Parallel Computing 19 (1993) 777–788
North-Holland

PARCO 769

Short communication

Hypercube matrix multiplication *

Philip A. Nelson

Computer Science Department 9062, Western Washington University, Bellingham, WA 98225-9062, USA

Received 4 February 1992
Revised 18 August 1992, 15 December 1992

Abstract

Nelson, P.A., Hypercube matrix multiplication, Parallel Computing 19 (1993) 777–788.

We present a parallel divide-and-conquer matrix multiplication algorithm whose *natural* communication structure is the hypercube. The complexity of the algorithm is O(log n) using $n^3/2$ processors and O(n) using O(n^2) processors. We show how to use the algorithm for practical computing giving a time of O(n^3/p) using a p processor hypercube. The practical algorithm maps a sub-matrix of each original matrix to each processor. The sub-matrix result in each processor is in the correct processor and form to be used in a subsequent matrix multiply without additional communication.

Keywords. Matrix multiplication; divide-and-conguer algorithm; hypercube architecture; implementation issues; experimental results

1. Introduction

Matrix multiplication is an operation used by many other algorithms. Due to this fact, much effort has been expended in finding parallel algorithms to compute it [6,10,11,9,14,4,7,2]. With a large collection of different algorithms, a programmer needing to use matrix multiplication should be able to find an algorithm that meets his requirements in both parallel machine architecture and data placement.

A parallel architecture that has become widely available to programmers is the binary n-cube or hypercube [19]. Several algorithms for matrix multiplication on hypercubes appear in the literature [6,14,7,18,2,8,13,3]. These algorithms fall into two categories. The first kind of algorithm uses the data dependencies of the standard O(n^3) sequential algorithm and schedules the required computations in the cube. These algorithms generally consist of two stages, one that moves the data to the correct processors and one that does scalar multiplies and distributed sums. There are algorithms for both O(log n) time using n^3 processors and

Correspondence to: P.A. Nelson, Computer Science Department 9062, Western Washington University, Bellingham, WA 98225-9062, USA. Email: phil@cs.wwu.edu

* Supported in part by National Science Foundation Grant DCR-8416878 and by Office of Naval Research Contract No. N00014-85-K-0328.

$O(n)$ time using n^2 processors [6]. The second category of algorithm takes algorithms for other architectures and shows how to efficiently map them to hypercubes. They typically start with an algorithm for a mesh connected computer and provide a mapping to a cube. Some provide enhancements to the algorithm to take advantage of the communication bandwidth available on the hypercube [2,13].

This paper presents a divide-and-conquer matrix multiplication algorithm [14,15] whose *natural* communication pattern is the hypercube. Section 2 presents the basic algorithm for $n \times n$ matrices using both n^2 and $n^3/2$ processors having running times of $O(n)$ and $O(\log n)$ respectively. Section 3 deals with the real issues faced by programmers, including communication time versus computation time, program overhead, fewer than n^2 processors and non-square matrices. It includes a report of timings of the algorithm taken on an Intel iPSC/1 [5]. Finally, Section 4 compares this algorithm with two other hypercube matrix multiplication algorithms.

2. The basic algorithm

Assuming that we have enough processors, consider the multiplication of two dense $n \times n$ matrices,

$$AB = C$$

using n^2 processors, and assuming $n = 2^k$ for some constant k. (A version of this algorithm for shared memory machines was sketched by Horowitz and Zorat [9].) The processors are viewed as an $n \times n$ array where the processors are labelled PE_{ij} for $1 \le i, j \le n$. The matrices A and B are initially distributed in the n^2 processors such that a_{ij} and b_{ij} are contained in PE_{ij}. After computing the product we have c_{ij} contained in PE_{ij}.

To begin with, consider the 2×2 case. PE_{11} contains a_{11} and b_{11}. To compute c_{11} the values a_{12} and b_{21} are needed. Similarly, all other processors need only 2 elements not already stored at that processor. To provide for direct communication, a grid interconnection structure is used. The processors then send their a_{ij} value to the other processor in the same row, and their b_{ij} value to the other processor in the same column as shown in *Fig. 1*. After this communication, each processor, PE_{ij}, has all the data required to compute c_{ij}.

