A New Parallel Matrix Multiplication Algorithm on Distributed-Memory Concurrent Computers

Jaeyoung Choi
School of Computing
Soongsil University
1-1, Sangdo-Dong, Dongjak-Ku
Seoul 156-743, KOREA

Abstract

We present a new fast and scalable matrix multiplication algorithm, called DIMMA (Distribution-Independent Matrix Multiplication Algorithm), for block cyclic data distribution on distributed-memory concurrent computers. The algorithm is based on two new ideas; it uses a modified pipelined communication scheme to overlap computation and communication effectively, and exploits the LCM block concept to obtain the maximum performance of the sequential BLAS routine in each processor even when the block size is very small as well as very large. The algorithm is implemented and compared with SUMMA on the Intel Paragon computer.

1. Introduction

A number of algorithms are currently available for multiplying two matrices **A** and **B** to yield the product matrix $\mathbf{C} = \mathbf{A} \times \mathbf{B}$ on distributed-memory concurrent computers [12, 16]. Two classic algorithms are Cannon's algorithm [4] and Fox's algorithm [11]. They are based on a $P \times P$ square processor grid with a block data distribution in which each processor holds a large consecutive block of data.

Two efforts to implement Fox's algorithm on general 2-D grids have been made: Choi, Dongarra and Walker developed 'PUMMA' [7] for block cyclic data decompositions, and Huss-Lederman, Jacobson, Tsao and Zhang developed 'BiMMeR' [15] for the virtual 2-D torus wrap data layout. The differences in these data layouts results in different algorithms. These two algorithms have been compared on the Intel Touchstone Delta [14].

Recent efforts to implement numerical algorithms for dense matrices on distributed-memory concurrent computers are based on a block cyclic data distribution [6], in which an $M \times N$ matrix **A** consists of $m_b \times n_b$ blocks of data, and the blocks are distributed by wrapping around both row and column directions on an arbitrary $P \times Q$ processor grid. The distribution can reproduce most data distributions used in linear algebra computations. For details, see Section 2.2. We limit the distribution of data matrices to the block cyclic data distribution.

The PUMMA requires a minimum number of communications and computations. It consists of only $Q - 1$ shifts for **A**, $LCM(P, Q)$ broadcasts for **B**, and $LCM(P, Q)$ local multiplications, where $LCM(P, Q)$ is the least common multiple of P and Q. It multiplies the largest possible matrices of **A** and **B** for each computation step, so that performance of the routine depends very weakly on the block size of the matrix. However, PUMMA makes it difficult to overlap computation with communication since it always deals with the largest possible matrices for both computation and communication, and it requires large memory space to store them temporarily, which makes it impractical in real applications.

Agrawal, Gustavson and Zubair [1] proposed another matrix multiplication algorithm by efficiently overlapping computation with communication on the Intel iPSC/860 and Delta system. Van de Geijn and Watts [18] independently developed the same algorithm on the Intel paragon and called it SUMMA. Also independently, PBLAS [5], which is a major building block of ScaLAPACK [3], uses the same scheme in implementing the matrix multiplication routine, `PDGEMM`.

In this paper, we present a new fast and scalable matrix multiplication algorithm, called DIMMA (Distribution-Independent Matrix Multiplication Algorithm) for block cyclic data distribution on distributed-memory concurrent computers. The algorithm incorporates SUMMA with two new ideas. It uses 'a modified pipelined communication scheme', which makes the algorithm the most efficient by overlapping computation and communication effectively. It also exploits 'the LCM concept', which maintains the maximum performance of the sequential BLAS routine, `DGEMM`, in each processor, even when the block size is very small as well as very large. The details of the LCM concept is explained in Section 2.2.

DIMMA and SUMMA are implemented and compared on the Intel Paragon computer.

The parallel matrix multiplication requires $O(N^3)$ flops and $O(N^2)$ communications, i. e., it is computation intensive. For a large matrix, the performance difference between SUMMA and DIMMA may be marginal and negligible. But for small matrix of $N = 1000$ on a 16×16 processor grid, the performance difference is approximately 10%.

2. Design Principles

2.1. Level 3 BLAS

Current advanced architecture computers possess hierarchical memories in which access to data in the upper levels of the memory hierarchy (registers, cache, and/or local memory) is faster than to data in lower levels (shared or off-processor memory). One technique to exploit the power of such machines more efficiently is to develop algorithms that maximize reuse of data held in the upper levels. This can be done by partitioning the matrix or matrices into blocks and by performing the computation with matrix-matrix operations on the blocks. The Level 3 BLAS [9] 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.

The Level 3 BLAS have been successfully used as the building blocks of a number of applications, including LAPACK [2], a software library that uses block-partitioned algorithms for performing dense linear algebra computations on vector and shared memory computers.

On shared memory machines, block-partitioned algorithms reduce the number of times that data must be fetched from shared memory, while on distributed-memory machines, they reduce the number of messages required to get the data from other processors. Thus, there has been much interest in developing versions of the Level 3 BLAS for distributed-memory concurrent computers [5, 8, 10].

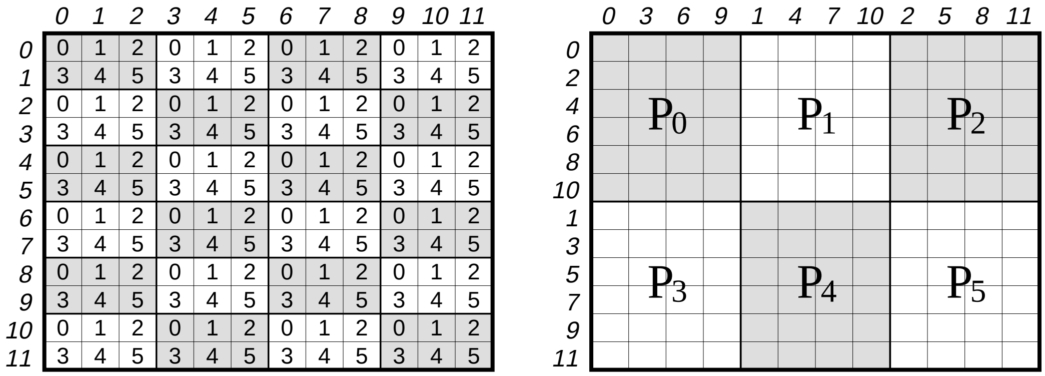
The most important routine in the Level 3 BLAS is `DGEMM` for performing matrix-matrix multiplication. The general purpose routine performs the following operation:

$$\mathbf{C} \Leftarrow \alpha \; op(\mathbf{A}) \;\cdot\; op(\mathbf{B}) \;+ \beta\, \mathbf{C}$$

where $op(\mathbf{X}) = \mathbf{X}, \mathbf{X}^T$ or $\mathbf{X}^H$. And "$\cdot$" denotes matrix-matrix multiplication. $\mathbf{A}$, $\mathbf{B}$ and $\mathbf{C}$ are matrices, and α and β are scalars. This paper focuses on the design and implementation of the non-transposed matrix multiplication routine of $\mathbf{C} \Leftarrow \alpha\, \mathbf{A} \cdot \mathbf{B} + \beta\, \mathbf{C}$, but the idea can be easily extended to the transposed multiplication routines of $\mathbf{C} \Leftarrow \alpha\, \mathbf{A} \cdot \mathbf{B}^T + \beta\, \mathbf{C}$ and $\mathbf{C} \Leftarrow \alpha\, \mathbf{A}^T \cdot \mathbf{B} + \beta\, \mathbf{C}$.

