December 1989

PARALLEL ASYNCHRONOUS HUNGARIAN METHODS FOR THE ASSIGNMENT PROBLEM[1]

by

Dimitri P. Bertsekas[2] and David A. Castañon[3]

Abstract

In this paper we discuss the parallel asynchronous implementation of the Hungarian method for solving the classical assignment problem. Multiple augmentations and price rises are simultaneously attempted starting from several unassigned sources and using possibly outdated price and assignment information. The results are then merged asynchronously subject to rather weak compatibility conditions. We show the validity of this algorithm and we demonstrate computationally that an asynchronous implementation is often faster than its synchronous counterpart.

[1] This work was supported in part by the BM/C3 Technology branch of the United States Army Strategic Defense Command. The authors would like to thank the Mathematics and Computer Science Division of the Argonne National Laboratory for providing access to the Advanced Computer Research Facility and training in the use of the Encore Multimax.

[2] Department of Electrical Engineering and Computer Science, M. I. T., Cambridge, Mass., 02139.

[3] ALPHATECH, Inc., Burlington, Mass., 01803.

1. INTRODUCTION

We consider the classical problem of optimal assignment of n persons to n objects, whereby given a benefit a_{ij} that person i associates with object j, we want to find an assignment of persons to objects, on a one-to-one basis, that maximizes the total benefit. The classical method for solving this problem is Kuhn's Hungarian method [Kuh55]. This method is of major theoretical interest and is still used widely. It maintains a price for each object and an (incomplete) assignment of persons and objects. At each iteration, the method chooses an unassigned person and computes a shortest augmenting path from this person to the set of unassigned objects, using reduced costs as arc lengths. An augmentation is then performed along the path and the object prices are adjusted to maintain complementary slackness; the process is repeated until there are no unassigned persons left. There are several discussions and implementations of this algorithm in the literature, some of which are referred to as sequential shortest path methods [CMT88], [Der85], [Eng82], [GGK82], [Hal56], [JoV87], [Law76], [McG83], [Pas82], [Roc84].

The classical version of the Hungarian method is serial in nature; only one shortest augmenting path is computed at a time. In a recent paper, Balas, Miller, Pekny, and Toth [BMP89] introduced an interesting and original idea for parallelization of the Hungarian method. In particular, they proposed the parallel construction of several shortest augmenting paths, each starting from a different unassigned person. They have shown that if these paths are pairwise disjoint, they can all be used to enlarge the current assignment; to preserve complementary slackness, the object prices should be raised to the maximum of the levels that would result from each individual shortest path calculation. Balas et al [BMP89] described an implementation of their parallel algorithm on the Butterfly Plus computer. Their computational results indicate a modest speedup (of the order of about 2) for the shortest path portion of their algorithm.

An obstacle in the way of a substantial speedup using the method of Balas et al is the synchronization required at the end of the parallel shortest path construction. To increase concurrency, a substantial number of shortest augmenting paths must be constructed at each iteration. However, these paths cannot be incorporated in the current assignment (and taken into account in subsequent shortest path calculations) until the end of the iteration. As a result, processors which have completed their computations must wait idly for other processors to complete their computations. In this paper, we rectify this shortcoming by developing a theory of asynchronous parallel Hungarian algorithms in which there is no concept of iteration and processors can proceed with the computation of new shortest paths regardless of the progress of other processors. We prove the validity of the corresponding asynchronous implementation and we demonstrate its merit by computational experimentation. As a special case, we recover both the algorithm of Balas et al [BMP89] and their convergence result. Our analysis extends nontrivially the analysis of [BMP89].

In the next section we describe an asynchronous algorithm that includes the classical Hungarian method as well as the synchronous parallel algorithm of Balas et al [BMP89] as special cases. In Section 3 we prove the validity of this asynchronous method, showing that it terminates with an optimal assignment in a finite number of steps, assuming existence of at least one feasible assignment.

In Section 4 we discuss a variety of synchronous and asynchronous implementations and we report on the results of our computational tests.

2. THE PARALLEL ASYNCHRONOUS ALGORITHM

In the assignment problem that we consider, n persons wish to allocate among themselves n objects, on a one-to-one basis. Each person i must select his object from a given subset $A(i)$. There is a given benefit a_{ij} that i associates with each object $j \in A(i)$. An *assignment* is a set of k person-object pairs $S = \big\{(i_1, j_1), \ldots, (i_k, j_k)\big\}$, such that $0 \leq k \leq n$, $j_m \in A(i_m)$ for all k, and the persons $i_1, \ldots, i_k$ and objects $j_1, \ldots, j_k$ are all distinct. The total benefit of the assignment is the sum $\sum_{m=1}^{k} a_{i_m j_m}$ of the benefits of the assigned pairs. An assignment is called *complete* (or *incomplete*) if it contains $k = n$ (or $k < n$, respectively) person-object pairs. We want to find a complete assignment with maximum total benefit, assuming that there exists at least one complete assignment.

The dual assignment problem is given by (see e.g. [BeT89], [PaS82], [Roc84])

$$\begin{aligned} &\text{minimize} \quad \sum_{j=1}^{n} p_j + \sum_{i=1}^{n} \pi_i \\ &\text{subject to} \quad p_j + \pi_i \geq a_{ij}, \qquad \forall\ (i,j) \text{ with } j \in A(i). \end{aligned} \tag{1}$$

Since we have $\pi_i = \max_{j \in A(i)}\{a_{ij} - p_j\}$ at the optimum, the dual variables π_i can be eliminated, thereby obtaining the following equivalent unconstrained problem

$$\begin{aligned} &\text{minimize} \quad \sum_{j=1}^{n} p_j + \sum_{i=1}^{n} \max_{j \in A(i)} \{a_{ij} - p_j\} \\ &\text{subject to} \quad \text{no constraints on } p_j,\ j = 1, \ldots, n. \end{aligned} \tag{2}$$

We refer to p_j as the *price* of object j. For a given price vector $p = (p_1, ..., p_n)$, the *reduced cost* of arc (i, j) is the nonnegative scalar r_{ij} given by

$$r_{ij} = \max_{m \in A(i)} \{a_{im} - p_m\} - (a_{ij} - p_j).$$

We say that a price vector p and an assignment S satisfy *complementary slackness* (or CS for short) if the corresponding reduced costs of the assigned arcs are zero,

$$r_{ij} = 0, \qquad \forall\ (i,j) \in S.$$

A classical optimality condition states that an assignment-price pair (S, p) solve the primal and dual problems, respectively, if and only if S is complete and satisfies CS together with p.

We now consider two operations that form the building blocks of our asynchronous algorithm. They operate on an assignment-price pair (S, p) and produce another assignment-price pair $(\hat{S}, \hat{p})$.

The first operation is *augmentation*, which is familiar from the theory of the Hungarian method. Here we have a person-object sequence $(i, j_1, i_1, j_2, i_2, \ldots, j_k, i_k, j)$, called an *augmenting path*, with the following properties:

(a) i and j are unassigned under S.

(b) $(i_m, j_m) \in S$, for all $m = 1, \ldots, k$.

(c) $j_1 \in A(i)$, $\quad j_m \in A(i_{m-1})$, for all $m = 1, \ldots, k$, and $j \in A(i_k)$.

(d) $r_{ij_1} = r_{i_1 j_2} = \cdots = r_{i_{k-1} j_k} = r_{i_k j} = 0$.

The augmentation operation leaves the price vector unaffected ($p = \hat{p}$), and yields an assignment $\hat{S}$ obtained from S by replacing (i_m, j_m), $m = 1, \ldots, k$ with $(i, j_1), (i_1, j_2), \ldots, (i_{k-1}, j_k), (i_k, j)$.

The second operation is called *price rise*. Here we have a set of assigned person-object pairs

$$\big\{(i_1, j_1), \ldots (i_k, j_k)\big\} \subset S$$

such that there is at least one i and $j \in A(i)$ with $i \in \{i_1, \ldots, i_k\}$ and $j \notin \{j_1, \ldots, j_k\}$, and such that the scalar γ given by

$$\gamma = \min\big\{r_{ij} \mid i \in \{i_1, \ldots, i_k\},\ j \notin \{j_1, \ldots, j_k\}\big\}$$

is positive. The operation consists of leaving S unchanged ($S = \hat{S}$) and setting

$$\hat{p}_j = \begin{cases} p_j, & \text{if } j \notin \{j_1, \ldots, j_k\} \\ p_j + \gamma & \text{if } j \in \{j_1, \ldots, j_k\}. \end{cases} \tag{3}$$

It can be seen that the following hold for the two operations just described:

(a) When an operation is performed on a pair (S, p) satisfying CS, it produces a pair $(\hat{S}, \hat{p})$ satisfying CS.

(b) As a result of an operation on (S, p), object prices cannot decrease (i.e., $p \leq \hat{p}$), and the price of an unassigned object cannot change (i.e., $p_j = \hat{p}_j$ if j is unassigned under S).

