ORNL/TM-12309

Engineering Physics and Mathematics Division

Mathematical Sciences Section

PARALLEL MATRIX TRANSPOSE ALGORITHMS ON DISTRIBUTED MEMORY CONCURRENT COMPUTERS

Jaeyoung Choi [§]
Jack J. Dongarra [§†]
David W. Walker [§]

[§] Mathematical Sciences Section
Oak Ridge National Laboratory
P.O. Box 2008, Bldg. 6012
Oak Ridge, TN 37831-6367

[†] Department of Computer Science
University of Tennessee at Knoxville
107 Ayres Hall
Knoxville, TN 37996-1301

Date Published: October 1993

Research was supported by the Applied Mathematical Sciences Research Program of the Office of Energy Research, U.S. Department of Energy, by the Defense Advanced Research Projects Agency under contract DAAL03-91-C-0047, administered by the Army Research Office, and by the Center for Research on Parallel Computing

Prepared by the
Oak Ridge National Laboratory
Oak Ridge, Tennessee 37831
managed by
Martin Marietta Energy Systems, Inc.
for the
U.S. DEPARTMENT OF ENERGY
under Contract No. DE-AC05-84OR21400

Contents

PARALLEL MATRIX TRANSPOSE ALGORITHMS ON DISTRIBUTED MEMORY CONCURRENT COMPUTERS

Jaeyoung Choi

Jack J. Dongarra

David W. Walker

Abstract

This paper describes parallel matrix transpose algorithms on distributed memory concurrent processors. We assume that the matrix is distributed over a $P \times Q$ processor template with a block scattered data distribution. P, Q, and the block size can be arbitrary, so the algorithms have wide applicability.

The communication schemes of the algorithms are determined by the greatest common divisor (GCD) of P and Q. If P and Q are relatively prime, the matrix transpose algorithm involves *complete exchange* communication. If P and Q are not relatively prime, processors are divided into GCD groups and the communication operations are overlapped for different groups of processors. Processors transpose GCD wrapped diagonal blocks simultaneously, and the matrix can be transposed with LCM/GCD steps, where LCM is the least common multiple of P and Q.

The algorithms make use of non-blocking, point-to-point communication between processors. The use of nonblocking communication allows a processor to overlap the messages that it sends to different processors, thereby avoiding unnecessary synchronization.

Combined with the matrix multiplication routine, $\mathbf{C} = \mathbf{A} \cdot \mathbf{B}$, the algorithms are used to compute parallel multiplications of transposed matrices, $\mathbf{C} = \mathbf{A}^T \cdot \mathbf{B}^T$, in the PUMMA package [5]. Details of the parallel implementation of the algorithms are given, and results are presented for runs on the Intel Touchstone Delta computer.

1. Introduction

Matrix transposition is a fundamental matrix operation of linear algebra [8, 13] and arises in many scientific and engineering applications. On a uniprocessor, an algorithm involving a transposed matrix may not actually require the matrix data to be transposed in physical memory. Instead, it may be accessed simply by exchanging the row and column indices. However, in a distributed-memory multiprocessor environment, we cannot simply interchange the global row and column indices. Instead, the data must be physically moved from one processor to another.

Transposition of a matrix is a redistribution of its elements. Many researchers have considered the problem for different architectures. In 1972, Eklundh [7] considered the problem of directly accessing rows or columns of a matrix when its size is larger than the available high-speed storage. O'Leary [12] implemented an algorithm for transposing an $N \times N$ matrix on a one-dimensional systolic array. Azari, Bojanczyk and Lee [1] developed an algorithm for transposing an $M \times N$ matrix on an $N \times N$ mesh-connected array processor, and Johnsson and Ho [10] presented an algorithm for a Boolean n-cube, or hypercube.

Current advanced architecture computers possess hierarchical memories in which accesses to data in the upper levels of the memory hierarchy (registers, cache, and/or local memory) are faster than those in lower levels (shared or off-processor memory). To exploit the power of such machines, block-partitioned algorithms are preferred for dense linear algebra computations, in which operations are performed on submatrices, rather than individual matrix elements. In distributing matrix data over processors we therefore assume a block scattered decomposition [4, 6]. The block scattered decomposition can reproduce the most common data distributions used in dense linear algebra, as described briefly in the next section.

In this paper, the parallel matrix transpose algorithms are presented based on the block scattered decomposition. The algorithms are implemented on the Intel Touchstone Delta computer. The communication schemes of the algorithms are determined by the greatest common divisor (GCD) of the number of rows and columns (P and Q) of the processor template. If P and Q are relatively prime, the matrix transpose algorithm involves *complete exchange* communication. This is called all-to-all personalized communication, in which each of $N_p = P \cdot Q$ processors is required to send distinct subblocks to each of the remaining $N_p - 1$ processors, and receive distinct subblocks from each of them. Bokhari and Berryman [2] have developed binary exchange and quadrant exchange algorithms on a circuit switched mesh, where P and Q are powers of 2. The complete exchange communication in our transpose algorithms arises for any processor configuration, and is not limited to the case where P and Q are powers of 2. We implemented the complicated two-dimensional complete exchange communication problem by generalizing the one-dimensional complete exchange communication based on direct point-to-point communication. Details are discussed in Section 3.1.

We have presented the Parallel Universal Matrix Multiplication Algorithms (PUMMA) in [5] for performing $\mathbf{C} \Leftarrow \alpha\, op(\mathbf{A}) \cdot op(\mathbf{B}) + \beta\, \mathbf{C}$, where $op(\mathbf{X}) = \mathbf{X}$ or $\mathbf{X}^T$, based on the block scattered decomposition. One of the cases in the PUMMA package, $\mathbf{C} \Leftarrow \alpha\, \mathbf{A}^T \cdot \mathbf{B}^T + \beta\, \mathbf{C}$, is implemented in two steps ($\mathbf{T} \Leftarrow \alpha\, \mathbf{B} \cdot \mathbf{A}$; $\mathbf{C} \Leftarrow \mathbf{T}^T + \beta\, \mathbf{C}$). The second step involves parallel matrix transposition. The performance of this algorithm for evaluating $\mathbf{C} = \mathbf{A}^T \cdot \mathbf{B}^T$ is compared with the algorithm for evaluating $\mathbf{C} = \mathbf{A} \cdot \mathbf{B}$ on the Intel Delta machine in Section 4.

2. Design Issues

The way in which an algorithm's data are distributed over the processors of a concurrent computer has a major impact on the load balance and communication characteristics of the concurrent algorithm, and hence largely determines its performance and scalability. The block scattered decomposition provides a simple, yet general-purpose way of distributing a block-partitioned matrix on distributed memory concurrent computers. In the block scattered decomposition, described in detail in [4, 6], an $M \times N$ matrix is partitioned into blocks of size $r \times s$, and blocks separated by a fixed stride in the column and row directions are assigned to the same processor. If the stride in the column and row directions is P and Q blocks respectively, then we require that $P \cdot Q$ equal the number of processors, N_p. Thus, it is useful to imagine the processors arranged as a $P \times Q$ mesh, or template. The processor at position (p, q) $(0 \leq p < P$, $0 \leq q < Q)$ in the template is assigned the blocks indexed by,

$$(p + i \cdot P,\ q + j \cdot Q), \tag{1}$$

where $i = 0, \ldots, \lfloor (M_b - p - 1)/P \rfloor$, $j = 0, \ldots, \lfloor (N_b - q - 1)/Q \rfloor$, and $M_b \times N_b$ is the size in blocks of the matrix ($M_b = \lceil M/r \rceil$, $N_b = \lceil N/s \rceil$).

Blocks are scattered in this way so that good load balance can be maintained in parallel algorithms, such as LU factorization [3, 6]. The nonscattered decomposition (or pure block distribution) is just a special case of the scattered decomposition in which the block size is given by $r = \lceil M/P \rceil$ and $s = \lceil N/Q \rceil$. A purely scattered decomposition (or two-dimensional wrapped distribution) is another special case in which the block size is given by $r = s = 1$.

