Accepted for publication in
Concurrency: Practice and Experience (1994)

PUMMA :
Parallel Universal Matrix Multiplication Algorithms on Distributed Memory Concurrent Computers[1]

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

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

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

Corresponding author:
Jaeyoung Choi
University of Tennessee
Department of Computer Science
101 Ayres Hall
Knoxville, TN 37996-1301
(615) 974-8492 (office)
(615) 974-8296 (fax)
choi@cs.utk.edu

[1] This work was supported in part by ARPA under contract number DAAL03-91-C-0047 administered by ARO, and in part by DOE under contract number DE-AC05-84OR21400 and by National Science Foundation Grant number ASC-ASC-9005933.

PUMMA :
Parallel Universal Matrix Multiplication Algorithms on Distributed Memory Concurrent Computers

Jaeyoung Choi
Jack J. Dongarra
David W. Walker

Abstract

This paper describes the Parallel Universal Matrix Multiplication Algorithms (PUMMA) on distributed memory concurrent computers. The PUMMA package includes not only the non-transposed matrix multiplication routine $\mathbf{C} = \mathbf{A} \cdot \mathbf{B}$, but also transposed multiplication routines $\mathbf{C} = \mathbf{A}^T \cdot \mathbf{B}$, $\mathbf{C} = \mathbf{A} \cdot \mathbf{B}^T$, and $\mathbf{C} = \mathbf{A}^T \cdot \mathbf{B}^T$, for a block scattered data distribution. The routines perform efficiently for a wide range of processor configurations and block sizes. The PUMMA together provide the same functionality as the Level 3 BLAS routine xGEMM. Details of the parallel implementation of the routines are given, and results are presented for runs on the Intel Touchstone Delta computer.

1 Introduction

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). One technique to more effectively exploit the power of such machines is to develop algorithms that maximize reuse of data held in the upper levels of the hierarchy, thereby reducing the need for more expensive accesses to lower levels. For dense linear algebra computations this can be done by using block-partitioned algorithms, that is by recasting algorithms in forms that involve operations on submatrices, rather than individual matrix elements. An example of a block-partitioned algorithm for LU factorization is given in [7,16]. The Level 3 Basic Linear Algebra Subprograms (BLAS) perform a number of commonly-used matrix-matrix operations, and are available in optimized form on most computing platforms ranging from workstations up to supercomputers [13].

The Level 3 BLAS have been successfully used as the building blocks of a number of applications, including LAPACK, a software library that uses block-partitioned algorithms for performing dense and banded linear algebra computations on vector and shared memory computers [2,3,9, 11,14]. On shared memory machines block-partitioned algorithms reduce the number of times that data most be fetched from shared memory, while on distributed memory machines they reduce the number of messages required to get data from other processors. Thus, there has been much interest recently in developing versions of the Level 3 BLAS for distributed memory concurrent computers [1,6,18,19].

An important routine in the Level 3 BLAS is xGEMM for performing matrix-matrix multiplication. The general purpose routine performs the following operations:

$$\begin{aligned}
\mathbf{C} &\Leftarrow \alpha\, \mathbf{A} \cdot \mathbf{B} + \beta\, \mathbf{C} \\
\mathbf{C} &\Leftarrow \alpha\, \mathbf{A}^T \cdot \mathbf{B} + \beta\, \mathbf{C} \\
\mathbf{C} &\Leftarrow \alpha\, \mathbf{A} \cdot \mathbf{B}^T + \beta\, \mathbf{C} \\
\mathbf{C} &\Leftarrow \alpha\, \mathbf{A}^T \cdot \mathbf{B}^T + \beta\, \mathbf{C}
\end{aligned}$$

where "$\cdot$" denotes matrix multiplication, $\mathbf{A}$, $\mathbf{B}$ and $\mathbf{C}$ are matrices, and α and β are scalars.

In this paper, we present the Parallel Universal Matrix Multiplication Algorithms (PUMMA) for performing the above operations on distributed memory concurrent computers. *Universal* means that the PUMMA include all the above multiplication routines and that their performance depends weakly on processor configuration and block size. A block scattered data distribution is used, which can reproduce many of the common data distributions used in dense linear algebra computations [8,16], as discussed in the next section. There have been many implementations of matrix multiplication algorithms on distributed memory machines [20,21,24]. Many of them are limited in their use since they are implemented with a pure block (non-scattered) distribution, or specific (not general-purpose) data distribution, and/or on square processor configurations with a specific number of processors (column and/or row numbers of processors are powers of 2). The PUMMA package eliminates all of these constraints.

The first part of this paper focuses on the design and implementation of the non-transposed matrix multiplication routine on distributed memory concurrent computers. We then deal with the other cases. A parallel matrix transpose algorithm, in which a matrix with a block scattered decomposition is transposed over a two-dimensional processor mesh, is presented in a separate paper [10]. All routines are implemented in Fortran 77 plus message passing and compared on the Intel Touchstone Delta computer.

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 (or block cyclic) 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 [8], a 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 $P\,Q$ equals the number of processors, N_p. Thus, it is useful to imagine the processors arranged as a $P \times Q$ mesh, or template. Then the processor at position (p, q) $(0 \leq p < P,\ 0 \leq q < Q)$ in the template is assigned the blocks indexed by,

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

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 of the matrix in blocks.

Blocks are scattered in this way so that good load balance can be maintained in algorithms, such as LU factorization [7,16], in which rows and/or columns of blocks of a matrix become eliminated as the algorithm progresses. However, for some of the distributed Level 3 BLAS routines a scattered decomposition does not improve load balance, and may result in higher concurrent overhead. The general matrix-matrix multiplication routine xGEMM is an example of such a routine for which a pure block (i.e., nonscattered) decomposition is optimal when considering the routine in isolation. However, xGEMM may be used in an application for which, overall, a scattered decomposition is best. We are faced with the choice of implementing a nonscattered distributed version of xGEMM, and transforming the data decomposition to this form if necessary each time xGEMM is called, or of providing a scattered version and thereby avoiding having to transform the data decomposition. We opt for the latter solution because it is more general, and does not *impose* on the user the necessity of potentially costly decomposition transformations. Since the nonscattered decomposition is just a special case of the scattered decomposition in which the block size is given by $r = \lceil M/P \rceil$ and $s = \lceil N/Q \rceil$, where the matrix size is $M \times N$, the user still has the option of using a nonscattered decomposition for the matrix multiplication and transforming between decompositions if necessary. The Basic Linear Algebra Communication Subprograms (BLACS) are intended to perform decomposition transformations of this type [4,12,17].

The decompositions of all matrices involved in a call to a Level 3 BLAS routine must be compatible with respect to the operation performed. To ensure compatibility we impose the condition that all the matrices be decomposed over the same $P \times Q$ processor template. Most distributed Level 3 BLAS routines will also require conditions on the block size to ensure compatibility. For example, in performing the matrix multiplication $\mathbf{C} = \mathbf{A} \cdot \mathbf{B}$, if the block size of $\mathbf{C}$ is $r \times s$ then that of $\mathbf{A}$ and $\mathbf{B}$ must be $r \times t$ and $t \times s$, respectively, where t is an arbitrary integer greater then 0.

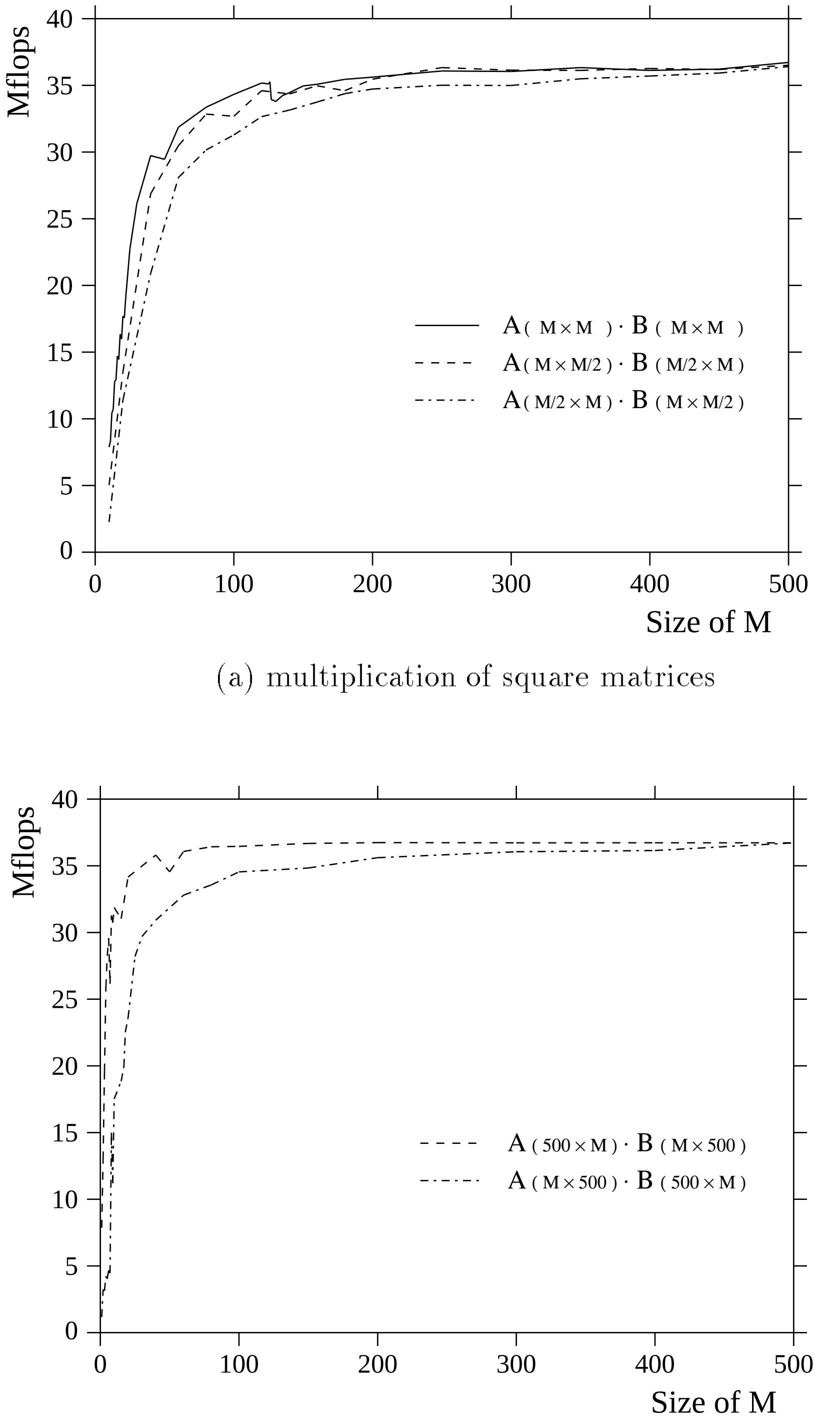
Another advantageous aspect of the distributed Level 3 BLAS is that often a distributed routine will call sequential Level 3 BLAS routines. For example, the distributed version of xGEMM, described in Section 3.2, consists of a series of steps in each of which each processor multiplies two local matrices by a call to the sequential version of xGEMM. Since highly optimized assembly-coded versions of the sequential Level 3 BLAS already exist on most processors we can take advantage of these in the distributed implementation.

Figure 1 (a) shows the performance of the DGEMM routine for square matrices on one i860 processor of the Intel Touchstone Delta. In general, performance improves with increasing matrix size and saturates for matrices of size greater than $M = 150$. Figure 1 (b) shows that for nonsquare matrices in our Fortran implementation, if matrix $\mathbf{A}$ is $500 \times M$ and $\mathbf{B}$ is $M \times 500$ ($M < 500$), then the matrix multiplication $\mathbf{A} \times \mathbf{B}$ is more efficient than $\mathbf{B} \times \mathbf{A}$. In both the square and nonsquare cases, the size of the matrices multiplied should be maximized in order to optimize performance of the sequential assembly-coded version of xGEMM routines. Thus, in the PUMMA routines, instead of multiplying individual blocks successively on each processor, blocks are conglomerated to form larger matrices which are then multiplied.

The distributed Level 3 BLAS routines have similar argument lists to the sequential Level 3 BLAS routines. In the distributed xGEMM routine, for example, original matrices $\mathbf{A}$ and $\mathbf{B}$, are preserved as in the sequential routine. Users, who are familiar with the sequential routines, should have no difficulty in using the distributed routines.

