W.P. #88-89-49

Management Science Research Report No. MSRR 552

1

A PARALLEL SHORTEST AUGMENTING PATH ALGORITHM FOR THE ASSIGNMENT PROBLEM

by

Egon Balas[1], Donald Miller[2], Joseph Pekny[3], and Paolo Toth[4]

April 1989

1) GSIA, Carnegie Mellon University
2) CRD, E.I. Du Pont de Nemours & Company Inc.
3) EDRC, Carnegie Mellon University
4) DEIS, University of Bologna

The research underlying this report was supported by Grant ECS-8601660 of the National Science Foundation, Contract N00014-85-K-0198 with the Office of Naval Research, and NATO Research Grant No. 228-88. Reproduction in whole or in part is permitted for any purpose of the U.S. Government.

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

91 3 28 258

Abstract

We describe a parallel version of the shortest augmenting path algorithm for the assignment problem. While generating the initial dual solution and partial assignment in parallel does not require substantive changes in the sequential algorithm, using several augmenting paths in parallel does require a new dual variable recalculation method. The parallel algorithm was tested on a 14-processor Butterfly Plus computer, on problems with up to 900 million variables. The speedup obtained increases with problem size. The algorithm was also embedded into a parallel branch and bound procedure for the traveling salesman problem on a directed graph, which was tested on the Butterfly Plus on problems involving up to 7,500 cities. To our knowledge, these are the largest assignment problems and traveling salesman problems solved so far.

1. Introduction

The assignment problem can be defined either on a directed graph, in which case an assignment (a solution) is a spanning union of directed cycles, or on an undirected bipartite graph, in which case an assignment is a perfect matching. We work with this latter formulation. Given a bipartite graph $G = (S \cup T, A)$ with arc costs c_{ij}, $(i,j) \varepsilon A$, where $|S| = |T| = n$, the assignment problem (AP) asks for a pairing (matching, assignment) of the nodes in S to those in T that minimizes the sum of costs of arcs in the pairing. It can be stated as

(1) $$\min \Sigma(c_{ij}x_{ij} : i \varepsilon S,\ j \varepsilon T)$$

subject to

(2) $$\Sigma(x_{ij} : j \varepsilon T) = 1 \qquad i \varepsilon S$$
$$\Sigma(x_{ij} : i \varepsilon S) = 1 \qquad j \varepsilon T$$

(3) $$x_{ij} \varepsilon \{0,1\}, \qquad (i,j) \varepsilon A$$

where $x_{ij} = 1$ if and only if node $i \varepsilon S$ is paired with node $j \varepsilon T$, i.e. arc (i,j) is in the matching. Condition (3) can be replaced by

(4) $$x_{ij} \geq 0, \qquad (i,j) \varepsilon A$$

because the resulting linear program has only integer basic solutions due to the total unimodularity of the coefficient matrix of the system (2).

The linear program dual to (AP) can be stated as

(5) $$\max \Sigma(u_i : i \varepsilon S) + \Sigma(v_j : j \varepsilon T)$$

subject to

(6) $$u_i + v_j \leq c_{ij}, \qquad (i,j) \varepsilon A.$$

It is well known that a vector x satisfying (2), (3) is an optimal assignment if and only if there exists a vector (u,v) satisfying (6) and such that

$$(7) \qquad c_{ij} - u_i - v_j \begin{cases} = 0 & \text{if } x_{ij} = 1 \\ \geq 0 & \text{otherwise.} \end{cases}$$

The numbers $\bar{c}_{ij} := c_{ij} - u_i - v_j$ are called reduced costs.

Both the primal and dual simplex methods have their specialized versions for the assignment problem [3,2,21]. The most popular early approach known as the Hungarian method [18,12,19,8,20], can be viewed as primal-dual in nature, although in fact it never uses or produces a primal basis. The same can be said about the Shortest Augmenting Path method [6,9,11,15,28], which treats (AP) as a specialized minimum cost network flow problem. Our reason for choosing this latter approach for parallelization is that (a) we consider it to be the most promising and (b) it contains parallelisms of somewhat higher order of granularity than the other approaches. A different approach that easily lends itself to parallelization, is based on relaxation techniques [4].

We note that the rows and columns of the cost matrix c correspond to the nodes in S and in T, respectively. Throughout the paper, we will refer to elements of S (of T) either as nodes, or as rows (as columns).

2. The Sequential Algorithm

The procedure starts by finding an initial solution (Phase 0), i.e. a partial assignment that is optimal in the subgraph induced by the assigned nodes. This is achieved by constructing a basic feasible solution to the dual of (AP) and then finding a maximum cardinality matching in the subgraph having only those arcs with zero reduced cost. We do this in $O(n \cdot z)$ time, where z is the number of arcs with zero reduced costs, and it typically yields a considerably larger number of initial pairings than the usual greedy heuristics.

Next the procedure enters an alternating sequence of two phases: augmenting path finding (Phase 1) and updating (recalculation) of the dual variables and of the (primal) assignment (Phase 2).

In Phase 1, an unassigned node $i \in S$ is selected and a shortest (with respect to the current reduced costs) augmenting path is found from it to some unassigned node $j \in T$. This is done by growing an alternating tree rooted at node i by a slightly modified version of Dijkstra's $O(n^2)$ labeling procedure [10]. The modification, necessitated by the fact that the shortest path to be found has to be an alternating one, consists in restricting Dijkstra's selection rule to successors of nodes in S, while the successors of nodes in T are uniquely determined and thus leave no choice.

In Phase 2 the labels generated in Phase 1 are used to calculate new values for the dual variables, and the augmenting path is used to generate a new assignment, namely the symmetric difference between the old assignment and the augmenting path. The new assignment matches one more pair of nodes than the previous one. The algorithm stops when all the nodes have been paired.