If P and Q are relatively prime, the matrix transpose algorithm involves a two-dimensional complete exchange communication, where each of N_p processors is required to send distinct subblocks to each of the remaining $N_p - 1$ processors, and receive distinct subblocks from each of them. We implemented the complicated two-dimensional complete exchange algorithm by generalizing the one-dimensional complete exchange algorithm. Three one-dimensional complete exchange communication schemes are shown in Figure 1, where each processor needs one subblock from each other processor, and the number in parentheses denotes the number of

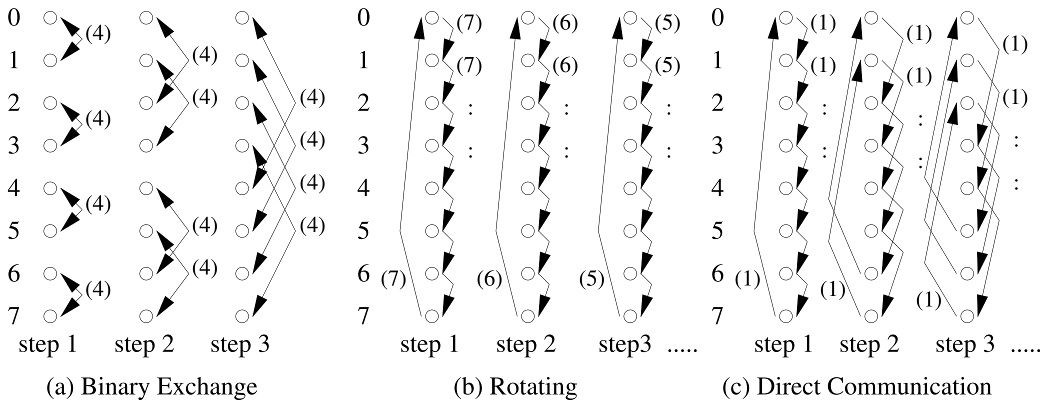


Figure 1: Three complete exchange communication schemes on 8 processors. The number in parentheses denotes the amount of data to transmit.

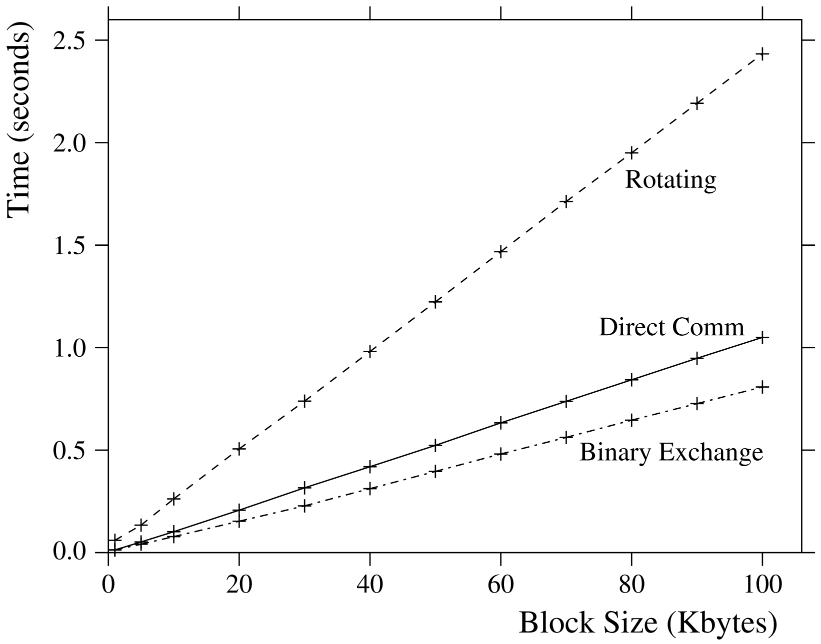


Figure 2: Comparison of three exchange communication schemes on 16 processors.

	0	1	2	3	4	5	6	7	8	9	10	11
0	0	1	2	0	1	2	0	1	2	0	1	2
1	3	4	5	3	4	5	3	4	5	3	4	5
2	0	1	2	0	1	2	0	1	2	0	1	2
3	3	4	5	3	4	5	3	4	5	3	4	5
4	0	1	2	0	1	2	0	1	2	0	1	2
5	3	4	5	3	4	5	3	4	5	3	4	5
6	0	1	2	0	1	2	0	1	2	0	1	2
7	3	4	5	3	4	5	3	4	5	3	4	5
8	0	1	2	0	1	2	0	1	2	0	1	2
9	3	4	5	3	4	5	3	4	5	3	4	5
10	0	1	2	0	1	2	0	1	2	0	1	2
11	3	4	5	3	4	5	3	4	5	3	4	5

(a) block distribution over template

	0	1	2	3	4	5	6	7	8	9	10	11
0	0	1	2	0	1	2	0	1	2	0	1	2
1	3	4	5	3	4	5	3	4	5	3	4	5
2	0	1	2	0	1	2	0	1	2	0	1	2
3	3	4	5	3	4	5	3	4	5	3	4	5
4	0	1	2	0	1	2	0	1	2	0	1	2
5	3	4	5	3	4	5	3	4	5	3	4	5
6	0	1	2	0	1	2	0	1	2	0	1	2
7	3	4	5	3	4	5	3	4	5	3	4	5
8	0	1	2	0	1	2	0	1	2	0	1	2
9	3	4	5	3	4	5	3	4	5	3	4	5
10	0	1	2	0	1	2	0	1	2	0	1	2
11	3	4	5	3	4	5	3	4	5	3	4	5

(b) LCM block distribution

Figure 3: A matrix with 12×12 blocks is distributed over a 2×3 processor template. (a) Each shaded and unshaded area represents different templates. The numbered squares represent blocks of elements, and the number indicates at which location in the processor template the block is stored – all blocks labeled with the same number are stored in the same processor. The *slanted* numbers, on the left and on the top of the matrix, represent indices of row block and column block, respectively. (b) The matrix has 2×2 LCM blocks. Blocks belong to the same processor if the relative locations of blocks are the same in each square LCM block. The definition of the LCM block is defined in the text.

subblocks to transmit.

The binary exchange scheme completes in $\lceil \log_2 P \rceil$ steps and the amount of data transmitted in each step is fixed at $2^{\lceil \log_2 P \rceil - 1}$ subblocks, where P is the number of processors. The rotating scheme can avoid network congestion, but requires $P-1$ steps and the amount of data transmitted in the initial steps is large. In the direct point-to-point communication scheme, the number of steps is the same in the rotating scheme, but the amount of data transmitted in each step is only one subblock.

The three schemes have been implemented on 16 nodes of the Delta and their performances are compared in Figure 2. The binary exchange and the rotating schemes are implemented with an odd-even communication scheme, which is preferable to a simultaneous communication scheme on the Delta [5, 11]. In this algorithm, odd-numbered processors send their own blocks and even-numbered processors receive them in the first step, and even-numbered processors send and odd-numbered processors receive in the next step. On $P = 2^d$ processors, as shown in Figure 2, the binary exchange scheme is the fastest. However, if P is not a power of 2, then this scheme becomes very complicated and may be slower than the direct communication scheme. The direct communication scheme is about 20% slower than the binary exchange scheme for the worst case ($P = 2^d$), but it is simple to implement on an arbitrary number of processors. We adopted the simple direct communication scheme for the implementation of the matrix transpose algorithms, which are explained in detail in the next section.

