Multiplication of Matrices of Arbitrary Shape on a Data Parallel Computer

Multiplication of Matrices of Arbitrary Shape on a Data Parallel Computer

Kapil K. Mathur
S. Lennart Johnsson

TR-01-92

January 1992

Parallel Computing Research Group

Center for Research in Computing Technology
Harvard University
Cambridge, Massachusetts

Multiplication of Matrices of Arbitrary Shape on a Data Parallel Computer

Kapil K. Mathur and S. Lennart Johnsson[1]

Thinking Machines Corp.
245 First Street
Cambridge, MA 02142

Abstract

Some level–2 and level–3 Distributed Basic Linear Algebra Subroutines (DBLAS) that have been implemented on the Connection Machine system CM–200 are described. No assumption is made on the shape or size of the operands. For matrix–matrix multiplication, both the nonsystolic and the systolic algorithms are outlined. A systolic algorithm that computes the product matrix in–place is described in detail. We show that a level–3 DBLAS yields better performance than a level–2 DBLAS. On the Connection Machine system CM–200, blocking yields a performance improvement by a factor of up to three over level–2 DBLAS. For certain matrix shapes the systolic algorithms offer both improved performance and significantly reduced temporary storage requirements compared to the nonsystolic block algorithms.

We show that, in order to minimize the communication time, an algorithm that leaves the largest operand matrix stationary should be chosen for matrix–matrix multiplication. Furthermore, it is shown both analytically and experimentally that the optimum shape of the processor array yields square stationary submatrices in each processor, i.e., the ratio between the length of the axes of the processing array must be the same as the ratio between the corresponding axes of the stationary matrix. The optimum processor array shape may yield a factor of five performance enhancement for the multiplication of square matrices. For rectangular matrices a factor of 30 improvement was observed for an optimum processor array shape compared to a poorly chosen processor array shape.

[1]Also affiliated with the Division of Applied Sciences, Harvard University, Cambridge, MA 02138.

1 Introduction.

This article describes the algorithms used for matrix–vector, vector–matrix multiplication, rank–1 updates, and matrix-matrix multiplication on matrices distributed across the memory of the Connection Machine system CM–200. This system has up to 2048 floating–point processors that support operations in both 32–bit and 64–bit precision. The memory is distributed among the processing units, with a maximum of 4 Mbytes of memory per unit and a total memory of 8 Gbytes. Each processing unit has a single 32-bit wide data path to its memory. Data paths internal to the floating–point unit are 64–bits wide. The processing units are interconnected as an 11–dimensional Boolean cube, with two channels between every pair of nodes. Data may be exchanged on all 22 (11×2) channels concurrently. For some phases of the algorithms described below this property is explored.

Throughout this article, the axis enumerating the rows is referred to as the *row axis*, and the axis enumerating the columns is referred to as the *column axis*. For a two–dimensional data array $A(i, j)$, the left index refers to the row axis, and the right index refers to the column axis. A *processing node*, or simply node, refers to a processor with associated local memory and communications facilities. Throughout this presentation, it is assumed that there are N nodes, which for a two–dimensional processor configuration, have N_r nodes along the row axis, and N_c nodes along the column axis ($N = N_r \times N_c$).

The algorithms described here have no restriction on the shape of the matrices or the number of processing nodes, other than that each operand is assumed to have at least one element assigned to each node. The processors may be configured as a one–dimensional array or a two–dimensional array of arbitrary shape. The algorithms presented here achieve perfect arithmetic load balance. For operand matrices assigned to a subset of processors, load balancing is an important issue. Algorithms that address load balancing issues in greater detail are discussed in [17].

The algorithms presented here are data parallel adaptations of the standard matrix multiplication algorithm requiring $2PQR$ arithmetic operations for the multiplication of a $P \times Q$ matrix by a $Q \times R$ matrix. The index space for these operations is depicted in Figure 1. $R = 1$ corresponds to matrix–vector multiplication, $P = 1$ to vector–matrix multiplication, and $Q = 1$ to rank–1 updates. The algorithms described here provide schedules for the operations in space and time that maintain perfect load balance both with respect to communication and computation whenever there is one data element per processor.

The data motion requirements and the performance depend strongly on the data allocation of the operands. The Connection Machine compilers support a global address space and allocate arrays based on their shapes. The processors are configured for each array such that the rank of the data array and the processor array are the same. The ordering of the axes is also the same. When there are more matrix elements than processors, consecutive elements along each data array axis (a block) are assigned to a processor.

The outline of this paper is as follows. The next section discusses data allocation issues,

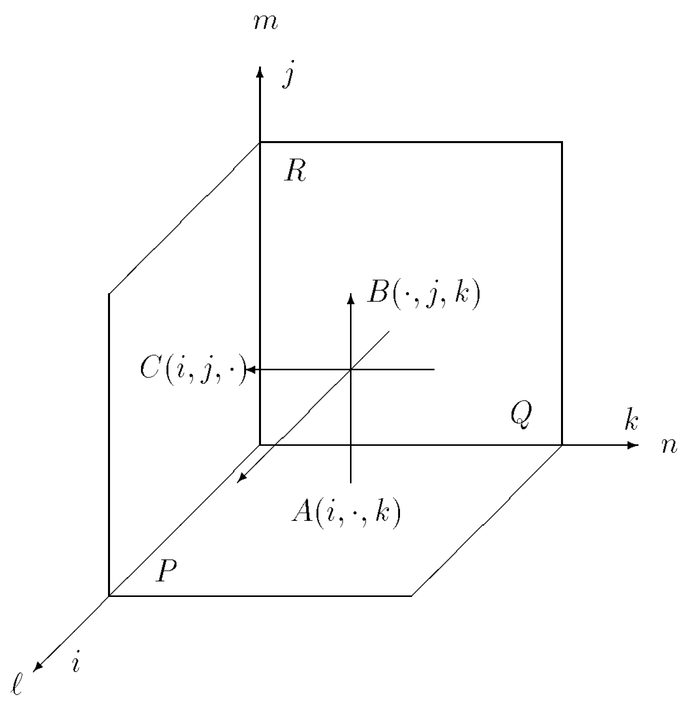

Figure 1: The index space for matrix multiplication.

and their consequences for the multiplication of matrices of arbitrary shapes. Matrix–vector, vector–matrix multiplication, and rank–1 updates on distributed data structures are discussed in Section 2. Algorithms for matrix-matrix multiplication are described in Section 4. A systolic algorithm that keeps the matrix C stationary is described in detail. The optimum shapes of the processor array for the different algorithms are also discussed in this section. Performance data for the Connection Machine system CM–200 are given in Section 5.

2 Data allocation.

All Connection Machine system compilers support a global address space and allocate data evenly among the memory modules. Multiple elements are assigned to the same memory module based on the *consecutive* (also known as block) data allocation scheme [7, 12].

For multidimensional arrays, the default mode of the Connection Machine compilers configures the processors such that the average number of local references for each remote reference is maximized when the data references are equally frequent along all array axes [26]. The area of the faces for a subarray of a given size is minimized. The user can control the shape of the processor array, that is, the data layout, through compiler directives. An axis can be forced to be local to a memory module by the directive SERIAL, if there is sufficient local memory or if the length of the local axis segment is changed by assigning weights to the axes. High weights are used for axes with frequent references and low weights for axes with infrequent references. A relatively high weight for an axis increases the length of the local segment of that axis at the expense of the length of the segments of the other axes. The total size of the subarray is independent of the assignment of weights. Only the shape of the subarray changes.

As an example of the data allocations possible under the model used by the Connection Machine compilers, consider a 64×128 array assigned to 64 processors. Each processor receives 128 elements. With the processors configured as an 8×8 array, each processor receives a subarray of 8×16 elements. In the consecutive allocation, the elements of the subarray are selected as successive elements along the axes. With the processors configured as a one-dimensional array, each processor is assigned a subarray that is either of shape 1×128, or of shape 64×2, depending upon the orientation of the processor array with respect to the data array. The shape of the processor array clearly does not affect the number of arithmetic operations to be performed, but it may affect both the communication requirements and the efficiency in utilizing the arithmetic units, for example, by changing the vector lengths.

The consecutive allocation scheme selects elements to be assigned to the same processor. Compiler directives, such as axis weights and SERIAL, address the issue of choosing the processor array shape. Another data layout issue is the assignment of data sets made up of consecutive elements along the different data array axis to processors. The network topology and the data reference pattern are two important characteristics in this assignment. In a mesh-connected machine it is natural to assign subarrays of one and two–dimensional data arrays to processors such that the adjacency in the data array is preserved when mapped to the processor array. Preserving adjacency in mapping data arrays of at least three dimensions to two–dimensional processor arrays is impossible [24].

The processors in the Connection Machine system, CM–200, are interconnected as a Boolean cube with two channels between each pair of processors. Meshes of any dimension up to the dimension of the Boolean cube are subgraphs thereof. A Boolean cube network of n dimensions has 2^n nodes. The number of dimensions in Connection Machine system CM-200 ranges from 7 to 11 depending upon system size (the largest system having 16 times more processors than the smallest system). The nodes of a Boolean cube can be given addresses such that adjacent nodes differ in precisely one bit in their binary encoding. Assigning subarrays to processors using the standard binary encoding of the subarray index along an axis, i.e., the *node address*, does not preserve adjacency along an axis. For instance, 3 and 4 differ in all three bits in the encoding of the addresses of 8 processors, and are at a distance of three apart. In general, $2^{n-1} - 1$ and 2^{n-1} differ in n bits in their binary encoding and are at a distance of n. The number of bits in which two indices differ translates directly into distance in a Boolean cube architecture.

Binary–reflected Gray codes [23] generate embeddings of arrays into Boolean cube networks that preserve adjacency [12]. Gray codes have the property that the encoding of successive integers differ in precisely one bit. In a Boolean cube network successive indices are assigned to adjacent processors. The binary–reflected Gray code is efficient both in preserving adjacency, and in processor utilization, when the length of the axes of the data array is a power of two [9]. For arbitrary data array axes' length, the Gray code may be combined with other techniques to generate efficient embeddings [3, 10]. Matrix multiplication algorithms can be formulated for one, two [2, 8, 12], and three–dimensional

[13] meshes, and Boolean cubes [4, 11, 28].

3 Level–2 Distributed BLAS (DBLAS).

All level–2 BLAS [19, 5, 6] involve operations on arrays of different shape. Functions, such as matrix–vector multiplication, vector–matrix multiplication, and rank–1 updates, involve both vectors and matrices. With data array allocation based on the shape of the array, an *alignment* of the operand arrays with respect to each other is necessary before the operations can be carried out and the result stored as required. In this presentation, BLAS operating on distributed data structures are referred to as Distributed BLAS (DBLAS). The data motion issues for some level–2 DBLAS are discussed briefly below. For a more extensive discussion the reader is referred to [17].

3.1 Matrix–vector and vector–matrix multiplication.

The evaluation of the matrix-vector product $y \leftarrow Ax$ requires the operations:

1. Aligning the vector x with the column axis of A.
2. Spreading the vector x along the row axis of A, such that there is one copy of the appropriate segment of x for every node.
3. Performing a matrix–vector multiplication concurrently on each node.
4. Performing a reduction along the column axis to form y, aligned with the row axis of A.
5. Aligning y with its original allocation.

In Step 1, the vector x is placed along a processor row of A, such that the range of inner indices (corresponding to the Q-axis) in a node is identical for A and x. Step 2 replicates x such that every node has a segment of the vector x that corresponds to the inner indices of the rows of A. Step 3 defines N concurrent matrix–vector products followed in Step 4 by N_r concurrent summations of N_c vectors of length $\frac{P}{N_r}$.

The Connection Machine system CM–200 implementation is based on the processing node description. Local level–2 BLAS [18] are used for the operations in each node. Vector–matrix multiplication is treated similarly. In [17] it is shown that the data motion for the alignment of the vectors with the matrix constitutes a shuffle on a suitably defined index set. Optimal implementations on mesh-connected networks is described in [22] and on Boolean cube networks in [16]. The communication efficiency can be improved further by combining the alignment and spread operations (Steps 1 and 2), and the reduction and alignment operations (Steps 4 and 5) [17].

3.2 Rank–1 updates.

The evaluation of the rank–1 update $A \leftarrow xy^T$ requires the operations:

1. Aligning the vector x with the row axis of A.
2. Aligning the vector y with the column axis of A.
3. Spreading the vector x along the column axis of A, such that there is one copy of the appropriate segment of x for every node.
4. Spreading the vector y along the row axis of A, such that there is one copy of the appropriate segment of y for every node.
5. Performing a rank–1 update on each node.

The above description is made with respect to the processing array. A similar description in terms of the data array is also possible.

The Connection Machine system CM–200 implementation uses the processing array description and local level–2 BLAS for the operations on each node. As with matrix–vector and vector–matrix multiplication, the alignment can be combined with the spread (Steps 1 and 3 and Steps 2 and 4).

4 Matrix–matrix multiplication (level–3 DBLAS).

Matrix–matrix multiplication is a part of level–3 BLAS. For distributed data structures it can be constructed out of the level–2 DBLAS described above. In [17] it is shown that for many distributed memory architectures, including the Connection Machine system CM–200, level–3 DBLAS is required for high efficiency with respect to data motion. A level–3 DBLAS also allows for the use of local level–3 BLAS, which may enhance the arithmetic efficiency. This section generalizes the matrix–vector and vector–matrix level–2 DBLAS to level–3 DBLAS. The systolic versions of these algorithms are then outlined. One important advantage of the systolic algorithms is that unlike the straightforward generalizations of the level–2 DBLAS, the systolic algorithms preserve the memory requirements.