A more precise statement of the algorithm follows.

```
begin
* * * Phase 0 * * *
    begin
        $u_i := \min\{c_{ij} : j \in T\}$,        $i \in S$
        $v_j := \min\{c_{ij} - u_i : i \in S\}$,     $j \in T$
        $\bar{A} := \{(i,j) \in A : c_{ij} - u_i - v_j = 0\}$
        find a maximum matching $A^*$ in $\bar{G} := (S \cup T, \bar{A})$
    end
    while $|A^*| < n$ do
* * * Phase 1 * * *
        begin
            choose an unassigned row $i_1 \in S$
* * * initialize the set of unlabeled (UC) and labeled (LC) columns * * *
            $UC := T,\ LC := \emptyset$
* * * initialize the labels $\lambda_j$ and the predecessors $p_j$ * * *
            for each $j \in T$ do $\lambda_j := c_{i_1 j} - u_{i_1} - v_j,\ p_j := 0$
* * * find shortest augmenting path * * *
            repeat
                find $j \in UC$ with $\lambda_j = \min\{\lambda_k : k \in UC\}$
```

```
                    $UC := UC \setminus \{j\}$, $LC := LC \cup \{j\}$
                    if j is assigned then
                        begin
                            $i$ := row assigned to column $j$
                            for each $k \in UC$ do
                                begin
                                    $\bar{\lambda} := \lambda_j + c_{ik} - u_i - v_k$
                                    if $\bar{\lambda} < \lambda_k$ then $\lambda_k := \bar{\lambda}$, $p_k := j$
                                end
                        end
                until j is unassigned
            end
*** Phase 2 ***
*** update dual variables ***
            for each $k \in LC \setminus \{j\}$
                begin
                    $i$ := row assigned to column $k$
                    $v_k := v_k + \lambda_k - \lambda_j$
                    $u_i := u_i - \lambda_k + \lambda_j$
                end
            $u_{i_1} := u_{i_1} + \lambda_j$
*** update current assignment ***
            while $p_j \neq 0$ do
                begin
                    $i$ := row assigned to column $p_j$
                    $A^* := A^* \cup \{(i,j)\} \setminus \{(i,p_j)\}$
                    $j := p_j$
                end;
            $A^* := A^* \cup \{(i_1,j)\}$
end
```

When the problems to be solved are large, it pays to use sparse matrix techniques, i.e. to restrict the search for a successor in the augmenting paths to the smallest k elements of each row of the reduced cost matrix for some properly chosen k, and check for dual feasibility of the solution found at the end of the procedure. If dual feasibility is violated, the corresponding rows and columns must be reassigned; but by proper choice of k, the probability of a need for such reassignments can be kept rather low (see [7,8] for a discussion of this procedure).

3. Parallelization: General Considerations

The efficiency of parallelization is usually measured by the *speedup*, i.e. the ratio between the times needed to solve a given problem with a single processor and with p processors. The speedup in turn depends on the time spent by all the processors on actual computing, versus the time spent on inter-processor communication or idling.

A key factor that affects the efficiency of a parallelization is its *granularity*. High granularity parallelism is one that allows each processor to execute substantial amounts of computation before the need for communication arises. Low granularity parallelism is one that requires frequent communication and data transfer between processors. An example of the first type of parallelism is a branch and bound procedure in which every processor works on a different subproblem; while an example of the second type is a sorting algorithm in which each processor compares two items and passes on the result. On most existing parallel computers higher granularity parallelization yields a higher speedup.

Another factor that affects the efficiency of parallelization is the frequency of *synchronization* points in the procedure. From time to time the processors that have finished a certain task have to wait until all others finish the same task, in order to exchange some information needed for continuation. This may create a substantial amount of idle time, so the less frequent the synchronization, the more efficient the procedure will tend to be.

Last, but not least, the efficiency of parallelization depends, for a given algorithm, on the architecture of the computer used. Parallel computers may be classified into Single Instruction Multiple Data (SIMD) and Multiple Instructions Multiple Data (MIMD). Within the latter category, one may distinguish between Uniform Memory Access (UMA), Non-Uniform Memory Access (NUMA), and No Remote Memory Access (NORMA) computers. In UMA machines all memory access is uniform, i.e. access times are independent of memory location. NUMA machines distribute memory across processors, thus some access (local) takes less time than other (remote). The difference may be small or large, depending on the kind of connection used for remote memory access. NORMA machines allow no remote memory access, all interprocessor communication takes place via explicit message passing. Our implementation is designed primarily for a NUMA architecture where the ratio between remote and local access times is on the order of ten (the ratio for our computer). Larger access time ratios may require modifying some of the implementation details. The algorithm should port as described to most UMA architectures. The only architectural requirements are interprocessor communication via shared memory, atomic operations on selected memory locations, and processor synchronization. Implementation on NORMA architectures should also be possible, but the data placement strategy must be modified according to the capabilities of the machine.

For a discussion of concepts and issues related to parallel computing see [17,26,25,13]. In our approach, the cost matrix is evenly distributed across processor memories in contiguous blocks of rows. This partitioning scheme makes it possible to store quite large matrices. In view of the non-uniform memory access times, calculations are designed so that each processor works primarily with the rows stored in local memory. Each processor's local memory holds a row buffer. Whenever a processor needs access to a row stored in another processor's memory, it copies the row in question into its row buffer, which it then uses to access locally individual row elements one at a time, as needed.

The current assignment A^* is stored as a single predecessor /successor list.

