obtained by summing up the results of these 2^{d_3} subproblems.

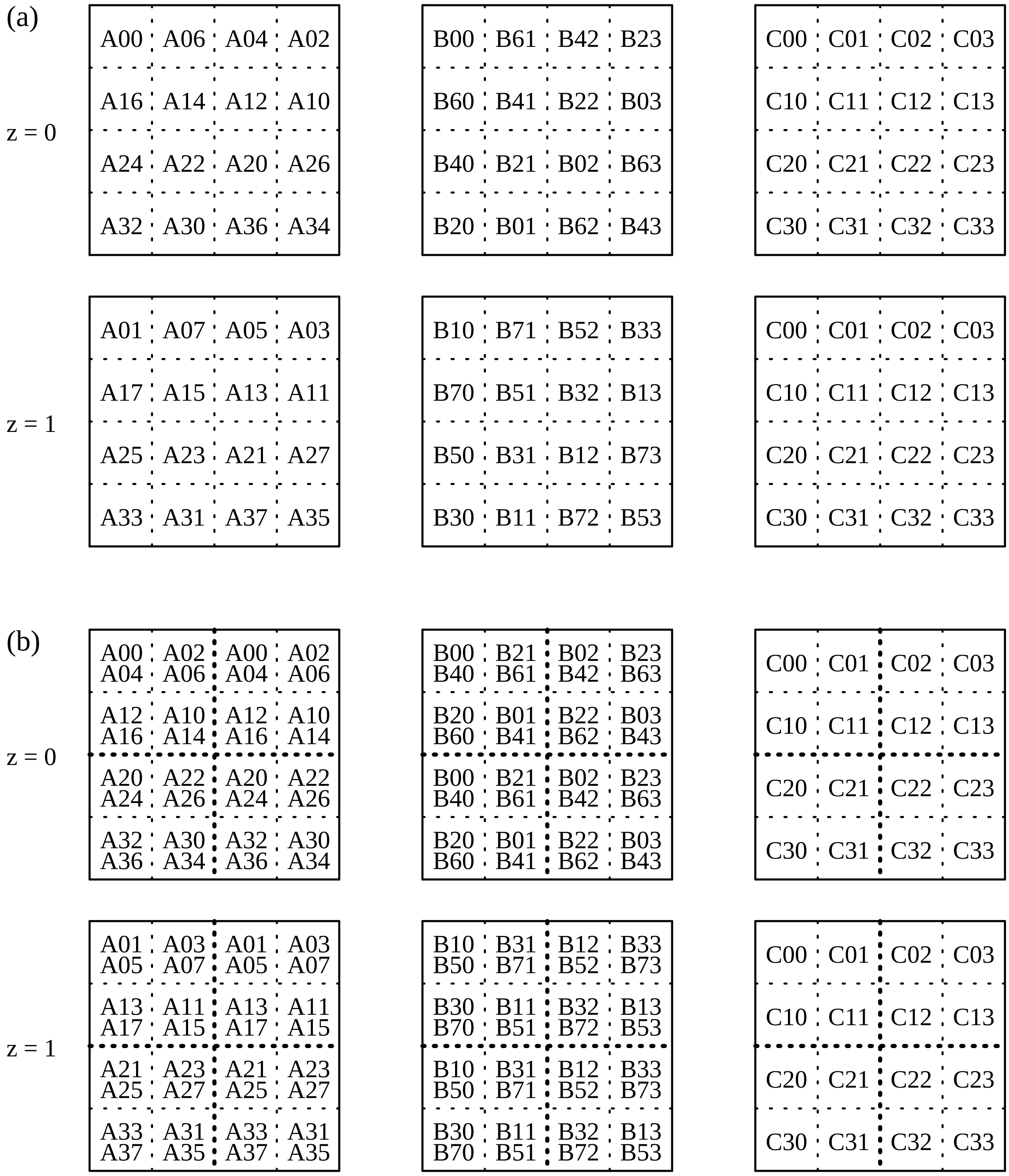

Figure 3: (a) Initial data distributions of matrices $A_{4\times 8}$, $B_{8\times 4}$, and $C_{4\times 4}$ on a $4 \times 4 \times 2$ cubic grid of processors, which is virtually divided into 2 groups and each group contains 2×2 blocks, in Algorithm 3. (b) Data distributions after executing Step 6 in Algorithm 3.

