A Matrix Product Algorithm and its Comparative Performance on Hypercubes

Calvin Lin Lawrence Snyder

Department of Computer Science & Engineering
University of Washington
Seattle, WA 98115

A matrix product algorithm is studied in which one matrix operand is transposed prior to the computation. This algorithm is compared with the Fox-Hey-Otto algorithm on hypercube architectures. The Transpose algorithm simplifies communication for nonsquare matrices and for computations where the number of processors is not a perfect square. The results indicate superior performance for the Transpose algorithm.

1 Introduction

Matrix multiplication is both a fundamental computation for engineering and scientific applications and an extensively studied parallel computation. However, most studies of parallel computation seek an optimal algorithm with respect to a particular architecture. Unfortunately, parallel machines exhibit considerable architectural diversity. Anyone who has attempted to port a program from one machine to another knows that these differences can greatly affect performance. Though basic computations such as matrix multiplication can be implemented in a machine specific manner and provided to users through libraries, most computations are not general enough to justify such effort. These computations must therefore be made as machine-independent as possible. Thus, we are interested in identifying those features that make a computation machine-dependent or machine-independent. In this study we use matrix product as a representative computation and study ways in which to make it machine independent.

For a given problem no single algorithm is fastest for all machines and all program parameters. So what does it mean to have a machine-independent computation? At an algorithmic level, a machine-independent computation is represented by an *Algorithm of Choice*: an algorithm that is competitive on a maximal set of MIMD multiprocessors. Here "competitive" is with respect to the best observed performance for a given machine. The results reported here can therefore be viewed in the context of the search for an Algorithm of Choice for matrix multiplication.

A number of dense matrix multiplication algorithms have been proposed in the literature [2, 3, 7, 8, 9, 10, 13, 14, 15], but most are targeted for a particular architecture or communication structure. For example, Kung and Leiserson's elegant systolic algorithm relies on fine grained movement of data, which is inefficient for many multiprocessors. Similarly, Johnsson and Ho's work is too hypercube-specific to be an Algorithm of Choice. We note, however, that their detailed cost analysis will likely be useful for developing future algorithms.

This paper focuses on two algorithms. The first, due to Fox, Hey and Otto [3], appears to be the most popular algorithm for hypercube multiprocessors. The second, which we call the Transpose algorithm, extends the common sequential algorithm to work on distributed memory computers. The performance of these algorithms is compared on three hypercube multiprocessors – two instances of the Intel iPSC/2 and the NCUBE/7.

2 The Fox-Hey-Otto Algorithm

The problem of interest is to compute the matrix product of a $p \times q$ **A** matrix and a $q \times r$ **B** matrix, placing the result in a $p \times r$ **C** matrix.

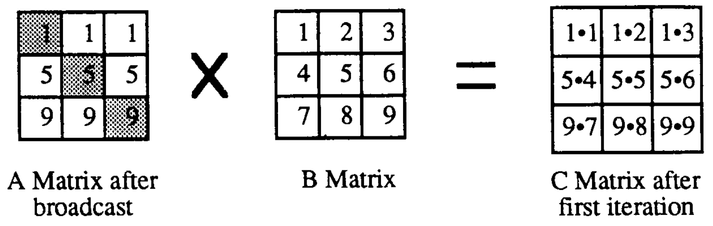
We describe the Fox-Hey-Otto algorithm [3] assuming that each process owns square submatrices of the **A**, **B** and **C** matrices, but results are gathered for the more general case. Each iteration of the algorithm has three phases (see Figure 1). First, the diagonal of the **A** matrix is broadcast across each row of processes.

0-8186-2775-1/92 $3.00 © 1992 IEEE

The Fox-Hey-Otto Algorithm:

```
for (i=0; i<n; i++)
{
    Broadcast Diagonal of A;
    Compute local matrix product;
    Rotate Rows of B North;
}
```

First Iteration of the Fox-Hey-Otto Algorithm:



The Transpose Algorithm:

```
for (i=0; i<n; i++)
{
    Compute local matrix product;
    Accumulate sums along each row;
    Rotate Rows of B North;
}
```

First Iteration of the Transpose Algorithm

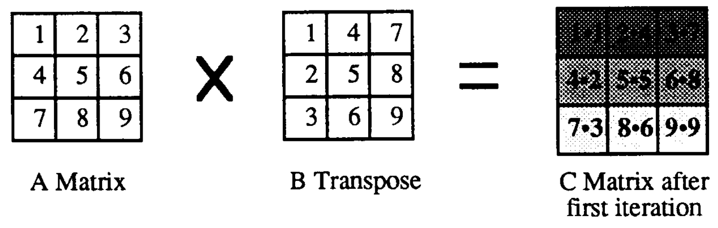


Figure 1: The FHO and Transpose Algorithms

Then, local matrix products are computed by each process. Finally, in preparation for the next iteration, the rows of the **B** matrix are circularly rotated to the North. Figure 1 shows the result of one iteration for a 3×3 matrix. Of course, since matrix multiplication can be recursively defined in terms of square submatrices, each element in this 3×3 matrix could represent an entire submatrix. Matrix multiplication can be viewed as a collection of independent dot-products, where the operands are rows and columns of the **A** and **B** matrices, respectively. We see that each iteration of the algorithm computes one portion of each of these dot-products. After n iterations, where n is the length of each operand, all dot-products have been computed and the computation is complete. In the worst case $n = q$; in the best case $n = \sqrt{P}$, where P is the number of processes.