3 Algorithms

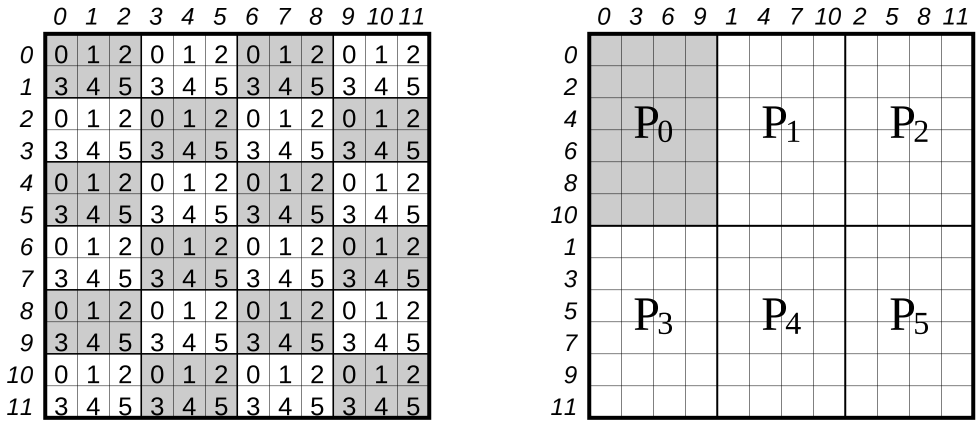
To illustrate the basic parallel algorithm we consider a matrix $\mathbf{A}$ distributed over a 2-dimensional processor template as shown in Figure 2 (a), where $\mathbf{A}$ with 12×12 blocks is distributed over a 2×3 template. If the matrix distribution is seen from the processor point-of-view



(a) multiplication of square matrices

(b) multiplication of non-square matrices

Figure 1: Performance of DGEMM on one i860 processor of the Delta. (a) The routine is tested with $\mathbf{A}_{M\times M}\cdot\mathbf{B}_{M\times M}$, $\mathbf{A}_{M\times M/2}\cdot\mathbf{B}_{M/2\times M}$, and $\mathbf{A}_{M/2\times M}\cdot\mathbf{B}_{M\times M/2}$, where "·" denotes matrix multiplication, and (b) tested with $\mathbf{A}_{500\times M}\cdot\mathbf{B}_{M\times 500}$ and $\mathbf{A}_{M\times 500}\cdot\mathbf{B}_{500\times M}$.



	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) matrix point-of-view

(b) processor point-of-view

Figure 2: A matrix $\mathbf{A}$ with 12×12 blocks is distributed over a 2×3 processor template. (a) From the matrix point-of-view. Each shaded and unshaded area represents a different template. 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 global indices of block row and block column, respectively. (b) From the processor point-of-view, each processor has 6×4 blocks.

as in Figure 2 (b), each processor has several blocks of the matrix and the scattered blocks, $A(0,0), A(2,0), A(4,0), \cdots, A(10,0)$ are vertically adjacent in the 2-dimensional array in the first processor P_0, and can be accessed as one long block column $A(0:11:2,0)$. In the same way, $A(0,0), A(0,3), A(0,6), A(0,9)$ are horizontally adjacent in P_0, and can be accessed as one long block row $A(0,0:11:3)$. We exploit this property in implementing the algorithms to deal with larger matrices instead of several small individual blocks. We assume data are stored by column in both our Fortran 77 and message passing implementation. In general, the algorithms are presented from the matrix point-of-view, which is simpler and easier to understand. In dealing with the implementation details, we explain the algorithms from the processor point-of-view.

3.1 The Basic Matrix Multiplication Algorithm

Our matrix multiplication algorithm is a block scattered variant of that of Fox, Hey, and Otto [20], that deals with arbitrary rectangular processor templates.

Suppose the matrix $\mathbf{A}$ has M_b block rows and L_b block columns, and the matrix $\mathbf{B}$ has L_b block rows and N_b block columns. Block (I,J) of $\mathbf{C}$ is then given by

$$C(I,J) = \sum_{K=0}^{L_b-1} A(I,K) \cdot B(K,J) \qquad (2)$$

```
DO $K = 0, L_b - 1$
   [Columncast one block of $\mathbf{B}$ $(B(I, \mathrm{MOD}(I+K, N_b)), I = 0 : L_b)$
    along each column across template]
   PARDO $I = 0, M_b - 1$
      $KP = \mathrm{MOD}(K + I, L_b)$
      PARDO $J = 0, N_b - 1$
         $\mathrm{C}(I,J) = \mathrm{C}(I,J) + \mathrm{A}(I,KP) \cdot \mathrm{B}(KP,J)$
      END PARDO
   END PARDO
   [Roll $\mathbf{A}$ leftwards]
END DO
```

Figure 3: A distributed block scattered matrix multiplication algorithm. The PARDO's indicate over which indices the data are decomposed. All indices refer to blocks of elements. Communication phases are indicated in square brackets.

where $I = 0, 1, \ldots, M_b - 1$, $J = 0, 1, \ldots, N_b - 1$. In Equation 2 the order of summation is arbitrary.

Fox et al. initially considered only the case of square matrices in which each processor contains a single row or a single column of blocks. That is, the blocks that start the summation lie along the diagonal. The summation is started at a different point for each block row of **C** so that in the phase of the parallel algorithm corresponding to summation index K, $A(I,K)$ and $B(K,J)$ can be multiplied in the processor to which $C(I,J)$ is assigned.

This requires each processor containing a block of **B** to be multiplied in step K to broadcast that block along the column of the processor template at the start of the step. Also **A** must be rolled leftwards at the end of the step so that each column is overwritten by the one to the right, with the first column wrapping round to overwrite the last column. The pseudocode for this algorithm is shown in Figure 3. Another variant of this algorithm involves broadcasting blocks of **A** over rows, and rolling **B** upwards.

In Figure 3 and subsequent figures a "columncast" is a communication phase in which one data item (typically a block, or set of blocks) is taken from each block column of the matrix and is broadcast to all the other processors in the same column of the processor template. A "rowcast" is similar, but broadcasts a data item from each block row of the matrix to all processors in the same row of the template.

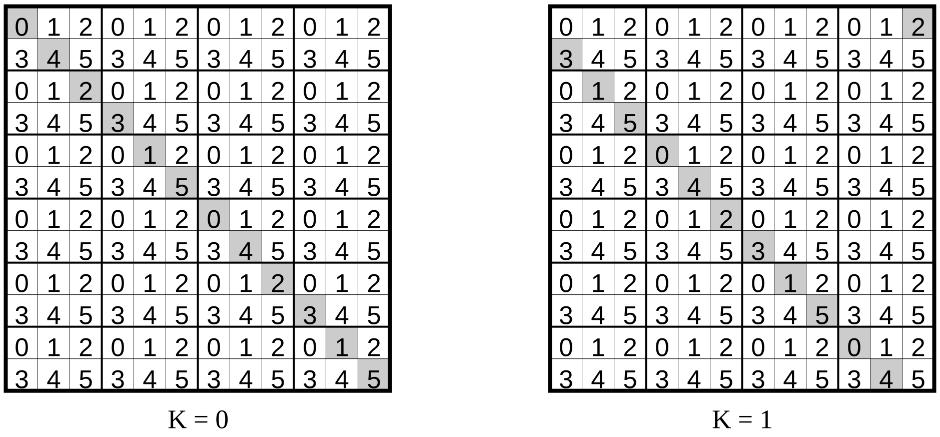


Figure 4: Snapshot of SDB algorithm. The blocks of the matrix **B** communicated in the first two stages of the matrix multiplication algorithm are shown shaded. In this case $P = 2$ and $Q = 3$. In each stage, only one wrapped diagonal is columncast. The total number of stages is L_b.

3.2 Matrix Multiplication Algorithm with Block Scattered Decomposition

We now consider the multiplication of matrices distributed with a block scattered decomposition. The block sizes for matrices **A** and **B** are $r \times s$ and $s \times t$, respectively, where r, s, and t are arbitrary. In this case the summation in row I starts at $K = I$, so the blocks of **B** broadcast in each stage lie along diagonal stripes. The parallel algorithm proceeds in L_b stages, in each of which one block of **B** is broadcast along each column of the template, and **A** is rolled leftwards. We call this the SDB (Single Diagonal Broadcast) algorithm.

Figure 4 shows, from the matrix point-of-view, the wrapped diagonal blocks of **B** broadcast in the first two stages of the SDB algorithm, where **B** with 12×12 blocks is distributed over a 2×3 template. Only one wrapped diagonal is columncast in each stage. In implementing the algorithm, the size of the submatrices multiplied in each processor should be maximized to optimize the performance of the sequential xGEMM routine. ¿From the processor point-of-view, as shown in Figure 2 (b), the first processor P_0 has $A(0:11:2, 0:11:3)$ and $B(0:11:2, 0:11:3)$, and it will have $C(0:11:2, 0:11:3)$ after the computation. In the first stage of Figure 4 ($K = 0$), P_0 multiplies $A(0,0), A(2,0), \cdots, A(10,0)$ with $B(0,0)$. These operations can be combined as one matrix multiplication since blocks of $A(0,0), A(2,0), \cdots, A(10,0)$ are vertically adjacent in P_0. The processor multiplies a long block column of **A** ($A(0:11:2, 0)$) with one block $B(0,0)$. This is the reason why we prefer a scheme of columnwise broadcasting **B** to a scheme of rowwise broadcasting **A** in our Fortran implementation, where 2-dimensional arrays are stored by columns.

Denoting the least common multiple of P and Q by LCM, we refer to a square of $LCM \times LCM$

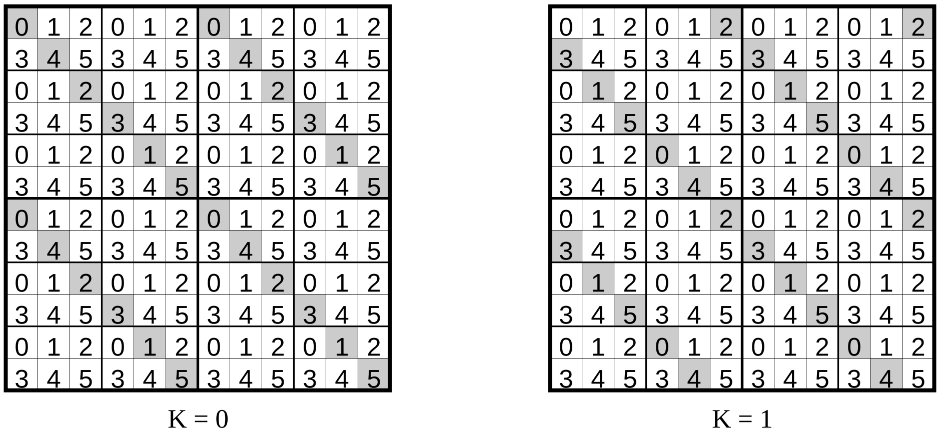
```
DO $K1 = 0, LCM - 1$
   [Columncast $L_b/LCM$ blocks of $\mathbf{B}$ ($B(I, J : N_b : LCM)$, $I = 0 : L_b$,
    $J = \mathrm{MOD}(I + K1, LCM)$) along each column of template]
   DO $K2 = 0, L_b/LCM - 1$
      $K = K1 + K2 \times LCM$
      PARDO $I = 0, M_b - 1$
         $KP = \mathrm{MOD}(K + \mathrm{MOD}(I, LCM), L_b)$
         PARDO $J = 0, N_b - 1$
            $\mathrm{C}(I, J) = \mathrm{C}(I, J) + \mathrm{A}(I, KP) \cdot \mathrm{B}(KP, J)$
         END PARDO
      END PARDO
   END DO
   [Roll $\mathbf{A}$ leftwards]
END DO
```

Figure 5: MDB1 algorithm, which is a distributed matrix multiplication algorithm suitable for a block scattered decomposition. The outer K loop has been split into loops over $K1$ and $K2$ so that the communication for several steps can be sent in a single message.

blocks as an LCM block. Blocks belong to the same processor if their relative locations are the same in each square LCM block. The concept of the LCM block is very useful, since an algorithm may be developed for the first LCM block, and then be applied to the other LCM blocks, which all have the same structure and 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.

For a block scattered decomposition the communication latency can be reduced by performing multiple instances of the outer K loop (see Figure 3) together. The communication latency is reduced when instances of the outer K loop separated by LCM are grouped together, as shown in Figure 5. We call this the MDB1 (Multiple Diagonal Broadcast 1) algorithm. In this case the parallel algorithm proceeds in LCM stages, in each of which $\lceil L_b/LCM \rceil$ blocks of the $\mathbf{B}$ matrix are broadcast down each column of the template by a single communication phase in the outer loop. In Figure 6 we show the two ($\lceil L_b/LCM \rceil = 12/6$) wrapped diagonal blocks of $\mathbf{B}$ broadcast in the first two stages of the algorithm. The size of the submatrices multiplied in each processor cannot be increased and it is the same as in the SDB algorithm.

