

1 Matrix–Vector Multiplication

Several scenarios are possible depending upon whether or not the data is external or internal to the array. If the data is external to the array, then a few temporary variables may suffice.

1.1 Matrix–vector multiplication on a linear array

1.1.1 Data external to the array

Systolic computations

If all the data must be entered into the array at the end points, and the result similarly delivered through the end points, the time is clearly proportional to $O(PQ)$ for the multiplication of a $P \times Q$ matrix by a Q vector. This bound is no better than a single processor.

In many *systolic* algorithms for matrix–vector multiplication, it is assumed that all the nodes of a linear array can input data. Systolic algorithms operate in a two–phase mode: compute, communicate, compute, communicate, etc. Systolic arrays, the hardware implementation of systolic algorithms, also have this characteristic. The arrays are of a *fine grain* in that the computations performed by each node are limited to operations on one set of operands (or possibly a few sets of operands). In systolic arrays, and in general in architectures of fine granularity, each operation has a very low overhead. Thus, for instance, sending a data item to one neighbor may indeed be equivalent to a register transfer operation [17]. In architectures of a *coarse grain*, communicating with another processor often incurs a measurable overhead due to address computation, routing and buffering. In our discussion of matrix–vector multiplication algorithms we will only consider the systolic model. The time to perform an arithmetic operation is t_a , and the time to perform a communication of one element with a neighbor node is assumed to be t_c . Moreover, we assume that a node can perform either a multiplication or an addition at a time.

Algorithm 1

For matrix–vector multiplication we assume that node i receives column $A(:, i)$ for the multiplication $y \leftarrow Ax$. Then, node i also requires $x(i)$, while y must be accumulated through communication between nodes.

Let $A(k, i)$ be entered into node i at time $A(k + i, i)$. Furthermore, let $temp(k, i) = temp(k, i - 1) + A(k, i) \times x(i)$, with $temp(k, -1) = 0$. Then, $temp(k, Q - 1) = y(k)$. Figure 1 shows the arrangement.

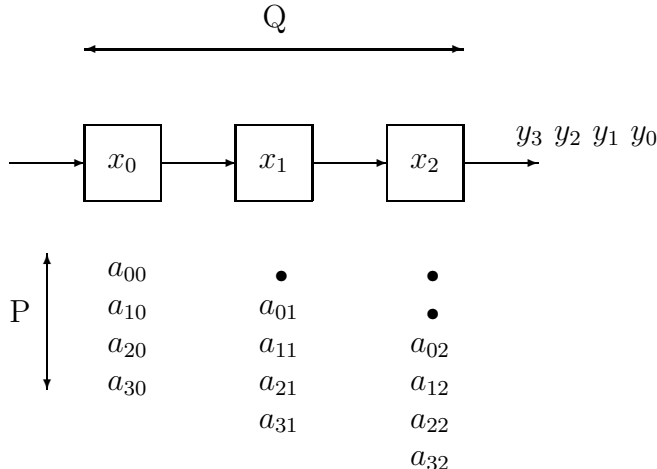


Figure 1: Multiplication of a 4×3 matrix on a linear array. Each node accepts a matrix column as input.

The required time is proportional to $P + Q - 1$, and the area is proportional to the number of nodes Q , since the memory per node is $O(1)$. In fine grain architectures, like systolic arrays, the overhead for most operations are ignored. An arithmetic operation is assumed to consume a time t_a and the exchange of a pair of elements between adjacent nodes is assumed to require a time t_c . In more coarse grained architectures, each communication is typically associated with an overhead and a time for message transmission that depends upon the number of bytes being sent. Thus, sending B bytes require a time $\tau + Bt_c$ in this model. For very large messages, the required buffer space may not be sufficient to hold the entire message in a packet oriented communications model. Then, in fact, the communication time may be better described by $\lceil \frac{B}{M} \rceil (\tau + Bt_c)$, where M is the buffer size. Furthermore, in a packet switching system, the cost to send a message between two nodes is proportional to the distance between the nodes in absence of congestion.

Using the systolic timing model, we arrive at the following estimates for the matrix–vector multiplication algorithm above:

$$\text{The required time is } (1 + 2(Q - 1) + 2(P - 1))t_a + (P + Q - 1)t_c = (2(P + Q) - 3)t_a + (P + Q - 1)t_c.$$

The first entry in the first term represents the single multiplication upon entering the first element of the matrix into the array. The second entry represents the remaining arithmetic operations on the first row of the matrix as the contributions to the first component of the result vector is computed. The third entry represents the additional time required for the rightmost node to compute remaining components of the vector y .

$$\text{The speedup is } \frac{2PQt_a}{(2(P+Q)-3)t_a + (P+Q-1)t_c}.$$

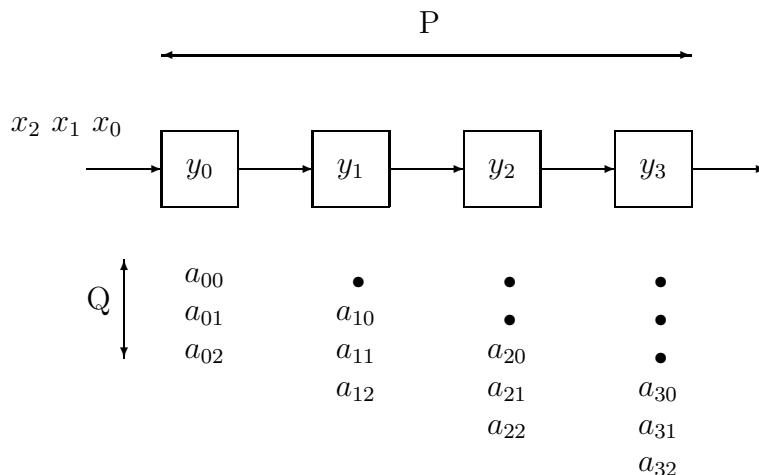


Figure 2: Multiplication of a 4×3 matrix on a linear array. Each node accepts a matrix row for input.

The efficiency is $\frac{2Pt_a}{(2(P+Q)-3)t_a+(P+Q-1)t_c}$.

For $P \gg Q$ and $t_a \approx t_c$, the speedup is $\sim \frac{2}{3}Q$, and the efficiency is $\sim \frac{2}{3}$. For $t_c \ll t_a$, the efficiency approaches 1. On the other hand, for $t_c \gg t_a$, the efficiency approaches $\sim \frac{2}{3} \frac{t_a}{t_c}$, which may be relatively small. Thus, we conclude that the efficiency is very sensitive to the ratio between the communication and computation times. Furthermore, we see that for a relatively small communication time, the speedup is close to ideal. Most parallel processors are not systolic. For most commercial systems the ratio t_a/t_c is in the range 0.1 – 0.01.

However, for $P \ll Q$ and $t_a \approx t_c$, the speedup is $\approx \frac{2}{3}P$ and the efficiency $\approx \frac{2}{3} \frac{P}{Q}$, which is small. The propagation time from input to output is large compared to the sequential time in each node, i.e., most processors in the array are idle most of the time.

For $P = Q$ and $t_a = t_c$, the speedup is $\sim \frac{1}{3}$.

Thus, we conclude that it is desirable to find a different arrangement of input data when $P \ll Q$.

Algorithm 2

Let a row of A be input to a single processor. Then, by passing x from one end of the array to the other, say left to right, and by skewing the row entries similar to the skewing of columns in the algorithm above, an algorithm that requires the same time using P processors can be constructed. Figure 2 shows the arrangement.

Each node computes an inner product of the vector x and a unique row of the matrix A . Node i accepts row $A(i)$ and the entire vector x as input. All nodes must receive every element of the vector x . Instead of broadcasting the vector, pipelining is used to distribute all elements

of x to every node. A direct broadcast would require either a bus, or an explicit broadcast network, say in the form of a tree. Either of these solutions will have relatively long wires, which may limit the clock rate of such a solution. However, pipelining introduces a delay between the computations in the different units. A given component of x visits the nodes in order during successive time steps. The duration of a time step is equal to a communication and an addition and a multiplication, assuming that there is only sufficient buffer memory to store one component of x in each node. The delay in the computations between different units is shown by filled circles in Figure 2.

With the algorithm implied by Figure 2, we have the following estimate for the time required using the systolic model:

The required time is $(2(P + Q) - 3)t_a + (P + Q - 1)t_c$.

The speedup is $\frac{2PQt_a}{(2(P+Q)-3)t_a+(P+Q-1)t_c}$.

The efficiency is $\frac{2Qt_a}{(2(P+Q)-3)t_a+(P+Q-1)t_c}$.

We notice that for $P \ll Q$ and $t_a \approx t_c$, the speedup is $\sim \frac{2}{3}P$, and the efficiency is $\sim \frac{2}{3}$. The propagation time from input to output is negligible compared to the sequential time in each node. Thus reorganizing the computations, and reducing the parallelism, improved the efficiency. As for the previous systolic array, the efficiency approaches 1 as $t_c/t_a \rightarrow 0$.

If $P \gg Q$ and $t_a \approx t_c$, then the speedup is $\sim \frac{2}{3}Q$. The efficiency is $\sim \frac{2}{3}\frac{Q}{P}$, which for $P \gg Q$ is poor. In this case, most of the array is idle most of the time.

With $P = Q$ and $t_a \approx t_c$, the efficiency is $\sim \frac{1}{3}$, just as for the previous array.

With $t_c = 10t_a$, which is not uncommon, the efficiency in the case $P \approx Q$ and the efficiency is 8.3%. If $Q \gg P$ then the efficiency is 16.7%.

Summary

We have given two algorithms for matrix–vector multiplication on a linear array with parallel input. Both algorithms may yield an efficiency of $O(1)$ depending on the shape of the matrix. In both cases a high efficiency is achieved when there is a substantial sequential component. For $P \gg Q$, the algorithm using Q nodes achieved high efficiency, while for $P \ll Q$ the algorithm using P nodes achieved the highest efficiency.

For both algorithms that we presented, poor efficiency was due to relatively long pipeline delays. These in turn were due to a need for global communication. In the first algorithm, every component of y was computed through an accumulation of products from all the nodes. In the second algorithm, every component of x was broadcast to every node. Because of the linear connections, the time for global operations are proportional to the number of nodes. Thus, just as in the case of sorting, the large diameter of the array limits the performance for problems with a small sequential component.

Thus, just as in the case of sorting, it is important to ask if there is a better organization of a given number of nodes for matrix–vector multiplication.

Another important question to ask is how can matrix–vector multiplication be performed on a linear array for other types of matrices, such as banded and sparse matrices. What are ideal

a_{00}	a_{01}	a_{02}	a_{03}	a_{04}	a_{05}	a_{06}
a_{10}	a_{11}	a_{12}	a_{13}	a_{14}	a_{15}	a_{16}
a_{20}	a_{21}	a_{22}	a_{23}	a_{24}	a_{25}	a_{26}
a_{30}	a_{31}	a_{32}	a_{33}	a_{34}	a_{35}	a_{36}

Figure 3: Partitioning of a matrix by columns.

processor arrangements for such matrices?

The systolic algorithms presented above have corresponding algorithms for the case where the data is already present in the array. But, before presenting such algorithms we will consider how to use a linear array where the number of nodes $N < \min(P, Q)$.

“Course-graining” the computations

If there are fewer nodes than columns, then in the first algorithm sets of N columns can be handled at a time. The result of each such computation is fed back into the array to achieve the total accumulation, i.e., the computation

$$y = Ax$$

is partitioned into the computation

$$y = \sum_{i=0}^{\lceil \frac{Q}{N} \rceil - 1} A_i x_i,$$

where

$$A_i = a(:, iN)a(:, iN + 1) \dots a(:, (i + 1)N - 1),$$

except possibly for the last block of columns. The partitioning of the matrix A is illustrated in Figure 3.

Another approach is to simply let each node emulate the work of a number of nodes. A *virtual* array in which there is one node per column in the algorithm for x stationary, or one node per row for y stationary, is mapped to the *physical* array. In effect, the matrix A would be blocked in the same way as above, with a $1 \times \lceil \frac{Q}{N} \rceil$ block of A entered into the array at a time for entire columns being entered into a node. For entire rows entered into a node, each node would receive a $\lceil \frac{P}{N} \rceil \times 1$ block of the matrix.

Note that in the first approach we outlined for the algorithm in which nodes receives entire columns, the partitioning in effect creates a *cyclic* mapping of columns to nodes. In the second approach, column index j is mapped to node $\lfloor \frac{j}{N} \rfloor$ in the physical array. Thus, the second strategy yields a *consecutive* or block mapping.

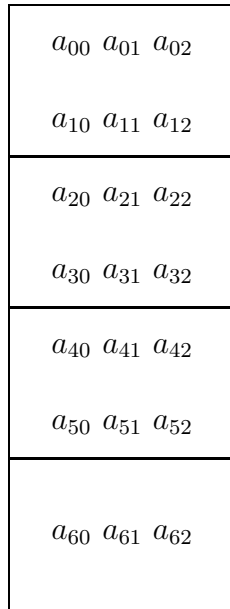


Figure 4: Partitioning of a matrix by rows.

Note also that when the axis length Q is not a multiple of the number of nodes N , then in the block approach, not all physical nodes have the same amount of data. On the Connection Machine systems CM-2 and CM-200, this is handled by assigning each node $\lceil \frac{P}{N} \rceil$ elements, marking extraneous elements as *garbage*.

In the algorithm accumulating the contributions to a component of y in-place with the vector x and the matrix A entering the array from outside, the matrix A is partitioned by rows. The partitioning of the matrix is shown in Figure 4.

With the matrix partitioned by rows and the array operating in a block mode, each node computes $\frac{P}{N}$ components of y . But, just as in the previous array, N components of y can be computed at a time, with a cyclic processing of row indices. Each block row has N matrix rows, one for each processing node. The number of block rows in this case is $\lceil \frac{P}{N} \rceil$.

1.1.2 Data internal to the array

The time required is naturally dependent upon the data organization prior to the computation, and the distribution of the result. In a general computing environment, the distribution may be determined based on array shape alone. An allocation dependent upon the usage of the data would require sophisticated analysis, which is currently not done.

Thus, we will assume that an array is distributed independent of the operations in which it participates. For x and y , which are linear arrays themselves, we will assume that they are evenly distributed among the nodes. For the matrix A we will consider both distribution by columns, as in the first systolic algorithm, and distribution by rows, as in the second systolic algorithm.

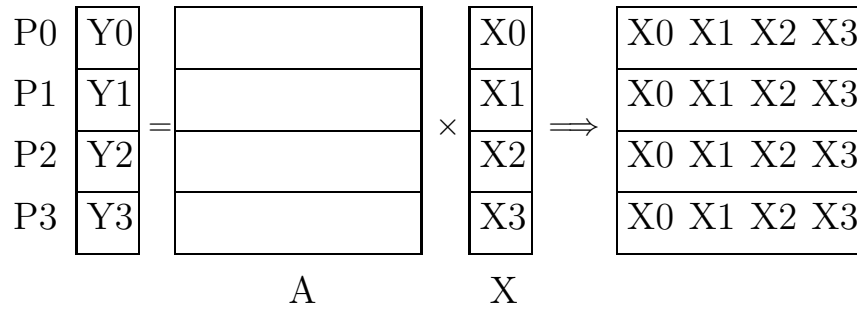


Figure 5: All-to-all broadcast for matrix-vector multiplication.

Considering the second algorithm first, i.e., the data distribution in which entire rows are allocated to a node, an *all-to-all broadcast* [14, 2, 20] of the vector x is required. In an all-to-all broadcast, each node sends all its elements to every other node. The operation is illustrated in Figure 5.

The algorithm can be formulated as

ALL-TO-ALL BROADCAST OF THE INPUT VECTOR.
 LOCAL MATRIX-VECTOR MULTIPLICATION.

Ignoring overhead, the time for the matrix-vector multiplication with the data internal to the array and partitioned by rows is $\frac{P}{N}(2Q - 1)t_a + \frac{Q}{N}(N - 1)t_c \approx \frac{P}{N}2Qt_a + Qt_c$. With both the x and y vectors distributed evenly across all nodes, communication is required to gather $N - 1$ segments of the vector x to each node. Each segment is of length $\frac{Q}{N}$. The gathering of segments can be accomplished through a sequence of cyclic shifts, as discussed later.

$$\text{Speedup: } \frac{2PQt_a}{\frac{P}{N}(2Q-1)t_a + \frac{Q}{N}(N-1)t_c}$$

$$\text{Efficiency: } \frac{2PQt_a}{P(2Q-1)t_a + Q(N-1)t_c} = \frac{1}{1 - \frac{1}{2Q} + \frac{(N-1)t_c}{2Pt_a}}$$

Since the matrix in the above algorithm is partitioned by rows, $P \geq N$. The efficiency is limited by the time required for the all-to-all broadcast. If $P \gg N$, then the efficiency is likely to be good for reasonable values of t_a and t_c . Q represents entirely sequential computations.

If instead entire columns are assigned to nodes, then no communication is required for x . However, just as in the systolic algorithm, partial contributions to the different components to y must be accumulated in space. We refer to the required reduction as *all-to-all reduction*, since contributions to each component of the result initially are distributed among all nodes, and at the end, each node contains some results. The idea of all-to-all reduction is shown in Figure 6.

With entire columns assigned to nodes, matrix-vector multiplication can be expressed as

LOCAL MATRIX-VECTOR MULTIPLICATION.
 ALL-TO-ALL REDUCTION FOR THE OUTPUT VECTOR.

The operations are illustrated in Figure 7. Ignoring overhead, the time for the matrix-vector multiplication with the data internal to the array and the matrix partitioned by columns is

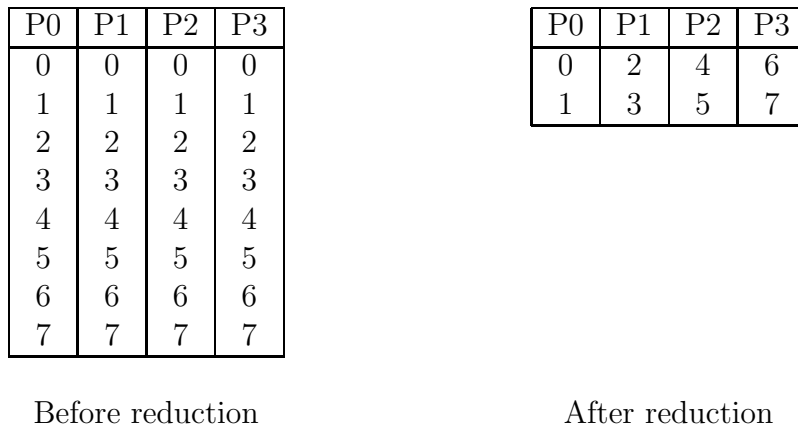


Figure 6: All-to-all reduction on a four node system.

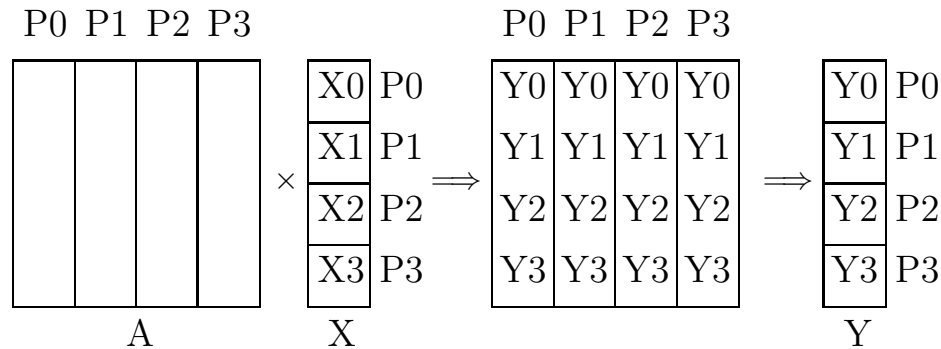


Figure 7: All-to-all reduction for matrix-vector multiplication.

$P(2\frac{Q}{N} - 1)t_a + \frac{P}{N}(N - 1)(t_a + t_c) \approx 2P\frac{Q}{N}t_a + Pt_c$. The communication time is due to all-to-all reduction resulting in N segments of the vector y , each of length $\frac{P}{N}$. After the all-to-all reduction each node owns a segment. The all-to-all reduction can also be accomplished through a sequence of cyclic shifts, as discussed later.

Speedup: $\frac{2PQt_a}{P(2\frac{Q}{N}-1)t_a + \frac{P}{N}(N-1)(t_a+t_c)}$

Efficiency: $\frac{2PQt_a}{(2Q-1)Pt_a + P(N-1)t_c} = \frac{1}{1 - \frac{1}{2Q} + \frac{(N-1)t_c}{2Qt_a}}$

Since the matrix is partitioned by columns, P represents entirely sequential computations. Moreover, $Q \geq N$. For $Q \gg N$, the efficiency is likely to be good for reasonable values of t_c and t_a .

In both algorithms A is stationary. The only communication is either all-to-all broadcast of x , or all-to-all reduction of y . The speedup of the arithmetic is proportional to N , the number of nodes, for either algorithm. The speedup is limited to P in the case the matrix is partitioned by rows, and by Q when the matrix is partitioned by columns.

Does there exist a distribution of matrix elements among the nodes of the linear array such that a lower communication time results?

Step	P0	P1	P2	P3
0	X0	X1	X2	X3
1	X0	X1	X2	X3
	X1	X2	X3	X0
2	X0	X1	X2	X3
	X1	X2	X3	X0
	X2	X3	X0	X1
3	X0	X1	X2	X3
	X1	X2	X3	X0
	X2	X3	X0	X1
	X3	X0	X1	X2

Figure 8: All-to-all broadcast through cyclic rotation.

It is interesting to note that for the matrix-vector multiplication algorithms with the data internal to the array, the communication time is bounded from below by the fan-in of the nodes, ignoring the overhead in the communications. For the all-to-all broadcast, all Q elements of x , except the elements initially allocated to a node must be gathered. For the all-to-all reduction, all P elements of the partial product sums of the vector y must be communicated to some other node, except the ones that reside in the local node.

We have only discussed dense matrix-vector multiplication on a linear array. How shall the elements of a banded matrix be distributed for minimum communication time and good load-balance? What is a good organization of sparse matrix-vector multiplication on a linear array?

Can the processing nodes be organized differently for improved performance?

With the entire matrix represented in memory the required area is $O(PQ)$. What is the minimum time for the multiplication of a matrix by a vector in area $O(PQ)$?

What area is required for matrix-vector multiplication in time $\log_2 Q$?

1.2 All-to-all broadcasts and all-to-all reductions

A lower bound for all-to-all broadcast of a vector of Q elements evenly distributed over N nodes clearly is $Q - Q/N$, since each node must receive this many elements, and there is only a single port (for the first and last processor). For a ring of processors, there is a potential improvement by a factor of two of this bound. For simplicity, we assume a ring for the all-to-all broadcast. A simple, yet optimal algorithm, is a cyclic shift of x . (If there is no wraparound, we can always simulate it, as illustrated above, with at most a constant slowdown.) If bidirectional communication is possible, we can use two cyclic shifts concurrently, one in the positive direction, one in the negative direction, each of which carries half of the data. The idea of cyclic shifts for all-to-all broadcast is illustrated in Figure 8. A corresponding all-to-all reduction algorithm is shown in Figure 9.