4.1 Matrix multiplication based on level–2 BLAS functions.

The algorithm for matrix–vector multiplication described in the previous section can be extended for the matrix–matrix multiplication, $C \leftarrow A \times B + D$. By extracting one column of B (and D) and performing a matrix–vector multiplication, one column of the product matrix C is generated. This column must be deposited into the matrix C. The extraction of the column of B (and D) implies a change of layout, if the default data allocation strategy of the Connection Machine compilers is used: B (D) is a two-dimensional object, and the extracted column is one–dimensional. Similarly, the column of C obtained by a matrix–vector multiplication is one–dimensional, but C is a two–dimensional object. A detailed description of the required data motion can be found in [17].

Using a matrix–vector multiplication algorithm from a level–2 DBLAS package requires that the matrix B be transposed, aligned and spread along the row axis one column at a time. R calls to the matrix–vector multiplication routine are required. After each such call, the computed product vector must be aligned and deposited in the appropriate column of C. The arithmetic operations local to a node can be based on level–1 or level–2 BLAS, but not level–3 BLAS.

The number of calls to the level–2 DBLAS can be reduced by blocking the columns of B (and C). For blocks consisting of b columns, blocking reduces the overhead for both arithmetic and data motion by a factor of b. The arithmetic efficiency may also be improved by the use of level–3 BLAS in each node. The communication efficiency also increases because of improved load balance in the communication system [17].

With a blocking of b columns, each processing node requires temporary storage corresponding to $\frac{Q}{N_c}b$ data elements for the b columns of B. When $b = R$, that is, the entire matrix B is transposed and aligned with A, then the temporary storage requirements, due to the blocking, increase by a factor of R compared to the unblocked algorithm. Such an increase in the temporary storage requirement is often unacceptable. The increased demand for the temporary storage is due to the spread operation (Step 2 of the matrix–vector multiplication algorithm). Its function is to assure that each column of B can interact with every row of A with no motion of A. By modifying the implementation of the spread such that it is performed in a stepwise manner, the memory requirements can be preserved. For example, if there is at least one column of B allocated to each of the N_r processor rows after the extraction and alignment of $b > N_r$ columns, then the spread function realizes an *all–to–all broadcast* [8, 14, 25], which can be implemented as $N_r - 1$ cyclic shifts. The all–to–all broadcast can utilize the full communications capacity of Boolean cube networks [1, 14]. The reduction operation required in computing C can be expressed similarly.

The level–3 DBLAS matrix–matrix multiplication routine based on extraction of b columns of B, transposition and alignment, all–to–all broadcasting, all–to–all reduction and alignment using a memory preserving all–to–all implementation will be referred to as a systolic matrix–matrix multiplication algorithm with A stationary. Analogous algorithms for matrix–matrix multiplication can be defined using either a straightforward generalization of the vector–matrix multiplication algorithms, or the rank–1 update algorithms, or systolic versions thereof. In a vector–matrix type algorithm, B is stationary, while in a rank–1 update algorithm, C is stationary.

The arithmetic requirements for the three algorithms is the same. There may be differences in the arithmetic efficiency due to the different shapes of the submatrices of the different operands assigned to each processing node. For the systolic matrix–matrix multiplication algorithm with A stationary and $b = R$, the transposition and alignment of B can be performed as one operation. On a Boolean cube network, the optimal implementation of this operation requires a time proportional to $\frac{QR}{2N}$. The communication time is independent of the shape of the processing array [16]. The all-to-all broadcast operation for B requires a time proportional to $\frac{QR}{N \log_2 N_r}(N_r - 1) \approx \frac{QR}{N_c \log_2 N_r}$ [14]. Similarly, the reduction for C requires a time proportional to $\frac{PR}{N_r \log_2 N_c}$ [14]. Finally, the alignment of C requires a time proportional to $\frac{PR}{2N}$. Therefore, the optimal configuration of the processing array satisfies $\frac{P}{Q} = \frac{N_r \log_2 N_c}{N_c \log_2 N_r}$, or $N_r \approx \sqrt{\frac{NP}{Q}}$ and $N_c \approx \sqrt{\frac{NQ}{P}}$ for $P \approx Q$.

The optimal shape of the processing array is approximately congruent to the shape of the stationary matrix A. Similarly, the optimum processing array shape of the systolic algorithm with B stationary is congruent to B, and the optimum processing array shape of the array with C stationary is congruent to C.

In the Connection Machine Scientific Software Library (CMSSL) [27] nonsystolic algorithms are used for the cases with A and B stationary, while either a nonsystolic or systolic algorithm is used for the case with C stationary. The performance depends strongly upon the shape of the operands. In [17], it is shown that as a first approximation, the optimal algorithm keeps the matrix with the largest number of elements stationary.

Next, the systolic matrix–matrix multiplication algorithm with C stationary that is used in CMSSL is presented in detail.

4.2 Systolic matrix multiplication with C stationary.

4.2.1 Square processor arrays, $N_c = N_r$.

This algorithm assumes that each of the three operands has at least one element assigned to each processor. The processors are configured as a two–dimensional array. A binary–reflected Gray code encoding is used for each axis, such that adjacency in the data array is preserved in the distributed memory organized as a Boolean cube. A consecutive data allocation scheme is assumed. This section describes a block algorithm for square processor arrays. The next section generalizes the algorithm for rectangular processor arrays. Blocking reduces the number of local memory moves and allows for the use of level–3 BLAS on each node.

With the product matrix C stationary, for each element $c_{ij} \leftarrow \sum_{k=0}^{Q-1} a_{ik} b_{kj}$ of C, the corresponding elements of A, a_{ik}, and B, b_{kj}, for $k \in \{0, 1, 2, \ldots, Q-1\}$, must be moved to the processing node where c_{ij} resides, for all $i \in \{0, 1, 2, \ldots, P-1\}$ and $j \in \{0, 1, 2, \ldots, R-1\}$. The set of processing nodes to which row i of C is assigned must all receive every element of row i of A. Similarly, the set of processing nodes to which column j of C is assigned must all receive every element of column j of B. In the index space, this is an all–to–all broadcast [14] within the rows of A and the columns of B. It is assumed that matrix rows are aligned with processor rows and matrix columns with processor columns. This assumption is consistent with the compiler generated data layout.

Recall that the nonsystolic rank–1 update algorithm realizes the all–to–all broadcast as a sequence of spreads. The temporary storage requirements for the blocked version of this algorithm increase by a factor of b compared to the nonblocked algorithm. To reduce the temporary storage, the all–to–all broadcast operation is performed stepwise through cyclic shifts [21]. This introduces an alignment requirement between A and B. In evaluating the expression $c_{ij} \leftarrow \sum_{k=0}^{Q-1} a_{ik}b_{kj}$, only processors where the inner index k is identical for both A and B can participate in the computation. For a square processing array, the inner indices initially are the same for A and B only in the processors on the diagonal of the array. This property is true for matrices of any shape and any square processor array[2]. To increase the processor utilization, the matrices must be "aligned" such that the inner indices are identical in all nodes. A transposition of B or A would clearly align the inner indices, but partial products for C would then have to be accumulated in space. The following algorithm aligns A and B such that all processors can participate in the evaluation of C, without any data motion for C.

Let $0 \leq i < P$, $0 \leq k < Q$, and $0 \leq j < R$ denote matrix element indices, and $0 \leq \ell < N_r$ and $0 \leq m < N_c$ denote indices for the processor array elements. Since $N_r = N_c$, the partitioning of the inner axis Q is the same for A and B. Assume that $P = \alpha N_r$, $Q = \beta N_r$, and $R = \gamma N_c$. An alignment such that processor (ℓ, m) is assigned matrix elements

A: $(\alpha\ell + \phi, \beta(\ell + m) \bmod Q + \chi)$, where $0 \leq \phi < \alpha$, $0 \leq \chi < \beta$
B: $(\beta(\ell + m) \bmod Q + \chi, \gamma m + \psi)$, where $0 \leq \chi < \beta$, and $0 \leq \psi < \gamma$
C: $(\alpha\ell + \phi, \gamma m + \psi)$

ensures that the range of inner indices for A and B are identical on each processor. This property is true for arbitrary values of α and γ. Moreover, the range of indices for the P axis is the same for A and C, and the range of indices for the R axis is the same for B and C in each processor.

The stepwise all–to–all broadcast operation can be performed by using cyclic shifts. The data motion for the multiplication of A with B at each step may be expressed as

A: $(\alpha\ell + \phi, \beta(\ell + m) \bmod Q + \chi) \leftarrow (\alpha\ell + \phi, \beta(\ell + m + 1) \bmod Q + \chi)$, where $0 \leq \phi < \alpha$, $0 \leq \chi < \beta$
B: $(\beta(\ell + m) \bmod Q + \chi, \gamma m + \psi) \leftarrow (\beta(\ell + m + 1) \bmod Q + \chi, \gamma m + \psi)$, where $0 \leq \chi < \beta$, and $0 \leq \psi < \gamma$.

The shift operation must be repeated $N_r - 1 = N_c - 1$ times. Clearly, the inner indices of the two matrices are identical for each step of the algorithm. The correctness of the algorithm follows. After the alignment, and after each cyclic shift, matrices of shape $\alpha \times \beta$ and $\beta \times \gamma$ are multiplied on each processor. In the Connection Machine system CM–200 implementation described here, level–2 BLAS are used for the local matrix multiplication.

For $P = N_r$ and $R = N_c$, the algorithm described above degenerates to the algorithm in [2]. For certain high degree networks, such as Boolean cubes, multiple exchange sequences can be used to make effective use of the communications bandwidth [11].

Remark 1. No local data motion is required between the cyclic shifts moving data between processors. Emulating a large virtual processing array naively on the physical

[2]Note, however, that if the layout rule is to minimize the surface area for a given subarray, then for a rectangular matrix the processors will not be configured as a square array.

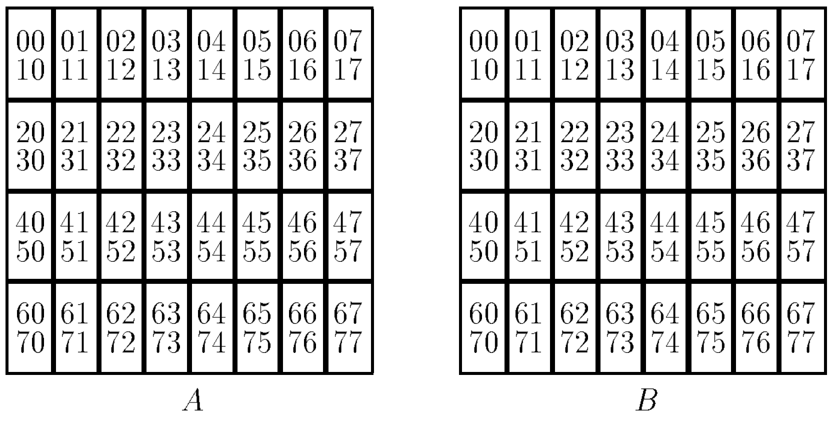

Figure 2: Allocation of 8×8 matrices to a 4×8 processor array.

array of shape $N_r \times N_c$ would result in excessive local data motion. The data motion between the processing nodes would be the same.

Remark 2. With the positive axis direction coinciding with increasing column indices and decreasing row indices, A is shifted in the negative direction and B in the positive direction. Shifting A in the positive axis direction and B in the negative direction also yields a valid algorithm. Further, the submatrices for A and B can be split into two parts, such that different parts are shifted in different directions. This observation is useful on architectures where the primitive communication operation is an exchange, which is the case for the Connection Machine system CM–200. Moreover, the data motion of A and B can be performed concurrently.

Remark 3. The correctness of the above algorithm relies on the range of the inner indices being identical for A and B. If $N_r \neq N_c$, this property is not true. This restriction is relaxed in the next section.

4.2.2 Rectangular processor arrays, $N_r \neq N_c$.

Figure 2 shows the allocation of square 8×8 matrices to 32 processors configured as a 4×8 array. The length of the segment of the inner axis assigned to a processor is *different* for A and B. Figure 3 shows the result of an alignment and the first two steps of the multiplication phase. For the example, in Figures 2 and 3, the all–to–all broadcast of the multiplication phase requires 8 cyclic rotation steps for A and 4 steps for B, since there are 8 processor columns and 4 processor rows. Figure 3 shows the locations of elements after the first and second cyclic shift of A and the first shift of B. After the alignment, *all* elements of A and half of the elements of B participate in the local multiplication. After the first cyclic shift of A, all its elements are again participating in local matrix multiplications, with the *previously unused* elements of B. After the second cyclic shift of A and the first cyclic shift of B, all elements of A and the "first" half of the elements of B are used in the same way as after the alignment. After this sequence is repeated four times, the matrix C is computed.

