A high-performance matrix-multiplication algorithm on a distributed-memory parallel computer, using overlapped communication

by R. C. Agarwal
F. G. Gustavson
M. Zubair

In this paper, we propose a scheme for matrix–matrix multiplication on a distributed-memory parallel computer. The scheme hides almost all of the communication cost with the computation and uses the standard, optimized Level-3 BLAS operation on each node. As a result, the overall performance of the scheme is nearly equal to the performance of the Level-3 optimized BLAS operation times the number of nodes in the computer, which is the peak performance obtainable for parallel BLAS. Another feature of our algorithm is that it can give peak performance for larger matrices, even if the underlying communication network of the computer is slow.

Introduction

One means of obtaining very good performance on a distributed-memory parallel computer is by keeping communication cost as small as possible. Furthermore, it is equally important that the single-processor computation performance be good, since any improvement in the computation performance of a single processor will have

The contribution of M. Zubair was made when he was visiting the IBM Thomas J. Watson Research Center from Old Dominion University during the summer of 1992.

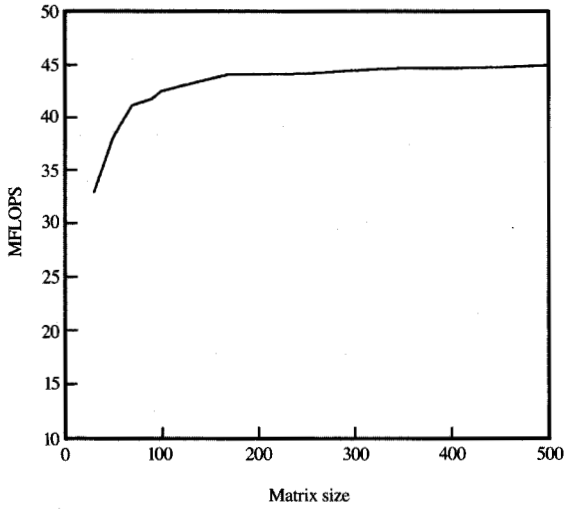


Figure 1

Performance of DGEMM on RS/6000 Model 530 as a function of matrix size (peak performance, 50 MFLOPS).

a multiplicative effect on the overall performance. For Level-3 Basic Linear Algebra Operations (BLAS) [1], which is a subroutine library, it has been established that a key for achieving good performance on a single processor with a memory hierarchy is the effective utilization of that hierarchy [2, 3]. For a given memory hierarchy, there is a minimum problem size above which nearly maximal performance can be obtained on a single processor, but below which performance drops off sharply as the problem size decreases [4]. As a consequence, when a problem is partitioned for execution on a parallel processor, it does not pay to subdivide the original problem into pieces below this minimum size. The minimum size implies a maximum number of processors that can be usefully applied; more processors do not increase performance proportionally and are therefore wasteful of resources. The main result of this paper is to combine good uniprocessor performance (BLAS) and overlapped computation and communication to produce a very high-performance algorithm for parallel matrix multiplication.

Our algorithm for matrix–matrix multiplication on a distributed-memory parallel computer has the following two properties: 1) it hides almost all of the communication cost with the computation, and 2) it uses the standard, optimized Level-3 BLAS matrix-multiply operation on each processor. Thus, when the problem size is large enough and the number of processors does not exceed the limit for efficient performance, the overall performance of the algorithm is nearly equal to the performance of the Level-3 optimized BLAS operation on a single processor times the number of processors used, which can be the peak performance obtainable for a parallel BLAS solution. Another feature of our algorithm is that it can yield peak performance, even if the underlying communication network of the computer is slow. This is because, for matrix multiplication, the interprocessor communication cost grows an order of magnitude more slowly than the computation cost. It follows from this that for any parallel computer, there is a break-even matrix size for which the computation cost balances the communication cost. Consequently, it is possible to choose a sufficiently large matrix size for which computation completely overlaps communication. (This assumes, of course, that the individual processors contain sufficient storage, which is almost always true, in practice.)

Several researchers have previously addressed the problem of parallelizing matrix–matrix multiplication, for example, [5–11]. A good survey of these efforts can be found in [12, 13]. More recently, an effort to develop a general-purpose matrix-multiplication algorithm for distributed-memory computers that uses optimized, single-processor BLAS has been described [14, 15]. We are not aware, however, of any algorithm that overlaps communication with computation.

The two main concepts presented here have very broad applicability. The first concept, embodied in our algorithm, is to anticipate the next computational step and prepare for it by communicating the data needed for it in advance (during the current computational step). The second main concept is to use very powerful processors and to compute on them nearly at their peak rate. This usually implies that one uses medium-grain to coarse-grain parallelism. Two other areas where these concepts, in combination, will work are dense linear algebra and signal processing (fast Fourier transform).