(c) An object that is assigned prior to an operation remains assigned after the operation.

We define a *generic iteration* on a pair (S, p) to be either an augmentation, or several price rises in succession followed by an augmentation. One way to implement the Hungarian method, given in [Law76], p. 206, consists of a sequence of generic iterations starting from a pair (S, p) satisfying CS. From this particular implementation it follows that if (S, p) satisfies CS, and S is an incomplete assignment, then it is possible to perform a generic iteration on (S, p).

A mathematically equivalent alternative implementation of the Hungarian method is based on constructing a shortest path from a single unassigned person to the set of unassigned sources using reduced costs as arc lengths; see e.g. [BMP89], [Der85], [JoV87]. In particular, at each iteration, given (S, p) satisfying CS, the *residual graph* is considered, which is the same as the bipartite graph of the assignment problem except that the direction of the assigned arcs $(i, j) \in S$ is reversed; the length of each arc is set to its reduced cost with respect to p. Then an unassigned person i is selected and the shortest path problem from i to the set of objects $j \notin S$ is solved. Let v_j be the shortest

distance from i to object j, let $\overline{j} = \arg\min_{j \notin S} v_j$, let $\overline{v} = v_{\overline{j}}$, and let P be the corresponding shortest path from i to $\overline{j}$. The successive shortest path iteration consists of the price change

$$\hat{p}_j = p_j + \max\{0, \overline{v} - v_j\}, \qquad j = 1, \ldots, n, \tag{4}$$

followed by an augmentation along the path P. The labeling process of the implementation in [Law76], p. 206, referred to earlier, amounts to the use of Dijkstra's method for the shortest path calculation. In particular, the price change of Eq. (4) can be decomposed into the sequence of price changes of the "permanently labeled" node sets in Dijkstra's method, and each of these price changes is a price rise operation in the sense of Eq. (3). Thus, a successive shortest path iteration may be viewed as a generic iteration of the type defined above.

We now describe the asynchronous algorithm. The assignment-price pair at the times

$$k = 1, 2, 3, \ldots$$

is denoted by $\big(S(k), p(k)\big)$. The initial pair $\big(S(1), p(1)\big)$ must satisfy CS. At each time k, a generic iteration is performed on a pair $\big(S(\tau_k), p(\tau_k)\big)$, where τ_k is a positive integer with $\tau_k \leq k$, to produce a pair $\big(\hat{S}(k), \hat{p}(k)\big)$. We say that the iteration (and the corresponding augmenting path) is *incompatible* if this path is not an augmenting path with respect to $S(k)$; in this case we discard the results of the iteration, that is, we set

$$S(k+1) = S(k), \qquad p(k+1) = p(k).$$

Otherwise, we say that the iteration (and the corresponding augmenting path) is *compatible*, and we set

$$p_j(k+1) = \max\big\{p_j(k), \hat{p}_j(k)\big\}, \qquad \forall\ j = 1, \ldots, n, \tag{5}$$

and we obtain $S(k+1)$ from $S(k)$ by performing the augmentation of the iteration. The algorithm terminates after at most n compatible iterations when $S(k)$ is complete.

We note that the definition of the asynchronous algorithm is not yet rigorous, because we have not yet proved that a generic iteration can be performed at any time prior to termination. This will be shown in the next section when we prove that all pairs $\big(S(k), p(k)\big)$ generated by the algorithm satisfy CS, so as discussed earlier, a generic iteration can be performed on $\big(S(k), p(k)\big)$ as long as the assignment $S(k)$ is incomplete.

The implementation of the asynchronous algorithm in a parallel shared memory machine is quite straightforward. A detailed description will be given in Section 4. The main idea is to maintain a "master" copy of the current assignment-price pair in the shared memory. To execute an iteration, a processor copies from the shared memory the current master assignment-price pair; during this copy operation the master pair is locked, so no other processor can modify it. The processor performs a generic iteration using the copy obtained, and then locks the master pair (which may by now differ from the copy obtained earlier). The processor checks if the iteration is compatible, and if so it modifies accordingly the master assignment-price pair. The processor then unlocks the master pair, possibly after retaining a copy to use at a subsequent iteration. The times when the master pair is copied and modified by processors correspond to the indexes τ_k and k of the asynchronous algorithm, respectively, as illustrated in Fig. 1.

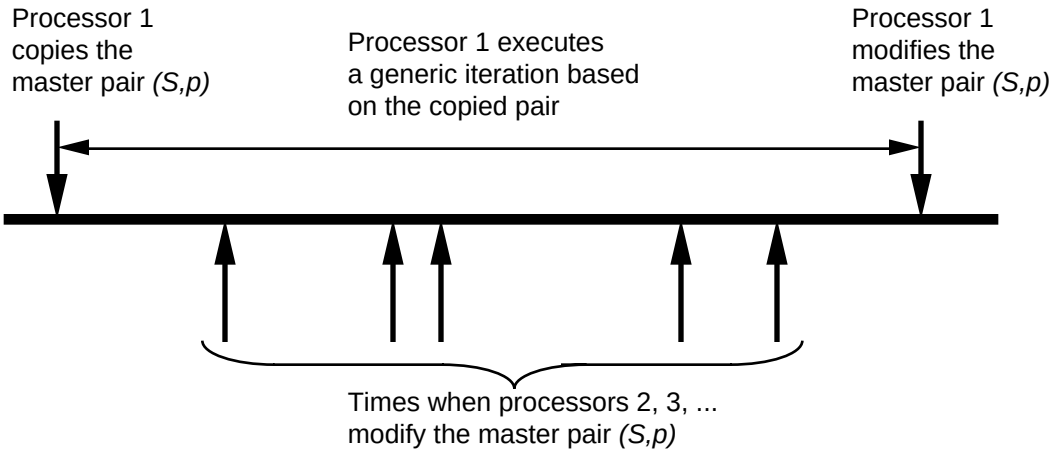

Figure 1: Operation of the asynchronous algorithm in a shared memory machine. A processor copies the master assignment-price pair at time τ_k, executes between times τ_k and k a generic iteration using the copy, and modifies accordingly the master assignment-price pair at time k. Other processors may have modified unpredictably the master pair between times τ_k and k.

3. VALIDITY OF THE ASYNCHRONOUS ALGORITHM

We want to show that the asynchronous algorithm maintains CS throughout its course. We first introduce some definitions and then we break down the main argument of the proof in a few lemmas.

An *alternating path* with respect to an assignment S is a sequence $(i_1, j_1, \ldots, i_k, j_k, \overline{j})$ such that

$$(i_m, j_m) \in S, \qquad m = 1, \ldots, k,$$

$$j_{m+1} \in A(i_m), \qquad m = 1, \ldots, k-1,$$

and $\overline{j} \in A(i_k)$. We say that the alternating path starts at j_1 and ends at $\overline{j}$.

Define the *cost length* of an alternating path P by

$$C(P) = \sum_{m=1}^{k-1} a_{i_m j_{m+1}} + a_{i_k \overline{j}} - \sum_{m=1}^{k} a_{i_m j_m}$$

and for a given price p, the *reduced cost length* of P by

$$R(p, P) = \sum_{m=1}^{k-1} r_{i_m j_{m+1}} + r_{i_k \overline{j}}.$$

These definitions are useful in relating the prices of the start and end objects of an alternating path.

Lemma 1: Assume that (S, p) satisfies CS. Let $P = (i_1, j_1, \ldots, i_k, j_k, \overline{j})$ be an alternating path with respect to S. Then

$$p_{\overline{j}} = p_{j_1} + C(P) + R(p, P).$$

Proof: Since by CS we have $a_{i_m j_m} - p_{j_m} = \max_{j \in A(i_m)}\{a_{ij} - p_j\}$ for all m,

$$r_{i_m j_{m+1}} = \big(a_{i_m j_m} - p_{j_m}\big) - \big(a_{i_m j_{m+1}} - p_{j_{m+1}}\big), \qquad m = 1, \ldots, k-1,$$

$$r_{i_k \overline{j}} = \big(a_{i_k j_k} - p_{j_k}\big) - \big(a_{i_k \overline{j}} - p_{\overline{j}}\big).$$

Adding these relations, we obtain the result. **Q.E.D.**

Let (S, p) be a pair satisfying CS, and let j be an assigned object under S. The *distance* of j with respect to (S, p), is the minimum of the reduced cost lengths of alternating paths starting at j and ending at some unassigned object; if there is no such path, the distance of j is defined to be ∞. For all objects j that are unassigned under S, the distance of j is defined to be zero. We denote for all k and j,

$$d_j(k) = \text{ Distance of } j \text{ with respect to } \big(S(k), p(k)\big).$$