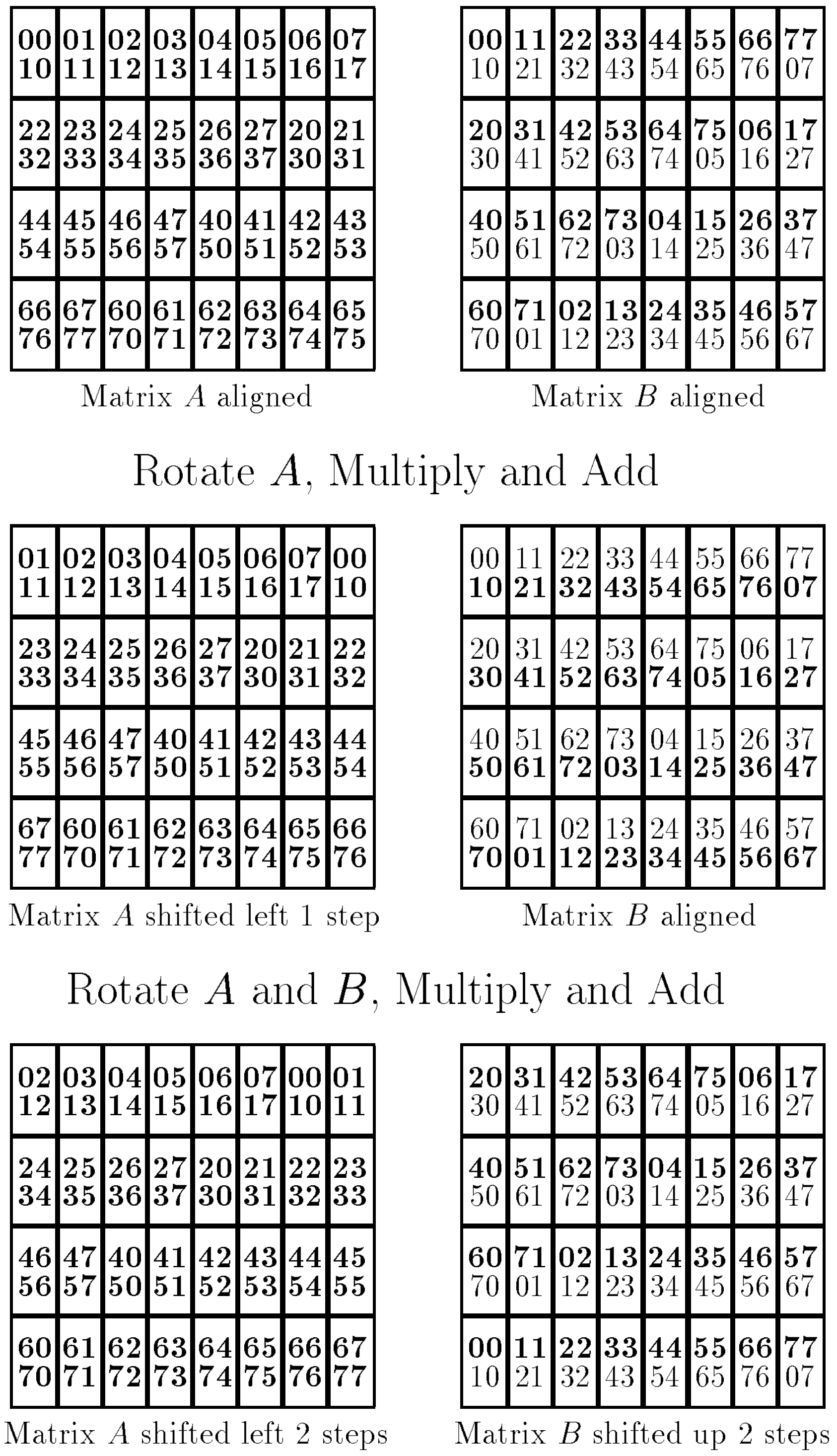
Matrix A aligned
Matrix B aligned
Rotate A, Multiply and Add
Matrix A shifted left 1 step
Matrix B aligned
Rotate A and B, Multiply and Add
Matrix A shifted left 2 steps
Matrix B shifted up 2 steps

Figure 3: Matrix multiplication on a 4×8 array.

In Figure 3 all operations requiring a given submatrix are carried out before the entire submatrix is moved to the adjacent processor. No local data motion is required. When the inner axis extent per processor is different for A and B, which is the case for a rectangular processor array, then only a fraction of the local submatrix with the largest inner axis segment is used in a local matrix multiplication for each rotation step of the submatrix with the shortest inner axis segment. The submatrices are fully used for each rotation step in which it participates. If the number of processors assigned to one axis is a multiple of the number of processors along the other axis, for example, $N_c > N_r$ as in Figure 3, then $\frac{N_c}{N_r}$ rotation steps are performed along the longer axis for every rotation step along the shorter axis. A more general case is shown in Figures 4 and 5.

Let $P, Q \geq N_r$ and $Q, R \geq N_c$, and $N = N_r \times N_c$, with no other restriction on N_r and N_c, and let $\beta_c = \frac{Q}{N_c}$ and $\beta_r = \frac{Q}{N_r}$. For arbitrary values of N_r and N_c, define a square *virtual* processor array of shape $\frac{N_r N_c}{\gcd(N_r,N_c)} \times \frac{N_r N_c}{\gcd(N_r,N_c)}$. Let ℓ^v, m^v identify a block in the virtual array: $(\ell^v, m^v) \in \{0, 1, \ldots, \frac{N_r N_c}{\gcd(N_r,N_c)} - 1\} \times \{0, 1, \ldots, \frac{N_r N_c}{\gcd(N_r,N_c)} - 1\}$. Let $\beta^v = \lceil \frac{Q gcd(N_r,N_c)}{N_r N_c} \rceil$. After the alignment of the operands with respect to each other and the establishment of a shared processor array shape, the index assignment for physical processor (ℓ, m) is:

A: $(\alpha\ell + \phi, \beta^v((m\frac{N_r}{\gcd(N_r,N_c)} + \ell\frac{N_c}{\gcd(N_r,N_c)}) \bmod \frac{N_r N_c}{\gcd(N_r,N_c)}) + \chi) = (\alpha\ell + \phi, (\beta_c m + \beta_r \ell) \bmod Q + \chi)$, where $0 \leq \phi < \alpha$ and $0 \leq \chi < \beta_c$.

B: $(\beta^v((\ell\frac{N_c}{\gcd(N_r,N_c)} + m\frac{N_r}{\gcd(N_r,N_c)}) \bmod \frac{N_r N_c}{\gcd(N_r,N_c)}) + \chi, \gamma m + \psi) = ((\beta_r \ell + \beta_c m) \bmod Q + \chi, \gamma m + \psi)$, where $0 \leq \chi < \beta_r$ and $0 \leq \psi < \gamma$.

C: $(\alpha\ell + \phi, \gamma m + \psi)$.

Note that after the alignment, all arrays are allocated to the processors assuming the same processor array configuration.

Clearly, for every processor, the smallest inner index for A and B is identical. But, the range of inner indices, χ, is different for A and B. The number of identical indices is $\min(\beta_r, \beta_c)$. The number of local blocks along the inner axis of A is $\frac{N_r}{\gcd(N_r,N_c)}$, and of B is $\frac{N_c}{\gcd(N_r,N_c)}$.

For the multiplication phase, N_c shifts are required for A and N_r shifts for B. When $N_c > N_r$, $\lfloor \frac{N_c}{N_r} \rfloor$ shifts of A are performed without any shift of B. For each such shift, a rank-β_c update is performed concurrently on all nodes, consuming the entire submatrix of A on a node. For each shift the indices of A on a node changes as: $(\alpha\ell + \phi, (\beta_c m + \beta_r \ell) \bmod Q + \chi) \leftarrow (\alpha\ell + \phi, (\beta_c(m+1) + \beta_r \ell) \bmod Q + \chi)$. The next shift of A, shift $\lfloor \frac{N_c}{N_r} \rfloor + 1$, brings in the indices of A that correspond to the remaining inner indices of B (if N_c is not a multiple of N_r) and some additional indices. A rank-mod$\frac{\beta_r}{\beta_c}$ update is performed first to consume all inner indices of B, followed by a move of B, and a rank-($\beta_c -$ mod$\frac{\beta_r}{\beta_c}$) before A is moved again. The cyclic shift of B brings about the index change: $((\beta_r \ell + \beta_c m) \bmod Q + \chi, \gamma m + \psi) \leftarrow ((\beta_r(\ell+1) + \beta_c m) \bmod Q + \chi, \gamma m + \psi)$. If $N_r > N_c$, then B is moved more often, and a similar analysis applies. The index sets for the blocks each processor receives are monotonically increasing, contiguous and periodic.

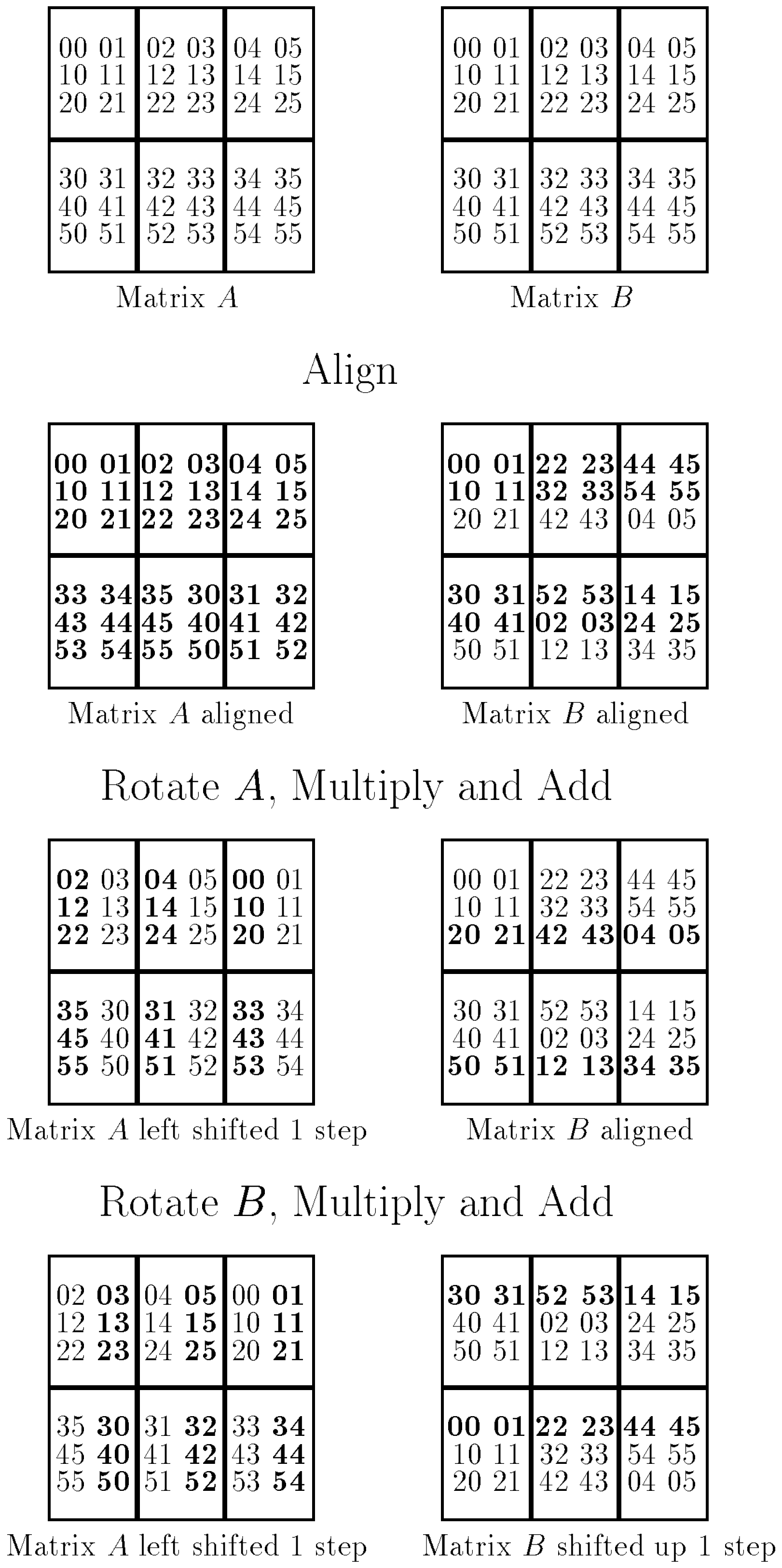
Matrix A
Matrix B
Align
Matrix A aligned
Matrix B aligned
Rotate A, Multiply and Add
Matrix A left shifted 1 step
Matrix B aligned
Rotate B, Multiply and Add
Matrix A left shifted 1 step
Matrix B shifted up 1 step

Figure 4: Matrix multiplication on a 2×3 array.

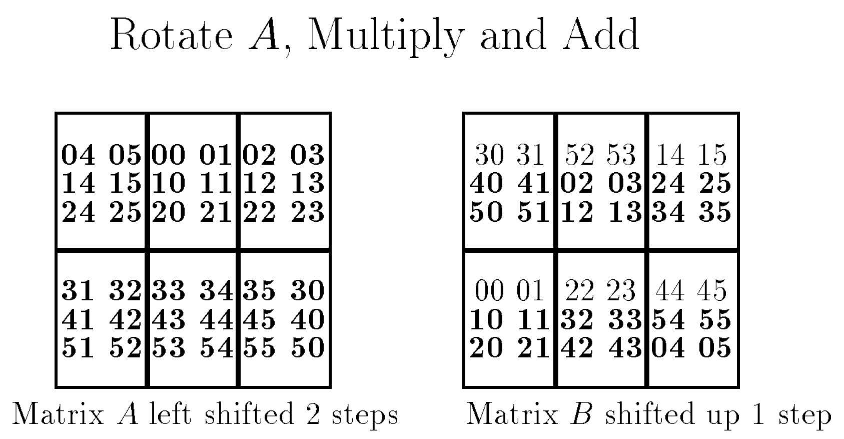

Figure 5: Matrix multiplication on a 2×3 array, last step.