4 Analytical Performance Comparisons

In this section, we analyze the performance of the three parallel algorithms. In distributed memory systems such as the nCUBE-2 computer, the message passing technique and wormhole routing schema are used for data communication. In general, it requires $t_s + M \cdot t_w$ time for sending a message of M words from one PE to another PE, where t_s is the start-up time for sending a message and t_w is the time for sending a consecutive word (4 bytes). Let t_f be the time of computing an iteration which includes a multiplication and an addition and other loop-control overhead generated by compilers, such as loop index incrementing operation, loop index comparison operation, and jump operation. Let t_a be the time of computing an iteration which includes an addition and other loop-control overhead generated by compilers. These parameters depend on target computers. For instance, the parameters of the nCUBE-2 computer are listed in Table 3.

parameter	t_s	t_w	t_f	t_a
average value (in μsec)	110	0.25	0.85	0.35

Table 3: Parameters used in describing the execution time of parallel matrix multiplication algorithms on the nCUBE-2 computer.

Then, the total execution time for computing Algorithm 1 is as follows: $\max\{2^{d_1}, 2^{d_2}\} \cdot [\frac{M}{2^{d_1}} \frac{K}{\max\{2^{d_1}, 2^{d_2}\}} \frac{N}{2^{d_2}} \cdot t_f + 2t_s + (\frac{M}{2^{d_1}} \frac{K}{\max\{2^{d_1}, 2^{d_2}\}} + \frac{K}{\max\{2^{d_1}, 2^{d_2}\}} \frac{N}{2^{d_2}}) \cdot t_w]$. The execution time includes the computation time: $\max\{2^{d_1}, 2^{d_2}\} \frac{M}{2^{d_1}} \frac{K}{\max\{2^{d_1}, 2^{d_2}\}} \frac{N}{2^{d_2}} \cdot t_f$ and the communication overhead due to pipelining submatrices A and B: $\max\{2^{d_1}, 2^{d_2}\}(2t_s + (\frac{M}{2^{d_1}} \frac{K}{\max\{2^{d_1}, 2^{d_2}\}} + \frac{K}{\max\{2^{d_1}, 2^{d_2}\}} \frac{N}{2^{d_2}}) \cdot t_w)$. In addition, in each PE, it requires $\frac{M}{2^{d_1}} \frac{K}{2^{d_2}}$ words (local memory) for submatrix A; $\frac{K}{2^{d_1}} \frac{N}{2^{d_2}}$ words for submatrix B; and $\frac{M}{2^{d_1}} \frac{N}{2^{d_2}}$ words for submatrix C. Note that, for simplicity, we assume that M is a multiple of 2^{d_1} and N is a multiple of 2^{d_2} in Algorithms 1, 2, and 3; K is a multiple of both 2^{d_1} and 2^{d_2} in Algorithms 1 and 2, and K is a multiple of both $2^{d_1+d_3}$ and $2^{d_2+d_3}$ in Algorithm 3.

The total execution time for computing Algorithm 2 is as follows: $\delta_1 t_s + (2^{\delta_1 - 1}) \frac{K}{2^{d_1}} \frac{N}{2^{d_2}} \cdot t_w + \delta_2 t_s + (2^{\delta_2 - 1}) \frac{M}{2^{d_1}} \frac{K}{2^{d_2}} \cdot t_w + \max\{2^{d_1 - \delta_1}, 2^{d_2 - \delta_2}\} \cdot [\frac{M}{2^{d_1}} \frac{K}{\max\{2^{d_1 - \delta_1}, 2^{d_2 - \delta_2}\}} \frac{N}{2^{d_2}} \cdot t_f + 2t_s + (\frac{M}{2^{d_1}} \frac{K}{\max\{2^{d_1 - \delta_1}, 2^{d_2 - \delta_2}\}} + \frac{K}{\max\{2^{d_1 - \delta_1}, 2^{d_2 - \delta_2}\}} \frac{N}{2^{d_2}}) \cdot t_w]$. The execution time includes the computation time: $\max\{2^{d_1 - \delta_1}, 2^{d_2 - \delta_2}\} \cdot$