The object distances play a central role in our analysis and are important for understanding how the algorithm works. To get a sense of their significance, consider a pair (S, p) satisfying CS, fix an object j which is assigned under S, and let $\mathcal{P}_j(S)$ be the set of all alternating paths starting at j and ending at an object which is unassigned under S. By Lemma 1, the distance of j is

$$d_j = \min_{P \in \mathcal{P}_j(S)} R(p, P) = \min_{P \in \mathcal{P}_j(S)} \big\{p_{\overline{j}(P)} - p_j - C(P)\big\},$$

where $\overline{j}(P)$ is the end object of the alternating path P. Thus we have

$$p_j + d_j = \min_{P \in \mathcal{P}_j(S)} \big\{p_{\overline{j}(P)} - C(P)\big\},$$

and we see that the quantity $p_j + d_j$ depends only on the assignment S and the prices of the unassigned nodes under S (not on the prices of the assigned objects under S). With a little thought it can be seen that, at least when all problem data and the vector p are integer, $\{p_j + d_j \mid j = 1, \ldots, n\}$ is the maximum set of prices that can be obtained from (S, p) by executing price rise operations exclusively (no augmentations). Thus $p_j + d_j$ may be viewed as a *ceiling price* that bounds from above the corresponding prices $\hat{p}_j$ obtained by a generic iteration; part (a) of the following lemma shows this fact. The lemma also proves two more properties: first, that if an object is part of an augmenting path of a generic iteration, its price becomes equal to the ceiling price prior to the iteration (Lemma 2(b)); second, that the ceiling prices are monotonically nondecreasing during the algorithm (Lemma 2(c)). The former property is important in showing that the parallel synchronous version of the algorithm, that is, the algorithm of Balas et al [BMP89], preserves CS. The latter property is important in showing that the same is true for the asynchronous algorithm.

Lemma 2: Let $k \geq 1$ be given and assume that $\big(S(t), p(t)\big)$ satisfies CS for all $t \leq k$. Then:

(a) For all objects j and all $t \leq k$, there holds

$$\hat{p}_j(t) \leq p_j(\tau_t) + d_j(\tau_t). \tag{6}$$

(b) For $t \leq k$, if $S(t+1) \neq S(t)$ (i.e., iteration t is compatible), and j is an object which belongs to the corresponding augmenting path, then we have

$$p_j(t) + d_j(t) = \hat{p}_j(t) = p_j(t+1). \tag{7}$$

(c) For all objects j and all $t \leq k-1$, there holds

$$p_j(t) + d_j(t) \leq p_j(t+1) + d_j(t+1). \tag{8}$$

Proof: (a) If j is unassigned under $S(\tau_t)$, we have $\hat{p}_j(t) = p_j(\tau_t)$ and the result holds. Thus, assume that j is assigned under $S(\tau_t)$. Since either $\hat{p}(t) = p(\tau_t)$ or else $\big(S(\tau_t), \hat{p}(t)\big)$ is obtained from $\big(S(\tau_t), p(\tau_t)\big)$ through a finite sequence of price rises, it follows that $\big(S(\tau_t), \hat{p}(t)\big)$ satisfies CS. Consider any alternating path P from j to an object $\overline{j}$, which is unassigned under $S(\tau_t)$. By Lemma 1, we have

$$p_{\overline{j}}(\tau_t) = p_j(\tau_t) + C(P) + R\big(p(\tau_t), P\big),$$

$$\hat{p}_{\overline{j}}(t) = \hat{p}_j(t) + C(P) + R\big(\hat{p}(t), P\big).$$

Since $\overline{j}$ is unassigned under $S(\tau_t)$, we have $p_{\overline{j}}(\tau_t) = \hat{p}_{\overline{j}}(t)$ and it follows that

$$\hat{p}_j(t) = p_j(\tau_t) + R\big(p(\tau_t), P\big) - R\big(\hat{p}(t), P\big) \leq p_j(\tau_t) + R\big(p(\tau_t), P\big).$$

Taking the minimum of $R\big(p(\tau_t), P\big)$ over all alternating paths P, starting at j and ending at unassigned objects under $S(\tau_t)$ the result follows.

(b), (c) We prove parts (b) and (c) simultaneously, by first proving a weaker version of part (b) (see Eq. (9) below), then proving part (c), and then completing the proof of part (b). Specifically, we will first show that for $t \leq k$, if $S(t+1) \neq S(t)$ and j is an object which belongs to the corresponding augmenting path, then we have

$$p_j(t) + d_j(t) \leq \hat{p}_j(t) = p_j(t+1). \tag{9}$$

Indeed, if j is unassigned under $S(t)$, Eq. (9) holds since we have $p_j(t) = \hat{p}_j(t)$ and $d_j(t) = 0$. Assume that j is assigned under $S(t)$. Let the augmenting path of iteration t end at object $\overline{j}$, and let P be the corresponding alternating path that starts at j and ends at $\overline{j}$. We have, using Lemma 1,

$$\hat{p}_{\overline{j}}(t) = \hat{p}_j(t) + C(P),$$

$$p_{\overline{j}}(t) = p_j(t) + C(P) + R\big(p(t), P\big).$$

Since $\overline{j}$ is unassigned under all $S(\tau)$ with $\tau \leq t$, we have $\hat{p}_{\overline{j}}(t) = p_{\overline{j}}(t)$, and we obtain

$$\hat{p}_j(t) = p_j(t) + R\big(p(t), P\big) \geq p_j(t) + d_j(t),$$

showing the left hand side of Eq. (9). Since $d_j(t) \geq 0$, this yields $p_j(t) \leq \hat{p}_j(t)$, so $\hat{p}_j(t) = \max\{p_j(t), \hat{p}_j(t)\} = p_j(t+1)$, completing the proof of Eq. (9).

We now prove part (c), making use of Eq. (9). Let us fix object j. If j is unassigned under $S(t+1)$, we have $p_j(t) = p_j(t+1)$ and $d_j(t) = d_j(t+1) = 0$, so the desired relation (8) holds. Thus, assume that j is assigned to i under $S(t+1)$, and let $P = (j, i, j_1, i_1, \ldots, j_k, i_k, \overline{j})$ be an alternating path with respect to $S(t+1)$, which is such that $\overline{j}$ is unassigned under $S(t+1)$ and

$$R\big(p(t+1), P\big) = d_j(t+1).$$

There are three possibilities:

(1) All the objects $j, j_1, \ldots, j_k$ have the same assignment under $S(t)$ as under $S(t+1)$. In this case we have using Lemma 1,

$$p_{\overline{j}}(t+1) = p_j(t+1) + C(P) + R\big(p(t+1), P\big),$$

$$p_{\overline{j}}(t) = p_j(t) + C(P) + R\big(p(t), P\big).$$

Since $\overline{j}$ is unassigned under $S(t+1)$, we have $p_{\overline{j}}(t+1) = p_{\overline{j}}(t)$, so we obtain

$$p_j(t+1) + R\big(p(t+1), P\big) = p_j(t) + R\big(p(t), P\big).$$

Since $R\big(p(t+1), P\big) = d_j(t+1)$ and $R\big(p(t), P\big) \geq d_j(t)$, we obtain

$$p_j(t) + d_j(t) \leq p_j(t+1) + d_j(t+1),$$

and the desired relation (8) is proved in this case.

(2) Iteration t is compatible and object j belongs to the augmenting path of iteration t, in which case, by Eq. (9), we have

$$p_j(t) + d_j(t) \leq p_j(t+1) \leq p_j(t+1) + d_j(t+1),$$

and the desired relation (8) is proved in this case as well.

(3) Iteration t is compatible, and there is an object j_m, $m \in \{1, \ldots, k\}$, which belongs to the augmenting path of iteration t, and is such that j and $j_1, \ldots, j_{m-1}$ did not change assignment at iteration t; see Fig. 2. Consider the following alternating paths with respect to $S(t+1)$

$$P' = (j, i, j_1, i_1, \ldots, j_{m-1}, i_{m-1}, j_m),$$
$$P'' = (j_m, i_m, j_{m+1}, i_{m+1}, \ldots, j_k, i_k, \overline{j}).$$

We have

$$R\big(p(t+1), P'\big) = R\big(p(t), P'\big) + \big(p_{j_m}(t+1) - p_{j_m}(t)\big) - \big(p_j(t+1) - p_j(t)\big),$$

and since by Eq. (9), $p_{j_m}(t+1) - p_{j_m}(t) \geq d_{j_m}(t)$, we obtain

$$R\big(p(t+1), P'\big) + p_j(t+1) \geq R\big(p(t), P'\big) + d_{j_m}(t) + p_j(t). \tag{10}$$

On the other hand, we have

$$R\big(p(t+1), P\big) = R\big(p(t+1), P'\big) + R\big(p(t+1), P''\big)$$

and since $R\big(p(t+1), P''\big) \geq 0$, we obtain

$$R\big(p(t+1), P\big) \geq R\big(p(t+1), P'\big). \tag{11}$$

Combining Eqs. (10) and (11), we see that

$$R\big(p(t+1), P\big) + p_j(t+1) \geq R\big(p(t), P'\big) + d_{j_m}(t) + p_j(t).$$

We have $R\big(p(t), P'\big) + d_{j_m}(t) \geq d_j(t)$, and $R\big(p(t+1), P\big) = d_j(t+1)$, so it follows that

$$p_j(t+1) + d_j(t+1) \geq p_j(t) + d_j(t),$$

and the proof of part (c) is complete.