Our algorithm for computing a product matrix **C** is based on an outer-product formulation that consists of k' basic steps, in each of which all processors participate in computing an update to the **C** matrix and in simultaneously transmitting and receiving data for the next update of **C**. Normally, interprocessor communication is required between steps; however, we overlap communication with computation by initiating the necessary interprocessor communication a step in advance. That is, at the ith step, when we are calculating the ith update to the **C** matrix, we are also distributing data that are required for forming the $(i + 1)$th update to the **C** matrix. During any step, two pairs of buffers are used on a processor—one pair for receiving data from other processors and the other pair for local computing. In the next step, the functionality of the two buffer pairs is interchanged; i.e., the pair used for

receiving data in step i is used for local computing in step $i + 1$, and vice versa.

We tested the proposed algorithm, using the SGEMM (a matrix–matrix multiplication BLAS) supplied by the manufacturer, on two distributed-memory computers, the Intel iPSC® System 860 and the Intel Touchstone Delta. All submatrices were the same size. We implemented broadcast communication of the data as a sequence of individual "sends" from processor to processor along the rows and columns of the processor mesh. We obtained a peak performance of 4.36 GFLOPS on a 128-processor Intel iPSC System 860 computer (approximately 34.0 MFLOPS per processor) for a 5440 × 5440 matrix. For this computation, we were able to overlap 93% of the computation with broadcast communication. The single-processor SGEMM rate for this size of matrix was measured at 36.7 MFLOPS. We obtained a peak performance of 19.0 GFLOPS on the Delta with 512 processors for a 14 720 × 14 720 matrix. For this computation, we were able to overlap 96% of the computation with broadcast communication. The single-processor SGEMM rate for this size of matrix was measured at 38.6 MFLOPS.

Single-node performance

The peak processing rate of a massively parallel system can be obtained when all of its nodes run simultaneously at their peak rate. One way to accomplish this is to divide the problem into pieces, at least one for each processor, that the processors can run, independently, at their peak performance. The execution times for the pieces should be identical. It is not a trivial exercise to write a program for which the pieces run at peak performance for all piece sizes. We have been able, however, to design and implement SGEMM/DGEMM programs [4] for uniprocessors that run near peak performance for a wide range of problem sizes. For example, in **Figure 1** we plot the performance (in MFLOPS) of DGEMM for square matrices of order 30 to 500 on the IBM RISC System/6000® (RS/6000) Model 530 computer, which has a peak rate of 50 MFLOPS. As can be seen, the performance rapidly approaches a peak rate of 45 MFLOPS, attaining a rate of 43 MFLOPS at matrix order 100. (Although we show this graph for square matrices only, DGEMM on the RS/6000 also gives uniformly good performance near its peak rate for rectangular matrices.)

On the other hand, some vendor-supplied software may not run near peak performance for a wide range of problem sizes. For example, in **Figure 2** we plot SGEMM performance on a node of the Intel iPSC System 860 computer, which has a peak rate of 80 MFLOPS for single-precision calculations, for square matrices of order 50 to 500. Clearly, with such an SGEMM performance curve, one must select the size of the pieces carefully in order to optimize performance of a parallel system.

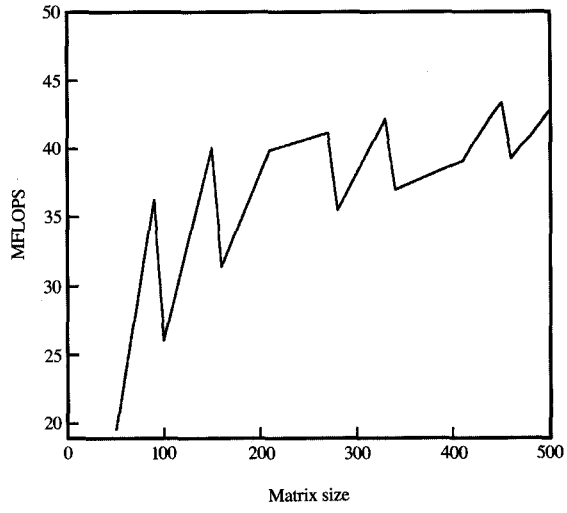


Figure 2

Performance of SGEMM on a node of Intel iPSC System 860 as a function of matrix size (peak performance, 80 MFLOPS).

In summary, the uniformly good performance of our matrix multiplication code depends crucially upon having uniprocessor code performance that is uniformly near peak for as wide a range of piece size as possible.

Multiplying matrices in parallel

We consider the matrix–matrix multiplication computation $\mathbf{C} = \mathbf{AB}$, where $\mathbf{C}$, $\mathbf{A}$, and $\mathbf{B}$ are $M \times N$, $M \times K$, and $K \times N$ matrices, respectively, on a computer with $P = m' \times n'$ processors. (We use primed symbols to denote quantities related to the array of physical processors on which the algorithm will run, and unprimed symbols to represent the sizes of various matrices on a given processor.) This computation is the Level-3 BLAS SGEMM computation (BLAS also incorporates scale factors α and β and matrix transposes).