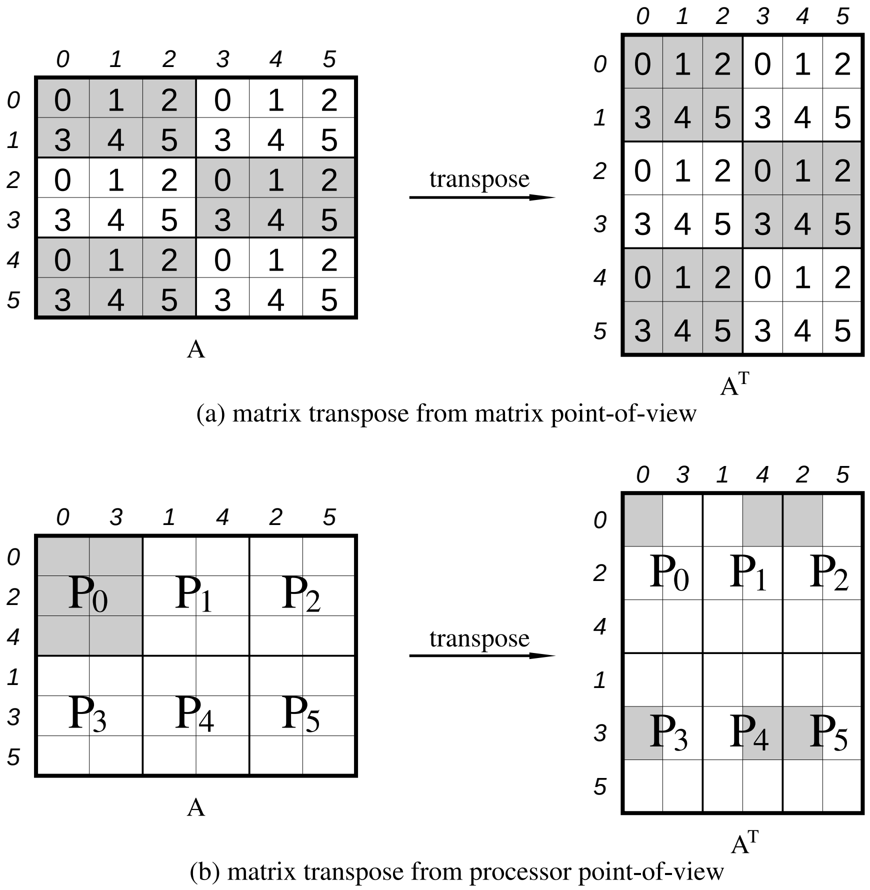


Figure 4: An example of matrix transpose for a block scattered decomposition, when $P = 2$, $Q = 3$, and $M_b = N_b = 6$.

3. Matrix Transpose Algorithms

We assume that a matrix is distributed over a two-dimensional processor mesh, or template, so that in general each processor has several blocks of the matrix as shown in Figure 3 (a), where a matrix with 12×12 blocks is distributed over a 2×3 template. Denoting the least common multiple of P and Q by LCM, we refer to a square of $LCM \times LCM$ blocks as an LCM block. Thus, the matrix may be viewed as a 2×2 array of LCM blocks, as shown in Figure 3 (b). The concept of the LCM block was introduced in [5], and is very useful for implementing algorithms that use a block scattered data distribution. Blocks belong to the same processor if their relative locations are the same in each square LCM block. An algorithm may be developed for the first LCM block, and then it can be directly applied to the other LCM blocks, which all have the same structure and the same data distribution as the first LCM block. That is, when an operation is executed on a block of the first LCM block, the same operation can be done simultaneously on other blocks, which have the same relative location in each LCM block.

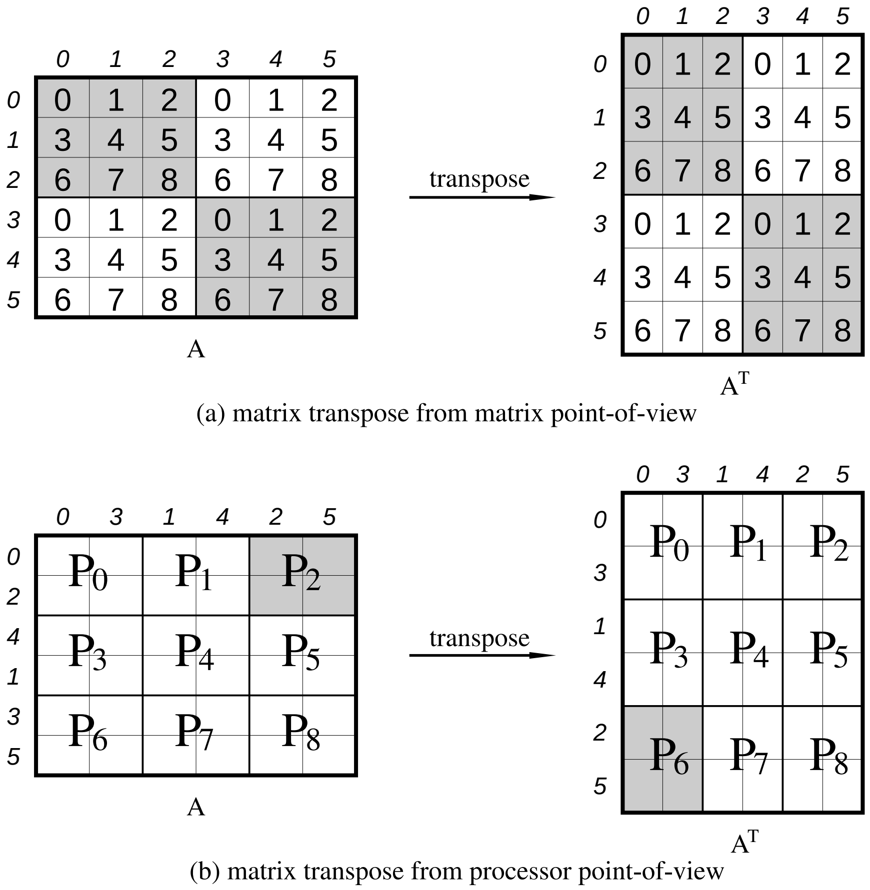


Figure 5: An example of matrix transpose for a block scattered decomposition, when $P = 3$, $Q = 3$, and $M_b = N_b = 6$.

We now describe parallel matrix transpose algorithms. A matrix $\mathbf{A}$, distributed over a $P \times Q$ processor template, has $M_b \times N_b$ blocks and each block consists of $r \times s$ elements, where r and s are arbitrary. Figure 4 (a) shows an example of a matrix transpose on a 2×3 template. If $\mathbf{A}$ is transposed, the transposed matrix $\mathbf{A}^T$ is distributed over the same $P \times Q$ template, and it has $N_b \times M_b$ blocks and each block has $s \times r$ elements. The elements of each block remain in the same block, but may be in a different processor, and each block is itself transposed. Figure 4 (b) shows the same example from the processor point-of-view. If P and Q are relatively prime, as shown in the figure, blocks in the first processor P_0 are scattered to all processors. As shown in Figure 5, which is the same example on a 3×3 square template, the blocks in each processor are not dispersed, but they are moved as one entity to a different processor. Parallel matrix transpose algorithms for the block scattered data distribution have several communication patterns determined by the greatest common divisor (GCD) of P and Q.

```
DO J = 0, Q − 1
    DO I = 0, P − 1
        [ Copy all blocks of A required by P⟨p + I, q − J⟩ to T1
          (in condensed and transposed form) ]
        [ Send T1 to P⟨p + I, q − J⟩ ]
        [ Receive T2 from P⟨p − I, q + J⟩ ]
        [ Copy T2 to C ]
    END DO
END DO
```

Figure 6: A parallel matrix transpose algorithm from the processor point-of-view, when P and Q are relatively prime. $P\langle p,q\rangle$ represents $P_{\mathrm{MOD}(p,P),\mathrm{MOD}(q,Q)}$. Processor $P_{p,q}$ ($0 \leq p < P$ and $0 \leq q < Q$) communicates with $P\langle p+I, q-J\rangle$ to send, and $P\langle p-I, q+J\rangle$ to receive based on direct point-to-point communication. '$p+I$' and '$q-J$' can be replaced with a different combination of signs.

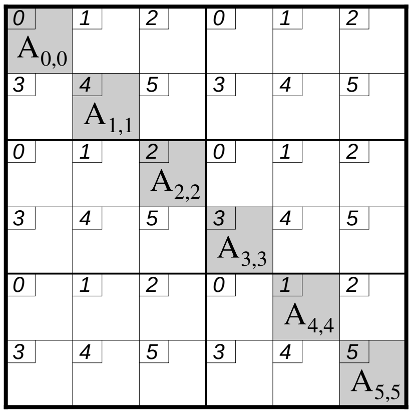
3.1. P and Q : relatively prime

We start with the simple case where P and Q are relatively prime, i. e. $GCD = 1$. In this case blocks in P_0 are scattered to all processors after being locally transposed as shown in Figure 4 (b). This case involves the two-dimensional complete exchange communication. That is, every processor needs to communicates with every other processor. The complete exchange problem is implemented by direct communication between sender and receiver.

