LOAD BALANCING TECHNIQUES FOR DYNAMIC PROGRAMMING ALGORITHMS ON HYPERCUBE MULTIPTOCESSORS*

Steve A. Strate
Roger L. Wainwright

Department of Mathematical and Computer Sciences
The University of Tulsa

Abstract

In this paper we investigate several load balancing strategies for implementing the dynamic programming technique on a distributed-memory, multiprocessor system. A general method for parallelizing the dynamic programming technique for solving the class of tabular problems is presented. The goal of the package is to relieve a researcher from writing hypercube code from scratch for each new algorithm. Currently, simple changes to key routines in the skeleton code is all that is needed. We tested our dynamic programming package on the Matrix Chain Product, Optimal Binary Search, Polygon Triangulation, and the Sequence Comparison problems. The parallel dynamic programming algorithm yielded good speedups for fixed sized problems, and excellent speedups for scaled problems. 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. Results show that excellent performance for the dynamic programming algorithmic technique can be achieved on a hypercube with the proper load balancing considerations.

Introduction

The dynamic programming technique can be thought of as the divide-and-conquer principle taken to an extreme. The fundamental principle of dynamic programming is to build a large table with previously calculated results from smaller problems. That is, one solves all possible subproblems and stores the results in a table to be used later in solving larger subproblems. This technique is called the tabular technique and often results in the most efficient implementation of the dynamic programming algorithm. Generally, the computation proceeds from the small subproblems to the larger subproblems. The essence of dynamic programming algorithms is that they trade space for time by storing solutions to subproblems rather than recomputing them.

Dynamic programming is based on the principle of optimality. Simply stated, this means the completion of an optimal sequence of decisions must be optimal. Hence dynamic programming is a powerful technique for solving certain optimization problems. Many problems can be solved using the dynamic programming technique. Such problems include context free language recognition, flow shop scheduling, 0/1 knapsack problem, traveling salesman problem, matrix chain product, optimal binary search tree, minimum triangulation of a polygon, comparison of sequences, maze routing, Washall's algorithm for transitive closure, all shortest paths in a weighted graph, and many others.

There has been very little research using the dynamic programming technique on a hypercube. Yao [13] discusses the parallelization of dynamic programming for a theoretical perspective. Lee *et al.* [8] implemented the 0/1 knapsack problem using dynamic programming on a hypercube. Lin *et al.* [9] discuss parallelization of the 0/1 knapsack problem on several architectures. Jeng and Sahni [6] implemented a parallel dynamic programming algorithm for the all pairs shortest paths algorithm on a hypercube. Their implementations were for a specific problem. Our implementation is a general problem solving tool. Their performance results in terms of speedups and efficiency, however, are similar to ours. Other references that discuss parallelization of dynamic programming implemented on transputers, an SIMD machine, and a mesh connected systolic array include [1,4,7].

This paper develops and compares several load balancing strategies for the dynamic programming tabular technique on a hypercube multiprocessor. We implemented four different dynamic programming problems to test the load balancing strategies. The problems we implemented are classic dynamic programming problems: the matrix chain product, optimal binary search tree, polygon triangulation of a convex hull, and the sequence comparison algorithm. These are "root" problems for many different applications.

* Research partially supported by OCAST Grant ARO-038 and Sun Microsystems Inc.

ACM-SAC '93/2/93/IN, USA
© 1993 ACM 0-89791-568-2/93/0002/0562...$1.50

Proceedings of the 1993 ACM/SIGAPP Symposium on Applied Computing (SAC'93), Indianapolis, IN., February 14-16, 1993, pp. 562-569.

Test Problems

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 matrices should be multiplied in order to minimize the total amount of computation. The following example is taken from Sedgewick [11], and Strate and Wainwright [12]. 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 $n \times m$ and $m \times p$ is defined to be $n*m*p$. If the example matrices above are multiplied from right to left, that is A(B(C(D(EF)))), the total cost is 64. If the matrices are multiplied from left to right, that is ((((AB)C)D)E)F, the total cost is 84.

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 initially 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 BD entry, which represents the best way to multiply BCD, is either B(CD) or (BC)D. Results for (CD) and (BC) have already been calculated and are located in the previous diagonal. It is easy to determine that the best ordering is (BC)D with a cost of 10. The CD under 10 means that the optimal order for multiplication will have a split between C and D, that is, ..C)(D..

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 splits 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). Substantial savings can be achieved when large matrices are involved. The matrix chain product 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.

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]) [12]. The additional test problems implemented in this paper are given below in less detail.

