4

SOLUTION OF LARGE DENSE TRANSPORTATION PROBLEMS USING A PARALLEL PRIMAL ALGORITHM

by

Donald L. Miller*
Joseph F. Pekny**
Gerald L. Thompson***

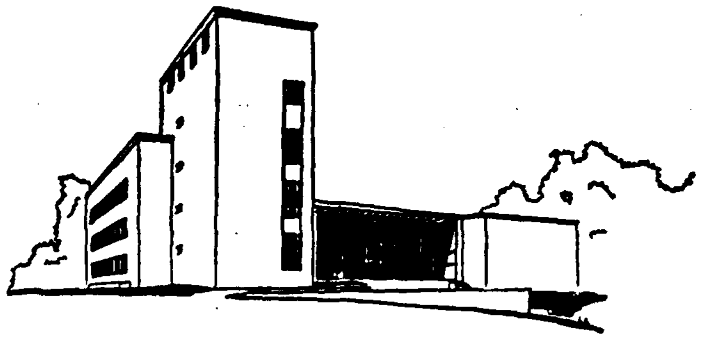
June 1988

Carnegie Mellon University

PITTSBURGH, PENNSYLVANIA 15213

GRADUATE SCHOOL OF INDUSTRIAL ADMINISTRATION

WILLIAM LARIMER MELLON, FOUNDER



4

W.P. No. 87-88-42

Management Science Research Report No. 546

SOLUTION OF LARGE DENSE TRANSPORTATION PROBLEMS USING A PARALLEL PRIMAL ALGORITHM

by

Donald L. Miller*
Joseph F. Pekny**
Gerald L. Thompson***

June 1988

* Central Research and Development Department,
E. I. duPont de Nemours and Company, Inc.

** Department of Chemical Engineering
Carnegie Mellon University

*** Graduate School of Industrial Administration
Carnegie Mellon University

This report was prepared in part as part of the activities of the Management Sciences Resarch Group, Carnegie Mellon University, under Contract No. N00014-85-K-0139 NR 047-048 with the Office of Naval Research. Reproduction in whole or in part is permitted for any purpose of the U.S. Government.

Management Sciences Research Group
Graduate School of Industrial Administration
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

SOLUTION OF LARGE DENSE TRANSPORTATION PROBLEMS USING A PARALLEL PRIMAL ALGORITHM

by

Donald L. Miller, Joseph F. Pekny, and Gerald L. Thompson

We implemented a version of the primal transportation code on a 14 processor BBN Butterfly computer and solved a variety of large, fully dense, randomly generated transportation and assignment problems ranging in sizes up to m = n = 3000. ~~We found that~~ the search phase of primal transportation algorthm was well suited for implementation on the Butterfly, but the pivot phase could not be parallelized. Computational experience is presented showing that speedup factors for a parallel over a sequential computer of approximately 70 percent of the theoretical maximum were obtained. With the parallel code the empirical difficulty of solving an nxn transportation problem was proportional to n^a where a varied between 2.0 and 2.2.

SOLUTION OF LARGE DENSE TRANSPORTATION PROBLEMS USING A PARALLEL PRIMAL ALGORITHM

by

Donald L. Miller, Joseph F. Pekny, and Gerald L. Thompson

1. INTRODUCTION

Among the most commonly solved and applied linear programs are transportation problems, and their close relations, assignment and network problems. They have applications of their own as well as being relaxations to others such as travelling salesman problems.

In the decade of the 1960's, the belief by most researchers was that dual (Kuhn [15]) and/or primal-dual (Ford-Fulkerson [9]) methods provided the most efficient algorithms for this class of problems. This belief was based on limited computation performed on very small problems by hand or by first generation computers.

In the early 1970's two groups of researchers, Srinivasan and Thompson [21,22] and Glover, Karney, and Klingman [11], wrote codes using the primal transportation algorithm, also called the stepping stone method or MODI method, see Charnes and Cooper [4] and Dantzig [6]. They used some newly invented data structures, as well as some provided in the computer science literature, to greatly improve the efficiency of the resulting primal codes, and concluded that primal transportation codes ran 100 to 200 times as fast as ordinary linear programming simplex codes and about 50 times as fast as primal dual methods on the same problems. These conclusions are summarized in Charnes, Karney, Klingman, Stutz and Glover [5]. Later Bradley, Brown and Graves [3] came to the same conclusion for network problems. Because of the computer memory limitations of that time, fully dense transportation problems were solved for sizes up to about $m = n = 200$. For larger dimensions, only sparse transportation problems having a relatively few arcs (approximately 10n) were considered. Further improvements in data structures used by the primal codes appeared in Barr, Glover and Klingman [1].

In the late 1970's and early 1980's there was a resurgance of interest in dual codes. Auction and bidding dual codes were proposed by Bertsekas [2], and Thompson [24]. Srinivasan and Thompson [23] proposed cost operator algorithms. A new class of dual codes, using shortest augmenting paths, based on the paper of Tomizawa [25] and improved by Dorhout [8], were developed. Hung and Rom [13], Derigs [7], and Martello and Toth [16] provided computational studies that compared their own shortest augmenting path codes with those of the other ideas. McGinnes [17] programmed both primal and dual codes and made computational studies and included that dual codes were superior. See also Hatch [12] and Glover and Klingman [10].

All of the computational experience discussed so far was obtained using sequential computers to solve small dense, or larger sparse problems. In the last five years a number of hardware companies have built computers which have a parallel architecture, that is, which consist of several small independent computers (processors) that can be programmed to work simultaneously on different parts of the same problem.

The purpose of the present paper is to discuss the implementation of a primal transportation code on a specific parallel machine, a 14 processor BBN Butterfly computer, and to give computational experience obtained from using it to solve a variety of large, fully dense, randomly generated transportation and assignment problems ranging in sizes up to $m = n = 3000$. Our conclusions are:

(a) The primal transportation algorithm is well suited for implementation on the Butterfly computer because of the way the memory is distributed among the processors which makes the search step efficient to parallelize.

(b) The pivot step of the primal code is not amenable to parallelization.

(c) The search for pivot candidates becomes the dominant activity of the primal algorithm as problem size increases.

(d) Speedup increases as the problem size increases.

(e) Parallel architecture permits the use of coding strategies that are difficult or not profitable to implement on sequential computers.

(f) The empirical difficulty of solving an $n \times n$ transportation or assignment problem using the parallel code is proportional to n^a where a varies between 2.0 and 2.2.

2. THE PRIMAL ALGORITHM

The transportation problem is to ship at minimum cost a homogeneous good from a set of m warehouses to a set of n markets. Let x_{ij} be the amount of the good shipped from warehouse i to market j, let c_{ij} be the unit cost of this shipment, let a_i be the availability (supply) of the good at warehouse i and let b_j be the amount of the good required (demanded) at market j. Assume $\Sigma_i a_i = \Sigma_j b_j$ so that there is a feasible shipping pattern. The mathematical statement of the transportation problem then is:

$$\text{Minimize} \left(\sum_{i=1}^{m} \sum_{j=1}^{n} c_{ij} x_{ij} \right) \qquad (1)$$

$$\text{Subject to} \sum_{j=1}^{n} x_{ij} = a_i \qquad \text{for } i=1,\dots,m \qquad (2)$$

$$\sum_{i=1}^{m} x_{ij} = b_j \qquad \text{for } j=1,\dots,n \qquad (3)$$

$$x_{ij} \geq 0 \qquad \text{for all } i \text{ and } j \qquad (4)$$

Letting u_i and v_j be the dual variables associated with (2) and (3) respectively, the dual problem can be stated as:

$$\text{Maximize} \left\{ \sum_{i=1}^{m} u_i a_i + \sum_{j=1}^{n} v_j b_j \right\} \qquad (5)$$

Subject to

$$u_i + v_j \leq c_{ij} \quad \text{for all } i \text{ and } j \qquad (6)$$

The following facts are well known concerning these two problems:

(1) The problem has the natural integer property: that is if a_i and b_j are positive integers for all i and j, then every feasible basic solution satisfying conditions (2), (3), and (4) gives x_{ij} equal to a nonnegative integer.