To complete the proof of part (b), we note that by using Eqs. (6) and (8), we obtain

$$\hat{p}_j(t) \leq p_j(\tau_t) + d_j(\tau_t) \leq p_j(t) + d_j(t),$$

which combined with Eq. (9) yields the desired Eq. (8). **Q.E.D.**

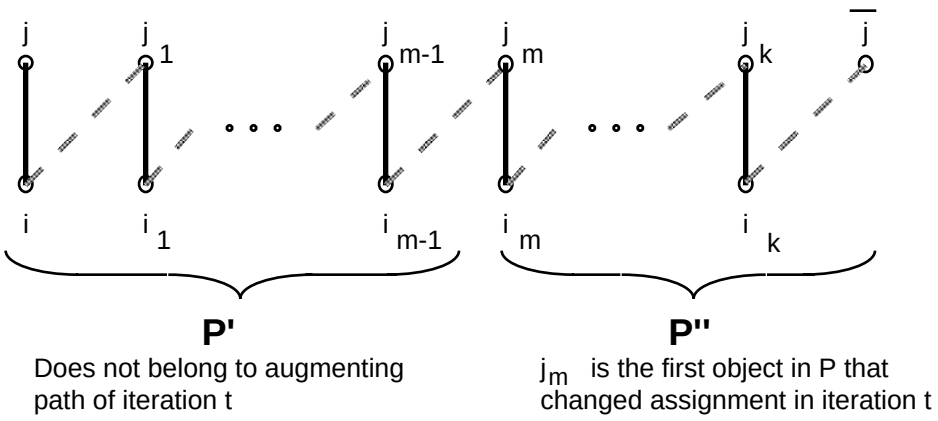

Figure 2: Decomposition of alternating path P used in the proof of Lemma 3.

We need one more simple technical lemma. Let us denote:

$r_{ij}(k)$: Reduced cost of arc (i,j) with respect to $p(k)$,
$\hat{r}_{ij}(k)$: Reduced cost of arc (i,j) with respect to $\hat{p}(k)$.

Lemma 3: Assume that the k^{th} iteration is compatible. Then:

(a) $\hat{p}_j(k) \geq p_j(k) \quad \Longrightarrow \quad \hat{r}_{ij}(k) \geq r_{ij}(k+1)$.
(b) $p_j(k) \geq \hat{p}_j(k) \quad \Longrightarrow \quad r_{ij}(k) \geq r_{ij}(k+1)$.

Proof: (a) Since $\hat{p}_j(k) = \max\{\hat{p}_j(k),\, p_j(k)\} = p_j(k+1)$, we have

$$a_{ij} - \hat{p}_j(k) = a_{ij} - p_j(k+1),$$

and since $p_m(k+1) \geq \hat{p}_m(k)$ for all m, we have

$$\max_{m \in A(i)} \big\{a_{im} - \hat{p}_m(k)\big\} \geq \max_{m \in A(i)} \big\{a_{im} - p_m(k+1)\big\}.$$

Subtracting the preceding two relations, we get

$$\hat{r}_{ij}(k) \geq r_{ij}(k+1).$$

(b) Very similar proof as for part (a). **Q.E.D.**

We can now prove that the asynchronous algorithm preserves CS.

Proposition 1: All pairs $\big(S(k), p(k)\big)$ generated by the asynchronous algorithm satisfy CS.

Proof: By induction. Suppose all iterations up to the k^{th} maintain CS, and let the k^{th} iteration be compatible. We will show that the pair $\big(S(k+1), p(k+1)\big)$ satisfies CS, that is,

$$(i,j) \in S(k+1) \qquad \Longrightarrow \qquad r_{ij}(k+1) = 0.$$

Let $(i,j) \in S(k+1)$. There are two possibilities:

(1) (i,j) belongs to the augmenting path of the k^{th} iteration, in which case $\hat{r}_{ij}(k) = 0$. By Lemma 2(b), we have $\hat{p}_j(k) \geq p_j(k)$, so by Lemma 3(a), $r_{ij}(k+1) \leq \hat{r}_{ij}(k) = 0$.

(2) (i,j) does not belong to the augmenting path of the k^{th} iteration, in which case, by the CS property (cf. the induction hypothesis), we have $(i,j) \in S(k)$ and $r_{ij}(k) = 0$. If $p_j(k) \geq \hat{p}_j(k)$, by Lemma 3(b) we have $r_{ij}(k+1) \leq r_{ij}(k) = 0$ and we are done. Assume therefore that $\hat{p}_j(k) > p_j(k)$, in which case there are two possibilities: (a) We have $(i,j) \in S(\tau_k)$, in which case $\hat{r}_{ij}(k) = 0$. By Lemma 3(a) we then obtain $r_{ij}(k+1) \leq \hat{r}_{ij}(k) = 0$. (b) We have $(i,j) \notin S(\tau_k)$. We will show that this case cannot arise. In particular, we will assume that for some $(i,j) \in S(k)$ we have

$$(i,j) \notin S(\tau_k) \qquad \text{and} \qquad \hat{p}_j(k) > p_j(k)$$

and arrive at a contradiction, thereby completing the proof. We first note that j must be assigned under $S(\tau_k)$, for otherwise we would have

$$\hat{p}_j(k) = p_j(\tau_k) \leq p_j(k),$$

contradicting the hypothesis $\hat{p}_j(k) > p_j(k)$. Let t_1 be the first iteration index such that $\tau_k < t_1 \leq k$ and $(i,j) \in S(t_1)$. Then by parts (a) and (c) of Lemma 2, we have

$$\hat{p}_j(k) \leq p_j(\tau_k) + d_j(\tau_k) \leq p_j(t_1 - 1) + d_j(t_1 - 1),$$

while by Lemma 2(b), we have

$$p_j(t_1 - 1) + d_j(t_1 - 1) = p_j(t_1)$$

Since object prices cannot decrease, we have $p_j(t_1) \leq p_j(k)$, and the preceding two inequalities yield $\hat{p}_j(k) \leq p_j(k)$, arriving at a contradiction. **Q.E.D.**

Proposition 1 shows that if the asynchronous algorithm terminates, the assignment-price pair obtained satisfies CS. Since the assignment obtained at termination is complete, it must be optimal. To guarantee that the algorithm terminates, we impose the condition

$$\lim_{k \to \infty} \tau_k = \infty.$$

This is a natural and essential condition, stating that the algorithm iterates with increasingly more recent information.

Proposition 2: If $\lim_{k\to\infty} \tau_k = \infty$, the asynchronous algorithm terminates with an optimal assignment.

Proof: There can be at most n compatible iterations, so if the algorithm does not terminate, all iterations after some index $\bar{k}$ are incompatible, and $S(k) = S(\bar{k})$ for all $k \geq \bar{k}$. On the other hand, since $\lim_{k\to\infty} \tau_k = \infty$, we have that $\tau_k \geq \bar{k}$ for all k sufficiently large, so that $S(\tau_k) = S(k)$ for all $k \geq \bar{k}$. This contradicts the incompatibility of the k^{th} iteration. **Q.E.D.**

4. COMPUTATIONAL RESULTS

In order to evaluate the relative performance of parallel synchronous and asynchronous Hungarian methods, we developed three different variations of the successive shortest path algorithm, the first two of which are synchronous and the third is asynchronous. These variations differ in the amount of work to be done by a processor in each iteration before the results of the processor's computation are used to change the current assignment and prices. These variations are:

(1) *Single path synchronous augmentation*: Here, at each iteration, every processor finds a single shortest augmenting path from an unassigned person to an unassigned object. When the number U of unassigned persons becomes less than the number of processors P, $P - U$ processors become idle.

(2) *Self-scheduled synchronous augmentation*: Here, at each iteration, every processor finds a variable number of shortest augmenting paths sequentially until the total number of augmenting paths equals some threshold number, which depends on the number of unassigned persons and the number of processors.

(3) *Single path asynchronous augmentation*: Here, at each iteration, each processor finds a single shortest augmenting path from an unassigned person to an unassigned object, but the processors execute the iterations asynchronously.

In the subsequent subsections, we describe in greater detail the above three algorithms, and we compare their performance on a shared-memory Encore Multimax for assignment problems with 1000 persons and varying density.

