Parallelization of the Dynamic Programming Algorithm for the Matrix Chain Product on a Hypercube

Steve A. Strate
Roger L. Wainwright

Department of Mathematical and Computer Science
The University of Tulsa

Abstract

In this paper we look at the dynamic programming technique implemented on a distributed-memory, multiprocessor system. We investigate the matrix chain product algorithm as an example problem. A method for parallelizing the dynamic programming technique for solving the matrix chain product problem is presented. Load balancing considerations are presented. The sequential dynamic programming technique is a fine grain algorithm, and considered by many researchers to be too fine grain to execute effectively on a hypercube. Our parallel algorithm yeilded modest speedups for fixed sized problems. Scaled problems promise even better speedups. Results show that respectable performance for the dynamic programming algorithmic technique can be achieved on a hypercube.

Keywords: Dynamic Programming, Distributed Memory Multiprocessor System,
Parallel Programming, Matrix Chain Product

Dynamic Programming

Dynamic programming is a decomposition technique for solving certain optimization problems, and is a powerful tool for solving combinatorial problems. To solve a large problem using the divide-and-conquer technique, one breaks the problem into smaller subproblems, each of which can be solved independently. These subproblems are in turn recursively divided into smaller subproblems and solved independently, and so on. There are many optimization problems for which the divide-and-conquer technique will not work. In many instances it is not clear which subproblems should be solved. That is, the best decomposition of a problem, in order to achieve the optimal solution, may not always be known. Using the dynamic programming technique, however, one simply solves all possible subproblems and stores the results to be used later in solving larger subproblems. The dynamic programming technique can be thought of as the divide-and-conquer principle taken to an extreme. Many classical problems have been solved using the dynamic programming technique. Such problems include all shortest paths in a weighted graph, matrix chain product, optimal binary search tree, traveling salesman problem, context free language recognition, 0/1 knapsack problem, Washall's algorithm for transitive closure, minimum triangulation of a polygon, comparison of sequences, flow shop scheduling, maze routing, and many other applications.

Matrix Chain Product

The Matrix Chain Product problem is described as follows: given a series of matrices of different sizes, the problem is to determine the order in which the

Proceedings of the 1990 ACM/Ieee Symposium on Applied Computing
April, 5-6, 1990, pp. 78-84.

matrices should be multiplied in order to minimize the total amount of computation. The following example is taken from Sedgewick [5]. Given the following matrices and their sizes, determine in what order they should be multiplied to minimize the computational cost.

MATRIX	A	B	C	D	E	F
SIZE	4x2	2x3	3x1	1x2	2x2	2x3

The cost of multiplying two matrices of size p x q and q x r is defined to be p*q*r. If the example matrices above are multiplied from left to right, that is ((((AB)C)D)E)F, the total cost is 84. If the matrices are multiplied from right to left, that is A(B(C(D(EF)))), the total cost is 64. The dynamic programming algorithm gives the optimal solution by establishing the following table of solved subproblems:

	B	C	D	E	F
A	24 AB	14 AB	22 CD	26 CD	36 CD
B		6 BC	10 CD	14 CD	22 CD
C			6 CD	10 CD	19 CD
D				4 DE	10 EF
E					12 EF

The sequential algorithm begins by solving all subproblems of length two. That is, the cost of multiplying matrices AB, BC, CD, DE, and EF are determined. The cost is 24, 6, 6, 4, and 12, respectively. These values are entered in the above table along the main diagonal. The next diagonal, entries AC, BD, CE, and DF are calculated based on the previous results. For example the AC entry, which represents the best way to multiply ABC, is either A(BC) or (AB)C. Results for (BC) and (AB) have already been calculated and are located in the previous diagonal. It is easy to determine that the best ordering is A(BC) with a cost of 14. The AB under 14 simply means that the optimal order for multiplication will have a split between A and B, that is, ..A)(B.. The process continues by calculating the remaining diagonals until finally the AF entry in the table is determined. This is the optimal solution. The optimal solution in this case is to split the matrices between CD, that is, (ABC)(DEF) with a cost of 36. The best way to multiply (ABC) and (DEF) has already been determined. Hence the final order is (A(BC))((DE)F). Of course substantial savings can be achieved when large matrices are involved.