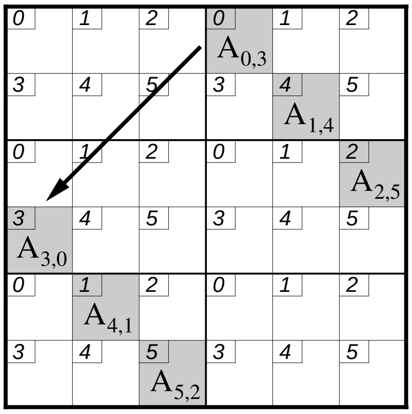
Figure 6 shows the pseudocode from the processor point-of-view, where $P\langle p,q\rangle$ represents $P_{\mathrm{MOD}(p,P),\mathrm{MOD}(q,Q)}$ in the processor template. Processor $P\langle p,q\rangle$ ($0 \leq p < P$ and $0 \leq q < Q$) starts to transpose blocks whose transposed blocks belong to itself. Then it deals with blocks whose transposition are in processors in the same column of the template ($P\langle p{-}i, q\rangle$, $0 \leq i < P$). The processor sends blocks to its top neighbor, $P\langle p-1, q\rangle$, and receives blocks from its bottom neighbor, $P\langle p+1, q\rangle$. Before sending the blocks, it is necessary to copy the blocks to be sent into a contiguous message buffer. Next it sends blocks to the next top processor, $P\langle p-2, q\rangle$, and receives blocks from the next bottom processor, $P\langle p+2, q\rangle$.

After it completes its operations with the processors in the same column, it sends blocks to the processors to the left in the template ($P\langle p-i, q-1\rangle$, $0 \leq i < P$), and receives blocks from the processors to the right ($P\langle p+i, q+1\rangle$). All operations are completed in $P \times Q = LCM$ steps.

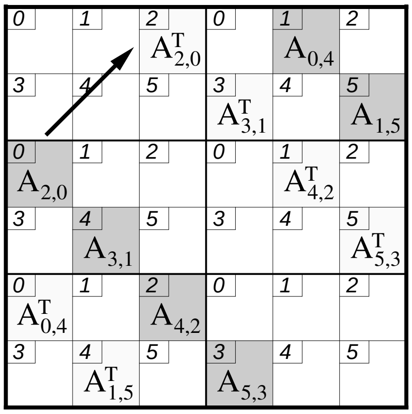
We interpret the algorithm from the matrix point-of-view. In the first LCM block, the above algorithm performs the operation by transposing one (wrapped) diagonal blocks at one step. (Actually the algorithm transposes every LCM diagonal blocks of the matrix at each step.) The first step of the algorithm in Figure 6 requires no explicit communication. It corresponds



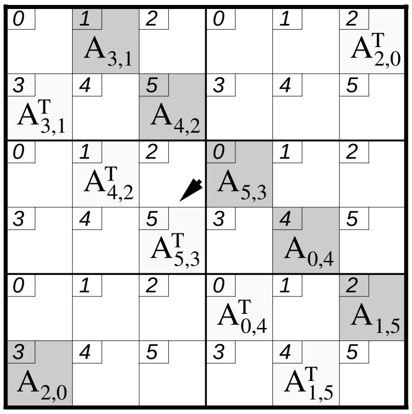
(a) zeroth diagonal (A(i,j), MOD(j-i,LCM)=0)



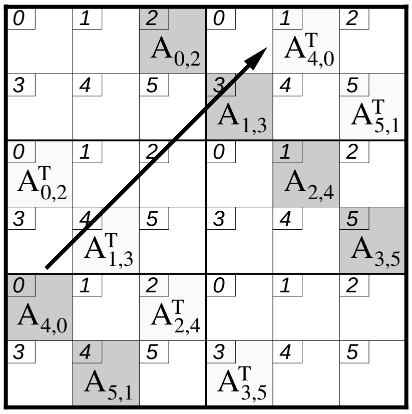
(b) third diagonal (A(i,j), MOD(j-i,LCM)=3)



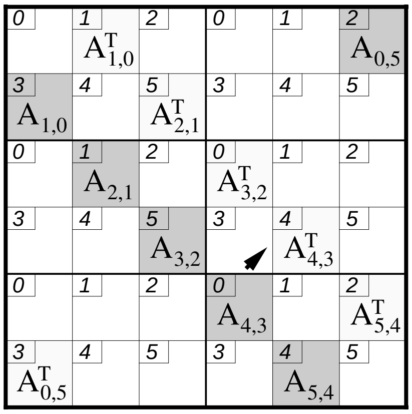
(c) fourth diagonal (A(i,j), MOD(j-i,LCM)=4)



(d) first diagonal (A(i,j), MOD(j-i,LCM)=1)



(e) second diagonal (A(i,j), MOD(j-i,LCM)=2)



(f) fifth diagoanl (A(i,j), MOD(j-i,LCM)=5)

Figure 7: Snapshots of matrix transposition when $P = 2$, $Q = 3$ and $M_b = N_b = 6$. The small *slanted* number in the upper left corner in each block represents processor identification number. One wrapped block diagonal is transposed in each step. The darkly shaded area represents blocks to be shifted, and lightly shaded area stands for their transpositions.

```
DO J = 0, Q - 1
    K = J        \* Deterimine K-th diagonal block to transpose *\
    WHILE (MOD(K, P) ≠ 0)   DO K = MOD(K + Q, LCM) END DO
    DO I = 0, P - 1
        [ Copy every (K : N_b : LCM)-th wrapped diagonal blocks in P_{p,q} to T1 ]
        [ Move T1 from P_{p,q} to P⟨p + I, q - J⟩ ]
        [ Copy the received T1 to C ]
        K = MOD(K + Q, LCM)
    END DO
END DO
```

Figure 8: A parallel matrix transpose algorithm from the matrix point-of-view, when P and Q are relatively prime. One diagonal block is transposed at one step. The 'While' statement should be executed until $\mathrm{MOD}(K, P)$ becomes 0. $(start : limit : intv)$ represents values of x, where $x = start + intv \cdot y$, $y = 0, 1, \cdots$, and x can't exceed $limit$.

to an in-place transpose of the diagonal blocks of $\mathbf{A}$ $(\mathbf{A}(i,i))$ (See Fig. 7(a)). Then every P diagonal blocks of $\mathbf{A}$ $(\mathbf{A}(i,j), \mathrm{MOD}(j-i,P) = 0)$ (See Fig. 7(b)) are transposed in the first outer loop $(J = 0)$ of Figure 6. In the next outer loop $(J = 1)$, the next P diagonal blocks $(\mathbf{A}(i,j), \mathrm{MOD}(j-i,P) = 1)$ are transposed. In Figures 7 (c) and (d), P_0 $(P\langle 0,0\rangle)$ sends blocks to P_2 $(P\langle 0,2\rangle)$, and receives from P_1 $(P\langle 0,1\rangle)$, where P_0, P_1 and P_2 are in the same row. Then P_0 sends blocks to P_5 $(P\langle 1,2\rangle)$, and receives from P_4 $(P\langle 1,1\rangle)$, and so on. The pseudocode for the algorithm from the matrix point-of-view is shown in Figure 8. Processors need to determine a diagonal block of $\mathbf{A}$ $(\mathbf{A}(i,j), \mathrm{MOD}(j-i, LCM) = K)$ which they start to transpose in the outer J loop in order to communicate with other processors in the same row of the template. The two lines before the inner DO-loop compute the value of K.

3.2. P and Q : not relatively prime

In the previous section, we have investigated the case when P and Q are relatively prime, which involves complete exchange communication. In this section we consider the case of $GCD > 1$. The former algorithm is a special case $(GCD = 1)$ of this algorithm.