The pseudocode for the above algorithm is given in the Appendix.

Claim 1. *By performing the alignment along the longest axis as if the processor array was square with the number of processors along each axis equal to the number of processors along the shortest axis, and the alignment along the shortest axis as if the processor array was square with the number of processors along each axis equal to the number of processors along the longest axis, the multiplication of two matrices can be accomplished by performing the minimum number of rotation steps along each axis.*

The correctness of the claim follows from the algorithm outlined above. Figures 4 and 5 show an example where the least common divisor of N_r and N_c is 1.

Remark 4. There is no local data motion (except what is required by the local BLAS) as was the case for a square processing array. Entire submatrices are moved between processors when necessary.

Remark 5. If one of the axes is not a multiple of the other, then not all local matrix multiplications are of the same rank. A uniform rank can be achieved without extra local memory moves, but at the expense of having some shifts use different block sizes and more complex memory management.

Remark 6. As in the case of a square processing array, A and B can be split into two halves, such that an exchange operation can be performed along each axis. Note also that moves on the two axes may not always be performed concurrently, since the longer axis requires more cyclic shifts than the shorter axis. Furthermore, in general the submatrices of A and B on each node are of different size, and a complete overlap of the motion of A and B is impossible even for a square processor array. If the lengths of the axes are relatively prime, then no communication is overlapped between A and B.

4.3 Optimum shape of the processor array.

Ignoring the effects of the ceiling functions, the speedup of the arithmetic operations is perfect and independent of the shape of the processor array. With the inner axis entirely instantiated in time, the matrix product requires $2\lceil\frac{P}{N_r}\rceil\lceil\frac{R}{N_c}\rceil\lceil\frac{Q\,\mathrm{gcd}(N_r,N_c)}{N_rN_c}\rceil\frac{N_rN_c}{\mathrm{gcd}(N_r,N_c)}$ arithmetic operations in sequence. The arithmetic time is proportional to the number of matrix elements of C per physical processor, the order of the rank $\lceil\frac{Q\,\mathrm{gcd}(N_r,N_c)}{N_rN_c}\rceil$ updates, and the number of such updates in sequence.

The communication time for the multiplication phase is proportional to $\lceil\frac{P}{N_r}\rceil\lceil\frac{Q}{N_c}\rceil(N_c-1)$ for A and to $\lceil\frac{Q}{N_r}\rceil\lceil\frac{R}{N_c}\rceil(N_r-1)$ for B. The communication time is entirely due to the all–to–all broadcast. Reference [14] shows that the above broadcast times are optimum when communication is restricted to one channel at a time per processor. With concurrent communication on all channels of every node of a Boolean cube network, the communication time can be reduced by a factor of $\log_2 N_c$ for A and a factor of $\log_2 N_r$ for B [11, 14].

The alignment phase for the Connection Machine system CM-200 implementation combines establishing a shared processor configuration for the three arrays with the alignment of the matrices with respect to each other. This operation is performed by the router. Reshaping the processor array for an operand, if necessary, constitutes a *shuffle* operation on a suitably defined index set. For the Connection Machine system, CM–200, reshaping the processor array implies that the number of processors along one axis is reduced by some power of two, while the number of processors along the other axis is increased by the same factor. Figure 6 gives an example in which a 2×4 processor array is reshaped into a 4×2 array. The number of partitions along the first axis doubles, while the number of partitions along the second axis is reduced by a factor of two. To verify that this operation is a shuffle, introduce a second index for the partitioning of the set of matrix rows initially in each processor. Then the content of the first two processor columns before the reshape operation is formed by blocks labeled 00,01,10,11,20,21,30 and 31, where the first index denotes the processor to which the block of rows is assigned. After reshaping the processor array, processor 0 contains blocks 00 and 20, processor 1 blocks 01 and 21, processor 2 blocks 10 and 30, and processor 3 blocks 11 and 31. This reallocation can be described as the reordering of the sequence 00,01,10,11,20,21,30,31 into the sequence 00,20,01,21,10,30,11,31. This reordering is known as a shuffle. Optimum Boolean cube algorithms are given in [15, 16]. With concurrent communication on all channels of every processor, as in the case of the Connection Machine system CM–200, the optimum time for the reshaping is independent of the axes extent and is proportional to the number of elements per processor.

The alignment of the matrices with respect to each other implies a skewing of rows with respect to each other for A and a skewing of columns with respect to each other for B, if a shared processor array shape has already been established. The skewing can be performed as a sequence of cyclic shifts, with the number of cyclic shifts depending upon the processor row for A and the processor column for B. On a Boolean cube network, it is conjectured [12] that the data motion for this operation can be pipelined, and that the communication time is proportional to the number of elements per processor and

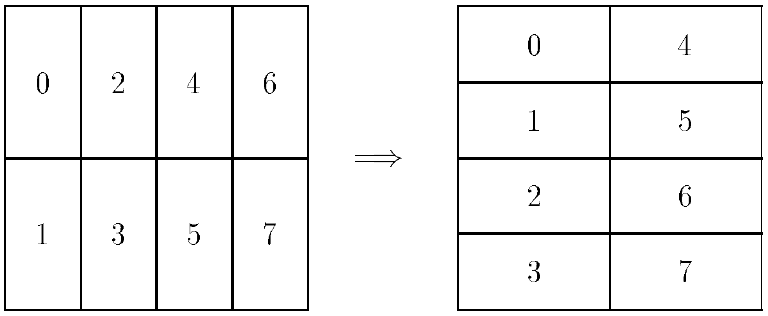

Figure 6: Reshaping a processor array.

essentially is independent of the shift length and the number of processors. Hence, it is expected that the time for the combined processor array reshaping and matrix alignment is proportional to the size of the submatrices assigned to each processor, and almost independent of the machine size. The measurements reported in Section 5 show that this assumption is valid for the Connection Machine system CM-200 router.

The number of arithmetic operations in sequence is independent of the processor array shape. To a first approximation, the alignment/reshaping is also independent of the array shape. Hence the optimum shape is determined solely by the time for the all–to–all broadcast in the multiplication phase.

Claim 2. *The optimum aspect ratio of the physical processor array is the same as the aspect ratio of the product matrix C, i.e.,* $N_r = \sqrt{\frac{PN}{R}}$ *and* $N_c = \sqrt{\frac{RN}{P}}$, *or* $\frac{N_r}{N_c} = \frac{P}{R}$.

The claim follows directly from the expressions for the time required for all–to–all broadcast, and is verified experimentally in Section 5. Note that the optimum two-dimensional processor array shape is the default processor array shape for C (assigned by the Connection Machine compilers).

The communication times for the multiplication phase may be reduced further by configuring the processors as a three–dimensional array [13] and by using all communication channels in Boolean cubes [11]. During the multiplication phase, the algorithm described here and used for the performance data reported in the next section, concurrently uses at most only four channels per processor.

5 Performance.

Table 1 and Figures 7, 8 and 9 give the performance of the matrix–vector, vector–matrix, and rank–1 update routines for square matrices. For real operands, the peak performance of the three algorithms in 64–bit precision is 4500 Mflops/s, 5825 Mflops/s, and 3266 Mflops/s, respectively. The corresponding peak performance of the three algorithms for complex operands is 9200 Mflops/s, 11950 Mflops/s, and 4900 Mflops/s respectively. The improvement in performance, for a fixed number of processors, is primarily due to an increased efficiency of the local level–2 BLAS.

Table 2 gives the performance for the local matrix–vector multiplication kernels [18]. The

Matrix	Mflops/s				Time (millisec)			
shape	Number of processors				Number of processors			
$P \times P$	256	512	1024	2048	256	512	1024	2048
Matrix–vector multiplication								
512	107	104	128	143	4.9	5.0	4.1	3.7
1024	233	279	405	411	9.0	7.5	5.2	5.1
2048	443	596	908	1122	18.9	14.0	9.2	7.5
4096	697	1067	1643	2330	48.2	31.4	20.4	14.4
8192	—	—	3056	4541	—	—	43.9	29.6
Vector–matrix multiplication								
512	105	124	125	147	5.0	4.2	4.2	3.6
1024	266	325	406	474	7.9	6.5	5.2	4.4
2048	496	838	918	1246	16.9	10.0	9.1	6.7
4096	790	1462	2121	2709	42.5	22.9	15.8	12.4
8192	—	—	3913	5829	—	—	34.3	23.0
Rank–1 update								
512	99	111	119	129	5.3	4.7	4.4	4.1
1024	213	278	380	426	9.9	7.5	5.5	4.9
2048	346	577	763	1078	24.3	14.5	11.0	7.8
4096	453	823	1450	2007	74.1	40.8	23.1	16.7
8192	—	—	1901	3266	—	—	70.6	41.1

Table 1: Performance data for matrix–vector, vector–matrix, and rank–1 updates on different Connection Machine system CM–200 configurations, 64–bit precision.

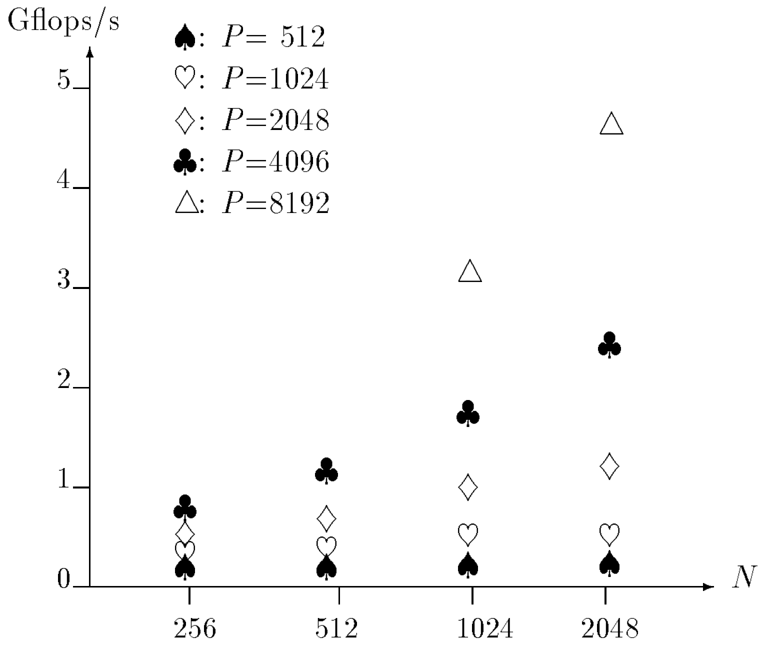

Figure 7: Execution rates for multiplication of a $P \times P$ matrix by a vector on Connection Machine system CM–200 with N processors, 64–bit precision.

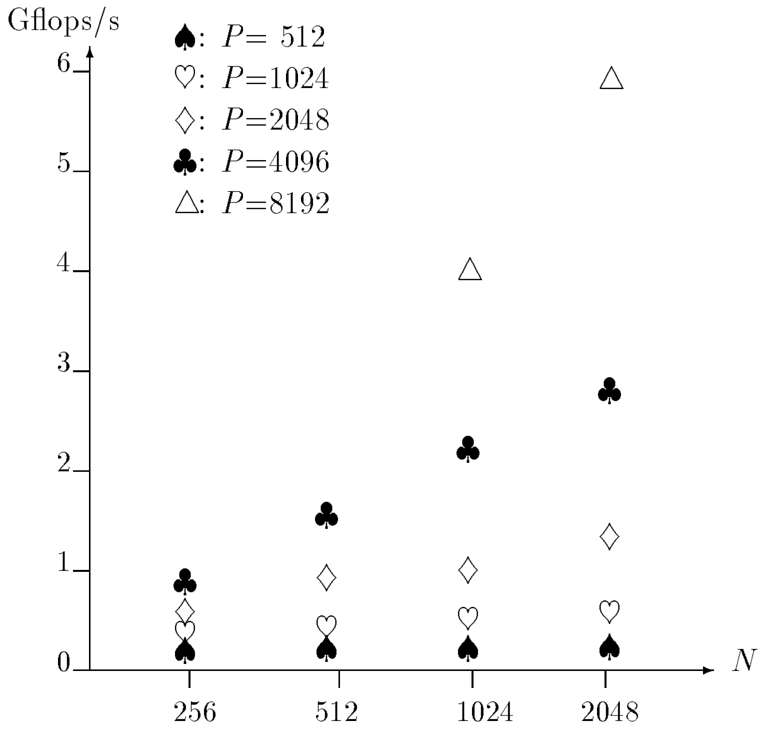

Figure 8: Execution rates for multiplication of a vector by a $P \times P$ matrix on Connection Machine system CM–200 with N processors, 64–bit precision.

