A Parallel Shortest Augmenting Path Algorithm for the Assignment Problem

EGON BALAS

Carnegie Mellon University, Pittsburgh, Pennsylvania

DONALD MILLER

E. I. Du Pont de Nemours and Company, Inc.

JOSEPH PEKNY

Purdue University

AND

PAOLO TOTH

University of Bologna, Bologna, Italy

Abstract. A parallel version of the shortest augmenting path algorithm for the assignment problem is described. Although 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-bit 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 30,000 cities.

Categories and Subject Descriptors: C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures (Multiprocessors)—*parallel processors*; F.2.1 [**Analysis of Algorithms and Problem Complexity**]: Numerical Algorithms and Problems—*computation on matrices*; G.1.0 [**Numerical Analysis**]: General—*parallel algorithms*; G.1.3 [**Numerical Analysis**]: Numerical Linear Algebra—*sparse and very large systems*; G.1.6 [**Numerical Analysis**]: Optimization—*linear programming*; G.2.1 [**Discrete Mathematics**]: Combinatorics—*combinatorial algorithms*; G.2.2 [**Discrete Mathematics**]:

The research of E. Balas was supported by Grant ECS-8601660 of the National Science Foundation and Contract N00014-85-K-0198 with the Office of Naval Research; that of D. Miller and Joseph Pekny by the Engineering Design Research Center of Carnegie Mellon University; and that of Paolo Toth by NATO Research Grant No. 228-88.

Authors' addresses: E. Balas, Graduate School of Industrial Administration, Carnegie Mellon University, Pittsburgh, PA 15213; D. Miller, Central Research and Development, E. I. Du Pont de Nemours and Company, Inc.; J. Pekny, School of Chemical Engineering, Purdue University; P. Toth, DEIS, University of Bologna, Italy.

Graph Theory—*graph algorithms*; network problems; path and circuit problems; I.1.2 [**Algebraic Manipulation**]: Algorithms—*analysis of algorithms*

General Terms: Algorithms, Experimentation, Performance

Additional Key Words and Phrases Assignment, matching, shortest augmenting paths, traveling salesman problem

1. *Introduction*

The assignment problem can be thought of as the problem of assigning n people to n tasks (each requiring a single person) in such a way as to minimize the total cost of the assignments, where each assignment has a fixed cost. This is a special case of linear programming and also of the minimum-cost network flow problem and there are several methods for solving it in $O(n^3)$ time. It has many direct and indirect applications; among others, it is the most commonly used relaxation of the asymmetric traveling salesman problem, which is itself a frequently encountered sequencing model.

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}, $(i, j) \in A$, 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

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

subject to

$$\begin{aligned} \Sigma(x_{ij} : j \in T) &= 1 \qquad i \in S, \\ \Sigma(x_{ij} : i \in S) &= 1 \qquad j \in T, \end{aligned} \tag{2}$$

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

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

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

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

$$\max \Sigma(u_i : i \in S) + \Sigma(v_j : j \in T) \tag{5}$$

subject to

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

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

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

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

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 refer to elements of S (of T) either as nodes, or as rows (as columns).

Both the primal and dual simplex methods have their specialized versions for the assignment problem [2, 3, 22]. The most popular early approach, known as the *Hungarian method* [8, 12, 19–21], 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, 29], 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 a somewhat higher order of granularity than the other approaches. A different approach that easily lends itself to parallelization, is based on relaxation techniques [4].

Our paper represents one of two recent attempts (the other is due to Kennington and Wang [17]) to parallelize the shortest augmenting path method for the assignment problem (we have no knowledge of earlier attempts). In the Kennington/Wang procedure, several processors together construct an augmenting tree to find a shortest augmenting path which is then used to augment the current solution. In our procedure, each augmenting tree is constructed by a single processor, but all processors construct trees simultaneously and thus several augmenting paths are identified in parallel. For this to be possible, one needs a new method of recalculating the dual variables and the reduced costs when the augmenting trees constructed in parallel are not disjoint. This is a genuine, substantive change in the shortest augmenting path algorithm, which makes our parallel implementation a nontrivial generalization of the corresponding sequential algorithm. Also, the generalization seems to be worth the effort, since the speedup and, in general, the computational results attest to the efficiency of our approach.

The paper is organized as follows. Section 2 describes the sequential version of the algorithm that forms the object of our parallelization, while Section 3 discusses some basic concepts concerning parallel computing, like speedup, granularity, synchronization, and architecture, as they relate to the task addressed in the paper. These two sections, along with Section 1, provide the foundations for the developments to follow. The next three sections describe the parallelization of the various ingredients of the algorithm. Thus, Section 4 deals with the maximum bipartite matching procedure used to find a "good" starting solution, while Sections 5 and 6 deal with the search for augmenting paths and updating of the dual variables. In particular, Section 5 contains the essential modification of the sequential algorithm mentioned above: a closed form expression for recalculating the dual variables and the reduced costs after simultaneous augmentation of the solution along several shortest paths, when the paths themselves are disjoint but the augmenting trees used to find them are not.