We view a parallel computer as a logical two-dimensional $m' \times n'$ array of processors. The processor at position (i', j') is denoted $p(i', j')$, where $0 \le i' < m'$ and $0 \le j' < n'$.

- *Basic algorithm (without overlap)*

The basic algorithm, in its simplest form, is as follows:

1. Partition the three matrices into submatrices, designated $\mathbf{A}_{i'j'}$, $\mathbf{B}_{i'j'}$, and $\mathbf{C}_{i'j'}$.

2. Store submatrices $\mathbf{A}_{i'j'}$ and $\mathbf{B}_{i'j'}$ in the local storage of processor $p(i', j')$.
3. In processor $p(i', j')$, calculate

$$\mathbf{C}_{i'j'} = \sum_{\sigma'=0}^{k'-1} \mathbf{A}_{i'\sigma'}\mathbf{B}_{\sigma'j'},$$

where $\mathbf{A}_{i'\sigma'}\mathbf{B}_{\sigma'j'}$ denotes a matrix multiplication, and σ' and k' are as described below. All processors make these computations simultaneously. Since only $\mathbf{A}_{i'j'}$ and $\mathbf{B}_{i'j'}$ are stored in $p(i', j')$, it is necessary that submatrices $\mathbf{A}_{i'\sigma'}$ and $\mathbf{B}_{\sigma'j'}$ be transmitted to $p(i', j')$ from the processors where they are stored [$p(i', \sigma')$ and $p(\sigma', j')$, respectively].

Matrix **A** is partitioned into $m' \times k'$ submatrices $\mathbf{A}_{i'\sigma'}$ of order m by k each, where $m = M/m'$ and $k \times k' = K$. [In the simplest case, the processor array is square ($m' = n'$), and we set $k' = n'$.] The submatrix $\mathbf{A}_{i'\sigma'}$ is defined as follows:

$$\mathbf{A}_{i'\sigma'} \equiv \mathbf{A}(i'm:[i'+1]m-1, \sigma'k:[\sigma'+1]k-1),$$
$$0 \le i' < m', 0 \le \sigma' < k'.$$

[$\mathbf{X}(a:b, c:d)$ denotes the submatrix of matrix **X** with row index ranging from a through b and column index ranging from c through d.] That is, the rows of **A** are divided among the rows of the processor array in blocks of m, and the columns of **A** are divided among the columns of the processor array in blocks of k. Similarly, matrix **B** is partitioned into $k' \times n'$ submatrices $\mathbf{B}_{\sigma'j'}$ of order k by n each, where $n = N/n'$:

$$\mathbf{B}_{\sigma'j'} \equiv \mathbf{B}(\sigma'k:[\sigma'+1]k-1, j'n:[j'+1]n-1),$$
$$0 \le \sigma' < k', 0 \le j' < n'.$$

That is, the columns of **B** are divided among the columns of the processor array in blocks of n, and the rows of **B** are divided among the rows of the processor array in blocks of k. Finally, **C** is partitioned into $m' \times n'$ submatrices $\mathbf{C}_{i'j'}$ of order m by n each:

$$\mathbf{C}_{i'j'} \equiv \mathbf{C}(i'm:[i'+1]m-1, j'n:[j'+1]n-1),$$
$$0 \le i' < m', 0 \le j' < n'.$$

That is, the rows of **C** are divided among the rows of the processor array in blocks of m, and the columns of **C** are divided among the columns of the processor array in blocks of n.

For example, we might wish to multiply a 1000×4000 matrix **A** by a 4000×6400 matrix **B** ($M = 1000$, $K = 4000$, $N = 6400$) on a 20×20 array of 400 processors ($m' = n' = 20$). Then, $m = 50$ and $n = 320$. If we choose $k' = 50$ and $k = 80$, each processor stores one 50×80 submatrix of **A** and an 80×320 submatrix of **B**, and calculates a 50×320 submatrix of **C**. As indicated above, submatrices $\mathbf{A}_{i'j'}$, $\mathbf{B}_{i'j'}$, and $\mathbf{C}_{i'j'}$ are stored in processor $p(i', j')$.

As stated earlier, our algorithm is based on an outer-product formulation and consists of k' basic steps, one for each of the k' columns of submatrices into which **A** is partitioned (and rows of submatrices into which **B** is partitioned). In a basic step, each processor $p(i', j')$ may transmit to other processors the data they need for making their computations, may receive such data from other processors, and computes $\mathbf{C}_{i'j'} = \mathbf{C}_{i'j'} + \mathbf{A}_{i'\sigma'}\mathbf{B}_{\sigma'j'}$, which is a matrix multiplication and accumulation. In addition to the local memory in a processor to store the submatrices of **A**, **B**, and **C**, two temporary buffers are used: $R0$ and $S0$, of size mk and kn, respectively.