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} = \begin{bmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{bmatrix}$$

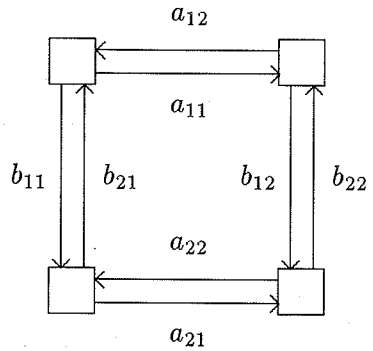

Fig. 1. 2×2 product and communication structure.

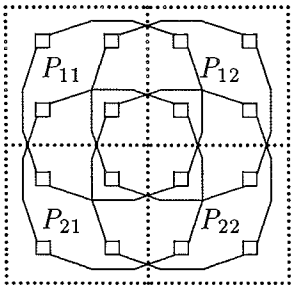

Fig. 2. 4×4 connections.

Now consider the $n \times n$ case. The algorithm uses Strassen's [20] matrix decomposition where two $n \times n$ matrices can be viewed as two 2×2 matrices of $(n/2) \times (n/2)$ matrices. The 2×2 matrices are then multiplied using matrix product and matrix addition on $(n/2) \times (n/2)$ matrices.

Let A_{11} be the upper left $(n/2) \times (n/2)$ submatrix of A. Similarly define the other 3 submatrices, A_{12}, A_{21}, and A_{22}, the submatrices of B, the submatrices of C, and the subarrays of the processors, P. Then A_{ij} and B_{ij} are contained in P_{ij}, $1 \leq i, j \leq 2$. As in the 2×2 case, A_{12} and B_{21} are required to compute C_{11}. If the corresponding processors in P_{11} and P_{12} are directly connected (see *Fig. 2*), A_{12} can be sent to P_{11} in parallel with one communication step. B_{21} can be sent to P_{11} using a similar connection scheme in one communication step. The full connection structure connects PE_{ij} with both $PE_{i \pm (n/2)j}$ and $PE_{ij \pm (n/2)}$. With A_{11}, B_{11}, A_{12}, and B_{12} in P_{11}, C_{11} can be computed by doing two $(n/2) \times (n/2)$ matrix products and one matrix addition. The two $(n/2) \times (n/2)$ matrix multiplies are done sequentially using this same algorithm to do the matrix multiplies. These $(n/2) \times (n/2)$ matrix multiplies are done using only the processors in P_{11}. The matrix addition is performed element by element in P_{11} with no communication. The recursion will stop when a 2×2 matrix product is done. C_{12}, C_{21}, and C_{22} are computed in parallel with C_{11} in the same way using the processors in P_{12}, P_{21}, and P_{22} respectively.

Figure 3 gives pseudo code for a single processor. There are several points of interest. First, each recursion level needs to have local storage for the values received from the neighbor processors. These can not be global variables because they need to be preserved across the recursive calls. Since there are log n levels of recursion, this algorithm requires O(log n) extra storage at each processor. Also, each recursion level requires its own interconnection structure to be able to communicate with the correct processors. The complete interconnection structure, supplying an edge for every communication in the algorithm on every level of recursion, is a hypercube. The next point of interest is the use of the phrase 'lower left or upper right'. In the 4×4 case (*Fig. 2*), all processors in P_{21} are in the lower left and all processors in P_{12} are in the upper right. For the 2×2 case (*Fig. 1*), PE_{21} is the lower left processor and PE_{12} is the upper right processor. Finally, the pseudo code does not include any specific code for recursion control. A real implementation will need a method to identify the recursion level.

The time required for this algorithm is $O(n)$. Consider the time required by a single processor. The time for the 2×2 case is

$$t(2) = 2t_c + t_a + 2t_m, \tag{1}$$

where t_c is the time to communicate a single data element, t_a is the time for a scalar addition,

```
Hypercube_Matrix_Multiply (A, B, C)
  /* Local Data */
  tempA, tempB, and tempC.

  /* Communication step. */
  Send A to correct horizontal neighbor
  Get tempA from correct horizontal neighbor
  Send B to correct vertical neighbor
  Get tempB from correct vertical neighbor

  /* Computation or recursion. */
  if doing a 2 by 2 then
    if processor is lower left or upper right of the 2 by 2 then
      C = A * tempB + tempA * B
    else
      C = A * B + tempA * tempB
    end if
  else
    if node is in the lower left or upper right of current group then
      Hypercube_Matrix_Multiply (A, tempB, C)
      Hypercube_Matrix_Multiply (tempA, B, tempC)
    else
      Hypercube_Matrix_Multiply (A, B, C)
      Hypercube_Matrix_Multiply (tempA, tempB, tempC)
    end if
    C = C + tempC
  end if
```