Figure 9 shows an example of transposing a 12×12 matrix on a 4×6 template from the processor point-of-view. Each processor has its own 3×2 $(= LCM/P \times LCM/Q)$ array of blocks. The processors can transpose the matrix with 6 $(= LCM/P \cdot LCM/Q = LCM/GCD)$ communications steps. As shown in Figure 10, a processor $P\langle p,q\rangle$ starts to communicate with $P\langle \tilde{p},\tilde{q}\rangle$, where $\tilde{p}$ and $\tilde{q}$ are computed from p and q (details are explained later of this section). Once $P\langle \tilde{p},\tilde{q}\rangle$ is determined, it communicates with other processors, whose vertical and horizontal intervals are GCD from $P\langle \tilde{p},\tilde{q}\rangle$. The two loops of the algorithm in Figure 6 are

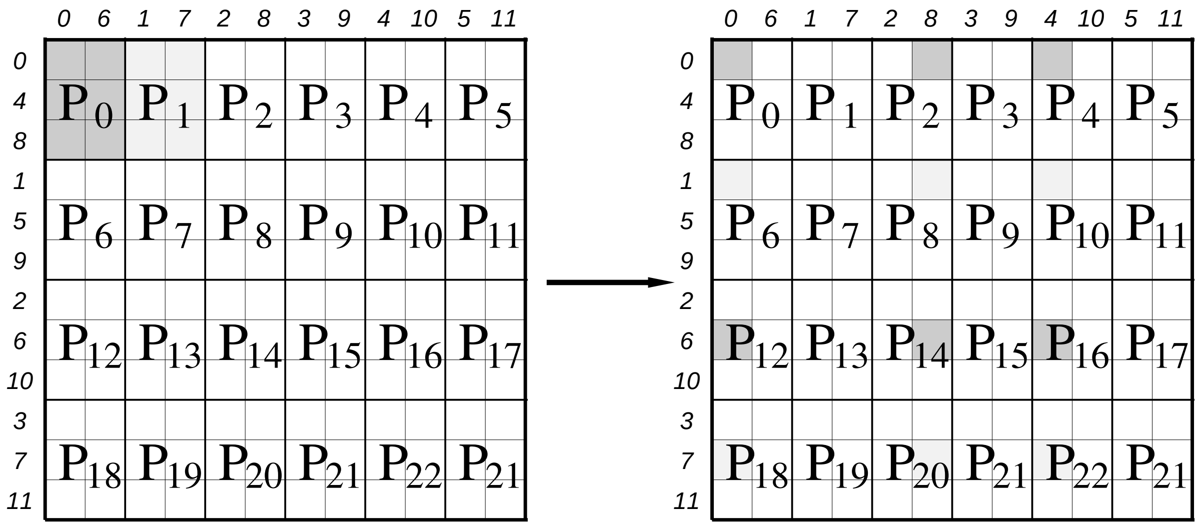


Figure 9: A matrix transpose example on a 4×6 template.

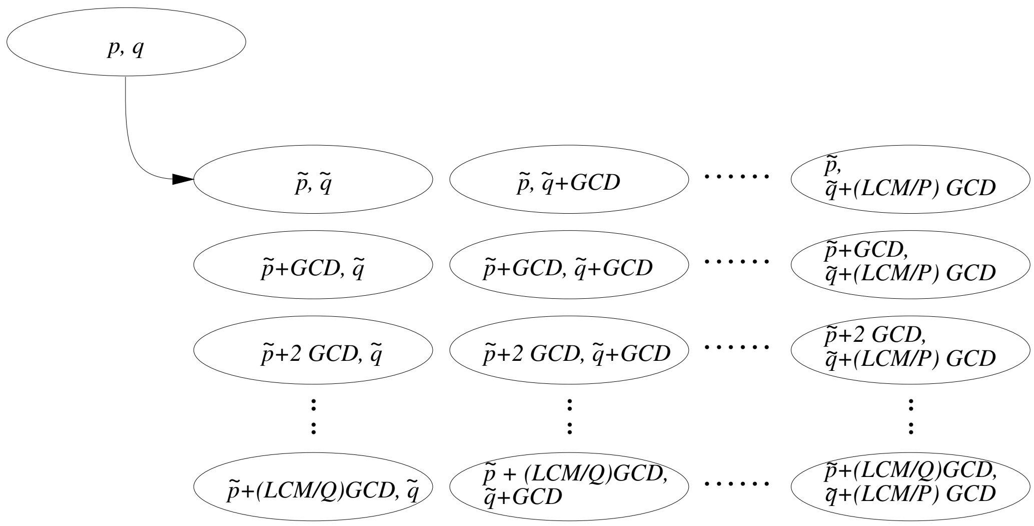
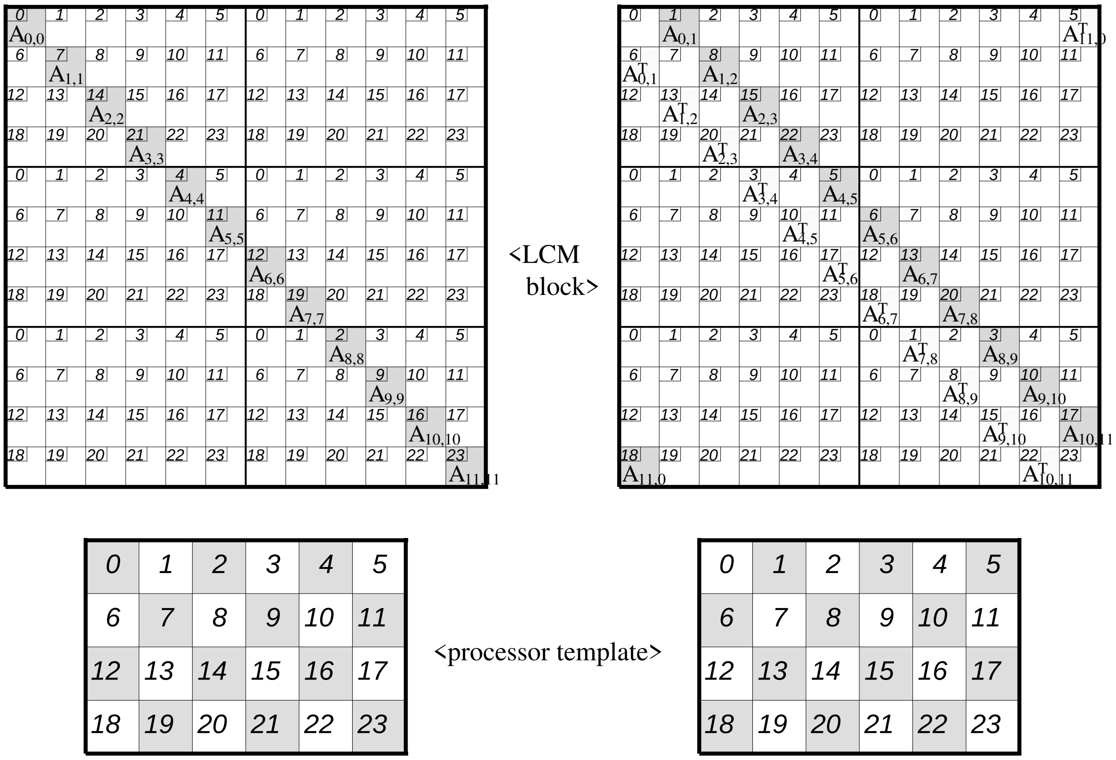


Figure 10: Processor map for communication. A processor $P\langle p, q\rangle$ starts to communicate with $P\langle \tilde{p}, \tilde{q}\rangle$, then it communicates with other processors, whose vertical and horizontal intervals are GCD from $P\langle \tilde{p}, \tilde{q}\rangle$.

PARDO $K = 1, GCD$
  $g = \mathrm{MOD}(q - p, GCD)$
  $\tilde{p} = \mathrm{MOD}(p + g, P)$; $\tilde{q} = \mathrm{MOD}(q - g, Q)$
  DO $J = 0, LCM/P - 1$
    DO $I = 0, LCM/Q - 1$
      [Copy to T1 (in condensed and transposed form) all blocks of **A** required by $P\langle \tilde{p} + I \times GCD, \tilde{q} - J \times GCD \rangle$]
      [Send T1 to $P\langle \tilde{p} + I \times GCD, \tilde{q} - J \times GCD \rangle$]
      [Receive T2 from $P\langle \tilde{p} - I \times GCD, \tilde{q} + J \times GCD \rangle$]
      [Copy T2 to **C**]
    END DO
  END DO
END PARDO

Figure 11: A modified matrix transpose algorithm from the processor point-of-view. Operations of GCD groups of processors are overlapped.