To calculate $\mathbf{C}_{i'j'}$ on $p(i', j')$, array $\mathbf{C}_{i'j'}$ is initially set to **0**. Then, for $\sigma' = 0$ to $k' - 1$, the following sequence (basic step) is repeated:

1. All processors in the σ'th column send a copy of their **A** submatrices to all the other processors in the same row; i.e., $p(\lambda', \sigma')$ broadcasts $\mathbf{A}_{\lambda'\sigma'}$ to all other $p(\lambda', j')$s, for all $j' \neq \sigma'$. For example, in step 0, $p(0, 0)$ broadcasts $\mathbf{A}_{00}$ to $p(0, 1), p(0, 2), \cdots, p(0, n'-1)$. The receiving processors receive such submatrices of **A** into buffer $R0$.
2. Next, all processors in the σ'th row send a copy of their **B** submatrices to all the other processors in the same column. The receiving processors receive such submatrices of **B** into buffer $S0$.
3. Finally, processor $p(i', j')$ computes the following:

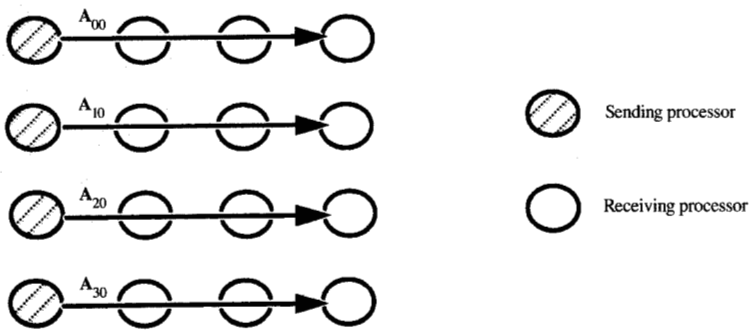
$$\mathbf{C}_{i'j'} = \mathbf{C}_{i'j'} + \mathbf{A}_{i'j'}\mathbf{B}_{i'j'}, \quad \text{if } i' = j' = \sigma',$$
$$\mathbf{C}_{i'j'} = \mathbf{C}_{i'j'} + \mathbf{R0}\,\mathbf{B}_{i'j'}, \quad \text{if } i' = \sigma', j' \neq \sigma',$$
$$\mathbf{C}_{i'j'} = \mathbf{C}_{i'j'} + \mathbf{A}_{i'j'}\mathbf{S0}, \quad \text{if } i' \neq \sigma', j' = \sigma',$$
$$\mathbf{C}_{i'j'} = \mathbf{C}_{i'j'} + \mathbf{R0}\,\mathbf{S0}, \quad \text{if } i' \neq \sigma', j' \neq \sigma'.$$

(**R0** and **S0** are the matrices in buffers $R0$ and $S0$, respectively.)

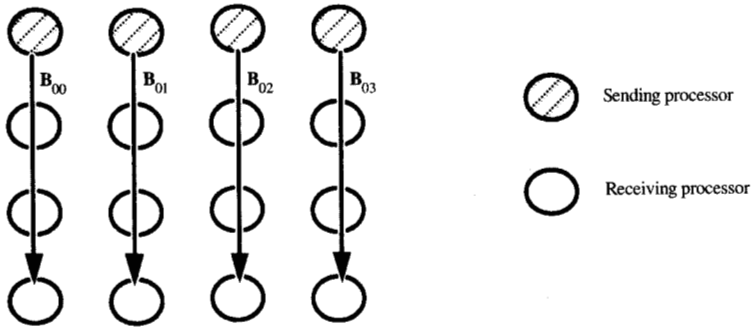
Figure 3 illustrates step 0 on an array of 4×4 processors. Phase 3 of each step requires a matrix multiplication and accumulation. After k' such basic steps, $p(i', j')$ contains the product submatrix $\mathbf{C}_{i'j'}$. Of course, if it is desired to calculate $\mathbf{C} = \mathbf{C} + \mathbf{AB}$, submatrices $\mathbf{C}_{i'j'}$ should not be set to **0** initially.