Optimal Binary Search Tree

An optimal binary search tree is a binary search tree where the weighted internal path length is minimized. The weighted internal path length is a measure of the search cost of the tree. It is defined as the sum over all of the nodes of the frequency of access of each node times its internal path length. The dynamic programming approach to this problem computes the best way to build each appropriate subtree of size two, then all appropriate subtrees of size three and so forth. Larger problems are determined by examining smaller problems. This problem is similar to the matrix chain product problem discussed above and implemented in a similar manner. Sedgewick [11] gives an excellent presentation of this problem. The sequential optimal binary search tree algorithm with n nodes takes $O(n^3)$ time using $O(n^2)$ space.

Polygon Triangulation

A polygon is a piecewise-linear, closed curve in the plane. A triangulation of a polygon is a set T of chords of the polygon that divide the polygon into disjoint triangles (three-sided polygon). In this problem, we are given a convex polygon, P, and a weight function, w, defined on triangles formed by sides and chords of P. The problem is to find a triangulation that minimizes the sum of the weights of the triangles in the triangulation. A typical weight function on triangles is the Euclidian distance, although any weight function will work.

There is a similarity between triangulation of a polygon and the matrix chain product algorithm. A triangulation of an n-sided polygon corresponds to a parse tree with n-1 leaves. Also, a fully parenthesized product of n matrices corresponds to a parse tree with n leaves. It also corresponds to a triangulation of an (n+1)-

vertex polygon. In fact, the matrix chain product problem is a special case of the optimal triangulation problem. Every instance of the matrix chain product can be cast as an optimal triangulation problem. Hence the polygon triangulation problem can be set up as a dynamic programming problem where larger problems are solved by knowing the solutions of all smaller subproblems. Cormen [2] gives an excellent description of this problem for the interested reader. The sequential optimal polygon triangulation algorithm with n vertices takes $O(n^3)$ time using $O(n^2)$ space. We chose this algorithm because it has a much higher computation to communication ratio compared to the matrix chain product and optimal binary search tree algorithms.

Sequence Comparisons

This topic has received considerable attention lately due to its applications to molecular biology. Given two strings, find the minimum number of edit steps (insert, delete, replace) to change one string into the other. The main technique used to solve this problem is dynamic programming. The process begins by solving the sequence comparisons for smaller substrings, recording this information in a table, and using this to decide the optimal way to solve longer strings. Eventually the optimal number of edit sequences is determined for the original strings. An excellent presentation of this problem is given by Manber [10]. The number of cells in the table needed to determine the next entry is considerably less than the previous three algorithms. We chose this problem because it has an extremely low computation to communication ratio and should prove to be an excellent challenge for our load balancing strategies.

Load Balancing Strategies

A Hypercube Multiprocessor of dimension d consists of 2^d identical nodes (or processors). 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/2 d5 (32 node) hypercube. Each node has between 4M and 8M 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 (hamming distance of 1), then they are called adjacent nodes (neighbors). 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. There are direct communication links between all pairs of adjacent nodes. Hence, 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.

A close look at the sequential code for the 0/1 knapsack problem, optimal binary search tree problem, and the matrix chain product problem as depicted by Sedgewick [11] reveals they differ very little. 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. Based on this observation, we have implemented a general parallel dynamic programming algorithm that assumes all entries in the table are a function of others entries in the table. All of the test problems fall into this category as well as many others. Our general parallel dynamic programming algorithm is driven by the particular functional dependencies on other entries in the table in order to solve a new problem. It is very easy to adapt this algorithm for solving a wide variety of dynamic programming problems as long as they can be solved by the tabular technique.

The parallel dynamic programming algorithm views the hypercube as a ring (nearest neighbor) topology. For simplicity the processors are numbered consecutively 1, 2, ..., p 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 calculates the first set of rows in the table, processor 2 calculates the next set of rows, and so forth where processor p calculates the last set of rows. In this arrangement processor 1 will eventually determine the solution. All software was written in C, and messages were recieved using a "blocking" mechanism.

Figure 1 shows an example partitioning of rows onto processors. In this example n=16 rows, and p=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. 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 hopefully is waiting in the input message queue. 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..p), completes an entire column, the column of entries is sent to processor i-1. Furthermore, each time processor i, (i=1..p-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 [12].