Step	P0	P1	P2	P3
0	Y0 Y1 Y2 Y3	Y0 Y1 Y2 Y3	Y0 Y1 Y2 Y3	Y0 Y1 Y2 Y3
1	Y0 Y1 Y2+Y2 –	– Y1 Y2 Y3+Y3	Y0+Y0 – Y2 Y3	Y0 Y1+Y1 – Y3
2	Y0 Y1+Y1+Y1 – –	– Y1 Y2+Y2+Y2	– – Y2 Y3+Y3+Y3	Y0+Y0+Y0 – – Y3
3	Y0+Y0+Y0+Y0 – – –	– Y1+Y1+Y1+Y1 –	– – Y2+Y2+Y2+Y2 –	– – – Y3+Y3+Y3+Y3

Figure 9: All-to-all reduction through cyclic rotation.

1.3 Matrix-vector multiplication on 2-D arrays

In considering matrix-vector multiplication on linear arrays we showed how to adopt the algorithm to the data organization prior to the computation, as well as after the computation, and showed the dependence of speedup and efficiency, and hence the preferred choice of algorithm with respect to performance, as a function of the matrix shape.

We will now consider matrix-vector multiplication on two-dimensional nodal arrays. If the matrix is external to the array, and data can only be entered along the boundary, then a linear array will be as fast as a two-dimensional array. Thus, we will only consider algorithms for the multiplication of a matrix by a vector on a two-dimensional array where the matrix and the vectors are already allocated within the array. Moreover, we will assume that $P, Q \geq N$, the total number of nodes in an array of shape $N_0 \times N_1$.

Before considering algorithms for the multiplication of a matrix by a vector, $y \leftarrow Ax$, we need to define the allocation of the three objects, A , x and y , to the memory units. One plausible allocation is that each array is distributed over all nodes, with the idea that parallelism will be maximized. For certain operations such an allocation may not be optimal, but determining an optimal allocation is likely to require sophisticated analysis of a program. Thus, we will first consider data allocations in which each operand is allocated across all nodes, then discuss other data layouts. For the vectors x and y , treating the set of nodes as a one dimensional array is natural, i.e., the vector is embedded in the two-dimensional nodal array according to the node numbers. The only decision to be made is which elements of the vector are assigned to a given node, when there are more vector elements than nodes.

We will assume a *consecutive (block)* assignment [8], i.e., a set of successive indices are assigned to each node, with some or all nodes receiving $\lceil \frac{Q}{N} \rceil$ elements. On the Connection Machine systems, if $Q \bmod N \neq 0$, then the first $\lfloor Q/\lceil \frac{Q}{N} \rceil \rfloor$ nodes receive $\lceil \frac{Q}{N} \rceil$ elements each, followed

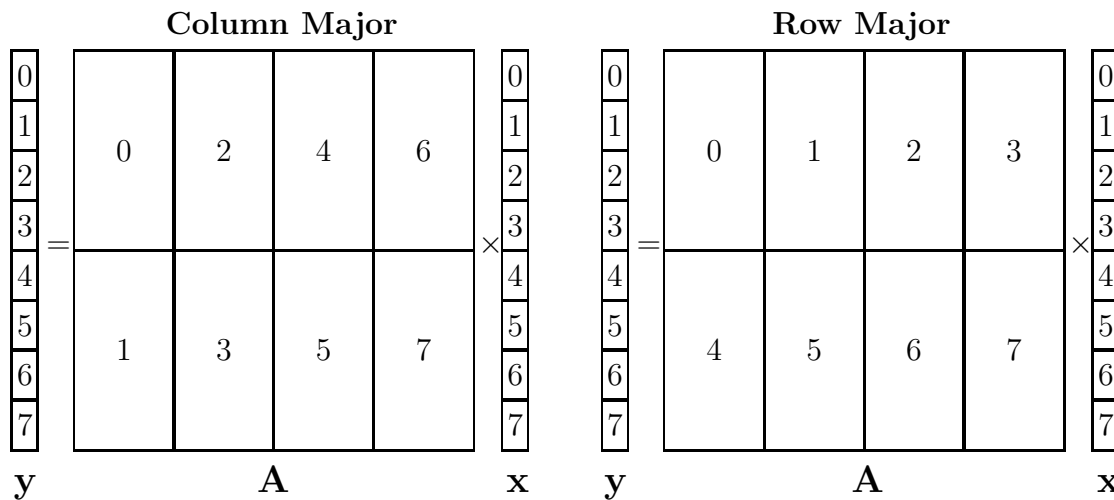


Figure 10: Data allocation on a rectangular nodal array.

by one node that receives $Q - \lceil \frac{Q}{N} \rceil \lfloor \frac{Q}{N} \rfloor$ elements, and possibly a few nodes receiving no elements. An alternate consecutive allocation strategy is to assign $\lceil \frac{Q}{N} \rceil$ elements to some nodes, and $\lfloor \frac{Q}{N} \rfloor$ to the remaining nodes.

For the matrix there is the additional choice, compared to the vectors, of whether the array shall be treated as a two-dimensional array or a linear array, as for the vectors. We will assume that the matrix A is allocated consecutively to a nodal array of shape $N_0 \times N_1$.

Figure 10 illustrates the data allocation of A , x and y for both row major and column major ordering of the matrix. (The data allocation shown in Figure 10 is typical on Connection Machine systems.)

In order to perform the matrix–vector multiplication, a node can only perform operations relevant for matrix–vector multiplication if the indices of the elements of x and the column indices of the elements of A are the same. Thus, an *alignment* of x and the columns of A is required. Similarly, an alignment of the row indices of A with the indices of y is required. Moreover, for a given column index of A , the corresponding element of x must interact with all elements in that column of A .

Since there are $P \times Q$ elements of A , we choose to move elements of x as required for the multiplication, and partial product sums as required for accumulation of y . Thus, an alignment followed by a spread is required for x , while for y a reduction followed by an alignment is required. But, in the column major ordering, the result of the alignment followed by the spread is an *all-to-all broadcast* within nodal columns. Similarly, in the row major ordering, the reduction followed by the alignment can be performed as *all-to-all reduction* within nodal rows.

Thus, in the column major ordering of the matrix, an all-to-all broadcast of x within nodal columns is followed by a local matrix–vector multiplication. After this operation, each node contains a segment of the result vector y . The nodes in a row contain partial contributions to the same segment of y , while different rows of nodes contain contributions to different segments of y .

No communication between rows of nodes is required for the computation of y . Communication within the rows of the nodes suffices.

The different segments of y can be computed by all-to-all reduction within nodal rows, resulting in a row major ordering of y . But, the node labeling is in column major ordering, and a reordering from row to column major ordering is required in order to establish the final allocation of y . Thus, for a column major ordering of the matrix elements to the nodes, matrix-vector multiplication can be expressed as:

ALL-TO-ALL BROADCAST OF THE INPUT VECTOR WITHIN COLUMNS OF NODES
 LOCAL MATRIX-VECTOR MULTIPLICATION
 ALL-TO-ALL REDUCTION WITHIN ROWS OF NODES TO ACCUMULATE
 PARTIAL CONTRIBUTIONS TO THE RESULT VECTOR
 REORDERING OF THE RESULT VECTOR FROM ROW MAJOR TO COLUMN MAJOR ORDER.

The reordering from row major ordering to column major ordering is a *shuffle*.

If the elements of the matrix A had been allocated in row major order instead of column major order, then a reordering from row major order to column major order must be performed prior to the all-to-all broadcast of the input vector. No reordering is required for y . Thus, for a row major ordering of matrix elements to nodes, the sequence of operations are:

REORDERING OF THE INPUT VECTOR FROM ROW MAJOR TO COLUMN MAJOR ORDER
 ALL-TO-ALL BROADCAST OF THE INPUT VECTOR WITHIN COLUMNS OF NODES
 LOCAL MATRIX-VECTOR MULTIPLICATION
 ALL-TO-ALL REDUCTION WITHIN ROWS OF NODES TO ACCUMULATE
 PARTIAL CONTRIBUTIONS TO THE RESULT VECTOR.

With the matrix uniformly distributed across all nodes, the arithmetic is load-balanced for both row major and column major order. The all-to-all broadcasts and all-to-all reductions are performed within the columns of the nodes and within the rows of the nodes, respectively. The different broadcast operations and the different reduction operations are completely independent of each other.

Thus, for the matrix-vector multiplication on a two-dimensional array, the required all-to-all communication is carried out on one-dimensional subarrays. We can use the algorithms described previously for linear arrays. The communication times are proportional to

All-to-all broadcast of x : $\sim \frac{Q}{N}(N_0 - 1)$;

All-to-all reduction for y : $\sim \frac{P}{N}(N_1 - 1)$.

In addition, a reordering from row to column major order is required. Ignoring the communication time for the reordering, the communication time is minimized when N_0 and N_1 are chosen such that the total time for all-to-all communication is minimized, i.e.,

$$\min_{N_0 \times N_1 = N} \left(\frac{Q}{N}(N_0 - 1) + \frac{P}{N}(N_1 - 1) \right)$$

The optimum value of $N_0 = \sqrt{\frac{PN}{Q}}$ and $N_1 = \sqrt{\frac{QN}{P}}$ and the minimum communication time is

$\sim 2\sqrt{\frac{PQ}{N}}$. Thus, ignoring the reordering time, for the optimum values of N_0 and N_1 , $\frac{N_0}{N_1} = \frac{P}{Q}$. For this shape of the nodal array, the matrix A has square submatrices assigned to each node.

Remark. The geometry manager in the Connection Machine Run-Time System attempts to configure the set of nodes such that all axis segments assigned to a node of an array is of the same length. Thus, ignoring the reordering from row to column major ordering, we conclude that the geometry manager indeed attempts to create the optimum nodal array shape for matrix-vector multiplication. Indeed, for binary cubes the reordering time is independent of N_0 and N_1 [16] and square submatrices are indeed optimal with respect to performance.

Comparing the above result with the results for one-dimensional arrays, we see that except possibly for very few nodes, a two-dimensional nodal array yields a considerably lower communication time than a one-dimensional array with the same number of nodes. With either array shape, the communication time is due to all-to-all communication, which is either proportional to $\sim Q$ or $\sim P$ for a one-dimensional array. Since we assumed that $P, Q \geq N$, $\sqrt{\frac{PQ}{N}} < Q$ and $\sqrt{\frac{PQ}{N}} < P$.

Is there another allocation of A to a two-dimensional nodal array that yields lower communication requirements?

The communication requirements for *vector-matrix* multiplication is very similar to those for matrix-vector multiplication.

1.3.1 Outer products

For *outer products*, yx^T , where y and x are column vectors, the communication issues for x are the same as in matrix-vector multiplication. For y , the communication issues are the same as for the input vector in vector-matrix multiplication. All-to-all broadcast and all-to-all reduction are also required in matrix-matrix multiplication [3, 4, 13, 19], as we will see later.

1.3.2 Reordering between row and column major ordering

Figure 11 shows row to column reordering on a 3×6 nodal array. The top part of the picture shows the initial row major ordering to the left, and the column major ordering to the right. The two arrays below show the linear representation of the row and column major orderings. The reordering can be viewed as a N_1 -way shuffle on the index set of size N , as illustrated by the lines between the two linear representations. The second array can be obtained from the first array by partitioning the first array into six segments, one for each column, and by partitioning the second array into three segments, one for each row. Then, the first segment of the second array is simply filled with the first entry of each of the segments of the first array, taken in order. The second segment of the the second array is filled with the second entry of each of the segments in the first array, etc.

On a square nodal array, the reordering from row to column major order is identical to matrix transposition. Thus, with $\frac{P}{N}$ elements per node, the reordering time is in the range $2\frac{P}{\sqrt{N}}$ to $\frac{P}{4\sqrt{N}}$ depending on whether a simple algorithm or an all-port, pipelined algorithm is used.

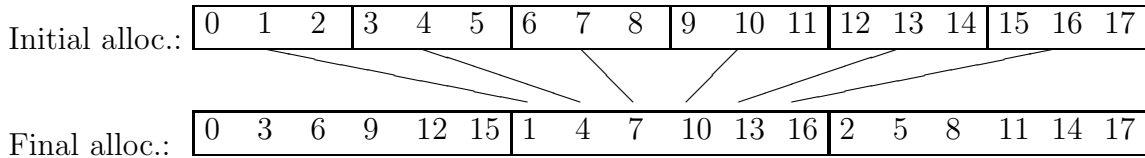
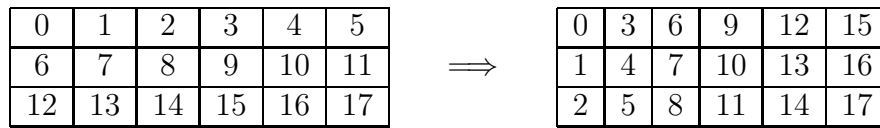


Figure 11: Reordering between row and column major ordering.

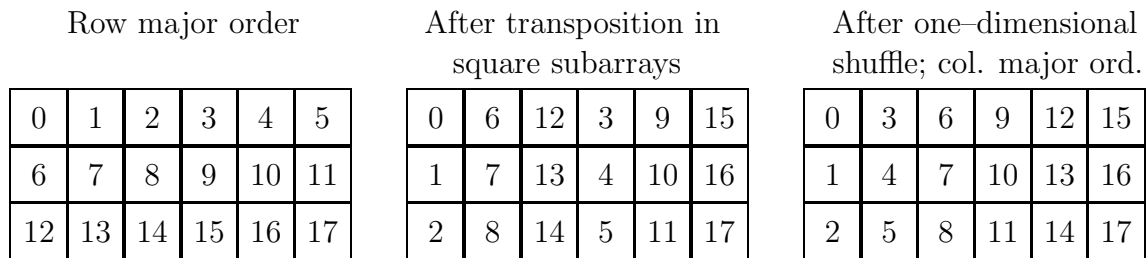


Figure 12: Reordering from row to column major order on a rectangular array.

If one axis is a multiple of another axis, then the reordering can be decomposed into a matrix transposition followed by a one-dimensional shuffle. The idea is illustrated in Figure 12.

What is a lower bound for shuffle operations on a two-dimensional array?

What is a good algorithm for row to column reordering on two-dimensional arrays with arbitrary axes lengths?

On binary cube networks, shuffle operations have the same complexity as matrix transposition [16].

2 Matrix–matrix multiplication

Algorithms for matrix–matrix multiplication on two-dimensional nodal arrays can be devised based on the linear array algorithms presented earlier by connecting together a number of linear arrays. We will consider two-dimensional array algorithms based on both of our linear array algorithms, i.e., we will consider two-dimensional array algorithms in which either the multiplier *or* multiplicand is fixed in space and the products accumulated by passing partial product sums in space, and algorithms in which the inner products are accumulated in-place. In the latter case, the product matrix is fixed in space, and both the multiplier and multiplicand move in space. For the multiplication $C \leftarrow A \times B$ we assume that C is of shape $P \times R$, A is

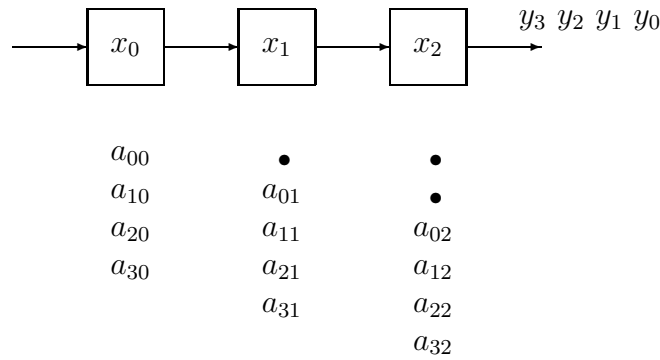


Figure 13: Multiplication of a 4×3 matrix on a linear array.

of shape $P \times Q$ and B of shape $Q \times R$.

2.1 Systolic algorithms for 2-D nodal arrays

Our first linear array algorithm for matrix–vector multiplication is shown again in Figure 13. The vector x for the multiplication $y \leftarrow Ax$ is distributed over the nodes with one component per node. The components of y are accumulated in space. For the matrix multiplication $C \leftarrow A \times B$, x corresponds to one column of B . Thus, for B of shape $Q \times R$, R linear arrays with Q nodes each is required in order to treat all columns of B concurrently. Figure 14 shows an array for the case where A is of shape 4×5 and B of shape 5×3 .

The matrix A is entered sheared by columns, just as in the matrix–vector multiplication case. The values of B are assigned to nodes in the array as if B was rotated 90–degrees, with columns aligned with rows of the array, and successive columns assigned to successive rows of the array. The result C is appearing in sheared by columns, with the first column being output by the bottom row, and the last column by the top row. The elements within a column appear in order of increasing row index. The number of steps required is

$$(1+2(Q-1+P-1+R-1))t_a + (P+Q-1+R-1+1)t_c = (2(P+Q+R)-5)t_a + (P+Q+R-1)t_c$$

The speedup is $\frac{2PQRt_a}{(2(P+Q+R)-5)t_a + (P+Q+R-1)t_c}$

and the efficiency is $\frac{2Pt_a}{(2(P+Q+R)-5)t_a + (P+Q+R-1)t_c}$.

With $t_a \approx t_c$ and $P \gg Q, R$, the efficiency is $\sim \frac{2}{3}$. It approaches 1 as $t_c/t_a \rightarrow 0$. For $Q \gg P, R$ and $t_a \approx t_c$, the efficiency is $\sim \frac{2P}{3Q}$, while for $R \gg P, Q$ the efficiency is $\sim \frac{2P}{3R}$. Thus, as for matrix–vector multiplication on a linear array, the efficiency is poor unless the serial component is dominating, i.e., $P \gg Q, R$ in the array above.

For $R \gg P, Q$, a two–dimensional array can be constructed from linear arrays for vector–matrix multiplication. In this case the two–dimensional array holds the matrix A , while the matrix B is entered on one side of the array, and the product C is output from a side perpendicular to

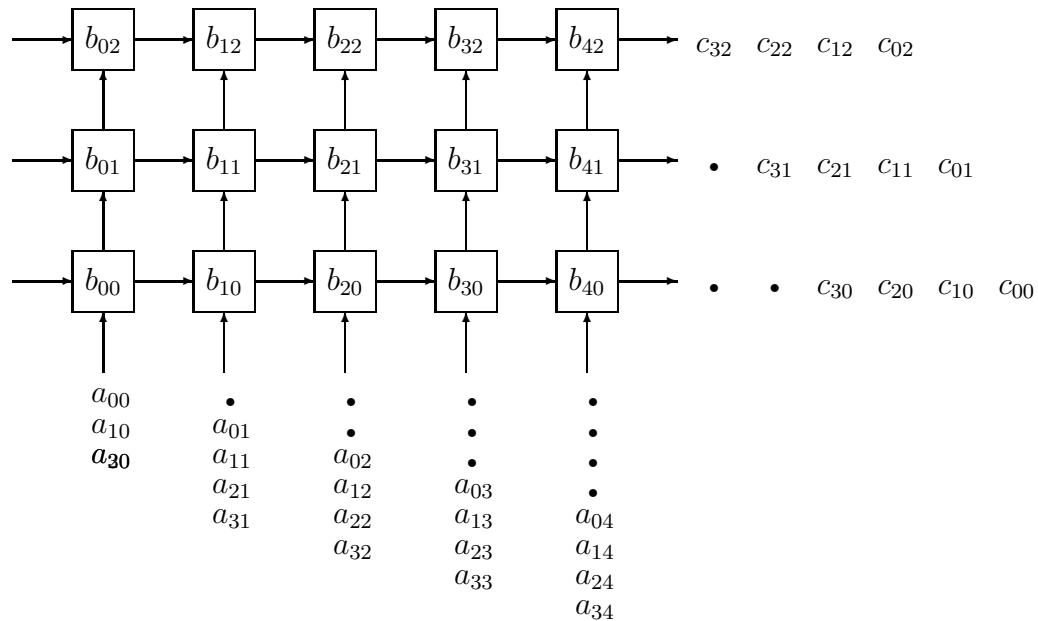


Figure 14: Multiplication of a 4×5 matrix by a 5×3 matrix on a 3×5 array.

the one on which B is entered. The arrangement is shown in Figure 15. A row of B is aligned with a column of the array. B is effectively being input transposed and sheared by rows. The rows of A appear in reverse order traversing the array from bottom to top. The rows of C appear in the same order as the rows of A , with the first column of C being output first. C is sheared by rows.

The time required is

$$(2(P + Q + R) - 5)t_a + (P + Q + R - 1)t_c$$

and the speedup is $\frac{2PQRt_a}{(2(P+Q+R)-5)t_a+(P+Q+R-1)t_c}$, i.e., the same as for the previous array. However, the efficiency is now

$$\frac{2Rt_a}{(2(P + Q + R) - 5)t_a + (P + Q + R - 1)t_c}$$

For $R \gg P, Q$ and $t_a \approx t_c$, the efficiency is $\sim \frac{2}{3}$. If $P \gg Q, R$ or $Q \gg P, R$ the efficiency is poor due to the pipeline delay.

For the first array the speedup was limited to QR , and for the second it was limited to PQ , with the efficiency being good in the first case if $P \gg Q, R$ and in the second case if $R \gg P, Q$. Poor efficiency is due to pipeline delays, and can be ignored if several problems shall be treated, one after the other.

A third possibility suitable for $Q \gg P, R$ is to accumulate the elements of the product matrix in-place. In the case of matrix-vector multiplication on a linear array with y accumulated in

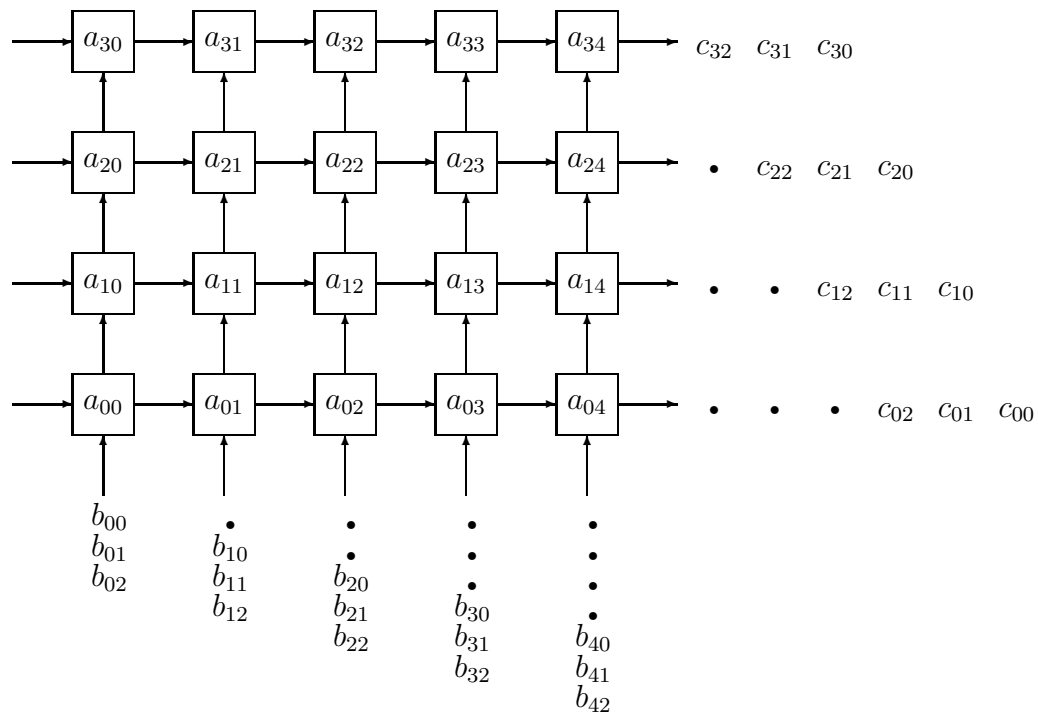


Figure 15: Multiplication of a 4×5 matrix by a 5×3 matrix on a 4×5 two-dimensional array.

place, the matrix A and the vector x were both entered into the array as shown in Figure 16. x was in fact passed through all of the elements of the array.