- *General case (without overlap)*

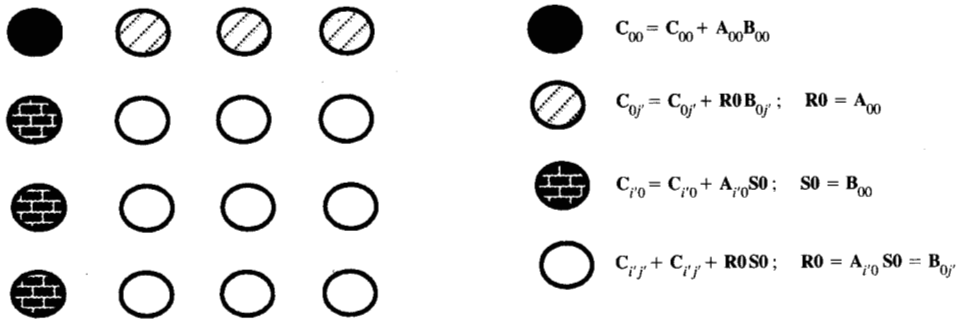
In the more general case, the processor array need not be square, and we need not set $k' = n'$. We still assume that M is an integral multiple of m' ($m = M/m'$), that N is an integral multiple of n' ($n = N/n'$), and that k and k' are integers ($k \times k' = K$). Then, k' may be chosen to be



Phase 1 — Broadcasting submatrices of **A**



Phase 2 — Broadcasting submatrices of **B**



Phase 3 — Calculating $C_{i'j'}$

Figure 3

Illustration of step 0 of the basic algorithm on a 4 × 4 array of processors.

larger than n' and m', and more than one submatrix of **A** and **B** may be mapped to processor $p(i', j')$, according to the following:

- Submatrix $\mathbf{A}_{i'\sigma'}$ is assigned to processor $p(i', \sigma' \bmod n')$. That is, the columns of **A** are divided among the columns of the processor array in blocks of k, in wraparound fashion; the rows of **A** are divided among the rows of the processor array, just as in the basic case.
- $\mathbf{B}_{\sigma' j'}$ is assigned to processor $p(\sigma' \bmod m', j')$.
- $\mathbf{C}_{i'j'}$ is assigned to processor $p(i', j')$, as before.

Note that each processor stores only one submatrix of **C**. On the other hand, there may be more than one submatrix of **A** and **B** assigned to a processor. The exact number of submatrices mapped to a processor is determined by i' and j' and on the relationship of k' to m' and n'.

For example, consider multiplying the matrices **A** and **B** described in the previous subsection (**A** is 1000×4000, and **B** is 4000×6400) on a 20×25 processor array of size 500. Then, $m' = 20$, $m = 50$, $n' = 25$, $n = 256$. If we choose $k' = 80$ (thus, $k = 50$), some of the processors will be assigned three submatrices of **A**, and some will be assigned four. All processors will be assigned four submatrices of **B**.

The k' steps of the algorithm are as before, each processor broadcasting its appropriate **A** and **B** submatrices during each step, receiving broadcast submatrices, and calculating $\mathbf{C}_{i'j'}$.

Finally, we may remove the restriction that M is an integral multiple of m', and the other similar constraints given above. The basic principle is the same. The details for handling the irregularities involved with matrices of arbitrary size are described in [16].

- *Memory requirements*

In order to describe how much memory is required in each processor, we introduce notation for quotients and remainders:

$$k' = q_m m' + r_m ,$$

$$k' = q_n n' + r_n .$$

Here q_m, r_m, q_n, and r_n are nonnegative integers ($r_m < m'$ and $r_n < n'$). Then, the memory requirement, in number of words, for accommodating submatrices of **A** and **B** at processor $p(i', j')$ is given by

$$\begin{array}{lll}
(mk)(q_n + 1) + (kn)(q_m + 1) & 0 \le i' < r_m , & 0 \le j' < r_n , \\
(mk)q_n + (kn)(q_m + 1) & 0 \le i' < r_m , & r_n \le j' < n' , \\
(mk)(q_n + 1) + (kn)q_m & r_m \le i' < m' , & 0 \le j' < r_n , \\
(mk)q_n + (kn)q_m & r_m \le i' < m' , & r_n \le j' < n' .
\end{array}$$

- *Algorithm with overlap*

In order to adapt our algorithm to overlap communication and computation, we use two additional buffers at each processor, $R1$ of size mk and $S1$ of size kn. This algorithm is similar to the one without overlap, except that during a basic step, a processor receives data in the buffer pair $R0$ and $S0$ (or in the pair $R1$ and $S1$) and performs computation using $R1$ and $S1$ (or $R0$ and $S0$). This is to ensure that the computation and communication can be done concurrently. If a processor uses $R0$ and $S0$ for receiving data in one step, it uses the matrices in these buffers for computing in the next step. The σ'th step is similar to the one outlined for the nonoverlap case, with the exception that for odd steps a processor uses $R0$ and $S0$ for receiving data and $R1$ and $S1$ for computation. The converse is true for the even steps.

Note that in all steps but one, we are able to overlap communication with computation.