(a) transposing the zeroth wrapped block

(b) transposing the first wrapped block

Figure 12: Two snapshots of matrix transposition for transposing the zeroth and first wrapped block diagonals, when $P = 4$, $Q = 6$ and $M_b = N_b = 12$. In this example, transposing of even numbered wrapped block diagonals can be overlapped with that of odd numbered.

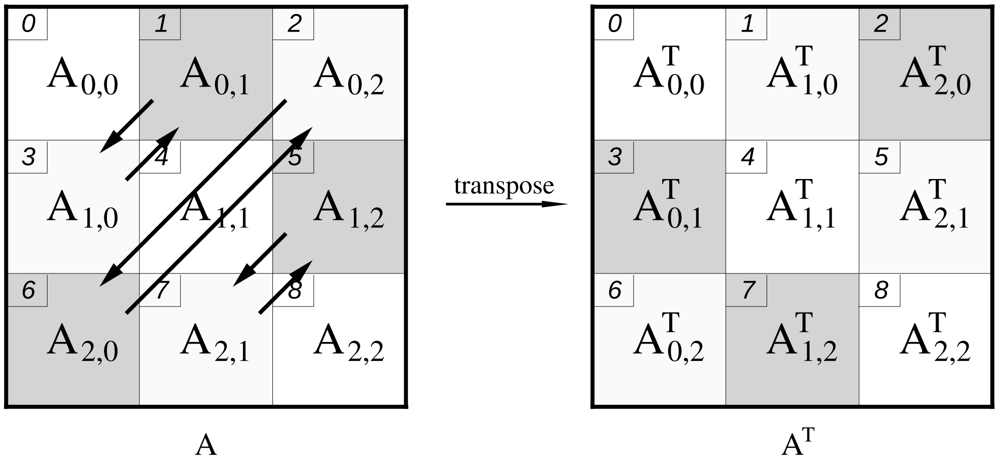


Figure 13: Matrix transposition when $P = Q = GCD = 3$. Processors transpose 3 ($= GCD$) diagonal blocks at one step, so that the transposition is done in one step.

```
PARDO K = 1, GCD
    g = MOD(q − p, GCD)
    p̃ = MOD(p + g, P); q̃ = MOD(q − g, Q)
    DO J = 0, LCM/P − 1
        K = J          \* Deterimine K-th diagonal block to transpose *\
        WHILE (MOD(K − g, P) ≠ 0)   DO K = MOD(K + Q, LCM) END DO
        DO I = 0, LCM/Q − 1
            [ Copy every (K : N_b : LCM)-th diagonal blocks in P⟨p, q⟩ to T1 ]
            [ Move T1 from P⟨p, q⟩ to P⟨p̃ + I × GCD, q̃ − J × GCD⟩ ]
            [ Copy the received T1 to C ]
            K = MOD(K + Q, LCM)
        END DO
    END DO
END PARDO
```

Figure 14: A modified matrix transpose algorithm from matrix point-of-view. GCD diagonal blocks are transposed simultaneously.

changed from Q and P to LCM/P and LCM/Q. The pseudocode of the algorithm is shown in Figure 11.

Figure 12 shows two snapshots of the same example, from the matrix point-of-view, to transpose the zeroth and the first diagonal blocks of $\mathbf{A}$ ($\mathbf{A}(i, j)$, $\mathrm{MOD}(j - i, LCM) = 0$ and 1, respectively.) The processors which have the blocks to send out are shaded at the bottom. In the example, only $P \cdot Q \,/\, GCD$ processors are involved in block communication in each step. Processors are split into GCD groups of processors, and a processor $P\langle p, q\rangle$ belongs to a group g if it has the same value of $g = \mathrm{MOD}(q - p, GCD)$. Processors in a group g send and receive their blocks to other processors in another group $g' = \mathrm{MOD}(GCD - g, GCD)$. The operations of each group can be overlapped.

The problem is interpreted from the matrix point-of-view. In general, for transposing the k-th diagonal block of $\mathbf{A}$ ($\mathbf{A}(i, j)$, $\mathrm{MOD}(j - i, LCM) = k$), a group of processors $g_k = \mathrm{MOD}(k, GCD)$ send the blocks to another group $g'_k = \mathrm{MOD}(GCD - g_k, GCD)$. Since the operations are overlapped over different groups of processors, processors transpose GCD diagonal blocks simultaneously. So, the matrix can be transposed with LCM/GCD steps. For the extreme case of $P = Q = GCD = 3$ as shown in Figure 13, processors transpose 3 ($= GCD$) diagonal blocks at one step. That is, the transposition is done in one step. A processor $P\langle p, q\rangle$ exchanges data with processor $P\langle q, p\rangle$. The pseudocode of the algorithm from the matrix point-of-view is shown in Figure 14. The code includes the case of $GCD = 1$.

96 processors		64 processors		48 processors	
$P \times Q$	Time (second)	$P \times Q$	Time (second)	$P \times Q$	Time (second)
6×16	0.404	4×16	0.596	4×12	0.652
8×12	0.330	8×8	0.572	6×8	0.546
12×8	0.307	16×4	0.475	8×6	0.527
16×6	0.381			12×4	0.547

Table 1: Dependence of performance on template configuration for fixed number of processors ($M = N = 2400$).

4. Results

In this section we present performance results of the parallel matrix transpose algorithms on the Intel Touchstone Delta computer. The performance of the transpose algorithms cannot be represented in floating point operations per second (flops), since there is no multiplications or additions in the transpose algorithms. The algorithms are combined with a matrix multiplication routine in the PUMMA to compute $\mathbf{C} = \alpha\,\mathbf{A}^T \cdot \mathbf{B}^T + \beta\,\mathbf{C}$ in two steps ($\mathbf{T} \Leftarrow \alpha\,\mathbf{B} \cdot \mathbf{A}$; $\mathbf{C} \Leftarrow \mathbf{T}^T + \beta\,\mathbf{C}$). We assume that $\alpha = 1$ and $\beta = 0$ in our test. The performance of $\mathbf{A}^T \cdot \mathbf{B}^T$ is compared with that of $\mathbf{A} \cdot \mathbf{B}$.

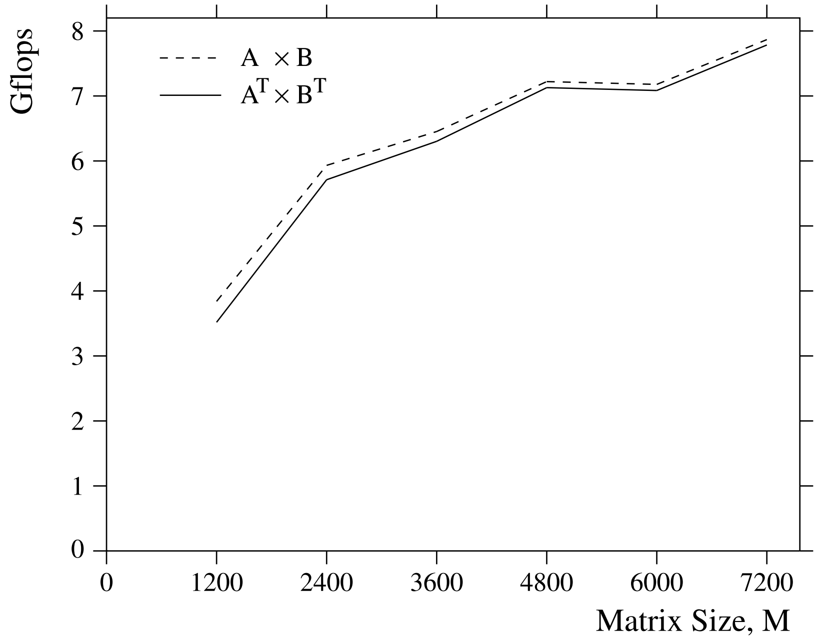


Figure 15: Performance comparison of $\mathbf{A}\cdot\mathbf{B}$ and $\mathbf{A}^T\cdot\mathbf{B}^T$ on 15×16 template. ($P=15, Q=16$, and $GCD=1$). $\mathbf{C}\Leftarrow\mathbf{A}^T\cdot\mathbf{B}^T$ is implemented in two steps, $\mathbf{T}\Leftarrow\mathbf{B}\cdot\mathbf{A}$, and then $\mathbf{C}\Leftarrow\mathbf{T}^T$.