(2) In any basic solution, at most $m+n-1$ values of x_{ij} are positive and exactly $m+n-1$ variables x_{ij} are basic..

(3) Let $V = \{R_1,\ldots,R_m, C_1,\ldots,C_n\}$ be the set of rows and columns of the problem, let x_{ij} be a feasible solution and let $B = \{(i,j) | x_{ij} \text{ is basic}\}$; then the graph $G = \{V,B\}$ is a connected acyclic graph called the *basis tree*.

(4) There are well known procedures such as northwest corner, VAM, matrix minimum, modified row minimum, etc. (see [22]) for finding an initial basic primal feasible solution to (2), (3), and (4).

(5) Given a primal feasible basis tree, any node (row or column) can be denoted as the *root node* and the tree can be rehung so that the root node is at the top.

(6) The nodes of degree one in a basis tree are called *pendant nodes*. Starting from each pendant node and working upward to the root node it is possible to determine the shipping amounts x_{ij}. Similarly,

starting at the root node and working downwards it is possible to determine the optimal dual variables u_i and v_j associated with nodes R_i and C_j, respectively.

(7) If k is any (row or column) node, not the root node r, of a basis tree, then there is a unique path in the tree from k to r. The first encountered node on this path is the *predecessor* $p(k)$ of k, and the number of arcs on the path is the *distance* $d(k)$ of k from r. Note that r has no predecessor and we define $d(r) = 0$.

(8) If x_{ij} is basic then (i,j) is an arc in the basis tree so that either (a) $p(R_i) = C_j$ or (b) $p(C_j) = R_i$. In case (a) we let $x(R_i) = x_{ij}$ be the *shipping amount* and in case (b) we let $x(C_j) = x_{ij}$.

(9) If h and k are distinct nodes of the basis tree and k is on the path between h and the root node r then node h is a *descendent* of k. The basis tree is said to be stored in a *recursive list* if it has the following property: if node h is stored after node k on the list and h is not a descendent of k, then all descendents of k are stored on the list between k and h. The descendents of a node in a recursive list can be searched by calling a recursive subroutine.

(10) The dual variables are stored as node lists where $u(R_i) = u_i$ and $v(C_j) = v_j$. (If k is any node then exactly one of $u(k)$ or $v(k)$ is defined depending on whether k is a row or column node.)

Figure 6 gives the flow diagram of the primal transportation code. It was implemented by using the five node functions $p(k)$, $d(k)$, $x(k)$, $u(k)$, $v(k)$, and by storing the basis tree in a recursive list. The sequential implementation of the primal code is discussed in more detail in [11], [14], and [21].

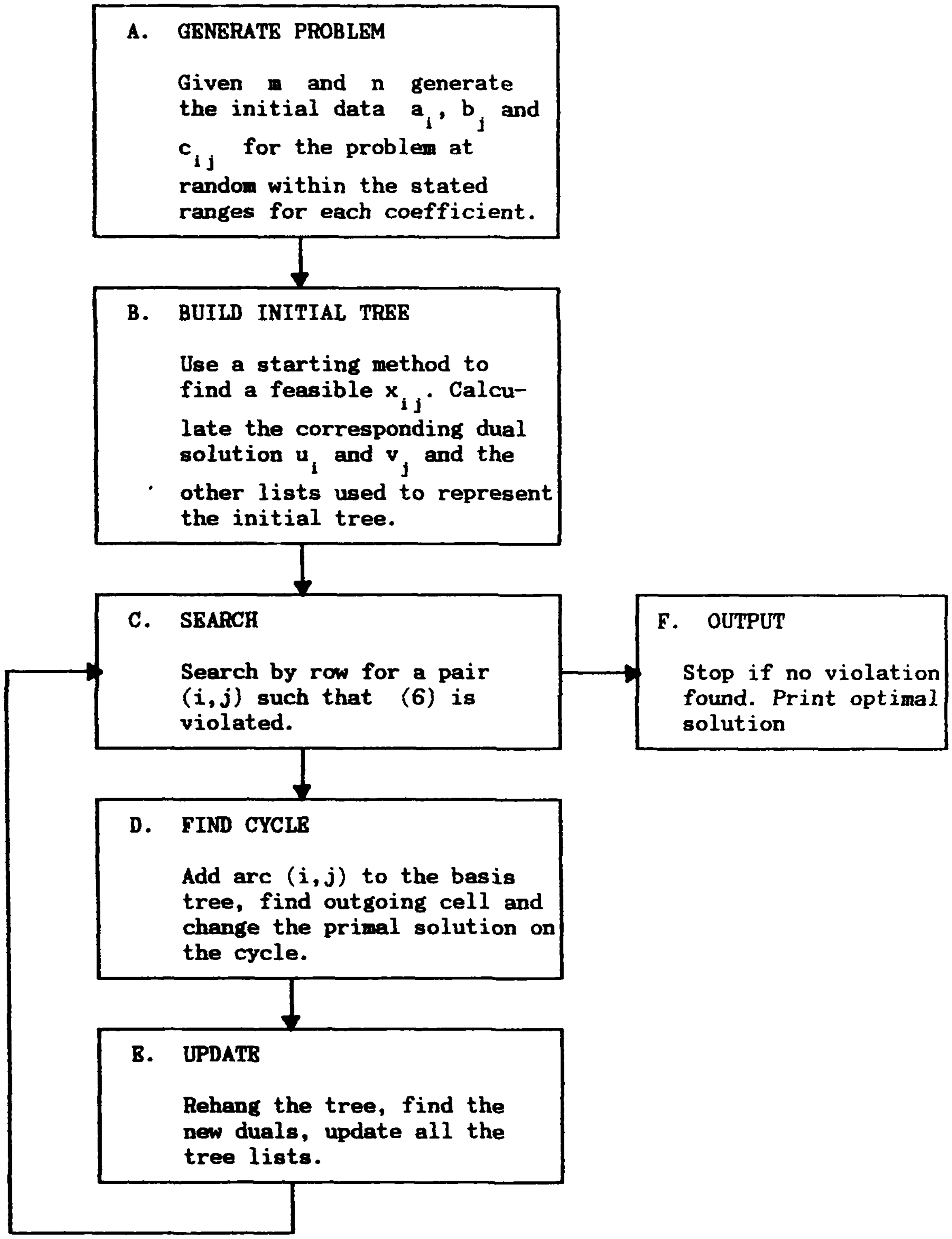

A. GENERATE PROBLEM
Given m and n generate the initial data a_i, b_j and c_{ij} for the problem at random within the stated ranges for each coefficient.
B. BUILD INITIAL TREE
Use a starting method to find a feasible x_{ij}. Calculate the corresponding dual solution u_i and v_j and the other lists used to represent the initial tree.
C. SEARCH
Search by row for a pair (i,j) such that (6) is violated.
D. FIND CYCLE
Add arc (i,j) to the basis tree, find outgoing cell and change the primal solution on the cycle.
E. UPDATE
Rehang the tree, find the new duals, update all the tree lists.
F. OUTPUT
Stop if no violation found. Print optimal solution

Figure 1. Flow diagram for the sequential primal transportation algorithm.