For matrix-matrix multiplication, $C \leftarrow A \times B$, extending this linear array algorithm to two-dimensional nodal arrays implies that there is one linear array for each column of B . A column of B must pass through each of the nodes of this linear array. Figure 17 shows a two-dimensional array in which the collection of linear arrays are organized such that the matrix C is laid out in normal row and column order. The linear arrays for the columns of B are laid out vertically such that columns of B are aligned with columns of the array (and columns of C).

The time and speedup is the same as for the previous arrays, i.e.,

$$(2(P + Q + R) - 5)t_a + (P + Q + R - 1)t_c$$

and

$$\frac{2PQRt_a}{(2(P + Q + R) - 5)t_a + (P + Q + R - 1)t_c},$$

respectively. However, the efficiency is now

$$\frac{2Qt_a}{(2(P + Q + R) - 5)t_a + (P + Q + R - 1)t_c}.$$

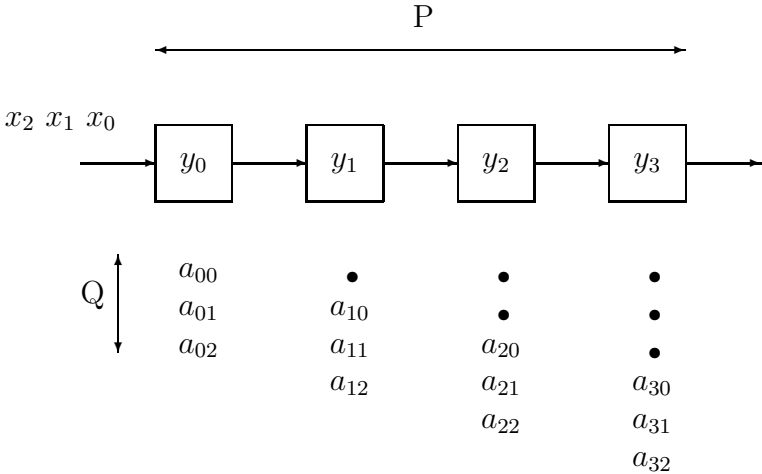


Figure 16: Multiplication of a 4×3 matrix on a linear array. Each node accepts a matrix row for input.

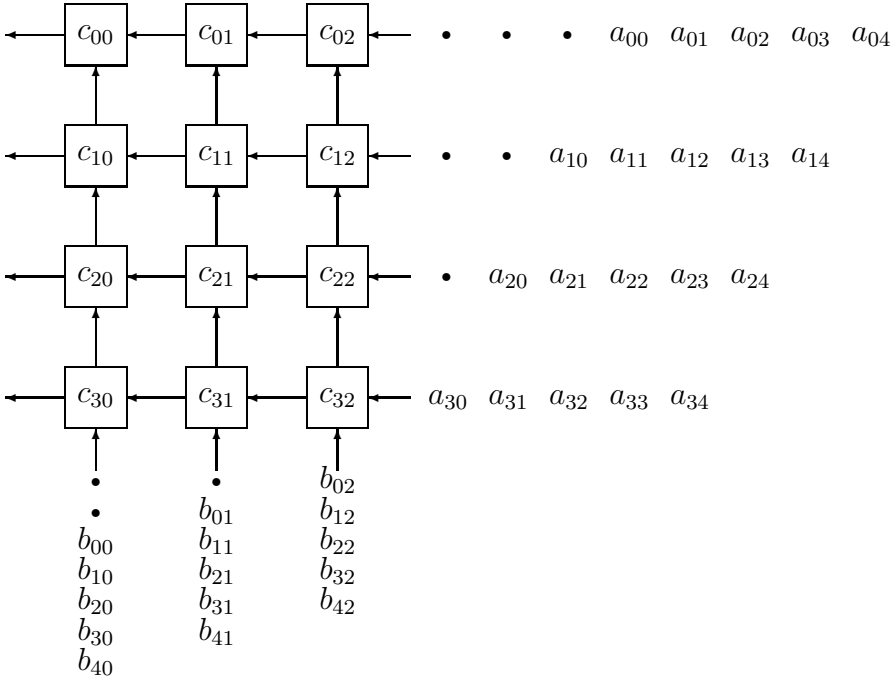


Figure 17: Multiplication of a 4×5 matrix by a 5×3 matrix on a 3×4 two-dimensional array.

Thus, if $t_a \approx t_c$ and $Q \gg P, R$, then the efficiency is $\sim \frac{2}{3}$ for an array with $P \times R$ nodes.

We have now given efficient algorithms for all three cases with respect to the longest axis, P , Q , or R , in matrix multiplication. The speedup is QR , PR , and PQ , respectively. The inefficiency is due to the pipeline delays. It can be diminished if several matrix–matrix multiplications shall be performed.

If there are fewer nodal columns than there are matrix columns of B and C , then several columns of B and C are mapped to the same nodal column. The shearing required between columns is performed with respect to nodal columns. Similarly, if there are fewer nodal rows than matrix rows, then several matrix rows may be mapped to the same nodal row. For instance, in the case of the last array, the submatrix of B that is entered into a node during a time step is of shape $1 \times \frac{R}{N_1}$, where N_1 is the number of nodal node columns. The submatrix of A , entered into a node during a time step, is of shape $\frac{P}{N_0} \times 1$. Hence, the shape of the submatrix of C that is updated in each step is $\frac{P}{N_0} \times \frac{R}{N_1}$. Q time steps are required. Both consecutive and cyclic mapping may be used. Note that if $\frac{P}{N_0} \neq \frac{R}{N_1}$, then the communication in the row and column directions are not balanced for all–port communication.

2.1.1 Algorithms for 2-D nodal arrays with matrices stored internally

The matrix–matrix multiplication algorithms described above have corresponding algorithms for the case where all arrays are stored internal to the array. The shearing of the operands in the systolic algorithms above must be performed as an *alignment* operation when the matrices are internal to the nodal array. A prealignment is required for operands that are used for input in the systolic algorithms, and a postalignment is required for operands that are computed by the systolic algorithms.

B stationary The systolic algorithm for B stationary is easily generalized for the multiplication of two $\sqrt{N} \times \sqrt{N}$ matrices on a two–dimensional nodal array of the same shape and size. The steps are illustrated in Figure 18. The steps are

```

Transpose  $A$ 
Align  $A$  through shearing by rows
Align  $C$  through shearing by columns
Compute the local products and add them to  $C$ 
For  $k = 1$  step 1 to  $\sqrt{N} - 1$  do
    Rotate  $A$  one step left
    Rotate  $C$  one step up
    Compute the local products and add them to  $C$ 
endfor
Align  $C$  with the array through shearing by columns.

```

Note that the rotation of A implements an all–to–all broadcast within nodal rows, while the rotation of C implements an all–to–all reduction within nodal columns.

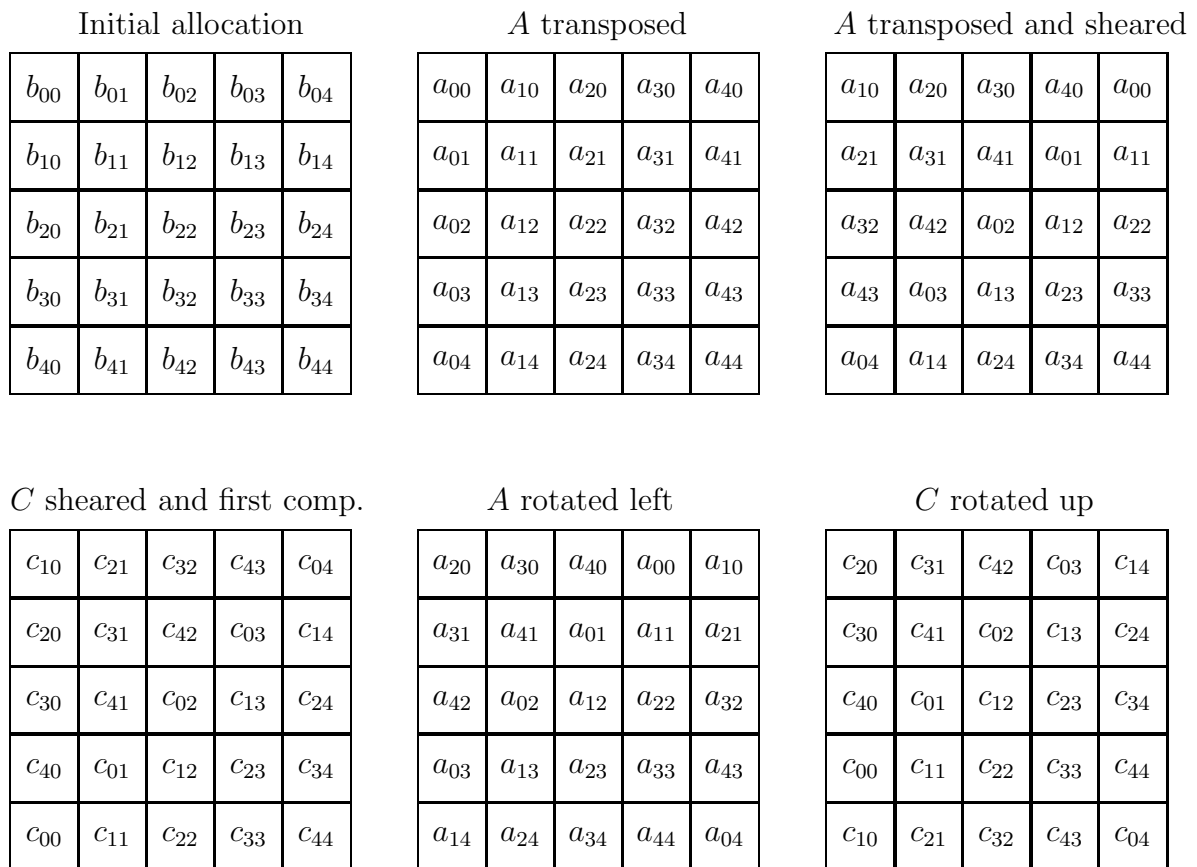


Figure 18: Multiplication of two square matrices on same size square nodal array with B stationary. The alignment of A and the first multiplication step are shown.

The algorithm can fairly easily be generalized to rectangular nodal arrays and large matrices, i.e., matrices for which $P, Q \geq N_0$ and $P, R \geq N_1$. The only issue in generalizing the algorithm to nonsquare nodal arrays is the fact that the P -axis is partitioned into N_1 segments for A , but N_0 segments for C . Yet, for the accumulation to take place the corresponding row indices of A and C must be present in a node at the same time. If N_1 is a multiple of N_0 , say $N_1 = kN_0$, then k rotation steps are performed along axis N_1 (A in Figure 18) for each rotation step along axis N_0 (C). That such a scheme yields the correct result can be seen by viewing the nodal array as an $N_1 \times N_1$ array by partitioning the subarrays of C in each node k times along the axis enumerating rows. For each shift along the axis of length N_1 one of the k local blocks along the axis of length N_0 can be updated. A different block is updated for each rotation step along the axis of length N_1 . In the case none of the axis is a multiple of the other, then the situation is more complex. We discuss this issue further for the algorithm with C stationary.

Assuming that $P, Q \geq N_0$ and $P, R \geq N_1$, we now will consider the optimal shape of the nodal array. The all-to-all communication for A requires a time proportional to $\frac{Q}{N_0} \frac{P}{N_1} (N_1 - 1) \approx \frac{PQ}{N_0}$. The shearing requires about half this time by shifting a row either in the positive or the negative direction. The all-to-all reduction of C requires a time proportional to $\frac{P}{N_0} \frac{R}{N_1} (N_0 - 1) \approx \frac{PR}{N_1}$. The shearing at the end requires about half this time. Thus, ignoring the initial transpose, the optimal array shape is determined by minimizing

$$\frac{3}{2} \left(\frac{PQ}{N_0} + \frac{PR}{N_1} \right)$$

The minimum is obtained for $N_0 = \sqrt{\frac{QN}{R}}$ and $N_1 = \sqrt{\frac{RN}{Q}}$. For these values of N_0 and N_1 , $\frac{N_0}{N_1} = \frac{Q}{R}$. Thus, ignoring the transposition of A , a nodal array shape such that B has a square submatrix in each node is optimal. The total communication time, again ignoring the transpose, is $3P\sqrt{\frac{QR}{N}}$.

Since the ideal nodal array shape for a transpose is a square, accounting for the transpose changes these estimates somewhat. It can be shown that with the transpose time included the ideal nodal array shape is $N_0 = \sqrt{\frac{5QN}{2Q+3R}}$ and $N_1 = \sqrt{\frac{(2Q+3R)N}{5Q}}$ with the ratio $\frac{N_0}{N_1} = \frac{5Q}{2Q+3R}$. The optimal ratio between the number of nodes along the two axes is now $\frac{5Q}{2Q+3R}$, and the communication time with the transposition on a mesh included is $P\sqrt{\frac{5Q(2Q+3R)}{N}}$ for the optimal mesh shape.

A stationary With A stationary instead of B , the systolic algorithms can be generalized in a way similar to the case with B stationary, as seen in Figure 19. It can be shown that ignoring the transposition, the optimal two-dimensional array shape is such that $\frac{N_0}{N_1} = \frac{P}{Q}$. The submatrix per node for A is square. The minimum communication time, ignoring the transposition, is $3R\sqrt{\frac{PQ}{N}}$, and for the optimal array shape with the transposition time included it is $R\sqrt{\frac{5Q(2Q+3P)}{N}}$.

Since all nodes participate equally in the computation, we can compare the algorithms by comparing the communication times. The ratio between the communication time TA for the

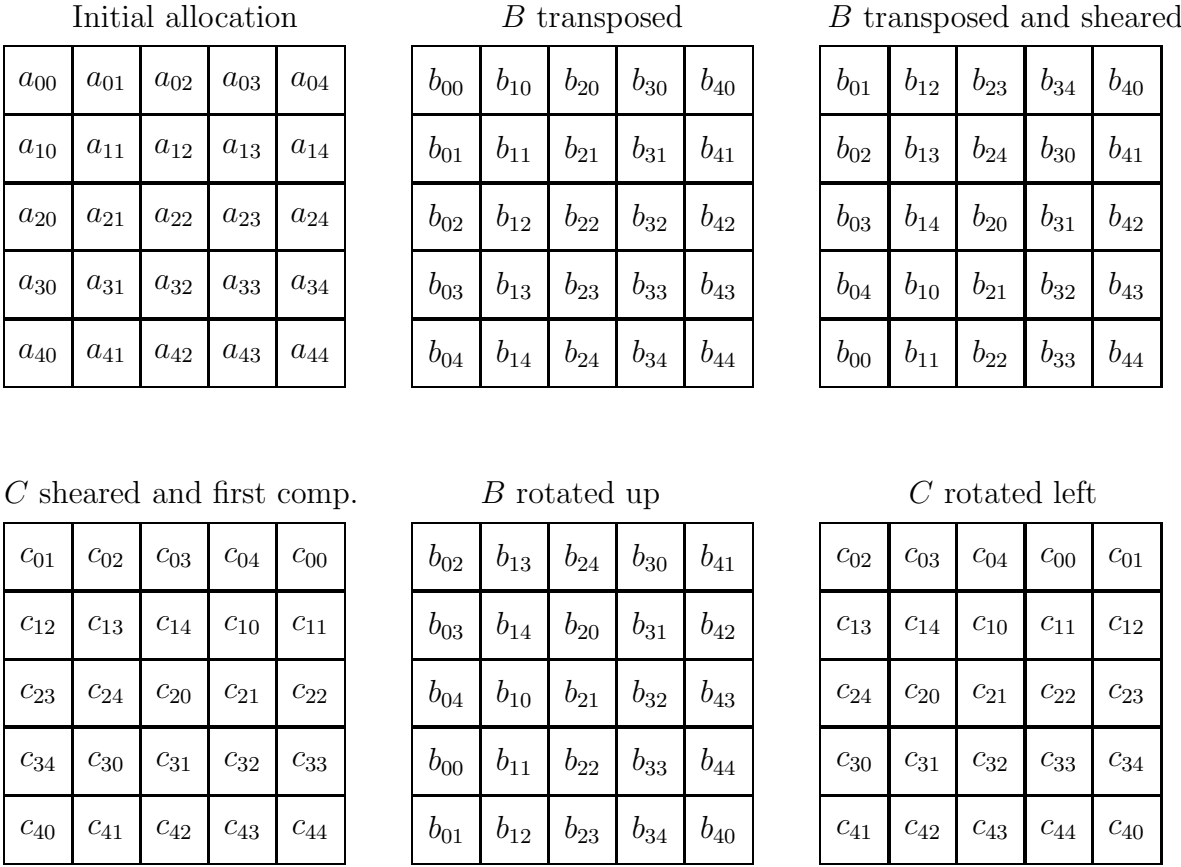


Figure 19: Multiplication of two square matrices on same size square nodal array with A stationary. The alignment of B and the first multiplication step are shown.

algorithm with A stationary and the communication time TB for the algorithm with B stationary is

$$\frac{TA}{TB} = \frac{R}{P} \sqrt{\frac{P}{R}} = \sqrt{\frac{R}{P}},$$

ignoring the transposition time. Including the transposition time and assuming optimal array shapes, the ratio is

$$\frac{TA}{TB} = \frac{R}{P} \sqrt{\frac{5Q(2Q + 3P)}{5Q(5Q + 3R)}} = \sqrt{\frac{R}{P}} \sqrt{\frac{\frac{2}{3}Q/P + 1}{\frac{2}{3}Q/R + 1}}.$$

Thus, whether we account for the transposition or not, if $P > R$, then the algorithm with A stationary shall be chosen, otherwise the algorithm with B stationary shall be chosen. Stated differently, of the two algorithms the one that keeps the largest matrix stationary shall be chosen. We will now see that this conclusion also holds with respect to algorithms with C stationary.

C stationary In this case, the inner axis, the Q -axis, is instantiated in time, while the P and R axis are partially or totally instantiated in space. We will now adapt the systolic algorithm for C stationary to an algorithm with all arrays internal to the array, and C stationary for the computations. We will assume that all arrays, i.e., A , B , and C are allocated to the two-dimensional array in the same way. Thus, the axis enumerating rows of a matrix is aligned with the axis enumerating rows of the two-dimensional array. Similarly, the axis enumerating columns of a matrix is aligned with the axis enumerating the columns of the two-dimensional array. Each node is assigned a submatrix of A of shape $\frac{P}{N_0} \times \frac{Q}{N_1}$, while the submatrix of B in a node is of shape $\frac{Q}{N_0} \times \frac{R}{N_1}$. The submatrix of C assigned to a node is of shape $\frac{P}{N_0} \times \frac{R}{N_1}$. Note that if $N_0 \neq N_1$, then the number of indices of the *inner axis* Q in a node is not the same for the multiplier and the multiplicand. Thus, since the inner indices must match for matrix multiplication, $c_{ij} = \sum_k a_{ik} b_{kj}$, the case $N_0 \neq N_1$ requires special attention.

Matrix multiplication on a square array, C stationary

For $N_0 = N_1$, the partitioning of the Q -axis is the same for A and B . The systolic algorithm for C stationary is easily generalized to the case with all matrices internal to the array:

1. Shear A by $\frac{Q}{N_1}$ columns for each nodal row.
2. Shear B by $\frac{Q}{N_0}$ columns for each nodal column.
3. Accumulate C by cyclic rotations of A within rows and of B within columns.
4. For each rotation, perform a local matrix–matrix multiplication.

Remark. Note that the cyclic rotation of A indeed implements an all-to-all broadcast of a row of A within the nodal array row to which it is assigned. Similarly, the cyclic rotation of B implements an all-to-all broadcast within nodal array columns.

What would the consequences be of performing the all-to-all broadcasts first, then local matrix-matrix multiplications?

In order to describe the algorithm more precisely, let $0 \leq i < P$, $0 \leq k < Q$, and $0 \leq j < R$ denote matrix element indices, and $0 \leq \ell < N_0$ and $0 \leq m < N_1$ denote indices for the nodal array elements. Since $N_0 = N_1$, the partitioning of the inner axis Q is the same for A and B . Assume that $P = \alpha N_0$, $Q = \beta N_0$, and $R = \gamma N_1$. An alignment such that node (ℓ, m) is assigned matrix elements:

$$\begin{aligned} A: & (\alpha\ell + \phi, \beta(\ell + m) \bmod Q + \chi), \text{ where } 0 \leq \phi < \alpha, 0 \leq \chi < \beta, \\ B: & (\beta(\ell + m) \bmod Q + \chi, \gamma m + \psi), \text{ where } 0 \leq \chi < \beta, \text{ and } 0 \leq \psi < \gamma, \\ C: & (\alpha\ell + \phi, \gamma m + \psi), \end{aligned}$$

ensures that the range of inner indices for A and B are identical on each node. 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 node.

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:

$$\begin{aligned} A: & (\alpha\ell + \phi, \beta(\ell + m) \bmod Q + \chi) \leftarrow (\alpha\ell + \phi, \beta(\ell + m + 1) \bmod Q + \chi), \text{ where } 0 \leq \phi < \alpha, \\ & 0 \leq \chi < \beta, \\ B: & (\beta(\ell + m) \bmod Q + \chi, \gamma m + \psi) \leftarrow (\beta(\ell + m + 1) \bmod Q + \chi, \gamma m + \psi), \text{ where } 0 \leq \chi < \beta, \\ & \text{and } 0 \leq \psi < \gamma. \end{aligned}$$

The shift operation must be repeated $N_0 - 1 = N_1 - 1$ times. Clearly, the inner indices of the two matrices are identical for each step of the algorithm, and 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 node.

The algorithm above [13, 11] is a block version of Cannon's matrix multiplication algorithm [3]. Cannon's algorithm assumes $P = Q = N_0$ and $Q = R = N_1$. For certain high degree networks, such as Boolean cubes, multiple exchange sequences can be used to make effective use of the communications bandwidth [7].

Remark 1. No local data motion is required between the cyclic shifts moving data between nodes. Emulating a large virtual nodal array naively on the physical array of shape $N_0 \times N_1$ would result in excessive local data motion. The data motion between the nodes would be the same. For instance, a naive algorithm would require $\frac{PQ}{N}Q$ local memory moves for A and $\frac{PQ}{N_0}$ element communications between adjacent nodes for A . Thus, $\frac{Q}{N_1}$ local memory moves per element communication would be required. Hence, even if local memory moves were 10 times faster than communication between adjacent nodes, local memory moves would require 10 times as much time as communication with a local axis segment of 10 along the Q -axis. Block algorithms can offer a substantial performance enhancement.

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.