The dual variables u_i and v_j are stored as follows. Every processor keeps a copy of each column variable v_j, but there is only one centrally stored copy of the row variables u_i. The reason for this is that the reduced costs $\bar{c}_{ij} = c_{ij} - u_i - v_j$ are calculated row-wise, i.e. u_i remains unchanged for an entire row while v_j changes with every $\bar{c}_{ij}$. This way there are no simultaneous memory access requests for the frequently used column variables, and only a small chance of simultaneous access requests for the much less frequently needed row variables. Besides the aspect of minimizing conflicting memory access requests, this storage scheme also capitalizes on the fact that local memory access is cheaper than remote access.

The specifics of the parallelization in each phase are discussed below.

4. The Initial Solution Phase

The calculation of the initial feasible solution to the dual of (AP) is done in parallel. First, each processor calculates the value of u_i as the minimum of c_{ij}, $j \varepsilon T$, for each of its own rows. Then each processor calculates, for each column $j \varepsilon T$, the minimum of $c_{ij} - u_i$ over its own set of rows, and one processor calculates v_j as the smallest of these partial minima. Finally, each processor calculates the reduced costs $c_{ij} - u_i - v_j$ for its own rows, and constructs its part of the admissible graph $\bar{G} := (\bar{S} \cup \bar{T}, \bar{A})$, where $\bar{A} := \{(i,j) \varepsilon A : c_{ij} - u_i - v_j = 0\}$

Next, a maximum matching is found in the admissible graph by a sequential algorithm. The time required for this is too small to justify parallelization, which could only be of relatively low granularity.

Next the sparse matrix structure is created, again in parallel, by having each processor find the k smallest costs in each of its rows. For problems with $n \geq 500$ we used $k = n/50$, and the solution obtained in this way was always dual feasible.

5 The Augmenting Path Finding Phase

At the end of Phase 0, q rows and q columns have been assigned to each other, for some positive integer $q \leq n$, and n-q rows and columns are unassigned. In principle there are two ways in which the search for augmenting paths can be parallelized: (i) have each processor choose a different unassigned node in S and apply the labeling technique in an attempt to find an augmenting path. (ii) have several processors jointly attempt to find an augmenting path from some unassigned node in S. In the first case, several augmenting paths are generated in parallel, but each path is found by a single sequential procedure. In the second case, each path is constructed in parallel. The second kind of parallelization is of lower granularity, so on most existing parallel computers the first kind ought to have priority. In fact, a preliminary investigation of the second kind of parallelization showed so little promise on our computer that we have so far not implemented it. (On other types of computers it may be advantageous to implement both (i) and (ii)). The rest of this discussion will concentrate on (i).

Suppose each processor uses the labeling technique to construct an alternating tree rooted at a different node, and as a result finds an augmenting path. Can all the paths found in this way be used simultaneously to augment the current partial assignment while leaving it optimal? If the alternating trees found by labeling are pairwise disjoint, then clearly performing the augmentation and the recalculation of the dual variables in parallel has the same effect as performing them sequentially, hence the procedure is legitimate. On the other hand, if two augmenting paths have some node in common, then one of them cannot be used for augmentation. Thus the paths used for augmentation have to be disjoint. The crucial question is, what happens if we have a collection of augmenting paths that are pairwise disjoint, but the corresponding alternating trees are not? In this case performing the augmentation and the recalculation of the dual variables in parallel may not have the same effect as doing them sequentially, and in the sequential case it may even seem doubtful whether using a first path for augmentation leaves the remaining paths "shortest" in terms of the modified reduced costs. The next Theorem dispels these doubts by showing that if the procedure for updating the dual variables is duly modified, then any set of pairwise disjoint augmenting paths can be used in parallel, even when their associated alternating trees collide.

Let A^* be a matching of the nodes of $\bar{S} \subseteq S$ to those of $\bar{T} \subseteq T$, and let $(u,v)\varepsilon R^{2n}$ satisfy

$$(8) \qquad u_i + v_j \begin{cases} \leq c_{ij} & (i,j)\varepsilon A \\ = c_{ij} & (i,j)\varepsilon A^*. \end{cases}$$

Condition (8) is necessary and sufficient for A^* to be a minimum-cost perfect matching in the subgraph induced by $\bar{S}\cup\bar{T}$.

For $h = 1, \ldots, m$, let P_h be an augmenting path with respect to A^* from node $i_h \varepsilon S\setminus\bar{S}$ to node $j_h \varepsilon T\setminus\bar{T}$, let LC_h be the set of columns labeled in the process of generating P_h, and for $j\varepsilon LC_h$, let λ_j^h be the label assigned column j in that process. Further, let $M := \{1, \ldots, m\}$, and let LC be the set of all columns labeled while generating the m augmenting paths P_h, $h\varepsilon M$, i.e. $LC := \bigcup_{h\varepsilon M} LC_h$.

Assume now that the augmenting paths P_h, $h\varepsilon M$, are pairwise node disjoint, and let A^{**} be the matching in G obtained by augmenting A^* along each of the paths P_h, $h\varepsilon M$, i.e.

$$A^{**} := A^{*}\oplus(\bigcup_{h\varepsilon M} P_h),$$

where $\oplus$ denotes the symmetric difference (for sets X, Y, $X\oplus Y = X\cup Y\setminus X\cap Y$). Then clearly A^{**} is a matching of the nodes of $\bar{S}\cup\{i_1,\ldots,i_m\}$ to those of $\bar{T}\cup\{j_1,\ldots,j_m\}$. Furthermore, A^{**} has the following property.