The load balancing problem is basically how to map rows onto processors. The goal is to keep processor 1 busy, while at the same time minimizing the idle time of the other processors. Several factors must be taken into consideration. Calculating each table entry by processor i requires more CPU time than calculating a table entry by processor i+1, for i=1..p-1. This is because all previously calculated column entries from higher numbered processors must be considered. Thus, the size of the messages (previously calculated column entries in the table) passed from processor i to processor i-1, for i=2..p in the ring,

increases as i decreases.

We have developed several load balancing strategies. The first strategy, *Naive*, divides the rows of the table as evenly as possible among the processors. Clearly, this is a bad idea since processor 1 will have the most entries to calculate, processor 2 the next most, and so forth, where processor p will have the least. However this will serve as a point of reference for the other strategies. The second strategy, *Cell*, distributes the total number of table entries evenly among the processors. Figure 1 is partitioned in this manner as best as possible for such a small problem. The number of cells in the Figure 1 totals 120 and is distributed 29, 36, 27 and 28 among processors 1 to 4, respectively. In the third strategy, *Cost*, each table entry is given a weighted cost equal to the number of cells it is dependent on. The load is partitioned by balancing among the processors the weighted cost of the cells. Finally, a strategy called *Theory* analyzes a specific problem by determining the number of internal calculations required and the amount of message traffic required. This is coupled with the known computation rate of the processor and message passing cost, including latency cost and transfer rate. The *Naive* and *Cell* strategies do not take into consideration the characteristics of the problem. The *Cost* and *Theory* strategies are problem dependent, and we expect them to perform better.

For a given problem the user provides the functional dependencies, size of the problem, and the parameters specifying computation required to process each cell. This is sufficient for the parallel dynamic programming algorithm to automatically determine any of our load balancing strategies. Furthermore, in our system we allow the researcher to input his own partitioning arrangement. In this way, for a given problem, the proper load balance could be determined empirically.

Results

Table I through Table IV give the CPU times implementing various load balancing strategies for the matrix chain product, optimal binary search tree, polygon triangulation, and the sequence comparison problems, respectively. The CPU time includes communication delay. The problem sizes in each of the Tables represent the maximum size problem executable on each of the dimensions of the hypercube. The limiting factor on the problem size was based on what the *Cost* strategy could handle. As a simple illustration, consider two processors and a problem size of $n = 100$. The *Naive* strategy splits the work (50,50), but the *Cost* strategy might split the work (25,75). Hence larger problems can be executed using the *Naive* strategy versus the *Cost* strategy. Of course, the *Cost* strategy gives better performance.

Table I through Table IV indicates both scaled and fixed size problems. A fixed size problem is when a problem size is held fixed, but additional processors are added. This represents data entries in each of the columns. A scaled size problem is when the problem size increases as the number of processors increase. This is represented along the main diagonal of each Table. Hence the upper triangular part of the Tables represent impossible sized problems due to memory constraints. The *Cost* strategy is much easier to calculate than the complicated theoretical analysis needed to determine the *Theory* strategy. Fortunately, our empirical results showed the *Cost* strategy was virtually identical to the *Theory* strategy. Therefore in all of the Tables, the *Theory* strategy is not included, and the *Cost* entry represents both strategies.

Table V through Table VIII give the speedups for the matrix chain product, optimal binary search tree, polygon triangulation, and the sequence comparison problems, corresponding to the CPU times in Table I through Table IV, respectively. There have been several ways that have been suggested to determine scaled speedup [3,5]. In our example problems, given an $n \times n$ table, the total computation time using one processor is the sum of the weighted cost of each cell in the table.

Let t_1 be the time to execute a problem on one processor. The closed form for the sum of the weighted cost of each cell in the matrix product chain problem is $n(n-1)(n+1)/3$. Hence $t_1 = cn(n-1)(n+1)/3$, where c is the computation time required per cell in the table. The value of c is determined empirically using one processor. By increasing n as needed and keeping c constant, we are able to estimate the time for one processor for larger problems. The closed form for the sum of the weighted cost of each cell in the optimal binary chain problem is also $n(n-1)(n+1)/3$. Hence $t_1 = cn(n-1)(n+1)/3$. These problems use the same cells for determining new entries in the table, but different calculations. The closed form for the sum of the weighted cost of each cell in the polygon triangulation problem is $n(n-1)(n-2)/3$. Hence $t_1 = cn(n-1)(n-2)/3$ in this problem. The closed form for the sum of the weighted cost of each cell in the sequence comparison problem is $3nm$. Hence $t_1 = c3nm$, where n and m are the lengths of the two strings. In all of our sequence comparison problems we chose $m = n - 2$.