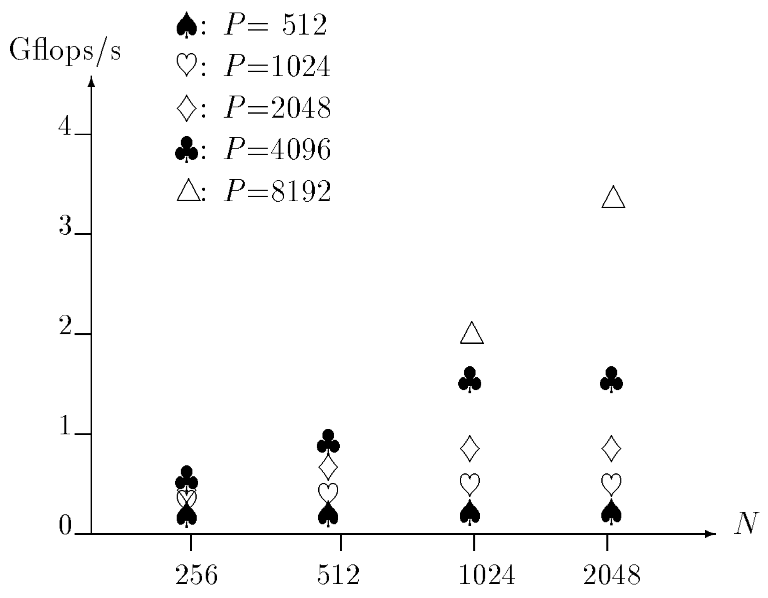

Figure 9: Execution rates for rank–1 updates on $P \times P$ matrices on Connection Machine system CM–200 with N processors, 64–bit precision.

performance of the local kernels for 256×512 submatrices is almost three times higher than for 4×8 submatrices. Note that the performance numbers in this table correspond to *one* floating point unit of the Connection Machine system CM–200. The peak performance for the rank–1 update kernels is approximately 25% of the peak performance of the matrix–vector multiplication kernels.

When there is at least one data element per processor for each operand, almost 30% of the total time of the matrix–vector multiply is spent in the local level–2 BLAS. Another 35% of the total time is spent in the two align operations (Steps 1 and 5 of the algorithm in Section 3). The remaining 35% is spent in the spread (Step 2) and reduction (Step 4) operations. The main reason for the difference in the peak performance of the matrix–vector and the vector–matrix implementations is the asymmetry in the communication times for the align operations in Steps 1 and 5 are significantly different. Moreover, for rectangular processor layouts, the time for the spread and the reduction operations are considerably different. The peak performance of the rank–1 update is significantly less than the peak performance of the matrix–vector and the vector–matrix functions primarily because of the lower efficiency of the corresponding local level–2 BLAS.

Tables 3 – 4 and Figures 10 – 14 give some performance data for the systolic matrix–matrix multiplication algorithm with the product matrix stationary and with processors configured as a two–dimensional array. For a given machine size, the performance increases as the size of the submatrices on the processing nodes increases. The overall performance for the square matrix multiplication increases by a factor of nearly 15 as the

DGEMV											
Number of	Number of columns										
Rows	2	3	4	5	8	16	32	64	128	512	2048
2	0.64	0.89	1.10	1.29	2.03	3.04	3.93	4.67	5.17	5.62	5.74
3	0.91	1.25	1.53	1.78	2.71	4.20	5.15	5.80	6.21	6.56	6.66
4	1.13	1.54	1.88	2.17	3.22	4.44	5.51	6.26	6.72	7.12	7.23
5	1.33	1.80	2.18	2.51	3.64	5.23	6.21	6.84	7.22	7.54	7.63
8	1.80	2.39	2.87	3.26	4.47	5.82	6.85	7.51	7.90	8.11	8.22
16	2.54	3.29	3.79	4.25	5.58	6.89	7.80	8.35	8.65	8.90	8.97
32	2.79	3.60	4.11	4.69	5.92	7.14	7.95	8.44	8.70	8.92	—
64	3.19	4.10	4.60	5.12	6.37	7.52	8.26	8.69	8.93	9.11	—
128	3.33	4.20	4.74	5.22	6.49	7.60	8.30	8.70	8.92	—	—
512	3.45	4.33	4.86	5.30	6.53	7.57	8.22	8.60	—	—	—
2048	3.49	4.89	4.91	5.34	6.57	7.60	—	—	—	—	—

Table 2: Execution rate in Mflops/s for matrix–vector multiplication on *one* Connection Machine system CM–200 floating–point processor, 64–bit precision.

size of the submatrices increases from 4×8 to 256×512 (Figure 12). For a given processor configuration the communication time for A increases in proportion to P and Q, and for B in proportion to Q and R. The number of arithmetic operations increases in proportion to P, Q, and R. Hence, the relative influence of the communication decreases with an increased matrix size. However, the influence decreases less rapidly than predicted by this simple analysis, because the effective floating point rate of the local BLAS increases with the size of the submatrices. It is clear from Table 4 that there are no size restrictions on the extents of the matrix axes.

Table 5 provides some additional insights into the behavior of the systolic matrix–matrix multiplication routine with the product matrix stationary. As a first approximation, the alignment time is proportional to the number of matrix elements per processor. There is a slight increase in the alignment time with the machine size (about 6 – 7% for a factor of 4 increase in machine size (not reported in the tables)). The time for cyclic shifts is proportional to the number of elements per processor and the number of shifts along an axis. The dependence of the time for cyclic shifts upon the number of elements per node is verified by the measurements in Table 5. The Connection Machine model CM–200 communication system allows data to be exchanged concurrently on all channels of every processor. Each of the operands A and B is split into two halves, one half moving in the positive direction of an axis and the other half in the negative direction. Moreover, equal–sized parts of A and B are moved concurrently whenever possible. Recall that for rectangular arrays more cyclic shifts are required for the longer axis and that for different sized matrices the submatrices are of different size.

The overall performance as a function matrix sizes and the number of processors is shown in Figures 10 and 11. Increasing the number of processors by a factor of four and configuring the machine as a square array reduces the number of matrix elements per processor by a factor of four, but doubles the processor axes' lengths. The time for the alignment is reduced by almost a factor of two, and the time for the cyclic shifts is reduced by a factor of two. The number of arithmetic operations per processor for square matrices and processing node configurations decreases by a factor of eight for each local matrix multiply. However, the number of calls to the local matrix multiply function increases by a factor of two. The loss in efficiency in the local level–2 BLAS because of a reduced size of the submatrices, does not quite reduce the arithmetic time by a factor of four. The relative importance of the communication increases by about a factor of two.

For double precision operands, the systolic multiplication phase (local matrix multiply and cyclic shifts in Table 3) has a peak floating–point rate of approximately 11 Gflops/s. After accounting for the alignment and reconfiguration phases, the overall algorithm peaks at nearly 10 Gflops/s. The overall peak performance of the algorithm for complex matrices ($N = 8192$) is 12.5 Gflops/s.

Figures 13 and 14 show how the performance of the systolic algorithm for the multiplication of a rectangular matrix and a square matrix varies with the machine size (C stationary). The speedup with the machine size is almost identical for the different matrix shapes and is linear in the machine size. The efficiency for matrices of a given size decreases with machine size, as explained above. The speedup is approximately 5 for an increase in machine size by a factor of 8.

Figure 15 shows the dependence of the performance upon the shape of the product matrix for the systolic algorithm with C stationary. For a given inner dimension Q, the performance is approximately independent of the values of P and R for $P \times R = const$. The performance increases with the value of the constant and increased values of Q. The deviation from perfect symmetry is due mostly to the fact that a Connection Machine system CM–200 with 2048 processors cannot be configured as a square array. In this case, one of the pairs in each symmetric case ($P \times R$ and $R \times P$) may incur a reallocation of the product matrix, while the other case does not. The asymmetry is very small for machine sizes for which the number of floating-point processors is a square, as seen from Table 3. Note that in Figure 15 P and R are expressed relative to Q. Hence, the increased performance as Q increases, since all three local axes increases with an increased value of Q for fixed ratios of P/Q and R/Q.

Figures 16 and 17 illustrate the influence of the shape of the shared processing node configuration on the overall performance of the systolic matrix multiply algorithm with C stationary. For square operand matrices, when the processing nodes are configured as a linear array ($N_r = 1$ or $N_c = 1$), the number of cyclic shifts that are required for the all–to–all broadcast in the multiplication phase completely dominates the algorithm. For the square matrices considered in Figure 16, the total number of cyclic shifts is minimum when $N_r = N_c$. For the rectangular shapes illustrated in Figure 17, performance is optimal when the shared processor configuration is of the same shape as the product matrix when $P < R$. For the case when $P > R$, the default processor layout for the product matrix makes it essential for the Connection Machine system CM-200 implementation to do one extra route operation to reconfigure the product matrix. This extra communication step shifts the optimum floating point rate versus shared processor configuration to the left.

Matrix shape $P \times Q \times R$	Mflops/s				Time (sec)			
	Number of processors				Number of processors			
	256	512	1024	2048	256	512	1024	2048
$1024 \times 1024 \times 128$	300	365	704	838	0.895	0.735	0.381	0.320
$1024 \times 1024 \times 256$	377	548	826	1339	1.424	0.980	0.650	0.401
$1024 \times 1024 \times 512$	559	736	1289	1729	1.921	1.459	0.833	0.621
$1024 \times 1024 \times 1024$	744	1079	1788	2647	4.022	2.544	1.745	1.054
$512 \times 1024 \times 1024$	534	844	1231	2038	2.011	1.272	0.872	0.527
$256 \times 1024 \times 1024$	403	597	913	1366	1.332	0.899	0.588	0.393
$128 \times 1024 \times 1024$	271	389	609	987	0.991	0.690	0.441	0.272
$2048 \times 2048 \times 128$	521	674	1244	1608	2.061	1.593	0.863	0.668
$2048 \times 2048 \times 256$	476	622	1236	1520	4.512	3.453	1.737	1.413
$2048 \times 2048 \times 512$	649	815	1515	2507	6.618	5.270	2.835	1.713
$2048 \times 2048 \times 1024$	908	1257	2252	3177	9.460	6.834	3.814	2.704
$2048 \times 2048 \times 2048$	1128	1724	2975	4646	15.230	9.965	5.775	3.698
$1024 \times 2048 \times 2048$	871	1417	2194	3687	9.862	6.062	3.915	2.330
$512 \times 2048 \times 2048$	685	1039	1686	2509	6.270	4.134	2.547	1.712
$256 \times 2048 \times 2048$	470	740	1167	1849	4.569	2.902	1.840	1.161
$128 \times 2048 \times 2048$	481	879	1189	1909	2.232	1.222	0.903	0.562
$4096 \times 4096 \times 256$	743	1074	2044	2797	11.560	8.000	4.203	3.071
$4096 \times 4096 \times 512$	706	1033	1967	2608	24.334	16.631	8.734	6.587
$4096 \times 4096 \times 1024$	1020	1336	2546	4351	33.686	25.718	13.500	7.897
$4096 \times 4096 \times 2048$	—	1892	3513	5365	—	36.321	19.561	12.809
$4096 \times 4096 \times 4096$	1516	2348	4327	7318	90.650	58.534	31.763	18.781
$2048 \times 4096 \times 4096$	1263	2031	3486	6106	54.410	33.835	19.713	11.254
$1024 \times 4096 \times 4096$	1050	1636	2816	4552	32.724	21.002	12.202	7.548
$512 \times 4096 \times 4096$	786	1152	2025	3425	21.857	14.913	8.484	5.016
$256 \times 4096 \times 4096$	748	1392	2004	3539	11.484	6.171	4.286	2.427
$256 \times 256 \times 256$	240	320	497	668	0.140	0.105	0.068	0.050
$512 \times 512 \times 512$	439	609	987	1400	0.612	0.441	0.272	0.192
$1024 \times 1024 \times 1024$	744	1079	1788	2647	2.888	1.991	1.201	0.811
$2048 \times 2048 \times 2048$	1128	1724	2975	4646	15.232	9.965	5.776	3.698
$4096 \times 4096 \times 4096$	1517	2348	4327	7318	90.650	58.547	31.767	18.781
$8192 \times 8192 \times 8192$	—	—	—	9929	—	—	—	110.737

Table 3: Performance of the systolic matrix multiplication algorithm with C stationary on some Connection Machine system CM–200 configurations, 64–bit precision.

Matrix shape	Gflops/s	
$P \times P \times P$	Real	Complex
6400	9.3	12.5
6656	9.5	12.7
6912	9.6	12.8
7168	9.8	12.9
7424	9.9	13.0
7680	10.1	13.2
7936	10.3	13.3
8192	10.6	13.6
8448	10.6	13.6
8704	10.7	13.6
8960	10.8	13.7
9216	11.0	13.9
9472	11.1	14.0
9728	11.3	14.1
9984	11.3	14.1
10240	11.5	14.3
10496	11.6	14.3
10752	11.7	14.4
11008	11.8	14.5
11264	11.9	14.6
11520	11.9	14.5
11776	12.1	14.7
12032	12.2	14.7
12288	12.3	14.8
12544	12.3	14.9
12800	12.4	14.9
13056	12.5	14.9
13312	12.6	15.0
13568	12.7	
13824	12.8	
14080	12.8	
14336	12.9	
14592	12.9	
14848	13.0	
15104	13.1	
15360	13.2	
15616	13.2	
15872	13.3	

Table 4: Performance of the systolic matrix multiplication algorithm with C stationary on a 64K Connection Machine system CM–200 configuration, 64–bit precision.