$\frac{M}{2^{d_1}}\frac{K}{\max\{2^{d_1-\delta_1},2^{d_2-\delta_2}\}}\frac{N}{2^{d_2}}\cdot t_f$; the communication overhead due to multicasting submatrices B and A: $\delta_1 t_s+(2^{\delta_1-1})\frac{K}{2^{d_1}}\frac{N}{2^{d_2}}\cdot t_w+\delta_2 t_s+(2^{\delta_2-1})\frac{M}{2^{d_1}}\frac{K}{2^{d_2}}\cdot t_w$; and the communication overhead due to pipelining submatrices A and B: $\max\{2^{d_1-\delta_1},2^{d_2-\delta_2}\}\cdot(2t_s+(\frac{M}{2^{d_1}}\frac{K}{\max\{2^{d_1-\delta_1},2^{d_2-\delta_2}\}}+\frac{K}{\max\{2^{d_1-\delta_1},2^{d_2-\delta_2}\}}\frac{N}{2^{d_2}})\cdot t_w)$. In addition, in each PE, it requires $\frac{M}{2^{d_1}}\frac{K}{2^{d_2-\delta_2}}$ words for submatrix A; $\frac{K}{2^{d_1-\delta_1}}\frac{N}{2^{d_2}}$ words for submatrix B; and $\frac{M}{2^{d_1}}\frac{N}{2^{d_2}}$ words for submatrix C.

The total execution time for computing Algorithm 3 is as follows: $\delta_1 t_s+(2^{\delta_1-1})\frac{K}{2^{d_1+d_3}}\frac{N}{2^{d_2}}\cdot t_w+\delta_2 t_s+(2^{\delta_2-1})\frac{M}{2^{d_1}}\frac{K}{2^{d_2+d_3}}\cdot t_w+\max\{2^{d_1-\delta_1},2^{d_2-\delta_2}\}\cdot[\frac{M}{2^{d_1}}\frac{K}{\max\{2^{d_1-\delta_1+d_3},2^{d_2-\delta_2+d_3}\}}\frac{N}{2^{d_2}}\cdot t_f+2t_s+(\frac{M}{2^{d_1}}\frac{K}{\max\{2^{d_1-\delta_1+d_3},2^{d_2-\delta_2+d_3}\}}+\frac{K}{\max\{2^{d_1-\delta_1+d_3},2^{d_2-\delta_2+d_3}\}}\frac{N}{2^{d_2}})\cdot t_w]+d_3(\frac{M}{2^{d_1}}\frac{N}{2^{d_2}}\cdot t_a+t_s+\frac{M}{2^{d_1}}\frac{N}{2^{d_2}}\cdot t_w)$. The execution time includes the computation time: $\max\{2^{d_1-\delta_1+d_3},2^{d_2-\delta_2+d_3}\}\frac{M}{2^{d_1}}\frac{K}{\max\{2^{d_1-\delta_1+d_3},2^{d_2-\delta_2+d_3}\}}\cdot\frac{N}{2^{d_2}}\cdot t_f$; the communication overhead due to multicasting submatrices B and A: $\delta_1 t_s+(2^{\delta_1-1})\frac{K}{2^{d_1+d_3}}\cdot\frac{N}{2^{d_2}}\cdot t_w+\delta_2 t_s+(2^{\delta_2-1})\frac{M}{2^{d_1}}\frac{K}{2^{d_2+d_3}}\cdot t_w$; the communication overhead due to pipelining submatrices A and B: $\max\{2^{d_1-\delta_1+d_3},2^{d_2-\delta_2+d_3}\}(2t_s+(\frac{M}{2^{d_1}}\frac{K}{\max\{2^{d_1-\delta_1+d_3},2^{d_2-\delta_2+d_3}\}}+\frac{K}{\max\{2^{d_1-\delta_1+d_3},2^{d_2-\delta_2+d_3}\}}\frac{N}{2^{d_2}})\cdot t_w)$; and the communication overhead due to summing up submatrix C: $d_3(\frac{M}{2^{d_1}}\frac{N}{2^{d_2}}\cdot t_a+t_s+\frac{M}{2^{d_1}}\frac{N}{2^{d_2}}\cdot t_w)$. In addition, in each PE, it requires $\frac{M}{2^{d_1}}\frac{K}{2^{d_2-\delta_2+d_3}}$ words for submatrix A; $\frac{K}{2^{d_1-\delta_1+d_3}}\frac{N}{2^{d_2}}$ words for submatrix B; and $\frac{M}{2^{d_1}}\frac{N}{2^{d_2}}$ words for submatrix C.