Fig. 3. Pseudo code.

and t_m is the time for a scalar multiplication. For the $n \times n$ case, the recurrence relation for the time is

$$t(n) = 2t_c + t_a + t_o + 2t\left(\frac{n}{2}\right), \tag{2}$$

where t_o is the overhead time for each recursion level. We include in the overhead time all time required for recursion control and for identifying if the processor is in the lower left or the upper right of the current processor group. The closed form for the time is

$$t(n) = (2n-2)t_c + (n-1)t_a + (n-2)t_o + nt_m. \tag{3}$$

Therefore the time for this matrix multiply algorithm using n^2 processors is $O(n)$.

A simple modification and the addition of more processors, produces an algorithm which runs in $O(\log n)$ time. (We again assume we have the we have enough processors.) This is achieved by evaluating all 8 $(n/2) \times (n/2)$ matrix products at the same time, instead of 4 at a time. *Figure 4* shows the connections needed for an 8×8 matrix product. Each box represents a 8×8 plane of processors. The algorithm starts with n^2 processors active, represented by plane 0. These processors contain the original matrices, A and B. The processors communicate as in the $O(n)$ algorithm. At this point, each $(n/2) \times (n/2)$ block of processors has two matrix products to compute. One of the products is sent to a set of previously inactive processors, represented by plane 2. The connection shown between plane 0 and plane 2 has a communication channel for each of the processors, connecting corresponding processors in

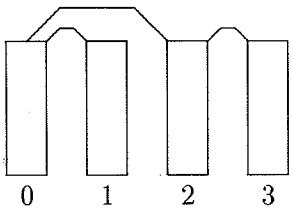

Fig. 4. 8×8 connections for O(log n) matrix product.

each plane. This doubles the number of active processors. The same algorithm is used to compute the 'new' products. Notice that each processor needs only a constant amount of memory. This is because only one of the two $(n/2)\times(n/2)$ remains in a processor and the other one does not need to be remembered during the recursive call. After the $(n/2)\times(n/2)$ products have been computed, the processors that were sent the second product, send back their result. The results of the two products are added element by element to form the result of the $n\times n$ matrix product. The recursion for this algorithm stops when a 2×2 product is to be computed. Each processor involved in a 2×2 multiply does both scalar multiplications and the one scalar addition.

To see that the execution time is O(log n), consider the time of a single processor in plane 0. The 2×2 case is the same as Eq. 1. For the $n\times n$ case, the recurrence relation is

$$t(n) = 5t_c + t_a + t_o + t\left(\frac{n}{2}\right) \tag{4}$$

where constants measure the same quantities as before. This recurrence relation is for the original n^2 active processors. The $5t_c$ comes from two t_c's for the original communication, two t_c's from sending one subproblem to a 'new' processor and a t_c for getting the result back from the 'new' processor. The closed form is

$$t(n) = (5(\log n - 1) + 2)t_c + \log n t_a + (\log n - 1)t_o + 2t_m. \tag{5}$$

This algorithm uses $n^3/2$ processors. It starts with n^2 active processors. After the initial communication, the n^2 processors are divided up into 4, $(n/2)\times(n/2)$ sections, each having two matrix products to compute. Every processor sends two values, its part of one matrix product, to an inactive processor, thus activating it. This doubles the number of processors. We now have 8, $(n/2)\times(n/2)$ problems using $2n^2$ processors. Each matrix product is then computed by a 'recursive call'. This is one recursion level. At each successive recursive level the number of active processors is doubled. There are log $n-1$ levels of recursion. This gives $2^{\log n-1}n^2$ or $n^3/2$ active processors at the evaluation of the 2×2 products.

What we really have is a single algorithm for matrix multiply on a hypercube that uses $n^3/2$ processors and takes O(log n) time. The O(n) version is just a result of reduced parallelism. This algorithm is not a parallelized version of the standard sequential algorithm or a mapping to a hypercube of an algorithm for a different architecture.