Matrix elements are generated uniformly on the interval $[-1,1]$ in double precision. Conversions between measured runtimes and performance in gigaflops (Gflops) are made assuming an operation count of $2MNL$ for the multiplication of a $M\times L$ by a $L\times N$ matrix. In our test examples, all processors have the same number of blocks so there is no load imbalance. The algorithms were implemented with force type communication [9].

First, we considered how, for a fixed number of processors $N_p = P\times Q$, performance depends on the configuration of the processor template. Some typical results are presented in Table 1 for a fixed number of processors. In the test, the block size is fixed at 5×5 elements. It may be seen that the template configuration does have some effect on performance. The performance difference is between 19 and 24 %. For rectangular templates with different aspect ratios, the algorithm prefers those with small Q to those with small P. On the Delta, communication speed along vertical links seems faster than along horizontal links.

Figures 15 $\sim$ 19 show the performance of the routines on 15×16 ($GCD=1$, i.e., P and Q are relatively prime), 14×16 ($GCD=2$), 12×16 ($GCD=4$), 8×16 ($GCD=8$), and 16×16 ($P=Q=GCD=16$) templates, respectively. In all cases the block size is fixed at 5×5 elements. The solid and the dashed lines show the performance of $\mathbf{A}^T\cdot\mathbf{B}^T$ and $\mathbf{A}\cdot\mathbf{B}$, respectively. The difference of the two lines shows the loss of performance due to matrix transposition.

The transposed multiplication routine shows good performance relative to matrix multiplication. The loss of performance due to the matrix transpose routine is about 2 or 3 %. The

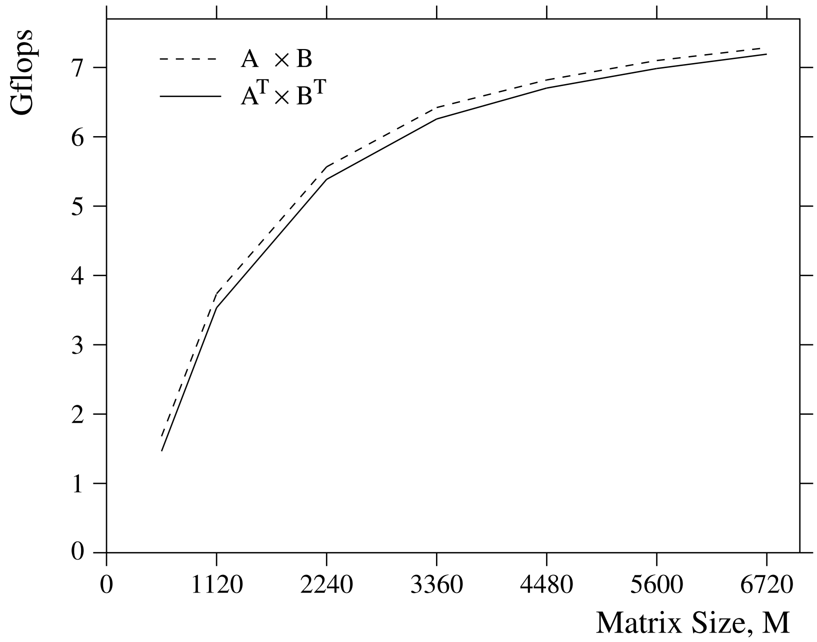


Figure 16: Performance comparison of $\mathbf{A} \cdot \mathbf{B}$ and $\mathbf{A}^T \cdot \mathbf{B}^T$ on 14×16 template. ($P = 14, Q = 16$, and $GCD = 2$)

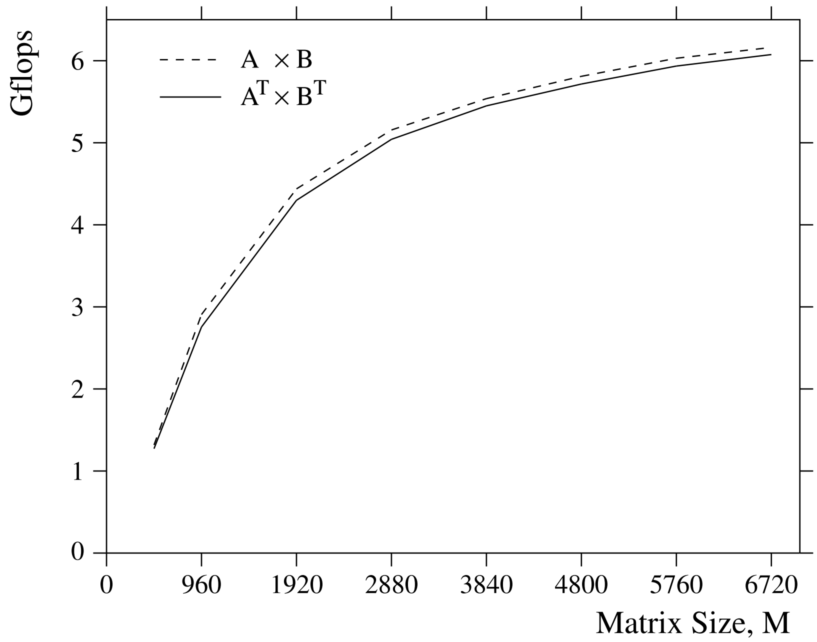


Figure 17: Performance comparison of $\mathbf{A} \cdot \mathbf{B}$ and $\mathbf{A}^T \cdot \mathbf{B}^T$ on 12×16 template. ($P = 12, Q = 16$, and $GCD = 4$)

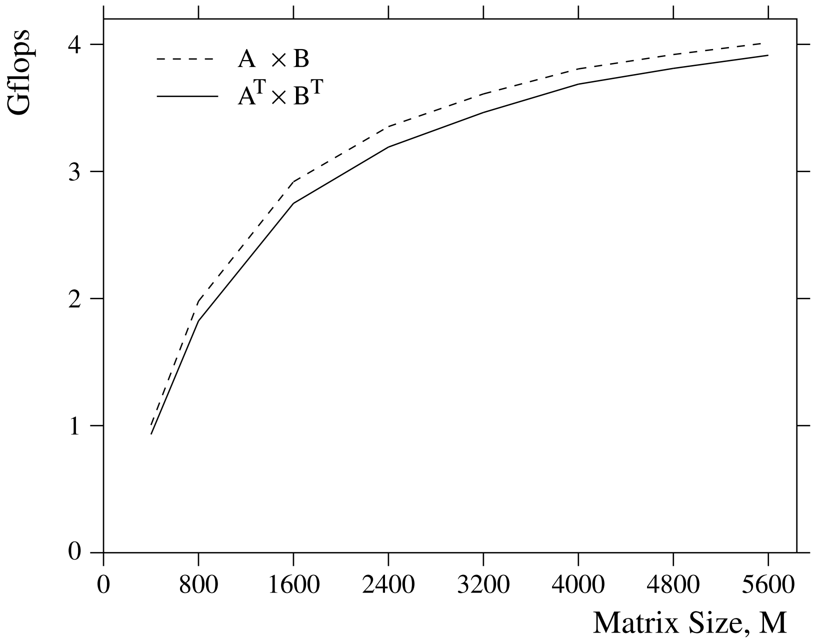


Figure 18: Performance comparison of $\mathbf{A} \cdot \mathbf{B}$ and $\mathbf{A}^T \cdot \mathbf{B}^T$ on 8×16 template. ($P = 8, Q = 16$, and $GCD = 8$)

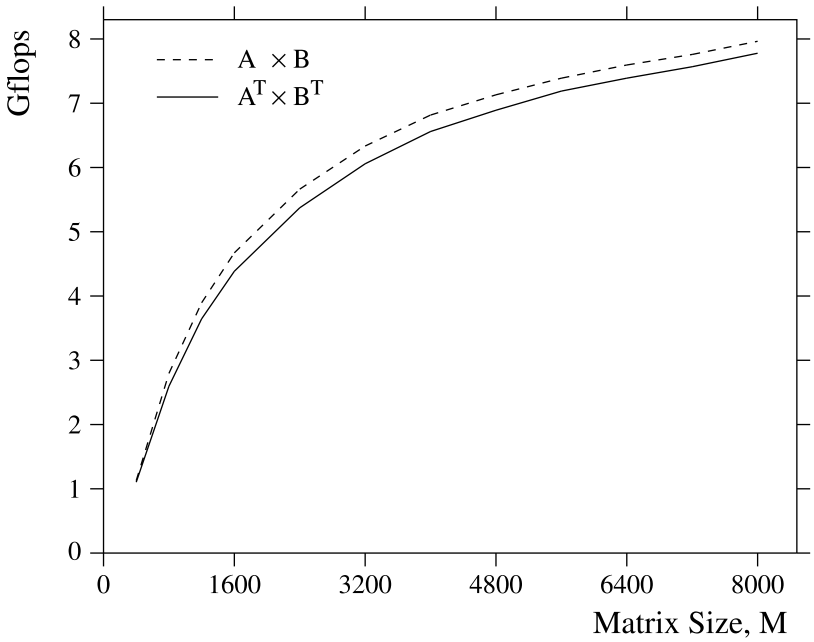


Figure 19: Performance comparison of $\mathbf{A} \cdot \mathbf{B}$ and $\mathbf{A}^T \cdot \mathbf{B}^T$ on 16×16 template. ($P = Q = GCD = 16$).