In the following, we restrict ourselves to discussing the implementations of matrix multiplication on distributed memory computers whose topologies are either linear arrays, or square meshes, or cubes whose bottoms are square meshes. That is, if both d_1 and d_2 are not equal to zero, then $d_1 = d_2$; if both δ_1 and δ_2 are not equal to zero, then $\delta_1 = \delta_2$. Table 4 shows execution time and memory required for 14 cases with variants d_1, d_2, d_3, δ_1, and δ_2.

Case 1 is relevant to the sequential algorithm; Cases 2 through 4 are special cases of Algorithm 1; Cases 5 through 7 are special cases of Algorithm 2; Cases 8 through 14 are special cases of Algorithm 3. The computation time of Cases 2 through 14 are all the same if iterations are evenly distributed among PEs; however, their communication overhead and the requirements of local memory are different. In general, if data are replicated among PEs, communication overhead will be reduced; however, each PE requires more local memory.

case and constraint	topo-logy	execution time		memory in each PE		
		comput.	communication overhead	A	B	C
case 1 – 7: $d_3 = 0$						
1. $d_1 = d_2 = 0$	1 PE	$MKN \cdot t_f$	0	MK	KN	MN
2. $d_1 = 0,\ d_2 \neq 0$	array	$\frac{MKN}{2^{d_2}} \cdot t_f$	$2^{d_2}(t_s + \frac{MK}{2^{d_2}} \cdot t_w)$	$\frac{MK}{2^{d_2}}$	$\frac{KN}{2^{d_2}}$	$\frac{MN}{2^{d_2}}$
3. $d_1 \neq 0,\ d_2 = 0$	array	$\frac{MKN}{2^{d_1}} \cdot t_f$	$2^{d_1}(t_s + \frac{KN}{2^{d_1}} \cdot t_w)$	$\frac{MK}{2^{d_1}}$	$\frac{KN}{2^{d_1}}$	$\frac{MN}{2^{d_1}}$
4. $d_1 = d_2 = d \neq 0$	mesh	$\frac{MKN}{2^{2d}} \cdot t_f$	$2^{d}(2t_s + \frac{MK+KN}{2^{2d}} \cdot t_w)$	$\frac{MK}{2^{2d}}$	$\frac{KN}{2^{2d}}$	$\frac{MN}{2^{2d}}$
5. $d_1 = 0,\ 0 < \delta_2 \leq d_2$	array	$\frac{MKN}{2^{d_2}} \cdot t_f$	$\delta_2 t_s + (2^{\delta_2} - 1)\frac{MK}{2^{d_2}} \cdot t_w + 2^{d_2-\delta_2}(t_s + \frac{MK}{2^{d_2-\delta_2}} \cdot t_w)$	$\frac{MK}{2^{d_2-\delta_2}}$	$\frac{KN}{2^{d_2}}$	$\frac{MN}{2^{d_2}}$
6. $0 < \delta_1 \leq d_1,\ d_2 = 0$	array	$\frac{MKN}{2^{d_1}} \cdot t_f$	$\delta_1 t_s + (2^{\delta_1} - 1)\frac{KN}{2^{d_1}} \cdot t_w + 2^{d_1-\delta_1}(t_s + \frac{KN}{2^{d_1-\delta_1}} \cdot t_w)$	$\frac{MK}{2^{d_1}}$	$\frac{KN}{2^{d_1-\delta_1}}$	$\frac{MN}{2^{d_1}}$
7. $0 < \delta_1 = \delta_2 = \delta \leq d_1 = d_2 = d$	mesh	$\frac{MKN}{2^{2d}} \cdot t_f$	$2\delta t_s + (2^{\delta} - 1)\frac{MK+KN}{2^{2d}} \cdot t_w + 2^{d-\delta}(2t_s + \frac{MK+KN}{2^{2d-\delta}} \cdot t_w)$	$\frac{MK}{2^{2d-\delta}}$	$\frac{KN}{2^{2d-\delta}}$	$\frac{MN}{2^{2d}}$
case 8 – 14: $d_3 \neq 0$						