3. Implementation issues

While the basic algorithm is conceptually beautiful, many practical uses of matrix multiplication do not conform to its presuppositions. Several things appear to make this algorithm unusable in practice. First, many applications do not use $n\times n$ matrices of exactly $n=2^k$. Also, applications requiring the multiplication of non-square matrices are common. Even if we did have $n=2^k$ for $n\times n$ matrix multiply and n^2 or $n^3/2$ processors, the communication

overhead of most available hypercube computers is so large that having one data element from each matrix in a processor would make the communication time the largest component of the final runtime. Such programs would not be very efficient from the standpoint of balancing communication and computation. Also, with a single data element from each matrix in a processor, the overhead would be a significant contributor to the total time. Finally, for many applications having n^2 processors is impractical. Having n^3 processors is even more impractical.

3.1. Contraction

We first look at the problem of having fewer than n^2 processors for the $n \times n$ matrix multiply. The method of solving the general problem, having fewer real processors than the algorithms needs, is called *contraction* [1,17]. In contracting an algorithm, one must place data for more than one of the algorithm's processors, called logical processors, on a physical processor. On the physical processor, the logical processors could become processes or a single process could do the combined work of all logical processors.

Nelson [15,16,17] describes a method for performing good contractions. It assumes that each logical processor in the algorithm has the same load and that a good contraction will have the physical processors load balanced. This implies that the logical processors will be evenly distributed to the physical processors. Nelson shows that minimizing communication between real processors will often produce the best contraction. This is based on the fact that two logical processors on the same physical processor can 'communicate' using local variables, but two logical processors on different physical processors must communicate using processor to processor communication. Since the use of a local variable for communication is much faster than processor to processor communication, we expect that a good contraction will map logical processors connected by the highest volume communication channels (edges) to the same physical processor. Since these contractions are performed using the high volume edges, the contracted algorithm may have a different processor to processor communication graph than the original algorithm.

To apply this method to the contraction of the matrix multiplication algorithm, we must observe the communication patterns of the algorithm. But before we can talk about which edges have the most communication, we will need to talk about the hypercube structure. Hypercube multiprocessors are built in cubes of order k which have 2^k processors, each of which have a number between 0 and $2^k - 1$. Each processor is an order 0 cube, pairs of processors, $2i$ and $2i + 1$, are order 1 cubes, and similarly for each block of 2^j processors, $2^j i$, $2^j i + 1, \ldots, 2^{j+1} i - 1$ is a j cube for $1 \leq j \leq k$. An edge between two processors is an *order l edge* if it connects corresponding processors in order $l - 1$ cubes and therefore connects processors whose numbers differ by 2^{l-1}.

The $O(n)$ time matrix multiply algorithm runs in an order 2 log n cube. We choose to map the matrix multiply algorithm into the hypercube using row-major ordering. This places each row in an order log n cube. (See *Fig. 5.*) The longest edge connecting processors that contain the same row of data is an order log n edge. Similarly, the longest edge connecting processors that contain the same column of data is an order 2 log n edge.

A simple analysis of the algorithm reveals which of these edges have the most messages. At the first level of recursion, the order k and $2k$ edges were used to send a message each way. These messages are the only communication at the first level of recursion. At the second level of recursion, two matrix products are computed using the order $k - 1$ and $2k - 1$ edges. Each of the products send one message each way on those edges yielding twice as many messages on the $k - 1$ and $2k - 1$ edges as the k and $2k$ edges. At recursion level l, there are 2^{l-1} messages sent over the order $k - (l - 1)$ and $2k - (l - 1)$ edges. The recursion stops at the

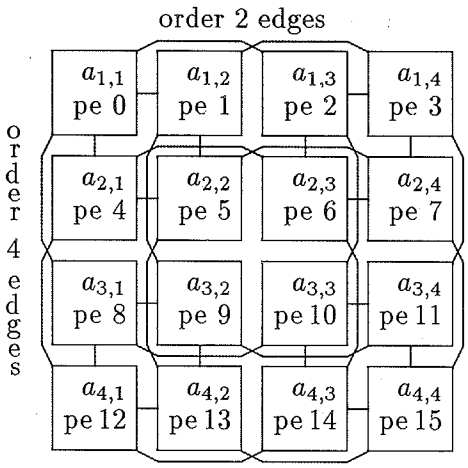

Fig. 5. Mapping of a 4×4 Matrix, a, into a 4-cube.

log n level of recursion where $n/2$ matrix multiplies are done by the order 2 cubes. These order 2 cubes use the order 1 and $k+1$ edges and there are $n/2$ messages passed each way over these edges.