3. THE PARALLEL PRIMAL ALGORITHM

The efficient implementation of any algorithm on a parallel computer necessarily depends on the exact architecture of that computer as well as the characteristics of the algorithm. We discuss the implementation of the primal algorithm of the previous section on a BBN Butterfly Plus computer. The Butterfly Plus is a tightly coupled heterogeneous, shared memory multiprocessor consisting of several Motorola 68020/68881 processors each having 4 megabytes of its own local memory, and accessing the memories of the other processors through a packet switched network. Our code was implemented on a 14 processor machine. Discussions of the implementation on the same parallel computer of a shortest augmenting path algorithm for assignment and travelling salesman problems can be found in Miller and Pekny [18] and Pekny and Miller [19].

Because the Butterfly computer has its memory distributed over the 14 processors it was necessary to distribute the $m \times n$ cost matrix C so that each processor stored several contiguous rows of C in its local memory. From this fact it is obvious from Figure 1 that steps A and C can readily be done in parallel. The time to generate the problem data, step A, was not counted in the computation time so will not be discussed further. The search subroutine, step C, was performed in parallel and was, in fact, the only major subroutine that we were able to perform in parallel in this version of the code.

Subroutine B, build initial tree, was carried out sequentially because (a) it was called only once (b) requires only 5 to 10 percent of the total computation time. Computational experiments with parallel starting tree bases will be reported on elsewhere.

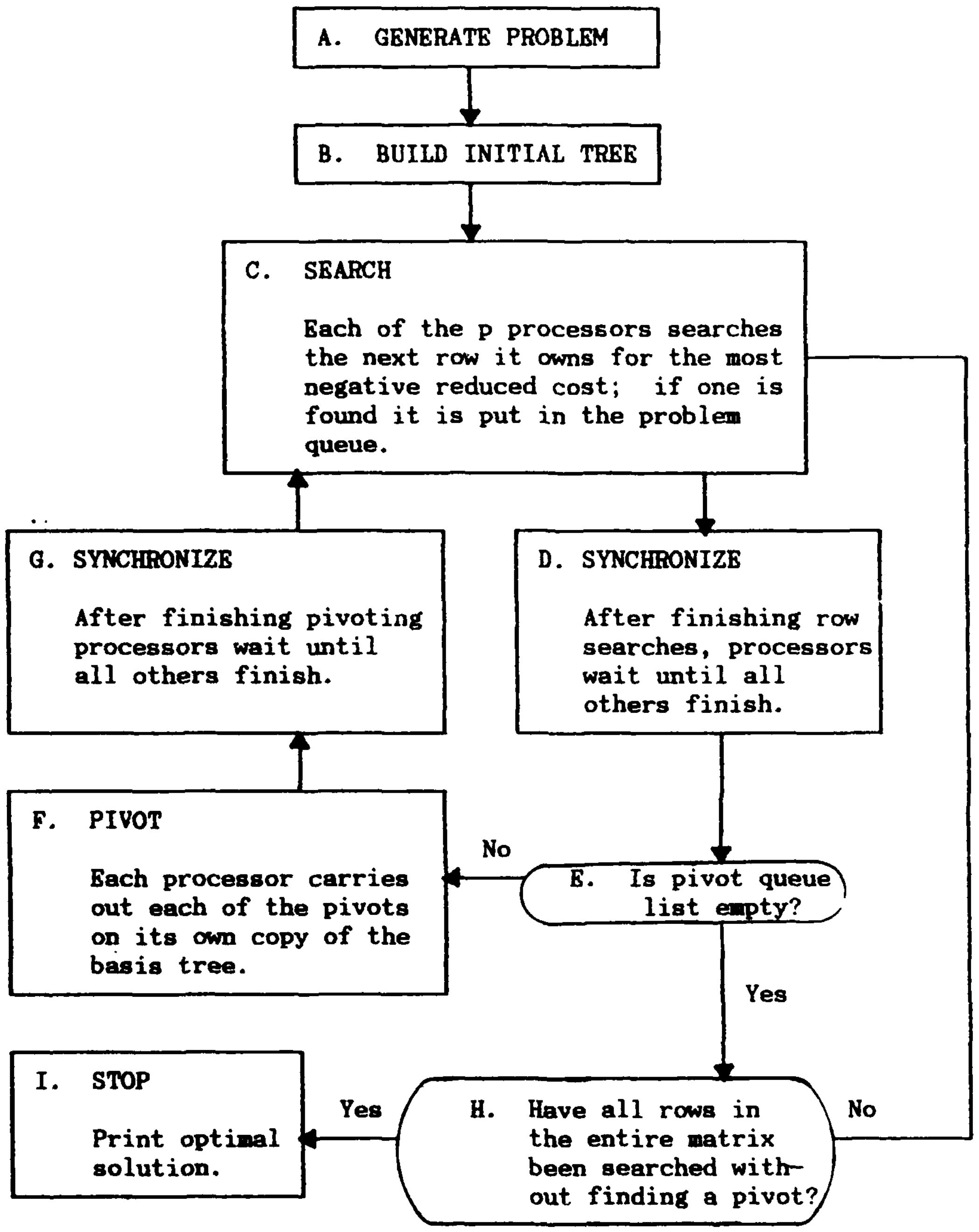

A. GENERATE PROBLEM
B. BUILD INITIAL TREE
C. SEARCH
Each of the p processors searches the next row it owns for the most negative reduced cost; if one is found it is put in the problem queue.
G. SYNCHRONIZE
After finishing pivoting processors wait until all others finish.
D. SYNCHRONIZE
After finishing row searches, processors wait until all others finish.
F. PIVOT
Each processor carries out each of the pivots on its own copy of the basis tree.
No
E. Is pivot queue list empty?
Yes
I. STOP
Print optimal solution.
Yes
H. Have all rows in the entire matrix been searched without finding a pivot?
No

Figure 2. Flow diagram for the parallel primal transportation algorithm.

We now describe the parallel primal transportation algorithm shown in Figure 3 on the assumption that p (>1) processors are available when the problem is being solved. Each processor stores the rows it owns in a wrap-around list so that when it reaches the end of its list it goes back to the beginning. It maintains a pointer to the current row that is next to be searched; the pointer initially points to the first row it owns.

After the initial tree is built each processor obtains its own copy of the lists defining the initial tree. Then subroutine C is invoked, and each processor searches its current row for the entry having the most negative reduced cost. If such a most negative reduced cost is found by a processor it writes the row and column in which it was found and its reduced cost in the pivot queue list; then the processor updates its current row pointers and goes to synchronization step D where it waits until all processors have finished their searches. This synchronization step is necessary in order that the pivot queue list is completely defined before pivot subroutine F is called.

Control then passes to step E in which the question is asked whether the pivot queue list is empty. If the answer is yes control passes to step H in which the question is asked whether all of the rows in the entire matrix have been searched without finding a pivot. If the answer is yes, control is passed to step I and the optimal solution is printed. An answer of no sends the computer back to search step C.