Section 7 summarizes the computational results obtained by solving several classes of assignment problems, and compares our results to those obtained with other parallel algorithms for the assignment problem. Finally, Section 8 deals with the application of our parallel assignment algorithm to the solution of large asymmetric traveling salesman problems.

2. *The Sequential Algorithm*

The procedure starts by finding an initial solution (Phase 0), that is, 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. This is done 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 of 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
    $u_i := \min\{c_{ij} : j \in T\}$,    $i \in S$
    $v_j := \min\{c_{ij} - u_i : i \in S)$,    $j \in T$
    $A_0 := \{(i, j) \in A : c_{ij} - u_i - v_j = 0\}$
    find a maximum matching $\bar{A}$ in $G_0 := (S \cup T, A_0)$.
  end
  while $|\bar{A}| < n$ 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 $p_j$
        $\bar{A} := \bar{A} \cup \{(i, j)\} \setminus \{(i, p_j)\}$
        $j := p_j$
      end
    $\bar{A} := \bar{A} \cup \{(i_1, j)\}$
  end
```

When the problems to be solved are large, it pays to use sparse matrix techniques, that is, to restrict the search for a successor in the augmenting paths to those elements of each row of the reduced cost matrix smaller than a properly chosen threshold value, 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 the threshold value, 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*, that is, 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 interprocessor 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, that is, access times are independent of memory location. NUMA machines distribute memory across processors; thus, some access (local) takes less time than others (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 be portable 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 [13], [18], [26], and [27]. 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 nonuniform memory access times, calculations are designed so that each processor works primarily with 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 $\bar{A}$ 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, that is, 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}, $j \in T$ for each of its own rows. Then each processor calculates, for each column $j \in T$, 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 $G_o := (S \cup T, A_o)$, where $A_o := \{(i, j) \in A : c_{ij} - u_i - v_j = 0\}$.

Next, a maximum matching $\bar{A}$ is found in the admissible graph by a parallel bipartite matching algorithm. This parallel algorithm alternates between two phases: (i) each processor chooses a different unassigned node (row) in S and attempts to find an augmenting path in the admissible graph; (ii) a set of disjoint augmenting paths is chosen from among those found in (i). The set of disjoint paths is used to increase the cardinality of the initial matching. Those rows not belonging to the set of disjoint paths are still considered unassigned and are subject to further processing in phase (i). Those rows for which no augmenting path is found are not processed further in the initial matching phase. Note that processors must synchronize after phase (i) and (ii) so that the correctness of computations is guaranteed. The initial matching algorithm terminates with matched node sets $\bar{S} \subseteq S$ and $\bar{T} \subseteq T$ when the matching in the admissible graph is maximum.

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 overlap.

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

$$u_i + v_j \begin{cases} \leq c_{ij} & (i, j) \in A, \\ = c_{ij} & (i, j) \in \bar{A}. \end{cases} \tag{8}$$

Condition (8) is necessary and sufficient for $\bar{A}$ 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 $\bar{A}$ from node $i_h \in S \setminus \bar{S}$ to node $j_h \in T \setminus \bar{T}$, let LC_h be the set of columns labeled in the process of generating P_h, and for $j \in LC_h$, 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, $h \in M$, that is, $LC := \bigcup_{h \in M} LC_h$.

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

$$A^* := \bar{A} \oplus \left(\bigcup_{h \in M} P_h \right),$$

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

$$v_j^* := \begin{cases} v_j - \max\limits_{h: j \in LC_h} \{\lambda_{j_h}^h - \lambda_j^h\} & \text{if } j \in LC, \\ v_j & \text{otherwise}; \end{cases}$$

$$u_i^* = \begin{cases} u_i + \max\limits_{h: j \in LC_h} \{\lambda_{j_h}^h - \lambda_j^h\} & \text{if } i = i(j) \text{ for some } j \in LC, \\ u_i + \lambda_{j_h}^h & \text{if } i = i_h \text{ for some } h \in M, \\ u_i & \text{otherwise}. \end{cases} \tag{9}$$

Then

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

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