2.2. Block Cyclic Data Distribution

For performing the matrix multiplication $\mathbf{C} = \mathbf{A} \cdot \mathbf{B}$, we assume that $\mathbf{A}$, $\mathbf{B}$ and $\mathbf{C}$ are $M \times K$, $K \times N$, and $M \times N$, respectively. The distributed routine also requires a condition on the block size to ensure compatibility. That is, if the block size of $\mathbf{A}$ is $m_b \times k_b$, then that of $\mathbf{B}$ and $\mathbf{C}$ must be $k_b \times n_b$ and $m_b \times n_b$, respectively. So the number of blocks of matrices



	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 1: Block cyclic data distribution. A matrix with 12 × 12 blocks is distributed over a 2 × 3 processor grid. (a) The shaded and unshaded areas represent different grids. (b) It is easier to see the distribution from the processor point-of-view to implement algorithms. Each processor has 6 × 4 blocks.

A, **B**, and **C** are $M_g \times K_g$, $K_g \times N_g$, and $M_g \times N_g$, respectively, where $M_g = \lceil M \,/\, m_b \rceil$, $N_g = \lceil N \,/\, n_b \rceil$, and $K_g = \lceil K \,/\, k_b \rceil$.

The way in which a matrix is distributed over the processors has a major impact on the load balance and communication characteristics of the concurrent algorithm, hence, largely determines its performance and scalability. The block cyclic distribution provides a simple, general-purpose way of distributing a block-partitioned matrix on distributed-memory concurrent computers.

Figure 1(a) shows an example of the block cyclic data distribution, where a matrix with 12 × 12 blocks is distributed over a 2 × 3 grid. The numbered squares represent blocks of elements, and the number indicates the location in the processor grid – 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 a row of blocks and of a column of blocks, respectively. Figure 1(b) reflects the distribution from a processor point-of-view, where each processor has 6 × 4 blocks.

Denoting the least common multiple of P and Q by LCM, we refer to a square of LCM × LCM blocks as an LCM block. Thus, the matrix in Figure 1 may be viewed as a 2 × 2 array of LCM blocks. Blocks belong to the same processor if their relative locations are the same in each LCM block. A parallel algorithm, in which the order of execution can be intermixed such as matrix multiplication and matrix transposition, may be developed for the first LCM block. Then it can be directly applied to the other LCM blocks, which have the same structure and the same data distribution as the first LCM block, that is, when an operation is executed on the first LCM block, the same operation can be done simultaneously on other LCM blocks. And the LCM concept is applied to design software libraries for dense linear algebra computations with algorithmic blocking [17, 19].

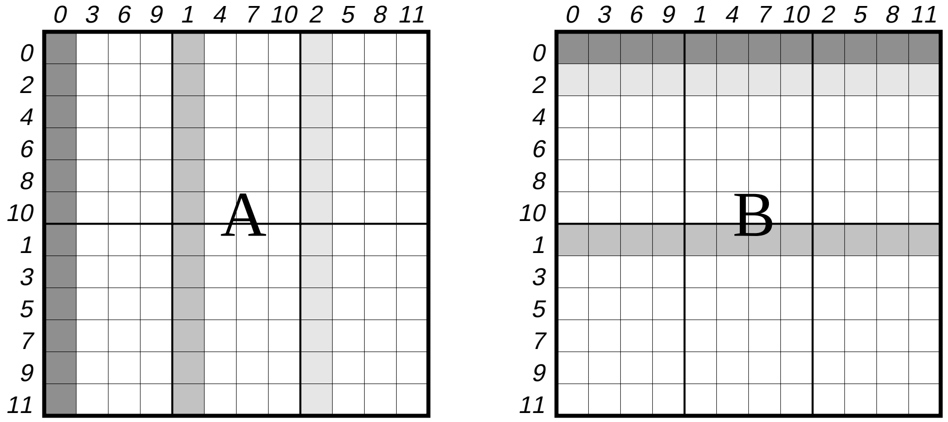


Figure 2: A snapshot of SUMMA. The darkest blocks are broadcast first, and lightest blocks are broadcast later.

3. Algorithms

3.1. SUMMA

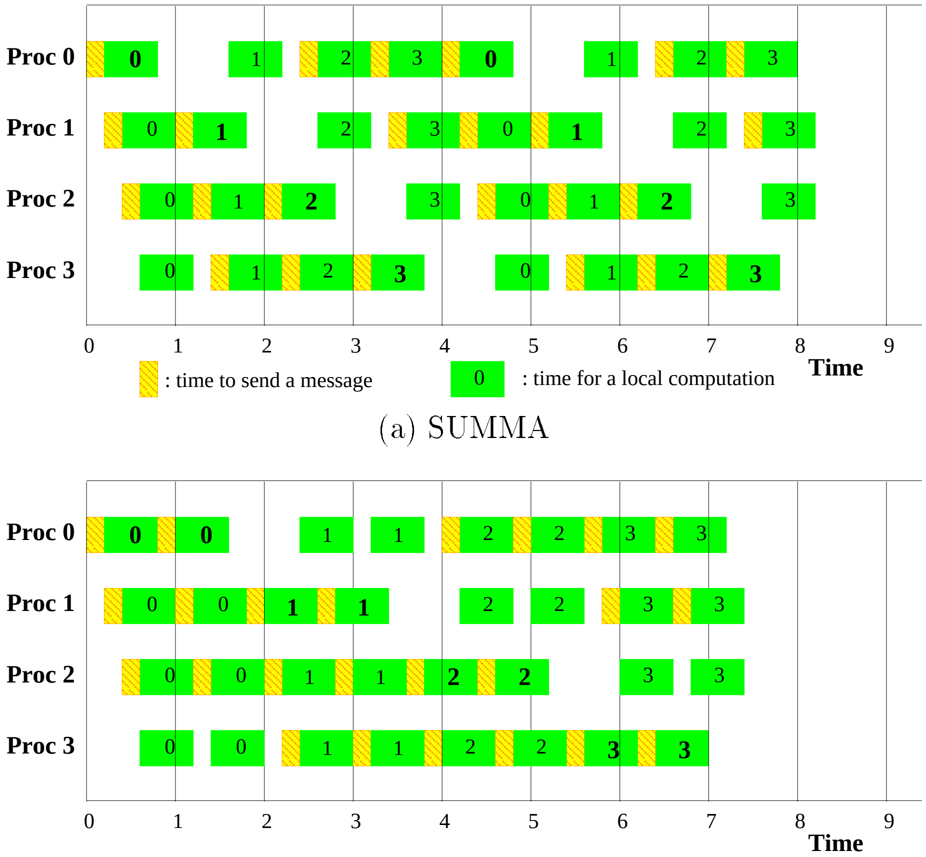
SUMMA is basically a sequence of rank-k_b updates. In SUMMA, **A** and **B** are divided into several columns and rows of blocks, respectively, whose block sizes are k_b. Processors multiply the first column of blocks of **A** with the first row of blocks of **B**. Then processors multiply the next column of blocks of **A** and the next row of blocks of **B** successively.