The communication latency can be reduced even further by noting that the data for matrix $\mathbf{A}$



K = 0

0	1	2	0	1	2	**0**	1	2	0	1	2
3	**4**	5	3	4	5	3	**4**	5	3	4	5
0	1	**2**	0	1	2	0	1	**2**	0	1	2
3	4	5	**3**	4	5	3	4	5	**3**	4	5
0	1	2	0	**1**	2	0	1	2	0	**1**	2
3	4	5	3	4	**5**	3	4	5	3	4	**5**
0	1	2	0	1	2	**0**	1	2	0	1	2
3	**4**	5	3	4	5	3	**4**	5	3	4	5
0	1	**2**	0	1	2	0	1	**2**	0	1	2
3	4	5	**3**	4	5	3	4	5	**3**	4	5
0	1	2	0	**1**	2	0	1	2	0	**1**	2
3	4	5	3	4	**5**	3	4	5	3	4	**5**

K = 1

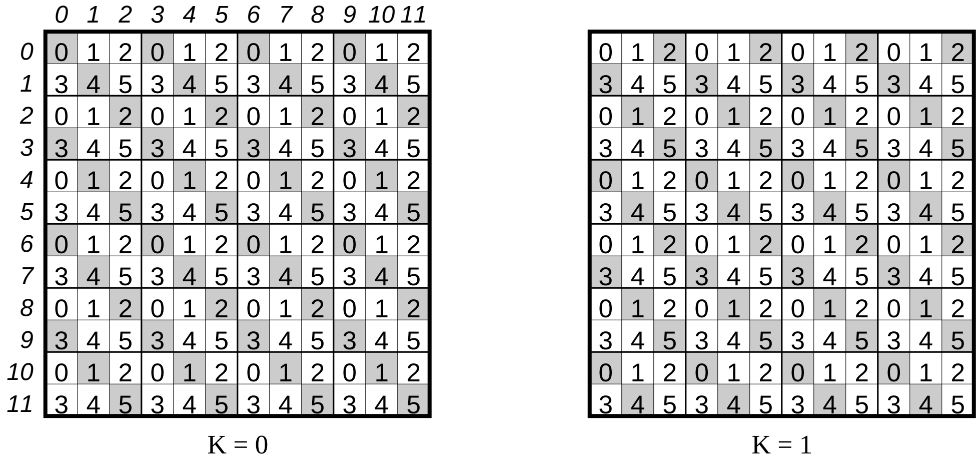
0	1	2	0	1	**2**	0	1	2	0	1	**2**
3	4	5	3	4	5	**3**	4	5	3	4	5
0	**1**	2	0	1	2	0	**1**	2	0	1	2
3	4	**5**	3	4	5	3	4	**5**	3	4	5
0	1	2	**0**	1	2	0	1	2	**0**	1	2
3	4	5	3	**4**	5	3	4	5	3	**4**	5
0	1	2	0	1	**2**	0	1	2	0	1	**2**
3	4	5	3	4	5	**3**	4	5	3	4	5
0	**1**	2	0	1	2	0	**1**	2	0	1	2
3	4	**5**	3	4	5	3	4	**5**	3	4	5
0	1	2	**0**	1	2	0	1	2	**0**	1	2
3	4	5	3	**4**	5	3	4	5	3	**4**	5

Figure 6: Snapshot of MDB1 algorithm. In this case $P = 2$, $Q = 3$, and so the LCM of P and Q is 6. In each stage, two ($\lceil L_b/LCM \rceil = 12/6$) wrapped diagonals are columncast. The total number of stages is LCM.

returns to the processor in which it started after $\mathbf{A}$ has been rolled Q times. Thus, we introduce a third variant of the parallel algorithm that proceeds in Q stages, in each of which $\lceil L_b/Q \rceil$ blocks of $\mathbf{B}$ are broadcast down each template column by a single communication phase in the outer loop. Figure 7 shows the four ($\lceil L_b/Q \rceil = 12/3$) wrapped diagonal blocks of $\mathbf{B}$ broadcast in each stage. The pseudocode for this version of the algorithm is the same as that shown in Figure 5, except that "LCM" is replaced by "Q." This is called the "MDB2 (Multiple Diagonal Broadcast 2)" algorithm.

In implementing the MDB2 algorithm, the granularity of the algorithm is increased. In the first stage shown in Figure 7 ($K1 = 0$), the first processor P_0 multiplies a column block $\mathbf{A}$ ($A(0 : 11 : 2, 0)$) with $B(0,0), B(0,3), B(0,6)$ and $B(0,9)$. These blocks of $\mathbf{B}$ are horizontally adjacent in the 2-dimensional submatrix in P_0, and form a long block row $B(0, 0 : 11 : 3)$. These operations are replaced by one multiplication. P_0 multiplies a long block column of $\mathbf{A}$ ($A(0 : 11 : 2, 0)$) with a long block row of $\mathbf{B}$ ($B(0, 0 : 11 : 3)$). The combined multiplication looks like a *block* version of the *outer product* operation. Since $\lceil L_b/LCM \rceil = 2$, P_0 needs to do another outer product operation at the same step, $A(0 : 11 : 2, 6)$ with $B(6, 0 : 11 : 3)$, as shown in Figure 8 (a).

In MDB2 algorithm, the granularity of the algorithm is maximized. P_0 has two block rows of $\mathbf{B}$ to broadcast ($B(0, 0 : 11 : 3)$ and $B(6, 0 : 11 : 3)$), which are condensed to one large matrix ($B([0,6], 0 : 11 : 3)$) for economical communications, where $0 : 11 : 6 = [0,6]$. If block columns of $\mathbf{A}$ are presorted with radix LCM in the beginning of the algorithm (or radix LCM/Q in each processor) as shown in Figure 8 (b), two block columns of $\mathbf{A}$ ($A(0 : 11 : 2, 0)$ and $A(0 : 11 : 2, 6)$) are accessed as one large matrix ($A(0 : 11 : 2, [0,6])$). Now, P_0 can complete its operation with



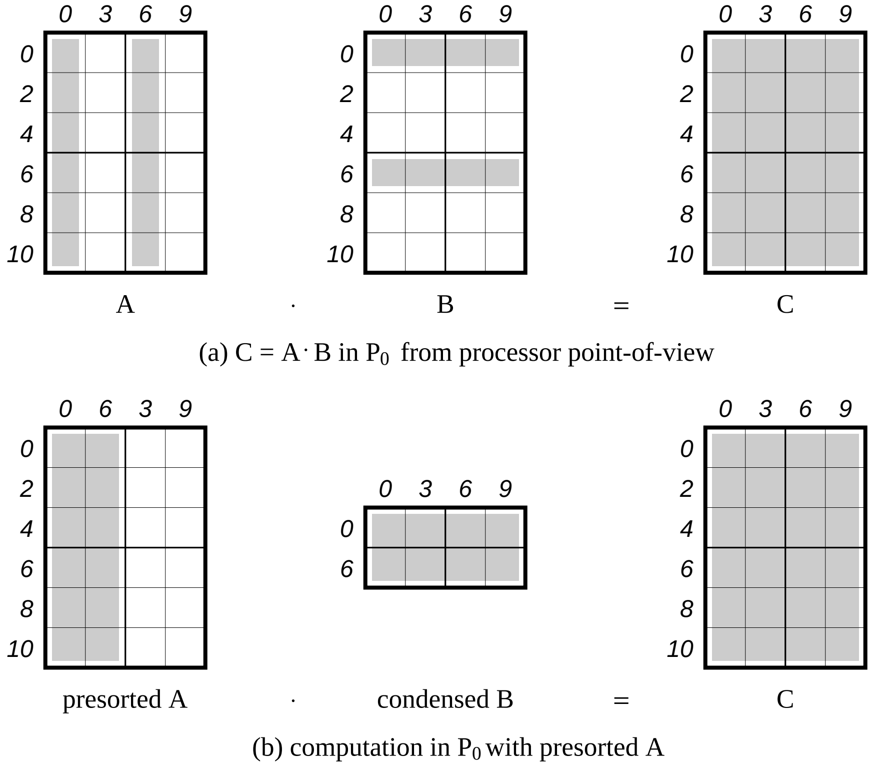
	0	1	2	3	4	5	6	7	8	9	10	11
0	0	1	2	0	1	2	0	1	2	0	1	2
1	3	4	5	3	4	5	3	4	5	3	4	5
2	0	1	2	0	1	2	0	1	2	0	1	2
3	3	4	5	3	4	5	3	4	5	3	4	5
4	0	1	2	0	1	2	0	1	2	0	1	2
5	3	4	5	3	4	5	3	4	5	3	4	5
6	0	1	2	0	1	2	0	1	2	0	1	2
7	3	4	5	3	4	5	3	4	5	3	4	5
8	0	1	2	0	1	2	0	1	2	0	1	2
9	3	4	5	3	4	5	3	4	5	3	4	5
10	0	1	2	0	1	2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

K = 0

0	1	2	0	1	2	0	1	2	0	1	2
3	4	5	3	4	5	3	4	5	3	4	5
0	1	2	0	1	2	0	1	2	0	1	2
3	4	5	3	4	5	3	4	5	3	4	5
0	1	2	0	1	2	0	1	2	0	1	2
3	4	5	3	4	5	3	4	5	3	4	5
0	1	2	0	1	2	0	1	2	0	1	2
3	4	5	3	4	5	3	4	5	3	4	5
0	1	2	0	1	2	0	1	2	0	1	2
3	4	5	3	4	5	3	4	5	3	4	5
0	1	2	0	1	2	0	1	2	0	1	2
3	4	5	3	4	5	3	4	5	3	4	5

K = 1

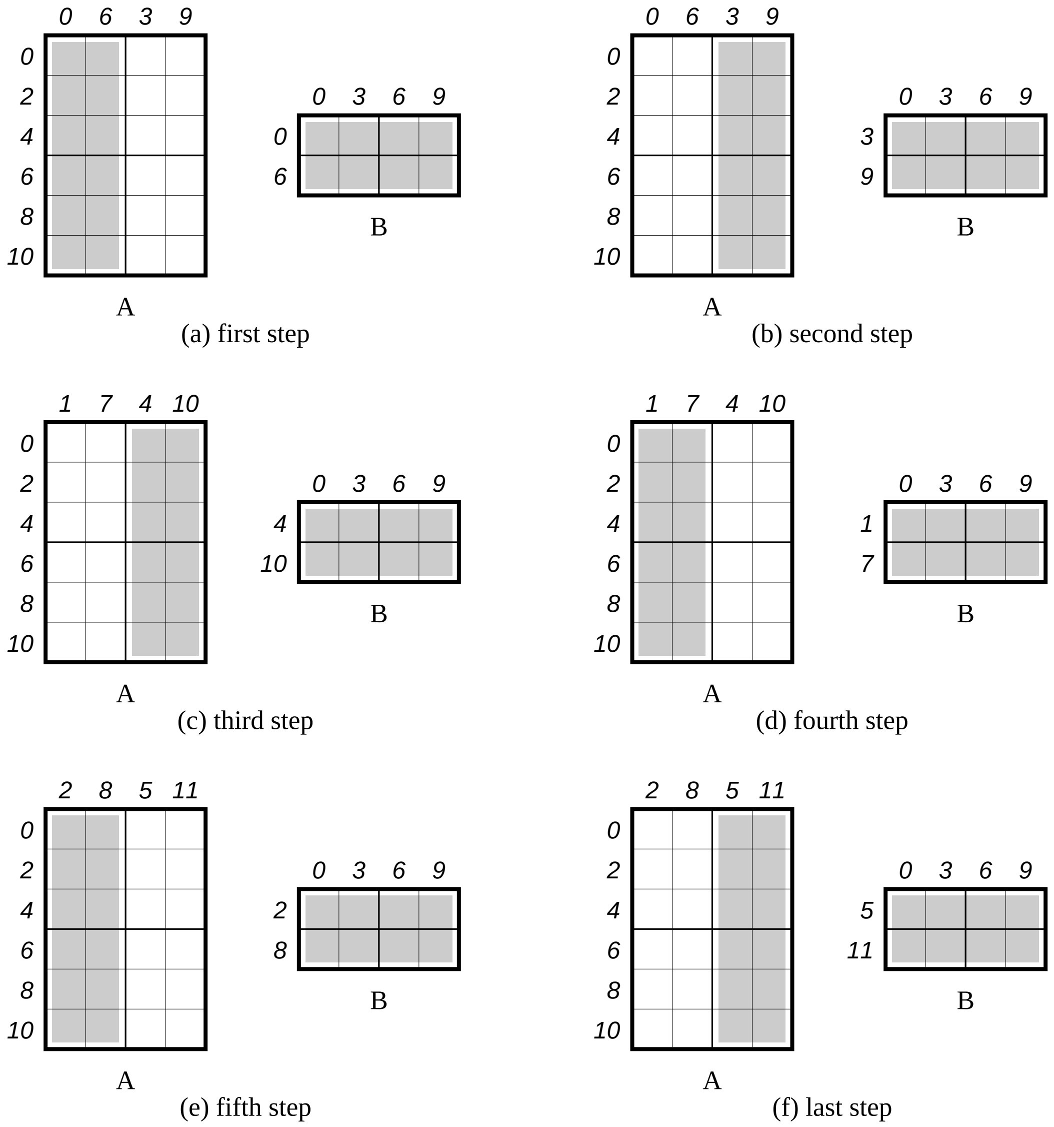
Figure 7: Snapshot of MDB2 algorithm. In each stage, four ($L_b/Q = 12/3$) wrapped diagonals are columncast. The total number of stages is Q.



(a) C = A · B in P_0 from processor point-of-view

(b) computation in P_0 with presorted A

Figure 8: An initial snap shot of $\mathbf{C} = \mathbf{A} \cdot \mathbf{B}$ in P_0 from the processor point-of-view, where $P = 2$, $Q = 3$ and $M_b = N_b = L_b = 12$. Columns of **A** are presorted in (b). The shaded area of **A** and **B** represents blocks to be multiplied, and that of **C** represents blocks to be updated by the multiplication.