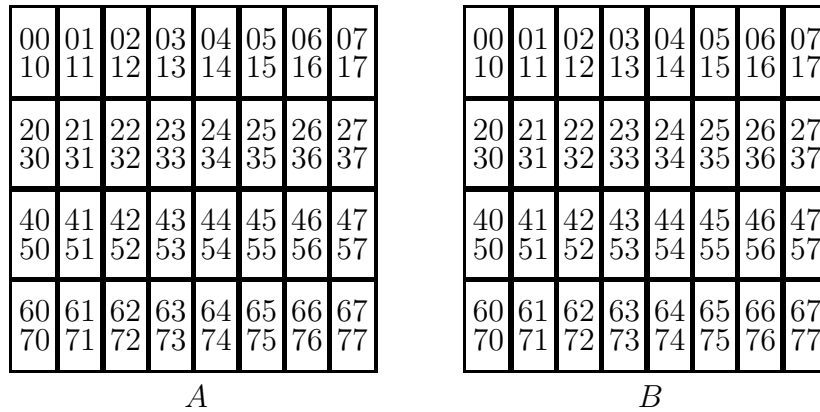


Figure 20: Allocation of 8×8 matrices to a 4×8 nodal array.

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 instance, for the Connection Machine systems CM-2 and 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 $N_0 \neq N_1$, this property is not true. This restriction is relaxed in the next section.

Matrix multiplication on rectangular nodal arrays, C stationary

Figure 20 shows the allocation of three square 8×8 matrices to a 4×8 nodal array. The length of the segment of the inner axis assigned to a node is *different* for A and B . Figure 21 shows the result of an alignment and the first two steps of the multiplication phase. For the example, in Figures 20 and 21, 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 nodal array columns and 4 nodal array rows. Figure 21 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.

In Figure 21, all operations requiring a given submatrix are carried out before the entire submatrix is moved to the adjacent node. No local data motion is required. When the inner axis extent per node is different for A and B , which is the case for a rectangular nodal 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 nodes assigned to one axis is a multiple of the number of nodes along the other axis, for example, $N_1 > N_0$, as in Figure 21, then $\frac{N_1}{N_0}$ rotation steps are performed along the longer

00	01	02	03	04	05	06	07
10	11	12	13	14	15	16	17
22	23	24	25	26	27	20	21
32	33	34	35	36	37	30	31
44	45	46	47	40	41	42	43
54	55	56	57	50	51	52	53
66	67	60	61	62	63	64	65
76	77	70	71	72	73	74	75

Matrix *A* aligned

00	11	22	33	44	55	66	77
10	21	32	43	54	65	76	07
20	31	42	53	64	75	06	17
30	41	52	63	74	05	16	27
40	51	62	73	04	15	26	37
50	61	72	03	14	25	36	47
60	71	02	13	24	35	46	57
70	01	12	23	34	45	56	67

Matrix *B* aligned

Rotate *A*, Multiply and Add

01	02	03	04	05	06	07	00
11	12	13	14	15	16	17	10
23	24	25	26	27	20	21	22
33	34	35	36	37	30	31	32
45	46	47	40	41	42	43	44
55	56	57	50	51	52	53	54
67	60	61	62	63	64	65	66
77	70	71	72	73	74	75	76

Matrix *A* shifted left 1 step

00	11	22	33	44	55	66	77
10	21	32	43	54	65	76	07
20	31	42	53	64	75	06	17
30	41	52	63	74	05	16	27
40	51	62	73	04	15	26	37
50	61	72	03	14	25	36	47
60	71	02	13	24	35	46	57
70	01	12	23	34	45	56	67

Matrix *B* aligned

Rotate *A* and *B*, Multiply and Add

02	03	04	05	06	07	00	01
12	13	14	15	16	17	10	11
24	25	26	27	20	21	22	23
34	35	36	37	30	31	32	33
46	47	40	41	42	43	44	45
56	57	50	51	52	53	54	55
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

Matrix *A* shifted left 2 steps

20	31	42	53	64	75	06	17
30	41	52	63	74	05	16	27
40	51	62	73	04	15	26	37
50	61	72	03	14	25	36	47
60	71	02	13	24	35	46	57
70	01	12	23	34	45	56	67
00	11	22	33	44	55	66	77
10	21	32	43	54	65	76	07

Matrix *B* shifted up 2 steps

Figure 21: Matrix multiplication on a 4×8 array.

axis for every rotation step along the shorter axis. A more general case is shown in Figures 22 and 23.

Let $P, Q \geq N_0$ and $Q, R \geq N_1$, and $N = N_0 \times N_1$, with no other restriction on N_0 and N_1 , and let $\beta_c = \frac{Q}{N_1}$ and $\beta_r = \frac{Q}{N_0}$. For arbitrary values of N_0 and N_1 , define a square *virtual* nodal array of shape $\frac{N_0 N_1}{\gcd(N_0, N_1)} \times \frac{N_0 N_1}{\gcd(N_0, N_1)}$. Let ℓ^v, m^v identify a block in the virtual array: $(\ell^v, m^v) \in \{0, 1, \dots, \frac{N_0 N_1}{\gcd(N_0, N_1)} - 1\} \times \{0, 1, \dots, \frac{N_0 N_1}{\gcd(N_0, N_1)} - 1\}$. Let $\beta^v = \lceil \frac{Q \gcd(N_0, N_1)}{N_0 N_1} \rceil$. After the alignment of the operands with respect to each other and the establishment of a shared nodal array shape, the index assignment for physical node (ℓ, m) is:

A: $(\alpha\ell + \phi, \beta^v((m \frac{N_0}{\gcd(N_0, N_1)} + \ell \frac{N_1}{\gcd(N_0, N_1)}) \bmod \frac{N_0 N_1}{\gcd(N_0, N_1)} + \chi) = (\alpha\ell + \phi, (\beta_c m + \beta_r \ell) \bmod Q + \chi)$, where $0 \leq \phi < \alpha$ and $0 \leq \chi < \beta_c$.

B: $(\beta^v((\ell \frac{N_1}{\gcd(N_0, N_1)} + m \frac{N_0}{\gcd(N_0, N_1)}) \bmod \frac{N_0 N_1}{\gcd(N_0, N_1)} + \chi, \gamma m + \psi) = ((\beta_r \ell + \beta_c m) \bmod Q + \chi, \gamma m + \psi)$, 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 nodes assuming the same nodal array configuration.

Clearly, for every node, 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 $\frac{N_0}{\gcd(N_0, N_1)}$, and along the inner axis of *B* is $\frac{N_1}{\gcd(N_0, N_1)}$.

For the multiplication phase, N_1 shifts are required for *A* and N_0 shifts for *B*. When $N_1 > N_0$, $\lfloor \frac{N_1}{N_0} \rfloor$ 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 $\lfloor \frac{N_1}{N_0} \rfloor + 1$, brings in the indices of *A* that correspond to the remaining inner indices of *B* (if N_1 is not a multiple of N_0) and some additional indices. A rank- $\bmod \frac{\beta_r}{\beta_c}$ update is performed first to consume all inner indices of *B*, followed by a move of *B* and a rank- $(\beta_c - \bmod \frac{\beta_r}{\beta_c})$, 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 $N_0 > N_1$, then *B* is moved more often, and a similar analysis applies. The index sets for the blocks each node receives are monotonically increasing, contiguous, and periodic.

Claim 1. *By performing the alignment along the longest axis as if the nodal array were square with the number of nodes along each axis equal to the number of nodes along the shortest axis, and the alignment along the shortest axis as if the nodal array were square with the number of nodes along each axis equal to the number of nodes 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 22 and 23 show an example where the greatest common divisor of N_0 and N_1 is 1.

00 01	02 03	04 05
10 11	12 13	14 15
20 21	22 23	24 25
30 31	32 33	34 35
40 41	42 43	44 45
50 51	52 53	54 55

Matrix *A*

00 01	02 03	04 05
10 11	12 13	14 15
20 21	22 23	24 25
30 31	32 33	34 35
40 41	42 43	44 45
50 51	52 53	54 55

Matrix *B*

Align

00 01	02 03	04 05
10 11	12 13	14 15
20 21	22 23	24 25
33 34	35 30	31 32
43 44	45 40	41 42
53 54	55 50	51 52

Matrix *A* aligned

00 01	22 23	44 45
10 11	32 33	54 55
20 21	42 43	04 05
30 31	52 53	14 15
40 41	02 03	24 25
50 51	12 13	34 35

Matrix *B* aligned

Rotate *A*, Multiply and Add

02 03	04 05	00 01
12 13	14 15	10 11
22 23	24 25	20 21
35 30	31 32	33 34
45 40	41 42	43 44
55 50	51 52	53 54

Matrix *A* left shifted 1 step

00 01	22 23	44 45
10 11	32 33	54 55
20 21	42 43	04 05
30 31	52 53	14 15
40 41	02 03	24 25
50 51	12 13	34 35

Matrix *B* aligned

Rotate *B*, Multiply and Add

02 03	04 05	00 01
12 13	14 15	10 11
22 23	24 25	20 21
35 30	31 32	33 34
45 40	41 42	43 44
55 50	51 52	53 54

Matrix *A* left shifted 1 step

30 31	52 53	14 15
40 41	02 03	24 25
50 51	12 13	34 35
00 01	22 23	44 45
10 11	32 33	54 55
20 21	42 43	04 05

Matrix *B* shifted up 1 step

Figure 22: Matrix multiplication on a 2×3 array.

Rotate A , Multiply and Add

04 05	00 01	02 03
14 15	10 11	12 13
24 25	20 21	22 23
31 32	33 34	35 30
41 42	43 44	45 40
51 52	53 54	55 50

Matrix A left shifted 2 steps

30 31	52 53	14 15
40 41	02 03	24 25
50 51	12 13	34 35
00 01	22 23	44 45
10 11	32 33	54 55
20 21	42 43	04 05

Matrix B shifted up 1 step

Figure 23: Matrix multiplication on a 2×3 array, last step.

Remark 4. There is no local data motion (except what is required by the local BLAS) as was the case for a square nodal array. Entire submatrices are moved between nodes 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 nodal 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. A complete overlap of the motion of A and B is impossible even for a square nodal array. If the lengths of the axes are relatively prime, then no communication is overlapped between A and B .

Choosing the optimum shape of the nodal array

Ignoring the effects of the ceiling functions, the speedup of the arithmetic operations is perfect and independent of the shape of the nodal array, assuming $P \geq N_0$, $R \geq N_1$, and $Q \geq N_0, N_1$. With the inner axis entirely instantiated in time, the matrix product requires $2 \lceil \frac{P}{N_0} \rceil \lceil \frac{R}{N_1} \rceil \lceil \frac{Q \gcd(N_0, N_1)}{N_0 N_1} \rceil \frac{N_0 N_1}{\gcd(N_0, N_1)}$ 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 $\lceil \frac{Q \gcd(N_0, N_1)}{N_0 N_1} \rceil$ updates, and the number of such updates in sequence.

The communication time for the multiplication phase is proportional to $\lceil \frac{P}{N_0} \rceil \lceil \frac{Q}{N_1} \rceil (N_1 - 1)$ for A and to $\lceil \frac{Q}{N_0} \rceil \lceil \frac{R}{N_1} \rceil (N_0 - 1)$ for B . The communication time is entirely due to the all-to-all broadcast. The alignment phase also requires cyclic rotations. By shifting a row of A either left or right, the number of shift steps for the alignment of A can be reduced to $\frac{N_1}{2}$. Similarly, the number of shifting steps for B is $\frac{N_0}{2}$. Thus, ignoring the ceiling functions, the communication time is minimized when

$$N_0 = \sqrt{\frac{PN}{R}}, \text{ and } N_1 = \sqrt{\frac{RN}{P}}$$

which implies that $\frac{N_0}{N_1} = \frac{P}{R}$. Thus, the optimum aspect ratio of the nodal array is the same as the aspect ratio of the product matrix. The optimum shape of the nodal array yields square submatrices for the product matrix, irrespective of the shapes of A and B . The optimal communication time is proportional to $3Q\sqrt{\frac{PR}{N}}$, accounting for alignment and all-to-all broadcast. The arithmetic time is proportional to $Q\sqrt{\frac{PR}{N}}$, and the speedup proportional to N . The efficiency is $O(1)$.

Remark. In our algorithm with C stationary no transposition is required, unlike in the algorithms with either A or B stationary. Only two all-to-all communications are required.

2.1.2 Nodal array reshaping

We have so far ignored the issue with the initial and final data allocation. In the algorithmic descriptions above it was implicitly assumed that the three operands, A , B , and C were all allocated to a nodal array of a given shape. Though this strategy is sensible for a fixed nodal array, attempting to minimize the communication by assigning as close to square subarrays as possible to each node is often a good strategy. In fact, we have seen that this is indeed optimal for the matrix that is stationary in the matrix multiplication algorithms above. If this strategy is applied, which is the case for the Connection Machine systems, then the nodal array shape for A , B , and C may all be different. Thus, establishing a shared nodal array shape prior to the computations, and returning operands according to the initial allocation, may also be required.

Reshaping the nodal array for an operand, if necessary, constitutes a *shuffle* operation. Figure 24 gives an example in which a 2×4 nodal 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, we introduce a second index for the partitioning of the matrix rows in each node. Then, the content of the first two nodal columns before the reshape operation consists of blocks 00, 01, 10, 11, 20, 21, 30 and 31, where the first index denotes the node to which the block of rows is assigned. After reshaping the nodal array, node 0 contains blocks 00 and 20, node 1 blocks 01 and 21, node 2 blocks 10 and 30, and node 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 a shuffle.

2.1.3 Comparing two-dimensional array implementations

In comparing the algorithms with A and B stationary we found that, omitting the time for the matrix transposition, the best choice of algorithm with respect to performance is one that keeps the largest matrix stationary. Comparing the algorithm with B stationary and the algorithm with C stationary the communication times have the ratio

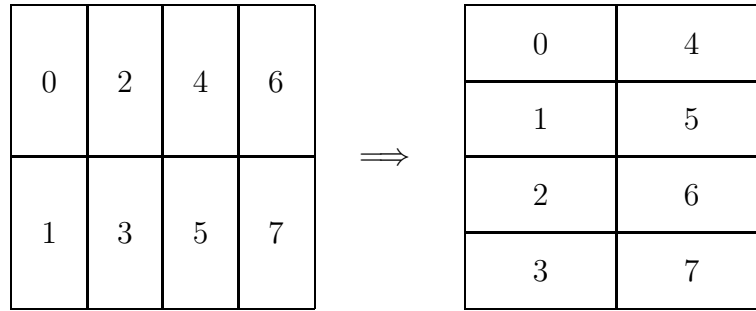


Figure 24: Reshaping a nodal array.

$$\frac{TB}{TC} = \frac{3P\sqrt{\frac{RQ}{N}}}{3Q\sqrt{\frac{PR}{N}}} = \sqrt{\frac{P}{Q}}$$

where we again have ignored the time for matrix transposition in the algorithm with B stationary. Thus, if $P > Q$ then the algorithm which keeps C stationary shall be selected. Note that if $P > Q$ then the size of C is greater than the size of B , since $PR > QR$. Hence, again, we find that the best choice of algorithm keeps the largest matrix stationary. We have

$$TA : TB : TC = \sqrt{R} : \sqrt{P} : \sqrt{Q}$$

2.1.4 Comments on a real implementation

The algorithms mentioned above are all used in the matrix multiplication routine in the Connection Machine Scientific Software Library, CMSSL [26]. In this implementation, the alignment phase is combined with establishing a shared nodal array shape for the three matrices. This operation is performed by the router. Even though the router is used for the operation it is of interest to predict the expected performance. With all-port communication on a binary cube, as in the case of the Connection Machine system CM-200, the optimum time for the reshaping is independent of the extent of the axes. The reshape time is proportional to the number of elements per node. Optimum algorithms are given in [15, 16]. For the alignment (shearing) operation, it is conjectured [8] that the data motion can be pipelined and hence that the communication time is proportional to the number of elements per node. Thus, it is expected that the time for the combined nodal array reshaping and matrix alignment is proportional to the size of the submatrices assigned to each node, and that the time is almost independent of the nodal array shape. Experience shows that the router exhibits such a behavior [19].

With all-port communication on a binary cube network, the communication time of all-to-all communication can be reduced by a factor of $\log_2 N_1$ for A and a factor of $\log_2 N_0$ for B [7, 14], compared to the times given for the two-dimensional mesh.

2.2 Matrix multiplication on three-dimensional mesh

The communication time for all-to-all broadcast and for reshaping and alignment depends upon the network configuration. The all-to-all broadcast and the alignment are both proportional to the amount of data in a row for A and a column for B . By partitioning the third matrix axes, the Q -axis into N_2 segments, and assigning the multiplication of a $P \times \frac{Q}{N_2}$ matrix by a $\frac{Q}{N_2} \times R$ matrix to a distinct plane of nodes, the all-to-all broadcast and alignment times are reduced by a factor of $\sqrt{N_2}$. (Recall that for C stationary, the communication time for all-to-all broadcast and alignment was proportional to $Q\sqrt{\frac{PR}{N}} = \frac{Q}{N_2}\sqrt{\frac{PR}{N/N_2}} = \frac{Q}{\sqrt{N_2}}\sqrt{\frac{PR}{N}}$.) However, splitting the inner axis (the Q -axis), creates the need for a reduction along the new axis. This all-to-all reduction requires a time proportional to $\frac{PR}{N/N_2}$. Thus, there is a tradeoff between the length of the different axes. In [10, 12] it is shown that depending upon the matrix shapes, one-dimensional, two-dimensional, or three-dimensional nodal arrays may be optimal. We will discuss matrix multiplication on three-dimensional arrays further below.

2.2.1 Algorithm description

With the nodes configured as a three-dimensional array, there are N_2 nodes assigned to the third axis, and N_0 and N_1 nodes assigned to the other two axes, as before. The total number of nodes is $N = N_0 \times N_1 \times N_2$. The structure of our matrix multiplication algorithm on a three-dimensional nodal array is as follows:

- Align the matrices A , B and C on the three-dimensional nodal array such that each plane contains corresponding subsets of the inner indices of A and B and all indices of C .
- Perform matrix multiplication concurrently and independently for all planes.
- Compute C by an all-to-all reduction between planes.

The possible advantage of a three-dimensional nodal array for matrices large enough to have one matrix element assigned to each node is a reduction in the communication time for the alignment and all-to-all broadcast [12, 13].

For a three-dimensional nodal array, the computations are parallelized with respect to all three index axes in matrix multiplication. An “inner-product” partitioning of A and B is used in parallelizing the computations associated with the inner index. Corresponding pairs of partitions are assigned to the same plane of nodes, while different pairs are assigned to different planes along the third array axis. Thus, each plane performs a matrix multiplication according to any of the algorithms presented for two-dimensional nodal arrays.

Figure 25 gives an example of reshaping a two-dimensional nodal array to a three-dimensional array, such that the algorithms devised earlier can be applied to each plane. In an actual implementation, the reshaping and prealignment would be combined [18]. The left half of the figure illustrates the allocation of A and B to an 8×8 nodal array. The three least significant bits of the node addresses are used to encode node column indices. The three most significant

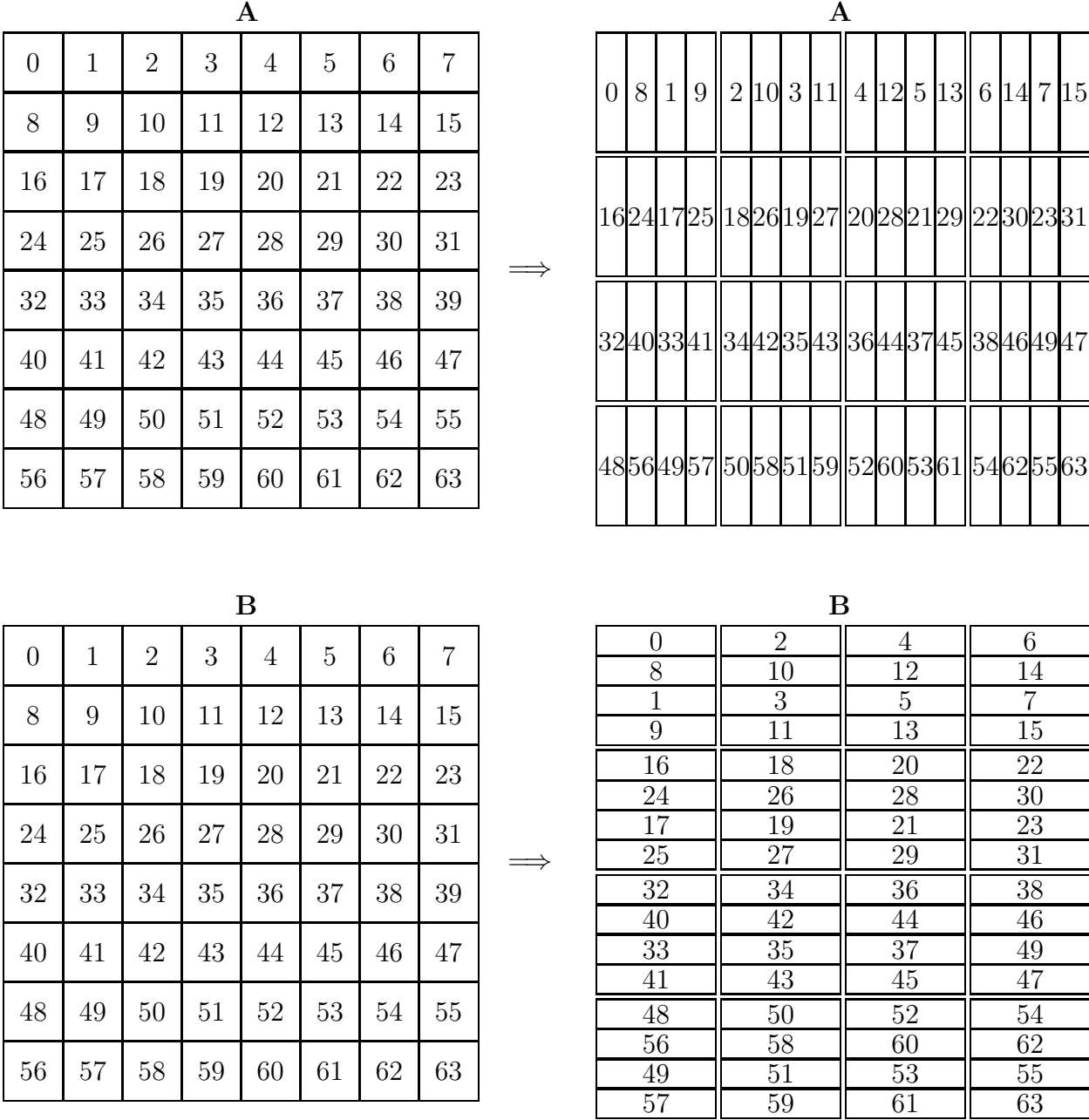


Figure 25: Reallocation of A and B to a three-dimensional nodal array.

bits are used to encode node row indices. The right half of Figure 25 shows the data allocation of A and B for nodes configured as a three-dimensional array with 4 planes, each of shape 4×4 . The least significant bit of the row encoding and of the column encoding in the two-dimensional nodal array are used for the third axis in the three-dimensional array.

In Figure 25, the third array axis is formed out of 2×2 nodal subarrays defined by abutting nodes. Reshaping of the nodal array such that the 2×2 subarrays form a single axis implies local reallocation of the submatrices assigned to the nodal subarrays. The reallocation is made such that the nodes along the third nodal array axis are aligned with the axis for the inner index, Q . Hence, the third nodal array axis is aligned with rows of A and columns of B . Moreover, the reallocation must be performed such that corresponding segments of the inner axis of A and B are allocated to the same node.