As the snapshot of Figure 2 shows, the first column of processors P_0 and P_3 begins broadcasting the first column of blocks of **A** ($A(:,0)$) along each row of processors (here we use MATLAB notation to simply represent a portion of a matrix.) At the same time, the first row of processors, P_0, P_1, and P_2 broadcasts the first row of blocks of **B** ($B(0,:)$) along each column of processors. After the local multiplication, the second column of processors, P_1 and P_4, broadcasts $A(:,1)$ rowwise, and the second row of processors, P_3, P_4, and P_5, broadcasts $B(1,:)$ columnwise. This procedure continues until the last column of blocks of **A** and the last row of blocks of **B**.

Agrawal, Gustavson and Zubair [1], and van de Geijn and Watts [18] obtained high efficiency on the Intel Delta and Paragon, respectively, by exploiting the pipelined communication scheme, where broadcasting is implemented as passing a column (or row) of blocks around the logical ring that forms the row (or column).

3.2. DIMMA

We show a simple simulation in Figure 3. It is assumed that there are 4 processors, each has 2 sets of data to broadcast, and they use blocking send and non-blocking receive. In the figure, the time to send a data set is assumed to be 0.2 seconds, and the time for local computation is 0.6 seconds. Then the pipelined broadcasting scheme takes 8.2 seconds as in Figure 3(a) .



(a) SUMMA

(b) DIMMA

Figure 3: Communication characteristics of SUMMA and DIMMA. It is assumed that blocking send and non-blocking receive are used.

A careful investigation of the pipelined communication shows there is an extra waiting time between two communication procedures. If the first processor broadcasts everything it contains to other processors before the next processor starts to broadcast its data, it is possible to eliminate the unnecessary waiting time. The modified communication scheme in Figure 3(b) takes 7.4 seconds. That is, the new communication scheme saves 4 communication times ($8.2 - 7.4 = 0.8 = 4 \times 0.2$). Figures 4 and 5 show a Paragraph visualization [13] of SUMMA and DIMMA on the Intel Paragon computer, respectively. Paragraph is a parallel programming tool that graphically displays the execution of a distributed-memory program. These figures include spacetime diagrams, which show the communication pattern between the processes, and utilization Gantt charts, which show when each process is busy or idle. The dark gray color signifies idle time for a given process, and the light gray color signals busy time. DIMMA is more efficient in communication than SUMMA as shown in these figures. The details of analysis of the algorithms is shown in Section 4.

With this modified communication scheme, DIMMA is implemented as follows. After the first procedure, that is, broadcasting and multiplying $A(:,0)$ and $B(0,:)$, the first column