Consider the results for the matrix product chain problem (Table I and Table V). The CPU timings for the three strategies corresponding to the scaled problems for each dimension of the hypercube are plotted for comparison in Figure 2. Notice the *Cost* strategy is clearly the superior strategy. The *Cost* strategy for the scaled problems for each dimension of the hypercube resulted in speedups of 1.85, 3.44, 6.37, 12.40 and 27.16 representing efficiencies of 92.5, 86.0, 79.6, 77.5, and 84.9%, respectively.

Consider the results for the optimal binary search tree (Table II and Table VI). The CPU timings for the three strategies corresponding to the scaled problems for each dimension of the hypercube are plotted for comparison in Figure 3. Again, the *Cost* strategy is clearly the superior strategy. The *Cost* strategy for the scaled problems for each dimension of the hypercube resulted in speedups of 1.95, 3.62, 6.67, 13.02 and 28.90 representing efficiencies of 97.5, 90.5, 83.3, 81.3, and 90.3%, respectively.

Similarly, for the polygon triangulation problem (Table III and Table VII), the CPU timings for the three strategies corresponding to the scaled problems for each dimension of the hypercube are plotted for comparison in Figure 4. The *Cost* strategy is also the superior strategy. In this problem, the *Cost* strategy for the scaled problems for each dimension of the hypercube resulted in speedups of 1.72, 3.14, 5.65, 10.75 and 22.42 representing efficiencies of 86.0, 78.5, 83.3, 70.6, and 70.0%, respectively. These results are not as good as the previous two problems, but still represents excellent performance.

Finally, for the sequence comparison problem, (Table IV and Table VIII), the CPU timings for the three strategies corresponding to the scaled problems for each dimension of the hypercube are plotted for comparison in Figure 5. This problem has very little computation. It is almost totally communication. As expected, all three strategies gave identical results. In this problem, the results for the scaled problems for each dimension of the hypercube resulted in speedups of 1.36, 2.56, 4.98, 9.66 and 18.33 representing efficiencies of 68.0, 64.0, 62.3, 60.4, and 57.3%, respectively. This problem was expected to exhibit the worse possible results for our dynamic programming algorithm due to the low communication to computation ratio. However, results were consistent over all of the scaled problems at about 60% efficiency, representing excellent results for a worst case situation.

Conclusions and Future Research

The dynamic programming technique is a fine grain algorithm; that is, the ratio of computation to communication in a multiprocessor 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 a multiprocessor can provide excellent speedups in solving dynamic programming problems using the tabular technique. In fact, the results reported here are consistent in speedup and efficiency with other research on an hypercube for other problems solved using the dynamic programming technique [6,8]. Our results show that the dynamic programming technique can work very successfully on a multiprocessor system.

In every test case over all four example problems the *Cost* strategy was the superior strategy. This was expected, since it was the same as the *theoretical* strategy. In fact, our experience found the *Cost* strategy automatically split the rows of the table into the same partitions that we got doing the partitions by hand. This happened about 95% of the time. The remaining 5% of the time the *Cost* strategy was off by one row. Our algorithms can easily run on the iPSC/860 with similar results. All that is necessary is that the *Cost* strategy is adjusted by a factor corresponding to the ratio between the computation speeds of the two machines, followed by some minor fine tuning. Furthermore, with mesh architectures, the tabular technique appears to map naturally.

We are in the process of developing a dynamic programming shell for a hypercube. The topology and message passing code are already in place in skeleton form. A new user provides the functional specification for calculating each table entry, which is problem dependent. From this specification the shell will automatically partition the work load among the processors (or the user can specify the strategy), 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 relieve the researcher from writing hypercube code from scratch for each new algorithm, thus provide an environment for the ease of solving many different optimization problems. At the present time, a new user (solving a new problem) is required to make a small number of changes within several key functions within the code in order to change one problem into another. Although, this is significantly better than rewriting the entire code from scratch, it is not completely automated yet.

The authors' mailing address is Department of Mathematical and Computer Sciences, The University of Tulsa, 600 South College Avenue, Tulsa, OK 74104-3189, strate@euler.mcs.utulsa.edu, rogerw@penguin.mcs.utulsa.edu.

Acknowledgements

This research has been partially supported by OCAST Grant ARO-038. The authors also wish to acknowledge the support of Sun Microsystems Inc.

References

[1] B. Chardonnens, R. D. Hersch and O. Koelbl, "Multiprocessor Performances for Dynamic Programming", Microprocessing & Microprogramming, vol. 28, no. 1, March, 1990, pp. 91-94.