Theorem. For $j\varepsilon T$, let $i(j)$ be the row assigned to column j by A^{*}, and define

(9)
$$v_j^{*} := \begin{cases} v_j - \max\limits_{h:j\varepsilon LC_h} \{\lambda^h_{j_h} - \lambda^h_j\} & \text{if } j\varepsilon LC \\ v_j & \text{otherwise} \end{cases}$$

$$u_i^{*} = \begin{cases} u_i + \max\limits_{h:j\varepsilon LC_h} \{\lambda^h_{j_h} - \lambda^h_j\} & \text{if } i = i(j) \text{ for some } j\varepsilon LC \\ u_i + \lambda^h_{j_h} & \text{if } i = i_h \text{ for some } h\varepsilon M \\ u_i & \text{otherwise.} \end{cases}$$

Then

(10)
$$u_i^{*} + v_j^{*} \begin{cases} \le c_{ij} & \text{for } (i,j)\varepsilon A \\ = c_{ij} & \text{for } (i,j)\varepsilon A^{**}. \end{cases}$$

Proof. For $i,j)\varepsilon A$, we will denote by $\bar{c}_{ij}$ and c^{*}_{ij} the reduced costs associated with A^{*} and A^{**}, respectively, i.e. $\bar{c}_{ij} := c_{ij} - u_j - v_j$, $c^{*}_{ij} := c_{ij} - u^{*}_i - v^{*}_j$. Also, for each $h\varepsilon M$, we will denote by u^h_i and v^h_j the dual variables, and by $\bar{c}^h_{ij}$ the reduced costs (i.e. $\bar{c}^h_{ij} := c_{ij} - u^h_i - v^h_j$) that would be obtained if A^{*} were augmented by using *only* the path P_h. Since each P_h is a *shortest* augmenting path, it is well known that $\bar{c}^h_{ij} \ge 0$ for all $(i,j)\varepsilon A$ and all $h\varepsilon M$, and $\bar{c}_{ij} = 0$ for all $(i,j)\varepsilon A^{*}\oplus P_h$.

Our proof of $c^{*}_{ij} \ge 0$ will consist in showing for every $(i,j)\varepsilon A$, either that $c^{*}_{ij} \ge \bar{c}_{ij}$, or that $c^{*}_{ij} \ge c^h_{ij}$ for some $h\varepsilon M$.

For this purpose, we first state explicitly the values of u^h_i, and v^h_j. Recall that P_h connects $i_h\varepsilon S\setminus\bar{S}$ to $j_h\varepsilon T\setminus\bar{T}$ and that $i(j)$ is the row assigned by

A^* to column j. For $h \in M$, we have

$$v^h_j = \begin{cases} v_j - \lambda^h_{j_h} + \lambda^h_j & \text{if } j \in LC_h \\ v_j & \text{otherwise} \end{cases}$$

and

$$u^h_i = \begin{cases} u_i + \lambda^h_{j_h} - \lambda^h_j & \text{if } i = i(j) \text{ for some } j \in LC_h \\ u_i + \lambda^h_{j_h} & \text{if } i = i_h \\ u_i & \text{otherwise} \end{cases}$$

Note that, by construction, the labels satisfy $\lambda^h_{j_h} \geq \lambda^h_j$ for all $j \in LC_h$ and all $h \in M$.

We now examine c^*_{ij} for different positions of i and j. First if $i \in S \backslash \bar{S}$ and $i \neq i_h$ for all $h \in M$, then $u^*_i = u_i$ and

$$c^*_{ij} = \begin{cases} c_{ij} - u_i - v_j + \max\limits_{h: j \in LC_h} \{\lambda^h_{j_h} - \lambda^h_j\} & \text{if } j \in LC \\ c_{ij} - u_i - v_j & \text{otherwise} \end{cases}$$

hence $c^*_{ij} \geq \bar{c}_{ij} \geq 0$.

Next, if $i = i_h$ for some $h \in M$, then $u^*_i = u^h_i$ and

$$c^*_{ij} = \begin{cases} c_{ij} - u^h_i - v_j + \max\limits_{k: j \in LC_k} \{\lambda^k_{j_k} - \lambda^k_j\} & \text{if } j \in LC \\ c_{ij} - u^h_i - v^h_j & \text{otherwise} \end{cases}$$

Since $-v_j + \max\limits_{k: j \in LC_k} \{\lambda^k_{j_k} - \lambda^k_j\} \geq -v^h_j$ for all $j \in LC$, $c^*_{ij} \geq \bar{c}^h_{ij}$ follows.

Now let $i \in \bar{S}$, namely let $i = i(k)$ for some $k \in \bar{T}$. There are several cases to be considered.

Case 1. $k \in LC$, $j \in LC$. Then

$$c^*_{ij} = c_{ij} - u_i - \max_{h: k \in LC_h} \{\lambda^h_{j_h} - \lambda^h_k\} - v_j + \max_{h: j \in LC_h} \{\lambda^h_{j_h} - \lambda^h_j\}.$$

If $j = k$, then $c^*_{ij} = \bar{c}_{ij} \geq 0$. If $j \neq k$, let ℓ and m be the indices for which the two maxima in the above expression are attained. Then

$$c^*_{ij} = c_{ij} - u_i - \lambda^\ell_{j_\ell} + \lambda^\ell_k - v_j + \lambda^m_{j_m} - \lambda^m_j$$

$$= \begin{cases} \bar{c}^\ell_{ij} + (\lambda^m_{j_m} - \lambda^m_j) - (\lambda^\ell_{j_\ell} - \lambda^\ell_j) & \text{if } j\varepsilon LC_\ell \\ \bar{c}^\ell_{ij} + (\lambda^m_{j_m} - \lambda^m_j) & \text{if } j\varepsilon LC\setminus LC_\ell \end{cases}$$

and hence from the definition of m, $c^*_{ij} \geq \bar{c}^\ell_{ij}$.

Case 2. $k\varepsilon LC$, $j\varepsilon T\setminus LC$. Then

$$c^*_{ij} = c_{ij} - u_i - \max_{h:k\varepsilon LC_h} \{\lambda^h_{j_h} - \lambda^h_k\} - v_j \geq \bar{c}^\ell_{ij},$$

where ℓ is chosen as in Case 1.