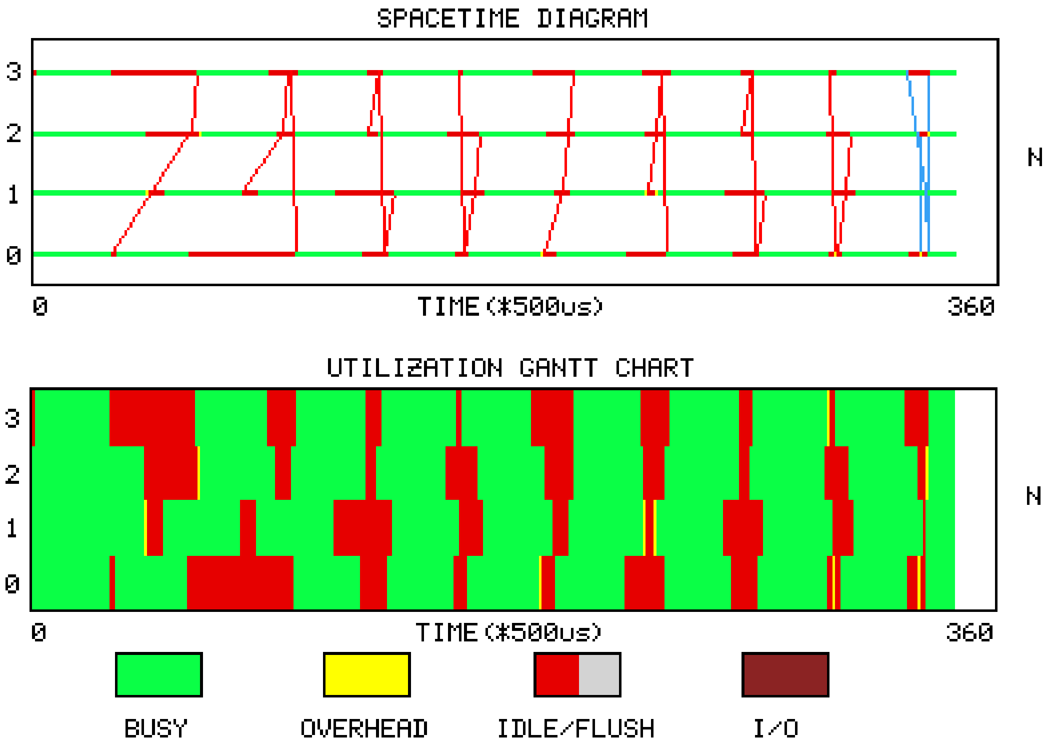


Figure 4: Paragraph visualization of SUMMA

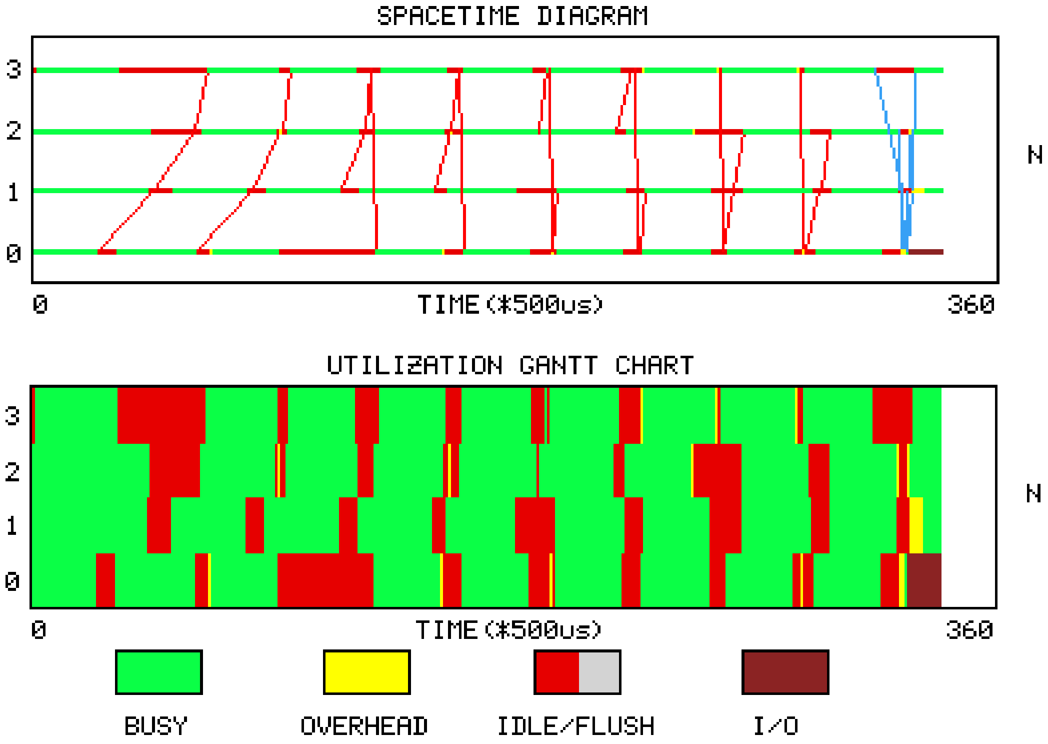


Figure 5: Paragraph visualization of DIMMA

of processors, P_0 and P_3, broadcasts $A(:,6)$ along each row of processors, and the first row of processors, P_0, P_1, and P_2 sends $B(6,:)$ along each column of processors, as shown in Figure 6. The value 6 appears since the LCM of $P = 2$ and $Q = 3$ is 6.

For the third and fourth procedures, the first column of processors, P_0 and P_3, broadcasts rowwise $A(:,3)$ and $A(:,9)$, and the second row of processors, P_3, P_4, and P_5, broadcasts columnwise $B(3,:)$ and $B(9,:)$, respectively. After the first column of processors, P_0 and P_3, broadcasts all of their columns of blocks of **A** along each row of processors, the second column of processors, P_1 and P_4, broadcasts their columns of **A**.

The basic computation of SUMMA and DIMMA in each processor is a sequence of rank-k_b updates of the matrix. The value of k_b should be at least 20 (Let k_{opt} be the optimal block size for the computation, then $k_{opt} = 20$) to optimize performance of the sequential BLAS routine, `DGEMM`, in the Intel Paragon, which corresponds to about 44 Mflops on a single node. The vectors of blocks to be multiplied should be conglomerated to form larger matrices to optimize performance if k_b is small.

DIMMA is modified with the LCM concept. The basic idea of the LCM concept is to handle simultaneously several thin columns of blocks of **A**, and the same number of thin rows of blocks of **B** so that each processor multiplies several thin matrices of **A** and **B** simultaneously in order to obtain the maximum performance of the machine. Instead of broadcasting a single column of **A** and a single row of **B**, a column of processors broadcasts several ($M_X = \lceil k_{opt}/k_b \rceil$) columns of blocks of **A** along each row of processors, whose distance is LCM blocks in the column direction. At the same time, a row of processors broadcasts the same number of blocks of **B** along each column of processors, whose distance is LCM blocks in the row direction as shown in Figure 7. Then each processor executes its own multiplication. The multiplication operation is changed from 'a sequence ($= K_g$) of rank-k_b updates' to 'a sequence ($= \lceil K_g \, / \, M_X \rceil$) of rank-($k_b \cdot M_X$) updates' to maximize the performance.

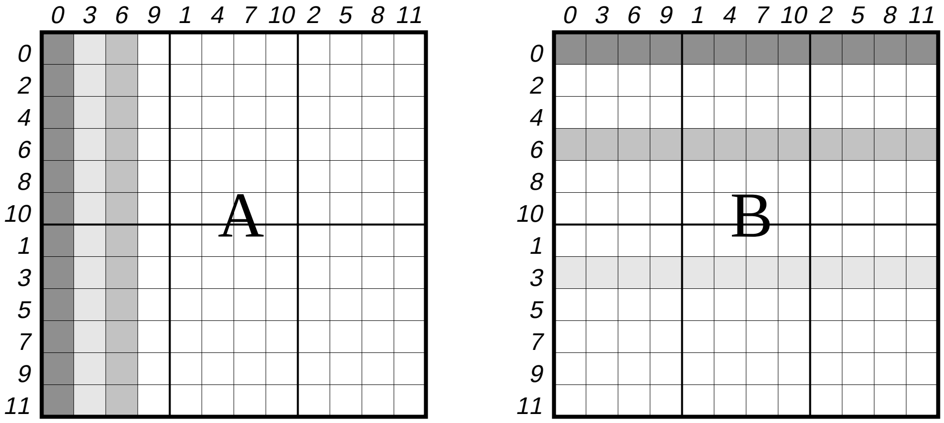


Figure 6: Snapshot of a simple version of DIMMA. The darkest blocks are broadcast first.

For example, if $P = 2, Q = 3, k_b = 10$ and $k_{opt} = 20$, the processors deal with 2 columns of blocks of **A** and 2 rows of blocks of **B** at a time ($M_X = \lceil k_b/k_{opt} \rceil = 2$). The first column of processors, P_0 and P_3, copies two columns of $A(:,[0,6])$ (that is, $A(:,0)$ and $A(:,6)$) to T_A,

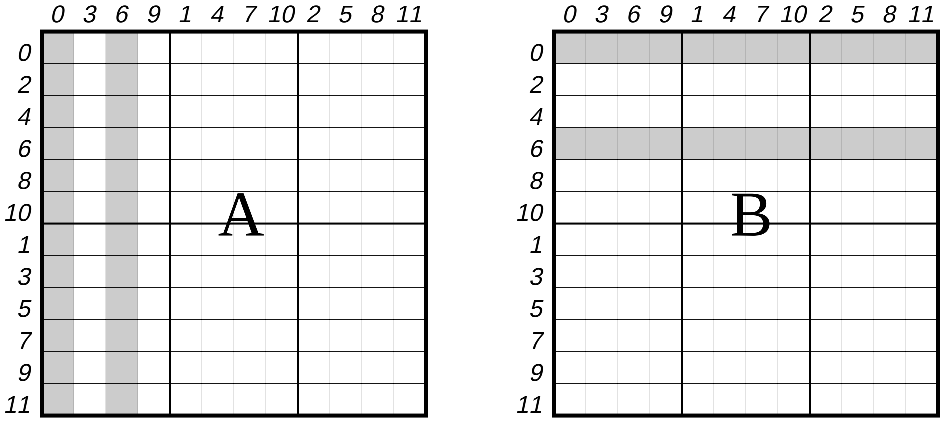


Figure 7: A snapshot of DIMMA

and broadcasts them along each row of processors. The first row of processors, P_0, P_1 and P_2, copies two rows of $B([0,6],:)$ (that is, $B(0,:)$ and $B(6,:)$) to T_B and broadcasts them along each column of processors. Then all processors multiply T_A with T_B to produce **C**. Next, the second column of processors, P_1 and P_4, copies the next two columns of $A(:,[1,7])$ to T_A and broadcasts them again rowwise, and the second row of processors, P_3, P_4 and P_5, copies the next two rows of $B([1,7],:)$ to T_B and broadcasts them columnwise. The product of T_A and T_B is added to **C** in each processor.