8. $d_1 = d_2 = 0$	array	$\frac{MKN}{2^{d_3}} \cdot t_f$	$d_3(MN \cdot t_a + t_s + MN \cdot t_w)$	$\frac{MK}{2^{d_3}}$	$\frac{KN}{2^{d_3}}$	MN
9. $d_1 = 0,\ d_2 \neq 0$	mesh	$\frac{MKN}{2^{d_2+d_3}} \cdot t_f$	$2^{d_2}(t_s + \frac{MK}{2^{d_2+d_3}} \cdot t_w) + d_3(\frac{MN}{2^{d_2}} \cdot t_a + t_s + \frac{MN}{2^{d_2}} \cdot t_w)$	$\frac{MK}{2^{d_2+d_3}}$	$\frac{KN}{2^{d_3+d_2}}$	$\frac{MN}{2^{d_2}}$
10. $d_1 \neq 0,\ d_2 = 0$	mesh	$\frac{MKN}{2^{d_1+d_3}} \cdot t_f$	$2^{d_1}(t_s + \frac{KN}{2^{d_1+d_3}} \cdot t_w) + d_3(\frac{MN}{2^{d_1}} \cdot t_a + t_s + \frac{MN}{2^{d_1}} \cdot t_w)$	$\frac{MK}{2^{d_1+d_3}}$	$\frac{KN}{2^{d_1+d_3}}$	$\frac{MN}{2^{d_1}}$
11. $d_1 = d_2 = d \neq 0$	cube	$\frac{MKN}{2^{2d+d_3}} \cdot t_f$	$2^{d}(2t_s + \frac{MK+KN}{2^{2d+d_3}} \cdot t_w) + d_3(\frac{MN}{2^{2d}} \cdot t_a + t_s + \frac{MN}{2^{2d}} \cdot t_w)$	$\frac{MK}{2^{2d+d_3}}$	$\frac{KN}{2^{2d+d_3}}$	$\frac{MN}{2^{2d}}$
12. $d_1 = 0,\ 0 < \delta_2 \leq d_2$	mesh	$\frac{MKN}{2^{d_2+d_3}} \cdot t_f$	$\delta_2 t_s + (2^{\delta_2} - 1)\frac{MK}{2^{d_2+d_3}} \cdot t_w + 2^{d_2-\delta_2}(t_s + \frac{MK}{2^{d_2-\delta_2+d_3}} \cdot t_w) + d_3(\frac{MN}{2^{d_2}} \cdot t_a + t_s + \frac{MN}{2^{d_2}} \cdot t_w)$	$\frac{MK}{2^{d_2-\delta_2+d_3}}$	$\frac{KN}{2^{d_2+d_3}}$	$\frac{MN}{2^{d_2}}$
13. $0 < \delta_1 \leq d_1,\ d_2 = 0$	mesh	$\frac{MKN}{2^{d_1+d_3}} \cdot t_f$	$\delta_1 t_s + (2^{\delta_1} - 1)\frac{KN}{2^{d_1+d_3}} \cdot t_w + 2^{d_1-\delta_1}(t_s + \frac{KN}{2^{d_1-\delta_1+d_3}} \cdot t_w) + d_3(\frac{MN}{2^{d_1}} \cdot t_a + t_s + \frac{MN}{2^{d_1}} \cdot t_w)$	$\frac{MK}{2^{d_1+d_3}}$	$\frac{KN}{2^{d_1-\delta_1+d_3}}$	$\frac{MN}{2^{d_1}}$
14. $0 < \delta_1 = \delta_2 = \delta \leq d_1 = d_2 = d$	cube	$\frac{MKN}{2^{2d+d_3}} \cdot t_f$	$2\delta t_s + (2^{\delta} - 1)\frac{MK+KN}{2^{2d+d_3}} \cdot t_w + 2^{d-\delta}(2t_s + \frac{MK+KN}{2^{2d-\delta+d_3}} \cdot t_w) + d_3(\frac{MN}{2^{2d}} \cdot t_a + t_s + \frac{MN}{2^{2d}} \cdot t_w)$	$\frac{MK}{2^{2d-\delta+d_3}}$	$\frac{KN}{2^{2d-\delta+d_3}}$	$\frac{MN}{2^{2d}}$

Table 4: Execution time and local memory required in each PE for 14 cases with variant d_1, d_2, d_3, δ_1, and δ_2.

4.1 Interplay Among the Numbers of PEs, the Communication Overhead, and the Requirements of Local Memory

Suppose that we have a sufficiently large amount of local memory in each PE. We now study the interplay between the numbers of PEs and the communication overhead.

(1) In general, when the number of PEs is small, Algorithm 3 has larger communication overhead. However, when the number of PEs grows, Algorithm 3 performs better than Algorithm 2, and Algorithm 2 performs better than Algorithm 1.