Case 3. $k\varepsilon T\setminus LC$, $j\varepsilon LC$. Then

$$c^*_{ij} = c_{ij} - u_i - v_j + \max_{h:j\varepsilon LC_h} \{\lambda^h_{j_h} - \lambda^h_j\} \geq \bar{c}_{ij}.$$

Case 4. $k\varepsilon T\setminus LC$, $j\varepsilon T\setminus LC$. Then

$$c^*_{ij} = c_{ij} - u_i - v_j = \bar{c}_{ij}.$$

This completes the proof of $c^*_{ij} \geq 0$, $(i,j)\varepsilon A$.

To prove that $(i,j)\varepsilon A^{**}$ implies $c^*_{ij} = 0$, it is sufficient to point to the fact that since the paths P_h, $h\varepsilon M$, are pairwise disjoint, every $(i,j)\varepsilon A^{**}$ is contained in at most one path P_h. Hence for every $(i,j)\varepsilon A^{**}$, $c^*_{ij} = \bar{c}^h_{ij}$ for the particular $h\varepsilon M$ for which P_h contains (i,j), and hence $c^*_{ij} = 0$.□

Corollary. A^{**} is a minimum-cost perfect matching in the subgraph induced by $\bar{S}\cup\{i_1,\ldots,i_m\}\cup\bar{T}\cup\{j_1,\ldots,j_m\}$.

The parallel search for augmenting paths is implemented as follows. After Phase 0, all processors work simultaneously either on Phase 1 or on Phase 2 of the algorithm, the two alternating phases being separated by a synchronization stage. During every Phase 1, each processor chooses an unassigned node $i\varepsilon S$ and uses the labeling technique to grow an alternating tree (in the same way as in the sequential algorithm), until an unassigned node $j\varepsilon T$ is labeled; at which point a *potential* augmenting path from i to j has been identified. If this path is node disjoint from all the undiscarded paths found by any of the processors during the current phase, it is stored as an *actual* augmenting path, along with all the labels assigned in the process or finding it; otherwise it is discarded. In either case, the processor in question chooses another unassigned node in S to search for another augmenting path. The phase ends when the number of potential augmenting paths generated exceeds a certain fraction α of the total number of unassigned nodes. At that point all processors stop the search for augmenting paths (synchronization) and simultaneously start working on Phase 2. The value of α, determined experimentally, was chosen to be 1/3. This choice is intended to balance the advantages of longer Phase 1 runs against their disadvantages: the longer a Phase 1 run, the higher the probability that the newly found augmenting paths collide with earlier paths found during the given Phase 1 and have to be discarded; and the shorter a Phase 1 run, the higher the proportion of time lost by each processor in the last unfinished run (interrupted by the call for synchronization).

The procedure of checking each newly found path for collision with earlier paths as soon as it is found, is a heuristic which favors simplicity and ease of implementation over the benefits that could be gained from storing all potential augmenting paths until the synchronization point, and then selecting among them a maximum number of pairwise node disjoint ones to serve as actual augmenting paths.

During the labeling procedure, one has to repeatedly find the minimum of a set of labels. As the potential augmenting paths get longer and longer, this step becomes more and more expensive. To reduce its cost, we have implemented the data structure known as d-heap [27]. This saved considerable computational effort towards the end of the procedure, when the number of unassigned nodes is small and the potential augmenting paths are very long; but it did not justify itself in the earlier stages when the potential augmenting paths are shorter. As a result, our current implementation switches to the use of d-heaps only when the assignment is 98% complete.

6. The Updating Phase

For the updating of the dual variables, the changes $\Delta u_i := u_i^* - u_i$ and $\Delta v_j := v_j^* - v_j$ (see (9)) are stored centrally, initialized at zero at the beginning of Phase 1, and updated whenever a processor finds a new value that warrants a change. The new values are actually calculated during the labeling procedure of Phase 1, so that in fact Phase 2 consists simply in putting into effect the changes calculated during Phase 1.

To be specific, the updating is implemented as follows. At the start of Phase 1, Δu_i and Δv_j are set to 0 for all $i \in S$, $j \in T$. As the processor working on augmenting path P_h calculates the value $\lambda^h_{j_h} - \lambda^h_j$ for column j, if $\lambda^h_{j_h} - \lambda^h_j > -\Delta v_j$, then Δv_j is replaced by $-(\lambda^h_{j_h} - \lambda^h_j)$; and if i(j) is the row assigned to column j, $\Delta u_{i(j)}$ is replaced by $\lambda^h_{j_h} - \lambda^h_j$.

The actual changing of the dual variables in Phase 2 then consists of each processor replacing its own set of column variables v_j with v_j^*, $j \in T$, and of replacing the centrally stored row variables u_i with u_i^*.

As to the changing of the assignment, the operation $A^{**} = A^* \circledast (\bigcup_{h \in M} P_h)$, where M is the set of actual augmenting paths generated, is executed on the predecessor/successor list that stores the current assignment.

7. Computational Results

Our parallel shortest augmenting path algorithm was implemented in the programming language C on a 14 processor BBN Butterfly Plus computer, with 56 megabytes of shared memory. The Butterfly Plus is a non-uniform memory multiprocessor consisting of Motorola 68020/68881 processors accessing 4 megabytes of local memory each, and nonlocal (or remote) memory through a packet switched network. Remote memory access is channelled through a switch and is therefore slower than local memory access. The Butterfly does not allow simultaneous access to individual memory locations. When two or more requests are made for reading a memory location only one access is serviced. The other requests must be retried at a later time. A more complete description of the Butterfly architecture may be found in [25].