For a contraction of this algorithm we will assume that we are contracting a larger hypercube into a smaller one. Any contraction that maps the $n\times n$ matrix multiply into a cube with $p=2^m$ processors, where $m<2k$ will map n^2/p processes to each physical processor. Using the principle of mapping the busiest edges to 'internal edges,' we could map either the order 1 edges or the order k edges to internal edges. We choose to map the order k edges first. This is done by placing data for the logical processors connected by those edges to the same physical processor. The next edges to be mapped into the same processor would be the order 1 edges. The mapping would proceed in similar fashion until data for n^2/p logical processors are mapped to each physical processor. *Figure 6* shows the data placement using this mapping for an 8×8 matrix multiplication (which uses a 6 cube for the full algorithm) to a 2 cube and a three 3 cube.

Once this mapping is completed, we now need to consider the resulting processor to processor communication graph. This graph is defined by the 'logical edges' that connect processes in different physical processors. It will be the actual communication graph for the contracted matrix multiply algorithm.

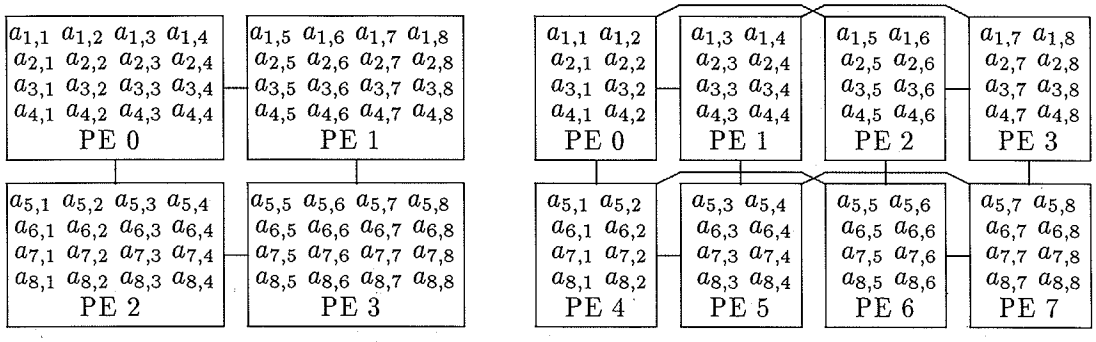

Fig. 6. Mappings of an 8×8 matrix.

For our contraction, the resultant communication graph is an order m cube. *Figure 6* also shows the required processor to processor communication links. The resulting algorithm can either implement the full matrix multiply algorithm with n^2/p real processes at each processor or have a single process at each processor contain all the data and do the proper communication and computation. We choose the second approach as the more practical because the data contained at each processor is just a sub-matrix of the original matrix.

In fact, the approach of keeping a sub-matrix of the original matrix at each processor yields other advantages. When m is even, the sub-matrix mapped to a processor is square and can be considered a single element of a $\sqrt{p} \times \sqrt{p}$ matrix. In *Fig. 6(a)*, the $\sqrt{p} \times \sqrt{p}$ matrix is a 2×2 matrix and each element of the 2×2 matrix is a 4×4 matrix. The parallel algorithm is then used to multiply the matrix of sub-matrices. When the parallel algorithm algorithm communicates an element, it is a complete matrix that can be sent in a single message. When the parallel algorithm multiplies single elements, it is simply sequential matrix multiply of a smaller matrix. This sequential matrix multiply can be performed using any method of computing matrix multiply on a sequential processor. This would include any method of utilizing a pipelined sequential processor for matrix multiply and BLAS [12].

When m is odd (*Fig. 6(b)*), each processor contains a sub-matrix with twice as many rows as columns. The parallel algorithm must be modified to run on the odd cube by having the top level communication use the order $\lceil m/2 \rceil$ and order m edges. At the lowest level of recursion, communication happens only on the order 1 edges. At each communication the entire sub-matrix is sent. The scalar multiply of the original algorithm is now a more complex operation involving two sub-matrices of A and one sub-matrix of B. The resulting C can be computed by viewing each sub-matrix as two square sub-matrices and then performing 4 sequential matrix multiplies and a two matrix additions of the smaller square sub-matrices.