(2) Problem sizes influence algorithms' performances. For instance, when $M = K = N$, algorithms based on square meshes or cubes whose bottoms are square meshes perform better than others, such as cases 4, 7, 11, and 14 in Fig. 4-(a) and Fig. 4-(d). When $M > K > N$, Algorithms based on linear arrays perform better than others when the number of PEs is small, such as cases 3 and 10 in Fig. 4-(b). When $K > M$ and $K > N$, Algorithm 3 performs better than Algorithms 1 and 2, such as cases 10, 11, and 14 in Fig. 4-(c).

(3) When dealing with a large-size problem, we can improve the performance by increasing the number of PEs, such as the cases in Fig. 4-(a). This is because the communication overhead due to pipelining smaller grain submatrices drops faster than the communication overhead increased due to multicasting and summing up smaller grain submatrices.

(4) When dealing with a median-size problem, such as the cases in Fig. 4-(d), to increase the number of PEs results in increasing the communication overhead. This is because the communication overhead due to pipelining smaller grain submatrices drops slower than the communication overhead increased due to multicasting and summing up smaller grain submatrices. Therefore, algorithms' performances may deteriorate if we increase the number of PEs.

4.2 Experimental Studies

Table 5 lists experimental results of implementing three algorithms on a 32-node nCUBE-2 computer currently in Academia Sinica. In this computer, each node has 4 Mega bytes memory, runs

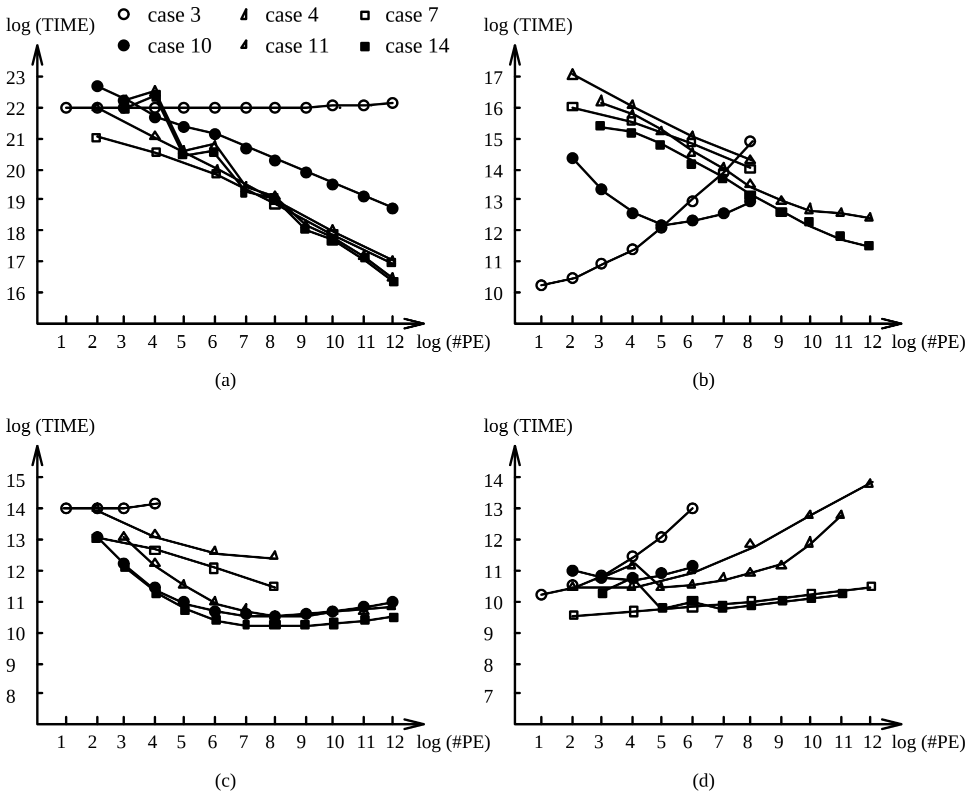

Figure 4: Communication overhead by applying cases 3; 4; 7; 10; 11; and 14 when (a) $M = K = N = 4096$; (b) $M = 4096$, $K = 256$, $N = 16$; (c) $M = 16$, $K = 4096$, $N = 16$; and (d) $M = K = N = 64$.

at a modest clock rate of 20 MHz, and is rated at 7.5 MIPS (Mega Instructions Per Second), and 3.5 MFLOPS (Mega FLOating-point operations Per Second) single precision. Because of the memory limitation, each algorithm only can be implemented up to a certain size. We present the implementations of the cases 3, 4, 7, 10, 11, and 14 in Table 4 when $M = K = N$.