The essence of dynamic programming algorithms is that they trade space for time by storing solutions to subproblems rather than recomputing them. The Matrix Chain Algorithm takes $O(N^3)$ time using $O(N^2)$ space, where N is the number of matrices. Notice the matrices are not actually multiplied together, only the cost of the multiplications is determined. Dynamic programming is based on the principle of optimality. Simply stated, this means the completion of an optimal sequence of decisions must be optimal. Generally, the computation proceeds from the small subproblems to the larger subproblems, as in the above example, storing the intermediate results in a table. This technique is called the tabular technique and often results in the most efficient implementation of the dynamic programming algorithm. Furthermore, for dynamic programming to apply, the subproblems must be complete self-contained problems with no hidden dependencies. In the above example this is illustrated by the fact that the five entries on the main diagonal may be determined independently of each other, and thus could be calculated in parallel. After that, the four entries on the next diagonal may be performed in parallel, and so on.

Hypercube Implementation

A Hypercube Multiprocessor of dimension d consists of 2^d identical nodes (or processors). The dimensions range from $0 <= d <= 10$ in most systems. Each processor is a general-purpose computer with its own local memory, and resident copy of the operating system. The facilities used in this research is an

iPSC/1 d5 (32 node) hypercube. Each node has 512k bytes of local memory. The nodes are numbered from 0 to 2^{d-1} using an d-bit binary number. If two nodes have the same binary representation except for one bit, then they are called adjacent nodes (neighbors). There are direct communication links between all pairs of adjacent nodes. The nodes can be thought of as being arranged in a cube of dimension d. In this case each node is directly connected to d adjacent nodes. A communication path between any two nodes in an d-dimensional cube is at most d. It is assumed the reader is familiar with the fundamental concepts of a hypercube multiprocessor system.

The sequential algorithm solves all subproblems on the main diagonal of the table, followed by each of the upper diagonals until a solution is determined in the upper right corner of the table. Further, for the matrix chain product problem the i,j entry in the table is calculated as a function of the table entries in row i left of position i,j and the column j entries below position i,j in the table. Functionally this can be expressed as Table[i,j] = f(Table[i,i+1..j-1], Table[i+1..j-1,j]).

The parallel dynamic programming algorithm for the matrix chain product problem views the hypercube as a ring (nearest neighbor) topology. For simplicity the processors are numbered consecutively 1, 2, ..., n in the ring. The actual node numbers in a ring are numbered in gray code order. The table of entries containing results of all subproblems are partitioned among the processors by rows. Processor 1 will calculate the first set of rows in the table, processor 2 will calculate the next set of rows, and so forth where processor n calculates the last set of rows. In this arrangement processor 1 will eventually determine the solution. Clearly, if the rows of the table are evenly partitoned among the processors, processor 1 will have the most entries to calculate, processor 2 the next most, and so forth, where processor n will have the least. Thus the partition of the rows of the table among the processors is skewed to allow for a balanced work load among the processors. This is discussed in more detail later.

Each processor simultaneously calculates the entries in the portion of the table it is assigned. The entries in the table are processed by columns left to right, bottom to top. This is unlike the traditional sequential algorithm. Each time processor i, (i=2..n), completes an entire column, the column of entries is sent to processor i-1. Furthermore, each time processor i, (i=1..n-1), begins to work on a new column, it receives entries for the same column previously calculated from processor i+1. In this manner, the additional storage required by any one processor (in addition to its portion of the table entries) is at most the size of one entire column of the table (N cells). Once a column of entries is determined in a given processor, memory used to hold the same column entries from its neighboring processor is released and reused for the next column. Figure 1 illustrated these principles. In this example N=26 matrices, and n=4 processors. The numbers in the table entries represent the order in which they are calculated. Each processor has the same order. The x entries indicate calculated table entries, (each processor is shown to have processed the same number of entries). Notice in this skewed arrangement, when any processor requires table entries from its neighbor in order to continue calculations in the same column, the information is sent and is waiting in the input message queue.

The goal is to keep processor 1 busy, while at the same time minimizing the idle time of the other processors. Simply distributing the total number of table entries evenly among the processors is insufficient. Several factors must be taken into consideration. Notice calculating each table entry by processor i requires more CPU time than calculating a table entry by processor i+1, for i=1..n-1. This is because all previously calculated column entries from higher numbered processors must be considered. This also means the size of the messages (previously calculated column entries in the table) passed from processor i to processor i-1, for i=2..n in the ring, increases as i decreases. In considering all these factors the rows of the Table are partitioned among the processors in a skewed fashion to allow for an even work load among the processors. In our system we allow the researcher to input his own partitioning arrangement. In this way, for a given problem, the proper load balance can be determined empirically. Working in this manner with various size problems allowed us to determine an *a priori* row partitioning formula to be applied de-

pending on the number of matrices, and the number of processors.