The time for the contracted algorithm, $t(n, p)$ can be derived from Eq. 3. We must remember that the original algorithm was using n^2 processors and the contracted algorithm has only p processors. For the even m case, we have

$$t(n, p) = \left(2\sqrt{p} - 1\right)T_c + \left(\sqrt{p} - 1\right)T_a + \left(\sqrt{p} - 2\right)T_0 + \sqrt{p}\,T_m, \tag{6}$$

where T_c is the communication time for sending the complete sub-matrix, T_a is the time for sub-matrix addition, T_o is the overhead on each level of recursion and T_m is the time for sub-matrix multiplication. Since each sub-matrix is of size n^2/p, we have $T_c = (n^2/p)t_c$, $T_a = (n^2/p)t_a$, and $T_m = ((n-1)n^2/p)t_m$ where t_c, t_a, and t_m are the same as in Eq. 3. Also the overhead, T_o should be the same as t_o since that has to do with recursion which is related to the the number of processors and not the size of the sub-matrices. Substituting into Eq. 6 we get

$$t(n, p) = \left(2\sqrt{p} - 2\right)\frac{n^2}{p}t_c + \left(n + \sqrt{p} - 2\right)\frac{n^2}{p}t_a + \left(\sqrt{p} - 2\right)t_0 + \frac{n^3}{p}t_m. \tag{7}$$

3.2. Rectangular matrices

An approach similar to the solution for the contraction problem can be used to solve the problems of multiplication of $n \times n$ matrices where $n \neq 2^k$ and the multiplication of $n \times m$ by $m \times l$ matrices. For multiplication of an $n \times n$ matrices on a order $2r$ cube (an even order cube), where $p = 2^{2r}$ and $\sqrt{p} \leq n$, each $n \times n$ matrix is augmented with zeros to produce a $m \times m$ matrix where $m = \sqrt{p} * \lceil n/\sqrt{p} \rceil$. This yields at most $\sqrt{p} - 1$ extra rows and columns. The $m \times m$ matrix is then divided into sub-matrices of size $m/\sqrt{p} \times m/\sqrt{p}$. Each sub-matrix is then a single element in the standard $\sqrt{p} \times \sqrt{p}$ multiplication algorithm. The time is Eq. 7 with m replacing n in the equation.

For multiplication of $n \times n$ matrices on a 2r-1 order cube (an odd order cube), where $p = 2^{2r-1}$ and $2^r \leq n$, each $n \times n$ matrix is also augmented with zeros to produce an $m \times m$ matrix where $m = 2^r * \lceil n/2^r \rceil$. Each $m \times m$ matrix is then divided into sub-matrices of size $2m/2^r \times m/2^r$. The mapping of the sub-matrices and the computation is the same as with an odd order cube in contraction.

For multiplication of non-square matrices of size $n \times m$ by $m \times l$ on a 2r cube, where $p = s^{2r}$ and $\sqrt{p} \leq \text{minimum}(n, m, l)$, the matrices are augmented with zeros to produce matrices of size $n_1 \times m_1$ and size $m_1 \times l_1$, where $x_1 = \sqrt{p} * \lceil x/\sqrt{p} \rceil$ where x is n, m, and l. Each matrix is then partitioned into sub-matrices of size $n_1/\sqrt{p} \times m_1/\sqrt{p}$ and $m_1/\sqrt{p} \times l_1/\sqrt{p}$ respectively. The full multiplication is completed by the multiplication of two $\sqrt{p} \times \sqrt{p}$ matrices by the original algorithm where each element of the two matrices are sub-matrices of sizes $n_1/\sqrt{p} \times m_1/\sqrt{p}$ and $m_1/\sqrt{p}$ and $m_1/\sqrt{p} \times l_1/\sqrt{p}$ and can be multiplied by any sequential algorithm. The odd cube multiplication is done in a similar way as before.

3.3. Implementation results

The full algorithm for $n \times m$ by $m \times l$ multiplication was implemented in C on an Intel iPSC/1 [5] for both even and odd cubes. The test implementation loaded two floating point matrices from the host computer to the cube, computed the matrix multiply, and the sent the result to the host. Timings were taken of just the time to compute the result. No time was included for host to cube communication of the matrices. Since the iPSC/1 is an old and slow multiprocessor, *Table 1* reports the timings as speedups using p processors. These timings are the average of 10 runs and each run reported the maximum time taken by any processor. Timing differences in the I/O subsystem caused the timings to vary from run to run. The traditional $O(n^3)$ algorithm was run on a single processor of the iPSC/1 for speedup calculations. Since no communication was done while computing matrix multiply on a single processor, all runs had the same time.