The reallocation of A corresponds to a block matrix transpose within columns of the nodal subarrays. For instance, in Figure 25, one block transposition is made between nodes 0 and 8 which form the first column in one of the subarrays. Another block transpose is made between nodes 1 and 9 which form the second column of the same subarray. The block transposition is performed similarly for the other nodal subarrays.

The reallocation for B can be accomplished by first performing a matrix transposition within the nodal subarrays used to form an instance of the third array axis. After this transposition within nodal subarrays, the orientation of the axis for the inner index is the same for B as it is in the initial allocation for A . Applying a block transpose to B identical to the one performed on A will generate the desired allocation. The two transpositions on B can be combined into a generalized shuffle permutation [16].

After the reallocation of A and B , each plane along the third nodal array axis contains the same set of inner indices for A and B . For instance, the submatrices of A assigned to the node set $X = \{0, 2, 4, 6, 16, 18, 20, 22, 32, 34, 36, 38, 48, 50, 52, 54\}$ form a partition of A consisting of all rows, and the columns c through $c + Q/16 - 1$ for $c \in \{0, Q/4, Q/2, 3Q/4\}$. The same set of nodes is also assigned submatrices of B forming a partition of B consisting of the same set of row indices, and all column indices of B . Thus, the node set X is assigned corresponding pairs of partitions.

After the multiplication phase is complete in each of the nodal subarrays along the third axis, each plane of nodes is assigned one product $A_i \times B_i$ for the matrix product $C \leftarrow \sum_{i=0}^{N_q-1} A_i \times B_i$. It remains to perform the addition of the products. Note that each plane stores a $P \times R$ array after the multiplication phase. For a third nodal array axis of length N_q , the temporary storage requirement is a factor of N_q higher than for the two-dimensional array configuration. For instance, if A and B are of shape 16×16 , then initially 2×2 submatrices of each operand are assigned to each node as shown in Figure 26. After the reallocation, the submatrix of A assigned to a node is of shape 4×1 , and the submatrix of B is of shape 1×4 , as shown in Figure 27. After the multiplication of partitions is complete within the subarrays, each node holds a 4×4 submatrix representing a partial contribution to C . With a third nodal array axis of length four, there are four such partial contributions to C . The addition of the partial contributions must be consistent with the allocation of C . The addition constitutes an all-to-all reduction [14, 20].

In summary, for a three-dimensional nodal array the operations are:

	000	001	010	011	100	101	110	111
000	0,0 0,1 1,0 1,1	0,2 0,3 1,2 1,3	0,4 0,5 1,4 1,5	0,6 0,7 1,6 1,7	0,8 0,9 1,8 1,9	0,10 0,11 1,10 1,11	0,12 0,13 1,12 1,13	0,14 0,15 1,14 1,15
001	2,0 2,1 3,0 3,1	2,2 2,3 3,2 3,3	2,4 2,5 3,4 3,5	2,6 2,7 3,6 3,7	2,8 2,9 3,8 3,9	2,10 2,11 3,10 3,11	2,12 2,13 3,12 3,13	2,14 2,15 3,14 3,15
010	4,0 4,1 5,0 5,1	4,2 4,3 5,2 5,3	4,4 4,5 5,4 5,5	4,6 4,7 5,6 5,7	4,8 4,9 5,8 5,9	4,10 4,11 5,10 5,11	4,12 4,13 5,12 5,13	4,14 4,15 5,14 5,15
011	6,0 6,1 7,0 7,1	6,2 6,3 7,2 7,3	6,4 6,5 7,4 7,5	6,6 6,7 7,6 7,7	6,8 6,9 7,8 7,9	6,10 6,11 7,10 7,11	6,12 6,13 7,12 7,13	6,14 6,15 7,14 7,15
100	8,0 8,1 9,0 9,1	8,2 8,3 9,2 9,3	8,4 8,5 9,4 9,5	8,6 8,7 9,6 9,7	8,8 8,9 9,8 9,9	8,10 8,11 9,10 9,11	8,12 8,13 9,12 9,13	8,14 8,15 9,14 9,15
101	10,0 10,1 11,0 11,1	10,2 10,3 11,2 11,3	10,4 10,5 11,4 11,5	10,6 10,7 11,6 11,7	10,8 10,9 11,8 11,9	10,10 10,11 11,10 11,11	10,12 10,13 11,12 11,13	10,14 10,15 11,14 11,15
110	12,0 12,1 13,0 13,1	12,2 12,3 13,2 13,3	12,4 12,5 13,4 13,5	12,6 12,7 13,6 13,7	12,8 12,9 13,8 13,9	12,10 12,11 13,10 13,11	12,12 12,13 13,12 13,13	12,14 12,15 13,14 13,15
111	14,0 14,1 15,0 15,1	14,2 14,3 15,2 15,3	14,4 14,5 15,4 15,5	14,6 14,7 15,6 15,7	14,8 14,9 15,8 15,9	14,10 14,11 15,10 15,11	14,12 14,13 15,12 15,13	14,14 14,15 15,14 15,15

Figure 26: Initial data allocation for A and B . Nodal row and column addresses are shown outside the array. Matrix indices are shown inside the array.

		A				B			
		000	010	100	110	000	010	100	110
000		0,0 2,0 1,0 3,0	0,4 2,4 1,4 3,4	0,8 2,8 1,8 3,8	0,12 2,12 1,12 3,12				
		4,0 6,0 5,0 7,0	4,4 6,4 5,4 7,4	4,8 6,8 5,8 7,8	4,12 6,12 5,12 7,12				
		8,0 10,0 9,0 11,0	8,4 10,4 9,4 11,4	8,8 10,8 9,8 11,8	8,12 10,12 9,12 11,12				
		12,0 14,0 13,0 15,0	12,4 14,4 13,4 15,4	12,8 14,8 13,8 15,8	12,12 14,12 13,12 15,12				
010		4,0 4,2 4,1 4,3	4,4 4,6 4,5 4,7	4,8 4,10 4,9 4,11	4,12 4,14 4,13 4,15				
		8,0 8,2 8,1 8,3	8,4 8,6 8,5 8,7	8,8 8,10 8,9 8,11	8,12 8,14 8,13 8,15				
		12,0 12,2 12,1 12,3	12,4 12,6 12,5 12,7	12,8 12,10 12,9 12,11	12,12 12,14 12,13 12,15				

Figure 27: One plane in the three-dimensional partitioning of A and B . Nodal row and column addresses are shown outside the array. Matrix indices are shown inside the array.

1. Perform a block matrix transposition on A within subarrays defining an instance of the third nodal array axis.
2. Perform a block generalized shuffle permutation on B within subarrays defining an instance of the third nodal array axis.
3. Perform matrix multiplication within subarrays defined by the first two nodal array axes.
4. Perform an all-to-all reduction along the third nodal array axis.

Compared to a two-dimensional nodal array, the three-dimensional array requires additional communication for adding partial results of C . The prealignment is different for the two array shapes, but the required optimal communication time is the same.

2.2.2 Complexity analysis

The complexity analysis below applies to binary cube networks. With all-port communication, the number of element transfers in sequence for the reallocation of A and B is independent of the length of the third nodal array axis. The number of element transfers in sequence for A is $\frac{PQ}{2N}$ and for B is $\frac{QR}{2N}$ [14]. Moreover, the reallocation of A can be combined with its alignment to form a generalized shuffle with bit-inversion [16, 21]. In bit-inversion a node sends its data to a node with an address obtained by complementing some of the bits in the sending node's address. Similarly, the reallocation of B and its alignment can be combined into a generalized shuffle with bit-inversion. The lower bound for a generalized shuffle with bit-inversion is the

same as the lower bound for bit-inversion. With no bit-inversion the lower bound for the generalized shuffle is the same as that of all-to-all personalized communication [14]. In all-to-all personalized communication, each node sends a unique message to every other node. Thus, the complexity estimates given for the prealignment of A and B to a shared two-dimensional nodal array shape also hold for the prealignment to a shared three-dimensional nodal array shape embedded in a binary-cube. For a three-dimensional nodal array, it may also be required to postalign C , which would require a time proportional to $\frac{PR}{2N}$.

The time for the all-to-all reduction is proportional to the length of the third nodal array axis, N_q , and inversely proportional to the logarithm of the length of this axis [14]. The number of element transfers in sequence is $\frac{PR}{N}(N_q - 1)$ [14]. The communication complexity for the multiplication phase is $\max(\frac{PQ}{N}(N_c - 1), \frac{QR}{N}(N_r - 1))$, as before.

The total number of element transfers in sequence for a three-dimensional nodal array is

$$T^{3D} \approx \max(P, R) \frac{Q}{N} + \frac{PR}{N}(N_q - 1) + \max(\frac{PQ}{N}(N_c - 1), \frac{QR}{N}(N_r - 1))$$

where we have assumed that the alignment and reallocation for A and B are combined into one permutation route. The communication complexity for the all-to-all reduction increases with N_q , while the communication complexity for the multiplication phase decreases with N_q , since $N_r N_c N_q = N$. The optimal length of the third nodal array axis is determined as a trade-off between the communication for the multiplication phase and the all-to-all reduction. Ignoring lower order terms, the optimal array shape is approximately: $N_r = (\frac{P^2 N}{2QR})^{\frac{1}{3}}$, $N_c = (\frac{4R^2 N}{PQ})^{\frac{1}{3}}$ and $N_q = (\frac{Q^2 N}{2PR})^{\frac{1}{3}}$. The corresponding number of element transfers in sequence is

$$T_{\min}^{3D} \approx \max(P, R) \frac{Q}{N} + 3 \left(\frac{PQR}{2N} \right)^{\frac{2}{3}}.$$

The optimal ratios $N_r : N_c : N_q$ are approximately $P : R : Q/2$. Similar results were reported in [12, 13]. A three-dimensional nodal array of optimal shape yields a lower communication complexity than a two-dimensional array of optimal shape when $N > \frac{3^6 RP}{16 Q^2}$. Asymptotically, the reduction is a factor of order $O(N^{\frac{1}{6}})$.

Theorem 1 *The multiplication of a $P \times Q$ matrix A with a $Q \times R$ matrix B , on a three-dimensional mesh of shape $N_r \times N_c \times N_q$, $P \geq N_r$, $R \geq N_c$, and $Q \geq N_q$ can be performed with at most $2 \frac{PQR}{N} + \frac{PRN_q}{N}$ arithmetic operations. The number of element transfers in sequence is at most $T_{\min}^{3D} \approx \max(P, R) \frac{Q}{N} + 3 \left(\frac{PQR}{2N} \right)^{\frac{2}{3}}$ for nodal array axes of lengths $N_r = (\frac{P^2 N}{2QR})^{\frac{1}{3}}$, $N_c = (\frac{4R^2 N}{PQ})^{\frac{1}{3}}$, and $N_q = (\frac{Q^2 N}{2PR})^{\frac{1}{3}}$.*

2.2.3 Load-balance

If at least one of the arrays is small compared to the nodal array, then load balance may be an issue. For instance, in multiplying a 6×6 matrix by a $6 \times 10,000$ matrix on say 100 nodes, the 6×6 matrix can clearly not be allocated across all nodes. Such cases are not uncommon

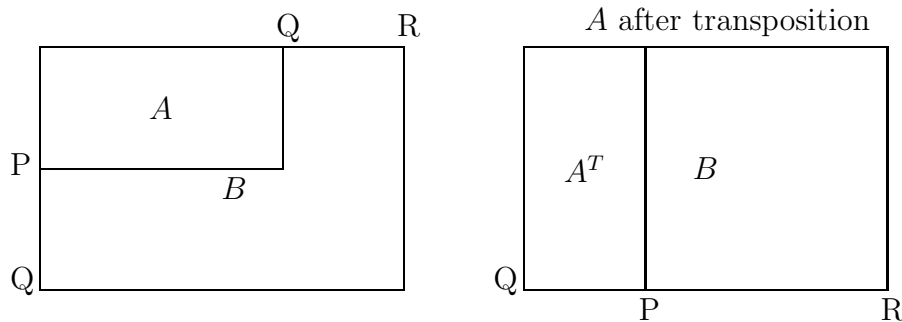


Figure 28: Matrix multiplication when B is large.

in practice. In this particular case, a replication of the small matrix such that all nodes have a copy of the complete matrix is likely to be the most effective algorithm, assuming an efficient broadcast algorithm can be found. A two-dimensional nodal array can broadcast in less time than a one-dimensional nodal array. And, a three-dimensional array is even better than a two-dimensional nodal array.

We will now consider matrix multiplication where load-balance is an issue in more detail. Let N_P , N_Q and N_R be the number of nodes over which the P -axis, the Q -axis and the R -axis are distributed, i.e., $N_P = \min(P, N_0)$, $N_Q^A = \min(Q, N_1)$, $N_Q^B = \min(Q, N_0)$ and $N_R = \min(R, N_1)$.

B large If $P < Q, R$, then A has the fewest elements and B the largest number of elements. In analogy with the cases where all matrices have at least one element per node, we consider algorithms keeping B stationary, while A and C are communicated as required. Since B is stationary, our first objective is to employ in the computation all $N_Q^B \times N_R$ nodes to which B is allocated. Our second goal is to use all N nodes, if $N_Q^B \times N_R < N$.

Figure 28 shows the initial allocation of A and B , and the allocation after the transposition of A . For the multiplication, A is rotated left and C is rotated up. After the transposition, A^T covers $N_P^{A^T}$ columns, a subset of the nodal columns to which B is allocated. However, the rotation of A^T must cover all N_R nodes to which columns of B are allocated, i.e., the all-to-all broadcast of the rows of A^T must cover all nodal columns to which B is allocated. Hence, the all-to-all communication time for A is no longer $\sim P \lceil \frac{Q}{N_Q^B} \rceil$, but $\sim \lceil \frac{P}{N_P^{A^T}} \rceil \lceil \frac{Q}{N_Q^B} \rceil N_R$. Thus, for $\frac{N_R}{N_P^{A^T}} > 1$, the all-to-all broadcast time may be significantly higher than what is determined by the number of elements received.

In order to reduce the all-to-all broadcast time for A^T to $\sim P \lceil \frac{Q}{N_Q^B} \rceil$, it is necessary to shorten the effective axis for the all-to-all broadcast. This can be done by replicating A^T along the R -axis $\frac{N_R}{N_P^{A^T}}$ times. However, having done so, we notice that the all-to-all reduction for C requires N_Q^B steps. Thus, if $N_Q^B > N_P^{A^T}$, then the all-to-all reduction time is proportional to $\sim Q \lceil \frac{R}{N_R} \rceil$, whereas if the all-to-all reduction would have been limited to a nodal axis of length $N_P^{A^T}$, then the all-to-all reduction time would be $\sim \lceil \frac{Q}{N_Q^B} \rceil \lceil \frac{R}{N_R} \rceil N_P^{A^T}$. Therefore, if $N_Q^B > N_P^{A^T}$ we partition the Q -axis into $k = \frac{N_Q^B}{N_P^{A^T}}$ partitions such that the matrix product can

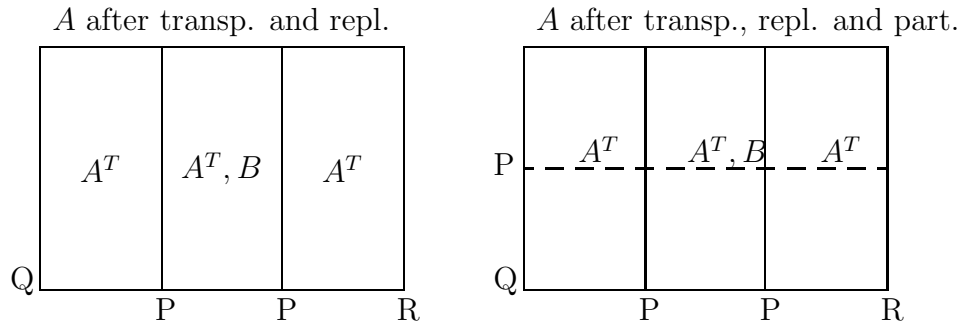


Figure 29: Matrix multiplication when B is large, replication and partitioning.

be expressed as $C = \sum_{i=1}^k A_i B_i$, where A_i is a $P \times \frac{Q}{k}$ matrix and B_i is a $\frac{Q}{k} \times R$ matrix, with the inner indices matching for a given i . Having made this partitioning of the inner axis, partial contributions to C are computed on a number of independent arrays, each of shape $N_P^{A^T} \times N_P^{A^T}$. The multiplication within each of these subarrays can now be made with the algorithm for B stationary, as described earlier. C is obtained through a reduction over the k partitions within nodal columns. The replication of A^T and partitioning of A^T and B are illustrated in Figure 29.

Thus, for B large compared to A and C we have the following algorithm

1. Transpose A
2. Replicate $A^T \frac{N_R}{N_P^{A^T}}$ times along the R -axis.
3. Partition the Q -axis into $\frac{N_Q^B}{N_P^{A^T}}$ partitions.
4. Perform matrix multiplication within $N_P^{A^T} \times N_P^{A^T}$ subarrays using the algorithms for B stationary, i.e.,
 - (a) Shear A^T by rows.
 - (b) Perform $N_P^{A^T}$ steps of left rotations of A^T and up rotations of C
 - (c) Shear C by columns
5. Compute C through “block” reduction within nodal columns.

Remark. Whether or not replication of A^T followed by all-to-all broadcast of A^T within row segments of length $N_P^{A^T}$ is faster than all-to-all broadcast within the entire nodal row of length N_R , depends upon whether or not the broadcast operation can be made faster than the corresponding steps of the all-to-all broadcast. Similarly, whether or not all-to-all reduction along axis segments followed by a reduction between segments is any faster than an all-to-all reduction along the entire axis depends upon the relative times for the operations. On some networks, like binary cubes, broadcast and reduction can be made faster than all-to-all communication.

If $N_Q^B < N_0$ and/or $N_R < N_1$, then there are nodal rows and columns that are not participating in an algorithm as outlined above. If $Q \times R > N$, then reshape the nodal array such that B is allocated to all nodes, then use the algorithm above. If $N > Q \times R$, then partition the Q -axis as discussed above, i.e., into $\frac{Q}{P}$ partitions with inner axis segments of length P each. Note however that this partitioning does not improve load-balance. For improved load-balance, partition the Q -axis further into $\frac{N}{QR}$ partitions for a total of $k = \frac{N}{PR}$ partitions. Each set of PR nodes receives a $\frac{Q}{k} \times P$ partition of A^T , and a $\frac{Q}{k} \times R$ partition of B . Note that for each such partition the Q -axis is the shortest, and an algorithm for $Q < P, R$ shall be used within each partition.

A large The case with A large is handled analogously to the case for B large. Figures 30 and 31 shows the transposition, replication and partitioning steps for B . After these steps, the algorithm presented earlier for A large can be used within each subarray of shape $N_R^{B^T} \times N_R^{B^T}$. We have the following algorithm

1. Transpose B
2. Replicate $B^T \frac{N_R^{B^T}}{N_P^A}$ times along the P -axis.
3. Partition the Q -axis into $\frac{N_Q^A}{N_R^{B^T}}$ partitions.
4. Perform matrix multiplication within $N_R^{B^T} \times N_R^{B^T}$ subarrays using the algorithms for A stationary, i.e.,
 - (a) Shear B^T by columns.
 - (b) Perform $N_R^{B^T}$ steps of up rotations of B^T and left rotations of C
 - (c) Shear C by rows
5. Compute C through “block” reduction within nodal rows.

If $N_Q^A < N_1$ and/or $N_P < N_0$, then there are nodal rows and columns that are not participating in an algorithm as outlined above. If $P \times Q > N$, then reshape the nodal array such that A is allocated to all nodes, then use the algorithm above. If $N > P \times Q$, then partition the Q -axis into $\frac{Q}{R}$ partitions, as discussed above. Note, however, just as in the case with B stationary, this partitioning of the Q -axis does not improve load-balance. In order to improve the load-balance, partition the Q -axis further into $\frac{N}{PQ}$ partitions, for a total of $k = \frac{N}{PR}$ partitions. Each set of PR nodes receives a $P \times \frac{Q}{k}$ partition of A , and a $R \times \frac{Q}{k}$ partition of B^T . For each such partition the Q -axis is the shortest, and an algorithm for $Q < P, R$ shall be used within each partition.

C large In the case with C large relative to A and B , i.e., $Q < P, R$, then A is replicated $\frac{N_R}{N_Q^A}$ times along the R -axis, and B is replicated $\frac{N_P}{N_Q^B}$ times along the P -axis, as shown in Figure

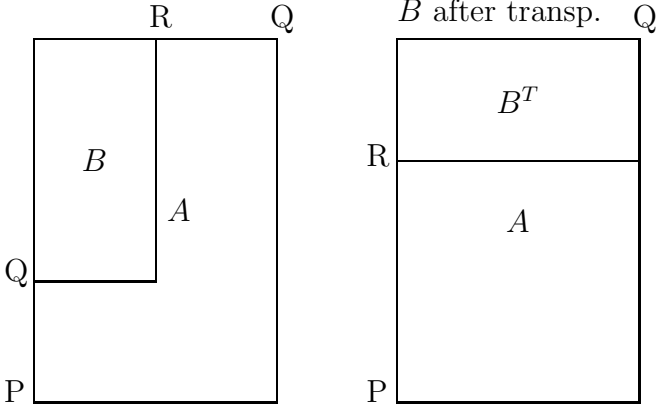


Figure 30: Matrix multiplication when A is large.

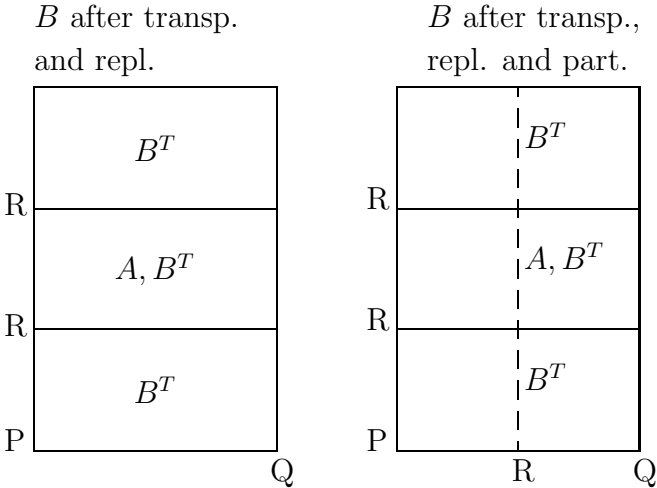


Figure 31: Matrix multiplication when B is large, replication and partitioning.

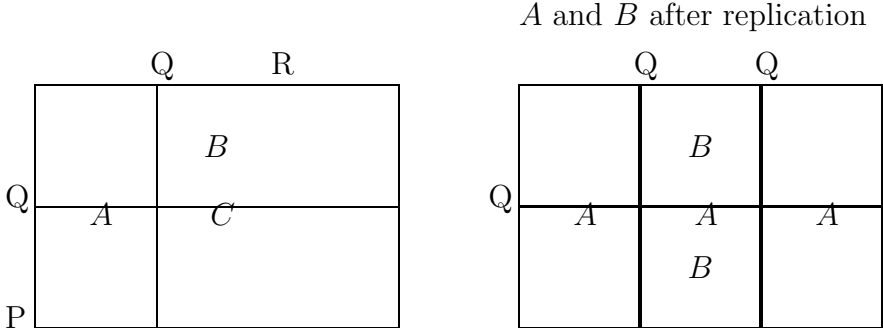


Figure 32: Matrix multiplication when C is large.