(a) first step

(b) second step

(c) third step

(d) fourth step

(e) fifth step

(f) last step

Figure 9: Snapshots of $\mathbf{C} = \mathbf{A} \cdot \mathbf{B}$ in P_0. In the first step (a), P_0 computes $\mathbf{C}_0$, where $\mathbf{C}_0$ is local portion of $\mathbf{C}$ in P_0 ($C(0:11:2, 0:11:3)$). In the next step (b), P_0 receives $B([3,9], 0:11:3)$ from P_3 and updates $\mathbf{C}_0$. $\mathbf{A}$ is shifted rowwise and P_0 receives new presorted $\mathbf{A}$ ($A(0:11:2, [1,7,4,10])$) from P_1, and updates $\mathbf{C}_0$ with $B([4,10], 0:11:3)$ in the third step (c). For the next step (d), P_0 receives $B([1,7], 0:11:3)$ from P_3 and updates $\mathbf{C}_0$ again. At the end of sixth step (f), P_0 completes its computation.

```
DO $I = 0, M_b - 1$
    PARDO $J = 0, N_b - 1$
        $IP = \mathrm{MOD}(I + J, M_b)$
        PARDO $K = 0, L_b - 1$
            $\mathrm{T}(K) = [\mathrm{A}(K, IP)]^T \cdot \mathrm{B}(K, J)$
        END PARDO
        DO $K = 0, L_b - 1$
            $\mathrm{C}(IP, J) = \mathrm{C}(IP, J) + \mathrm{T}(K)$
        END DO
    END PARDO
    [Roll **A** leftwards]
END DO
```

Figure 10: The basic transposed matrix multiplication algorithm, $\mathbf{C} = \mathbf{A}^T \cdot \mathbf{B}$ for a block scattered decomposition. $[A(K, IP)]^T$ is the transpose of block $A(K, IP)$. This algorithm needs a sequential DO loop to compute $C(IP, J)$ by adding the temporary results $T(K)$ columnwise.

one large matrix multiplication of $A(0 : 11 : 2, [0, 6])$ and $B([0, 6], 0 : 11 : 3)$. All processors compute one matrix multiplication in each step instead of $\lceil L_b/LCM \rceil$ multiplications. The computation is like a *block* version of *matrix-matrix* multiplication. Figure 9 shows procedures of computing **C** in P_0 from the processor point-of-view. In the second, the fourth, and the sixth steps (Figure 9(b), (d), and (e), respectively), **B** is received from P_3. And presorted **A** is shifted rowwise after the second and the fourth steps.

The communication scheme of the MDB2 algorithm can be changed to rowwise broadcasting of $\lceil L_b/P \rceil$ blocks of **A** and columnwise shifting of presorted **B** without decreasing its performance. The two schemes have the same number of steps and the same amount of computation per processor in each step, but they have different communication strategies.

3.3 Transposed Matrix Multiplication Algorithm, $\mathbf{C} = \mathbf{A}^T \cdot \mathbf{B}$

We now describe the multiplication of transposed matrices, that is, multiplications of the form, $\mathbf{C} = \mathbf{A}^T \cdot \mathbf{B}$ and $\mathbf{C} = \mathbf{A} \cdot \mathbf{B}^T$. The multiplication algorithm of two transposed matrices, $\mathbf{C} = \mathbf{A}^T \cdot \mathbf{B}^T$, is presented in Section 3.4. Lin and Snyder [24] has given an algorithm computing $\mathbf{C} = \mathbf{A} \cdot \mathbf{B}$ based on a block distribution, that first transposes one of the matrices and then uses a series of block multiplication and reduction steps to evaluate **C**.

```
DO $I1 = 0, Q - 1$
    DO $I2 = 0, M_b/Q - 1$
        $I = I1 + I2 \times Q$
        PARDO $J = 0, N_b - 1$
            $IP = \mathrm{MOD}(I + \mathrm{MOD}(J, Q), M_b)$
            PARDO $K1 = 0, P - 1$
                $T(K1) = 0.0$
                DO $K2 = 0, L_b/P - 1$
                    $K = K1 + K2 \times P$
                    $\mathrm{T}(K1) = \mathrm{T}(K1) + [\mathrm{A}(K, IP)]^T \cdot \mathrm{B}(K, J)$
                END DO
            END PARDO
            DO $K1 = 0, P - 1$
                $\mathrm{C}(IP, J) = \mathrm{C}(IP, J) + \mathrm{T}(K1)$
            END DO
        END PARDO
    END DO
    [Roll $\mathbf{A}$ leftwards]
END DO
```

Figure 11: The transposed matrix multiplication algorithm, $\mathbf{C} = \mathbf{A}^T \cdot \mathbf{B}$. The outer loop has been split into loops over $I1$ and $I2$ so that the communication for several steps can be sent in a single message.

Consider first $\mathbf{C} = \mathbf{A}^T \cdot \mathbf{B}$, where $\mathbf{A}$ and $\mathbf{B}$ are $L_b \times M_b$ and $L_b \times N_b$ blocks, respectively, and they are distributed with a block scattered decomposition. $C(I, J)$ is then computed by

$$C(I, J) = \sum_{K=0}^{L_b-1} [A(K, I)]^T \cdot B(K, J) \tag{3}$$

where $I = 0, 1, \ldots, M_b - 1$, $J = 0, 1, \ldots, N_b - 1$ and $[A(K, I)]^T$ is the transposed block of $A(K, I)$. As in Equation 2, block indices are used, and the order of summation is arbitrary.

Figure 10 gives the pseudocode of the basic transposed matrix multiplication algorithm. The algorithm proceeds in L_b steps, in each of which blocks of $\mathbf{C}$ lying along a wrapped diagonal are evaluated. Each step consists of block matrix multiplication to form contributions to a wrapped diagonal block of $\mathbf{C}$, followed by summation over columns. Finally, a communication phase shifts $\mathbf{A}$ to the left by one block.

As in the MDB1 matrix multiplication algorithm of Section 3.2, the communication latency is reduced by simultaneously performing multiple instances of the outer I loop separated by LCM. Again the communication latency is reduced further when instances of the outer loop separated by Q are executed together as in the MDB2 algorithm. The blocks of $\mathbf{A}$ return to the same processor from which they started after they have been rolled Q times. So the algorithm proceeds in Q stages, in each of which $\lceil L_b/Q \rceil$ wrapped diagonal blocks of $\mathbf{C}$ are computed. The pseudocode of the modified algorithm is shown in Figure 11.

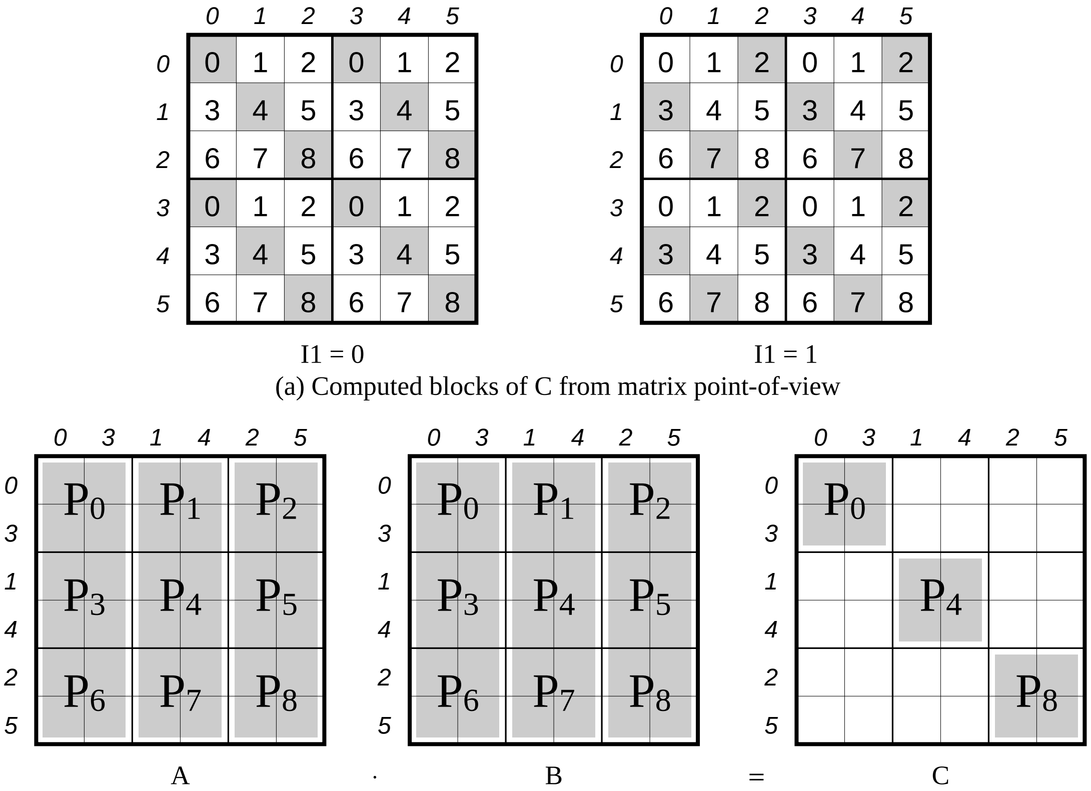
The transposed matrix multiplication algorithm is conceptually simpler than the non-transposed matrix multiplication algorithm. In $\mathbf{C} = \mathbf{A}^T \cdot \mathbf{B}$, processors in the same column of the template compute and add their products, and distribute the summations to the appropriate positions. The most difficult aspect when implementing the algorithm is how to add and distribute the products efficiently.

As an example, consider the matrix multiplication $\mathbf{C} = \mathbf{A}^T \cdot \mathbf{B}$ where matrices $\mathbf{A}$ and $\mathbf{B}$, each consisting of 6×6 blocks, are distributed over a 3×3 processor template as shown in Figure 12. In each stage, every Q-th wrapped block diagonal of $\mathbf{C}$ is computed. In the first stage, as shown in Figure 12 (b), the processors in the first column of the template, P_0, P_3, and P_6, multiply the zeroth and third block columns of $\mathbf{A}$ ($A(:, 0:5:3)$) with the zeroth and third block columns of $\mathbf{B}$ ($B(:, 0:5:3)$). They compute their own portion of multiplications and add them to obtain 2×2 blocks of $\mathbf{C}$ ($C(0:5:2, 0:5:2)$), which are placed in P_0. In this example, where the template is square, only the diagonal processors P_0, P_4, and P_8 have the computed blocks of $\mathbf{C}$ for each column of the template. After the first stage, $\mathbf{A}$ shifts to the left. The next wrapped diagonal processors P_2, P_3, and P_7 have the computed blocks of $\mathbf{C}$ in the second stage.

Figure 13 shows the case of $P = 3$, $Q = 2$, where $\mathbf{C}$ is computed in two stages. The first column of processors, P_0, P_2, and P_4, compute 3×3 blocks of $\mathbf{C}$ ($C(0:5:2, 0:5:2)$), by multiplying the zeroth, second and fourth block columns of $\mathbf{A}$ ($A(:, 0:5:2)$) with the zeroth, second and fourth block columns of $\mathbf{B}$ ($B(:, 0:5:2)$). After summing over columns they have computed their own row blocks of $\mathbf{C}$.

When Q is smaller than P, processors need more memory to store the partial products, if they compute their own products first and then add them together. Imagine the case when $P = 4, Q = 1$ and $M_b = N_b = L_b = 4$. Each processor has 1×4 blocks of $\mathbf{A}$ and $\mathbf{B}$, and it has 1×4 blocks of $\mathbf{C}$ after the computation. But processors need 4×4 blocks to store their own partial products. Thus, memory requirements do not scale well.