The three algorithms were developed by suitably modifying the sequential shortest path portion of the code of Jonker and Volgenant [JoV87] for sparse assignment problems. This code is described in terms of four phases in [JoV87]:

(1) Column Reduction

(2) Reduction Transfer

(3) Augmenting Row Reduction

(4) Successive Shortest Paths

The Column Reduction and Reduction Transfer phases are initialization phases, which obtain an initial set of prices p and an initial assignment S satisfying complementary slackness. A brief description of these two phases is as follows:

(a) S : empty;

(b) $p_j = \max_{\{i \mid j \in A(i)\}} a_{ij},\ j = 1, \ldots, n;$

(c) $i^*(j) = \min\{i \mid j \in A(i), a_{ij} = p_j\},\ j = 1, \ldots, n;$

(d) $j^*(i) = \begin{cases} \min\{j \mid i^*(j) = i\} & \text{if } \{j \mid i^*(j) = i\}\text{: nonempty} \\ 0 & \text{otherwise} \end{cases} \qquad i = 1, \ldots, n;$

(e) For $i = 1, \ldots, n$, if $j^*(i) > 0$, then $S = S \cup \big(i, j^*(i)\big)$.

(f) For $(i, j) \in S$, set $p_j := p_j - \max_{\{j' \mid j' \in A(i), j' \neq j\}} \{a_{ij'} - p_{j'}\}$.

The Augmenting Row Reduction phase does not consist of Hungarian iterations but rather it is a sequence of single-person relaxation iterations as proposed in [Ber81]; equivalently, it may be viewed as a sequence of auction algorithm iterations where $\epsilon = 0$ (see [Ber87], [BeT89], [BeC89]). Thus, to obtain a purely Hungarian method we discarded the Augmenting Row Reduction phase. The Successive Shortest Paths phase is a straightforward implementation of Dijkstra's shortest path algorithm that uses no special data structures such as D-heaps or R-heaps. Thus, our codes may be viewed as parallel versions of the code of [JoV87] except that the Augmenting Row Reduction phase has been eliminated.

Synchronous Single-Path Augmentation Algorithm (SS)

At each iteration of the synchronous single-path augmentation algorithm, each processor selects a different unassigned person and finds a shortest augmenting path from that person to the set of unassigned objects. The algorithm is synchronous because all of the processors use the same assignment and price data: the ones produced by the previous iteration. Once a processor finds an augmenting path, it checks for compatibility of this path versus the paths already incorporated in the assignment by other processors; if the path is compatible, the assignment and prices are updated as described in Section 2; otherwise, the path is discarded. The processor then waits until all the other processors complete their augmenting path computations before starting a new iteration.

For a more precise description, let $(S(k), p(k))$ be the assignment-price pair available at the start of the k^{th} iteration, and let M_k be the minimum of the number of processors and the number of unassigned persons under $S(k)$. Then, each processor $m = 1, \ldots, M_k$ selects a different person i_m from the queue of unassigned persons according to $S(k)$, and computes a shortest augmenting path P_m from i_m to the set of unassigned objects under $S(k)$, and a corresponding price vector $\hat{p}^m(k)$. Without loss of generality, assume that the order in which the processors complete this computation is $1, 2, \ldots, M_k$. Then, $(S(k), p(k))$ is updated as follows:

$$(S', p') := (S(k), p(k)) \tag{12}$$

Do $m = 1, \ldots, M_k$

If P_m is an augmenting path with respect to S', update S'
by performing the corresponding augmentation and set

$$p'_j := \max\{p'_j, \hat{p}^m_j(k)\}, \qquad j = 1, \ldots, n. \tag{13}$$

End do

$$S(k+1) := S', \qquad p(k+1) := p' \tag{14}$$

The temporary pair (S', p') is maintained for checking the compatibility of each new augmenting path with the previous augmenting paths. After all processors have completed their compatibility check and attendant updating of (S', p'), the master assignment-price pair is set to (S', p').

The overall logic of the synchronous single-person augmentation algorithm is illustrated in Fig. 3. In our implementation, the master pair $(S(k), p(k))$ and the temporary pair (S', p') are stored in shared memory. The set of unassigned persons under $S(k)$ is maintained in a queue; a lock on this queue is used in order to guarantee that each processor searches for an augmenting path starting at a different person. Note the synchronization barrier at the end of each iteration, and the sequential operation of copying the temporary assignments and prices to the permanent assignment and prices. A single synchronization lock on the temporary assignment and prices is used to guarantee that the updates of Eq. (13) are done in a sequential manner. Note that, whenever an augmentation is deemed incompatible with the temporary assignment, the unassigned person associated with that augmentation is reinserted into the unassigned persons queue.

The two principal drawbacks of the synchronous single-person augmentation algorithm are the idle time spent by each processor at the barrier while other processors are still computing augmenting paths, and the overhead required for copying the temporary assignments and prices onto the permanent assignments and prices. The second drawback can be alleviated by using additional processors, since less iterations will be required for convergence. However, increasing the number of processors will increase the variability of the computation times for the different augmentations in each iteration, and thus increase the overall percentage of time that a processor spends waiting at the barrier.

Table 1 illustrates the performance of the parallel synchronous single-path augmentation algorithm for a 1000 person fully dense assignment problem with cost range [1,1000] as a function of the number of processors used in the Encore Multimax. The table contains the time required for

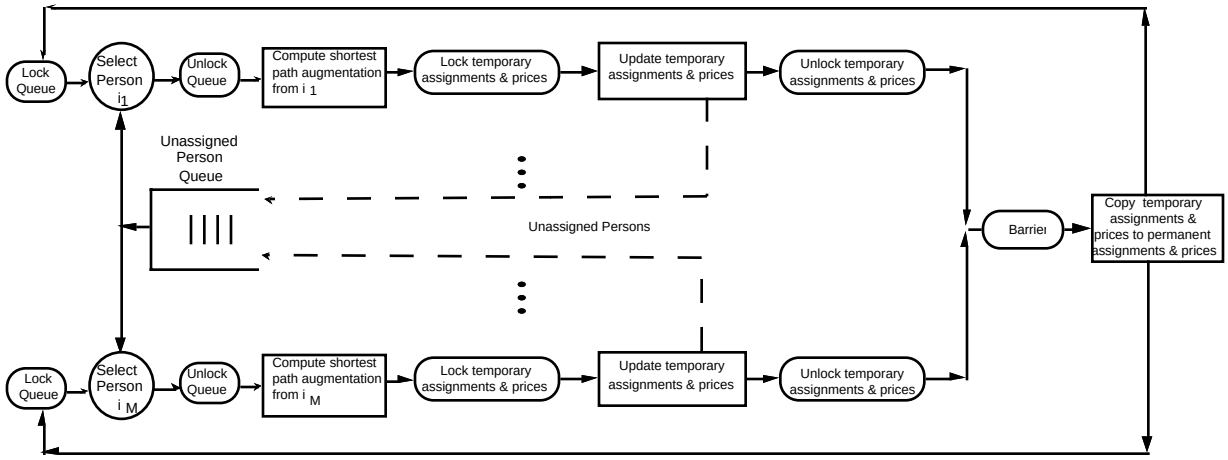

Figure 3: Design of parallel synchronous single-path augmentation algorithm.

the Successive Shortest Paths phase of three different runs and the number of incompatible augmentations as a function of the number of processors used. The total number of unassigned persons at the beginning of the Successive Shortest Paths phase was 397; thus, the overhead for incompatible augmentations is a small fraction of the overall computation in these experiments. Note the variability of the run times for different runs; this is due to randomness in the order of completion of the individual processors, which can lead to differences as to which augmentations are declared incompatible.

Table 1: Run times for the synchronous single-path augmentation algorithm

# Proc.	1	2	4	6	8
Time, Run # 1	104	86.14	76.04	69.32	73.2
Time, Run # 2	103.3	85.59	85.07	60.3	69.98
Time, Run # 3	103.68	85.331	80.9	63.64	70.88
Rejected augmentations, Run # 1	0	4	24	28	44
Rejected augmentations, Run # 2	0	4	25	24	44
Rejected augmentations, Run # 3	0	4	28	24	44

Table 1 suggests that the synchronous single-path augmentation algorithm can achieve a relatively limited speedup. There are two potential reasons for this: the synchronization overhead arising from processors waiting at the barrier for other processors to complete their computations, and the sequential part of the computation which arises at the last iteration when there is only one unassigned person remaining. In order to verify these hypotheses, we measured the total number

of seconds that each processor spent waiting for other processors to complete their computations averaged across processors (the *average wait time*), and the number of seconds taken by the last iteration assuming only one person was assigned at the last iteration (this is called the *sequential computation time*); if the last iteration involves more than one path, the sequential computation time is zero. The sequential computation time is an indication of the amount of sequential work in the Successive Shortest Paths phase which was not parallelizable.

As Table 2 indicates, the average wait time of each processor when using multiple processors is close to 17% of the single-processor computation time! This additional synchronization overhead reduces the multiprocessor efficiency of the asynchronous algorithm. In addition, the sequential computation time is also nearly 11% of the single-processor computation time, further limiting the obtainable speedup. The last augmenting path is typically one of the longest. Note that, for some of the runs using four processors, the sequential computation time was 0; this corresponds to the case where the last two paths are found simultaneously in the last iteration without conflict, and depends critically on the sample path followed by the computations.