In the case that the answer to the question in E is no then subroutine F is called and all p processors *simultaneously* perform all the pivots listed in the pivot queue on their own copies of the basis tree. The reason this is done is that it is faster for each processor to carry out the calculation rather than have just one of them do the calculation and communicate the result. As discussed in the next section, we also tried to break the pivoting process down into small tasks and have individual processors carry out these tasks. However, the sizes of the tasks, sometimes referred to as their *granularity*, was found to be too small for a parallel processing strategy to reduce the time over the previously reported strategy.

After each processor completes its pivot task, it moves to synchronization step G where it waits until all processors have finished pivoting. The reason for the synchronization step here is that it is necessary to prevent the pivot queue list from being altered before one or more of the processors have completed their pivot tasks.

Once all processors indicate in step G that they have finished pivoting control returns to search step C, completing the main computational loop.

4. ATTEMPT TO PARALLELIZE THE PIVOT STEP

At the end of the search step each processor has located the most negative element in its current row, or has determined that the row has no negative elements. Hence the problem queue has up to 14 possible pivots that can be performed. However it is not always possible to perform two (or more) pivots simultaneously because the pivot operations may require changes to be made on the basis functions of some of the same nodes.

To make these ideas precise let Q be the set of potential pivots, and let k be an element of Q. Define $T(k)$ to be the set of basis tree nodes at which one of the five node functions $p(k)$, $d(k)$, $x(k)$, $u(k)$, $v(k)$ is changed. Two pivots h and k are said to be *independent* if $T(h) \cap T(k) = \emptyset$, and are said to be *dependent* if $T(h) \cap T(k) \neq \emptyset$. Consider the graph $P = \{Q,E\}$ whose nodes consist of the pivots k in the pivot queue, and whose edges (h,k) belong to E if pivots h and k are dependent. We would like to find a *maximal independent subset* S in P which would then consist of a set of pivots which can be done in parallel. Since finding a maximal independent subset of a set is a well known NP hard problem, we used the following heuristic program:

```
(1) Let $S = \emptyset$.
(2) Find a node $k$ of largest degree in $P$.
(3) Remove $k$ from $Q$; put $k$ in $S$.
(4) For each edge $(k,h)$ in $E$, remove $h$ from $Q$, and remove $(k,h)$ from $E$.
(5) If $Q \neq \emptyset$ go to (2) else go to (6).
(6) Stop. Set $S$ consists of independent pivots.
```

Since at least one node is removed from Q at each step of the algorithm, it runs very quickly. The set S will not necessarily be the maximal independent subset of P, but will usually be quite good.

We implemented the above procedure but found that it was not computationally as efficient as the idea described in the previous section. The main difficulty is that the computational effort of determining for each k in Q the set $T(k)$ is essentially the same as actually performing the pivot on k. To that time the work of determining the independent set S and actually carrying out the independent pivots in S must be added. Overall the computation time was increased by the attempt to parallelize the pivot step, so it was abandoned.

5. COMPUTATIONAL RESULTS

In the course of our computational experiments we solved more than 500 randomly generated fully dense transportation problems on the Butterfly computer. The averaged data is presented graphically in Figures 1 - 5. The raw computational data appears in the appendix.

Figure 1 shows the performance of the parallel primal algorithm on square transportation problems for sizes $n = 500,1000,\ldots,3000$, and for shipping amounts ranging from 1 (assignment problems) to 1000. As noted in [22] for much smaller problem sizes, the execution times increase with problem size and with shipping amounts. We fitted these data points with an exponential function of the form Kn^a and found that the exponent a ranged from $a = 2.0$ for assignment problems to $a = 2.2$ for transportation problems having shipping amounts in the range [1,1000]. The correlation coefficients for all of these were greater than .999, indicating an extremely good fit.

Figure 2 shows the effect of varying cost ranges and problem size on execution times for assignment problems. The larger the cost range the less the dual degeneracy the problem has. To put it differently the smaller the cost range is the larger is the number of alternative optimal solutions. The primal transportation algorithm is able to take advantage of dual degeneracy, and is able to solve problems having smaller cost ranges much faster than those having larger cost ranges. In contrast, some dual methods find the opposite phenomena to be the case, see [19].

In order to determine the speedup factor of the parallel primal algorithm we solved assignment problems with $n = 500$, 750, and 1000 on the Butterfly computer with just one processor (no larger problems could be solved on a single processor) and also solved the same problems with 2, 4, 6, 7, 8, 10, 12, and 14 processors. The averages of the execution times of these runs are plotted in Figure 3. For problems having the same cost range the speedup factor is the ratio of the execution time on one processor divided by the smallest execution time achieved by using any number of processors. Note that the speedup factor increases with problem size being about 2.2 for $n = 500$ and increasing to about 2.5 for $n = 1000$. Note also that the most effective number of processors seems to be in the range 6 to 8 for this range of problem sizes. As problems get larger, the primal algorithm should be able to use more processors effectively to improve the efficiency of the search part of the algorithm, as will become evident in the next two figures.

Since we were able to parallelize the search phase of the algorithm but not the pivot phase it is important to find out whether the percentage of time spent increased with problem size. We made runs on the sequential version of the algorithm using a SUN work station. The average percentage of search time for three assignment problems with sizes ranging from $n = 100$ to $n = 2500$ and for two cost ranges [0,1000] and [0,10000], are shown in the bar graph of Figure 4. The search percentage ranged from about 55% for $n = 100$ to about 95% for $n = 2500$. The fact that we parallelized the part of the sequential algorithm that increases with n was the main reason for the good performance of the parallel primal algorithm.

In order to investigate further the speedup of the parallel versus the sequential code we made use of a speedup formula due to Rettberg and Thomas [20]. Let T be the total execution time, let f the fraction of time spent in the search phase of the algorithm, and let S be the speedup factor. Then speedup, S, is

$$S = \frac{T}{(1-f)T + fT/p} = \frac{p}{(1-f)p + f}$$

In Figure 5 we have plotted this curve for problems having costs in the range [0,1000] with $f = .83$ as can be observed in Figure 4. We also computed the observed speedup values by dividing the execution time in Figure 3 for each value of p into its value for $p = 1$. These points are also plotted in Figure 5. The maximum speedup observed when $p = 8$ was 70 percent of its theoretical maximum value. We were unable to continue this study for larger values of p because of the memory limitation of a single processor of the Butterfly computer.

6. CONCLUSIONS

This paper is a continuation of the experimental investigations begun in the 1970's on primal simplex transportation codes. It presents evidence that the primal code is well suited for parallel computation at least with the computer architecture of the Butterfly computer. Significant speedup factors for parallel over sequential machines were achieved which enabled the parallel computer to solve fully dense transportation and assignment problems which are at least one order of magnitude larger than those previously reported.