The value of M_X can be determined by the block size, available memory space, and machine characteristics such as processor performance and communication speed. If it is assumed that $k_{opt} = 20$, the value of M_X should be 4 if the block size is 5, and the value of M_X should be 2 if the block size is 10.

If k_b is much larger than the optimal value (for example, $k_b = 100$), it may be difficult to obtain good performance since it is difficult to overlap the communication with the computation. In addition, the multiplication routine requires a large amount of memory to send and receive **A** and **B**. It is possible to divide k_b into smaller pieces. For example, if $k_b = 100$, processors divide a column of blocks of **A** into five thin columns of blocks, and divide a row of blocks of **B** into five thin rows of blocks. Then they multiply each thin column of blocks of **A** with the corresponding thin row of blocks of **B** successively. The two cases, in which k_b is smaller and larger than k_{opt}, are combined, and the pseudocode of the DIMMA is shown in Figure 8.

4. Analysis of Multiplication Algorithms

We analyze the elapsed time of SUMMA and DIMMA based on Figure 3. It is assumed that $k_b = k_{opt}$ throughout the computation. Then, for the multiplication $\mathbf{C}_{M\times N} \Leftarrow \mathbf{C}_{M\times N} + \mathbf{A}_{M\times K} \times \mathbf{B}_{K\times N}$, there are $K_g = \lceil K/k_b \rceil$ columns of blocks of **A** and K_g rows of blocks of **B**.

At first, it is assumed that there are P linearly connected processors, in which a column

```
$C = 0$ $(C(:,:) = 0)$
$M_X = \lceil k_{opt}/k_b \rceil$
DO $L1 = 0, Q - 1$
    DO $L2 = 0, \mathrm{LCM}/Q - 1$
        $L_X = \mathrm{LCM} \cdot M_X$
        DO $L3 = 0, \lceil K_g/L_X \rceil - 1$
            DO $L4 = 0, \lceil k_b/k_{opt} \rceil - 1$
                $L_m = L1 + L2 \cdot Q + L3 \cdot L_X + [L4] : \mathrm{LCM} : (L3 + 1) \cdot L_X - 1$
                [Copy $A(:, L_m)$ to $T_A$ and broadcast it along each row of processors]
                [Copy $B(L_m, :)$ to $T_B$ and broadcast it along each column of processors]
                $C(:,:) = C(:,:) + T_A \cdot T_B$
            END DO
        END DO
    END DO
END DO
```

Figure 8: The pseudocode of DIMMA. The DO loop of $L3$ is used if k_b is smaller than k_{opt}, where the routine handles M_X columns of blocks of A and M_X rows of blocks of B, whose block distance are LCM, simultaneously, L_m is used to select them correctly. The innermost DO loop of $L4$ is used if k_b is larger than k_{opt}, and the bracket in $[L_4]$ represents the L_4-th thin vector.

of blocks of $\mathbf{A}$ $(= T_A)$ is broadcast along P processors at each step and a row of blocks of $\mathbf{B}$ $(= T_B)$ always stays in each processor. It is also assumed that the time for sending a column T_A to the next processor is t_c, and the time for multiplying T_A with T_B and adding the product to $\mathbf{C}$ is t_p. Actually $t_c = \alpha + (M\,k_b) \cdot \beta$ and $t_p = 2\,M\,\frac{N}{P}\,k_b \cdot \gamma$, where α is a communication start-up time, β is a data transfer time, and γ is a time for multiplication or addition.

For SUMMA, the time difference between successive two pipelined broadcasts of T_A is $2t_c + t_p$. The total elapsed time of SUMMA with K_g columns of blocks on an 1-dimensional processor grid, t^{1D}_{summa}, is

$$t^{1D}_{summa} = K_g\,(2t_c + t_p) - t_c + (P-2)\,t_c = K_g\,(2t_c + t_p) + (P-3)\,t_c.$$

For DIMMA, the time difference between the two pipelined broadcasts is $t_c + t_p$ if the T_As are broadcast from the same processor. However, the time differences is $2t_c + t_p$ if they are in different processors. The total elapsed time of DIMMA, t^{1D}_{dimma}, is

$$t^{1D}_{dimma} = K_g\,(t_c + t_p) + (P-1)t_c + (P-2)\,t_c = K_g\,(t_c + t_p) + (2P-3)\,t_c.$$

On a 2-dimensional $P \times Q$ processor grid, the communication time of SUMMA is doubled in order to broadcast T_B as well as T_A. Assume again that the time for sending a column T_A and a row T_B to the next processor are t_{ca} and t_{cb}, respectively, and the time for multiplying T_A with T_B and adding the product to $\mathbf{C}$ is t_p. Actually $t_{ca} = \alpha + (\frac{M}{P}\, k_b) \cdot \beta$, $t_{cb} = \alpha + (\frac{N}{Q}\, k_b) \cdot \beta$, and $t_p = 2\, \frac{M}{P}\, \frac{N}{Q}\, k_b \cdot \gamma$. So,

$$t^{2D}_{summa} = K_g\,(2t_{ca} + 2t_{cb} + t_p) + (Q-3)\, t_{ca} + (P-3)\, t_{cb}. \tag{1}$$

For DIMMA, each column of processors broadcasts T_A until everything is sent. Meanwhile, rows of processors broadcast T_B if they have the corresponding T_B with the T_A. For a column of processors, which currently broadcasts $\mathbf{A}$, P/GCD rows of processors, whose distance is GCD, have rows of blocks of $\mathbf{B}$ to broadcast along with the T_A, where GCD is the greatest common divisor of P and Q. The extra idle wait, caused by broadcasting two T_Bs when they are in different processors, is $\mathrm{GCD} \cdot t_{cb}$. Then the total extra waiting time to broadcast T_Bs is $Q\,(P/\mathrm{GCD})\,\mathrm{GCD} \cdot t_{cb} = P\,Q \cdot t_{cb}$.

However, if $\mathrm{GCD} = P$, only one row of processors has T_B to broadcast corresponding to the column of processors, and the total extra waiting time is $P \cdot t_{cb}$. So,

$$\begin{aligned} t^{2D}_{dimma} &= K_g\,(t_{ca} + t_{cb} + t_p) + (2Q-3)t_{ca} + (P+Q-3)t_{cb} && \text{if GCD} = P \\ &= K_g\,(t_{ca} + t_{cb} + t_p) + (2Q-3)t_{ca} + (PQ+P-3)t_{cb} && \text{otherwise.} \end{aligned} \tag{2}$$

The time difference between SUMMA and DIMMA is

$$\begin{aligned} t^{2D}_{summa} - t^{2D}_{dimma} &= (K_g - Q)\, t_{ca} + (K_g - P)\, t_{cb} && \text{if GCD} = P, \\ &= (K_g - Q)\, t_{ca} + (K_g - PQ)\, t_{cb} && \text{otherwise.} \end{aligned} \tag{3}$$