It is interesting that the average wait time does not increase significantly with the number of processors. The reason is that, although there is increased variability in the computation times of the different augmentations by different processors, the number of times for which the processors need to be synchronized is reduced (because more augmenting paths are found in parallel); these two effects appear to cancel each other out, leading to a nearly constant average wait time.

Table 2: Average wait and sequential computation times for the synchronous single-path algorithm

# Proc.	1	2	4	6	8
Average Wait Time, Run # 1	0	17.3	17.57	18	17.06
Average Wait Time, Run # 2	0	18.6	17.75	14.24	14.4
Average Wait Time, Run # 3	0	15.4	17.48	16.95	16.42
Sequential Computation Time, Run # 1	NA	11.7	0	11.3	11.3
Sequential Computation Time , Run # 2	NA	11.6	11.1	11.3	11.3
Sequential Computation Time, Run # 3	NA	11.2	0	11.8	12

Self-Scheduled Synchronous Augmentation Algorithm (SSS)

One of the main limitations in efficiency of the synchronous single-path augmentation algorithm is the average wait time incurred by each processor after finding an augmenting path. In order to reduce this limitation, the self-scheduled synchronous augmentation algorithm allows each processor to find several augmenting paths before attempting to synchronize the results with the computations of other processors. Specifically, during each iteration, each processor selects a different unassigned person and finds a shortest augmenting path from that person to the set of unassigned objects. The algorithm is synchronous because all of the processors use the same assignment-price

pair $\big(S(k), p(k)\big)$. Once a processor finds an augmenting path, it checks for the compatibility of this augmentation versus augmentations already found by other processors, and updates the temporary pair (S', p') as in the synchronous single-path augmentation algorithm. However, instead of proceeding to a barrier and waiting for the remaining processors to complete their computations, the processor checks whether the total number of unassigned persons considered during this iteration is less than a threshold $T(k)$ (which is iteration-dependent). If the number is less than $T(k)$, the processor retrieves another unassigned person, finds another augmenting path, and repeats the process; otherwise, the processor proceeds to a barrier and waits for other processors to complete their computations.

Figure 4 illustrates the overall logic of the self-scheduled synchronous augmentation algorithm. As before, our implementation stores the permanent pair $\big(S(k), p(k)\big)$ and temporary pair (S', p') in shared memory. The set of unassigned persons in the assignment $S(k)$ is maintained in a queue (with a shared lock) and a synchronization barrier is used at the end of each iteration k.

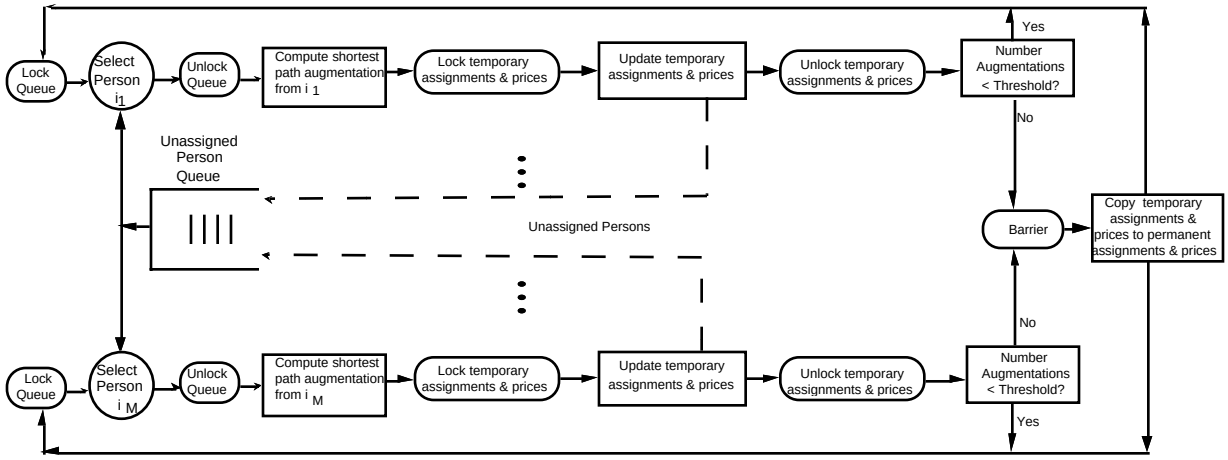

Figure 4: Design of parallel self-scheduled synchronous augmentation algorithm.

The iterations of the self-scheduled synchronous augmentation algorithm are similar to those described in Eqs. (12)-(14). The only difference is that each processor can modify the temporary assignments S' and the temporary prices p' more than once. The total number of augmentations at iteration k is determined by the threshold $T(k)$, which was chosen as follows: Let $U(k)$ denote the number of unassigned persons at the beginning of iteration k, and let P denote the number of processors. Then

$$T(k) = \begin{cases} \max\left\{ \left\lfloor \frac{U(k)}{3} \right\rfloor, P \right\} & \text{if } U(k) > P \\ U(k) & \text{otherwise.} \end{cases}$$

This is similar to the approach used in Balas et al [BMP89].

Table 3 illustrates the performance of the parallel self-scheduled synchronous algorithm for the same 1000 person dense assignment problem with cost range [1,1000] used previously, as a function of the number of processors used. The table contains the time required for the Successive Shortest

Paths phase of three different runs and the number of incompatible augmentations, as well as the average results. The total number of unassigned persons at the beginning of the Successive Shortest Paths phase was 397. Here the incompatible augmentations represent a significant fraction of the required work; this is easily seen by comparing the single processor times in Table 1 with those in Table 3, which indicate an increase of nearly 90% additional computation. The increase in incompatible augmentations is indicative of the use of older assignment-price pair information.

Table 3: Run times for the self-scheduled synchronous augmentation algorithm

# Proc.	1	2	4	6	8
Time, Run # 1	198.4	94.9	83.1	69.3	68.5
Time, Run # 2	199.5	99.5	81.6	53.6	68.3
Time, Run # 3	200	99.5	83	53	68.3
Rejected augmentations, Run # 1	112	100	103	96	103
Rejected augmentations, Run # 2	112	100	103	104	103
Rejected augmentations, Run # 3	112	100	103	104	103

The results of Table 3 indicate that parallelization of the self-scheduled synchronous augmentation algorithm is more efficient than parallelization of the synchronous single-path augmentation algorithm, in spite of the increased computation load associated with a larger number of augmentations. Table 4 illustrates the average wait time and the sequential computation time associated with the above runs. Contrasting the results of Tables 2 and 4, we see that the average wait times in Table 4 are nearly 40% smaller than the comparable times in Table 2, in spite of the greater number of augmenting paths computed. The sequential computation times are similar in both tables; note, however, that the 0 sequential computation times using 6 processors lead to very efficient timings in the self-scheduled synchronous augmentation algorithm. Unfortunately, this event is a function of the sample path followed by the computations, rather than an inherent property of the algorithm.

Table 4: Average wait and sequential computation times for the self-scheduled synchronous algorithm

# Proc.	1	2	4	6	8
Average Wait Time, Run # 1	0	11	11	10.2	14
Average Wait Time, Run # 2	0	8	10.1	11.1	13
Average Wait Time, Run # 3	0	9	11	11.1	13
Sequential Computation Time, Run # 1	NA	0	11	12.2	11.4
Sequential Computation Time , Run # 2	NA	0	11.5	0	11.5
Sequential Computation Time, Run # 3	NA	0	11.5	0	11.6

Single-Path Asynchronous Augmentation Algorithm (AS)

One of the major factors which limited the speedup of the previous parallel synchronous algorithms was the average wait time incurred by each processor. The single-path asynchronous augmentation algorithm was designed to reduce this overhead by allowing processors to directly augment the permanent assignment $S(k)$ and modify the price vector $p(k)$ without waiting for other processors to complete their computations. In this algorithm, each processor selects a different unassigned person, and finds an augmenting path from this person to the set of unassigned objects. Once a processor finds an augmenting path, it checks the compatibility of the path with the current state of the network (as described in Section 2), and augments the assignment and raises the object prices if the augmentation is compatible. The processor then obtains a copy of the current network state, selects another unassigned person and proceeds to find another augmenting path using the updated state of the network.

Figure 5 illustrates the logic of the single-path asynchronous augmentation algorithm. In contrast with the previous synchronous algorithms, there is no barrier at the end of each iteration where processors must wait for other processors to complete their computations. Instead, each processor locks the master copy of the current assignment-price pair, modifies it according to the compatibility of its augmenting path, and releases the lock on the master copy. The processor then repeats the process with a new unassigned person.