We now discuss guidelines for selecting m, n, and k (consequently, m', n', and k'), the submatrix dimensions:

1. $m' \times n'$ should equal the number of processors available.
2. m, n, and k should be large enough that the uniprocessors executing DGEMM are computing in their range of nearly peak performance.
3. k should be as small as possible in order to minimize the communication cost. As k decreases, k' increases, resulting in the communication overhead of the first step (of k') being a smaller fraction of the total execution time.

In general, it is possible to satisfy all of these constraints when the problem size (M, N, K) is large enough. Otherwise, fewer processors than the number available may be used [see guideline 1 above].

Experimental results

We tested our implementation on a 128-processor Intel iPSC System 860 computer and a 512-processor Intel Touchstone Delta computer. Both computers use the same processor. The Intel iPSC System 860 is a distributed-memory parallel computer with a hypercube interconnection network. For our experiments, we chose a two-dimensional mesh; however, our algorithm is not restricted to a mesh or even a hypercube connectivity. The standard SGEMM BLAS supplied by the vendor was used on each processor. All experiments were done on square matrices, i.e., for $M = N = K$, with $k' = n'$ (the basic algorithm, with overlap).

Although one might expect that all processors executing the algorithm would require the same amount of time to execute each iteration step, some factors (e.g., system programs running concurrently) affect this synchronism.

Table 1 Performance results of matrix multiplication (single precision) executed on Intel iPSC 860 parallel computer.

Processor configuration	*Matrix order*	*m*	*n*	*k*	*Algorithm MFLOPS*	*Overlap fraction*	*MFLOPS per processor with overhead*	*MFLOPS per processor without overhead*
1 × 2	700	700	350	350	73	0.96	36.5	38.0
2 × 2	1000	500	500	500	157	0.94	39.3	41.8
2 × 4	1400	700	350	350	295	0.94	36.9	39.3
4 × 4	2000	500	500	500	607	0.95	37.9	39.9
4 × 8	2800	700	350	350	1132	0.94	35.4	37.7
8 × 8	4000	500	500	500	2382	0.94	37.2	39.6
8 × 16	5440	680	340	340	4356	0.93	34.0	36.7

As a result, we carried out experiments to determine whether the processors remained in synchronism during the execution of the algorithm. (The experiment involved terminating the program with a global synchronization step at the end of the last step of the computation.) We observed a very minor variation (<1%) in the timings, thereby verifying that our algorithm does indeed have the property that all the processors finish execution at approximately the same time. The implementation of our algorithm on these two platforms showed 1) uniform computation and communication load and 2) nearly perfect synchronization between any two steps of the computation. We first give our results on the Intel iPSC System 860.

- *Intel iPSC System 860 computer results*

We measured a performance of 4.36 GFLOPS, which is approximately 34.0 MFLOPS per processor for 5440 × 5440 matrices multiplied on an 8 × 16-processor array of size 128. The performance of SGEMM on a single processor was 36.7 MFLOPS. We were able to overlap 93% of the communication with computation (other overheads were negligible).

Table 1 illustrates the dependency of SGEMM performance on the number of processors and matrix size. The first column lists the processor configuration used. In the next four columns, we list the order of the matrix and the three submatrix dimensions. The column labeled *Algorithm MFLOPS* gives the computation rate (in MFLOPS) of the parallel algorithm as calculated by dividing the total number of FLOPs of the computation ($2N^3$) by the elapsed wall-clock time. In the last two columns, we list MFLOPS per processor with communication overhead (i.e., the column labeled *Algorithm MFLOPS* divided by the number of processors, which gives the actual MFLOPS measured) and MFLOPS per processor without communication overhead (the computation for the specified submatrix sizes executed on a single processor), respectively. The column labeled *Overlap fraction* (calculated by dividing the next-to-last column by the last column) represents the fraction of overhead, primarily communication, that was able to be overlapped by computation. The value is relatively constant (≈0.95), and the variations should not be considered significant. Table 1 shows, as expected, that as we increase the number of processors in the computer and the matrix size, the performance and overlap remain relatively constant.

The data for the last column of Tables 1 and 2 are taken from Figure 2, which also includes some additional points. Note that the experiment in Table 1 with the largest number of processors corresponds to a value of $n = 340$. Had we decreased the value of n to 270, Figure 2 indicates that the performance would have been approximately 43 MFLOPS. This would lead us to expect a result of 5.1 GFLOPS for a matrix of order 4320 ($m = 540$, $n = 270$, $k = 270$).

- *Intel Delta computer results*

We obtained a peak performance of 19.0 GFLOPS (37.1 MFLOPS per processor) on the Delta computer with 512 processors, for a 14 720 × 14 720 matrix. For this computation, we were able to overlap 96% of the computation with broadcast communication. All submatrices processed by SGEMM were of the same size. The single-processor SGEMM rate for 14 720 × 14 720 matrices was measured at 38.6 MFLOPS. We also measured the peak performance obtainable on the Delta computer using different processor configurations. These results are summarized in **Table 2**, which is similar to Table 1.