5. Implementation and Results

We implemented three algorithms, called them SUMMA0, SUMMA and DIMMA, and compared their performance on the 512 node Intel Paragon at the Oak Ridge National Laboratory, Oak Ridge, U.S.A., and the 256 node Intel Paragon at Samsung Advanced Institute of Technology, Suwon, Korea. SUMMA0 is the original version of SUMMA, which has the pipelined broadcasting scheme and the fixed block size, k_b. The local matrix multiplication in SUMMA0 is the rank-k_b update. SUMMA is a revised version of SUMMA0 with the LCM block concept for the optimized performance of `DGEMM`, so that the local matrix multiplication is a rank-k_{approx} update, where k_{approx} is computed in the implementation as follows:

$$\begin{aligned} k_{approx} &= \lfloor k_{opt} \,/\, k_b \rfloor \;\cdot\; k_b && \text{if} k_{opt} \geq k_b, \\ &= \lfloor k_b \,/\, \lceil k_b / k_{opt} \rceil \rfloor && \text{otherwise.} \end{aligned}$$

First of all, we changed the block size, k_b, and observed how the block size affects the

$P \times Q$	Matrix Size	Block Size	SUMMA0	SUMMA	DIMMA
8×8	2000×2000	1×1	1.135	2.678	2.735
		5×5	2.488	2.730	2.735
		20×20	2.505	2.504	2.553
		50×50	2.633	2.698	2.733
		100×100	1.444	1.945	1.948
8×8	4000×4000	1×1	1.296	2.801	2.842
		5×5	2.614	2.801	2.842
		20×20	2.801	2.801	2.842
		50×50	2.674	2.822	2.844
		100×100	2.556	2.833	2.842
12×8	4000×4000	1×1	1.842	3.660	3.731
		5×5	3.280	3.836	3.917
		20×20	3.928	3.931	4.006
		50×50	3.536	3.887	3.897
		100×100	2.833	3.430	3.435

Table 1: Dependence of performance on block size (Unit: Gflops)

performance of the algorithms. Table 1 shows the performance of $\mathbf{A} = \mathbf{B} = \mathbf{C} = 2000 \times 2000$ and 4000×4000 on 8×8 and 16×8 processor grids with block sizes $k_b = 1, 5, 20, 50,$ and 100. At first SUMMA0 and SUMMA are compared. With the extreme case of $k_b = 1$, SUMMA with the modified blocking scheme performed at least 100% better than SUMMA0. When $k_b = 5$, SUMMA shows 7 - 10% enhanced performance. If the block size is much larger than the optimal block size, that is, $k_b = 50$, or 100, SUMMA0 becomes inefficient again and it has a difficulty in overlapping the communications with the computations. SUMMA outperformed SUMMA0 about $5 \sim 10\%$ when $A = B = C = 4000 \times 4000$ and $k_b = 50$ or 100 on 8×8 and 12×8 processor grids.

Note that on an 8×8 processor grid with 2000×2000 matrices, the performance of $k_b = 20$ or 100 is much slower than that of other cases. When $k_b = 100$, the processors in the top half have 300 rows of matrices, while those in the bottom half have just 200 rows. This leads to load imbalance among processors, and the processors in the top half require 50% more local computation.

Now SUMMA and DIMMA are compared. Figures 9 and 10 show the performance of SUMMA and DIMMA on 16×16 and 16×12 processor grids, respectively, with the fixed block size, $k_b = k_{opt} = 20$. DIMMA always performs better than SUMMA on the 16×16 processor grid. These matrix multiplication algorithms require $O(N^3)$ flops and $O(N^2)$ communications, that is, the algorithms are computation intensive. For a small matrix of $N = 1000$, the performance difference between the two algorithms is about 10%. But for a large matrix, these algorithms require much more computation, so that the performance difference caused by the different communication schemes becomes negligible. For $N = 8000$, the performance difference is only about $2 \sim 3\%$. On the 16×12 processor grid, SUMMA

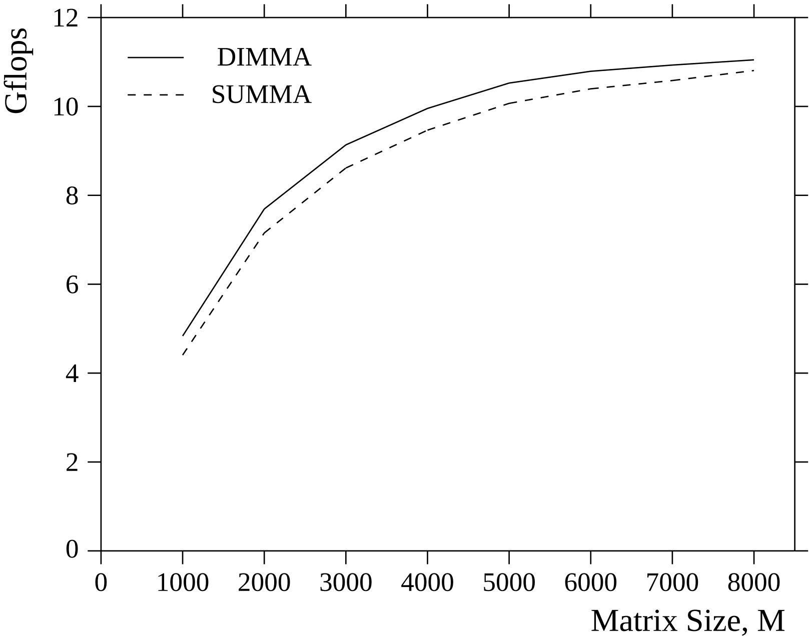


Figure 9: Performance of SUMMA and DIMMA on a 16×16 processor grid. ($k_{opt} = k_b = 20$).

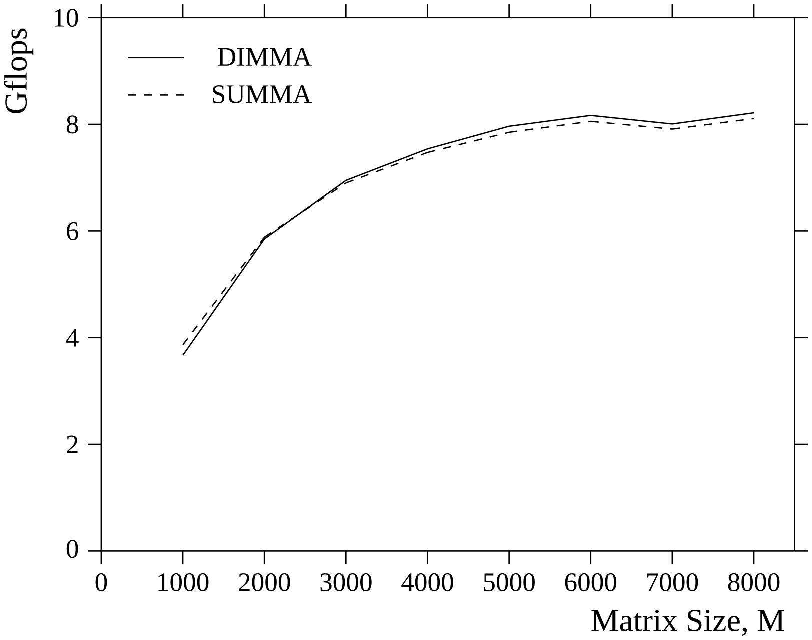


Figure 10: Performance of SUMMA and DIMMA on a 16 × 12 processor grid.