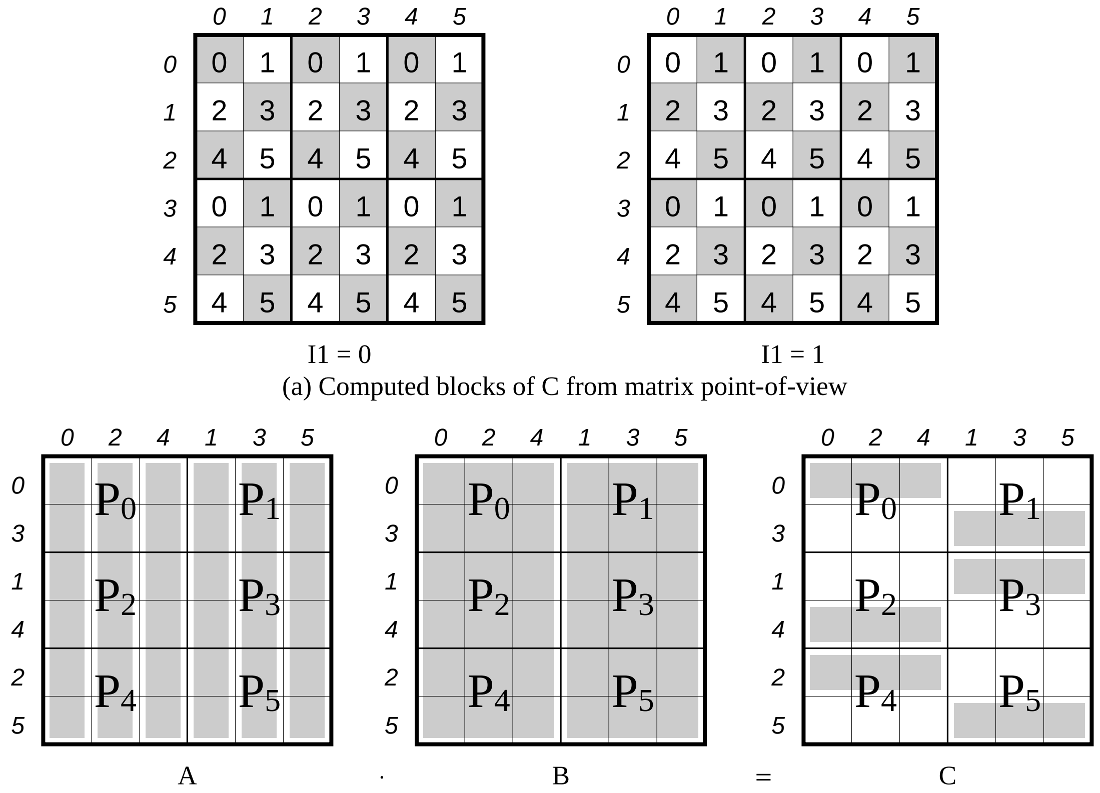
Processors can multiply one block column of $\mathbf{A}$ with whole blocks of $\mathbf{B}$ in each step to avoid nonscalable memory use. In the first step of Figure 13, P_0, P_2, and P_4 compute $C(0, 0:5:2)$ by multiplying $A(:, 0)$ with $B(:, 0:5:2)$. The computed blocks of $\mathbf{C}$ are placed in P_0. These processors then compute $C(2, 0:5:2)$,which is placed in P_4, and finally compute $C(4, 0:5:2)$, which is placed in P_2. After this stage $\mathbf{A}$ is shifted to the left. With this scheme, the processors require three steps to compute $C(0:5:2, 0:5:2)$ for the first stage of the algorithm. This



(a) Computed blocks of C from matrix point-of-view

(b) Snapshot of the first stage from processor point-of-view

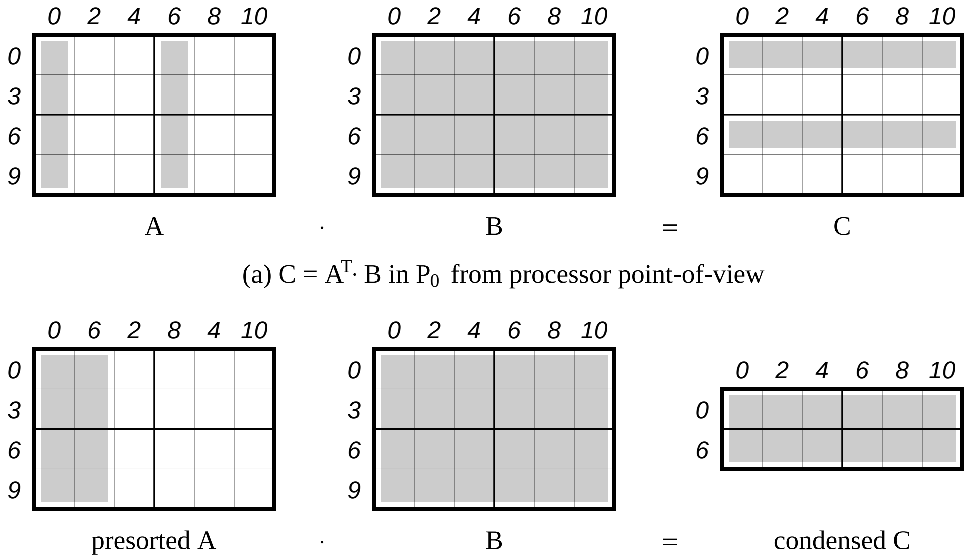
Figure 12: Snapshot of $\mathbf{C} = \mathbf{A}^T \cdot \mathbf{B}$ when $P = Q = 3$ and $M_b = N_b = L_b = 6$. (a) From the matrix point-of-view, the computed blocks of the matrix **C** in the first two stages of the transposed matrix multiplication algorithm are shaded. (b) Snapshot of the first stage from the processor point-of-view. The shaded area of **A** and **B** represents blocks to be multiplied, and that of **C** denotes blocks computed from the multiplication. Only diagonal processors have results in the first stage. After each stage, **A** is shifted to the left.



(a) Computed blocks of C from matrix point-of-view

(b) Snapshot of the first stage from processor point-of-view

Figure 13: Snapshot of $\mathbf{C} = \mathbf{A}^T \cdot \mathbf{B}$ when $P = 3$, $Q = 2$, and $M_b = N_b = L_b = 6$. (a) From matrix point-of-view, the computed blocks of the matrix $\mathbf{C}$ in the first two stages of the transposed matrix multiplication algorithm are shaded. (b) Snapshot of the first stage from processor point-of-view. If P and Q are relatively prime, the computed blocks of $\mathbf{C}$ are scattered over all processors in each stage.



(a) C = $A^{T}\cdot$ B in P_0 from processor point-of-view

(b) computation in P_0 with presorted A^T

Figure 14: $\mathbf{C} = \mathbf{A}^T \cdot \mathbf{B}$ in P_0 from processor point-of-view, where $P = 3$, $Q = 2$ and $M_b = N_b = L_b = 12$. The shaded area of **A** and **B** represents blocks to be multiplied. And that of **C** stands for the result blocks to be placed after multiplication and summation processes over the column of the template.

procedure is less efficient, but needs less memory to hold partial products.

The loss of efficiency can be offset by overlapping computation and communication. Consider a modified algorithm in which the blocks of **C** rotate downwards over the processor template after each stage. Each processor computes its own products and updates the received blocks. The processors receive their own desired blocks of **C** after $P - 1$ communications. If P and Q are relatively prime as shown in Figure 13, all processors have their own blocks of **C** in each stage. They receive partial products from the processor above, add their contributions to the partial products, and then send them to the processor below. If processors are waiting to receive the products before multiplying some processors have to wait a long time when $P = Q$ as in Figure 12 (or P and Q are not relatively prime). For these cases, processors compute their own multiplications first, and then add them after they receive the products. This can be implemented effectively with *asynchronous message passing* to minimize processors' waiting time to receive the products.

As an example, consider Figure 14 (a), where 12×12 block matrices are distributed over a 3×2 processor template. P_0 computes two ($\lceil M_b/LCM \rceil$) transposed matrix multiplications of block columns of **A** ($A(0:11:3,0)$ and $A(0:11:3,6)$) with its own submatrix **B** ($B(0:11:3,0:11:2)$), and generates two block rows of **C** ($C(0,0:11:2)$ and $C(6,0:11:2)$). The two rows of $\mathbf{C}$ are condensed for fast communications as in the MDB2 algorithm in Section 3.2. If block columns of $\mathbf{A}$ are presorted with radix LCM (or radix LCM/Q for each processor) at the beginning of the algorithm, processors compute *one* transposed matrix multiplication in each step instead of $\lceil L_b/LCM \rceil$ multiplications as shown in Figure 14 (b). Again, the computation is like a *block* version of *(transposed) matrix-matrix* multiplication.

The case $\mathbf{C} = \mathbf{A} \cdot \mathbf{B}^T$ is similar to the $\mathbf{C} = \mathbf{A}^T \cdot \mathbf{B}$ algorithm, but the partial result blocks of $\mathbf{C}$ rotate horizontally in each step, and $\mathbf{B}^T$ shifts upwards after each stage.

3.4 Multiplication of Transposed Matrices, $\mathbf{C} = \mathbf{A}^T \cdot \mathbf{B}^T$

Suppose we need to compute $\mathbf{C} = \mathbf{A}^T \cdot \mathbf{B}^T$, where $\mathbf{A}$ is $L_b \times M_b$ blocks, $\mathbf{B}$ is $N_b \times L_b$ blocks, and $\mathbf{C}$ is $M_b \times N_b$ blocks. One approach is to evaluate the product

$$C(I,J) = \sum_{K=0}^{L_b-1} [A(K,I)]^T \cdot [B(J,K)]^T, \tag{4}$$

directly using a variant of the matrix multiplication routine in Section 3.2, but in which blocks of $\mathbf{A}$ are columncast in each step, and blocks of $\mathbf{B}$ are rotated leftwards. The resultant matrix then has to be blockwise transposed, i.e., block $C(I,J)$ must be swapped with block $C(J,I)$, in order to obtain $\mathbf{C}$. Thus, for this approach the algorithm is as follows,

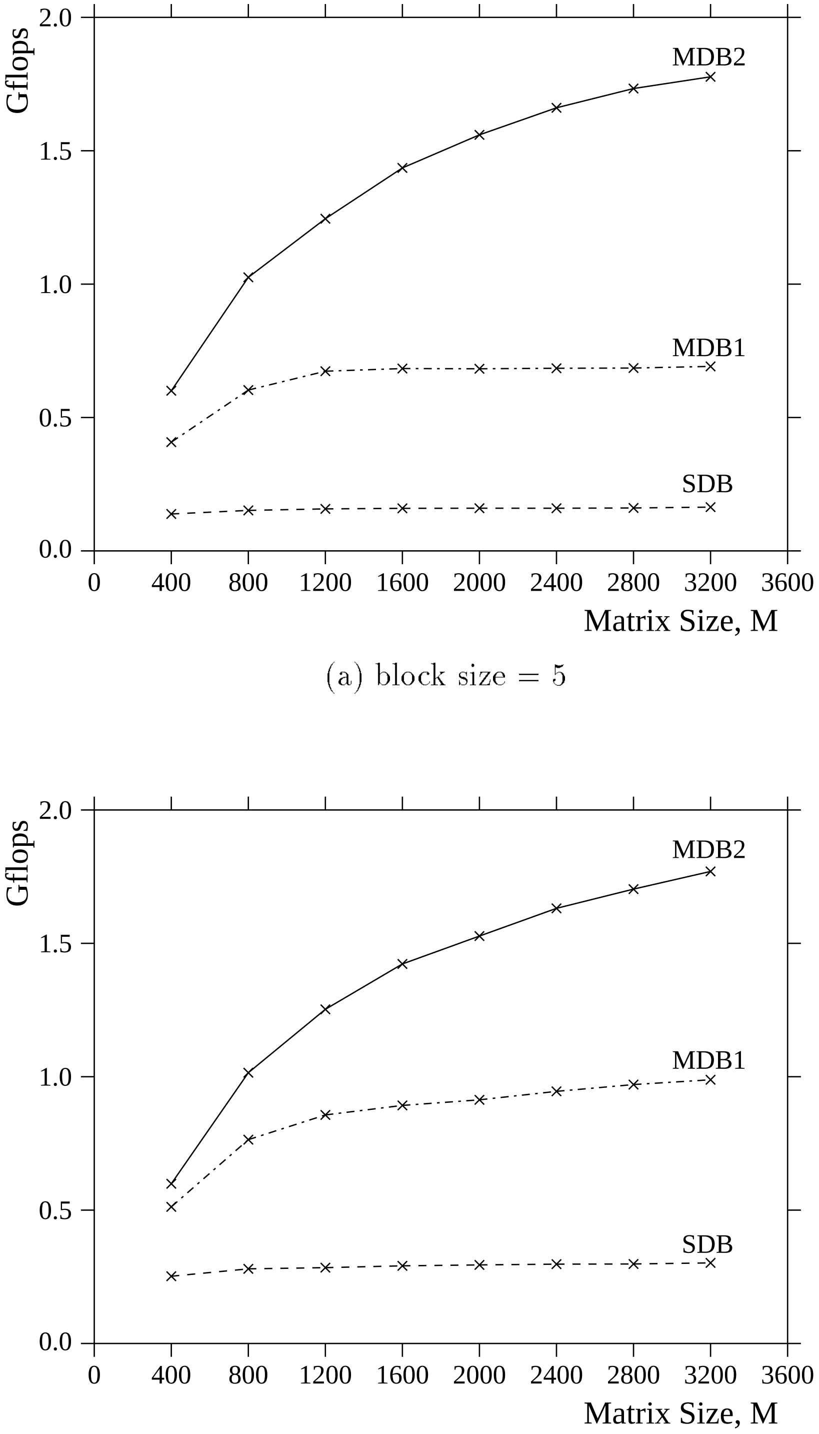
1. locally transpose each block of $\mathbf{A}$ and $\mathbf{B}$,
2. multiply $\mathbf{A}$ and $\mathbf{B}$ using variant of parallel algorithm,
3. do a blockwise transpose of the result to get $\mathbf{C}$.

In an actual implementation, the local transpose in (1) can be performed within the calls to the assembly-coded sequential xGEMM routine.

Another approach is to evaluate $\mathbf{C}^T = \mathbf{B} \cdot \mathbf{A}$ and then transpose the resulting matrix to obtain $\mathbf{C}$. In this case the algorithm is as follows,

1. multiply $\mathbf{B}$ and $\mathbf{A}$ using the parallel algorithm in Section 3.2,
2. locally transpose each block of result,
3. do a blockwise transpose to get $\mathbf{C}$.