32. Then, matrix multiplication as described above for C stationary is performed within each subarray of shape $N_Q^B \times N_Q^A$.

Thus, the algorithm for C large and $N < P \times R$ is as follows

1. Replicate A $\frac{N_R}{N_Q^A}$ times along the R -axis.
2. Replicate B $\frac{N_P}{N_Q^B}$ times along the P -axis.
3. Concurrent matrix multiplication within subarrays of shape $N_Q^B \times N_Q^A$.

If $PR < N$, then first reshape the nodal array if necessary such that the elements of C are allocated to all nodes of the array. Then, partition the Q -axis into $k = \frac{N}{PR}$ partitions and assign one partition of shape $P \times Q/k$ of A and one partition of shape $Q/k \times R$ of B to each set of nodes of shape $P \times R$. After the multiplication for C stationary in each such partition, compute C through a reduction, where each element of C is the sum of $\frac{N}{PR}$ partial results, one from each partition.

2.2.4 Discussion

In the case the number of nodes is large compared to the largest matrix, i.e., $N > P \times R$, $N > P \times Q$ or $N > Q \times R$, then improved load-balance only comes from partitioning the inner axis into $\frac{N}{PR}$ partitions and using an algorithm for C stationary in sets of $P \times R$ nodes.

If at least one of the matrices is sufficiently large to cover all nodes, then an algorithm should be chosen that keeps the largest matrix stationary. Full load-balance can be achieved through replication, if necessary. Moreover, the all-to-all communication can be partitioned into two phases:

- for broadcast
 - broadcast to axis segments
 - all-to-all broadcast within axis segments
- for reduction
 - all-to-all reduction within axis segments
 - reduction between axis segments

In some architectures, such as binary cubes, this partitioning of the all-to-all communication may yield improved efficiency.

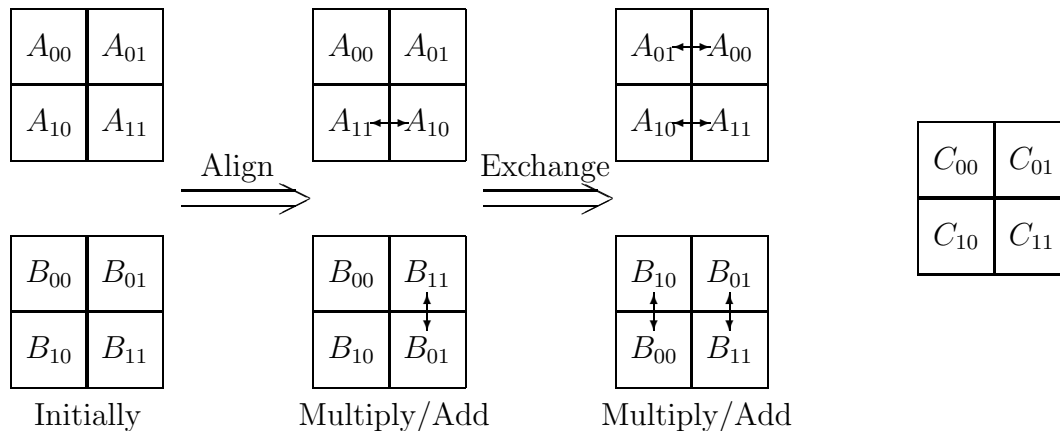


Figure 33: Block matrix multiplication through exchanges.

2.3 Matrix multiplication on a binary cube

Since the binary cube contains meshes as subgrids, the above algorithms can be emulated on a binary cube. However, the all-to-all communication can be performed in reduced time using all n channels of every node [10]. For the algorithm with C stationary, the data motion of A within rows results in an all-to-all communication. Every processor receives all elements of every row of A assigned to that processor row. Similarly, the shifting of B implements an all-to-all broadcast within processor columns. For binary cube networks, the shifting makes use of the fact that a cycle can be embedded in a subcube with dilation one, if the matrices are sufficiently large for each matrix to be distributed over all processing nodes with the selected processing array shape. But, on binary cubes there are ways of performing the all-to-all communication that allows for a more efficient use of the binary cube network when bandwidth utilization is important.

Consider the block matrix multiplication in Figure 33. It can be viewed as multiplication of two matrices on a 2×2 processing array. The alignment is exactly the same as before, and the rotation of A and B amounts to an exchange. However, we now apply the partitioning idea recursively. Then, in the next recursion step, each of the blocks are multiplied together by a 2×2 subarray for a total array of shape 4×4 . The idea is shown in Figure 34.

The initial alignment is

$$A_{i,j} \rightarrow A_{i,i \oplus j}, \quad B_{i,j} \rightarrow B_{i \oplus j,j}$$

Between each multiplication/addition step, the exchanges

$$A_{i,j} \rightarrow A_{i,j \oplus 2^k}, \quad B_{i,j} \rightarrow B_{i \oplus 2^k,j}$$

are performed, where $t_k = \{0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, \dots\}$. This sequence is precisely the transition sequence in a binary-reflected Gray code. This algorithm is due to Dekel et. al. [4]. This algorithm has been generalized to nonsquare binary cubes, and matrices of arbitrary

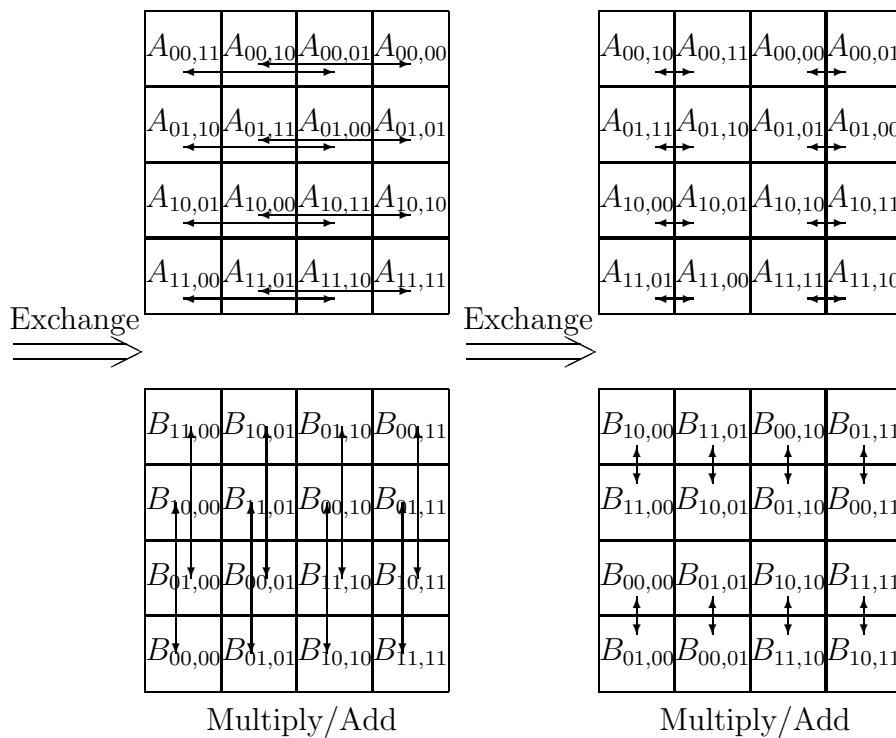
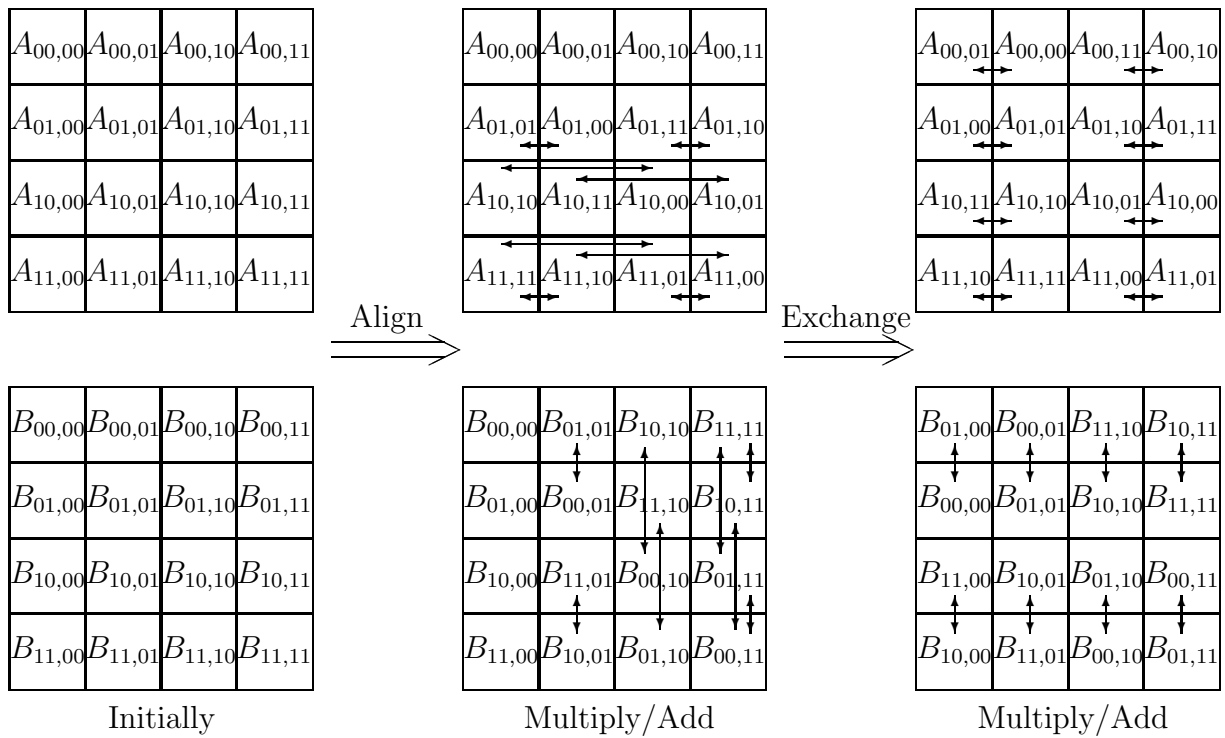


Figure 34: Recursive block matrix multiplication.

shapes by Johnsson et. al. [10]. The generalization has also been made to fully utilize the bandwidth of the binary cube. As originally described by Dekel, the recursive algorithm does not have any great advantage over the emulated mesh algorithm. However, the generalized algorithm in [10] offers an improved communication performance by a factor of $O(\log_2 N)$. Before describing the algorithm we present an all-to-all broadcast algorithm that makes full use of the binary cube communications bandwidth.

2.3.1 All-to-all broadcast on binary cubes

A binary n -cube has $N = 2^n$ nodes, with one adjacent node for every bit in a node address. There are n cube dimensions, also called channels, labeled from 0. We now show how all n channels of every node can be used concurrently for all-to-all broadcast [14]. Concurrent communication on all channels of every node is known as *all-port* communication. All-port communication is supported on the Connection Machine systems CM-2 and CM-200.

Figure 35 shows a 3-cube and the first four steps of three different recursive exchange sequences. The last three steps (which are not shown) are identical to the first three steps. Each exchange sequence has the property that upon completion every node has been visited by every index. For instance, consider Sequence 0. After the first three steps, indices 0 through 3 have visited the four nodes that were initially assigned these four indices. Similarly, the indices 4 through 7 have visited the nodes that were initially assigned the indices 4 through 7. In the fourth exchange step in Sequence 0, the indices are exchanged between nodes such that the indices 0 through 3 now appear in nodes that during the first three exchange steps only received indices in the range 4 through 7. Conversely, the indices 4 through 7 are visiting nodes that during the first three exchange steps received indices in the range 0 through 3. By repeating exchange steps 0 through 3, it is clear that every node has received every index. It is also clear that the memory has been conserved in this all-to-all broadcast scheme [14]. It is easily verified that in sequences 1 and 2, every node has also been visited by every index upon completion of the respective exchange sequence.

The exchanges for Sequence 0 are performed in cube dimensions $(0, 1, 0, 2, 0, 1, 0)$. For Sequence 1 the exchange sequence is $(1, 2, 1, 0, 1, 2, 1)$, and for Sequence 2 it is $(2, 0, 2, 1, 2, 0, 2)$. Sequence 1 is generated from Sequence 0 by replacing the dimensions $(0, 1, 2)$ by the dimensions $(1, 2, 0)$. Sequence 2 is generated by replacing in Sequence 0 the dimensions $(0, 1, 2)$ by the dimensions $(2, 0, 1)$. In every step the three sequences use different dimensions, and there is no contention. The sequence of exchange dimensions for Sequence 0 is indeed the transition sequence for a binary-reflected Gray code [23]. Furthermore, each sequence defines 8 Hamiltonian paths in a 3-cube, each starting from a distinct node.

We now formally define the exchange sequences that allow balanced, minimum-contention communication. Let T_n be the transition sequence in the n -bit binary-reflected Gray code [23]. For instance, $T_3 = (0, 1, 0, 2, 0, 1, 0)$. T_n can be recursively defined as $T_0 = (0)$ and $T_n = T_{n-1}|n-1|T_{n-1}$, where “|” is the concatenation operator of two sequences. T_n is a sequence of $2^n - 1$ numbers that defines a Hamiltonian path on an n -cube. We also define $T_{n,s}$, where $0 \leq s < n$, from T_n by adding s (modulo n) to each number in the sequence T_n . For instance, $T_{3,0} = T_3 = (0, 1, 0, 2, 0, 1, 0)$, $T_{3,1} = (1, 2, 1, 0, 1, 2, 1)$ and $T_{3,2} = (2, 0, 2, 1, 2, 0, 2)$. In Figure 35, Sequence 0 corresponds to $T_{3,0}$ and Sequences 1 and 2 to $T_{3,1}$ and $T_{3,2}$, respectively.

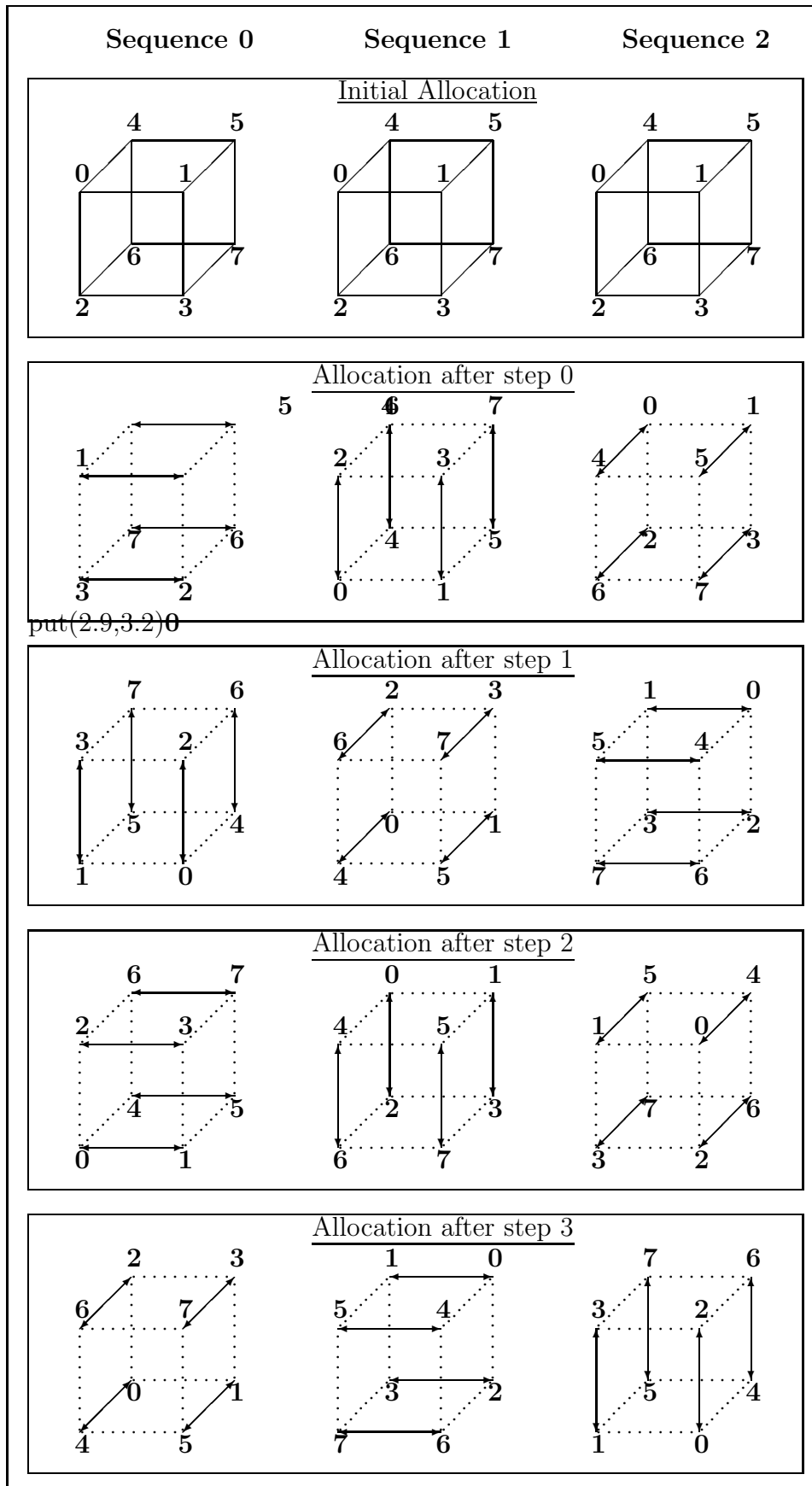


Figure 35: Three concurrent exchange sequences in a 3-cube.

Lemma 1 *Performing exchanges in a binary n -cube such that the ℓ -th element, $0 \leq \ell < 2^n - 1$, in any sequence $T_{n,s}$, $0 \leq s < n$, determines the dimension subject to exchange in step ℓ , makes the content of each node visit every other node precisely once.*

Proof: It suffices to prove that traversing the cube dimensions according to the sequence $T_{n,s}$ for any n and s defines a Hamiltonian path for any starting node. Since the sequence $T_{n,s}$ is based on a permutation of the dimensions used for the sequence T_n , it suffices to prove the lemma for T_n . The lemma follows from the definition of T_n . ■

The communication implemented by each of the sequences $T_{n,s}$ is known as all-to-all broadcast [1, 6, 14, 24]. With multiple elements per node, Lemma 1 gives a construction of n minimum-contention, concurrent exchange sequences for all-to-all broadcast in binary n -cubes. For a data set of size M distributed evenly over an N node binary-cube, the number of element exchanges in sequence is $\frac{M}{nN}(N - 1)$, which is optimal [14]. Note that for a given data set the communication time decreases with an increased size of the cube over which the data set is distributed, even though the amount of data a node must receive is independent of the number of nodes over which it is spread initially. This property has been verified experimentally [2, 9].

2.3.2 A binary cube algorithm for matrix multiplication

The all-to-all broadcast function presented above together with the partitioning of the inner axis of A and B , form the basis for our matrix multiplication algorithm. The *prealignment* assures that a suitable set of inner indices are shared between A and B in each node, regardless of the initial allocation of A and B . The prealignment also assures that the set of row indices for A and C is the same in each node, and that the set of column indices of B and C is the same in each node. The *postalignment* restores the initial allocation of all matrices. The *multiplication phase* performs all necessary computations, as well as the required data motion therefore. The data motion for a square nodal array is simpler than for a rectangular nodal array. We therefore present first the algorithm for a square nodal array.

The order in which the products in $c(i, j) = \sum_m a(i, m)b(m, j)$ are computed and accumulated is immaterial, assuming associativity of addition. By using exchange sequence s , defined above for the all-to-all broadcast of the two partitions A_s and B_s , no interference between the communications for different partitions will occur. The contention is minimized by evenly distributing the communication load over all channels. The algorithm is arithmetically load-balanced and requires minimal temporary storage.

2.3.3 Square nodal arrays

For square nodal arrays, the algorithm proceeds as follows

Align the segments of
 the P -axis of A and C ,
 the Q -axis of A and B , and

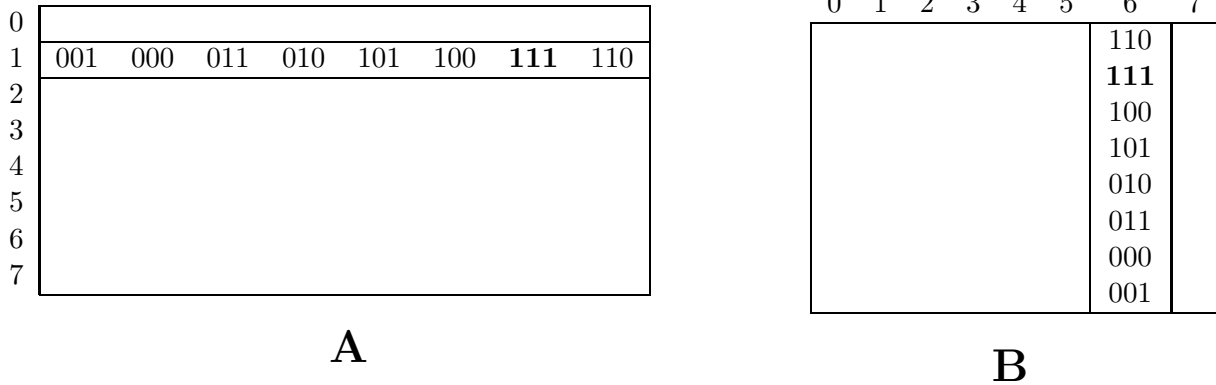


Figure 36: The result of the prealignment for row 1 of A and row 6 of B . $P = Q = R = 8$.

the R -axis of B and C
 assigned to each processing node
for $\ell = 1$ **to** $N^{\frac{1}{2}}$ **do**
 Perform N local matrix multiplications concurrently
 Perform one step of $N^{\frac{1}{2}}$ concurrent all-to-all broadcasts
endfor
Realign A , B , and C with their initial layouts

For the description of the algorithm and proof of its correctness, we first consider the case with one matrix element per partition and node. Then, we present a block algorithm for matrices of arbitrary shapes.

$\log_2 N$ matrix elements per node Figure 36 shows the inner indices of row 1 of A and column 6 of B after prealignment. A and B are both 8×8 matrices. Clearly, the inner indices in node $(1, 6)$ are the same for A and B . Moreover, if the same exchange sequence is applied to row 1 of A and column 6 of B , then the inner indices of A and B in node $(1, 6)$ will always be the same. For instance, an exchange in dimension 0 within the row and column subcubes will bring inner index 110 into node $(1, 6)$. A subsequent exchange in dimension 1 will bring in index 100, assuming that an exchange is always applied to all nodes. (Index 100 was exchanged with index 101 when the exchange was made in dimension 0.)

Let $n' = n/2$, where n is even, and let \oplus denote addition modulo 2. Then, the prealignment assigns elements as follows

$$\mathbf{A}:(i, j \cdot n' + s) \leftarrow (i, (j \oplus i)n' + s)$$

$$\mathbf{B}:(i \cdot n' + s, j) \leftarrow ((i \oplus j)n' + s, j)$$

for $0 \leq s < n'$ and any node (i, j) . This allocation implicitly assumes that n' consecutive elements along the inner axis are allocated to the same node. Consecutive data allocation is supported on the Connection Machine systems, and is included in Fortran D [5], Vienna Fortran

[27], and the emerging High Performance Fortran Standard. Theorem 2 below defines the data motion for the multiplication phase.