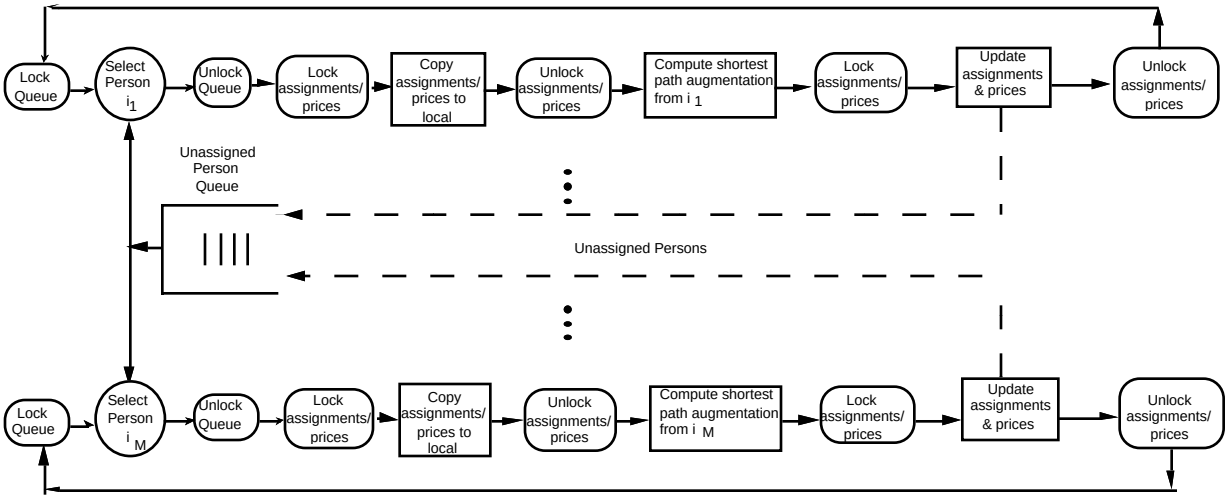

Figure 5: Design of parallel asynchronous single-path augmentation algorithm.

Thus, the asynchronous algorithm maintains $M+1$ copies (where M is the number of processors currently used) of the assignment-price pair (S, p). In our implementation, a master copy is maintained in shared memory, and local copies are maintained in the local memory of each processor. In contrast, the synchronous algorithms discussed previously maintain 2 copies of (S, p) (permanent and temporary), both in shared-memory.

Table 5 illustrates the performance of the single-path asynchronous augmentation algorithm for

the same 1000 person dense assignment problem described previously. Again, the computation times reported are the times for the Successive Shortest Paths phase. Contrasting Table 5 with the corresponding results in Tables 1 and 3, we see that the asynchronous algorithm tends to be faster than the corresponding synchronous algorithms; however, the improvement in run-time is greater for smaller numbers of processors. Note that the number of augmenting paths rejected increases rapidly as the number of processors increases, thereby increasing the computation load. Surprisingly, this number is larger than the corresponding number for the single-path synchronous augmentation algorithm (although in principle the asynchronous algorithm is using more recent information concerning the assignments and prices).

Table 5: Run times for the asynchronous single-path augmentation algorithm

# Proc.	1	2	4	6	8
Time, Run # 1	107	64.5	60.4	64.9	64.5
Time, Run # 2	107	73.5	58.5	52.2	66.3
Time, Run # 3	106	63.4	58.9	61.6	62.7
Rejected augmentations, Run # 1	0	13	31	44	60
Rejected augmentations, Run # 2	0	15	30	45	62
Rejected augmentations, Run # 3	0	13	30	43	52

Although there is no synchronization barrier, there is some delay associated with acquiring a lock on the master copy of the assignment-price pair due to possible conflicts with other processors which are also trying to write or copy this pair. Thus, we define the *average wait time* per processor to be the total amount of time (averaged across processors) that a processor spent waiting to acquire access to locked data structures. This time increases with the number of processors and therefore limits the achievable speedup.

Table 6 contains the average wait time and sequential computation time of the single-path asynchronous augmentation algorithm. Comparing the results of Table 6 with the corresponding results of Table 2, we see that the average wait time in the single-path asynchronous augmentation algorithm was reduced considerably relative to the corresponding synchronous algorithm, leading to substantial reductions in computation time. However, this computation advantage is often limited because of an increase in sequential computation time in the asynchronous algorithm. This increase is due to the asynchronous nature of the algorithm, which may require that the last augmenting path be computed twice (because of potential incompatibility due to the use of old assignment information); in contrast, the synchronous algorithms guarantee that the last augmenting path need only be computed once. Note the substantial variability in nonzero sequential computation times for the asynchronous algorithm when compared to those of the synchronous algorithms in Tables 2 and 4.

Table 6: Average wait and sequential computation times for the asynchronous single-path algorithm

# Proc.	1	2	4	6	8
Average Wait Time, Run # 1	0	.12	.31	.6	1
Average Wait Time, Run # 2	0	.1	.26	.67	.95
Average Wait Time, Run # 3	0	.13	.3	.7	1
Sequential Computation Time, Run # 1	NA	12.1	14.9	18.7	23.7
Sequential Computation Time, Run # 2	NA	15.2	.75	0	16
Sequential Computation Time, Run # 3	NA	12	.6	15	13

Performance of Parallel Hungarian Algorithms for Sparse Assignment Problems

The performance results in the previous subsections were obtained using a dense 1000-person assignment problem. For such problems, the ratio of the time required to find a shortest path (from an unassigned person to the set of unassigned objects) to the time required to make a copy of the current assignment-price pair is large. As the density of the assignment problem decreases, this ratio is likely to decrease because the time required to make a copy of the current assignment-price pair remains constant (depending only on the number of objects), while the time required to find a shortest path will decrease by exploiting sparsity. In this subsection, we illustrate the effects of sparsity on the relative performance of the three parallel Hungarian algorithms discussed previously.

Figure 6 illustrates the computation time in the Successive Shortest Paths phase (averaged across three runs) of the three parallel Hungarian algorithms for a 1000 person, 30% dense assignment problem, cost range [1,1000]. When a single processor is used, the self-scheduled synchronous augmentation algorithm is the slowest because it must find additional shortest paths (as a result of incompatibility problems). The asynchronous algorithm is the fastest because of the reduced synchronization overhead. Similarly, the self-scheduling synchronous augmentation algorithm is faster than the synchronous single-path augmentation algorithm because of the reduced synchronization overhead.

Figure 7 illustrates the average wait time for each of the algorithms in Fig. 6. These curves illustrate the dramatic reductions in synchronization overhead which can be achieved by the asynchronous algorithm. The average wait time of the asynchronous algorithm grows almost linearly with the number of processors but is a small fraction of the overall computation time (less than 10%).

Figure 8 illustrates the computation time in the Successive Shortest Paths phase (averaged across three runs) of the three algorithms for a 1000-person, 2% dense assignment problem, with cost range [1,1000]. As in Fig. 6, the asynchronous algorithm makes very effective use of a small number of processors (≤ 2). However, as the number of processors increases, the speedup achievable by the asynchronous algorithm is much smaller than the speedup achievable by the two synchronous algorithms. The principal reason for this is a relative increase in the average wait time per processor for the asynchronous algorithm.

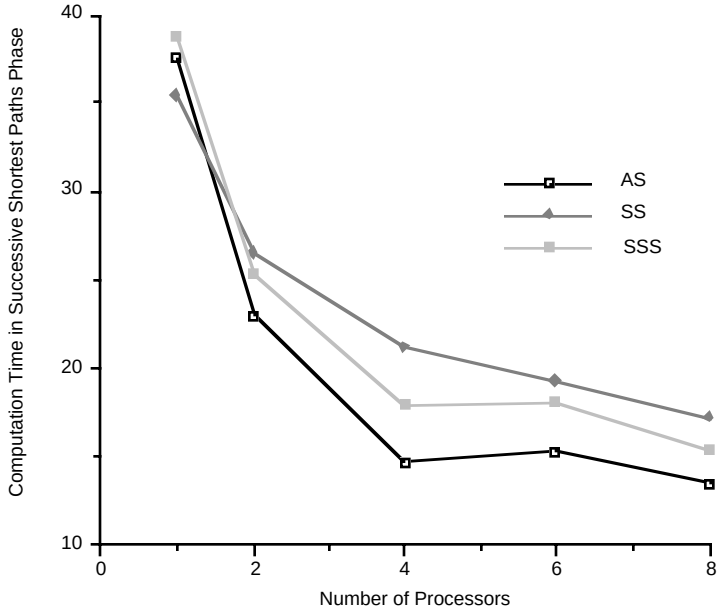

Figure 6: Computation time of parallel Hungarian algorithm for 1000 person, 30% dense assignment problem, with cost range [1,1000], as a function of the number of processors used on the Encore Multimax.