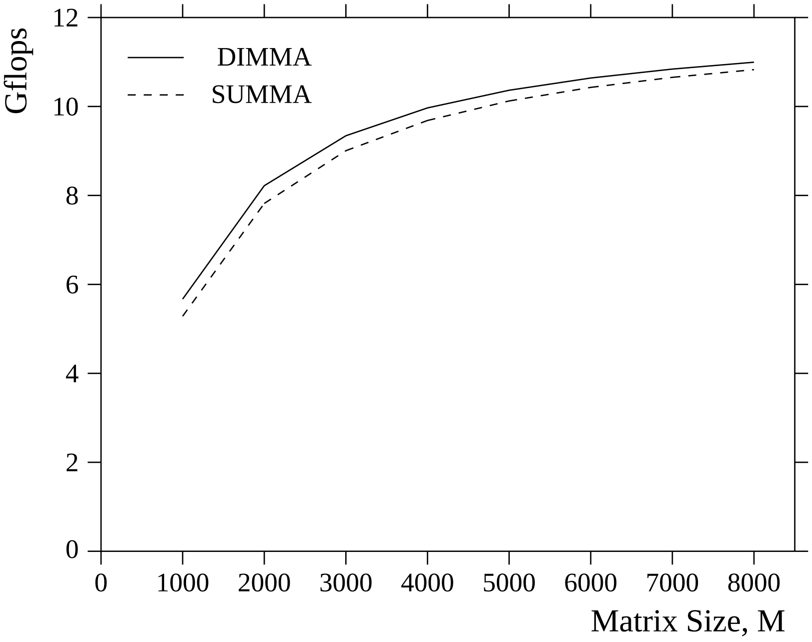


Figure 11: Predicted Performance of SUMMA and DIMMA on a 16 × 16 processor grid.

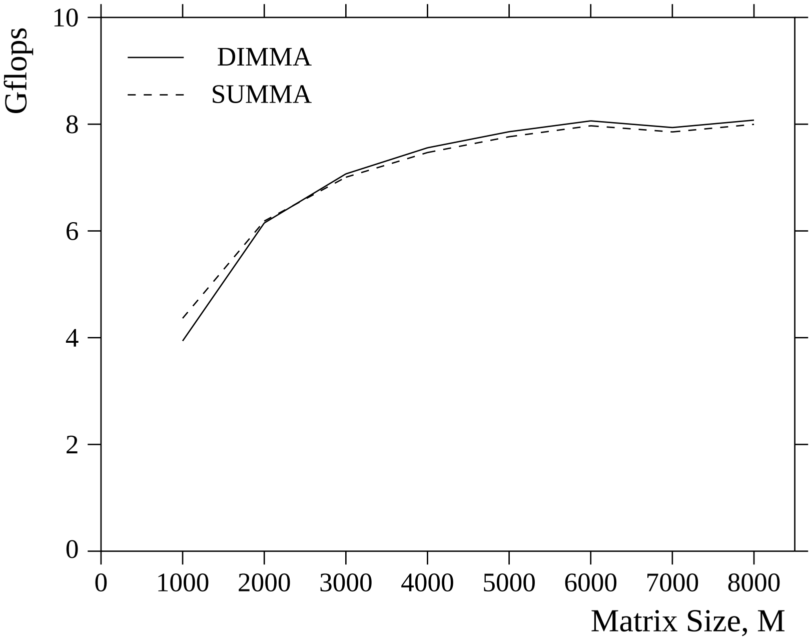


Figure 12: Predicted Performance of SUMMA and DIMMA on a 16 × 12 processor grid.

performs slightly better than DIMMA for small size of matrices, such as $N = 1000$ and 2000. If $P = 16$ and $Q = 12$, GCD $= 4\,(\neq P)$. For the problem of $M = N = K = 2000$ and $k_{opt} = k_b = 20$, $K_g = K/k_b = 100$. From Eq. 3,

$$t^{2D}_{summa} - t^{2D}_{dimma} = (100 - 12)\,t_{ca} + (100 - 16 \times 12)\,t_{cb} = 88t_{ca} - 92t_{cb}.$$

From the result, it is expected that the SUMMA is faster than DIMMA for the problem if $t_{ca} = t_{cb}$.

We predicted the performance on the Intel Paragon using Eqs 1 and 2. Figures 11 and 12 show the predicted performance of SUMMA and DIMMA corresponding to Figures 9 and 10, respectively. We used $\alpha = 94.75\mu sec$, $\beta = 0.02218$ (45 Mbytes/sec), $\gamma = 22.88nsec$ (43.7 Mflops per node) for the predicted performance. (Those values are observed in practice.)

In Eq. 2, the idle wait, $(2Q - 3)t_{ca} + (PQ + P - 3)t_{cb}$ when GCD $\neq P$, can be reduced by a slight modification of the communication scheme. For example, when $P = 4, Q = 8$ (that is, GCD $= Q$) if a column of processors sends all columns of blocks of **B** instead of a row of processors send all rows of blocks of **A** as in Figure 8, the waiting time is reduced to $(P + Q - 3)t_{ca} + (2P - 3)t_{cb}$.

The following example has another communication characteristic. After the first column and the first row of processors send their own **A** and the corresponding **B**, respectively, then, for the next step, the second column and the second row of processors send their **A** and **B**, respectively. The communication resembles that of SUMMA, but the processors send all corresponding blocks of **A** and **B**. The waiting time is $(\mathrm{LCM}+Q-3)t_{ca} + (\mathrm{LCM}+P-3)t_{cb}$. This modification is faster if $2 \leq \mathrm{GCD} < \mathrm{MIN}(P, Q)$.

The performance per node of SUMMA and DIMMA is shown in Figures 13 and 14, respectively, when memory usage per node is held constant. Both algorithms show good performance and scalability, but DIMMA is always better. If each processor has a local problem size of more than 200×200, the DIMMA always reaches 40 Mflops per processor, but the SUMMA obtained about 38 Mflops per processor.

Currently the modified blocking scheme in DIMMA uses the rank-k_{approx} update. However it is possible to modify the DIMMA with the exact rank-k_{opt} update by dividing the virtually connected LCM blocks in each processor. The modification complicates the algorithm implementation, and since the performance of `DGEMM` is not sensitive to the value of k_{opt} (if it is larger than 20), there would be no improvement in performance.