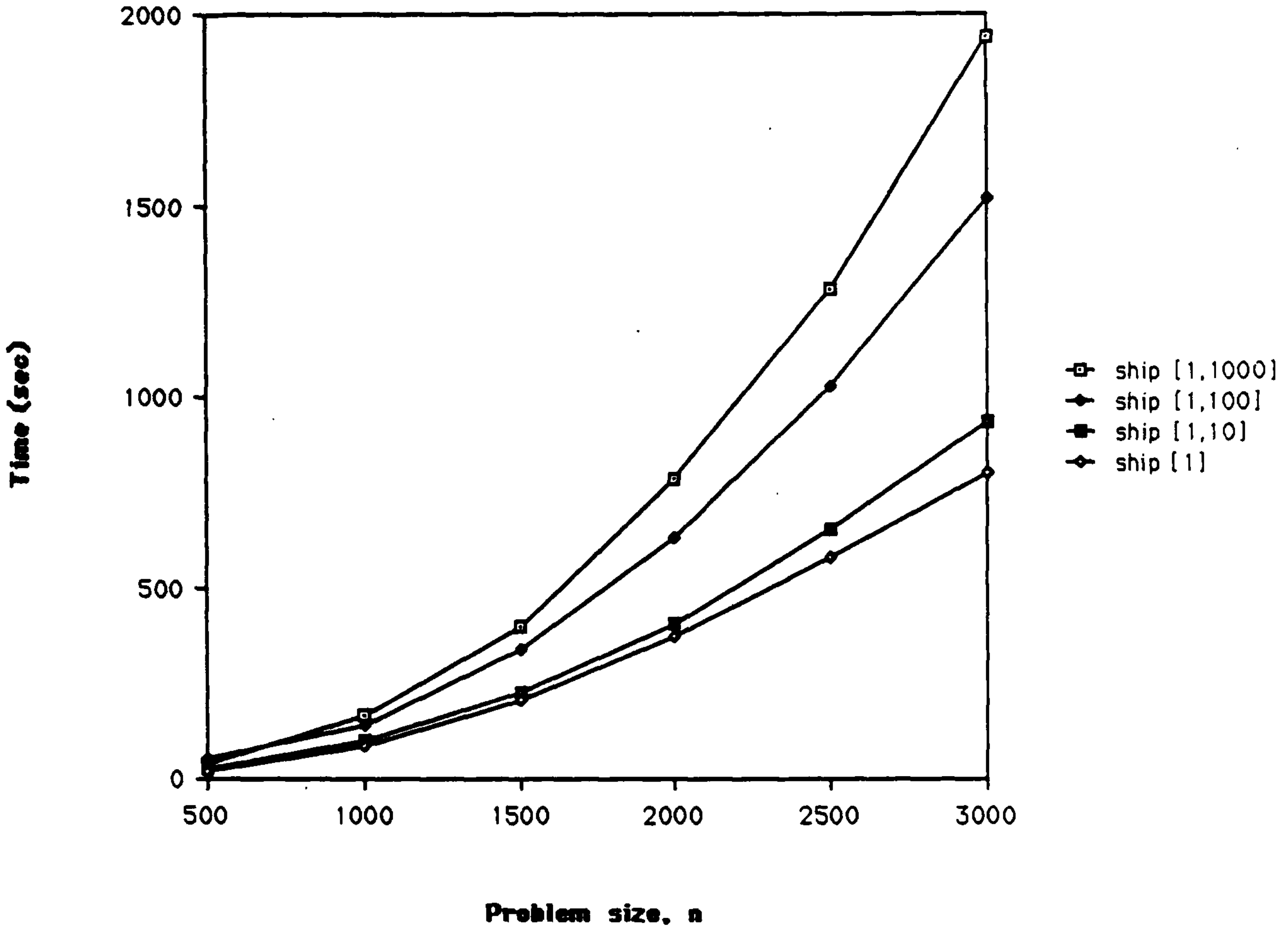


Figure 1. The effect on execution time of varying shipping amounts for square dense transportation problems. The cost range was [0,1000]. The lowest curve gives solution times for assignment problems. Fourteen parallel processors were used.

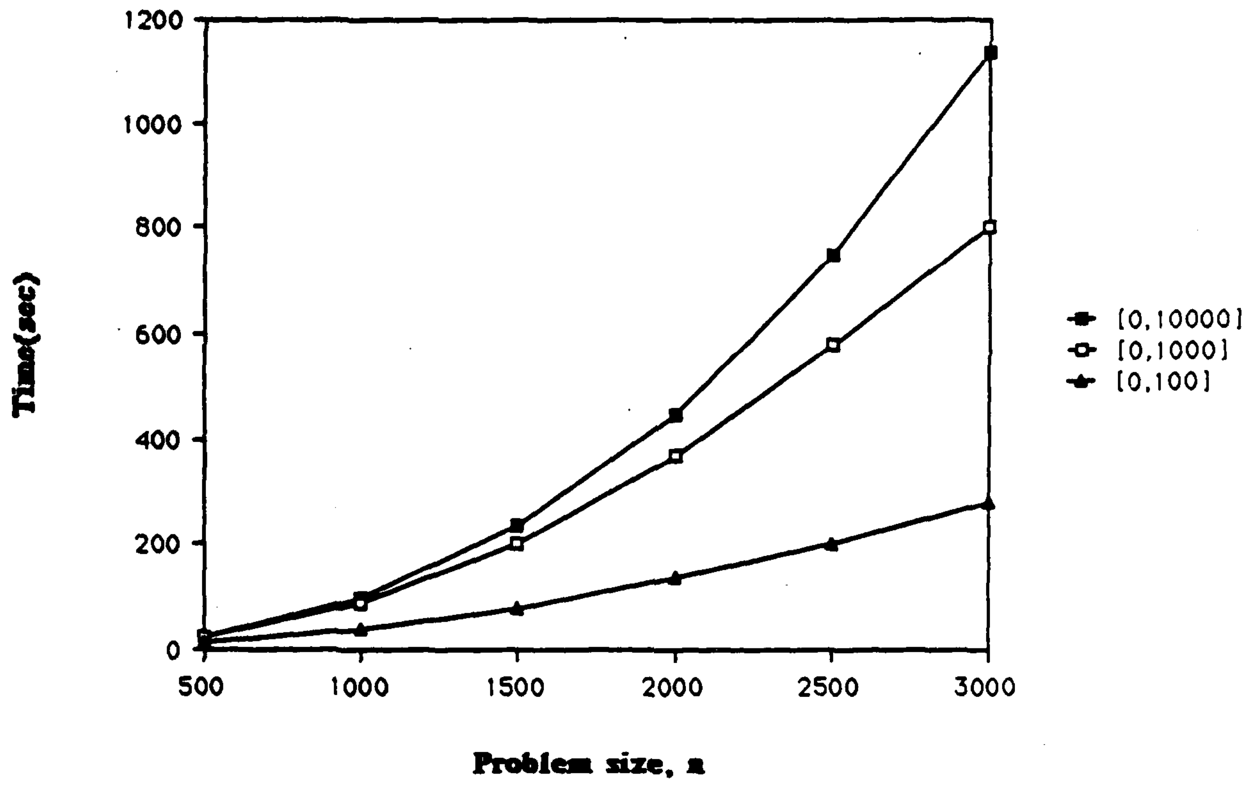


Figure 2. The effect on execution times of varying cost ranges for square assignment problems. Fourteen parallel processors were used.