[2] Cormen, Thomas H., Charles E. Leierson and Ronald L. Rivest, Introduction to Algorithms, The MIT Press, Cambridge, Mass., 1990, pp 301-336.

[3] Denning, P.J., "The Science of Computing - Speeding up Parallel Processing", American Scientist, vol. 76, pp. 347-349, July-August, 1988.

[4] 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.

[5] Gustafson, "Reevaluating Amdahl's Law", Commun. ACM, vol. 31, no. 5, pp.532-533, May, 1988.

[6] 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.

[7] 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.

[8] 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.

[9] J. Lin and J. Storer, "Processor-efficient Hypercube Algorithms for the Knapsack Problem", Journal of Parallel and Distributed Computing, vol. 13, no. 3, Nov., 1991, pp. 332-337.

[10] U. Manber, "Introduction to Algorithms, A Creative Approach", Addison-Wesley, 1989.

[11] R. Sedgewick, "Algorithms", Addison-Wesley, 1988.

[12] S. Strate and R. Wainwright, "Parallelization of the Dynamic Programming Algorithm for the Matrix Chain Product on a Hypercube", Proceedings of the 1990 ACM/IEEE Symposium on Applied Computing (SAC'90), April, 1990, pp. 78-84.

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

Number Nodes	Strategy	Problem Size					
		530	617	709	821	956	1132
1		642.6	1014*	1533*	2380*	3757*	6237*
2	Naive	591.6	937.3	—	—	—	—
	Cell	413.7	652.7	—	—	—	—
	Cost	347.1	547.4	—	—	—	—
4	Naive	365.1	577.9	877.5	—	—	—
	Cell	212.5	337.2	511.0	—	—	—
	Cost	186.5	293.5	445.3	—	—	—
8	Naive	201.7	318.7	481.8	748.1	—	—
	Cell	108.6	169.2	258.5	400.4	—	—
	Cost	100.8	168.4	243.0	373.4	—	—
16	Naive	105.5	164.9	252.8	393.1	616.6	—
	Cell	53.8	85.1	129.4	205.2	321.1	—
	Cost	52.7	82.2	124.7	179.5	303.0	—
32	Naive	52.8	84.7	129.5	197.3	310.8	526.1
	Cell	27.8	44.3	66.1	103.6	162.7	270.1
	Cost	25.5	38.0	57.2	89.7	137.9	229.6

Table I: CPU Times (sec.) for The Matrix Chain Product Problem for Various Problem Sizes and Load Balancing Strategies.
*** Estimated CPU Times using c = .0000129**

Number Nodes	Strategy	Problem Size					
		540	609	684	768	859	958
1		1413.4	2029*	2878*	4076*	5706*	7917*
2	Naive	1371.5	1981.8	—	—	—	—
	Cell	986.3	1424.4	—	—	—	—
	Cost	809.9	1177.8	—	—	—	—
4	Naive	877.7	1267.0	1808.6	—	—	—
	Cell	525.0	762.1	1079.2	—	—	—
	Cost	444.2	646.9	916.7	—	—	—
8	Naive	492.7	707.4	1016.2	1438.5	—	—
	Cell	266.0	388.0	555.6	793.9	—	—
	Cost	245.2	353.9	532.9	722.0	—	—
16	Naive	257.1	369.2	531.6	732.5	1065.2	—
	Cell	136.8	194.9	273.2	387.4	557.9	—
	Cost	119.0	172.5	245.8	376.1	530.6	—
32	Naive	128.6	185.4	273.2	378.9	537.3	747.4
	Cell	67.6	98.6	141.3	200.9	275.8	392.1
	Cost	66.9	92.5	131.6	181.8	255.5	353.2

Table III: CPU Times (sec.) for The Optimal Polygon Triangulation for Various Problem Sizes and Load Balancing Strategies.
*** Estimated CPU Times using c = .0000271**

Number Nodes	Strategy	Problem Size					
		530	617	709	821	956	1132
1		640.8	1014*	1533*	2380*	3757*	6237*
2	Naive	572.0	895.1	—	—	—	—
	Cell	399.1	623.0	—	—	—	—
	Cost	334.7	520.7	—	—	—	—
4	Naive	352.0	551.7	830.9	—	—	—
	Cell	204.8	322.1	484.5	—	—	—
	Cost	181.6	281.3	423.2	—	—	—
8	Naive	194.3	304.4	456.4	705.5	—	—
	Cell	105.0	161.9	245.1	377.9	—	—
	Cost	97.6	154.7	232.5	356.6	—	—
16	Naive	102.0	157.9	239.8	370.5	578.7	—
	Cell	52.1	81.7	123.1	193.8	302.0	—
	Cost	51.7	79.2	119.3	183.8	288.6	—
32	Naive	51.1	81.2	123.1	186.5	292.1	493.3
	Cell	26.9	42.6	62.9	98.0	153.4	253.7
	Cost	24.8	36.6	54.1	85.1	130.5	215.8