Table 1 contains a summary of the computational results obtained by solving fully dense assignment problems on graphs ranging in size from 1000 to 30,000 nodes. This is to our knowledge the first time that problems of this size were solved. The costs for these problems were drawn from a uniform distribution of the integers in the range [0,100], [0,1000], and [0,10000]. The statistics represent an average of three problems for each size and cost range. The column headings in Table 1 have the following meaning: $n = |S| = |T|$. The number of initial assignments is the number of assigned nodes at the end of Phase 0. Setup time is the time required to initialize the dual variables, construct the admissible graph $\bar{G}$, and derive the sparse cost matrix from the original one. Initial matching time and augmenting path time are the times used for those respective operations. Optimality check time is the time required to check that the optimal matching found using the sparse matrix is dual feasible on the complete cost matrix. Finally, total time is the execution time of the complete algorithm.

The data of Table 1 show that the setup time and augmenting path time together account for roughly 4/5 of the total time, with the augmenting path time alone taking up between 1/2 and 2/3 of total time, except for the smallest cost range. While the algorithm that finds an initial matching is run sequentially by a single processor, it typically requires 3-13% of total time, with the exception of the cost range [0,100], where there is a large number of optimal solutions and the initial matching turns out to be optimal in every case. As the data for the [0,1000] and [0,10000] cost ranges show, the initial matching routine determines an increasing fraction of the total number of assignments as the problem size becomes large relative to the cost range. The [0,100] cost range results represent a culmination of this effect, in that the shortest augmenting path procedure becomes in this case unnecessary for determining an optimal solution.

The problems of size 10000, 20000, and 30000 shown in the last segment of Table 1 were solved using a special version of the algorithm. This version generates the dense cost matrix a row at a time on each processor, determines a sparse row from each of these full rows, and discards the full rows so that the complete cost matrix does not have to be stored in memory at any one time. After an optimal solution is found on the sparse cost matrix, dual feasibility is verified on the complete cost matrix by having processors regenerate and examine complete rows one at a time. Thus the special version of the algorithm requires two complete cost matrix generations to guarantee optimality on the dense cost matrix. For the sake of comparability, the execution times reported in Table 1 do not include matrix generation times but do include the time required to determine the sparse cost matrix and check dual feasibility.

Table 1. Algorithm Performance with 14 Processors

Cost range [0,100]

n	Number of initial assignments	Setup time (sec)	Initial Matching time (sec)	Augmenting path time (sec)	Optimality check time (sec)	Total Time (sec)
1000	1000.0	1.33	0.68	-	-	2.01
2000	2000.0	4.05	1.47	-	-	5.52
3000	3000.0	8.37	2.29	-	-	10.66

Cost range [0,1000]

n	Number of initial assignments	Setup time (sec)	Initial Matching time (sec)	Augmenting path time (sec)	Optimality check time (sec)	Total Time (sec)
1000	864.0	1.98	0.30	6.86	0.25	9.39
2000	1856.3	6.49	1.21	14.59	0.90	23.20
3000	2944.0	14.51	5.02	19.01	2.04	40.59

Cost range [0,10000]

n	Number of initial assignments	Setup time (sec)	Initial Matching time (sec)	Augmenting path time (sec)	Optimality check time (sec)	Total Time (sec)
1000	811.7	1.99	0.29	9.17	0.25	11.70
2000	1644.0	6.51	1.00	21.69	0.89	30.09
3000	2472.7	14.62	2.13	40.33	2.03	59.11
10000	8674.0	80.95	22.66	175.06	37.12	315.79
20000	18569.0	282.25	84.99	717.46	148.51	1233.21
30000	29467.7	651.08	395.24	1665.61	324.47	3036.40

Table 2. Algorithm Performance on 1 Processor

Cost range [0,100]						
n	Number of initial assignments	Setup time (sec)	Initial Matching time (sec)	Augmenting path time (sec)	Optimality check time (sec)	Total Time (sec)
1000	1000.0	18.19	0.39	-	-	18.58
2000	2000.0	74.49	0.74	-	-	75.23
3000	3000.0	171.07	1.11	-	-	172.18

Cost range [0,1000]						
n	Number of initial assignments	Setup time (sec)	Initial Matching time (sec)	Augmenting path time (sec)	Optimality check time (sec)	Total Time (sec)
1000	864.0	25.72	0.28	10.20	3.67	39.87
2000	1856.3	108.20	1.09	32.45	14.30	156.05
3000	2944.0	258.50	3.77	29.56	37.61	329.45

Cost range [0,10000]						
n	Number of initial assignments	Setup time (sec)	Initial Matching time (sec)	Augmenting path time (sec)	Optimality check time (sec)	Total Time (sec)
1000	811.7	25.77	0.28	13.93	3.67	43.65
2000	1644.0	108.28	0.97	48.99	14.26	172.50
3000	2472.7	258.55	2.10	117.96	37.64	416.25

Table 3. Algorithm Speedup

Cost Range [0,100]				
n	Setup	Augmenting Path	Optimality Check	Global
1000	13.68	-	-	9.24
2000	18.39	-	-	13.62
3000	20.44	-	-	16.15

Cost Range [0,1000]				
n	Setup	Augmenting Path	Optimality Check	Global
1000	12.98	1.48	14.68	4.24
2000	16.67	2.22	15.89	6.72
3000	17.81	1.55	18.43	8.11

Cost Range [0,10000]				
n	Setup	Augmenting Path	Optimality Check	Global
1000	12.94	1.51	14.68	3.73
2000	16.63	2.26	16.02	5.73
3000	17.68	2.92	18.54	7.04

Table 2 contains a summary of the computational results obtained by solving the same fully dense assignment problems shown in Table 1 using a single processor. The performance data shown in Table 2 were obtained by distributing the complete cost matrix across all 14 processor memories but using only a single processor to execute the algorithm. Such a partitioning approach was necessary since the complete cost matrix could not be stored in any one processor memory.