Note that the experiment in Table 2 with the largest number of processors corresponds to a value of $n = 460$. Had we decreased the value of n to 450, Figure 2 indicates that the performance would have been approximately 43 MFLOPS. This leads us to expect a result of 21.1 GFLOPS for a matrix of order 14 400 ($m = 900$, $n = 450$, $k = 450$).

Conclusion

In this paper we have proposed an algorithm for multiplying two matrices on a distributed-memory parallel

Table 2 Performance results of matrix multiplication (single precision) executed on Intel Touchstone Delta parallel computer.

Processor configuration	*Matrix order*	*m*	*n*	*k*	*Algorithm MFLOPS*	*Overlap fraction*	*MFLOPS per processor with overhead*	*MFLOPS per processor without overhead*
1 × 2	850	850	425	425	76	0.99	38.0	38.5
2 × 2	1300	650	650	650	159	0.99	39.9	40.3
2 × 4	1700	850	425	425	306	0.99	38.3	38.7
4 × 4	2600	650	650	650	624	0.98	39.0	39.8
4 × 8	3400	850	425	425	1193	0.97	37.3	38.5
8 × 8	5000	625	625	625	2444	0.97	38.2	39.4
8 × 16	6800	850	425	425	4644	0.96	36.3	37.8
16 × 16	10400	650	650	650	9673	0.96	37.8	39.4
16 × 32	14720	920	460	460	18977	0.96	37.1	38.6

computer. We have shown that the performance of the proposed algorithm is nearly equal to the performance of the optimized SGEMM BLAS times the number of the processors in the computer. Our algorithm consists of k' basic steps in which the communication and computation are overlapped, except during the first step. The impact of the nonoverlap of the first step can be reduced by increasing the number of steps k'. One has to be cautious in increasing k', however, so as not to decrease k $(= K/k')$ below the critical value at which DGEMM performance drops off sharply. A value of k lower than a critical value results in a lower DGEMM BLAS performance. This performance loss is significant and can offset the gain due to the increase in k'. We performed experiments on the Intel iPSC System 860 and Intel Touchstone Delta computers to see the effect of varying k. As expected, we observed a performance improvement for larger values of k.

Acknowledgment

We are grateful to the NASA Ames Research Center for providing access to the 128-processor Intel iPSC System 860 computer. The work of M. Zubair was supported by the Institute for Computer Applications in Science and Engineering (ICASE), NASA Langley Research Center, Hampton, VA, when he was in residence there.

References

1. J. J. Dongarra, J. DuCroz, I. S. Duff, and S. Hammarling, "Algorithms 679: A Set of Level 3 Basic Linear Algebra Subprograms," *ACM Trans. Math. Soft.* **16,** No. 1, 18–28 (1990).
2. R. C. Agarwal and F. G. Gustavson, "A Parallel Implementation of Matrix Multiplication and LU Factorization on the IBM 3090," *Proceedings of the IFIP WG 2.5 Working Conference on Aspects of Computation on Asynchronous Parallel Processors*, Palo Alto, CA, Elsevier/North-Holland, 1989, pp. 217—221.
3. K. Gallivan, W. Jalby, U. Meier, and A. Sameh, "Impact of Hierarchical Memory Systems on Linear Algebra Algorithms Design," *Intl. J. Supercomputer Appl.* **2,** 12–48 (1988).
4. *IBM Engineering and Scientific Subroutine Library, Guide and Reference*, Order No. SC23-0526, 1992; available through IBM branch offices.
5. E. Dekel, D. Nassimi, and S. Sahni, "Parallel Matrix and Graph Algorithms," *SIAM J. Comput.* **10,** No. 4, 657–673 (1981).
6. G. Fox, S. Otto, and A. Hey, "Matrix Algorithms on a Hypercube I: Matrix Multiplication," *Parallel Computing*, Elsevier/North-Holland, Amsterdam, 1987, pp. 17–31.
7. L. Johnsson and C. T. Ho, "Matrix Multiplication on Boolean Cubes Using Generic Communication Primitives," *Parallel Processing and Medium Scale Multiprocessors*, Society for Industrial and Applied Mathematics, 1989, pp. 108–156.
8. C. T. Ho, S. L. Johnsson, and A. Edelman, "Matrix Multiplication on Hypercubes Using Full Bandwidth and Constant Storage," *Sixth Distributed Memory Computing Conference Proceedings*, IEEE Computer Society Press, New York, 1991, pp. 447–451.
9. L. Johnsson and C. T. Ho, "Algorithms for Multiplying Matrices of Arbitrary Shapes Using Shared-Memory Primitives on Boolean Cubes," *Technical Report YALEU/DCS/TR-569*, Department of Computer Science, Yale University, New Haven, CT, 1987.
10. C. Lin and L. Snyder, "A Matrix Product Algorithm and Its Comparative Performance on Hypercubes," *Proceedings of Scalable High-Performance Computer Conference (SHPCC-92)*, IEEE, Williamsburg, VA, 1992, pp. 190–194.
11. J. Choi, J. J. Dongarra, and D. W. Walker, "PUMMA: Parallel Universal Matrix Multiplication Algorithms on Distributed Memory Concurrent Computers," *Concurrency: Pract. & Exper.* **6,** No. 7, 543–570 (1994); also in Technical Reports of Oak Ridge National Laboratory, Mathematical Sciences Section, TM-12252, August 1993.
12. J. W. Demmel, M. T. Heath, and H. A. Vorst, "Parallel Numerical Linear Algebra," *LAPACK Working Note 60*, University of Tennessee, Knoxville, TN, CS-93-192, 1993.
13. K. A. Gallivan, R. J. Plemmons, and A. H. Sameh, "Parallel Algorithms for Dense Linear Algebra Computations," *SIAM Rev.* **32,** No. 1, 54–135 (1990).
14. S. Huss-Lederman, E. Jacobson, and A. Tsao, "Comparison of Scalable Parallel Matrix Multiplication