Matrix size	256 × 256		512 × 512		1024 × 1024		2048 × 2048		4096 × 4096	
Operation	Time	%	Time	%	Time	%	Time	%	Time	%
Alignment	0.030	22	0.117	19	0.473	16	1.914	13	7.758	9
Local matmul	0.020	14	0.133	22	0.968	34	7.529	49	59.738	66
Cyclic shifts	0.089	64	0.362	59	1.447	50	5.789	38	23.153	26
Total	0.139	100	0.613	100	2.889	100	15.232	100	90.650	100

Table 5: Relative execution times for the global matrix multiplication algorithm on a 256 processor Connection Machine system CM–200, 64–bit precision.

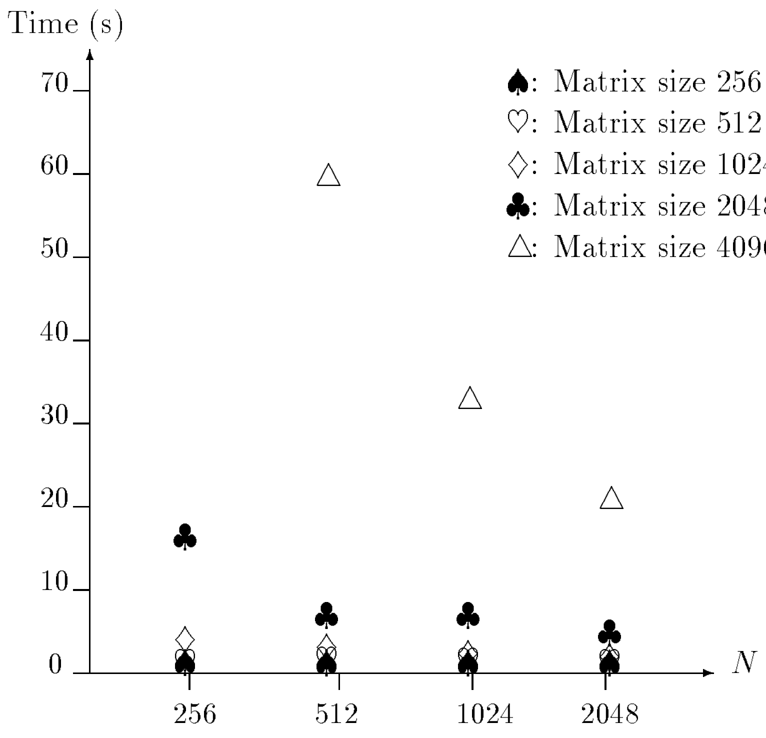

Figure 10: Execution time for multiplication of square matrices, 64–bit precision. N is the size of the Connection Machine system CM–200.

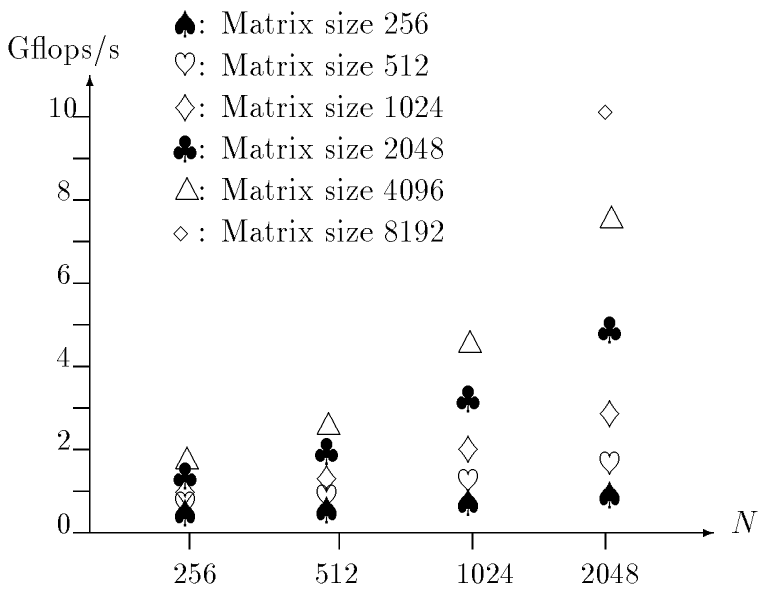

Figure 11: Execution rates for multiplication of square matrices, 64–bit precision. N is the size of the Connection Machine system CM–200.

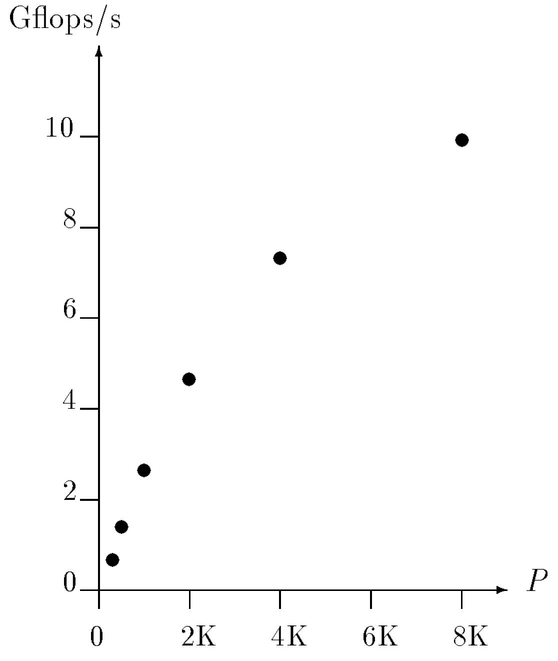

Figure 12: Performance of square matrix multiplication as a function of matrix size P on a 2048 processor Connection Machine system CM–200, 64–bit precision.

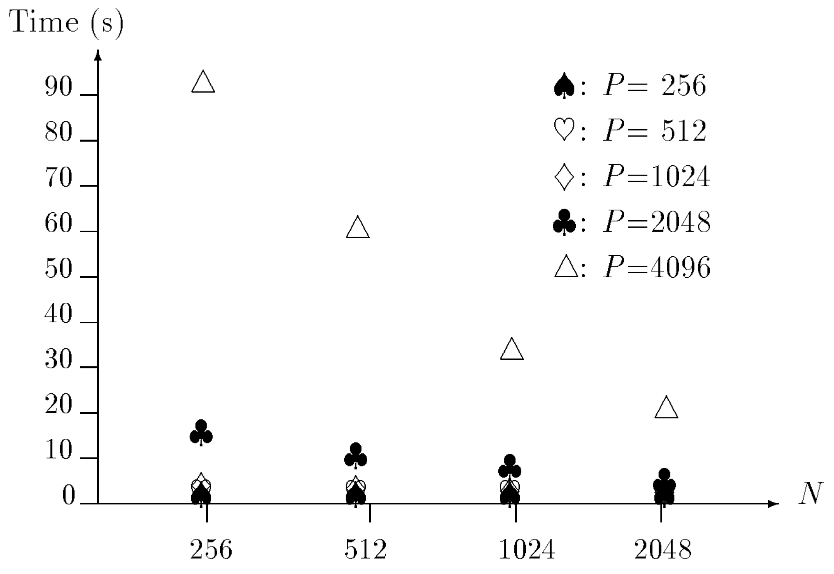

Figure 13: Execution time for a $P \times 4096 \times 4096$ matrix multiplication, 64–bit precision. N is the size of the Connection Machine system CM–200.

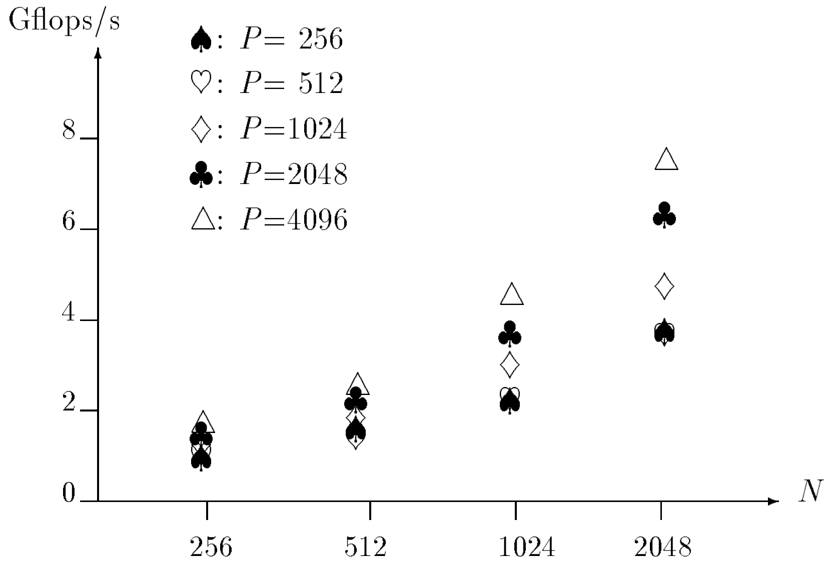

Figure 14: Execution rates for $P \times 4096 \times 4096$ matrix multiplication, 64–bit precision. N is the size of the Connection Machine system CM–200.

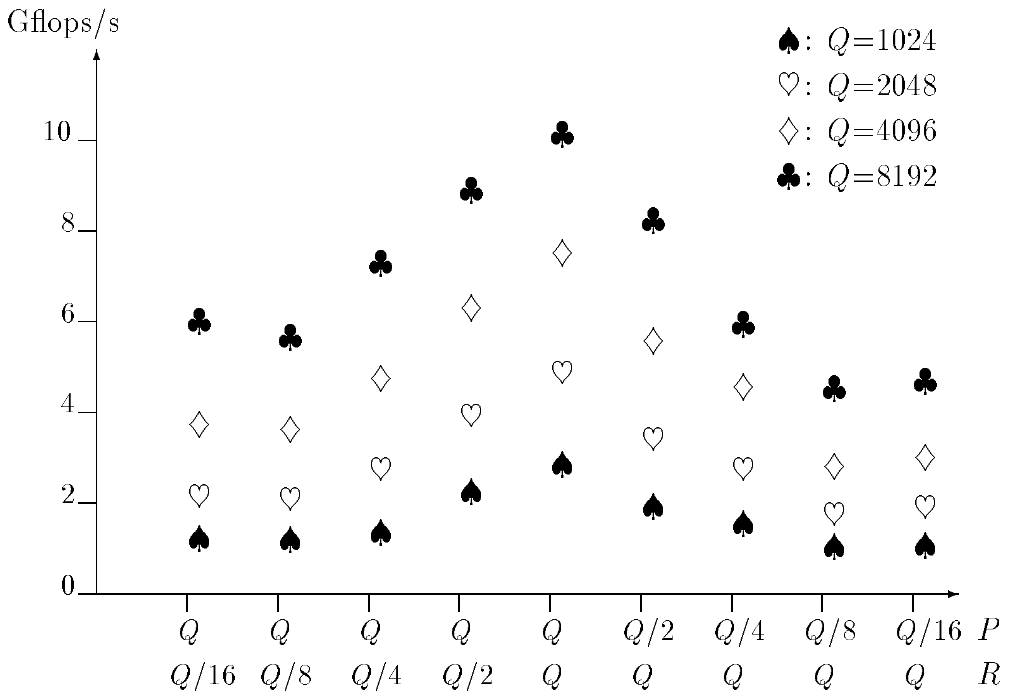

Figure 15: Execution rates for multiplication of rectangular matrices on a 2048 processor Connection Machine system CM–200, 64–bit precision.

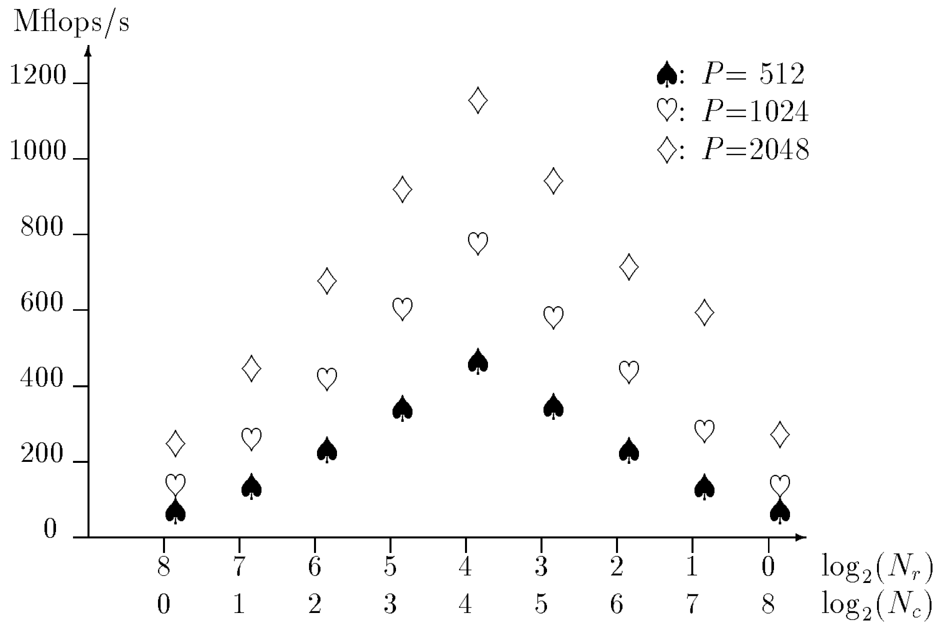

Figure 16: Influence of shared processor configuration on the performance for multiplication of square matrices of size P, 64–bit precision. The shape of the 256 processor Connection Machine system CM–200 is $N_r \times N_c = 256$.