$P \times Q$	Matrix Size	Block Size	Time (second)
		1×1	1.857
8×16	4800×4800	5×5	1.612
		300×300	1.564
		1×1	1.280
12×16	4800×4800	5×5	0.893
		100×100	0.882
		1×1	1.484
14×16	5600×5600	5×5	1.193
		50×50	1.161
		1×1	1.740
15×16	6000×6000	5×5	1.437
		25×25	1.426
		1×1	1.967
16×16	6400×6400	5×5	1.967
		400×400	1.967

Table 2: Dependence of performance on block size.

transpose routine has excellent performance if P and Q are relatively prime. In other cases ($GCD \geq 2$), network congestion may degrade the performance of the routine.

$P \times Q$	Matrix Size $(\mathbf{A}, \mathbf{B})$	$\mathbf{A} \cdot \mathbf{B}$ (%)	$\mathbf{A}^T \cdot \mathbf{B}^T$ (%)
1×1	500×500	36.70 (100.0)	35.04 (100.0)
8×16	5600×5600	32.05 (87.3)	30.57 (87.3)
12×16	6720×6720	32.09 (87.4)	31.64 (90.3)
14×16	6720×6720	32.52 (88.6)	32.11 (91.6)
15×16	7200×7200	32.78 (89.3)	32.43 (92.6)
16×16	8000×8000	31.22 (85.1)	30.38 (86.7)

Table 3: Performance per node in Mflops. Block size is fixed to 5×5 elements. 1×1 template gives performance of assembly-coded matrix multiplication. Numbers in parentheses represent efficiency compared with the performance on 1 processor.

Table 2 shows how the block size affects the performance of the algorithms. It includes three cases of the block size, two extreme cases – the smallest and largest possible block sizes – and 5×5 block of elements. If $P = Q$, processors directly copy all blocks at once, so block size does not affect the performance. For the case of the smallest block size (1×1 element) when $P \neq Q$, processors make a copy element by element, so it takes a little more time to make a copy. The routines with the smallest block sizes are slower than those with the largest possible block sizes by between 15% and 31%. This difference is negligible, compared with the total elapsed time of the matrix multiplication.

Performance per node is shown in Table 3. The 1×1 template gives the performance of the assembly-coded level 3 BLAS matrix multiplication routine for the two cases $\mathbf{A} \cdot \mathbf{B}$ and

$\mathbf{A}^T \cdot \mathbf{B}^T$. Processors have about 85% efficiency for $\mathbf{A} \cdot \mathbf{B}$, and 87% for $\mathbf{A}^T \cdot \mathbf{B}^T$ if $P = Q = 16$. The routines perform better on templates for which $P \neq Q$. Processors achieve about 89%, and 93% of efficiency for each case if P and Q are relatively prime.

5. Conclusions and Remarks

We have presented parallel matrix transpose algorithms based on the block scattered decomposition. The algorithms have good performance for arbitrary processor configurations on the Intel Delta computer.

If P and Q are relatively prime, the transpose routine involves complete exchange communication on a two-dimensional template. We have approached this complicated problem with a direct point-to-point communication scheme (see Section 2). When P and Q are not relatively prime ($GCD > 1$), the processors' operations are overlapped over different groups, so that only LCM/GCD communications are required.

In our Fortran implementation, we assume that the first dimension of the matrix may be different from the number of rows of the matrix in a processor. Even when $P = Q$, the processor needs to copy blocks of $\mathbf{A}$ to a communication buffer before sending, and copy the received buffer to blocks of $\mathbf{C}$ after receiving.

The parallel matrix transpose algorithms have been combined with matrix multiplication routines. The integrated routines comprise a general-purpose matrix multiplication package, called PUMMA [5], for MIMD message-passing computers. The package has good performance for a wide range of decomposition parameters, that is, its performance depends weakly on processor configuration and block size.

The PUMMA package is currently available only for double precision real data, but will be implemented in the near future for other data types, i.e., single precision real and complex, and double precision complex. To obtain a copy of the software and a description of how to use it, send the message "`send pumma from misc`" to `netlib@ornl.gov`.

Acknowledgments

The authors would like to thank Eduardo D'Azevedo at ORNL for his helpful suggestions to improve the quality of the paper. This research was performed in part using the Intel Touchstone Delta System operated by the California Institute of Technology on behalf of the Concurrent Supercomputing Consortium. Access to this facility was provided through the Center for Research on Parallel Computing.

6. References

[1] N. G. Azari, A. W. Bojanczyk, and S.-Y. Lee. Synchronous and asynchronous algorithms

for matrix transposition on MCAP. In *SPIE Vol. 975, Advanced Algorithms and Architecture for Signal Processing III*, pages 277–288, 1988.

[2] S. H. Bokhari and H. Berryman. Complete exchange on a circuit switched mesh. In *Proceedings of the 1992 Scalable High Performance Computing Conference*, pages 300–306. IEEE Press, 1992.

[3] J. Choi, J. J. Dongarra, R. Pozo, and D. W. Walker. ScaLAPACK: A scalable linear algebra library for distributed memory concurrent computers. In *Proceedings of Fourth Symposium on the Frontiers of Massively Parallel Computation (McLean, Virginia)*. IEEE Computer Society Press, Los Alamitos, California, October 19-21 1992.

[4] J. Choi, J. J. Dongarra, and D. W. Walker. The design of scalable software libraries for distributed memory concurrent computers. In *Proceedings of Environment and Tools for Parallel Scientific Computing Workshop, (Saint Hilaire du Touvet, France)*. Elsevier Science Publishers, September 7-8 1992.

[5] J. Choi, J. J. Dongarra, and D. W. Walker. PUMMA : Parallel universal matrix multiplication algorithms on distributed memory concurrent computers. Technical Report TM-12252, Oak Ridge National Laboratory, Mathematical Sciences Section, August 1993.

[6] J. J. Dongarra, R. van de Geijn, and D. Walker. A look at scalable linear algebra libraries. In *Proceedings of the 1992 Scalable High Performance Computing Conference*, pages 372–379. IEEE Press, 1992.

[7] J. O. Eklundh. A fast computer method for matrix transposing. *IEEE Transactions on Computers*, 21:801–803, 1972.

[8] G. H. Golub and C. V. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, Baltimore, MD, 1989. Second Edition.

[9] Intel Corporation. *Touchstone Delta Fortran Calls Reference Manual*, April 1991.

[10] S. L. Johnsson and C.-T. Ho. Algorithms for matrix transposition on boolean n-cube configured ensemble architecture. *SIAM J. Matrix Anal. Appl*, 9:419–454, July 1988.

[11] R. Littlefield. Characterizing and tuning communications performance for real applications. In *Proceedings, First Intel Delta Application Workshop, CCSF-14-92, Pasadena, California*, pages 179–190, February 1992. presentation overheads.

[12] D. P. O'Leary. Systolic arrays for matrix transpose and other reorderings. *IEEE Transactions on Computers*, 36:117–122, January 1987.

[13] G. Strang. *Linear Algebra and Its Applications.* Harcourt Brace Jovanovich, Inc., San Diego, CA, 1988. Third Edition.