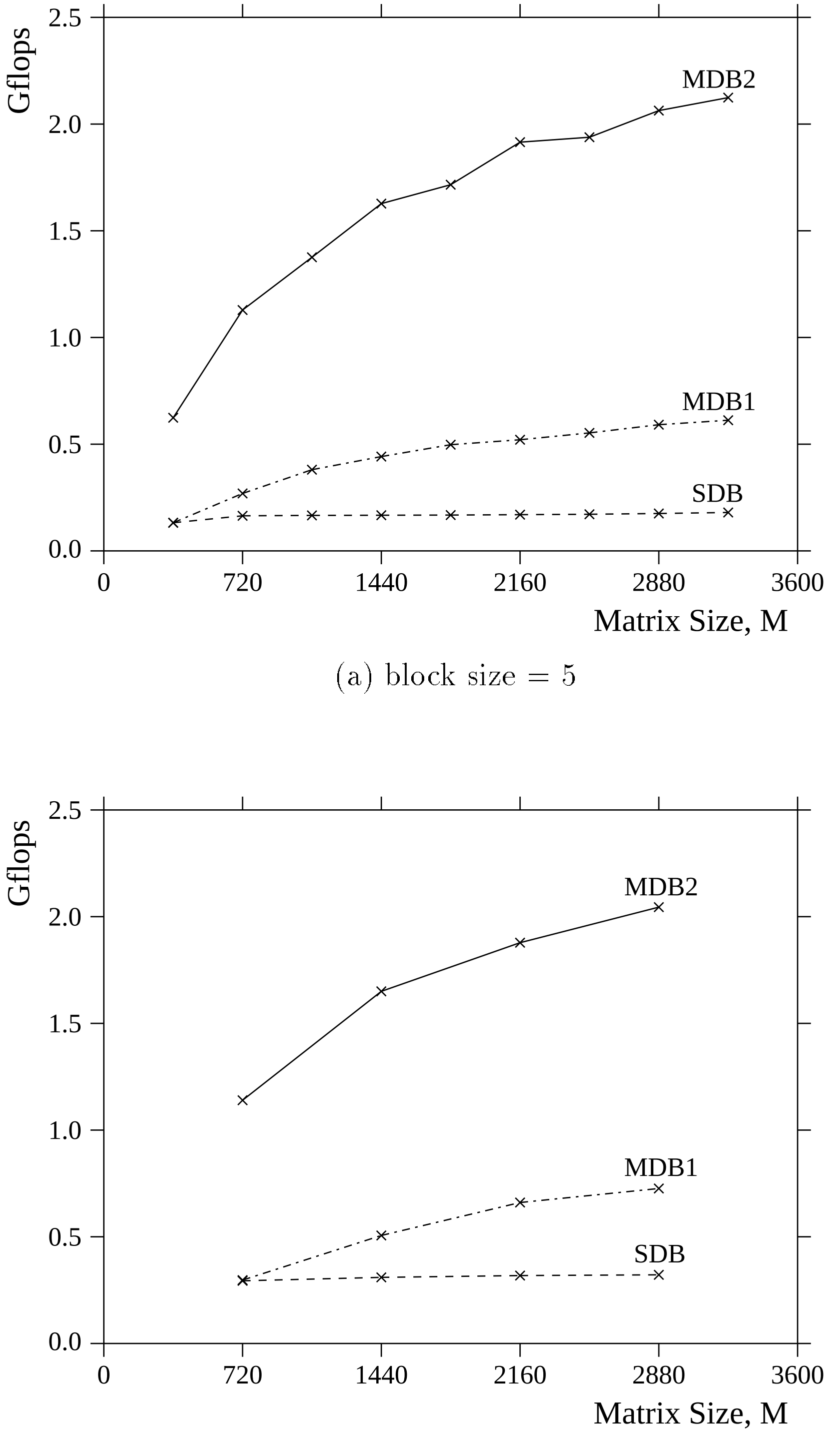
These last two steps together transpose $\mathbf{C}^T$, and may be done in any order. The performance of both approaches is very nearly the same, but the second approach has the advantage of using the existing algorithm for finding $\mathbf{B} \cdot \mathbf{A}$, as described in Section 3.2, without any modification being necessary. Parallel matrix transpose algorithms are described in [10], and are used to compute $\mathbf{C} = \alpha\, \mathbf{A}^T \cdot \mathbf{B}^T + \beta\, \mathbf{C}$ as described above in two steps: $\mathbf{T} = \alpha\, \mathbf{B} \cdot \mathbf{A}$, then $\mathbf{C} = \mathbf{T}^T + \beta\, \mathbf{C}$.



(a) block size = 5

(b) block size = 10

Figure 15: Performance comparison of the three matrix multiplication routines on an 8 × 8 processor template.



(a) block size = 5

(b) block size = 10

Figure 16: Performance comparison of the three matrix multiplication routines on a 9 × 8 processor template.

96 processors		64 processors		48 processors	
$P \times Q$	Gflops	$P \times Q$	Gflops	$P \times Q$	Gflops
6×16	1.972	4×16	1.373	4×12	1.101
8×12	2.007	8×8	1.447	6×8	1.181
12×8	2.008	16×4	1.444	8×6	1.200
16×6	2.002			12×4	1.130

Table 1: Dependence of performance on template configuration ($M = N = L = 1600$).

4 Results

In this section we present performance results for the PUMMA package on the Intel Touchstone Delta system. 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.

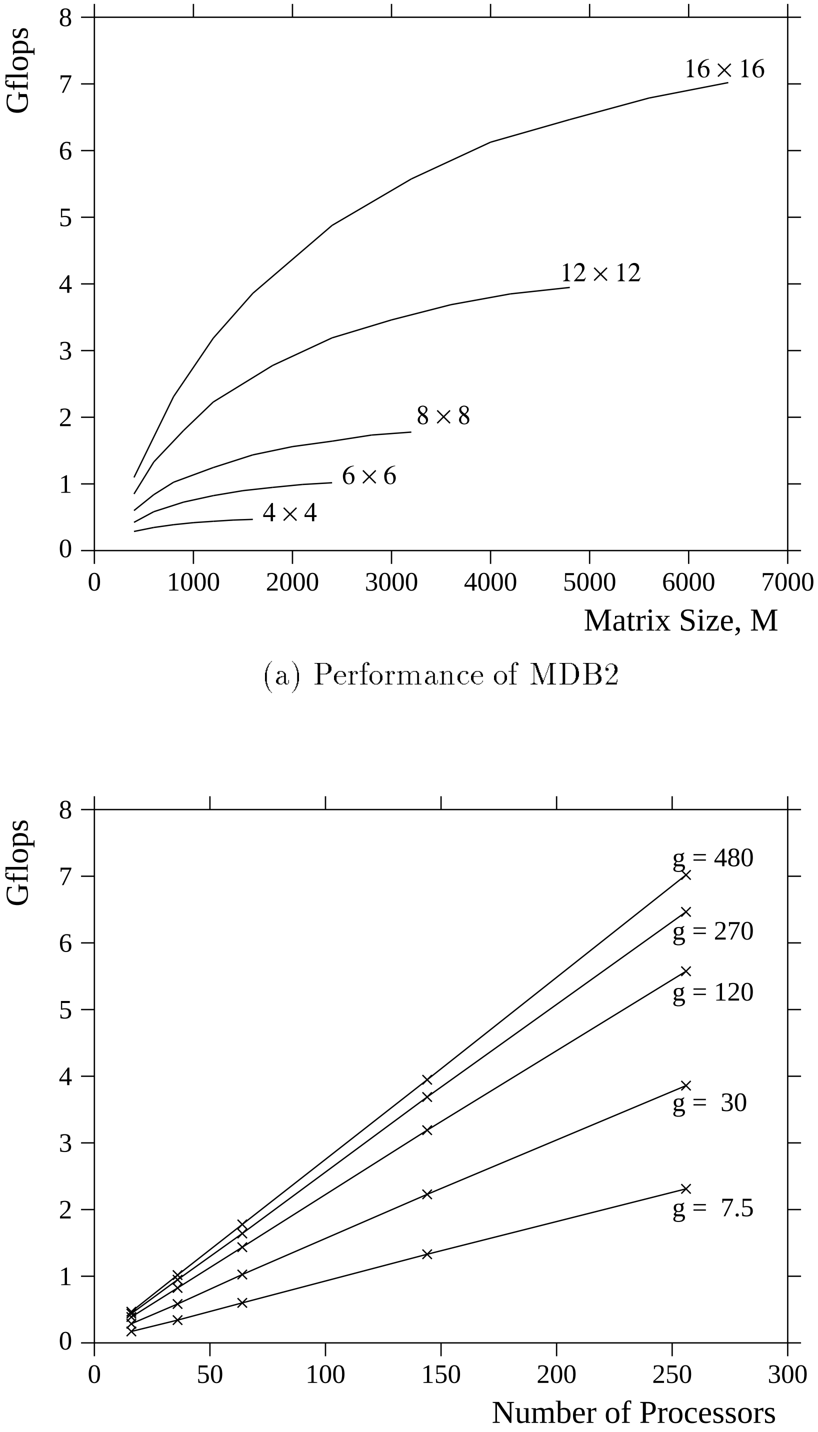
4.1 Comparison of Three Matrix Multiplication Algorithms

We first compared the three matrix multiplication algorithms, SDB, MDB1, and MDB2 on two fixed processor templates. Figures 15 and 16 show the performance of the algorithms on a square processor template (8×8, $P = Q$) and a nonsquare template (9×8, P and Q are relatively prime), respectively. Two different block sizes are considered to see how block size affects the performance of the algorithms for a number of different sized matrices.

The performance of the SDB and MDB1 algorithms improves as the block size is increased from 5 to 10, but this change of the block size has almost no effect on the performance of the MDB2 algorithm, since in MDB2 the size of the submatrices multiplied in each processor (using the assembly-coded Level 3 BLAS) is independent of block size. For a square template, the number of communication steps is the same in the MDB1 and MDB2 algorithms since $LCM = Q$, but there is a big difference in their performance. This difference arises because the basic operation of the MDB1 algorithm is a multiplication of a block column of $\mathbf{A}$ with a single block of $\mathbf{B}$, where as, in the MDB2 algorithm, larger matrices are multiplied in each step, as explained in Section 3.2.

The block size is selected by the user. In most cases, the optimal block size is determined by the size and shape of the processor template, floating-point performance of the processor, communication bandwidth between processors, and the size of the matrices. However, for the MDB2 algorithm, the performance is independent of the block size. We adopted a block size of 5×5 in all subsequent runs of the matrix multiplication routines.

We next considered how, for a fixed number of processors $N_p = P \times Q$, performance depended on



(a) Performance of MDB2

(b) Isogranularity Plot

Figure 17: Performance of MDB2 algorithm. (a) Performance in gigaflops as a function of matrix size for different numbers of processors. (b) Isogranularity curves in the (G, N_p) plane. The curves are labeled by the granularity g in units of 10^3 matrix elements per processor.

the configuration of the processor template. Some typical results are presented in Table 1 from which it may be seen that the template configuration does have a small effect on performance, with squarer templates giving better performance than long, thin templates. For a fixed number for processors, a larger value of Q increases the number of outer loops performed, but reduces the time to broadcast blocks of $\mathbf{B}$ across the template. The relative importance of these two factors determines the optimal template configuration. For rectangular templates with different aspect ratios, those with small Q show better performance than those with small P. For a fixed processor template with small P, an MDB2 algorithm, in which $\mathbf{A}$ is broadcast rowwise and $\mathbf{B}$ is shifted columnwise, is preferable to the version described in Section 3.2, in which $\mathbf{B}$ is broadcast columnwise and $\mathbf{A}$ is shifted rowwise.

Figure 17 (a) shows the performance of the MDB2 algorithm on the Intel Touchstone Delta as a function of problem size for different numbers of processors for up to 256 processors. In all cases a square processor template was used, i.e. $P = Q$, the block size was fixed at 5×5 elements, and the test matrices were of size up to 400×400 elements per processor.

In Figure 17 (b) we show how performance depends on the number of processors for a fixed grain size. The fact that these isogranularity plots are almost linear indicates that the distributed matrix multiplication routine scales well on the Delta, even for small granularity.

4.2 Comparison with Transposed Matrix Multiplication Algorithms

We compared the performance of the MDB2 version of the matrix multiplication routine $\mathbf{C} = \mathbf{A} \cdot \mathbf{B}$ with that of the transposed matrix multiplication routines, $\mathbf{C} = \mathbf{A}^T \cdot \mathbf{B}$, and $\mathbf{C} = \mathbf{A}^T \cdot \mathbf{B}^T$. For $\mathbf{C} = \mathbf{A} \cdot \mathbf{B}$, we adopted a routine with rowwise broadcasting of $\mathbf{A}$ and columnwise shifting of $\mathbf{B}$. $\mathbf{C} = \mathbf{A}^T \cdot \mathbf{B}$ is implemented as described in Section 3.3. For $\mathbf{C} = \mathbf{A}^T \cdot \mathbf{B}^T$, $\mathbf{B}$ is directly multiplied with $\mathbf{A}$ to form $\mathbf{B} \cdot \mathbf{A}$, which is then transposed to give $\mathbf{C}$.