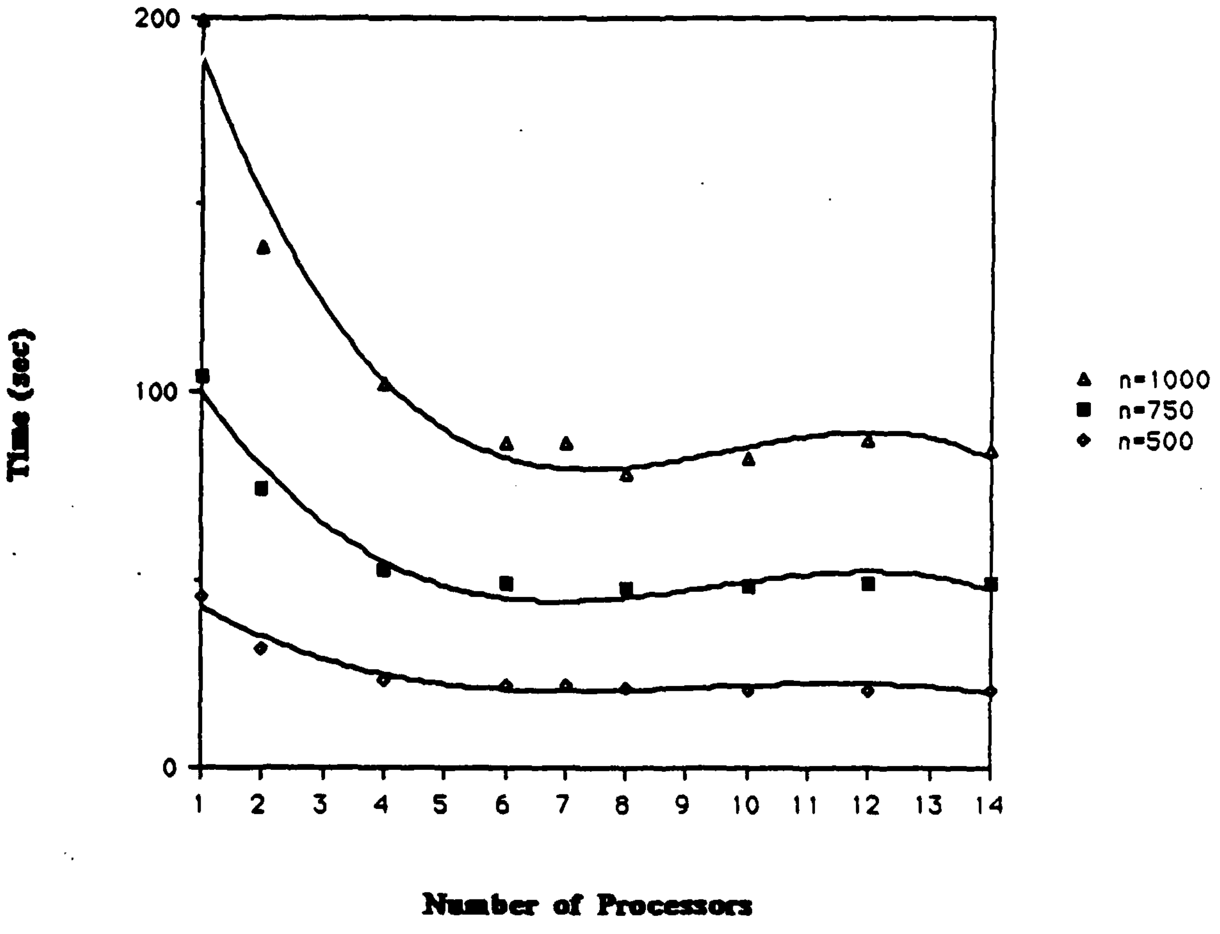


Figure 3. The effect on execution times of changing the number of parallel processors for three different assignment problem sizes. For each curve the maximum speedup factor is the ratio of the time for one processor divided by the smallest time for any number of processors.

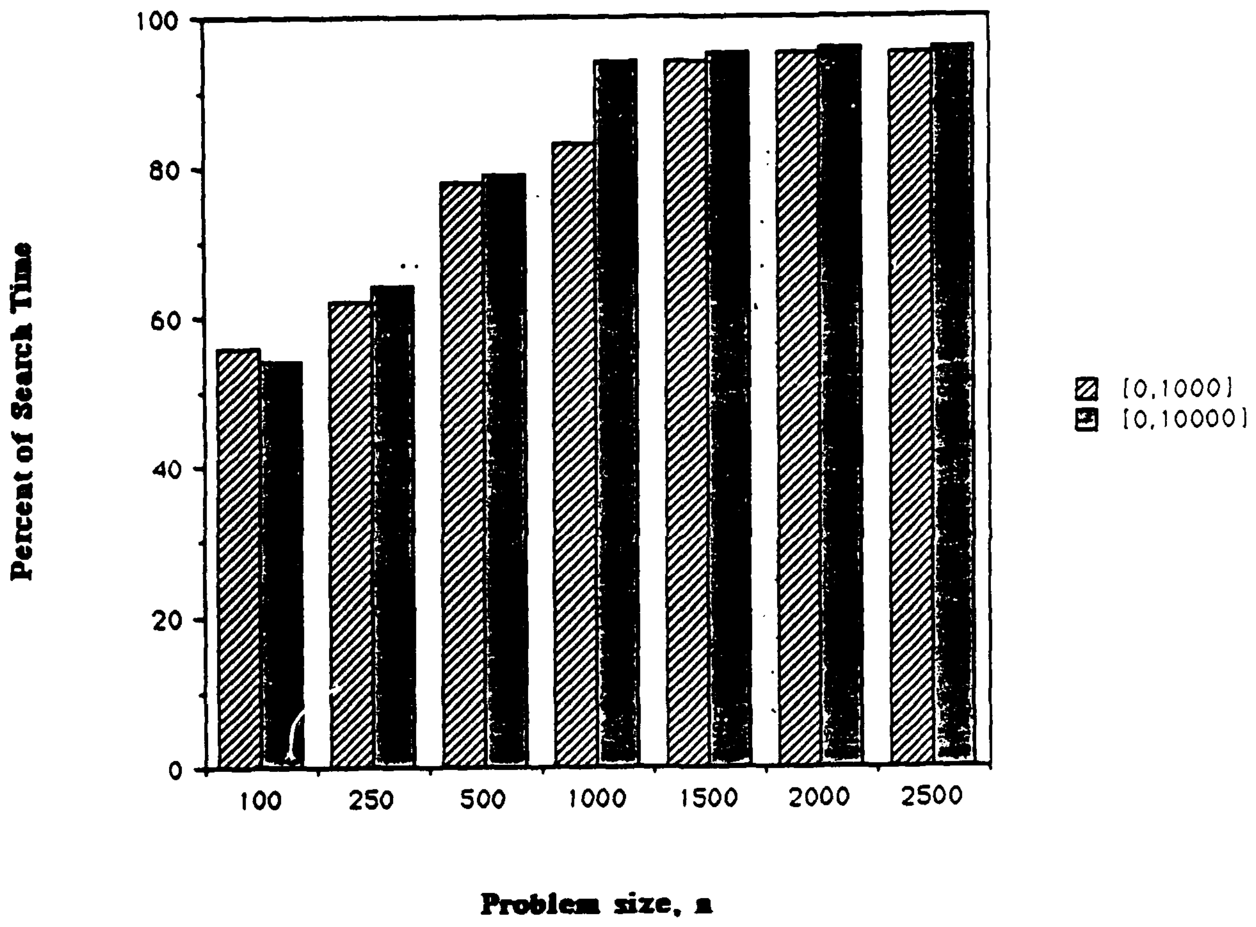


Figure 4. Search time as a percentage of run time on a sequential (SUN) computer. Two different cost ranges [0,1000] and [0,10000] were used.