Table 3 shows the speedup of the individual portions of the algorithm as well as the overall speedup. Speedup is defined here to be the sequential execution time shown in Table 2 divided by the parallel execution time shown in Table 1. As Table 3 shows, the portions of the algorithm accounted for by the setup times and by the optimality check times yield high speedups. In fact, for many entries of Table 3, the speedup actually exceeds the number of processors. These superlinear speedups represent a distortion mostly due to the matrix partitioning strategy described in the previous paragraph. A processor executing the algorithm sequentially must make a large number of nonlocal cost matrix accesses during the setup and optimality check phases. During the multiprocessor execution, processors always work out of local memory during the setup and optimality phases. Thus the sequential execution is penalized by the nonlocal memory accesses.

Table 3 also shows that the speedup for the augmenting path phase is substantially less than the number of processors. One reason for this is the need for synchronization. During any iteration of the algorithm, all processors must wait for the slowest processor before moving to the next phase of the algorithm, and the length of the interval between two consecutive synchronizations is limited by the need to keep disjoint the potential augmenting paths constructed. Furthermore, towards the end of the algorithm, when the number of unassigned nodes is less than the number of processors, some of the processors must lie idle. This phenomenon is exacerbated by the fact that the last few augmenting paths always take many times longer to find than the earlier ones.

Despite the lower speedup for the augmenting path phase, the overall algorithm speedup remains quite good. In fact, Table 3 indicates that both the augmenting path speedup and overall speedup increase with problem size for all three cost ranges.

Other parallel algorithms for the assignment problem have been proposed and/or implemented by Bertsekas [5] and Hatay [14], based on the auction method, and by Kennington and Wang [16], based on the shortest augmenting path method. Bertsekas' implementation is a simulated parallel algorithm run on a sequential VAX 11/750 on sparse assignment problems ranging in size from 500 to 2500 nodes. Speedups of about 10 were obtained for a number of virtual processors equal to problem size. Hatay has implemented the auction algorithm on a 20-processor Sequent Balance 21000. For fully dense problems of size 1600 and cost range [0,10000] speedups are reported to be about 4 and 7 for 5 and 10 processors, respectively, with a decrease for a larger number of processors. On the 20 processor computer, the fastest reported solution time (48 seconds) was obtained when using only 10 processors.

Kennington and Wang's implementation of the shortest augmenting path algorithm for an 8-processor Sequent Symmetry S81 is based on several processors simultaneously constructing an augmenting path. Tested on problems of size 800 to 1200, the algorithm obtains speedups between 2.73 and 7.64 for the cost range [0,100], between 2.24 and 5.63 for the cost range [0,1000], and between 2.46 and 4.89 for the cost range [0,10000]. Kennington and Wang ran extensive comparisons of the shortest augmenting path algorithm with the auction algorithm, and concluded that for dense assignment problems the former dominates the latter.

Finally, a parallel version of the primal simplex method for the transportation problem was implemented by Miller, Pekny and Thompson [23] and tested on the same 14-processor Butterfly Plus computer on which our algorithm was run. This code of course is meant for a more general problem, but it can be used to solve assignment problems. For assignment problems of size 3000 with costs in the ranges [0,1000] and [0,10000], the speedups are about 7, but the absolute times are about 500 and 900 seconds, respectively, for the two ranges.

8. Application to the Asymmetric Traveling Salesman Problem

One well known application of the assignment problem is its use in branch and bound algorithms to solve the traveling salesman problem on a directed graph, also called the asymmetric traveling salesman problem (ATSP) (see [1] for a survey). The assignment problem (AP) obtained from the standard integer programming formulation of the ATSP by removing the subtour elimination constraints is a relaxation of the ATSP whose strength is best illustrated by the fact that for randomly generated costs the value of an optimal assignment is within 1% of the value of an optimal tour for problems with 100 nodes, and this percentage decreases with problem size. As a result,applying branch and bound to the ATSP with random costs typically results in search trees of manageable size. Miller and Pekny [22,24] have implemented a parallel branch and bound algorithm for the ATSP, using the AP as a relaxation. Table 4 shows the effect of solving the AP at the root node of the search tree with the parallel algorithm described in this paper, as opposed to the corresponding sequential algorithm. The data of the table represent average times for three problems in each class, except for the largest size (n = 7500), for which only two problems were run. All problems up to n = 3000 were run on the 14-processor Butterfly Plus, while the problems for n = 5000 and n = 7500 were run on a 100-processor Butterfly Plus.

Table 4. Performance of Parallel Branch and Bound Algorithm for the Asymmetric Traveling Salesman Problem on 14 Processors

Cost Range [0,1000]		
n	Execution time (sec) with parallel assignment algorithm	Execution time (sec) with sequential assignment algorithm
1000	71.33	101.81
1500	137.62	213.21
2000	272.98	405.81
2500	585.09	784.45
3000	451.71	740.57

Cost Range [0,10000]		
n	Execution time (sec) with parallel assignment algorithm	Execution time (sec) with sequential assignment algorithm
1000	249.09	326.04
1500	302.31	379.97
2000	1017.05	1159.46
2500	777.19	1007.24
3000	1392.12	1749.26
5000*	1606.30	--
7500*	8809.40	--

*100-processor Butterfly Plus

References

1. E. Balas and P. Toth, "Branch and Bound Methods." In E. L. Lawler, et al. (editors), *The Traveling Salesman Problem: A Guided Tour to Combinatorial Optimization*. Wiley, 1985.

2. M. Balinski, "A Competitive (Dual) Simplex Method for the Assignment Problem," *Mathematical Programming, 34*, 1986, pp. 125-141.