Figures 18, 19, 20, and 21 show the performance of the algorithms on 8×8, 8×9, 8×10, and 8×12 templates, respectively. In all cases the block size is fixed at 5×5 elements. The solid and the dashed lines show the performance of $\mathbf{A} \cdot \mathbf{B}$ and $\mathbf{A}^T \cdot \mathbf{B}^T$, respectively. The difference of the two lines is due to the matrix transpose routine used in evaluating $\mathbf{A}^T \cdot \mathbf{B}^T$. In most cases, the performance of the $\mathbf{A}^T \cdot \mathbf{B}$ algorithm, which is drawn with the dot-dashed lines, lies between that of the $\mathbf{A} \cdot \mathbf{B}$ and $\mathbf{A}^T \cdot \mathbf{B}^T$ algorithms, but for the square template in Figure 18, its performance is worse than that of $\mathbf{A}^T \cdot \mathbf{B}^T$. In the $\mathbf{A}^T \cdot \mathbf{B}$ routine, processors in the same column of the template sequentially update their own $\mathbf{C}$. Some of the processors have to wait a long time to receive the partial products if $P = Q$.

Table 2 shows how the block size has an effect on 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. The algorithms depend only weakly on the block size. Even for the case of the smallest block size (1×1 element), the algorithms show good performance.

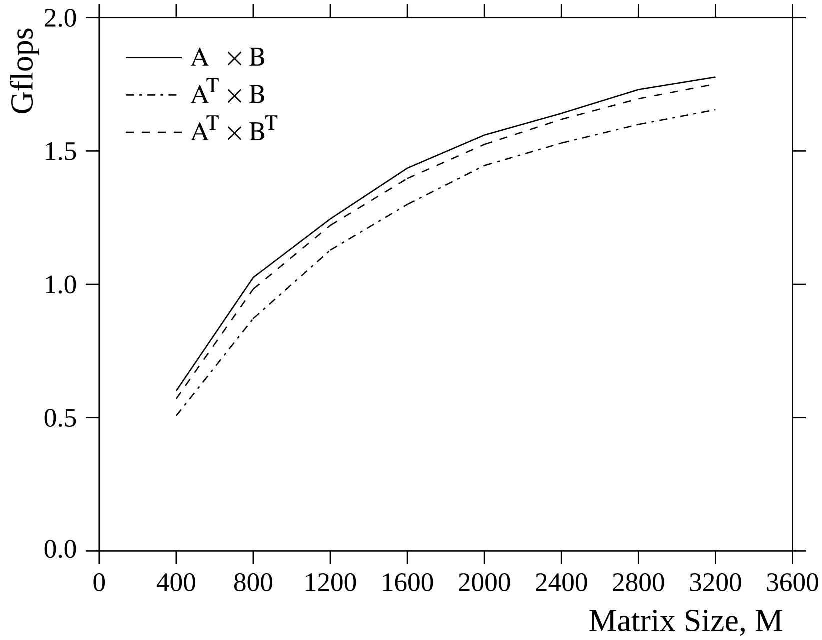


Figure 18: Performance comparison of three routines on an 8×8 template. $P = Q = LCM = 8$, and $GCD = 8$

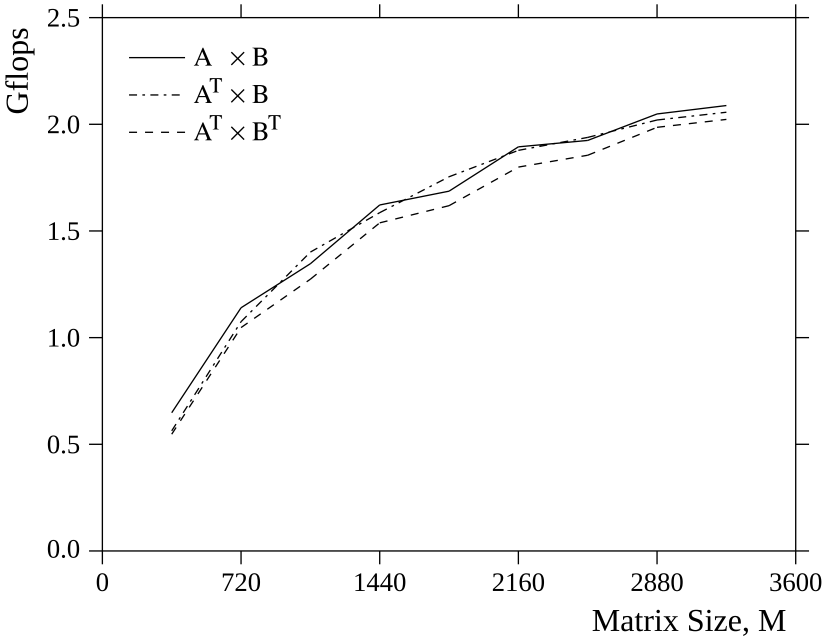


Figure 19: Performance comparison of three routines on an 8×9 template. $P = 8, Q = 9, LCM = 72$, and $GCD = 1$

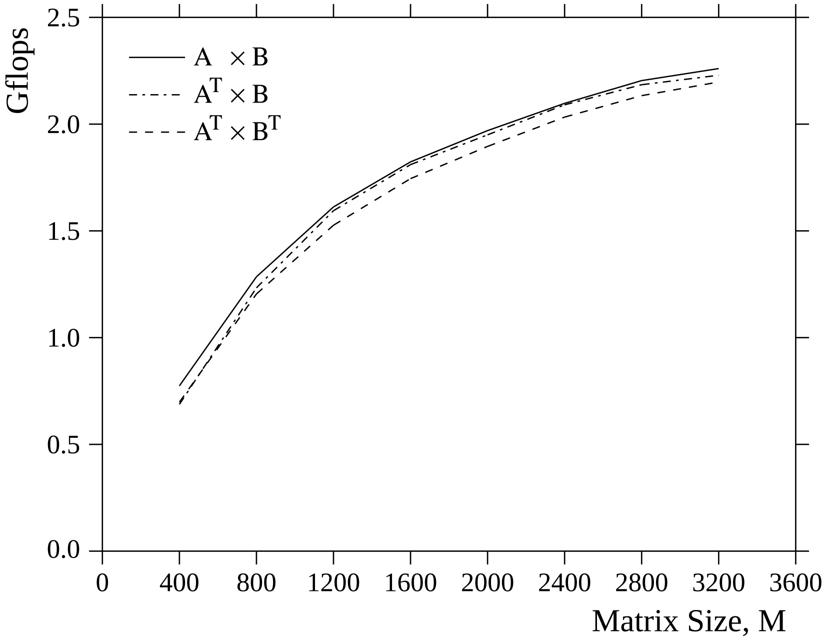


Figure 20: Performance comparison of three routines on an 8 × 10 template. $P = 8, Q = 10, LCM = 40, \text{and } GCD = 2$

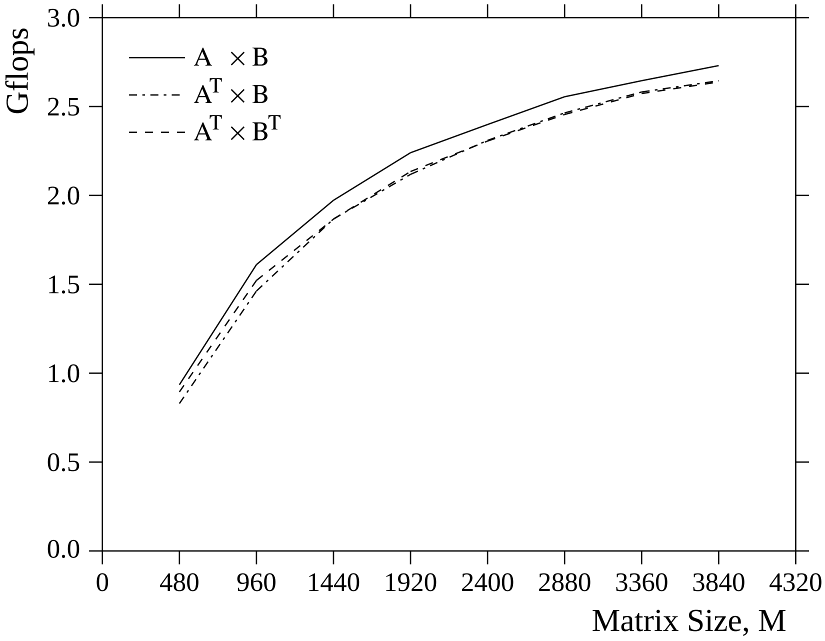


Figure 21: Performance comparison of three routines on an 8 × 12 template. $P = 8, Q = 12, LCM = 24, \text{and } GCD = 4$

$P \times Q$	Matrix Size	Block Size	$\mathbf{A} \cdot \mathbf{B}$	$\mathbf{A}^T \cdot \mathbf{B}$	$\mathbf{A}^T \cdot \mathbf{B}^T$
8×8	2400×2400	1×1	1.640	1.529	1.607
		5×5	1.641	1.530	1.619
		300×300	1.643	1.531	1.618
8×9	2520×2520	1×1	1.902	1.904	1.732
		5×5	1.924	1.939	1.850
		35×35	1.926	1.946	1.860
8×10	2400×2400	1×1	2.085	2.067	1.961
		5×5	2.107	2.110	2.033
		60×60	2.096	2.123	2.028
8×12	2400×2400	1×1	2.374	2.121	2.265
		5×5	2.389	2.310	2.306
		100×100	2.397	2.338	2.317

Table 2: Dependence of performance on block size (Unit: Gflops)

$P \times Q$	Matrix Size	$\mathbf{A} \cdot \mathbf{B}$	$\mathbf{A}^T \cdot \mathbf{B}$	$\mathbf{A}^T \cdot \mathbf{B}^T$
1×1	400×400	36.21 (100.0)	35.54 (100.0)	34.58 (100.0)
8×8	3200×3200	27.77 (76.7)	25.86 (72.8)	27.36 (79.1)
8×9	3240×3240	29.00 (80.1)	28.56 (80.4)	28.10 (81.3)
8×10	3200×3200	28.25 (78.0)	27.74 (78.1)	27.47 (79.4)
8×12	3200×3200	28.44 (78.5)	27.55 (77.5)	27.48 (79.5)

Table 3: Performance per node in Mflops. Block size is fixed at 5×5 elements. Entries for the 1×1 template case give the performance of the assembly-coded Level 3 BLAS matrix multiplication routine. Numbers in parenthesis are concurrent efficiency.

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. The numbers in parentheses are concurrent efficiency, which is the relative performance of nodes compared with the maximum performance of the assembly-coded Level 3 BLAS routine. Approximately 77% efficiency is achieved for $\mathbf{A} \cdot \mathbf{B}$, 73% for $\mathbf{A}^T \cdot \mathbf{B}$, and 79% for $\mathbf{A}^T \cdot \mathbf{B}^T$ if $P = Q$. The routines perform better on templates for which $P \neq Q$. More than 80% efficiency is achieved for all cases if P and Q are relatively prime.

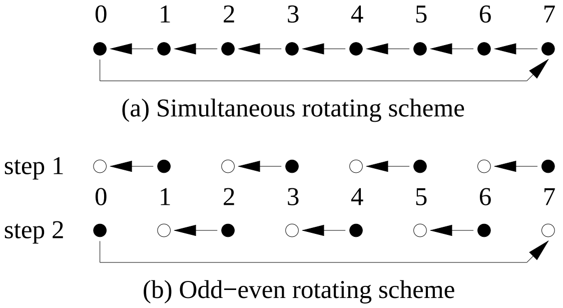

0
1
2
3
4
5
6
7
(a) Simultaneous rotating scheme
step 1
step 2
(b) Odd−even rotating scheme

Figure 22: Two rotating schemes. (a) Nodes first send to the left and then receive from the right. (b) In the first step, odd-numbered processors send data blocks and even-numbered processors receive them. In the next step, even-numbered processors send and odd-numbered processors receive. Odd-even rotating is faster on Delta, but simultaneous rotating is faster on iPSC/860 hypercube.

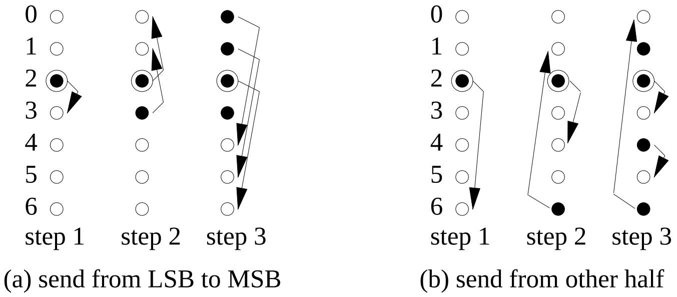
4.3 Results with Optimized Communication Routines for the Intel Delta

For the implementation of the PUMMA package, blocking and nonblocking communication schemes were used. In this section, we modify the algorithms with optimized communication schemes specifically for the Intel Touchstone Delta.