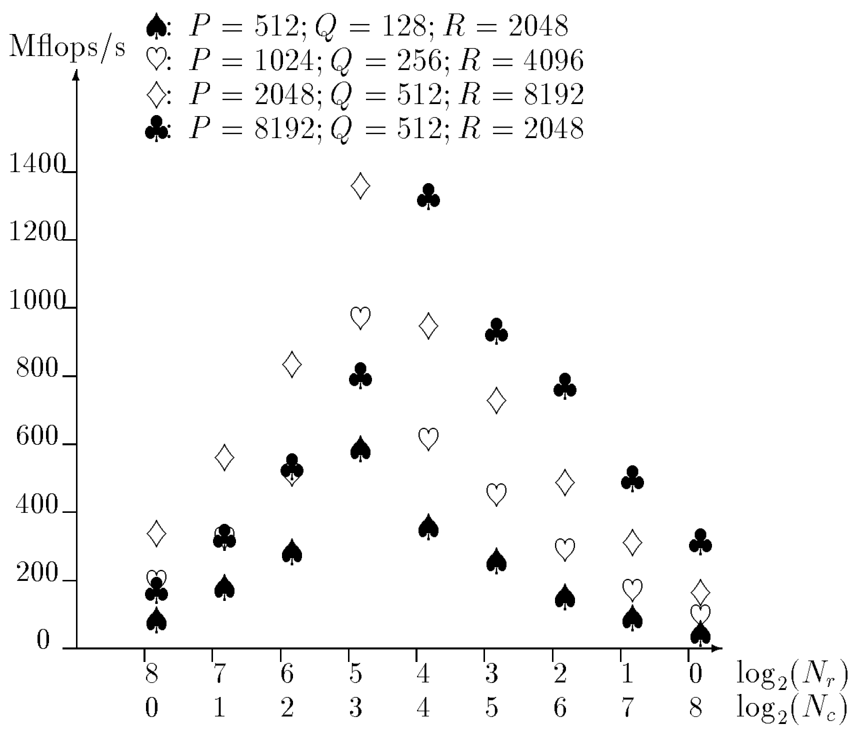

Figure 17: Influence of shared processor configuration on multiplication of rectangular matrices of shape $P \times Q \times R$, 64–bit precision. The shape of the 256 processor Connection Machine system CM–200 is $N_r \times N_c = 256$.

Finally, Table 6 and Figure 18 show the performance for the multiplication of a square matrix A by a rectangular matrix B. The number of columns in B varies from one to 8192. The effect of the blocking introduced for the nonsystolic "matrix–vector" multiplication algorithm, and the selection of algorithm for matrix–matrix multiplication as a function of the shape of the operands is clear. When $\frac{P}{R} \geq 8$, a matrix–vector type algorithm is used with a block size of $\min(R, 64)$ (A is stationary). When $\frac{R}{P} \geq 16$, a vector–matrix algorithm is used to evaluate the product (B is stationary). The block size is $\min(P, 32)$. For the data reported in Table 6 and Figure 18, all other shapes use the systolic matrix multiplication algorithm with C stationary. For small matrices, blocking of vectors for the nonsystolic algorithm, yields more than a three-fold performance increase for square matrices, while for large matrices the performance increase is about 5%. For small matrices, blocking improves the performance of the local level–2 BLAS by about 50%.

6 Summary.

Level–2 and level–3 DBLAS used in the Connection Machine Scientific Software Library, CMSSL, are described briefly. Both the nonsystolic and the systolic algorithms for matrix–matrix multiplication on distributed data structures, with A, B, or C stationary are outlined. A systolic matrix–matrix multiplication algorithm keeping C stationary is described in detail. The systolic algorithms perform the all–to–all broadcast and reduction operations as a sequence of cyclic shifts, thereby reducing the need for temporary storage.

Matrix shape $P \times P \times R$	Mflops/s				Time (millisec)			
	Number of processors				Number of processors			
	256	512	1024	2048	256	512	1024	2048
$256 \times 256 \times 1$	33	35	43	49	4.0	3.7	3.0	2.7
$256 \times 256 \times 4$	62	74	104	105	8.5	7.1	5.0	5.0
$256 \times 256 \times 16$	74	93	154	171	28.3	22.6	13.6	12.3
$256 \times 256 \times 64$	104	145	206	303	80.7	57.9	40.7	27.7
$256 \times 256 \times 256$	240	320	497	668	139.8	104.9	67.5	50.2
$256 \times 256 \times 1024$	398	564	834	1151	337.2	238.0	160.9	116.6
$256 \times 256 \times 4096$	182	346	397	738	2949.8	1551.7	1352.3	727.5
$256 \times 256 \times 8192$	—	—	698	795	—	—	1538.3	1350.6
$512 \times 512 \times 1$	107	104	128	143	4.9	4.0	4.1	3.7
$512 \times 512 \times 4$	133	164	251	297	15.8	12.8	8.4	7.1
$512 \times 512 \times 16$	165	186	329	386	50.8	45.1	25.5	21.7
$512 \times 512 \times 64$	167	204	367	405	200.9	164.5	91.4	82.9
$512 \times 512 \times 256$	318	398	689	890	422.1	337.2	194.8	150.8
$512 \times 512 \times 1024$	528	826	1219	1948	1016.8	650.0	440.4	275.6
$512 \times 512 \times 4096$	796	1205	1981	2902	2697.8	1782.1	1084.0	740.0
$512 \times 512 \times 8192$	—	—	755	1435	—	—	5688.7	2993.0
$1024 \times 1024 \times 1$	233	279	405	411	9.0	5.2	5.1	5.1
$1024 \times 1024 \times 4$	253	344	570	652	33.2	24.4	14.7	12.9
$1024 \times 1024 \times 16$	308	390	621	800	108.9	86.0	54.0	41.9
$1024 \times 1024 \times 64$	294	391	708	865	456.5	343.3	189.6	155.2
$1024 \times 1024 \times 256$	377	548	826	1339	1424.1	979.7	650.0	400.9
$1024 \times 1024 \times 1024$	744	1079	1788	2647	2886.4	1990.3	1201.1	811.3
$1024 \times 1024 \times 4096$	1066	1652	2771	4478	8058.1	5199.7	3099.9	1918.3
$1024 \times 1024 \times 8192$	—	—	3219	5383	—	—	5337.0	3191.5
$4096 \times 4096 \times 1$	697	1067	1913	2330	48.1	31.4	17.5	14.4
$4096 \times 4096 \times 4$	719	1187	1995	2734	186.7	113.1	67.3	49.1
$4096 \times 4096 \times 16$	794	1217	2262	2966	676.2	441.1	237.3	181.0
$4096 \times 4096 \times 64$	787	1160	2258	2962	2728.7	1851.3	951.1	725.0
$4096 \times 4096 \times 256$	743	1074	2044	2797	11561.2	7998.1	4202.5	2883.5
$4096 \times 4096 \times 1024$	1020	1336	2546	4351	33686.0	25718.4	13495.6	7897.0
$4096 \times 4096 \times 4096$	1517	2348	4327	7318	90599.2	58534.5	31763.1	18780.9
$4096 \times 4096 \times 8192$	—	—	4952	8929	—	—	55508.5	30784.8
$8192 \times 8192 \times 1$	—	—	3056	4541	—	—	43.9	29.6
$8192 \times 8192 \times 4$	—	—	3354	4643	—	—	160.1	115.6
$8192 \times 8192 \times 16$	—	—	3416	5011	—	—	628.7	428.6
$8192 \times 8192 \times 64$	—	—	3416	4993	—	—	2514.6	1720.4
$8192 \times 8192 \times 256$	—	—	3164	4756	—	—	10859.6	7224.5
$8192 \times 8192 \times 1024$	—	—	2906	4308	—	—	47294.9	31903.2
$8192 \times 8192 \times 4096$	—	—	—	8027	—	—	—	68488.3
$8192 \times 8192 \times 8192$	—	—	—	9929	—	—	—	110737.4

Table 6: Performance for the multiplication of a square matrix A by a rectangular matrix B on some Connection Machine system CM–200 configurations using the matrix–matrix multiplication function in CMSSL, 64–bit precision.

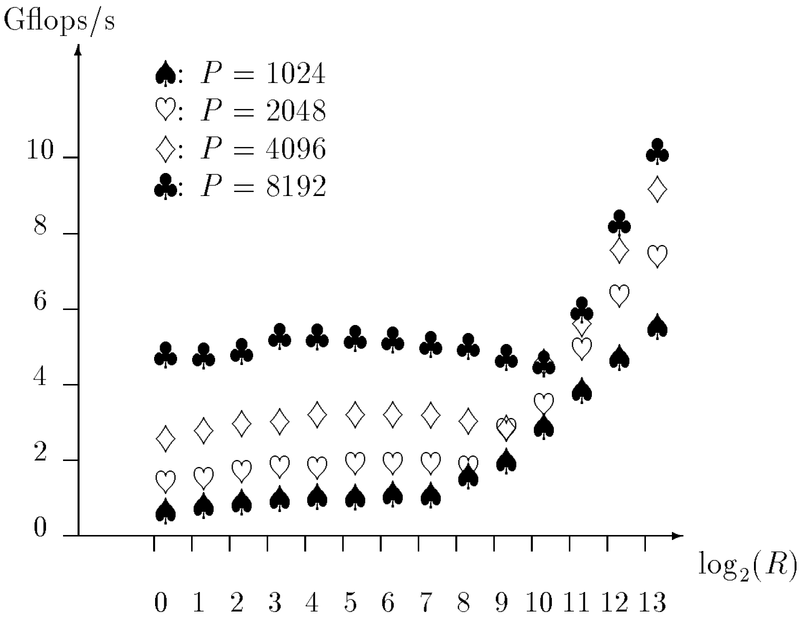

Figure 18: Performance of the matrix multiplication function in the Connection Machine Scientific Software Library for the multiplication of a $P \times P$ matrix by a $P \times R$ matrix on Connection Machine system CM–200, 64–bit precision.

All algorithms require that the operands first be aligned with respect to each other, and that a common machine configuration is established for all operands.

For the matrix–vector multiplication routine, about 35% of the time is spent in the alignment of the two vectors, while the spread–and–reduce operations account for another 30% of the total time. The matrix–vector and vector–matrix multiplication routines have comparable performance. The difference is due to the asymmetry in the route operation. The performance of the rank–1 update routine is always less than for the matrix–vector and vector–matrix routines. For small matrices, the performance difference is about 10%, but for large matrices the performance difference is close to a factor of two. The performance for the local level–2 BLAS used in the level–2 DBLAS is up to a factor of four higher for matrix–vector (vector–matrix) multiplication than for rank-1 updates.

It has been shown experimentally that introducing blocking in the matrix–vector and vector–matrix multiplication routines in their use for matrix–matrix multiplication yields a performance improvement by over a factor of three for small matrices. For large matrices, the performance gain by blocking is about 5%. Most of the performance gain for small matrices is due to improved performance of the local level–2 BLAS. For the same blocking factor, the performance of the matrix–vector and vector–matrix multiplication routines is always better than the performance of the rank-b update, primarily because of the better performance of the local level–2 BLAS functions.

For the systolic matrix–matrix multiplication routine with C stationary, about 20% of the time is spent for the alignment of small matrices. The alignment of large matrices accounts for nearly 10% of the total time. The fraction of time spent for the cyclic rotation decreases from about 65% for small matrices to about 25% for large matrices. For submatrices of shape $p \times q$, $q \times r$, and $p \times r$, the time for cyclic rotation is proportional to the number of data elements moved in sequence, which for A is $\frac{p \times q}{2} N_c$, and for B is $\frac{q \times r}{2} N_r$. These shifts are partially overlapped. The required number of floating point operations, $2p \times q \times r$ increases with the size of the local matrices. The total time increases at a slower rate, because the floating point rate increases with an increase in the local matrix size. The increase in the floating–point rate is a factor of 14 from 2×2 submatrices to large submatrices.

As a first approximation, the algorithm which keeps the matrix with the largest number of elements stationary should be used to compute the matrix product. Deviations from this rule are primarily due to particular implementation issues for communication routines and variations in arithmetic efficiency due to different vector lengths for the different local submatrices.

The aspect ratio of the shape of the processor array for optimum communication performance shall be the same as the aspect ratio of the shape of the stationary matrix. This result has also been verified experimentally. For square matrices, the performance varies by a factor of over five as a function of the shape of the processing array. For the rectangular matrices used in the experiments reported here, the influence of the shared processing node configuration is much more severe (18 Mflops/s versus the optimal 559 Mflops/s – a factor of more than 30).

The local level–2 BLAS used in the algorithm are well–optimized for the node architecture, having a peak efficiency in excess of 90%. For improved efficiency in the global matrix multiplication algorithm, a decrease in the need for communication is desirable, in particular, in the multiplication phase of the algorithm. Configuring the processors as a three–dimensional array may yield lower communication requirements for the multiplication phase [13, 20]. A decrease in communication time can also be achieved by exploiting more of the communications bandwidth of the Boolean cube network by using multiple concurrent exchange sequences as suggested in [11].