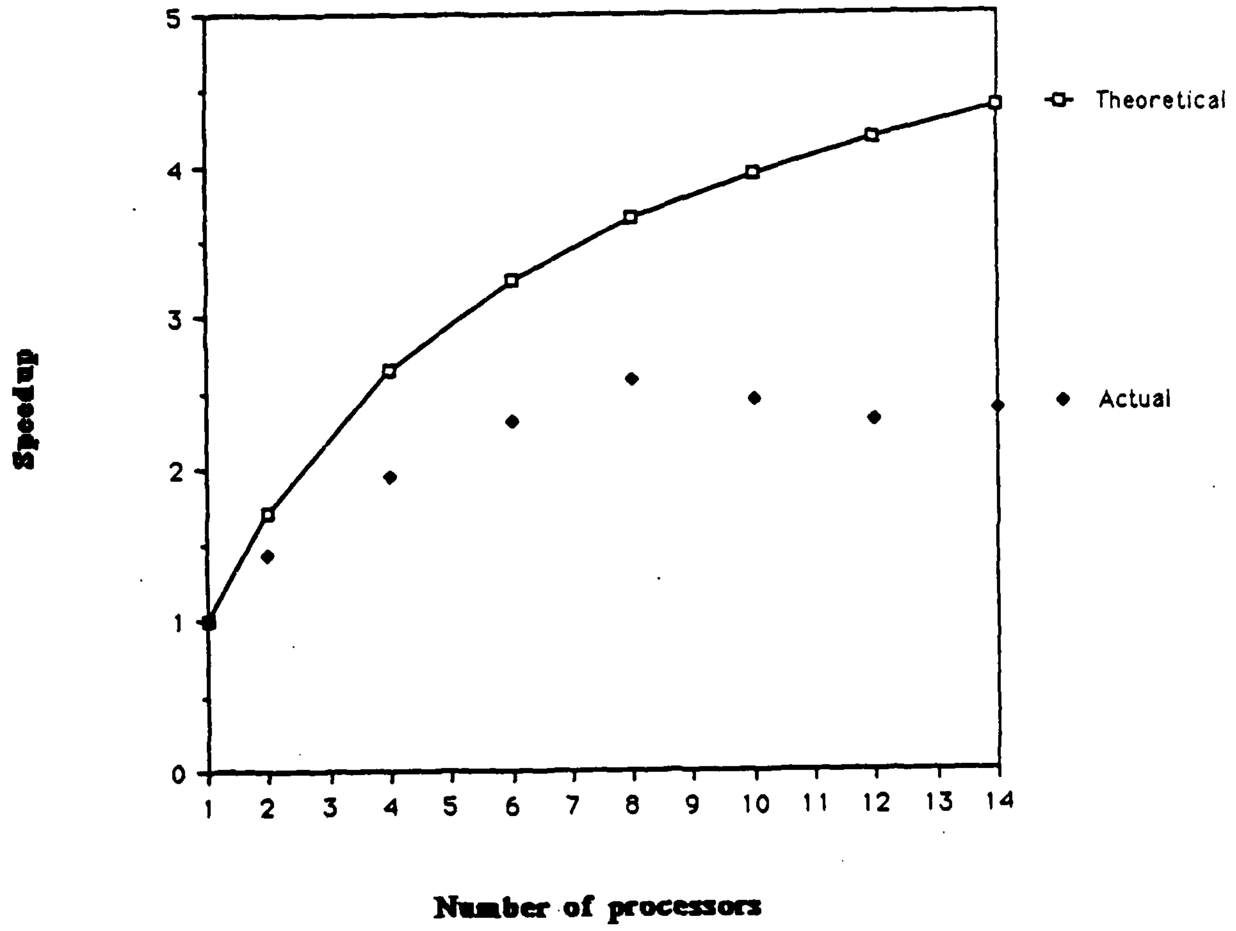


Figure 5. Theoretical maximum speedup and actual speedup as a function of the number of processors in solving assignment problems with cost ranges [0,1000]. The maximum speedup, achieved with 8 processors, was approximately 70 percent of the theoretical maximum.

REFERENCES

[1] Barr, R. S., F. Glover and D. Klingman. "The Alternating Basis Algorithm for Assignment Problems." *Mathematical Programming* 13, (1977) 1-13.

[2] Bertsekas, D. P.. "A New Algorithm for the Assignment Problem," *Mathematical Programming* 21 (1981) 152-171.

[3] Bradley, G. H., G. G. Brown and G. W. Graves. "Design and Implementation of Large Scale Primal Transshipment Algorithms." *Management Science* 24, (1977) 1-34.

[4] Charnes, A., and W. W. Cooper. *Management Models and Industrial Applications of LInear Programming,* Vol. I. Wiley, New York, 1961.

[5] Charnes, A., D. Karney, D. Klingman, J. Stutz and F. Glover. 1975. "Past, Present and Future of Large Scale Transshipment Computer Codes and Applications." *Comp. Opns. Res.* 2 (1975) 71-89.

[6] Dantzig, G. B.. *Linear Programming and Extensions.* Princeton U. Press, Princeton, N.J., 1963.

[7] Derigs, V., "The Shortest Augmenting Path Method for Solving Assignment Problems — Motivation and Computational Experience," *Annals of Operations Research* 4 (1985/6) 57-102.

[8] Dorhout, B., "Het lineaire toewijzingsproblem, vergelijking van algorithmen," Report BN21, Stichting Mathematisch Centrum, Amsterdam (1973).

[9] Ford, L. R. and D. R. Fulkerson, *Flows in Networks,* Princeton University Press, Princeton, NJ. 1962.

[10] Glover, F. and D. Klingman. "Comment on a Note by Hatch on Network Algorithms." *Operations Research,* 26, (1978) 370-374.

[11] Glover, F., D. Karney and D. Klingman. "Implementation and Computational Comparisons of Primal, Dual and Primal-Dual Computer Codes for Minimum Cost Network Flow Problems." *Networks* 4, (1974) 191-212.

[12] Hatch, R. S. "Bench Marks Comparing Transportation Codes Based on Primal Simplex and Primal-Dual Algorithms." *Operations Research* (1975) 23, 1167-1171.

[13] Hung, M. S. and W. O. Rom, "Solving the Assignment Problem by Relaxation," *Operations Research* 28 (1980) 969-982.

[14] Kennington, J. L., and R. V. Helgason, *Algorithms for Network Programming,* Wiley-Interscience, New York, 1980.

[15] Kuhn, H. W., "The Hungarian Method for the Assignment Problem," *Naval Research Logistics Quarterly* 2(1955) 83-97.

[16] Martello, S. and P. Toth, "Linear Assignment Problems," *Annals of Discrete Math.* 31 (1987) 259-282.

[17] McGinnis, L. F., "Implementation and Testing of a Primal-Dual Algorithm for the Assignment Problem," *Operations Research* 31 (1983) 277-291.

[18] Miller, D. L., and J. F. Pekny, "Results from a Parallel Branch and Bound Algorithm for Solving Large Asymmetric Travelling Salesman Problems," Working Paper, April, 1988.

[19] Pekny, J. F. and D. L. Miller, "A Parallel Branch and Bound Algorithm for Solving Large Asymmetric Traveling Salesman Problems," Working Paper, May, 1988.

[20] Rettberg, R., and R. Thomas, "Contention is no Obstacle to Shared-Memory Multiprocessing," *Comm. A.C.M.*, 29 (1986) 1202-1212.

[21] Srinivasan, V. and G. L. Thompson, "Accelerated algorithms for labelling and relabelling of trees, with applications to distribution problems," *J. Assoc. Comput. Mach.* 9(1972) 712-726.