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

For this purpose, we first state explicitly the values of u_i^h, and v_j^h. Recall that P_h connects $i_h \in S \setminus \bar{S}$ to $j_h \in T \setminus \bar{T}$ and that $i(j)$ is the row assigned by $\bar{A}$

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

$$v_j^h = \begin{cases} v_j - \lambda_{j_h}^h + \lambda_j^h & \text{if } \; j \in LC_h, \\ v_j & \text{otherwise}, \end{cases}$$

and

$$u_i^h = \begin{cases} u_i + \lambda_{j_h}^h - \lambda_j^h & \text{if } \; i = i(j) \;\; \text{for some } \; j \in LC_h, \\ u_i + \lambda_{j_h}^h & \text{if } \; i = i_h, \\ u_i & \text{otherwise}. \end{cases}$$

Note that, by construction, the labels satisfy $\lambda_{j_h}^h \geq \lambda_j^h$ 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 $i \in S \setminus \bar{S}$ and $i \neq i_h$ for all $h \in M$, then $u_i^* = u_i$ and

$$c_{ij}^* = \begin{cases} c_{ij} - u_i - v_j + \max\limits_{h.j \in LC_h} \left\{ \lambda_{j_h}^h - \lambda_j^h \right\} & \text{if } \; j \in LC, \\ c_{ij} - u_i - v_j & \text{otherwise}, \end{cases}$$

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

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

$$c_{ij}^* = \begin{cases} c_{ij} - u_i^h - v_j + \max\limits_{k : j \in LC_k} \left\{ \lambda_{j_k}^k - \lambda_j^k \right\} & \text{if } \; j \in LC, \\ c_{ij} - u_i^h - v_j^h & \text{otherwise}. \end{cases}$$

Since

$$-v_j + \max_{k : j \in LC_k} \left\{ \lambda_{j_k}^k - \lambda_j^k \right\} \geq -v_j^h$$

for all $j \in LC$, $c_{ij}^* \geq \bar{c}_{ij}^h$ 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

$$c_{ij}^* = c_{ij} - u_i - \max_{h : k \in LC_h} \left\{ \lambda_{j_h}^h - \lambda_k^h \right\} - v_j + \max_{h : j \in LC_h} \left\{ \lambda_{j_h}^h - \lambda_j^h \right\}.$$

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

$$\begin{aligned} c_{ij}^* &= c_{ij} - u_i - \lambda_{j_\ell}^\ell + \lambda_k^\ell - v_j + \lambda_{j_m}^m - \lambda_j^m \\ &= \begin{cases} \bar{c}_{ij}^\ell + \left(\lambda_{j_m}^m - \lambda_j^m \right) - \left(\lambda_{j_\ell}^\ell - \lambda_j^\ell \right) & \text{if } \; j \in LC_\ell, \\ \bar{c}_{ij}^\ell + \left(\lambda_{j_m}^m - \lambda_j^m \right) & \text{if } \; j \in LC \setminus LC_\ell, \end{cases} \end{aligned}$$

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

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

$$c_{ij}^* = c_{ij} - u_i - \max_{h : k \in LC_h} \left\{ \lambda_{j_h}^h - \lambda_k^h \right\} - v_j \geq \bar{c}_{ij}^\ell,$$

where ℓ is chosen as in Case 1.

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

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

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

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

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

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

Corollary 1. *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 synchronization points. During every Phase 1, each processor chooses an unassigned node $i \in S$ and uses the labeling technique to grow an alternating tree (in the same way as in the sequential algorithm), until an unassigned node $j \in T$ 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 of 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 parameter α whose value is a function of (i) the number of unassigned nodes, (ii) the number of processors, and (iii) the number of actual augmenting paths found during the last k iterations. At that point, all processors stop the search for augmenting paths (synchronization) and simultaneously start working on Phase 2. This choice of the function α 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 that 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* [28]. This saved considerable computational effort, particularly towards the end of the procedure, when the number of unassigned nodes is small and the potential augmenting paths are very long.

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 potential augmenting path P_h calculates the value $\lambda_{j_h}^h - \lambda_j^h$ for column j, if P_h is selected as an actual augmenting path and $\lambda_{j_h}^h - \lambda_j^h > -\Delta v_j$, then Δv_j is replaced by $-(\lambda_{j_h}^h - \lambda_j^h)$; and if $i(j)$ is the row assigned to column j, $\Delta u_{i(j)}$ is replaced by $\lambda_{j_h}^h - \lambda_j^h$.

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 $A^* = \bar{A} \oplus (\bigcup_{h \in M} P_h)$, 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 C programming language on a 14 processor BBN Butterfly GP1000 computer, with 56 megabytes of shared memory. The Butterfly GP1000 is a nonuniform 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 channeled 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. Each Motorola 68020 processor is capable of executing 2 million instructions per second. A more complete description of the Butterfly architecture may be found in [26].