Results

Table I gives CPU timings in seconds, and speedups for various problem sizes and number of processors used. The maximum problem size we could run was 190 matrices, which is a modest size problem. At this time we have run only fixed sized problems. That is, the problem size is held fixed, while the number of processors is allowed to increase. Note the fine grain nature of this problem; the 50 matrix chain product problem was solved on one processor in under 11 seconds. The best speedup generated was 12 using 32 nodes and 190 matrices. Clearly, there is a trend of better performance and the problem size increases. The ratio of computation to communication clearly increases as the problem size increases, which is desirable. Scaled problems are problems where the problem size increases as the number of processors increase, maximizing the computation to communication ratio. In general scaled problems yield better performance than fixed size problems. The results reported here are consistant in speedup and efficiency with other research on an hypercube for other problems solved using the dynamic programming technique [2,4]. Our results show that the dynamic programming technique can work very sucessfully on a hypercube.

Conclusions and Further Research

The sequential dynamic programming technique is a fine grain algorithm; that is, the ratio of computation to communication in a hypercube system is very small. Hence, the dynamic programming technique is considered by many researchers to be too fine grain to execute effectively on a hypercube. Our results show that the hypercube can provide good speedups in solving the matrix chain product problem.

Research in the area of dynamic programming on a hypercube is relatively scarce. However, some recent articles implementing specific algorithms via the dynamic programming technique on a hypercube have appeared in the literature. Jeng and Sahni [2] implemented a parallel dynamic programming algorithm for the all pairs shortest paths algorithm on a hypercube. Lee and others [4] implemented the 0/1 knapsack problem using dynamic programming on a hypercube. Both implementations differed from the implementation presented here, primarily due to the nature of the problems to be solved. Their performance results in terms of speedups and effeciency, however, are similar. Other references that discuss parallelization of dynamic programming implemented on an SIMD machine, and mesh connected systolic array include [1,3]. Yao [6] discusses the parallelization of dynamic programming for a theoritical perspective.

One of the features of the dynamic programming technique is the succinct manner in which each table entry is determined. Generally, most dynamic programming solutions that use the tabular technique calculate each table entry as a simple function of previously determined values in the table. Hence, the dynamic programming algorithm when implemented using the tabular technique can easily be written in functional form. A close look at the sequential code for the 0/1 knapsack problem, optimal binary search tree problem, and the matrix chain product problem reveals they differ very little [5]. Each algorithm basically has a triple nested loop and the only significant difference between each of the algorithms is in the inner most loop, where the functional relationship between various entries in the table are specified.

The second phase of our research in this area involves developing a dynamic programming shell for a hypercube. This shell will be similar to the work reported here, and will allow for scaled problems. The ring topology and message passing code will already be in place in skeleton form. All the user need do is provide the functional specification for calculating each table entry, which depends on what problem is to be solved. From this specification the shell will automatically partition the work load among the processors and perform the calculations according the functional specification. In this way a single shell can be used in an automatic manner to solve many different problems using the tabular dynamic programming technique. The goal is to releave the researcher from writing code from scratch for each new algorithm, thus provide an environment for the ease of solving many different optimization problems.

Researchers have recently become interested in solving large optimization problems using large numbers of processors. The dynamic programming technique offers optimal solutions to many important optimization problems. On a sequential machine for problem size n, timings from $O(n^3)$ time to exponential time are commonly required. Since this is too expensive, less time consuming algorithms have been developed such as greedy algorithms. These less expensive algorithms rapidly find a "good" solution, but not an optimal solution. With present day parallel hardware there is no need to settle for less than an optimal solution.

References

1. E. Edminston and R.A. Wagner, "Parallelization of the Dynamic Programming Algorithm for Comparison of Sequences", Proceedings of the 1987 International Conference on Parallel Processing, August, 1987, pp. 78-80.

2. J.F. Jeng and S. Sahni, "All Pairs Shortest Paths on a Hypercube Multiprocessor", Proceedings of the 1987 International Conference on Parallel Processing, August, 1987, pp. 713-716.

3. G. Lakhani and R. Dorairay, "A VLSI Implementation of all-pair shortest path problem", Proceedings of the 1987 International Conference on Parallel Processing, August, 1987, pp. 207-209.

4. J. Lee, E. Shragowitz and S. Sahni, "A Hypercube Algorithm for the 0/1 Knapsack Problem", Proceedings of the 1987 International Conference on Parallel Processing, August, 1987, pp. 699-706.