The results in *Table 1* demonstrate several interesting facts. First, when a processor has only a few data elements, the communication cost are much greater than the computation costs. This provides a slow-down for the 8×8 matrix on a 32 processor cube. When enough data elements are at the same processor, the computation costs are equal or greater than the communication costs and thus provide better speedups. This is expected by noticing that the $(n^3/p) * t_m$ is the largest term in Eq. 7. Also, to achieve the speedups reported, the overhead must be small. Again, this is expected since $(\sqrt{p} - 2)t_0$ is the smallest term in Eq. 7. Finally, these timings and the timings in *Table 2* show super-linear speedup. On the iPSC/2, this is due to cache memory effects. On a single processor, the number of cache hits are low due to a large data set. When the data is distributed over several processors, the number of cache hits increases, allowing for faster execution. On the iPSC/1, we are not aware of any cache

Table 1
Experimental speedup on an iPSC/1

p	Matrix size								
	8×8	16×16	24×24	32×32	40×40	48×48	56×56	64×64	32×128×32
2	2.17	2.41	2.45	2.46	2.46	2.46	2.46	2.46	2.46
4	2.50	4.76	4.82	4.97	4.95	4.98	4.98	5.00	4.98
8	2.83	6.40	7.75	8.51	9.27	9.40	9.51	9.56	9.54
16	1.26	8.21	9.76	14.09	15.85	16.70	18.24	18.64	18.10
32	0.86	5.56	11.87	17.84	24.11	25.75	29.46	30.93	18.12

Table 2
Comparison speedups on an iPSC/2

p	Matrix size and algorithm			
	Nelson		Fox	
	128×128	256×256	128×128	256×256
4	4.36	4.36	3.62	3.61
16	16.01	17.52	12.80	14.36

memory, but expect that there is some hardware reason for slower execution on large data sets.

4. Comparison to other algorithms

In comparing our algorithm to other algorithms we will consider two aspects of the algorithms, the computation time and ease of use as a subprogram for a larger algorithm. We believe that the second aspect is of major importance due to the fact that matrix multiply is a tool to solve a larger problem. It is possible that the data placement on processors may differ between the 'main' algorithm and the matrix multiply algorithm. If the data placements do differ, then part of the cost in using the algorithm comes from the required permutation to get the data ready.

Fox et al. [7] describe an algorithm that uses a mesh interconnection structure along with subblock decomposition of the matrices. Their algorithm is based on row broadcasts and column shifts of the submatrices. The time for their algorithm is

$$t(n, p) = \frac{2n^3}{p} t_f + \frac{2n^2}{\sqrt{p}} t_c + \sqrt{p}\left(\sqrt{p} - 1\right) * t_s \tag{8}$$

where t_f is the time for a floating point operation, t_c is the time to communicate a single data element, and t_s is the time required to start the 'data pipelines' per processor.

In comparing the time, we see that they combine the time for scalar multiply and addition into a single term and ignore any overhead for processing. Their algorithm requires the same number of scalar multiplies and additions as ours. If we ignore the overhead and combine the scalar operations, our time is the same as their first two terms of Eq. 8. Our algorithm will be faster by an additive term of $\sqrt{p}(\sqrt{p} - 1) * t_s$. Also, if the data placement is the same as our algorithm, then their algorithm will do communication to a processor that is not directly connected. Our algorithm always does communication with directly connected processors. *Table 2* gives experimental results comparing our algorithm to the Fox algorithm. The timings were taken on and iPSC/2 and are reported as speedups. (The timings were taken by Calvin Lin at the University of Washington.)

Berntsen [2] describes a similar algorithm to the Fox algorithm that uses only 'nearest neighbor' communication. The time for this algorithm is the same as our algorithm. To achieve the nearest neighbor communication, the algorithm maps the submatrices to the processors using a Grey code scheme. (The Fox algorithm can also be mapped with the Grey code scheme to get nearest neighbor communication.)

Berntsen also gives an algorithm with better asymptotic communication costs. This algorithm requires one matrix to be decomposed by rows while the other one is decomposed by columns. Corresponding rows and columns are mapped to sub-cubes. The result matrix will not be in the proper places for use as input to another matrix multiply. This will require more

communication to to get the result ready for another matrix multiply which was not part of their timings. Our algorithm's results are ready for use in further matrix multiplies without any data movement. This includes even the rectangular matrices. Berntsen's algorithm looses the communication advantage when used with multiple matrix multiplies. No timings were available for the Berntsen algorithms.