If the number of processes is not a perfect square, the submatrices owned by each process are no longer square. In such cases we can apply the algorithm to *subblocks* – where a subblock is the greatest common divisor of the width and height of each process' submatrix – and multiplex each process across multiple subblocks. Unfortunately, this approach takes O(s) iterations, where s is the number of subblocks in each row of the **A** matrix. While it is possible to optimize the communication for the Rotates (using twice as much memory to hold **B**), there appears to be no way to reduce the required number of Broadcasts. One typical method of reducing communication is to combine messages having the same source and destination into a single message. This does not work with subblocks because there are always cases where the subblocks of a given process must be broadcast to different destinations.

When the **A** and **B** matrices are not perfect squares the matrices can be padded with 0's to create square matrices. This results in poor performance when the aspect ratio is large [3].

3 The Transpose Algorithm

We now describe an algorithm that assumes the **B** matrix is stored as its transpose [11]. In its simplest form, rows of **A** and **C** and columns of **B** are assigned to each process. The submatrices of **A** and **C** do not move, while the values of **B** are circularly rotated North so that all processes can eventually access all columns of the **B** matrix. Each iteration of the algorithm computes the matrix product for the diagonal submatrices of **C**.

For square decompositions, this algorithm proceeds as before except now each process only computes a partial dot product. Figure 1 shows how the first iteration computes the diagonal of the **C** matrix. Since each dot product computation is distributed across multiple processes, the results of the local matrix products must be accumulated and summed along each row. A nice property of this algorithm is that as long as the **A** and **B** matrices are compatible for matrix multiplication, the submatrices stored on each process will be compatible. There is no need to multiplex the algorithm to work on smaller subblocks unless this is desirable for reasons of data locality. Furthermore, this algorithm handles non-square matrices with the same ease as square matrices. In particular, non-square matrices need not be padded as in the Fox-Hey-Otto algorithm. On the other hand, a disadvantage of this algorithm is the possible need to perform matrix transposes before and after the computation (the second transpose is necessary only if the **B** matrix is reused in other portions of the application and memory constraints force the transpose to occur in-place).

4 Results

The two algorithms were implemented in C using the Phase Abstractions programming model [1, 5, 16]. The resulting programs were then compared on three hypercube multiprocessors: two Intel iPSC/2's and the nCUBE/7. The Intel machines have 32 nodes with 80386 processors [6]. They differ in that one has, on each node, an iPSC SX floating point accelerator and 8MB of memory (we refer to this as iPSC/2 F), while the other (iPSC/2 S) has the slower Intel 80387 floating point coprocessor and only 4MB of memory. On the nCUBE/7 each of the 64 nodes has a custom main processor and 512 KB of memory. This, too, is a distributed memory hypercube [12].

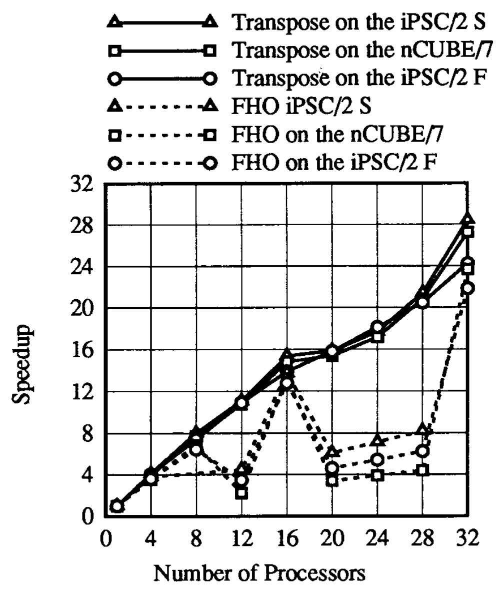


Figure 2: FHO vs. Transpose on various machines

Figure 2 gives speedup curves for the two programs on our three hypercubes. In all cases, the **A** and **B** matrices are 128 × 128 double precision values. We calculate speedup based on a sequential matrix multiplication algorithm running on each of the target machines. The raw execution times are presented in Tables 1, 2, and 3. These values do not include initialization costs (such as time to allocate and read in the matrices) because these are highly system dependent and not fundamental to the parallel aspects of the algorithm.