Libraries," *Proceedings of the Scalable Parallel Libraries Conference*, Starksville, MI, October 1993, pp. 142–149.
15. S. Huss-Lederman, E. Jacobson, A. Tsao, and G. Zhang, "Matrix Multiplication on the Intel Touchstone DELTA," *Technical Report SRC-TR-93-101*, Supercomputing Research Center, Bowie, MD, February 1994.
16. F. G. Gustavson and M. Zubair, "A High Performance Multiplication Algorithm on a Distributed-Memory Parallel Machine Using Overlapped Communication," *Research Report RC-18694 (81769)*, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, 1993.

Received August 2, 1993; accepted for publication March 18, 1994

Ramesh C. Agarwal *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (AGARWAL at YKTVMV, agarwal@watson.ibm.com).* Dr. Agarwal received a B.Tech. (Hons.) degree from the Indian Institute of Technology (IIT), Bombay. He was the recipient of The President of India Gold Medal while there. He received M.S. and Ph.D. degrees from Rice University and was awarded the Sigma Xi Award for best Ph.D. thesis in electrical engineering. He has been a member of the Mathematical Sciences Department at the IBM Thomas J. Watson Research Center since 1983. Dr. Agarwal has done research in many areas of engineering, science, and mathematics and has published over 60 papers in various journals. Currently, his primary research interest is in the area of algorithms and architecture for high-performance computing on workstations and scalable parallel machines. In 1974, Dr. Agarwal received the Senior Award from the IEEE Acoustics, Speech, and Signal Processing (ASSP) group, for best papers. He has received several Outstanding Achievement Awards and a Corporate Award from IBM. Dr. Agarwal is a Fellow of the IEEE and a member of the IBM Academy of Technology.

Fred G. Gustavson *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (GUSTAV at YKTVMV, gustav@watson.ibm.com).* Dr. Gustavson is manager of Algorithms and Architectures in the Mathematical Sciences Department at the IBM Thomas J. Watson Research Center. He received his B.S. in physics, and his M.S. and Ph.D. degrees in applied mathematics, all from Rensselaer Polytechnic Institute. He joined IBM Research in 1963. One of his primary interests has been in developing theory and programming techniques for exploiting the sparseness inherent in large systems of linear equations. Dr. Gustavson has worked in the areas of nonlinear differential equations, linear algebra, symbolic computation, computer-aided design of networks, design and analysis of algorithms, and programming applications. He and his group are currently engaged in activities that are aimed at exploiting the novel features of the IBM family of RISC processors. These include hardware design for divide and square root, new algorithms for POWER2™ for the Engineering and Scientific Subroutine Library (ESSL) and for other math kernels, and parallel algorithms for distributed memory processors. Dr. Gustavson has received an IBM Outstanding Contribution Award, an IBM Outstanding Innovation Award, an IBM Outstanding Invention Award, two IBM Outstanding Technical Achievement Awards, two IBM Corporate Technical Recognition Awards, and a Research Division Technical Group Award.

Mohammad Zubair *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (ZUBAIR at YKTVMV, zubair@watson.ibm.com).* Dr. Zubair received his Ph.D. degree in 1987 from the Indian Institute of Technology (IIT), New Delhi. From 1981 to 1987, he was at the Center for Applied Research in Electronics, IIT Delhi. In 1987, he became an Assistant Professor at Old Dominion University, Norfolk, VA, and in 1993 he became an Associate Professor. He joined IBM Research in 1994. Dr. Zubair's primary research interest is in the algorithm and architecture aspects of large-scale scientific computing. He has published more than 30 papers in various journals and conference proceedings.