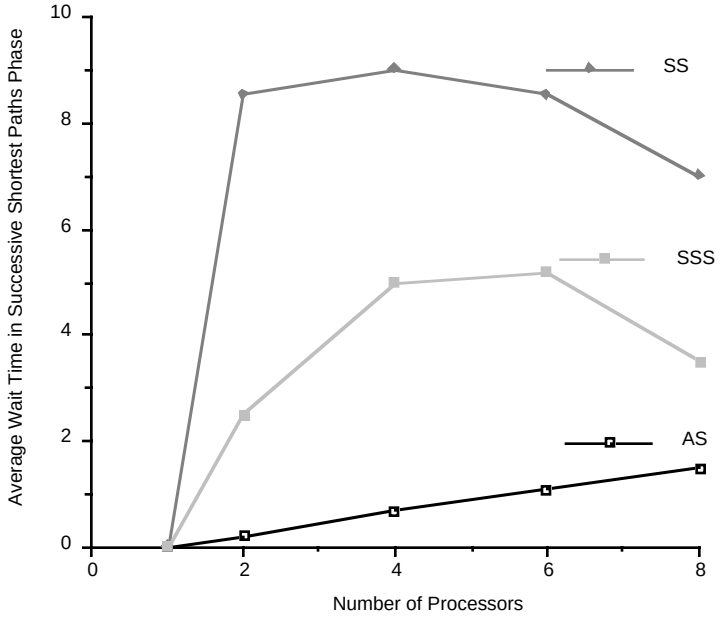

Figure 7: Average wait time per processor for 1000 person, 30% dense assignment problem, with cost range [1,1000], as a function of the number of processors used on the Encore Multimax.

Figure 9 describes the variation of the average wait time per processor as a function of the number of processors for the three algorithms. As the figure illustrates, the average wait times for

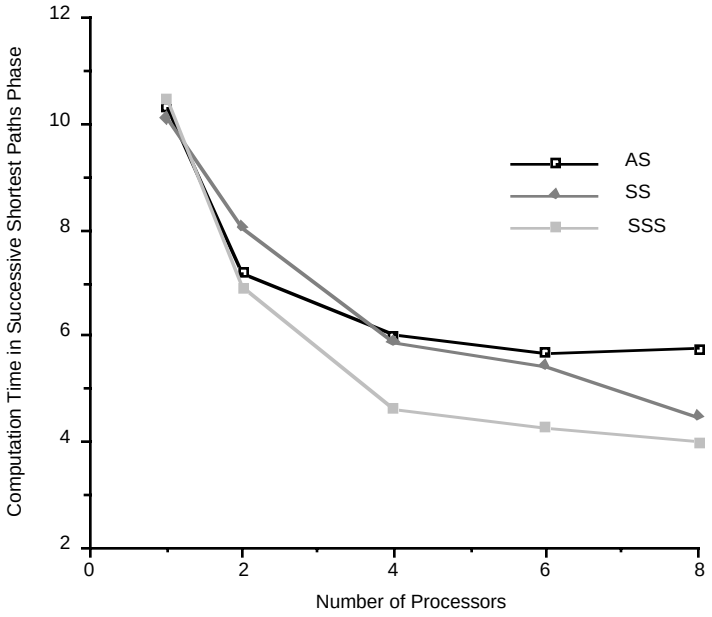

Figure 8: Computation time of parallel Hungarian algorithms for 1000 person, 2% dense assignment problem, with cost range [1,1000], as a function of the number of processors used on the Encore Multimax.

the synchronous algorithms is bounded as the number of processors increases, and is much smaller than the corresponding times in Fig. 7. In contrast, the average wait time for the asynchronous algorithm grows almost linearly with the number of processors and is larger than the corresponding time in Fig. 7! These phenomena are due to the reduced computation time for an augmenting path in the sparser (2% vs 30%) network. The main cause of average wait time in the synchronous algorithms is the variability in computation time for different augmenting paths computed synchronously by each processor. Thus, a reduction in the computation time of each augmenting path correspondingly reduces the average wait time of each processor; as Fig. 9 indicates, the average wait times of the synchronous algorithms are now smaller than those of the asynchronous algorithm when six or more processors are used.

The growth of the average wait time of the asynchronous algorithm is due to the implementation of the algorithm on the Encore Multimax and could be qualitatively different in another parallel architecture. As Fig. 5 illustrates, a maximum of one processor can be either reading or modifying the assignment-price pair at any one time. The number of copies which must be made by the asynchronous algorithm increases with the number of processors, and each copy must be made in a sequential manner. In contrast, the synchronous algorithms need only make a single copy of the assignment-price pair (into shared memory); thus, the synchronous algorithms are more efficient when the time required to copy the assignment-price pairs is significant compared to the total computation time of the algorithm. This is the case for the 2% dense problem, where the time required for synchronization of the read/write operations is nearly 30% of the overall computation time.

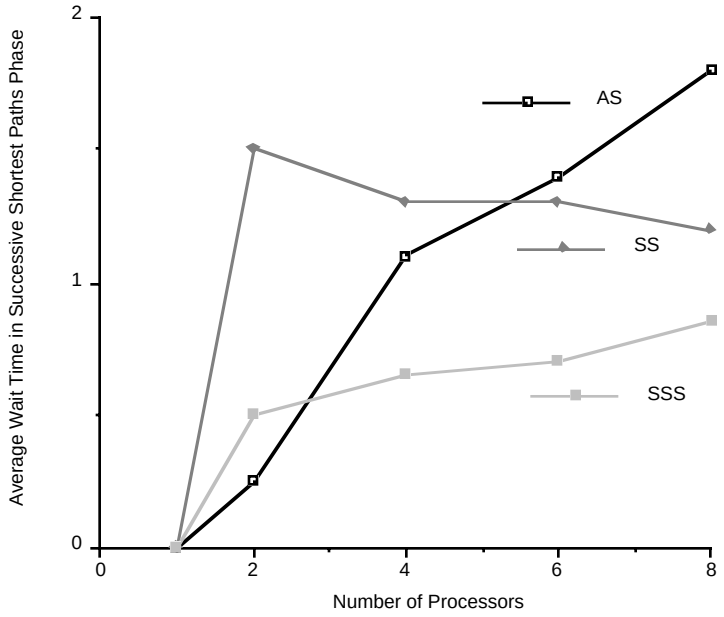

Figure 9: Average wait time per processor for 1000 person, 2% dense assignment problem, with cost range [1,1000], as a function of the number of processors used on the Encore Multimax.

REFERENCES

[BMP89] Balas, E., Miller, D., Pekny, J., and Toth, P., " A Parallel Shortest Path Algorithm for the Assignment Problem", Management Science Report MSRR 552, Carnegie Mellon Univ., Pittsburgh, PA, April 1989.

[BeC89] Bertsekas, D. P., and Castañon, D. A., "Parallel Synchronous and Asynchronous Implementations of the Auction Algorithm", Alphatech Report, Nov. 1989, submitted for publication.

[BeT89] Bertsekas, D. P., and Tsitsiklis, J. N., "Parallel and Distributed Computation: Numerical Methods", Prentice-Hall, Englewood Cliffs, N. J., 1989.

[Ber81] Bertsekas, D. P., "A New Algorithm for the Assignment Problem", Math. Programming, Vol. 21, 1981, pp. 152-171.

[CMT88] Carpaneto, G., Martello, S., and Toth, P., "Algorithms and Codes for the Assignment Problem", Annals of Operations Research, Vol. 13, 1988, pp. 193-223.

[Der85] Derigs, U., "The Shortest Augmenting Path Method for Solving Assignment Problems – Motivation and Computational Experience", Annals of Operations Research, Vol. 4, 1985, pp. 57-102.

[Eng82] Engquist, M., "A Successive Shortest Path algorithm for the Assignment Problem", INFOR, Vol. 20, 1982, pp. 370-384.

[GGK82] Glover, F., Glover, R., and Klingman, D., "Threshold Assignment Algorithm", Center for Business Decision Analysis Report CBDA 107, Graduate School of Business, Univ. of Texas at Austin, Sept. 1982.

[Hal56] Hall, M., Jr., "An Algorithm for Distinct Representatives", Amer. Math. Monthly, Vol. 51, 1956, pp. 716-717.

[JoV87] Jonker, R., and Volgenant, A., "A Shortest Augmenting Path Algorithm for Dense and Sparse Linear Assignment Problems", Computing, Vol. 38, 1987, pp. 325-340.

[Kuh55] Kuhn, H. W., "The Hungarian Method for the Assignment Problem", Naval Research Logistics Quarterly, Vol. 2, 1955, pp. 83-97.

[Law76] Lawler, E., "Combinatorial Optimization: Networks and Matroids", Holt, Rinehart and Winston, New York, 1976.

[McG83] McGinnis, L. F., "Implementation and Testing of a Primal-Dual Algorithm for the Assignment Problem", Operations Research J., Vol. 31, 1983, pp. 277-291.

[PaS82] Papadimitriou, C. H., and Steiglitz, K., Combinatorial Optimization: Algorithms and Complexity, Prentice-Hall, Englewood Cliffs, N. J., 1982.

[Roc84] Rockafellar, R. T., Network Flows and Monotropic Programming, Wiley-Interscience, N. Y., 1984.