First, all six cases show scalable improvements when the number of PEs grows. Second, the experimental results quite match the analytical performance model in Fig. 4 when the number of PEs is small. We see that case 4 and case 7 perform better than other cases; and case 3, which is implemented on a linear array without replicated data, can compete with case 10, case 11, and case 14 when the number of PEs is less than or equal to 32.

$M = K = N$	case 3				case 4				case 7			
	4 P	8 P	16 P	32 P	4 P	8 P	16 P	32 P	4 P	8 P	16 P	32 P
64	0	0	0	0	1	*	0	*	1	*	0	*
128	3	2	1	0	3	*	1	*	3	*	0	*
256	21	10	6	3	19	*	5	*	19	*	5	*
512	160	81	41	22	154	*	38	*	154	*	38	*
1024	1280	642	324	170	1229	*	307	*	*	*	306	*
2048	*	*	2568	1330	*	*	2458	*	*	*	*	*
$M = K = N$	case 10				case 11				case 14			
	4 P	8 P	16 P	32 P	4 P	8 P	16 P	32 P	4 P	8 P	16 P	32 P
64	0	0	0	0	*	0	0	0	*	0	0	0
128	3	1	1	1	*	2	1	1	*	2	1	0
256	21	11	6	3	*	11	5	3	*	11	5	3
512	167	85	42	22	*	84	42	21	*	85	42	21
1024	*	668	337	171	*	661	332	166	*	674	338	167
2048	*	*	*	1341	*	*	*	1323	*	*	*	*

Table 5: Execution time (second) of matrix multiplication algorithms using 4 PEs, 8 PEs, 16 PEs, and 32 PEs, respectively, on a 32-node nCUBE-2 computer. '*' means 'not implement'.

5 Conclusions

Algorithms 1, 2, and 3 show the interplay among the number of PEs, the communication overhead, and the requirements of local memory. Algorithm 2 requires more memory than Algorithm 1; however, Algorithm 2 incurs less communication overhead than Algorithm 1. In the theoretical model, when the number of PEs is large and memory has no limitation, Algorithm 3 performs better than Algorithm 2, and Algorithm 2 performs better than Algorithm 1. However, in practice, the number of PEs is small, Algorithm 3 has larger communication overhead than the other two algorithms. Thus, we may conclude that Algorithm 1 is suitable for the coarse grain computation; Algorithm 3 is suitable for the fine grain computation; and Algorithm 2 is suitable for the case when memory has no limitation.

References

[1] J. Berntsen. Communication efficient matrix multiplication on hypercubes. *Parallel Computing*, 12:335–342, 1989.

[2] V. Cherkassky and R. Smith. Efficient mapping and implementation of matrix algorithms on a hypercube. *The Journal of Supercomputing*, 2:7–27, 1988.

[3] T. F. Coleman and C. Van Loan. *Handbook for Matrix Computations*. SIAM, Philadelphia, PA, 1988.

[4] E. Dekel, D. Nassimi, and S. Sahni. Parallel matrix and graph algorithms. *SIAM J. Comput.*, 10(4):657–675, November 1981.

[5] E. Anderson *et al. LAPACK Users' Guide*. SIAM, Philadelphia, PA, 1992.

[6] G. C. Fox, S. W. Otto, and A. J. G. Hey. Matrix algorithms on a hypercube I: Matrix multiplication. *Parallel Computing*, 4:17–31, 1987.

[7] C. T. Ho. *Optimal Communication Primitives and Graph Embeddings on Hypercubes*. PhD thesis, Yale Univ., 1990.

[8] S. L. Johnsson, T. Harris, and K. K. Mathur. Matrix multiplication on the connection machine. In *Proc. of ACM Supercomputing '89*, pages 326–332. ACM, 1989.

[9] P.-Z. Lee and T. B. Tsai. Compiling efficient programs for tightly-coupled distributed memory computers. In *Proc. of International Conference on Parallel Processing*, pages II–161–165, St. Charles, IL, August 1993.

[10] O. A. Mcbryan and E. F. Van de Velde. Hypercube algorithms and implementations. *SIAM J. Sci. Stat. Comput.*, 8(2):117–125, March 1987.

[11] W. F. Tichy. Parallel matrix multiplication on the connection machine. *International Journal of High Speed Computing*, 1(2):247–262, 1989.