When the number of processors is a perfect square, the Transpose algorithm is slightly faster *even with the cost of transpose added in.* In the other cases, the Fox-Hey-Otto algorithm sees poor performance because of small subblock size. As explained in Section 2, the communication costs of the Fox-Hey-Otto algorithm grows as O(s), where s is the number of subblocks in each row of the **A** matrix. Table 2 shows that the subblock size used in the Fox-Hey-Otto algorithm is inversely related to execution time.

5 Conclusion

We have presented a matrix multiplication algorithm and experimental evidence that it performs better than the Fox-Hey-Otto algorithm on three hypercube multiprocessors. The new algorithm is primarily superior in terms of generality. It performs well for all numbers of processors and for all matrix shapes.

Acknowledgments. We would like to thank Walter Rudd and the Oregon Advanced Computing Institute for providing access to their nCUBE/7 and iPSC/2. This research was supported in part by Office of Naval Research Contract N00014-89-J-1368.

References

[1] G. Alverson, W. Griswold, D. Notkin, and L. Snyder. A flexible communication abstraction for nonshared memory parallel computing. In *Proceedings of Supercomputing '90*, November 1990.

[2] M.-Y. Chern and T. Murata. Efficient matrix multiplications on a concurrent data-loading array. In *Proceedings of the International Conference on Parallel Processing*, pages 90–94, August 1983.

[3] G. Fox, A. Hey, and S. Otto. Matrix algorithms on the hypercube i: Matrix multiplication. In *Parallel Computing*, volume 4,17, 1987.

[4] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker. *Solving Problems on Concurrent Processors, vol I.* Prentice Hall, Englewood Cliffs, NJ, 1988.

[5] W. Griswold, G. Harrison, D. Notkin, and L. Snyder. Scalable abstractions for parallel programming. In *Proceedings of the Fifth Distributed Memory Computing Conference*, 1990. Charleston, South Carolina.

[6] Intel Corporation. *iPSC/2 User's Guide.* October 1989.

[7] S. L. Johnsson and C.-T. Ho. Algorithms for multiplying matrices of arbitrary shapes using shared memory primitives on boolean cubes. Technical Report YALEU/DCS/TR-569, Yale University, Department of Computer Science, October 1987.

[8] S. L. Johnsson and C.-T. Ho. Multiplication of arbitrarily shaped matrices on boolean cubes using the full communication bandwidth. Technical Report YALEU/DCS/TR-721, Yale University, Department of Computer Science, July 1989.

[9] S. Kak. A two-layered mesh array for matrix multiplication. *Parallel Computing*, pages 383–385, March 1988.

[10] H. T. Kung and C. Leiserson. *Introduction to VLSI Systems.* Addison-Wesley, Reading, MA, 1980. Section 8.3, by C. Mead and L. Conway.

[11] C. Lin and L. Snyder. A comparison of programming models for shared memory multiprocessors. In *Proceedings of the International Conference on Parallel Processing*, pages II 163–180, 1990.

[12] NCUBE Corporation. *NCUBE Product Report.* 1986. Beaverton OR.

[13] P. Nelson. *Parallel Programming Paradigms.* PhD thesis, University of Washington, Department of Computer Science, 1987.

[14] D. P. O'Leary and G. W. Stewart. Data-flow algorithms for parallel matrix computations. *Communications of the ACM*, 28(8):840–853, August 1985.

[15] I. Ramakrishnan and P. J. Varman. Modular matrix multiplication on a linear array. *IEEE Transactions on Computers*, C-33(11):952–958, November 1984.

[16] L. Snyder. Applications of the "Phase Abstractions" for portable and scalable parallel programming. In *Proceedings of the ICASE Workshop on Programming Distributed Memory Machines.*

iPSC/2 S				
Fox-Hey-Otto			Transpose	
P	time	subblock size	P	time
1	42306	–	1	42306
4	11457	64	4	10319
8	7439	32	8	5288
12	14755	1	12	3825
16	3155	32	16	2757
20	11297	1	20	2670
24	9600	1	24	2393
28	8317	1	28	1976
32	2260	16	32	1489

Table 1: Execution Times (time in msecs)

iPSC/2 F			
Fox-Hey-Otto		Transpose	
P	time	P	time
1	30084	1	30084
4	8326	4	7436
8	4678	8	4070
12	8665	12	2762
16	2351	16	2165
20	6532	20	1905
24	5567	24	1662
28	4833	28	1469
32	1378	32	1241

Table 2: Execution Times (time in msecs)

nCUBE/7			
Fox-Hey-Otto		Transpose	
P	time	P	time
1	117228	1	117228
4	33185	4	30017
8	17131	8	15206
12	52117	12	10935
16	8854	16	7920
20	34617	20	7621
24	30012	24	6836
28	26660	28	5636
32	4950	32	4302

Table 3: Execution Times (time in msecs)