Table II: CPU Times (sec.) for The Optimal Binary Search Tree for Various Problem Sizes and Load Balancing Strategies.
*** Estimated CPU Times using c = .0000129**

Number Nodes	Strategy	Problem Size					
		530	760	1075	1520	2150	3035
1		7.1	14.6*	29.4*	58.8*	117.8*	234.7*
2	Naive	5.3	10.8	—	—	—	—
	Cell	5.3	10.8	—	—	—	—
	Cost	5.2	10.7	—	—	—	—
4	Naive	2.9	5.8	11.5	—	—	—
	Cell	2.9	5.8	11.5	—	—	—
	Cost	2.9	5.8	11.5	—	—	—
8	Naive	1.6	3.2	6.1	11.8	—	—
	Cell	1.6	3.2	6.1	11.8	—	—
	Cost	1.6	3.2	6.1	11.8	—	—
16	Naive	1.0	1.8	3.4	6.4	12.2	—
	Cell	1.0	1.8	3.4	6.4	12.2	—
	Cost	1.0	1.8	3.4	6.4	12.2	—
32	Naive	0.6	1.1	2.0	3.6	6.8	12.8
	Cell	0.6	1.1	2.0	3.6	6.8	12.8
	Cost	0.6	1.1	2.0	3.6	6.8	12.8

Table IV: CPU Times (sec.) for The Sequence Comparisons Problem for Various Problem Sizes and Load Balancing Strategies.
*** Estimated CPU Times using c = .0000085**

	Rows	Columns 1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Node 1	1	.	1	3	5	7	9	11	13	15	x	x					
	2		.	2	4	6	8	10	12	14	x	x					
Node 2	3			.	1	3	6	9	12	15	18	x	x				
	4				.	2	5	8	11	14	17	x	x				
	5					.	4	7	10	13	16	x	x				
Node 3	6						.	1	3	6	9	12	15	x			
	7							.	2	5	8	11	14	x			
	8								.	4	7	10	13	x			
Node 4	9									.	1	3	6	10	15	21	x
	10										.	2	5	9	14	20	x
	11											.	4	8	13	19	x
	12												.	7	12	18	x
	13													.	11	17	x
	14														.	16	x
	15															.	x
	16																.

Figure 1: Example Load Balancing for n = 16, p = 4 using the Cell Strategy

Number Nodes	Strategy	Problem Size 530	617	709	821	956	1132
1		—	—	—	—	—	—
2	Naive	1.09	1.08	—	—	—	—
	Cell	1.55	1.55	—	—	—	—
	Cost	1.85	1.85	—	—	—	—
4	Naive	1.76	1.75	1.78	—	—	—
	Cell	3.02	3.01	3.00	—	—	—
	Cost	3.45	3.45	3.44	—	—	—
8	Naive	3.19	3.18	3.18	3.18	—	—
	Cell	5.92	5.99	5.93	5.94	—	—
	Cost	6.38	6.02	6.31	6.37	—	—
16	Naive	6.09	6.15	6.06	6.05	6.09	—
	Cell	11.94	11.91	11.85	11.60	11.70	—
	Cost	12.19	12.33	12.29	13.26	12.40	—
32	Naive	12.17	11.97	11.84	12.06	12.09	11.86
	Cell	23.16	22.89	23.19	22.97	23.09	23.09
	Cost	25.2	26.68	26.80	26.53	27.24	27.16

Table V: Speedup for The Matrix Chain Product Problem for Various Problem Sizes and Load Balancing Strategies.