Tables I through V contain statistics describing algorithm performance for a variety of cost matrix structures and computer configurations. The algorithm shows very little ($< 5\%$) run-to-run variation; therefore, all the statistics reported in the tables represent solution of a single problem. The column headings in the tables have the following meaning: $n = |S| = |T|$. The number of initial assignments is the number of assigned nodes (of S) at the end of Phase 0. Setup time is the time required to initialize the dual variables and construct the admissible graph G_0. Initial matching time and augmenting path time are the times used for those respective operations. Finally, total time is the execution time of the complete algorithm. The difference between the total time and the sum of the other times represents the time required to dynamically manage working memory, initialize working vectors, and compute and verify the optimality of the final solution for the original cost matrix (see below).

Table I contains a summary of the computational results obtained by solving fully dense assignment problems on graphs ranging in size from 10,000 to 30,000 nodes. The costs for these problems were drawn from a uniform distribution of the integers in the range [0, 1000], [0, 10000], and [0, 100000]. The fully dense problems were solved using a special version of the algorithm. This version generates the dense cost matrix a row at a time, determines a sparse row from each full row by keeping elements of value no greater than a threshold λ (sparsification), and removes the full rows so that the complete dense cost matrix never resides in memory. The times reported in Table I do not include the initial dense matrix sparsification time since we coded matrix generation and sparsification as a seamless operation due to memory limitations. However, sparsification is an $O(n^2)$ operation that is trivially parallel and yields linear speedup. Extrapolation from smaller problems, whose fully dense matrices could be completely stored in primary memory, show that assuming the use of 14 processors, accounting for initial matrix sparsification time would add about 30, 119, and 267 seconds to the total execution times listed in Table I for size 10,000, 20,000, and 30,000, respectively. Since these initial sparsification times fall linearly with the number of processors, they may be made vanishingly small for a large enough number of processors.

After a tentative optimal solution is found on the sparse cost matrix, dual feasibility is verified on the complete cost matrix. A fast, but overly conservative, method of verifying dual feasibility on the complete matrix simply requires checking that $\lambda + 1 - v_{\max} - u_i$ is not less than zero for each row i in the cost matrix, where $v_{\max}$ is the largest column dual variable and u_i is the row dual variable. We use this criterion which is a sufficient but not necessary condition for the optimality of the solution, since otherwise we would have to use an $O(n^2)$ procedure that explicitly checked all the elements in the fully dense matrix. If the tentative optimal solution violates the above condition, λ is doubled. Next, for every row i for which the condition is violated, the actual reduced costs are calculated and if any of them is negative, the assignment is broken and the row dual variable u_i is decreased by the absolute value of the most negative reduced cost. Then a new tentative optimal solution is determined using the now denser sparse matrix. If necessary, the algorithm will increase λ until the sparse matrix encompasses the fully dense matrix. When the tentative optimal solution is dual feasible, then it is optimal on the fully dense cost matrix. Note that no cost matrix elements larger than the final threshold value λ_f affect the optimal solution, so that solving fully dense randomly generated (from a uniform distribution) assignment problems can be considered no harder than solving problems of λ_f/R density, where R is the range of the cost matrix elements. Table I lists the initial values of λ that were used to generate the results for the fully dense problems. The values of λ shown in Table I, which never had to be increased, were estimated from solving small fully dense randomly generated assignment problems ($n = 1000$).

The data of Table I show that the initial solution phase (Phase 0) determines an increasing fraction of assignments as problem size increases relative to the cost range. The [0, 1000] cost range results represent a culmination of this effect in that the shortest augmenting path procedure becomes unnecessary. For the [0, 10000] cost range, the contribution of the initial matching algorithm