6. Conclusions

We present a new parallel matrix multiplication algorithm, called DIMMA, for block cyclic data distribution on distributed-memory concurrent computers. DIMMA is the most efficient and scalable matrix multiplication algorithm. DIMMA uses the modified pipelined broadcasting scheme to overlap computation and communication effectively, and exploits the LCM block concept to obtain the maximum performance of the sequential BLAS routine regardless of the block size. DIMMA always shows the same high performance even when

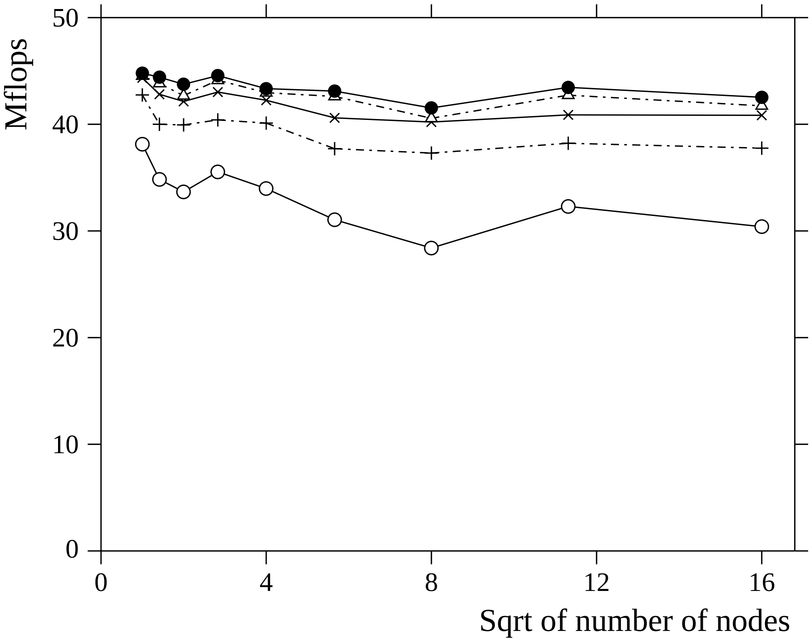


Figure 13: Performance per node of SUMMA where memory use per node is held constant. The five curves represent 100 × 100, 200 × 200, 300 × 300, 400 × 400, and 500 × 500 local matrices per node from the bottom.

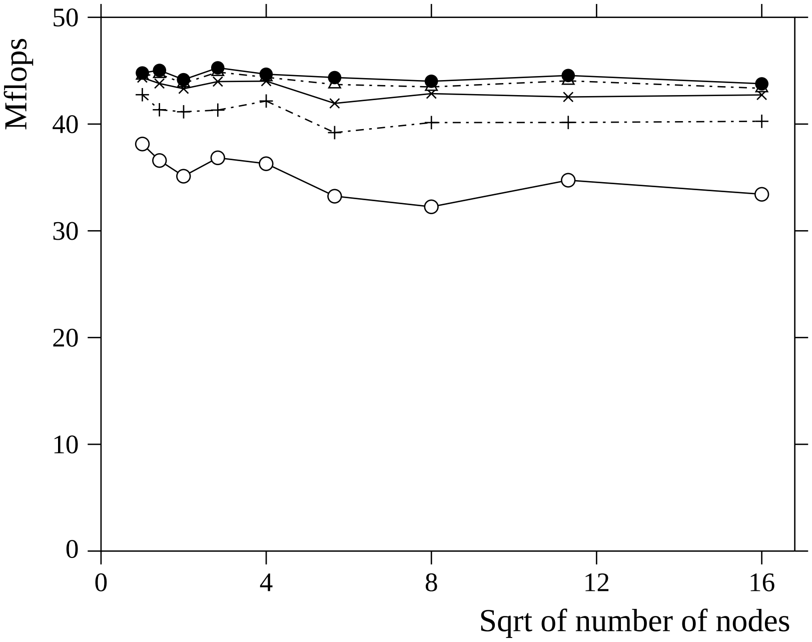


Figure 14: Performance per node of DIMMA.

the block size k_b is very small as well as very large if the matrices are evenly distributed among processors.

Acknowledgement

The author appreciates the anonymous reviewers for their helpful comments to improve the quality of the paper The author appreciates Dr. Jack Dongarra at Univ. of Tennessee at Knoxville and Oak Ride National Laboratory for providing computing facilities to perform this work. And he appreciates the anonymous reviewers for their helpful comments to improve the quality of the paper. This research was performed in part using the Intel Paragon computers both at the Oak Ridge National Laboratory, Oak Ridge, U.S.A. and at Samsung Advanced Institute of Technology, Suwon, Korea. And it was supported by the Korean Ministry of Information and Communication under contract 96087-IT1-I2.

7. References

[1] R. C. Agarwal, F. G. Gustavson, and M. Zubair. A High-Performance Matrix-Multiplication Algorithm on a Distributed-Memory Parallel Computer Using Overlapped Communication. *IBM Journal of Research and Development*, 38(6):673–681, 1994.

[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] L. Blackford, J. Choi, A. Cleary, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. Whaley. ScaLAPACK: A Portable Linear Algebra Library for Distributed Memory Computers - Design Issues and Performance. In *Proceedings of Supercomputing '96*, 1996. (http://www.supercomp.org/sc96/proceedings/).

[4] L. E. Cannon. A Cellular Computer to Implement the Kalman Filter Algorithm. 1969. Ph.D. Thesis, Montana State University.

[5] J. Choi, J. J. Dongarra, S. Ostrouchov, A. P. Petitet, D. W. Walker, and R. C. Whaley. A Proposal for a Set of Parallel Basic Linear Algebra Subprograms. LAPACK Working Note 100, Technical Report CS-95-292, University of Tennessee, 1995.

[6] J. Choi, J. J. Dongarra, R. Pozo, D. C. Sorensen, and D. W. Walker. CRPC Research into Linear Algebra Software for High Performance Computers. *International Journal of Supercomputing Applications*, 8(2):99–118, Summer 1994.

[7] J. Choi, J. J. Dongarra, and D. W. Walker. PUMMA: Parallel Universal Matrix Multiplication Algorithms on Distributed Memory Concurrent Computers. *Concurrency: Practice and Experience*, 6:543–570, 1994.

[8] J. Choi, J. J. Dongarra, and D. W. Walker. PB-BLAS: A Set of Parallel Block Basic Linear Algebra Subprograms. *Concurrency: Practice and Experience*, 8:517–535, 1996.

[9] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. Duff. A Set of Level 3 Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 18(1):1–17, 1990.

[10] 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.

[11] 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.

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

[13] M. T. Heath and J. A. Etheridge. Visualizing the Performance of Parallel Programs. *IEEE Software*, 8(5):29–39, September 1991.

[14] S. Huss-Lederman, E. M. Jacobson, and A. Tsao. Comparison of Scalable Parallel Multiplication Libraries. In *The Scalable Parallel Libraries Conference, (Starksville, MS)*, pages 142–149. IEEE Computer Society Press, October 6-8, 1993.

[15] S. Huss-Lederman, E. M. Jacobson, A. Tsao, and G. Zhang. Matrix Multiplication on the Intel Touchstone Delta. *Concurrency: Practice and Experience*, 6:571–594, 1994.

[16] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing.* The Benjamin/Cummings Publishing Company, Inc., Redwood City, CA, 1994.

[17] A. Petitet. Algorithmic Redistribution Methods for Block Cyclic Decompositions. 1996. Ph.D. Thesis, University of Tennessee, Knoxville.

[18] R. van de Geijn and J. Watts. SUMMA Scalable Universal Matrix Multiplication Algorithm. LAPACK Working Note 99, Technical Report CS-95-286, University of Tennessee, 1995.

[19] R. A. van de Geijn. *Using PLAPACK.* The MIT Press, Cambridge, 1997.