Number Nodes	Strategy	Problem Size 540	609	684	768	859	958
1		—	—	—	—	—	—
2	Naive	1.03	1.02	—	—	—	—
	Cell	1.43	1.43	—	—	—	—
	Cost	1.75	1.72	—	—	—	—
4	Naive	1.61	1.60	1.59	—	—	—
	Cell	2.69	2.66	2.67	—	—	—
	Cost	3.18	3.14	3.14	—	—	—
8	Naive	2.87	2.87	2.83	2.83	—	—
	Cell	5.31	5.23	5.18	5.13	—	—
	Cost	5.76	5.73	5.40	5.65	—	—
16	Naive	5.50	5.50	5.41	5.56	5.36	—
	Cell	10.33	10.41	10.53	10.52	10.23	—
	Cost	11.88	11.76	11.71	10.83	10.75	—
32	Naive	10.99	10.94	10.53	10.76	10.62	10.59
	Cell	20.90	20.58	20.37	20.29	20.69	20.19
	Cost	21.13	21.93	21.87	22.42	22.33	22.42

Table VII: Speedup for The Optimal Polygon Triangulation for Various Problem Sizes and Load Balancing Strategies.

Number Nodes	Strategy	Problem Size 530	617	709	821	956	1132
1		—	—	—	—	—	—
2	Naive	1.12	1.13	—	—	—	—
	Cell	1.61	1.63	—	—	—	—
	Cost	1.91	1.95	—	—	—	—
4	Naive	1.82	1.84	1.84	—	—	—
	Cell	3.13	3.15	3.16	—	—	—
	Cost	3.53	3.60	3.62	—	—	—
8	Naive	3.30	3.33	3.36	3.37	—	—
	Cell	6.10	6.26	6.25	6.30	—	—
	Cost	6.57	6.55	6.59	6.67	—	—
16	Naive	6.28	6.42	6.39	6.42	6.49	—
	Cell	12.30	12.41	12.45	12.28	12.44	—
	Cost	12.39	12.80	12.85	12.95	13.02	—
32	Naive	12.54	12.49	12.45	12.76	12.86	12.64
	Cell	23.82	23.80	24.37	24.29	24.49	24.58
	Cost	25.84	27.70	28.34	27.97	28.79	28.90

Table VI: Speedup for The Optimal Binary Search Tree for Various Problem Sizes and Load Balancing Strategies.

Number Nodes	Strategy	Problem Size 530	760	1075	1520	2150	3035
1		—	—	—	—	—	—
2	Naive	1.34	1.35	—	—	—	—
	Cell	1.34	1.35	—	—	—	—
	Cost	1.36	1.36	—	—	—	—
4	Naive	2.45	2.52	2.56	—	—	—
	Cell	2.45	2.52	2.56	—	—	—
	Cost	2.45	2.52	2.56	—	—	—
8	Naive	4.44	4.56	4.82	4.98	—	—
	Cell	4.44	4.56	4.82	4.98	—	—
	Cost	4.44	4.56	4.82	4.98	—	—
16	Naive	7.10	8.11	8.65	9.19	9.66	—
	Cell	7.10	8.11	8.65	9.19	9.66	—
	Cost	7.10	8.11	8.65	9.19	9.66	—
32	Naive	11.83	13.27	14.70	16.33	17.32	18.33
	Cell	11.83	13.27	14.70	16.33	17.32	18.33
	Cost	11.83	13.27	14.70	16.33	17.32	18.33

Table VIII: Speedup for The Sequence Comparisons Problem for Various Problem Sizes and Load Balancing Strategies.

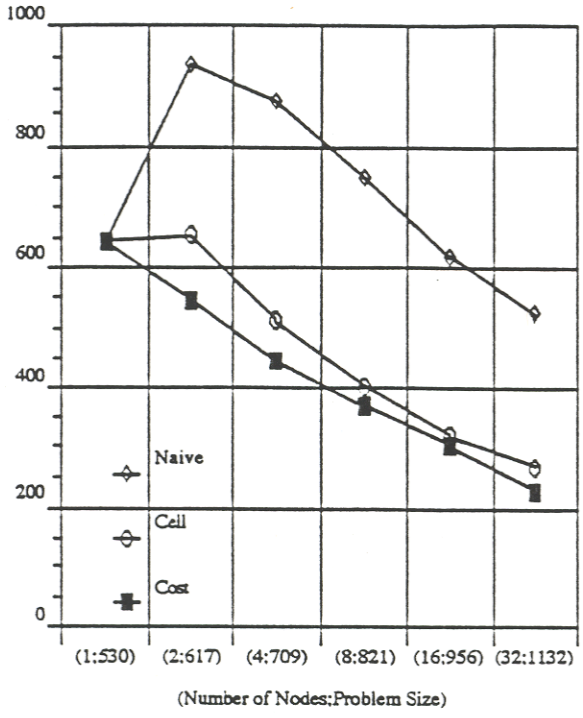

Figure 2: Matrix Chain Product CPU Timings (seconds)