TABLE I. Fully Dense Randomly Generated Assignment Problems (14 Processors)

		Cost Range [0, 1000]			
n	Number of Initial Assignments	Setup Time (sec)	Initial Matching Time (sec)	Augmenting Path Time (sec)	Total Time (sec)
10000	10000	3.30	9.90	—	15.4
20000	20000	6.26	20.30	—	31.0
30000	30000	9.42	34.30	—	50.6

		Cost Range [0, 10000]			
n	Number of Initial Assignments	Setup Time (sec)	Initial matching Time (sec)	Augmenting Path Time (sec)	Total Time (sec)
10000	8648	3.02	4.82	153.4	163.6
20000	18546	5.82	16.10	429.9	456.3
30000	29477	8.38	85.10	286.9	387.1

		Cost Range [0, 10000]			
n	Number of Initial Assignments	Setup Time (sec)	Initial Matching Time (sec) (sec)	Augmenting Path Time (sec)	Total Time (sec)
10000	8115	3.07	4.51	191.7	201.6
20000	16358	5.65	13.50	556.7	580.4
30000	24773	8.48	27.10	769.2	811.6

	Initial Value of λ (see text) Cost Range		
n	[0, 1000]	[0, 10000]	[0, 100000]
10000	5	50	500
20000	3	25	250
30000	2	17	166

grows with problem size, as it determines 86.5, 92.7, and 98.3 percent of the total assignments for the 10,000, 20,000, 30,000 node problems, respectively. For the [0, 100000] cost range, the contribution of the initial solution phase remains relatively constant with problem size, determining between 81 to 83 percent of the total assignments. This prominent role of the initial solution phase is partly due to the efficiency of the procedure used (maximum bipartite matching) in generating good solutions. When the contribution of the initial solution phase is at a minimum, the augmenting path time accounts for about 95 percent of the total time. The data of Table I illustrate the complications involved with testing assignment problem algorithms using fully dense randomly generated problems. Computational performance strongly depends on the ratio of cost range to problem size. As Table I shows, very small cost range problems are no more difficult to solve than unweighted bipartite matching problems.

Table II contains a summary of computational results obtained by solving randomly generated assignment problems on sparse graphs ranging in size from 10,000 to 50,000 nodes using both 1 and 14 processors. The density is the probability that an arc of the graph is present. Each arc cost was drawn from a uniform distribution of the integers in the range [0, 50]. The densities listed in Table II tend to place about 50 elements in each cost matrix row. Table II

TABLE II. SPARSE RANDOMLY GENERATED ASSIGNMENT PROBLEMS

	14 Processors					
n	Matrix Density (%)	Number of Initial Assignments	Setup Time (sec)	Initial Matching Time (sec)	Augmenting Path Time (sec)	Total Time (sec)
10000	0.5	8668	3.07	4.89	135.0	145.2
20000	0 25	17364	5.92	13.8	396.8	421.0
30000	0.17	26104	8.23	28.1	664.2	707.4
40000	0 125	34712	11 1	47.6	1050.1	1117.9
50000	0.10	43391	17.0	73.1	1968.1	2069.5

	1 Processor					
n	Matrix Density (%)	Number of Initial Assignments	Setup Time (sec)	Initial Matching Time (sec)	Augmenting Path Time (sec)	Total Time (sec)
10000	0.5	8668	24.2	21.3	423.0	470.1
20000	0.25	17364	48.6	83.7	1342.1	1477 3
30000	0.17	26104	73.1	189.3	2296.6	2563.6
40000	0.125	34712	99.6	336 6	5268.6	5711.0
50000	0.10	43391	126.6	531.9	10496.7	11162.8

	Speedup			
n	Setup	Initial Matching	Augmenting Path	Total
10000	7.88	4.36	3.13	3.24
20000	8.21	6.07	3.18	3.51
30000	8.88	6.74	3.46	3.62
40000	8.97	7.07	5.02	5 11
50000	7.45	7.27	5.33	5.39

shows that this problem generation strategy kept the fraction of total assignments found by the initial solution phase constant at about 87 percent, which is roughly equivalent to the fraction found when solving fully dense problems of size n and cost range $[0, n]$. The single processor data shown in Table II were obtained by distributing the cost matrix across all 14 processor memories while using only a single processor to execute the algorithm. Such a partitioning approach was necessary since the sparse cost matrices could not be stored in one processor memory.

Table II lists the speedup of the individual portions of the algorithm as well as overall speedup. Speedup is defined here to be the sequential execution time divided by the parallel execution time. As Table II shows, the portion of the algorithm accounted for by the setup time and the initial matching time yields high speedup. The speedup for the augmenting path phase is substantially less. 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 the augmenting paths disjoint. Furthermore, towards the end of the algorithm, when the number of unassigned nodes is small, the length of the augmenting paths tends to increase, which leads to a higher probability of path collision. Finally, when the number of unassigned nodes is less than the number of processors, some of the processors must lie idle. These last two phenomena are compounded by the fact that the last few augmenting paths always take many times longer to find than the earlier ones, even in a sequential algorithm.