5. Conclusion

We have presented a parallel matrix multiplication algorithm whose *natural* communication pattern is the hypercube. The algorithm has computational complexities of $O(\log n)$ using n^3 processors, $O(n)$ using n^2 processors and $O(n^3/p)$ using $p < n^2$ processors. The distribution of data to processors in the hypercube is such that sub-matrices of the original matrices are placed at each processor. Furthermore, the distribution of the submatrices to the processors uses row major ordering instead of a Grey code. The results are in the correct processors to use the results in further matrix multiplications without any further communication to rearrange the data. During the computation, the sub-matrices are sent in a single communication to directly connected processors in the hypercube. The submatrices are also involved in sequential matrix multiplication at each processor. We also provided experimental evidence that this algorithm performs well on both the iPSC/1 and the iPSC/2 hypercubes.

References

[1] F. Berman and L. Snyder, On mapping parallel algorithms into parallel architectures, *Proc. 1984 Internat. Conf. on Parallel Processing* (1984) 307–309.

[2] J. Berntsen, Communication efficient matrix multiplication on hypercubes, *Parallel Comput.* 12 (3) (1989) 335–342.

[3] C.H. Chang, $o(n)$ matrix multiplication algorithm for hypercube machines, *Electron. Letters* 27 (25) (1991) 2398–2399.

[4] M.Y. Chern and T. Murata, Efficient matrix multiplications on a concurrent data-loading array processor, *Proc. 1983 Internat. Conf. on Parallel Processing* (1983) 90–94.

[5] Intel Corporation, iPSC System Overview Manual. Intel Corporation, Beaverton, OR, 1986.

[6] E. Dekel, D. Nassimi and S. Sahni, Parallel matrix and graph algorithms, *SIAM J. Comput.* 10 (4) (1981) 657–675.

[7] G.C. Fox, S.W. Otto and A.J.G. Hey, Matrix algorithms on a hypercube I: Matrix multiplication, *Parallel Comput.* 4 (1) (1987) 17–31.

[8] C. Ho and S.L. Johnson, Matrix multiplication on boolean cubes using generic communication primitives, in: *Parallel Processing and Medium Scale Multiprocessors* (Soc. for Industrial and Applied Mathematics, 1989) 108–156.

[9] E. Horowitz and A. Zorat, Divide-and-conquer for parallel processing, *IEEE Trans. Comput.* C-32 (6) (June 1983) 582–585.

[10] H.T. Kung and C.E. Leiserson, Systolic arrays (for vlsi), in: I.S. Duff and G.W. Stewart, eds., *Sparse Matrix Proceedings 1978* (Soc. for Industrial and Applied Mathematics, 1979) 256–282.

[11] S.Y. Kung, K.S. Arun, R.J. Gal-Ezer and D.B. Bhaskr Rao, Wavefront array processor: Language, architecture, and applications, *IEEE Trans. Comput.* C-31 (11) (Nov. 1982) 1054–1065.

[12] C.L. Lawson, R.J. Hanson, D.R. Kincaid and F.T. Krogh, Basic linear algebra subprograms for fortran usage, *ACM Trans. Math. Software* 5 (3) (1979) 308–323.

[13] K.K. Mathur and S.L. Johnsson, Multiplication of matrices of arbitrary shape on a data parallel computer, Technical report, Thinking Machines Corp., December 1991.

[14] P.A. Nelson, A non-systolic matrix product algorithm, Technical report, Department of Computer Science, University of Washington, November 1985.

[15] P.A. Nelson, Parallel programming paradigms, PhD thesis, University of Washington, Seattle, WA, 1987.

[16] P.A. Nelson and L. Snyder, Programming solutions to the algorithm contraction problem, in: *Proc. 1986 Internat. Conf. on Parallel Processing* (IEEE Computer Society Press, Silver Spring, MD, August 1986) 258–261.

[17] P.A. Nelson and L. Snyder, Programming paradigms for nonshared memory parallel computers, in: L. Jamieson, D. Gannon and R. Douglass, eds., *The Characteristics of Parallel Algorithms* (MIT Press, Cambridge, MA, 1987) 3–20.
[18] L.M. Ni and C. King, On partitioning and mapping for hypercube computing, *Internat. J. Parallel Programming* 17 (6) (1988) 475–495.
[19] C.L. Seitz, The cosmic cube, *Commun. ACM,* 28 (1) (Jan. 1985) 22–33.
[20] V. Strassen, Gaussian elimination is not optimal, *Numer. Math.* 13 (1969) 354–356.