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

4. D. P. Bertsekas, "A New Algorithm for the Assignment Problem," *Mathematical Programming, 21*, 1981, pp. 152-171.

5. D. P. Bertsekas, "A Distributed Asynchronous Relaxation Algorithm for the Assignment Problem," *Proceeding of the 24th IEEE Conference on Decision and Control*, 1987.

6. R. E. Burkard and U. Derigs, *Assignment and Matching Problems Solutions Methods with FORTRAM Programs*, Springer, Berlin, 1980.

7. G. Carpaneto and P. Toth, "Algorithm for the Solution of the Assignment Problem for Sparse Matrices," *Computing, 31*, 1983, pp. 83-94.

8. G. Carpaneto and P. Toth, "Primal-Dual Algorithms for the Assignment Problem," *Discrete Applied Mathematics, 18*, 1987, pp. 137-153.

9. U. Derigs, "The Shortest Augmenting Path Method for Solving Assignment Problems -- Motivation and Computational Experience," *Annals of Operations Research, 4*, 1985, pp. 57-102.

10. E. W. Dijkstra, "A Note on Two Problems in Connection With Graphs," *Numer. Math., 1*, 1959, pp. 269-271.

11. J. Edmonds and R. M. Karp, "Theoretical Improvements in Algorithmic Efficiency of Network Flow Problems," *Journal of the ACM, 19*, 1972, pp. 248-261.

12. L. R. Ford and D. R. Fulkerson, *Flow in Networks*, Princeton University Press, New York, 1962.

13. A. Gottlieb, "The NYU Ultracomputer-Designing an MIMD Shared-Memory Parallel Computer," *IEEE Transactions on Computers, C-32*, 1983, pp. 175-189.

14. L. Hatay, "Bipartite Matching: A Computational Investigation of Parallel Algorithms," Department of Operations Research, Southern Methodist University, 1987.

15. R. Jonkers and T. Volgenant, "Shortest Augmenting Path Algorithm for Dense and Sparse Linear Assignment Problems," *Computing, 38*, 1987 pp. 325-390.

16. J. Kennington and Z. Wang, "Solving Dense Assignment Problems on a Shared Memory Multiprocessor," Technical Report 88-OR-16, Department of Operations Research, Southern Methodist University, Dallas, October 1988.

17. G. A. P. Kindervater and J. K. Lenstra, "Parallel Computing in Combinatorial Optimization," *Annals of Operations Research, 14,* 1988.

18. N. W. Kuhn, "The Hungarian Method for the Assignment Problem," *Naval Research Logistics Quarterly, 2,* 1955, pp. 83-97.

19. E. L. Lawler, *Combinatorial Optimization: Networks and Matroids,* holt, Rinehard, and Winston, New York, 1976.

20. S. Martello and P. Toth, "Linear Assignment Problems," *Annals of Discrete Mathematics, 31,* 1987, pp. 259-282.

21. L. F. McGinnes, "Implementation and Testing of a Primal-Dual Algorithm for the Assignment Problem," *Operations Research, 31,* 1983, pp. 277-291.

22. D. L. Miller and J. F. Pekny, "Results From a Parallel Branch and Bound Algorithm for Solving Large Asymmetric Traveling Salesman Problems." To appear in *Operations Research Letters, 8,* 1989.

23. D. L. Miller, J. F. Pekny, and G. L. Thompson, "Solution of Large Dense Transportation Problems Using a Parallel Primal Method," MSRR No. 546, Carnegie Mellon University, 1988.

24. J. F. Pekny and D. L. Miller, "A Parallel Branch and Bound Algorithm for Solving Large Asymmetric Traveling Salesman Problems," Report 05-27-88, Engineering Design Research Center, Carnegie Mellon University, 1988.

25. R. Rettberg and R. Thomas, "Contention is No Obstacle to Shared-Memory Multiprocessing," *Communications of the ACM, 29,* 1986 pp. 1202-1212.

26. C. Ribeiro, "Parallel Computer Models and Combinatorial Algorithms," *Annals of Discrete Mathematics, 31,* 1987, pp. 325-364.

27. R. E. Tarjan, *Data Structures and Network Algorithms,* SIAM, 1983.

28. N. Tomizawa "On Some Techniques Useful for the Solution of Transportation Network Problems," *Networks, 1,* 1971, pp. 173-194.

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER MSRR-552 / 2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) A Parallel Shortest Path Algorithm for the Assignment Problem	5. TYPE OF REPORT & PERIOD COVERED Technical Report, April 1989
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Egon Balas, Donald Miller, Joseph Pekny, Paolo Toth	8. CONTRACT OR GRANT NUMBER(s) N00014-85-K-0198
9. PERFORMING ORGANIZATION NAME AND ADDRESS Graduate School of Industrial Administration Carnegie Mellon University Pittsburgh, PA 15213	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Personnel and Training Research Programs Office of Naval Research (Code 434) Arlington, VA 22217	12. REPORT DATE April 1989
	13. NUMBER OF PAGES 25
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)	15. SECURITY CLASS. (of this report)
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Parallel Processing
Assignment Problem
Shortest Augmenting Path Method
Synchronous Algorithms

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

We describe a parallel version of the shortest augmenting path algorithm for the assignment problem. While generating the initial dual solution and partial assignment in parallel does not require substantive changes in the sequential algorithm, using several augmenting paths in parallel does require a new dual variable recalculation method. The parallel algorithm was tested on a 14-processor Butterfly Plus computer, on problems with up to 900 million variables. The speedup obtained increases with problem size. The algorithm was also embedded into a parallel branch and bound procedure for the traveling

salesman problem on a directed graph, which was tested on the Butterfly Plus on problems involving up to 7,500 cities. To our knowledge, these are the largest assignment problems and traveling salesman problems solved so far.