Table III shows the time (in milliseconds) needed by the shortest augmenting path procedure to generate one new assignment for the sparse 10,000 node problem of Table II, under the 1 processor and the 14 processor execution mode. The starting number of assignments (8668, i.e., 86.68% of the total) corresponds to the situation at the end of the initial matching phase. Clearly, up to the point where 99% of the assignments are in place, the speedup is much better than the corresponding number of Table II would suggest. However, the speedup deteriorates dramatically towards the end of the procedure, when the last 1%, and especially the last 0.5% of the assignments are found. This suggests that a hybrid parallel shortest augmenting path algorithm may result in better speedup. Such a hybrid algorithm could use processors to simultaneously determine multiple assignments during the early part of execution just as our current algorithm does. As processor utilization starts to decline, the hybrid algorithm would switch to determining single assignments using several processors. The low granularity involved in determining a single assignment prevented us from efficiently implementing such a hybrid algorithm on the BBN Butterfly GP1000.

Table IV lists the results obtained by our parallel algorithm in solving assignment problems with special cost structure. The sparse random symmetric cost matrices were generated by drawing each entry from a uniform distribution of integers in the range $[0, 50]$. Algorithm performance for the random symmetric cost matrices is comparable to that for random asymmetric cost matrices.

The "difficult" cost matrix problems of Table IV were obtained by drawing each element c_{ij} (of the fully dense matrix) from a uniform distribution of integers in the range $[0, i \times j]$, where i and j are the row and column indices, respectively. The initial values of λ used to solve these problems were 2,500, 50,000, 100,000 and 100,000 for problem sizes of 1,000, 5,000, 10,000 and 20,000, respectively. These values of λ never had to be increased. We were unable to solve problems with "difficult" cost matrices larger than 20,000 nodes due to the memory limitations of the 14 processor BBN Butterfly. As Table IV shows, problems having "difficult" cost matrices require longer solution times than problems with randomly generated asymmetric or symmetric cost matrices. However, overall speedup for these problems is only slightly worse than that reported in Table II for problems of the same size. Speedup for setup time deteriorates with problem size, because for any fixed λ this class of problems yields cost matrices that are much denser at the top than at the bottom, and so the workload is unevenly assigned among processors. (This could of course be corrected by changing the workload distribution for this class of problems).

Other parallel algorithms for the assignment problem have been proposed. In particular, Bertsekas and Castanon [5] have reported on the performance of various parallel synchronous and asynchronous implementations of the auction algorithm using an Encore Multimax. Speedups ranging from about 3 for a synchronous implementation to about 7 for an asynchronous implementation were obtained using a maximum of 16 processors on a randomly generated sparse assignment problem of size 1,000, 20 percent density, and cost range $[1, 1000]$. The fastest reported execution time for this problem was about 7 seconds using 16 processors.

TABLE III. SPEEDUP AT DIFFERENT STAGES OF THE AUGMENTING PATH PHASE (10,000)

Assignments (from-to)	Time (ms) per assignment: 1 Processor	Time (ms) per assignment: 14 Processors	Speedup
8668-8910	73.43	10.99	6.68
8911-9140	87.35	8.78	9 95
9141-9325	105.21	10.16	10.36
9326-9535	147.24	16.19	9.09
9536-9741	212.00	25.83	8.21
9742-9904	598.64	80 68	7.42
9905-9957	1175.20	328.30	3.58
9958-9999	3174.52	2136.67	1.49

TABLE IV SPECIALLY STRUCTURED ASSIGNMENT PROBLEMS

Sparse Random Symmetric Cost Matrices (14 processors)

n	Matrix Density (%)	Number of Initial Assignments	Setup Time (sec)	Initial Matching Time (sec)	Augmenting Path Time (sec)	Total Time (sec)
10000	0.50	8639	3.16	4.97	134.4	144.8
20000	0.25	17334	6.02	13.7	384.7	409.0
30000	0.17	26013	8.97	27.4	715.8	759 0

"Difficult" Cost Matrices (14 processors)

n	Number of Initial Assignments	Setup Time (sec)	Initial Matching Time (sec)	Augmenting Path Time (sec)	Total Time (sec)
1000	551	0.55	0 97	18.9	20.6
5000	2359	13 6	4.07	399.0	417.9
10000	4631	48.7	10.9	880.9	942.9
20000	8822	88.0	38.1	3438.9	3570.0

"Difficult" Cost Matrices (1 processor)

n	Number of Initial Assignments	Setup Time (sec)	Initial Matching Time (sec)	Augmenting Path Time (sec)	Total Time (sec)
1000	551	2.32	0.52	41.7	44.6
5000	2359	44.5	13.6	1214 8	1273.8