Theorem 2 *Let $T_{n',s}(\ell)$ be the ℓ -th element in the sequence $T_{n',s}$ and let element $a(i, (i \oplus j)n' + s)$ of A and element $b((i \oplus j)n' + s, j)$ of B be assigned to node (i, j) for all i and j , $0 \leq i, j < 2^{n'}$, and any s , $0 \leq s < n'$. Then, $A(i, m)$ and $B(m, j)$ reside in the same node for any i and j and all $0 \leq m < n'2^{n'}$ generated by applying the exchange sequence $T_{n',s}$ to the nodal part (j) of the column index of A_s , i.e., $a(i, jn' + s) \leftarrow a(i, (j \oplus 2^{T_{n',s}(\ell)})n' + s)$, and to the nodal part of the row index of B_s , i.e., $b(in' + s, j) \leftarrow b((i \oplus 2^{T_{n',s}(\ell)})n' + s, j)$. The exchange sequence $T_{n',s}$ also guarantees that every node (i, j) receives the complete range of indices m , $0 \leq m < n'2^{n'}$.*

Proof: The inner indices of the element of A_s and the element of B_s in node (i, j) are clearly the same after the prealignment, for any i and j . Applying the same exchange sequence $T_{n',s}$ to the column dimensions of A_s and the row dimensions of B_s after the alignment, clearly generates the same sequence of inner indices for both A_s and B_s . The claim that every node receives the complete range of indices follows from Lemma 1. ■

With $Q = n'N^{\frac{1}{2}}$, Theorem 2 gives an algorithm in which all matrix elements are moved concurrently in each step of the multiplication phase. Each partition has one matrix element per node. The contention for communication is minimal in binary-cube networks. With $Q = N^{\frac{1}{2}}$ and s constrained to zero, the algorithm above degenerates to the algorithm by Dekel, et al. [4].

Corollary 1 *The multiplication of an $N^{\frac{1}{2}} \times n'N^{\frac{1}{2}}$ matrix A and an $n'N^{\frac{1}{2}} \times N^{\frac{1}{2}}$ matrix B can be performed in $N^{\frac{1}{2}}$ steps on a binary $2n'$ -cube with all-port communication. Each step consists of one concurrent exchange, and the computation of one inner product on n' elements in each node.*

In the consecutive data allocation assumed in Theorem 2, the inner indices for partition s are the set $\{m | s = m \bmod n'\}$. The n' most significant bits of the inner index encode the elements within a partition. The alignment is made on these address bits (corresponding to the node column address for A and the node row address for B). In a cyclic data allocation [5, 8, 27], inner index m is assigned to node $m \bmod N^{\frac{1}{2}}$ in the column direction for A and the row direction for B . The inner indices for partition s is the set $\{m | s = \lfloor m/N^{\frac{1}{2}} \rfloor\}$. In the cyclic allocation, the n' least significant bits in the encoding of the inner index for A correspond to the column part of the node addresses. The prealignment for a cyclic data allocation is performed using the least significant bits of the column index of A . Similarly, the prealignment is based on the n' least significant bits of the row index of B for a cyclic allocation. In the following we only consider the consecutive data allocation.

Note that the alignment of A depends only upon the row index, and the alignment of B only on the column index. Different partitions of A have the same row indices. Hence, all partitions of A are subject to the same alignment. Similarly, all partitions of B have the same column indices, and all partitions of B are subject to the same alignment.

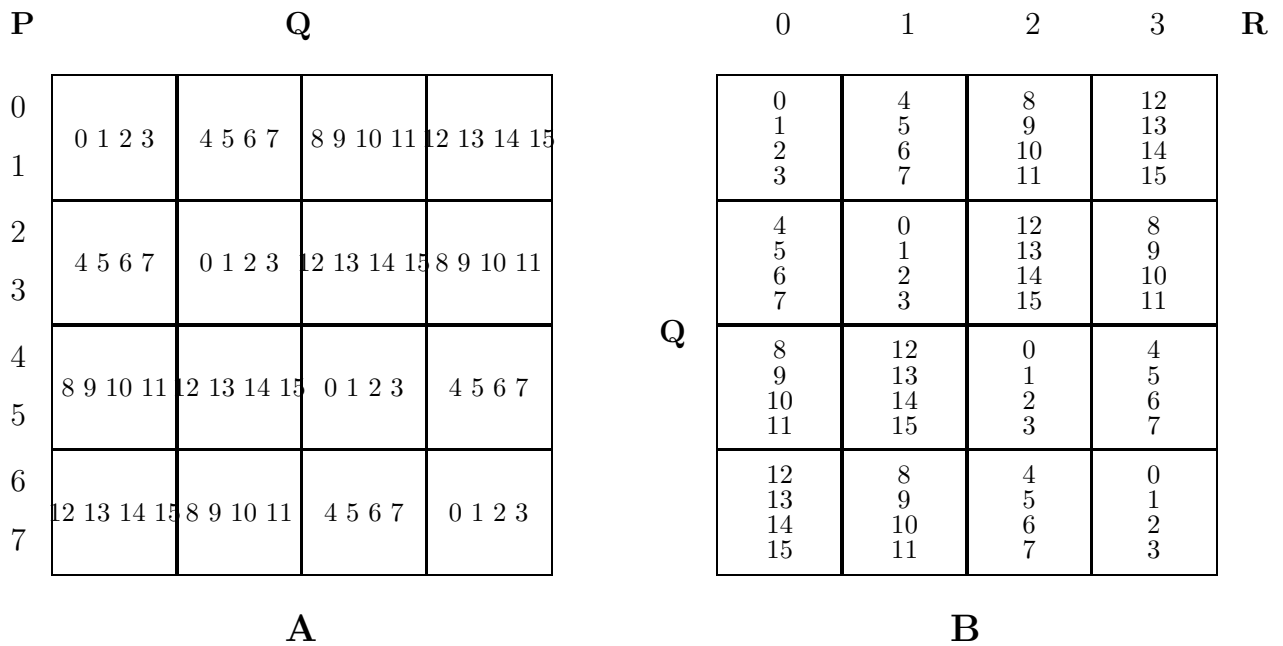


Figure 37: Alignment of an 8×16 matrix A and a 16×4 matrix B on a 4×4 array. Inner indices shown in the blocks.

Note also that although the routing described above is ideal for a binary-cube network, the correctness is guaranteed for any network. Source and destination addresses are based on node addresses, matrix indices, and data allocation, all of which are independent of the network topology.

2.3.4 Matrices of arbitrary shape

We now generalize the algorithm above to any $P, R \geq N^{\frac{1}{2}}$ and to any $Q \geq n'N^{\frac{1}{2}}$. Simply emulating a large virtual machine, in which there is one matrix element per node (of C), on a smaller machine implies excessive data motion. With an inner axis segment of length $Q/(n'N^{\frac{1}{2}})$ assigned to each node, an emulation would require $(P/N^{\frac{1}{2}}) \cdot (Q/(n'N^{\frac{1}{2}}))^2$ local memory moves in each node for matrix A_s , and $(R/N^{\frac{1}{2}}) \cdot (Q/(n'N^{\frac{1}{2}}))^2$ moves for B_s . The number of moves grows quadratically in the length of the local segment of the inner axis.

We now show that for each partition A_s and B_s , the submatrices assigned to a node can be viewed as a single block for both the alignment and multiplication phases. Thus, *node addresses* can be used to *control the data motion*, instead of the matrix indices, and the *local data motion eliminated*. However, performing the alignment based entirely on node addresses is not possible when the numbers of nodes assigned to rows and columns are different, as we show in the next section. Figure 37 shows the result of prealignment based on node addresses for an 8×16 matrix A and a 16×4 matrix B .

For $P = pN^{\frac{1}{2}}$, $Q = qn'N^{\frac{1}{2}}$ and $R = rN^{\frac{1}{2}}$, where p, q and r are arbitrary integers greater than

0, the indices of A assigned to node (i', j') , $0 \leq i', j' < N^{\frac{1}{2}}$, are $(i'p + \beta, j'q + \gamma)$. For B they are $(i'q + \gamma, j'r + \delta)$, where $0 \leq \beta < p$, $0 \leq \gamma < q$, and $0 \leq \delta < r$. In Figure 37, $p = 2$, $q = 4 = 2n'$, and $r = 1$. Performing the alignment using the node address (i', j') guarantees that the range of the inner indices in each node for A and B is the same. Applying the same exchange sequence to the node address of the inner index, i.e., j' for A and i' for B , preserves this property.

Further relaxation of the restriction that the number of rows of A is a multiple of the number of processor rows, and the number of columns of B is a multiple of the number of processor columns, is conceptually simple, but the notation becomes cumbersome. Relaxing the restriction that the inner index, Q , is a multiple of the number of processor rows and columns must be made with care. A given block of columns of A and the corresponding block of rows of B must have the same inner indices. The following result now ensues.

Theorem 3 *Let A be of shape $P \times Q$ and B be of shape $Q \times R$, where $P, R \geq N^{\frac{1}{2}}$ and $Q \geq \frac{n}{2}N^{\frac{1}{2}}$. Then, the matrix multiplication $C \leftarrow A \times B$ can be performed as a block algorithm with no data motion within the blocks, and with fully concurrent communication. The data motion for the prealignment and the multiplication phase can be based entirely on the node addresses and the partition index s .*

Note that the formulation of the block algorithm is independent of the network topology.

2.3.5 Matrix multiplication on rectangular arrays

The number of processing nodes is not necessarily a square. For instance, 128 nodes cannot be configured as a square array. The alignment of each partition A_s and B_s , described in the previous section, was critically dependent upon the inner axis Q being partitioned in the same way for A_s and B_s . In this section, we remove this restriction.

Let the shape of the nodal array be $N_r \times N_c = N$, where $N_r = 2^{n_r}$ is the number of nodes assigned to the row axis, and $N_c = 2^{n_c}$ is the number of nodes assigned to the column axis. We assume that $P, Q \geq N_r$, $Q, R \geq N_c$. Load-balancing becomes an issue for matrix shapes for which these conditions do not hold [11, 12]. We first consider the unpartitioned matrices A and B , then divide the matrices into $k = \max(n_r, n_c)$ partitions for concurrency in communication.

The prealignment of two 8×8 matrices assigned to an 8×4 nodal array is shown in Figure 38. The numbers in the figure represent the inner index. Dashed lines separate the indices of two elements assigned to the same node. For the modification of the prealignment and the multiplication phases, we let $n_r > n_c$, $P = pN_r$, $Q = q_rN_r = q_cN_c$, and $R = rN_c$. After the prealignment of A , B and C , the row indices of A and C assigned to node (i', j') , $0 \leq i' < N_r$, $0 \leq j' < N_c$, are $(i'p + \beta)$, $0 \leq \beta < p$. The set of column indices of B and C assigned to node (i', j') are $(j'r + \delta)$, $0 \leq \delta < r$.

For the prealignment, we conceptually partition the column axis of the nodal array into N_r blocks, instead of the N_c blocks imposed by the number of nodes along the column axis. Thus, we form conceptually a square nodal array for the prealignment. The multiplication phase will be based on the actual array shape. With the column axis divided into N_r blocks, there are

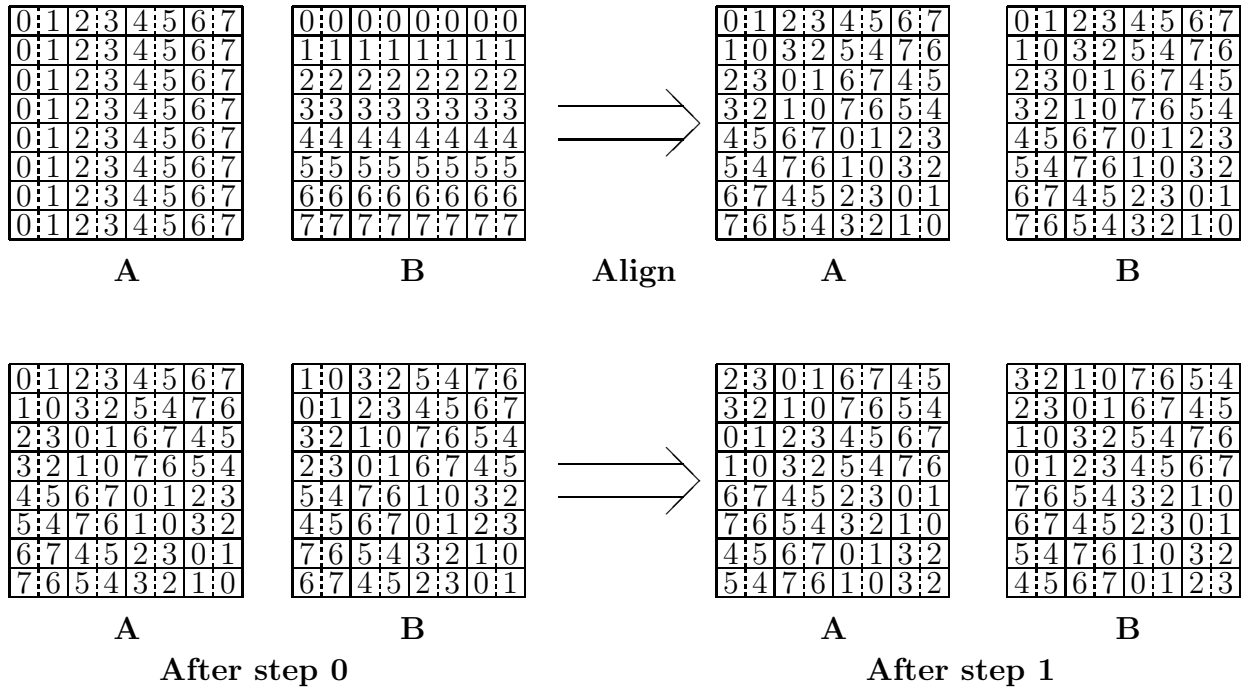


Figure 38: Location of inner indices initially, after alignment, and after the first two exchange steps for matrix multiplication on an 8×4 nodal array.

$N_r/N_c = 2^{n_r-n_c}$ blocks of shape $p \times q_r$ of A assigned to each node. The blocks have the same block row index (i'). Similarly, there are $2^{n_r-n_c}$ blocks of shape $q_r \times r$ of B assigned to each node. The block column indices, j'_v , for those blocks are in the range $j'2^{n_r-n_c} \leq j'_v < (j' + 1)2^{n_r-n_c}$. After the prealignment, block (i', j'_v) contains data with block index $(i' \oplus j'_v, j'_v)$. For instance, in Figure 38 element (1, 2) is assigned to node (3, 1) after prealignment, and element (1, 3) is assigned to node (2, 1). Note that i' and j'_v are encoded in the same number of bits (n_r), so that $(i' \oplus j'_v)$ is well-defined, whereas i' and j' are encoded in a different number of bits.

After the prealignment, block (i', j'_v) of A is assigned data with block index $(i', i' \oplus j'_v)$. The alignment of A implies that q_c consecutive inner indices are assigned to node (i', j') after prealignment, but the inner block indices for blocks of size $p \times q_r$ are permuted. The latter permutation is based on the $n_r - n_c$ least significant bits of i' . The local rearrangement is the same for all nodes within a row, but different for different rows, as shown in Figure 38.

In general, the prealignment is based on a blocking determined by the longest nodal axis. This blocking implies that for the alignment along the longer nodal axis (B in the example above), the blocks assigned to a node are likely to be received from different nodes. For the alignment along the shorter nodal axis (the alignment of A in our example), the blocks assigned to a node may often be received from a single node. One large block transfer may accomplish the task.

The multiplication phase can be performed by communicating entire submatrices assigned to a node. For $n_r > n_c$, emulating a square array with N_r nodes along each axis would require $2^{n_r-n_c} = \frac{N_r}{N_c}$ local (column) exchanges for A between each exchange between nodes. These local exchanges need not be performed with proper bookkeeping of block operations. For B ,

the same exchange sequence is applied to all $q_r \times q_r$ blocks in a node. Thus, the entire $q_r \times r$ submatrix assigned to a node can be moved as one block. Hence, in the multiplication phase, local blocks of A of size $\frac{P}{N_r} \times \frac{Q}{N_c}$ are exchanged between pairs of nodes in a row, once for every $\frac{N_r}{N_c}$ times blocks of size $\frac{Q}{N_r} \times \frac{R}{N_c}$ of B are exchanged between pairs of nodes in the same column.

The data motion for the prealignment and the first two steps of the multiplication phase are illustrated in Figure 38 for an 8×4 nodal array. After the prealignment, each node can perform two block matrix multiplications, one for each of two block columns. After exchange step 0, in which an exchange is performed only on B , but on the entire submatrix assigned to a node, the two inner indices for A and B in a node are still the same; a new pair of block matrix multiplications can be performed in each node. The only difference compared to the first block matrix multiplication is that the local matching of blocks of A and B is different. For instance, consider node $(0,0)$ in Figure 38. After the prealignment, the products $a(0,0)b(0,0)$ and $a(0,1)b(1,1)$ are computed for $c(0,0)$ and $c(0,1)$, respectively. After exchange step 0, the node $(0,0)$ computes the products $a(0,1)b(1,0)$ and $a(0,0)b(0,1)$ for the same two elements of C . Figure 39 shows all steps for matrix multiplication on a 4×2 nodal array. Striped blocks in the same node interact in each step. Similarly, clear blocks also interact.

To achieve the desired concurrency in the use of the communication channels, A and B are divided into $\max(n_r, n_c)$ partitions. The exchange sequences for A_s and B_s are based on $T_{\max(n_r, n_c), s}$. However, since one axis has fewer dimensions than the other, the column axis for $n_r > n_c$, $n_r - n_c$ dimensions are mapped into local memory for that axis. Exchanges along those dimensions should be replaced by pointer manipulation to avoid excessive data motion. For instance, when $n_r = 3$ and $n_c = 2$, A is partitioned into three column blocks and B into three row blocks. Blocks s , $s \in \{0, 1, 2\}$, of A and B follow the exchange sequence $(s, s+1, s, s+2, s, s+1, s)$ (modulo 3). Dimension 0 in the exchange sequence of A is mapped to local memory. Dimensions 1 and 2 are mapped to cube dimensions. Thus, at any given exchange step of B , two out of three partitions of A are involved in communication. The number of inner indices per partition and node for A_s is $\frac{N_r}{N_s}$ times higher than for B .

Theorem 4 *The matrix multiplication $C \leftarrow A \times B$, where A is of shape $P \times Q$ and B of shape $Q \times R$, $P, Q \geq N_r$ and $Q, R \geq N_c$, can be performed as block matrix multiplication using all communications channels for prealignment and for some multiplication steps on a binary n -cube with $N = N_r \times N_c$ nodes. For $N_r = N_c$, all communication channels can be used for every step of the multiplication phase. The prealignment can be based on a square nodal array with $2 \max(n_r, n_c)$ dimensions, while the data motion for the multiplication phase can be based entirely on node addresses.*

2.3.6 Complexity analysis

The complexity analysis below is limited to binary-cube networks. For the analysis, we assume that initially the matrices are assigned to a nodal array with the same shape for all matrices. On the Connection Machine systems this assumption is valid for square matrices. After the analysis we discuss the consequences of a canonical data array layout.

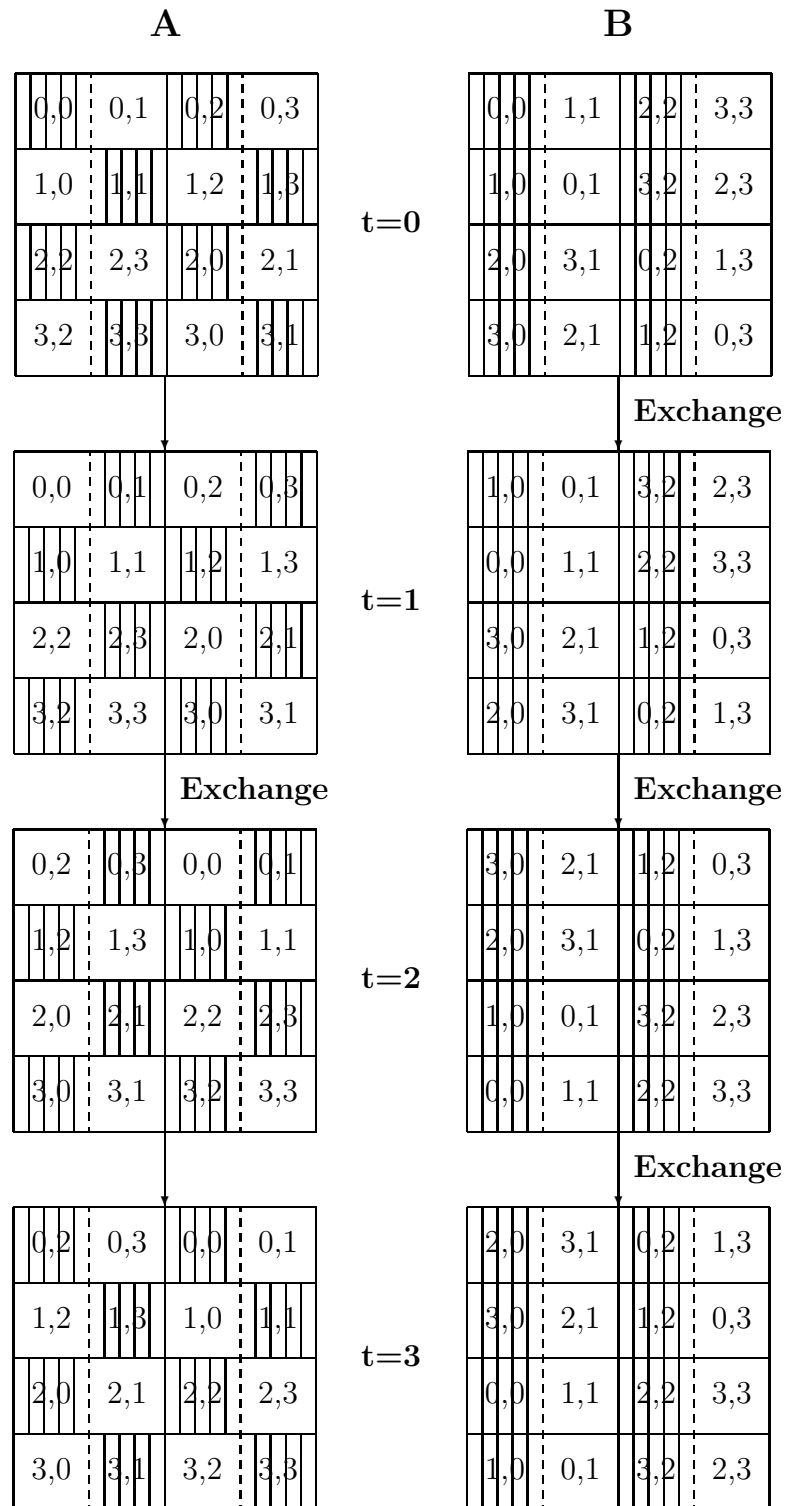


Figure 39: Matrix multiplication on a 4×2 nodal array.

Prealignment In the alignment phase, the communication needs for A depend upon the row index and for B upon the column index. The communication for A when $N_r = N_c$ is known as bit-inversion. A node either sends its entire submatrix to another node or does nothing. Bit-inversion is defined as $j \leftarrow j \oplus 1^n$, where 1^n is a string of n 1's. This communication is precisely the communication required for A in row subcube $i = N_r - 1$, if $N_r \geq N_c$, and for B in column subcube $j = N_c - 1$, if $N_c \geq N_r$. The communication for A in other row subcubes implies bit-inversion on subsets of cube dimensions, if $N_r > N_c$. The same property is true for B with respect to column subcubes, if $N_c > N_r$. Any tight bound for a complete cube is also a tight bound for a subcube.