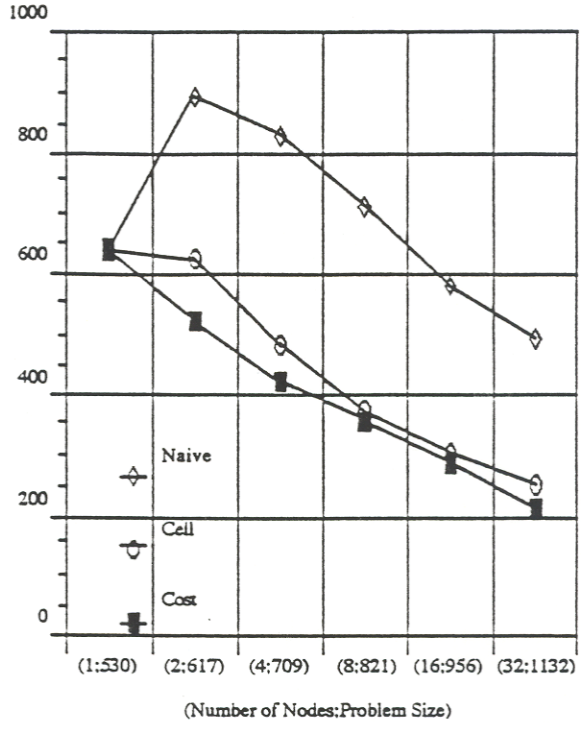

Figure 3: Optimal Binary Search Tree CPU Timings (seconds)

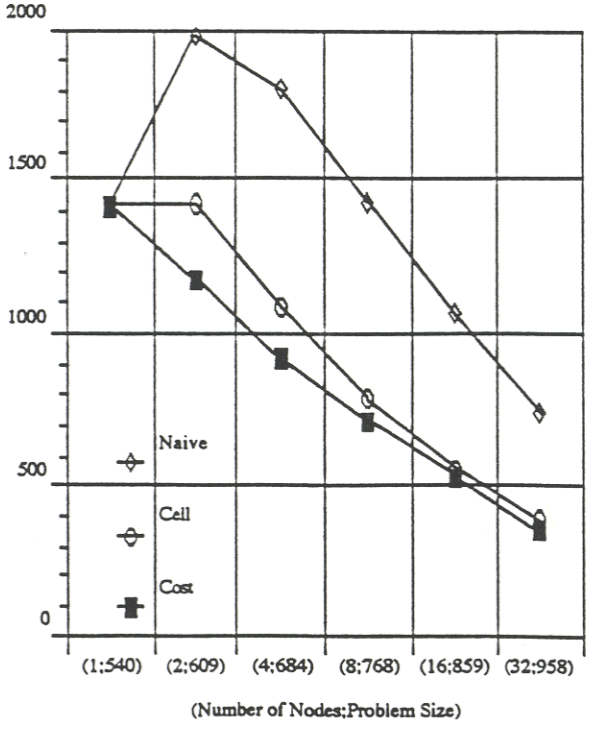

Figure 4: Optimal Polygon Triangulation CPU Timings (seconds)

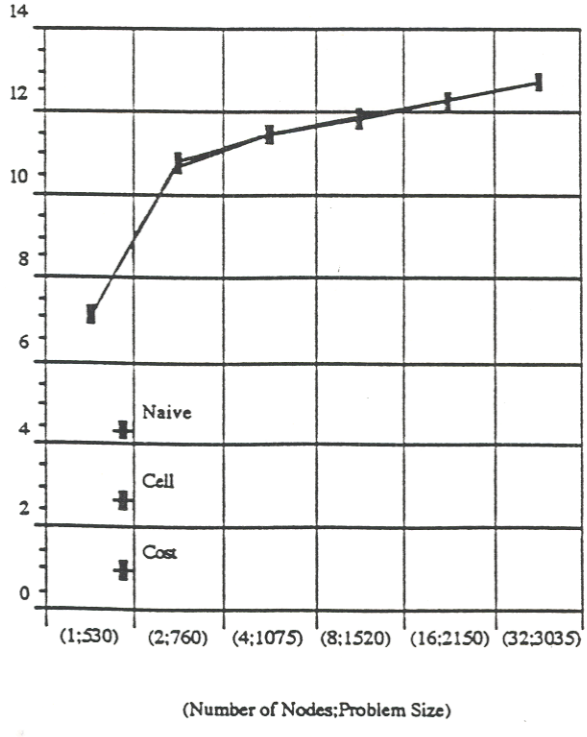

Figure 5: Sequence Comparison CPU Timings (seconds)