10000	4631	76.8	55.8	2616 8	2751.0

Speedup

n	Setup	Initial Matching	Augmenting Path	Total
1000	4.21	1.87	2.21	2.17
5000	3.27	3 35	3 04	3.05
10000	1.58	5.12	2.97	2 92

Hatay [14] has implemented the auction algorithm on a 20 processor Sequent Balance 21000. For fully dense problems of size 800 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 using 10 processors.

Kennington and Wang [17] have reported on a parallel implementation of the Jonkers and Volgenant shortest augmenting path algorithm [15] for fully dense randomly generated assignment problems using an 8-processor Sequent Symmetry S81. Their algorithm 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 ranges of [0, 1000], and between 2.46 and 4.89 for the cost range [0, 10000]. The fastest reported 8 processor execution times for problems of size 1,000 and cost range [0, 100], [0, 1000], [0, 10000], [0, 1000000] were 7.48, 12.10, 16.50, and 19.03 seconds, respectively.

Kempa et al. [16] exploited both the multiprocessor and vector-processing capabilities of an 8-processor Alliant FX computer with various synchronous implementations of the auction algorithm. They tested their implementations on fully dense randomly generated assignment problems of size ranging from 100 to 4000 and cost ranges of [0, 100], [0, 1000], and [0, 10000], and obtained speedups ranging from 2.8 to 10.8 depending on problem size and cost range. Speedups exceed 8 since they compared vectorized multiprocessor execution times against unvectorized single processor execution times. Parallel execution times for the largest problems solved were 6.75 seconds for n = 4,000 and cost range [0, 100], 255.6 seconds for n = 2,000 and cost range [0, 1000], and 32.5 seconds for n = 2,000 and cost range [0, 10000] using 8 vector processors.

Finally, a parallel version of the primal simplex method for the transportation problem was implemented by Miller et al [24] and tested on the same 14-processor Butterfly GP1000 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.

We conclude that the parallel algorithm discussed in our paper seems to perform remarkably well on assignment problems with various cost structures. This is only partly due to efficient parallelization; part of the success should be attributed to the efficiency of the sequential algorithm itself that was parallelized. This sequential shortest path algorithm differs from the standard version in several respects, the most important one being the use of a maximum cardinality matching algorithm to find a high-quality initial solution in the admissible graph G_0. Other factors, like the use of d-heaps for finding the minimum column label etc., also contribute to making the sequential algorithm a very fast one. In order to give the reader an idea of the efficiency of the sequential version of our shortest augmenting path algorithm, we show in Table V the outcome of running the sequential algorithm on a Sun Sparcstation 330 workstation with 32 megabytes of memory. The column "matrix conversion time" refers to converting the fully dense cost matrix into a sparse one. The values of λ shown (chosen here to be independent of n for the sake of simplicity) never had to be changed.

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

TABLE V. FULLY DENSE RANDOMLY GENERATED ASSIGNMENT PROBLEMS (SPARCSTATION 330)

Cost Range [0, 100]

n	Number of Initial Assignments	Matrix Conversion Time (sec)	Setup Time (sec)	Initial Matching Time (sec)	Augmenting Path Time (sec)	Total Time (sec)
1000	1000	0.52	0.06	0.09	—	0.7
2000	2000	2.11	0.25	0.18	—	2.6
4000	4000	10.6	1.14	0.60	—	12 5

Cost Range [0, 1000]

n	Number of Initial Assignments	Matrix Conversion Time (sec)	Setup Time (sec)	Initial Matching Time (sec)	Augmenting Path Time (sec)	Total Time (sec)
1000	872	0.69	0.27	0.11	2.5	3.6
2000	1871	2 80	1.03	0.49	10.8	15.2
4000	3981	13.6	4.69	2.03	10.4	30.8

Cost Range [0, 10000]

n	Number of Initial Assignments	Matrix Conversion Time (sec)	Setup Time (sec)	Initial Matching Time (sec)	Augmenting Path Time (sec)	Total Time (sec)
1000	816	0.58	0.13	0.12	2.50	3.30
2000	1645	2.42	0.50	0.47	11.0	14.5
4000	3325	11.41	2 68	1.80	38.2	54.2

Initial Value of λ (see text)

n	Cost Range [0, 100]	[0, 1000]	[0, 10000]
1000	1	100	500
2000	1	100	500
4000	1	100	500

survey). The assignment problem (AP) is the relaxation of the ATSP obtained from the standard integer programming formulation of the latter by removing the subtour elimination constraints. The strength of this relaxation is best illustrated by the fact that for randomly generated costs (from a uniform distribution) 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 [23, 25] have implemented a parallel branch and bound algorithm for the ATSP, using the AP as a relaxation. Table VI 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. All results are for random graphs of the densities shown in the table. The entries of each cost matrix were randomly drawn from a uniform distribution of the integers in the range [0, 50]. The time required to solve the assignment problem at the root node of the search tree is about 70% of the total time when the AP is solved by