Lemma 2 [16] *Any tight bound for communication in a binary n -cube is also a tight bound for the same communication in all disjoint n -dimensional subcubes of an m -cube, $m > n$, when the subcubes are identified by the same n dimensions.*

The significance of this lemma is that even though only a fraction of the total bandwidth of the m -cube is used, the communication time cannot be reduced when the communication in each subcube is optimal. The bits subject to inversion define a subcube, and the bits not inverted define the disjoint instances of the subcubes in which inversion is performed.

Lemma 3 [24] *For bit-inversion on a binary n -cube with all-port communication and K data elements per node, a tight bound is K for the number of element transfers in sequence.*

Proof: Since every element must traverse all cube dimensions, the required bandwidth is nNK . The available bandwidth is nN , which gives the lower bound K . For the upper bound, divide the local data set into n parts. Exchange part i , $0 \leq i \leq n - 1$, according to the sequence of dimensions $i, (i+1) \bmod n, \dots, (i+n-1) \bmod n$. All n data sets can be exchanged concurrently without channel conflict. ■

The proof of Lemma 3 also provides an optimal algorithm for bit-inversion when $N_r = N_c$.

If $N_r \neq N_c$, the alignment along the shorter axis (the alignment of A for $N_r > N_c$) is still bit-inversion, while for the other axis, the alignment implies a combination of bit-inversion and all-to-all personalized communication [6, 14, 24]. In all-to-all personalized communication, a node sends a unique piece of data to every other node. For the case $N_r > N_c$, the alignment of B implies all-to-all personalized communication in $n_r - n_c$ cubes and bit-inversion in n_c -cubes. The all-to-all personalized communication stems from the fact that in the alignment phase, different blocks of size $(\frac{Q}{N_r})^2$ in the same node must be moved to different nodes in subcubes of dimension $n_r - n_c$. In addition, the alignment on the leading n_c dimensions implies a move of the entire contents between $n_r - n_c$ dimensional subcubes.

Lemma 4 *Let every node x in an n_1 -cube with K elements per node send unique sets of $\frac{K}{2^{n_1-n_0}}$ elements to each of $2^{n_1-n_0}$ nodes defined by $x \oplus (1^{n_0} *^{n_1-n_0})$, $n_1 \geq n_0$, where $*^{n_1-n_0}$ defines an $(n_1 - n_0)$ -dimensional subcube. Then, for all-port communication, a tight bound on the number of element transfers in sequence is K , if $n_0 > 0$ and $\frac{K}{2}$ for $n_0 = 0$.*

Proof: For $n_0 = 0$, the communication is all-to-all personalized communication in an n_1 -cube. A tight bound for the number of element transfers in sequence with all-port communication is $\frac{K}{2}$ [14]. For $n_0 > 0$, the lower bound is determined by bit-inversion. The entire data set K in a node is subject to the same bit-inversion on n_0 bits. The required bandwidth is $Kn_02^{n_1}$. An n_1 -cube can be considered as an n_0 -cube of “supernodes”, each consisting of $2^{n_1-n_0}$ cube nodes. There are $2^{n_1-n_0}$ communication channels between every pair of “supernodes”. Hence, the bandwidth available for the permutation is $n_02^{n_1-n_0}2^{n_0}$, and the lower bound K follows.

For the upper bound and $n_0 > 0$, partition each of the $\frac{K}{2^{n_1-n_0}}$ local data sets further into n_1 sets. Set s of size $\frac{K}{n_12^{n_1-n_0}}$ is assigned an exchange sequence $s, (s+1) \bmod n_1, \dots, (s+n_1-1) \bmod n_1$. The exchange on a dimension is conditional, and determined by the dimensions involved in the all-to-all personalized communication of the set of size $\frac{K}{2^{n_1-n_0}}$ and the bit-inversion. All n_1 different exchange sequences can be performed concurrently; all blocks of size $\frac{K}{n_12^{n_1-n_0}}$ that need to be exchanged in a dimension can be exchanged as one transfer operation, should that be advantageous. The permutation can be completed in n_1 exchanges. Each set of $\frac{K}{2^{n_1-n_0}}$ elements require $\frac{K}{2^{n_1-n_0}}$ element transfers in sequence by this algorithm; the total number of element transfers in sequence is K . ■

Let $n_1 = \max(n_r, n_c)$ and $n_0 = \min(n_r, n_c)$ in the above lemma. The case $n_0 = 0$, i.e., $N_r = 1$ or $N_c = 1$, corresponds to a one-dimensional nodal array. In this case, only one of the operands needs to be aligned. This alignment is equivalent to a matrix transposition (i.e., all-to-all personalized communication).

We have now given algorithms and tight bounds for the alignment of arbitrary, large matrices on binary-cubes of arbitrary size, and the following lemma follows.

Lemma 5 *The prealignment of A and B on a binary-cube of shape $N_r \times N_c$ and with all-port communication requires $\max(\lceil \frac{P}{N_r} \rceil \lceil \frac{Q}{N_c} \rceil, \lceil \frac{Q}{N_r} \rceil \lceil \frac{R}{N_c} \rceil)$ element transfers in sequence for $N_r \neq 1$, $N_c \neq 1$. For $N_r = 1$ only A needs to be aligned, the number of element transfers in sequence being $P \lceil \frac{Q}{2N} \rceil$. If $N_c = 1$ only B needs to be aligned, the number of element transfers in sequence being $Q \lceil \frac{R}{2N} \rceil$.*

For a one-dimensional nodal array, the prealignment requires half as many element transfers in sequence as for a two-dimensional nodal array. All two-dimensional nodal arrays require the same number of element transfers in sequence. Thus, in order to minimize the prealignment time, a one-dimensional nodal array with $N_r = 1$ shall be chosen if $P < R$, otherwise a one-dimensional nodal array with $N_c = 1$ shall be chosen. The postalignment is the inverse of the prealignment, requiring the same time.

The multiplication phase For the multiplication phase, the arithmetic load-balance is ideal, the number of arithmetic operations in sequence being $2 \lceil \frac{P}{N_r} \rceil \lceil \frac{R}{N_c} \rceil Q$. The number of element transfers in sequence for A is $\lceil \frac{P}{N_r} \rceil \lceil \frac{Q}{\max(n_r, n_c)N_c} \rceil (N_c - 1)$, and for B is $\lceil \frac{Q}{\max(n_r, n_c)N_r} \rceil \lceil \frac{R}{N_c} \rceil (N_r - 1)$. The number of matrix partitions is $\max(n_r, n_c)$.

Lemma 6 *Matrix multiplication on a binary n -cube of shape $N_r \times N_c$ requires $2 \lceil \frac{P}{N_r} \rceil \lceil \frac{R}{N_c} \rceil Q$ arithmetic operations in sequence. The number of element transfers in sequence with all-port*

communication is

$$\max\left(\left\lceil \frac{P}{N_r} \right\rceil \left\lceil \frac{Q}{\max(n_r, n_c)N_c} \right\rceil (N_c - 1), \left\lceil \frac{Q}{\max(n_r, n_c)N_r} \right\rceil \left\lceil \frac{R}{N_c} \right\rceil (N_r - 1)\right).$$

For the degenerate cases $N_r = 1$ and $N_c = 1$, only one operand (A or B , respectively) is subject to all-to-all broadcasting. The above communication complexity is equal to the lower bound for this operation [14].

A two-dimensional nodal array yields a lower communication complexity than a one-dimensional array. The communication time is minimized when $\frac{N_r}{N_c} = \frac{P}{R}$, or $N_r = \left(\frac{PN}{R}\right)^{\frac{1}{2}}$ and $N_c = \left(\frac{RN}{P}\right)^{\frac{1}{2}}$, ignoring the quantization effects of the ceiling functions. The minimum number of element transfers in sequence is

$$T_{\min}^{2D} \approx \max(P, R) \frac{Q}{N} + \frac{Q}{\max(n_r, n_c)N} (PRN)^{\frac{1}{2}}.$$

The communication complexity for a two-dimensional nodal array is lower than for a one-dimensional array by a factor of approximately $\frac{N^{\frac{1}{2}}}{2}$ for large N , because of fewer communication steps. The factor of 2 in the denominator is due to the fact that the n communication channels are split between two operands in the two-dimensional nodal array.

The total complexity of matrix multiplication Combining the complexity expressions for the prealignment and the multiplication phase from Lemmas 5 and 6, we arrive at the following result:

Theorem 5 *The multiplication of a $P \times Q$ matrix A with a $Q \times R$ matrix B on a binary n -cube of shape $N_r \times N_c$, $P, Q \geq N_r$, $Q, R \geq N_c$, requires $2\left\lceil \frac{P}{N_r} \right\rceil \left\lceil \frac{R}{N_c} \right\rceil Q$ arithmetic operations in sequence. With all-port communication, the number of element transfers in sequence for prealignment, multiplication, and postalignment is*

$$\begin{cases} 2 \cdot \max\left(\left\lceil \frac{P}{N_r} \right\rceil \left\lceil \frac{Q}{N_c} \right\rceil, \left\lceil \frac{Q}{N_r} \right\rceil \left\lceil \frac{R}{N_c} \right\rceil\right) + \\ \quad + \max\left(\left\lceil \frac{P}{N_r} \right\rceil \left\lceil \frac{Q}{\max(n_r, n_c)N_c} \right\rceil (N_c - 1), \left\lceil \frac{Q}{\max(n_r, n_c)N_r} \right\rceil \left\lceil \frac{R}{N_c} \right\rceil (N_r - 1)\right), & \text{if } N_r \neq 1 \text{ and } N_c \neq 1, \\ 2 \cdot P \left\lceil \frac{Q}{2N} \right\rceil + P \left\lceil \frac{Q}{nN} \right\rceil (N - 1), & \text{if } N_r = 1, \\ 2 \cdot R \left\lceil \frac{Q}{2N} \right\rceil + R \left\lceil \frac{Q}{nN} \right\rceil (N - 1), & \text{if } N_c = 1. \end{cases}$$

For square product matrices, $P = R$, a two-dimensional nodal array yields a lower total communication complexity than a one-dimensional array for $N \geq 8$. The complexity is a factor of $n/2$ lower than fairly simple generalizations of the algorithm in [4]. As the ratio of $\max(P, R)/\min(P, R)$ increases, the range of cube sizes increases for which one-dimensional nodal arrays yield a lower communication complexity than two-dimensional arrays. For instance, for a ratio of 16 between the length of the two matrix axes, one-dimensional nodal arrays yield fewer element transfers in sequence for $N \leq 128$. According to Theorem 5, a two-dimensional nodal array of optimal shape always yield lower communication complexity for $N/n \gtrsim 2P/R$, when $P \geq R$, or for $N/n \gtrsim 2R/P$ when $R \geq P$. Figure 40 shows schematically the regions in which one- and two-dimensional nodal arrays are preferable with respect to communication complexity.

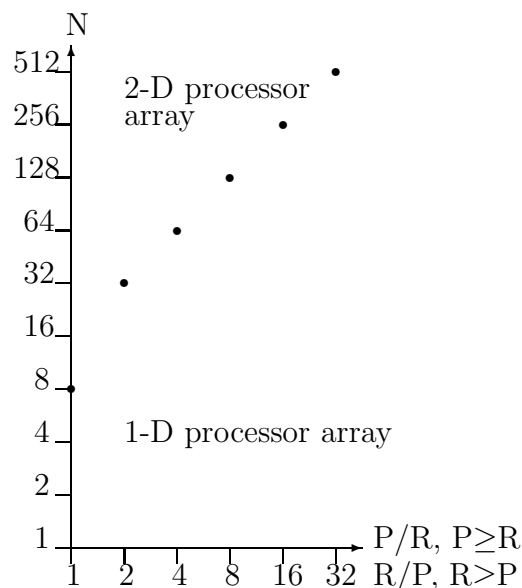


Figure 40: Preferred nodal array shape as a function of the shape of the product matrix.

2.3.7 Discussion

When the matrices A , B and C initially are allocated to the nodes with different nodal array shapes, such as for rectangular matrices on the Connection Machine systems, then the prealignment must also reshape the nodal array, such that after prealignment all operands are allocated with the same nodal array shape. Conceptually, the prealignment can be made in two stages: establish a shared nodal array shape, then align the operands as described above.

Reshaping the nodal array constitutes a generalized shuffle operation [16]. On a binary-cube network with all-port communication, the communication time for this operation depends only upon the number of data elements per node (K in the complexity estimates above). The shape of the nodal array before and after the reshape operation is irrelevant. Moreover, the generalized shuffle operation can be combined with bit-inversion into a single operation without an increase in the communication complexity.

2.4 Fast matrix multiplication

For many years, the conventional matrix multiplication algorithm that requires $2PQR$ arithmetic operations for the multiplication of a $P \times Q$ matrix with a $Q \times R$ matrix was the best known. But, in 1968 Strassen [25] discovered an algorithm for matrix multiplication that required at most $O(P^{\log_2 7})$ operations for the multiplication of two $P \times P$ matrices. Since then, a sequence of new matrix multiplication algorithms have been devised. The exponent have been decreasing slowly from $\log_2 7 = 2.807$ to currently about 2.3.... The overhead of the algorithms with the smaller exponent is quite substantial. For a presentation of the this type of algorithms see for instance [22].

Strassen’s algorithm is derived recursively. For the multiplication of 2×2 matrices the algorithm is as follows:

Let

$$A = \begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix} \quad B = \begin{bmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{bmatrix} \quad C = \begin{bmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{bmatrix},$$

where $C = A \times B$. Then, compute the intermediate results S_i and P_j as follows

Addition/subtraction	Multiplication
$S_0 = A_{00} + A_{01}$	$P_0 = S_2 S_7$
$S_1 = A_{10} + A_{11}$	$P_1 = S_1 B_{00}$
$S_2 = A_{00} + A_{11}$	$P_2 = A_{00} S_9$
$S_3 = -A_{00} + A_{10}$	$P_3 = S_3 S_5$
$S_4 = A_{01} - A_{11}$	$P_4 = S_0 B_{11}$
$S_5 = B_{00} + B_{01}$	$P_5 = A_{11} S_8$
$S_6 = B_{10} + B_{11}$	$P_6 = S_4 S_6$
$S_7 = B_{00} + B_{11}$	
$S_8 = -B_{00} + B_{10}$	
$S_9 = B_{01} - B_{11}$	

The elements of the product matrix C are then obtained by adding the products as follows

$$C_{00} = P_0 + P_5 - P_4 + P_6, \quad C_{01} = P_2 + P_4, \quad C_{10} = P_1 + P_5, \quad C_{11} = P_0 - P_1 + P_2 + P_3$$

The computation requires 7 multiplications and 18 additions/subtractions. The standard matrix multiplication would require 8 multiplications and 4 additions. Thus, one multiplication has been saved at the expense of 14 additions. If multiplication is considerably more time consuming than addition/subtraction, as in a bit–serial architecture where addition requires k steps and multiplication requires k^2 steps for k bit numbers, the method may be faster than the conventional algorithm for a 2×2 matrix. However, the main advantage of Strassen’s algorithm comes from applying the idea recursively. Thus, in the next step we consider the multiplication of two 4×4 matrices, then 8×8 matrices, etc.

Let $M(2^p)$ and $A(2^p)$ be the number of multiplications and additions respectively for the multiplication of two $P \times P$ real matrices. Then,

$$M(2^p) = 7M(2^{p-1}) \quad \text{and} \quad A(2^p) = 7A(2^{p-1}) + 18 \cdot 2^p \cdot 2^p$$

This recurrence results in

$$M(2^p) = 7^p = P^{\log_2 7}, \quad \text{and} \quad A(2^p) = 6(P^{\log_2 7} - P^2)$$

real operations. For complex matrices the operations count is

P	Operation	Real		Complex	
		Strassen	Conventional	Strassen	Conventional
512	Mult.	$0.40 \cdot 10^8$	$1.34 \cdot 10^8$	$1.61 \cdot 10^8$	$5.36 \cdot 10^8$
	Add/sub	$2.41 \cdot 10^8$	$1.34 \cdot 10^8$	$5.62 \cdot 10^8$	$5.36 \cdot 10^8$
	Total	$2.81 \cdot 10^8$	$2.68 \cdot 10^8$	$5.62 \cdot 10^8$	$10.72 \cdot 10^8$
1024	Mult.	$0.28 \cdot 10^9$	$1.07 \cdot 10^9$	$1.13 \cdot 10^9$	$4.29 \cdot 10^9$
	Add/sub	$1.69 \cdot 10^9$	$1.07 \cdot 10^9$	$3.94 \cdot 10^9$	$4.29 \cdot 10^9$
	Total	$1.97 \cdot 10^9$	$2.14 \cdot 10^9$	$5.07 \cdot 10^9$	$8.59 \cdot 10^9$
2048	Mult.	$1.98 \cdot 10^9$	$8.59 \cdot 10^9$	$7.91 \cdot 10^9$	$34.4 \cdot 10^9$
	Add/sub	$11.8 \cdot 10^9$	$8.59 \cdot 10^9$	$27.63 \cdot 10^9$	$34.4 \cdot 10^9$
	Total	$13.8 \cdot 10^9$	$17.18 \cdot 10^9$	$35.54 \cdot 10^9$	$68.7 \cdot 10^9$
4096	Mult.	$1.38 \cdot 10^{10}$	$6.87 \cdot 10^{10}$	$5.54 \cdot 10^{10}$	$27.5 \cdot 10^{10}$
	Add/sub	$8.29 \cdot 10^{10}$	$6.87 \cdot 10^{10}$	$19.36 \cdot 10^{10}$	$27.5 \cdot 10^{10}$
	Total	$9.68 \cdot 10^{10}$	$13.74 \cdot 10^{10}$	$24.89 \cdot 10^{10}$	$55.0 \cdot 10^{10}$
8192	Mult.	$0.97 \cdot 10^{11}$	$5.50 \cdot 10^{11}$	$3.88 \cdot 10^{11}$	$22.0 \cdot 10^{11}$
	Add/sub	$5.81 \cdot 10^{11}$	$5.50 \cdot 10^{11}$	$13.56 \cdot 10^{11}$	$22.0 \cdot 10^{11}$
	Total	$6.78 \cdot 10^{11}$	$11.0 \cdot 10^{11}$	$17.43 \cdot 10^{11}$	$44.0 \cdot 10^{11}$
16384	Mult.	$0.68 \cdot 10^{12}$	$4.40 \cdot 10^{12}$	$2.71 \cdot 10^{12}$	$17.59 \cdot 10^{12}$
	Add/sub	$4.07 \cdot 10^{12}$	$4.40 \cdot 10^{12}$	$9.49 \cdot 10^{12}$	$17.59 \cdot 10^{12}$
	Total	$4.07 \cdot 10^{12}$	$8.80 \cdot 10^{12}$	$12.20 \cdot 10^{12}$	$35.18 \cdot 10^{12}$

Table 1: Comparison of the operation count for Strassen’s algorithm and the conventional matrix multiplication algorithm.

$$M(2^p) = 4 \cdot 7^p = P^{\log_2 7}, \quad \text{and } A(2^p) = 2(7 \cdot P^{\log_2 7} - 6 \cdot P^2)$$

We compare the number of operations required for Strassen’s algorithm and the conventional algorithm for real and complex matrices in Table 1.

As we can see, since a complex addition is only twice as costly as real addition, while complex multiplication is four times as expensive as real multiplication, Strassen’s algorithm become is competitive with the conventional algorithm for a smaller matrix size for complex data.

The recursion can be stopped at any time. If the objective is to minimize the total operations count, then below the break even size the conventional method should be used.

References

- [1] D. P. Bertsekas, C. Ozveren, G.D. Stamoulis, P. Tseng, and J.N. Tsitsiklis. Optimal communication algorithms for hypercubes. *J. of Parallel and Distributed Computing*, 11:263–275, 1991.
- [2] 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.
- [3] L.E. Cannon. *A Cellular Computer to Implement the Kalman Filter Algorithm*. PhD thesis, Montana State Univ., 1969.
- [4] Eliezer Dekel, David Nassimi, and Sartaj Sahni. Parallel matrix and graph algorithms. *SIAM J. Computing*, 10:657–673, 1981.
- [5] Geoffrey Fox, S. Hiranandani, Kenneth Kennedy, Charles Koelbel, U. Kremer, C. Tseng, and M. Wu. Fortran D language specification. Technical Report TR90-141, Department of Computer Science, Rice University, December 1990.
- [6] Geoffrey C. Fox and Wojtek Furmanski. Optimal communication algorithms on the hypercube. Technical Report CCCP-314, California Institute of Technology, July 1986.
- [7] 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.
- [8] S. Lennart Johnsson. Communication efficient basic linear algebra computations on hypercube architectures. *J. Parallel Distributed Computing*, 4(2):133–172, April 1987.
- [9] S. Lennart Johnsson. Performance modeling of distributed memory architectures. *J. Parallel and Distributed Computing*, 12(4):300–312, August 1991.
- [10] S. Lennart Johnsson. Minimizing the communication time for matrix multiplication on multiprocessors. *Parallel Computing*, 19(11):1235–1257, 1993.
- [11] S. Lennart Johnsson, Tim Harris, and Kapil K. Mathur. Matrix multiplication on the Connection Machine. In *Supercomputing 89*, pages 326–332. ACM, November 1989.
- [12] 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.
- [13] S. Lennart Johnsson and Ching-Tien Ho. Matrix multiplication on Boolean cubes using generic communication primitives. In *Parallel Processing and Medium Scale Multiprocessors*, pages 108–156. SIAM, 1989.
- [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] H.T. Kung and Onat Menzilcioglu. Warp: A programmable systolic array processor. In *Proc. SPIE Symp. Real-Time Signal Processing VII*, 1984.
- [18] Kapil K. Mathur and S. Lennart Johnsson. Matrix multiplication on a three-dimensional array. Technical report, Thinking Machines Corp., November 1991.
- [19] Kapil K. Mathur and S. Lennart Johnsson. Multiplication of matrices of arbitrary shape on a Data Parallel Computer. *Parallel Computing*, 20(7):919–951, July 1994.
- [20] Kapil K. Mathur and S. Lennart Johnsson. All-to-all communication on the connection machine CM-200. *Journal of Scientific Programming*, 4:251 – 273, 1995.
- [21] David Nassimi and Sartaj Sahni. Optimal BPC permutations on a cube connected SIMD computer. *IEEE Trans. Computers*, C-31(4):338–341, April 1982.
- [22] Victor Pan. *How to Multiply Matrices Faster*. Springer Verlag Lecture Notes in Computer Science, no 179, 1984.
- [23] E.M. Reingold, J. Nievergelt, and N. Deo. *Combinatorial Algorithms*. Prentice-Hall, Englewood Cliffs. NJ, 1977.
- [24] 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.
- [25] Volker Strassen. Gaussian elimination is not optimal. *Numerische Matematik*, 13:354–356, 1969.
- [26] Thinking Machines Corp. *CMSSL for CM Fortran, Version 3.0*, 1992.
- [27] Hans Zima, Peter Brezany, Barbara Chapman, Piyush Mehrotra, and Andreas Schwald. Vienna Fortran – A language specification version 1.1. Technical report, ICASE, Interim Report 21, March 1992.