First, *force type* communications [22] are incorporated for faster communications. A force type message bypasses the normal flow control mechanism, and is not delayed by clogged message buffers on a processor. A force type message is discarded if no receive has been posted on the destination processor prior to its arrival. If force types are not used on the Delta, the routines can accommodate matrices up to 400×400 elements per processor without encounting problems arising from system buffer overflow [23]. With force type communication, the routines can handle larger matrices, up to 500×500 per processor, where the maximum size is determined by the available memory per processor rather than by system buffer constraints.

A block rotating scheme is used to shift $\mathbf{A}$ rowwise in the MDB2 algorithm of Section 3.2 and in the $\mathbf{A}^T \cdot \mathbf{B}$ routine of Section 3.3. A simultaneous rotating scheme, shown in Figure 22 (a), may be used on the Intel iPSC/860 hypercube. However, an odd-even rotating scheme is preferable on the Delta [25]. This scheme performs the communication in two steps as shown in Figure 22 (b). In the first step, odd-numbered processors send their own data blocks and even-numbered processors receive them. In the next step, even-numbered processors send and odd-numbered processors receive.

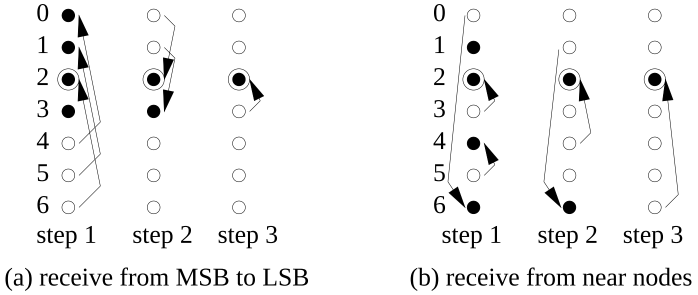
In the original MDB2 algorithm, blocks of $\mathbf{B}$ are broadcast in each column of the template based on a ring communication scheme. In the Delta-specific MDB2 algorithm, messages are broadcast based on a minimum spanning tree. A special broadcasting routine is desirable for the Delta,



(a) send from LSB to MSB

(b) send from other half

Figure 23: Broadcasting on linear array of $p = 7$, where nodes are numbered 0 through 6. P_2 is a root node. (a) is the hypercube algorithm and (b) is the mesh algorithm.



(a) receive from MSB to LSB

(b) receive from near nodes

Figure 24: Collecting on linear array. P_2 is a root node. (a) is the hypercube algorithm and (b) is the mesh algorithm.

which differs from that used on hypercubes [5]. Consider broadcasting a message on a linear array of $p = 7$ processors as shown in Figure 23,where nodes are numbered 0 through 6. In the hypercube scheme, the root node P_2, which has the message to be broadcast, first sends the message to P_3, whose least significant bit (LSB) is different from the root node. Then the message is delivered by toggling successive bits from LSB to the most significant bit (MSB). On a mesh topology such as the Delta, the network traffic becomes congested as the broadcast proceeds, as shown in Figure 23 (a).

In order to avoid network contention, the root node sends the message to the first node in the other half of the processors. By recursing for $\lceil \log_2 P \rceil$ similar steps, the message is delivered to all nodes without any contention as shown in Figure 23 (b). In general, each column of the template has P/GCD root nodes in a stage, which broadcast their blocks of $\mathbf{B}$ over GCD processors of the column, where GCD denotes the greatest common divisor of P and Q. These operations are a form of *group communication* [15].

For $\mathbf{A}^T \cdot \mathbf{B}$ in Section 3.3, the partial products in the same column of the processors are combined and the sum is stored in the root (destination) node. A special collecting scheme has been developed for the Delta to avoid network contention. The new collecting scheme on a linear array shown in Figure 24 (b) is based on the broadcasting scheme in Figure 23 (b). The partial

		$\mathbf{A} \cdot \mathbf{B}$		$\mathbf{A}^T \cdot \mathbf{B}$	
$P \times Q$	Matrix Size	RING	**TREE**	**RING**	TREE
4×4	2000×2000	0.515	0.530	0.488	0.496
6×6	3000×3000	1.130	1.159	1.081	1.073
8×8	4000×4000	1.985	2.056	1.908	1.901
12×12	6000×6000	4.438	4.507	4.260	4.150
16×16	8000×8000	7.844	8.001	7.542	7.325
8×9	3960×3960	2.326	2.326	2.321	2.321
8×10	4400×4400	2.561	2.641	2.493	2.486
8×12	4800×4800	2.965	3.091	2.956	2.918
8×16	5600×5600	3.858	4.022	3.707	3.622

Table 4: Performance in Gflops with optimized communication routines on two structures, ring and spanning tree. Block size is fixed to 5×5. The routine for $\mathbf{A} \cdot \mathbf{B}$ is faster for a tree structure, but the routine for $\mathbf{A}^T \cdot \mathbf{B}$ has better performance for a ring structure.

products are sent and added in nodes which are nearer to the root node. Generally, in each stage of the algorithm, each column of the template has P/GCD root nodes to collect the partial products. Partial products of a group of GCD processors are added first with the products of other group(s) in the same column. After $P/GCD-1$ communications and additions, the partial products in each group of GCD processors are effectively added to the root nodes.

The $\mathbf{A} \cdot \mathbf{B}$ and $\mathbf{A}^T \cdot \mathbf{B}$ routines have been implemented with the optimized communications for the Delta based on both the ring and the minimum spanning tree structure for broadcasts. Performance results are shown in Table 4. The non-transposed matrix multiplication routine for 8000×8000 matrices on 16×16 nodes performs at about 8.00 Gflops for the tree structure, and the transposed multiplication routine executes at about 7.54 Gflops for the ring structure. They obtain about 31.25 Mflops and 29.46 Mflops per processor, respectively, which correspond to concurrent efficiencies of 86% and 83%, respectively.

If P and Q are relatively prime, there is no performance difference between tree and ring versions. The $\mathbf{A} \cdot \mathbf{B}$ algorithm performs well for the tree structure. Though broadcasting a message to the entire column of the processors on the ring is slow, the overall performance is not influenced since the stages of the algorithm are pipelined. That is, processors directly proceed to the next stage as soon as they finish their multiplication at the current stage.

In a single stage of the $\mathbf{A}^T \cdot \mathbf{B}$ routine, collecting the partial products in a column of the processor template is faster for the tree algorithm. However, overall the ring algorithm is preferred for the $\mathbf{A}^T \cdot \mathbf{B}$ routine, since stages of the algorithm can be pipelined.

5 Conclusions and Remarks

We have presented a parallel matrix multiplication routine and its variants for the block scattered decomposition over a two dimensional processor template. We have described how to develop the algorithms for distributed memory concurrent computers from a matrix point-of-view, and given implementation details from a processor point-of-view. Finally we have shown how to adapt the communications for a specific target machine, the Intel Touchstone Delta computer, by exploiting its communication characteristics. The general purpose matrix multiplication routines developed are universal algorithms that can be used for arbitrary processor configuration and block size.

In general, the first dimension of the data matrix may be different from the number of rows of the matrix in a processor. If they are different, the data portion of $\mathbf{A}$ in a local processor should to be copied before it is sent out in the initial shifting of $\mathbf{A}$ in the MDB2 routine. Instead of a direct copy, the block columns of $\mathbf{A}$ are presorted so that each processor performs a block version of matrix-matrix multiplication in each step, as described in Section 3.3. Without this presorting, processors compute multiplications as a block version of the outer product operation, i.e., a column of blocks is multiplied by a row of blocks. The outer product operation performs well and its performance is almost the same as the routine with presorting for blocks larger than 5×5 elements. But for the case of small block sizes, presorting improves performance. If the first dimension of matrix $\mathbf{A}$ is the same as the number of rows, the presorting is not necessary, and $\mathbf{A}$ can be sent out directly, since after Q shifts of $\mathbf{A}$ processors have their original blocks, and $\mathbf{A}$ is unchanged. This scheme may also save communication buffer space. For the transposed matrix multiplication routines ($\mathbf{A}^T \cdot \mathbf{B}$ and $\mathbf{A} \cdot \mathbf{B}^T$), the presorting process improves the performance more than 10% for a block size of 5×5.

In some cases, the transposed matrix multiplication algorithm may be slower than the two combined routines, matrix transposition and matrix multiplication. That is, $\mathbf{C} \Leftarrow \mathbf{A}^T \cdot \mathbf{B}$ can be implemented with two steps, ($\mathbf{T} \Leftarrow \mathbf{A}^T$, $\mathbf{C} \Leftarrow \mathbf{T} \cdot \mathbf{B}$), where extra memory space for $\mathbf{T}$ is necessary. Users can choose the best routine according to their machine specifications and their application. The performance of the routines not only depends on the machine characteristics, but also the processor configuration and the problem size.

The performance of the PUMMA package can be improved with optimized assembly-coded routines, if available, such as a two dimensional buffer copy routine ($\mathbf{T} \Leftarrow \mathrm{op}(\mathbf{A})$, where $\mathrm{op}(\mathbf{A}) = \mathbf{A}$ or $\mathbf{A}^T$), and a two dimensional addition routine ($\mathbf{T} \Leftarrow \alpha\, \mathbf{A} + \beta\, \mathbf{T}$).

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 real and complex. To obtain a copy of the software and a description of how to use it, send the following message "`send pumma from scalapack`" to `netlib@ornl.gov`.

Acknowledgments

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.

References

[1] P. R. Amestoy, M. J. Dayde, I. S. Duff, and P. Morere. Linear algebra calculations on the BBN TC2000. In G. Goos and J. Hartmanis, editors, *Proceedings of Second Joint International Conference on Vector and Parallel Processing*, pages 319–330. Springer-Verlag, 1992.

[2] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. LAPACK: A portable linear algebra library for high-performance computers. In *Proceedings of Supercomputing '90*, pages 1–10. IEEE Press, 1990.

[3] E. Anderson, Z. Bai, J. Demmel, J. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide*. SIAM Press, Philadelphia, PA, 1992.

[4] E. Anderson, A. Benzoni, J. Dongarra, S. Moulton, S. Ostrouchov, B. Tourancheau, and R. van de Geijn. Basic Linear Algebra Communication Subprograms. In *Sixth Distributed Memory Computing Conference Proceedings*, pages 287–290. IEEE Computer Society Press, 1991.

[5] M. Barnett, D. G. Payne, and R. van de Geijn. Optimal minimum spanning tree broadcasting in mesh-connected architecture. Technical Report TM-91-38, The University of Texas at Austin, December 1991.

[6] P. Berger, M. J. Dayde, and P. Morere. Implementation and use of Level 3 BLAS kernels on a transputer T800 ring network. Technical Report TR/PA/91/54, CERFACS, June 1991.

[7] 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.

[8] 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.

[9] J. Choi, J. J. Dongarra, and D. W. Walker. Level 3 BLAS 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.

[10] J. Choi, J. J. Dongarra, and D. W. Walker. Parallel Matrix Transpose Algorithms on Distributed Memory Concurrent Computers. Technical Report TM-12309, Oak Ridge National Laboratory, Mathematical Sciences Section, October 1993.

[11] J. Demmel, J. J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, and D. Sorensen. Prospectus for the development of a linear algebra library for high performance computers. Technical Report 97, Argonne National Laboratory, Mathematics and Computer Science Division, September 1987.

[12] J. J. Dongarra. Workshop on the BLACS. LAPACK Working Note 34, Technical Report CS-91-134, University of Tennessee, 1991.

[13] J. J. Dongarra, I. Duff, J. Du Croz, and S. Hammarling. A set of level 3 basic linear algebra subprograms. *ACM TOMS*, 16:1–17, March 1990.

[14] J. J. Dongarra, I. S. Duff, D. C. Sorensen, and H. A. van der Vorst. *Solving Linear Systems on Vector and Shared Memory Computers*. SIAM, Philadelphia, PA, 1990.

[15] J. J. Dongarra, R. Hempel, A. J. G. Hey, and D. W. Walker. A proposal for a user-level, message passing interface in a distributed memory environment. Technical Report TM-12231, Oak Ridge National Laboratory, March 1993.

[16] 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.

[17] J. J. Dongarra and R. A. van de Geijn. Two dimensional basic linear algebra communication subprograms. LAPACK Working Note 37, Technical Report CS-91-138, University of Tennessee, 1991.

[18] A. C. Elster. Basic matrix subprograms for distributed memory systems. In D. W. Walker and Q. F. Stout, editors, *Proceedings of the Fifth Distributed Memory Computing Conference*, pages 311–316. IEEE Press, 1990.

[19] R. D. Falgout, A. Skjellum, S. G. Smith, and C. H. Still. The multicomputer toolbox approach to concurrent BLAS and LACS. In *Proceedings of the 1992 Scalable High Performance Computing Conference*, pages 121–128. IEEE Press, 1992.

[20] 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.

[21] S. Huss-Lederman, E. M. Jacobson, A. Tsao, and G. Zhang. Matrix multiplication on the Intel Touchstone Delta. Technical report, Supercomputing Research Center, 1993. in preparation.

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

[23] Intel Corporation. *Touchstone Delta System User's Guide*, October 1991.

[24] C. Lin and L. Snyder. A matrix product algorithm and its comparative performance on hypercubes. In *Proceedings of the 1992 Scalable High Performance Computing Conference*, pages 190–194. IEEE Press, 1992.

[25] 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.