TABLE VI. Sparse Randomly Generated ATSPs

n	Matrix Density (%)	Execution Time (sec) with Parallel Assignment Algorithm	Execution Time (sec) with Sequential Assignment Algorithm
10000	0.5	363.9	688.8
20000	0.25	1766.8	2823.1
30000	0.17	1432.6	3288.8

a sequential algorithm, but only about 30% of the total time when the AP is solved with our parallel algorithm.

REFERENCES

1. Balas, E. and Toth, P. Branch and bound methods. In E. L. Lawler, et al., *The Traveling Salesman Problem: A Guided Tour to Combinatorial Optimization*. Wiley, New York, 1985.
2. Balinski, M. A competitive (dual) simplex method for the assignment problem. *Math. Prog. 34* (1986), 125–141.
3. Barr, R. S., Glover, F., and Klingman, D. The alternating basis algorithm for assignment problems. *Math. Prog. 13* (1977), 1–13.
4. Bertsekas, D. P. A new algorithm for the assignment problem. *Math. Prog. 21* (1981), 152–171.
5. Bertsekas, D. P., and Castanon, D. A. Parallel synchronous and asynchronous implementations of the auction algorithm. *Electrical Engineering and Computer Science*. MIT, Cambridge, Mass., 1989.
6. Burkard, R. E. and Derigs, U. Assignment and Matching Problems: Solutions Methods with FOTRAN Programs. Springer, Berlin, 1980.
7. Carpaneto, G., and Toth, P. Algorithm for the solution of the assignment problem for sparse matrices. *Comp. 31* (1983), 83–94.
8. Carpaneto, G., and Toth, P. Primal-dual algorithms for the assignment problem. *Disc. App. Math. 18*, (1987), 137–153.
9. Derigs, U. The shortest augmenting path method for solving assignment problems—motivation and computational experience. *Ann. Op. Res. 4* (1985), 57–102.
10. Dijkstra, E. W. A note on two problems in connection with graphs. *Numer. Math. 1* (1959), 269–271.
11. Edmonds, J. and Karp, R. M. Theoretical improvements in algorithmic efficiency of network flow problems, *J. ACM 19* (1972), 248–261.
12. Ford, L. R., and Fulkerson, D. R. *Flow in Networks*. Princeton University Press, Princeton, N.J., 1962.
13. Gottlieb, A. The NYU ultracomputer-designing an MIMD shared-memory parallel computer. *IEEE Trans. Comp. C-32* (1983), 175–189.
14. Hatay, L. Bipartite matching: A computational investigation of parallel algorithms. Department of Operations Research, Southern Methodist University, Dallas, Tex., 1987.
15. Jonkers, R., Volgenant, T. Shortest augmenting path algorithm for dense and sparse linear assignment problems. *Comp. 38* (1987), 325–390.
16. Kempa, D., Kennington, J., and Zaki, H. Performance characteristics of the Jacobi and Gauss-Seidel versions of the auction algorithm on the Alliant FX18. Report OR-89-008, Mechanical and Industrial Engineering, University of Illinois at Urbana-Champaign, Urbana, Ill., 1989.
17. Kennington, J. and Wang, Z. Solving dense assignment problems on a shared memory multiprocessor, Technical Report 88-OR-16, Department of Operations Research, Southern Methodist University, Dallas, Tex., October 1988.
18. Kindervater, G. A. P., and Lenstra, J. K. Parallel computing in combinatorial optimization, *Ann. Op. Res. 14* (1988).
19. Kuhn, N. W. The Hungarian method for the assignment problem. *Nav. Res. Log. Quart. 2* (1955), 83–97.
20. Lawler, E. L. *Combinatorial Optimization: Networks and Matroids*. Holt, Rinehart, and Winston, New York, 1976.

21 MARTELLO, S. AND TOTH, P. Linear assignment problems. *Ann. Dis. Math. 31* (1987), 259–282.
22. McGINNES, L. F. Implementation and testing of a primal-dual algorithm for the assignment problem. *Op. Res. 31* (1983), 277–291.
23. MILLER, D. L. AND PEKNY, J. F. Results from a parallel branch and bound algorithm for solving large asymmetric traveling salesman problems. *Op. Res. Lett. 8* (1989), 129–135.
24 MILLER, D. L., PEKNY, J. F., AND THOMPSON, G. L. Solution of large dense transportation problems using a parallel primal method, MSRR No. 546. Carnegie Mellon University, Pittsburgh, Pa., 1988.
25. PEKNY, J. F., AND MILLER, D. L. A parallel branch and bound algorithm for solving large asymmetric traveling salesman problems. Report 05-27-88, Engineering Design Research Center, Carnegie Mellon University, Pittsburgh, Pa., 1988.
26. RETTBERG, R. AND THOMAS, R. Contention is no obstacle to shared-memory multiprocessing. *Commun. ACM 29* (1986), 1202–1212
27. TARJAN, R. E. *Data Structures and Network Algorithms.* CBMS Regional Conference Series No. 44, SIAM, New York, 1983.
28. TOMIZAWA, N. On some techniques useful for the solution of transportation network problems. *Networks 1* (1971), 173–194.

RECEIVED MAY 1989; REVISED MAY 1990; ACCEPTED JULY 1990