5. R. Sedgewick, "Algorithms", Addision-Wesley, 1988.

6. F. Yao, "Speed-up in Dynamic Programming", SIAM Journal on Algebraic and Descartes Mathematics, vol. 3, no. 4, December, 1982, pp. 532-540.

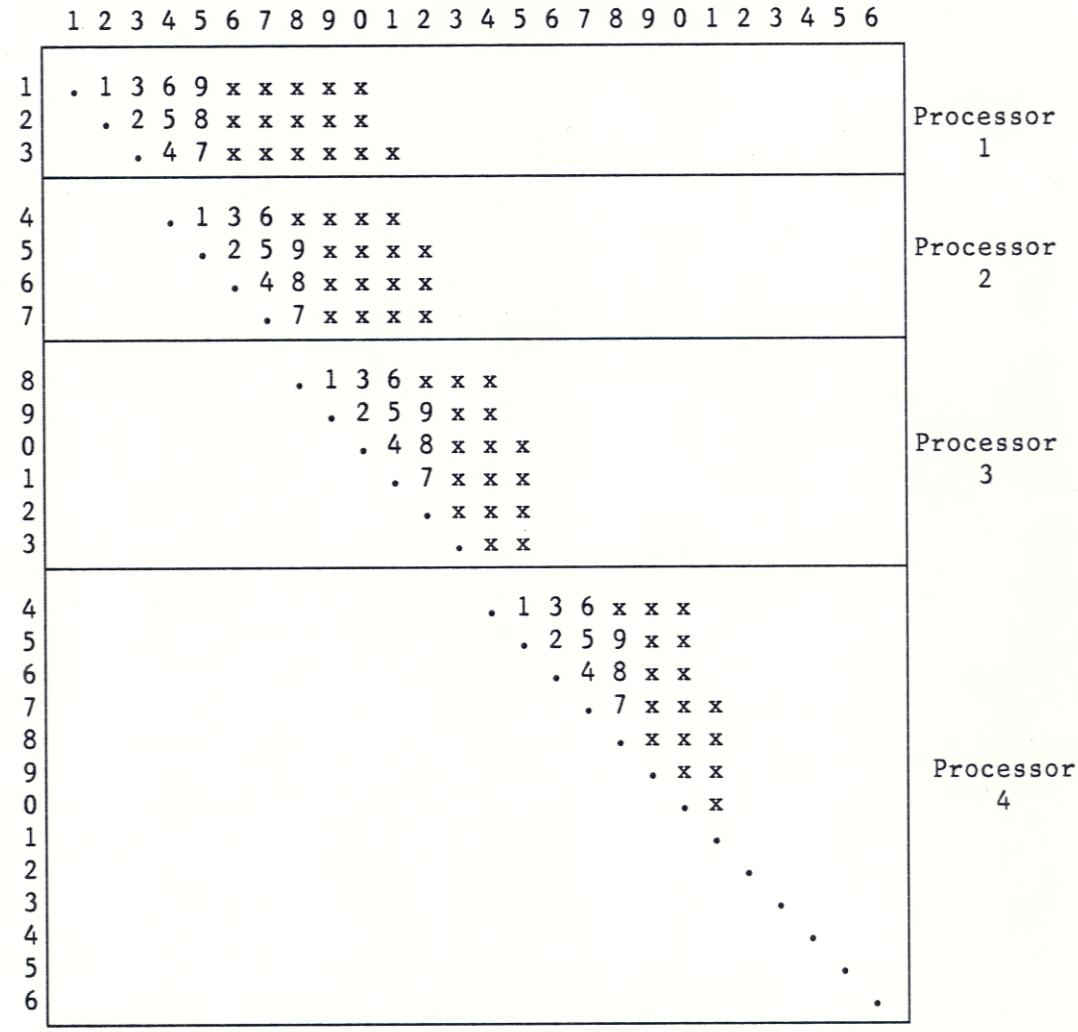

Fig. 1 Example Table Partition for N = 26, n = 4

Table I

Test Results — Matrix Chain Product

Number of Nodes	CPU Time (Seconds)			Speedup		
	Number of Matrices			Number of Matrices		
	50	120	190	50	120	190
1	10.55	141.78	557.35	1.00	1.00	1.00
2	5.85	72.50	285.32	1.80	1.96	1.95
4	3.81	51.81	195.55	2.77	2.74	2.85
8	2.57	29.42	118.34	4.11	4.82	4.71
16	2.33	16.36	61.94	4.53	8.67	9.00
32	1.82	12.17	44.42	5.80	11.65	12.55