Acknowledgment

Many people at Thinking Machines Corp. have provided support for the distributed BLAS functions that are described here. These include Mark Bromley, Alan Edelman, Tim Harris, Steve Heller, Doug MacDonald, and Luis Ortiz.

References

[1] Jean-Philippe Brunet and S. Lennart Johnsson. All-to-all broadcast with applications on the Connection Machine. *International Journal of Supercomputer Applications*, 6(3):241–256, 1992.

[2] L.E. Cannon. *A Cellular Computer to Implement the Kalman Filter Algorithm.* PhD thesis, Montana State Univ., 1969.

[3] M.Y. Chan. Embedding of grids into optimal hypercubes. *SIAM J. Computing*, 20(5):834–864, 1991.

[4] Eliezer Dekel, David Nassimi, and Sartaj Sahni. Parallel matrix and graph algorithms. *SIAM J. Computing*, 10:657–673, 1981.

[5] Jack J. Dongarra, Jeremy Du Croz, Iain Duff, and Sven Hammarling. A Set of Level 3 Basic Linear Algebra Subprograms. Technical Report Reprint No. 1, Argonne National Laboratories, Mathematics and Computer Science Division, August 1988.

[6] Jack J. Dongarra, Jeremy Du Croz, Iain Duff, and Sven Hammarling. A Set of Level 3 Basic Linear Algebra Subprograms: Model implementation and test programs. Technical Report Reprint No. 2, Argonne National Laboratories, Mathematics and Computer Science Division, August 1988.

[7] Geoffrey C. Fox and Wojtek Furmanski. Optimal communication algorithms on the hypercube. Technical Report CCCP-314, California Institute of Technology, July 1986.

[8] Geoffrey C. Fox, Mark A. Johnson, Gregory A. Lyzenga, Steve W. Otto, John K. Salmon, and David W. Walker. *Solving Problems on Concurrent Processors*. Prentice-Hall, 1988.

[9] I. Havel and J. Móravek. B-valuations of graphs. *Czech. Math. J.*, 22:338–351, 1972.

[10] Ching-Tien Ho and S. Lennart Johnsson. Embedding meshes in Boolean cubes by graph decomposition. *J. of Parallel and Distributed Computing*, 8(4):325–339, April 1990.

[11] Ching-Tien Ho and S. Lennart Johnsson. Matrix multiplication on hypercubes using full bandwidth and constant storage. In *The Sixth Distributed Memory Computing Conference*, pages 447–451. IEEE Computer Society Press, 1991.

[12] S. Lennart Johnsson. Communication efficient basic linear algebra computations on hypercube architectures. *J. Parallel Distributed Computing*, 4(2):133–172, April 1987.

[13] S. Lennart Johnsson and Ching-Tien Ho. Algorithms for multiplying matrices of arbitrary shapes using shared memory primitives on a Boolean cube. Technical Report YALEU/DCS/RR-569, Dept. of Computer Science, Yale University, October 1987.

[14] S. Lennart Johnsson and Ching-Tien Ho. Spanning graphs for optimum broadcasting and personalized communication in hypercubes. *IEEE Trans. Computers*, 38(9):1249–1268, September 1989.

[15] S. Lennart Johnsson and Ching-Tien Ho. Maximizing channel utilization for all-to-all personalized communication on Boolean cubes. In *The Sixth Distributed Memory Computing Conference*, pages 299–304. IEEE Computer Society Press, 1991.

[16] S. Lennart Johnsson and Ching-Tien Ho. Generalized shuffle permutations on Boolean cubes. *J. Parallel and Distributed Computing*, 16(1):1–14, 1992.

[17] S. Lennart Johnsson and Kapil K. Mathur. Distributed BLAS. Technical report, Thinking Machines Corp., 1992. In preparation.

[18] S. Lennart Johnsson and Luis F. Ortiz. Local Basic Linear Algebra Subroutines (LBLAS) for distributed memory architectures and languages with an array syntax. *The International Journal of Supercomputer Applications*, 6(4):322–350, 1992.

[19] C.L. Lawson, R.J. Hanson, D.R. Kincaid, and F.T. Krogh. Basic Linear Algebra Subprograms for Fortran Usage. *ACM TOMS*, 5(3):308–323, September 1979.

[20] Kapil K. Mathur and S. Lennart Johnsson. Matrix multiplication on a three-dimensional array. Technical report, Thinking Machines Corp., November 1991.

[21] Kapil K. Mathur and S. Lennart Johnsson. All–to–all communication algorithms for distributed BLAS. In *6th SIAM Conference on Parallel Processing for Scientific Computing*. SIAM, 1993. Harvard University Technical Report TR-07-93.

[22] David Nassimi and Sartaj Sahni. Optimal BPC permutations on a cube connected SIMD computer. *IEEE Trans. Computers*, C-31(4):338–341, April 1982.

[23] E.M. Reingold, J. Nievergelt, and N. Deo. *Combinatorial Algorithms*. Prentice-Hall, Englewood Cliffs. NJ, 1977.

[24] Arnold L. Rosenberg. Preserving proximity in arrays. *SIAM J. Computing*, 4:443–460, 1975.

[25] Quentin F. Stout and Bruce Wagar. Passing messages in link-bound hypercubes. In Michael T. Heath, editor, *Hypercube Multiprocessors 1987*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1987.

[26] Thinking Machines Corp. *CM Fortran optimization notes: slicewise model, version 1.0*, 1991.

[27] Thinking Machines Corp. *CMSSL for CM Fortran, Version 3.1*, 1993.

[28] Walter Tichy. Parallel matrix multiplication on the Connection Machine. In Horst D. Simon, editor, *Scientific Applications of the Connection Machine*, pages 174–187. World Scientific, 1989.

7 Appendix

This section describes the systolic algorithm where the product matrix C is kept stationary. The shape of the operand matrices is arbitrary. The processing nodes are assumed to be configured as a two–dimensional array of shape $N_r \times N_c$ for any N_r and N_c.

The alignment of each operand is expressed using two **forall** loops in the pseudocode below. The inner loop defines the maximum data set that is guaranteed to have the same data motion during alignment. For the matrix A, the alignment is performed on submatrices of shape $\alpha \times \beta^v$ ($\alpha = \lceil \frac{P}{N_r} \rceil, \beta^v = \lceil \frac{Q gcd(N_r,N_c)}{N_r N_c} \rceil$). For the matrix B the same properties hold for submatrices of shape $\beta^v \times \gamma$ ($\gamma = \lceil \frac{R}{N_c} \rceil$). All submatrices can be aligned concurrently, should there be sufficient communications bandwidth available.

The first three loops for the multiplication phase in the pseudocode below define a matrix multiplication local to a processor. It is the multiplication immediately succeeding the alignment. The main loop performs the remaining multiplications and a single cyclic shift. The local matrix multiplication is performed as a block outer product, or rank-$\lceil \frac{Q \operatorname{gcd}(N_r,N_c)}{N_r N_c} \rceil$ update. This update is defined in terms of a matrix–vector multiplication, where the matrix is of shape $\lceil \frac{P}{N_r} \rceil \times \lceil \frac{Q \operatorname{gcd}(N_r,N_c)}{N_r N_c} \rceil$. The matrix–vector product is expressed as a sequence of AXPY operations [19] with a vector length of $\lceil \frac{P}{N_r} \rceil$. This loop is for illustration only. The actual code calls a (local) matrix–vector routine. The local indices for matrix C are ϕ and ν; for matrix A the indices are ϕ and ψ, and for the matrix B the local indices are μ and ν. The local submatrix of A is treated in blocks of $\lceil \frac{Q \operatorname{gcd}(N_r,N_c)}{N_r N_c} \rceil$ columns, and the local submatrix of B in blocks of $\lceil \frac{Q \operatorname{gcd}(N_r,N_c)}{N_r N_c} \rceil$ rows. The inner index of the matrices A and B is $\beta^v \hat{Q} + \psi^v$, where ψ^v is the index local to a block, and $\hat{Q}$ is the block index. The local block index for A is $\hat{Q} \bmod \frac{N_r}{\operatorname{gcd}(N_r,N_c)}$ and the local block index for B is $\hat{Q} \bmod \frac{N_c}{\operatorname{gcd}(N_r,N_c)}$.

The organization of the computations corresponds to a reshaping of the data arrays to arrays with six axes. Two of the axes define the processor array. Two more axes are necessary for the enumeration of the virtual processors being emulated by a physical processor (one virtual axes for each of the physical processor array axis). Finally, two axes define the number of rows and columns for each virtual processor. For a square processor array a processor will emulate only one virtual processor, and four axes suffice.

Algorithm AM(C,A,B)

```
forall $\ell, m \in \{0,1,\ldots,N_r-1\} \times \{0,1,\ldots,N_c-1\}$ do

Alignment:

  forall $\hat{Q} := 0$ to $\frac{N_r}{\operatorname{gcd}(N_r,N_c)} - 1$ do
      forall $\phi, \psi^v \in \{0,1,\ldots,\alpha-1\} \times \{0,1,\ldots,\beta^v-1\}$ do
          $a(\alpha\ell+\phi, \beta^v(m\frac{N_r}{\operatorname{gcd}(N_r,N_c)} + \hat{Q}) + \psi^v) \leftarrow$
              $\leftarrow a(\alpha\ell+\phi, \beta^v(m\frac{N_r}{\operatorname{gcd}(N_r,N_c)} + \hat{Q} + \ell\frac{N_c}{\operatorname{gcd}(N_r,N_c)}) \bmod \frac{N_r N_c}{\operatorname{gcd}(N_r,N_c)} + \psi^v)$
```

```
        **endforall** $\phi, \psi^v$
    **endforall** $\hat{Q}$
    **forall** $\hat{Q} := 0$ **to** $\frac{N_c}{\gcd(N_r,N_c)} - 1$ **do**
        **forall** $\psi^v, \nu \in \{0,1,\ldots,\beta^v - 1\} \times \{0,1,\ldots,\gamma_1 - 1\}$ **do**
            $b(\beta^v(\ell\frac{N_c}{\gcd(N_r,N_c)} + \hat{Q}) + \psi^v, \gamma_1 m + \nu) \leftarrow$
                $\leftarrow b(\beta^v(\ell\frac{N_c}{\gcd(N_r,N_c)} + \hat{Q} + m\frac{N_r}{\gcd(N_r,N_c)}) \bmod \frac{N_r N_c}{\gcd(N_r,N_c)} + \psi^v, \gamma_1 m + \nu)$
        **endforall** $\psi^v, \nu$
    **endforall** $\hat{Q}$
```

Multiplication:

```
    **for** $\nu := 0$ **to** $\gamma - 1$ **do**
        **for** $\psi^v := 0$ **to** $\beta^v - 1$ **do**
            **for** $\phi := 0$ **to** $\alpha - 1$ **do**
                $c(\alpha\ell + \phi, \gamma m + \nu) \leftarrow c(\alpha\ell + \phi, \gamma m + \nu) + a(\alpha\ell + \phi, \psi^v) \times b(\psi^v, \gamma m + \nu)$
            **endfor** $\phi$
        **endfor** $\psi^v$
    **endfor** $\nu$
    **for** $\hat{Q}$:=1 **to** $\frac{N_r N_c}{\gcd(N_r,N_c)} - 1$ **do**
        **if** $\hat{Q} \bmod \frac{N_r}{\gcd(N_r,N_c)} = 0$ **then**
            **forall** $\phi, \psi \in \{0,1,\ldots,\alpha - 1\} \times \{0,1,\ldots,\beta_c - 1\}$ **do**
                $a(\alpha\ell + \phi, \beta_c m + \psi) \leftarrow a(\alpha\ell + \phi, \beta_c((m+1) \bmod N_c) + \psi)$
            **endforall** $\phi, \psi$
        **endif**
        **if** $\hat{Q} \bmod \frac{N_c}{\gcd(N_r,N_c)} = 0$ **then**
            **forall** $\mu, \nu \in \{0,1,\ldots,\beta_r - 1\} \times \{0,1,\ldots,\gamma - 1\}$ **do**
                $b(\beta_r\ell + \mu, \gamma m + \nu) \leftarrow b(\beta_r((\ell+1) \bmod N_r) + \mu, \gamma m + \nu)$
            **endforall** $\mu, \nu$
        **endif**
        **for** $\nu := 0$ **to** $\gamma - 1$ **do**
            **for** $\psi^v := 0$ **to** $\beta^v - 1$ **do**
                **for** $\phi := 0$ **to** $\alpha - 1$ **do**
                    $c(\alpha\ell + \phi, \gamma m + \nu) \leftarrow c(\alpha\ell + \phi, \gamma m + \nu) +$
                    $+a(\alpha\ell + \phi, \beta^v\hat{Q} \bmod \frac{N_r}{\gcd(N_r,N_c)} + \psi^v) \times b(\beta^v\hat{Q} \bmod \frac{N_c}{\gcd(N_r,N_c)} + \psi^v, \gamma m + \nu)$
                **endfor** $\phi$
            **endfor** $\psi^v$
        **endfor** $\nu$
    **endfor** $\hat{Q}$
**endforall** $\ell, m$
```