[22] Srinivasan, V. and G. L. Thompson, "Benefit-cost analysis of coding techniques for the primal transportation algorithm," *J. Assoc. Comput. Mach.* 20(1973) 194-213.

[23] Srinivasan, V. and G. L. Thompson, "Cost Operator Algorithms for the Transportation Problem," *Mathematical Programming* 12 (1977) 372-391.

[24] Thompson, G. L., "A Recursive Method for Solving Assignment Problems," in: *Studies on Graphs and Discrete Programming, Annals of Discrete Mathematics II*, ed. P. Hansen (North-Holland, Amsterdam, 1981) 319-343.

[25] Tomizawa, N., "On Some Techniques Useful for Solution of Transportation Network Problems." *Networks* 1 (1971) 173-194.

Appendix

The following four tables contain the run time information presented in Figures 1-5, and also additional information concerning the dispersion of run times, build initial tree time, search and pivot time, number of rows searched, number of pivots, and number of pivots per search. This information may be useful to others who are implementing the primal or other transportation algorithms on parallel computers.

Ship [1]

runs	size	run time avg/std	run time min/max	build tree time	search and pivot time	rows searched	number pivots	pivots /search
10	500	22.3/1.3	20.5/24.7	5.3	16.3	19961	9601	6.74
10	1000	86.3/5.3	78.0/93.5	20.9	64.2	59424	30536	7.21
10	1500	203.4/12.1	188.5/229.1	47.8	153.6	118073	60914	7.23
10	2000	370.4/16.7	346.8/402.4	82.0	285.8	192375	103853	7.56
10	2500	578.1/19.1	556.0/610.0	130.2	444.8	254412	145625	8.02
10	3000	801.4/32.5	753.1/854.3	181.8	615.6	309771	187116	8.46

Ship [1,10]

runs	size	run time avg/std	run time min/max	build tree time	search and pivot time	rows searched	number pivots	pivots /search
10	500	25.8/2.1	23.5/29.2	4.6	20.6	15187	5954	5.47
10	1000	99.9/13.6	87.9/134.4	17.8	80.9	40616	16829	5.75
10	1500	223.6/20.6	196.6/270.8	39.2	182.4	71565	30166	5.89
10	2000	405.4/35.4	375.0/485.0	71.4	331.3	109752	46122	5.87
10	2500	653.4/34.5	609.3/707.0	111.8	538.5	156575	66126	5.92
9	3000	931.2/85.9	849.2/1143.5	158.5	768.7	194868	85376	6.15

Ship [1,100]

runs	size	run time avg/std	run time min/max	build tree time	search and pivot time	rows searched	number pivots	pivots /search
10	500	56.3/66.1	32.0/244.4	4.5	51.2	14978	5148	4.80
10	1000	142.9/27.0	125.8/218.8	17.5	124.2	37595	13044	4.82
10	1500	337.9/43.2	308.3/447.2	40.0	296.1	64167	22166	4.84
10	2000	635.5/84.5	577.8/843.6	69.0	564.0	98075	33001	4.68
10	2500	1025.8/94.2	947.7/1250.0	109.0	913.7	124921	44339	4.96
10	3000	1522.7/179.5	1404.3/1948.3	155.8	1363.2	161806	56584	4.90

Ship [1,1000]

runs	size	run time avg/std	run time min/max	build tree time	search and pivot time	rows searched	number pivots	pivots /search
10	500	38.5/3.8	33.8/45.2	4.5	33.4	14683	5154	4.90
10	1000	168.4/33.9	146.3/261.3	17.5	149.6	39022	12891	4.60
10	1500	399.5/48.3	351.5/524.6	38.8	358.8	64849	21729	4.69
10	2000	784.2/97.0	696.0/1027.6	70.1	711.3	101129	32223	4.45
10	2500	1279.5/110.8	1185.2/1525.9	110.2	1166.1	128667	42681	4.66
8	3000	1937.1/262.6	1795.2/2578.5	156.7	1776.3	163039	54173	4.67

Table 1. Raw data for the graphs in Figure 1. All problems were randomly generated with costs in the range [0,1000]. All problems were solved using a 14 processor butterfly computer.

Cost [0,100]

runs	size	run time avg/std	run time min/max	build tree time	search and pivot time	rows searched	number pivots	pivots /search
10	500	13.9/0.8	12.5/15.5	7.2	5.9	7996	3880	6.81
10	1000	41.1/1.3	38.9/42.9	28.0	11.7	9805	5124	7.33
10	1500	79.3/3.1	74.1/83.2	60.7	16.4	11725	5871	7.02
10	2000	135.4/5.9	125.6/142.3	107.0	25.4	13386	6470	6.78
10	2500	200.3/4.0	195.5/207.5	168.5	28.0	15190	7222	6.66
10	3000	281.9/10.6	264.9/301.9	243.7	33.7	17133	7844	6.42

Cost [0,1000]

runs	size	run time avg/std	run time min/max	build tree time	search and pivot time	rows searched	number pivots	pivots /search
10	500	22.3/1.3	20.5/24.7	5.3	16.3	19961	9601	6.74
10	1000	86.3/5.3	78.0/93.5	20.9	64.2	59424	30536	7.21
10	1500	203.4/12.1	188.5/229.1	47.8	153.6	118073	60914	7.23
10	2000	370.4/16.7	346.8/402.4	82.0	285.8	192375	103853	7.56
10	2500	578.1/19.1	556.0/610.0	130.2	444.8	254412	145625	8.02
10	3000	801.4/32.5	753.1/854.3	181.8	615.6	309771	187116	8.46

Cost [0,10000]

runs	size	run time avg/std	run time min/max	build tree time	search and pivot time	rows searched	number pivots	pivots /search
10	500	23.6/1.6	19.6/25.2	5.3	17.7	22605	10718	6.64
10	1000	98.4/6.7	89.3/111.4	21.4	75.7	73509	38064	7.26
10	1500	237.9/10.8	219.6/257.5	46.6	189.3	157858	84369	7.50
10	2000	449.8/29.3	422.3/504.4	82.7	364.4	265005	146606	7.75
10	2500	748.7/41.0	668.8/819.5	135.7	609.8	402653	221668	7.71
10	3000	1136.4/60.0	1001.1/1220.7	185.2	947.2	567511	324035	8.00

Table 2. Raw data for Figure 2. All the problems were randomly generated assignment problems of stated sizes and costs chosen in the given ranges. All were solved using a 14 processor Butterfly computer.

Execution Time vs. Processors

Processors	n = 500	n = 750	b = 1000
1	45.7	104.3	199.3
2	32.0	73.7	138.3
4	23.5	52.3	102.2
6	22.23	49.2	86.5
7	22.3	—	86.3
8	21.6	47.7	77.6
10	21.0	48.3	81.8
12	21.0	48.7	86.8
14	21.0	49.3	83.9

Table 3. Data used for Figures 3 and 5. All problems were randomly generated assignment problems with costs in the range [0,1000]

Search Time as a Percentage of Execution Time

n	Cost [0,1000]	Cost [0,10000]
100	56.4	54.1
250	61.7	63.9
500	78.1	78.6
1000	82.8	94.2
1500	94.2	95.5
2000	95.0	96.0
2500	94.8	96.4

Table 4. Data for the graph in Figure 4. The numbers are average times for the solution of three randomly generated assignment problems for various problem sizes and two different cost ranges. All problems were solved by a sequential version of the primal code on a SUN work station.