Computational Optimization and Applications, 2, (1993), 317–336
© 1994 Kluwer Academic Publishers. Manufactured in The Netherlands.

Parallel Primal-Dual Methods for the Minimum Cost Flow Problem*

DIMITRI P. BERTSEKAS
Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139

DAVID A. CASTAÑON
Department of Electrical, Computer and Systems Engineering, Boston University, Boston, MA 02215

Received September 17, 1992; Revised April 21, 1993

Abstract. In this paper we discuss the parallel asynchronous implementation of the classical primal-dual method for solving the linear minimum cost network flow problem. Multiple augmentations and price rises are simultaneously attempted starting from several nodes with possibly outdated price and flow information. The results are then merged asynchronously subject to rather weak compatibility conditions. We show that this algorithm is valid, terminating finitely to an optimal solution. We also present computational results using an Encore MULTIMAX that illustrate the speedup that can be obtained by parallel implementation.

Keywords: Optimization, network programming, primal-dual, transshipment

1. Introduction

Consider a directed graph with node set $\mathcal{N}$ and arc set $\mathcal{A}$. Each arc (i, j) has a cost coefficient a_{ij}. We denote by f_{ij} the flow of an arc (i, j). The minimum cost flow (or transshipment) problem is

$$\text{minimize} \quad \sum_{(i,j)\in\mathcal{A}} a_{ij} f_{ij} \tag{LNF}$$

subject to

$$\sum_{\{j|(i,j)\in\mathcal{A}\}} f_{ij} - \sum_{\{j|(j,i)\in\mathcal{A}\}} f_{ji} = s_i, \quad \forall\, i \in \mathcal{N}, \tag{1}$$

$$b_{ij} \le f_{ij} \le c_{ij}, \quad \forall\, (i, j) \in \mathcal{A}, \tag{2}$$

where a_{ij}, b_{ij}, c_{ij}, and s_i are given integers. We assume that there exists at most one arc in each direction between any pair of nodes, but this assumption is made for notational convenience and can be easily dispensed with.

*This work supported in part by the BM/C3 Technology branch of the United States Army Strategic Defense Command.

A classical and still frequently used method for solving this problem is the primal-dual method due to Ford and Fulkerson [12, 13]. The basic idea is to maintain a price for each node and a flow for each arc, which satisfy complementary slackness. The method makes progress toward primal feasibility by successive augmentations along paths with certain properties and by making price changes to facilitate the creation of paths with such properties (see the description in the next section). The paths and the corresponding price adjustments can also be obtained by a shortest path computation (see the next section). The search for the augmenting path may be initiated from a single node or from multiple (or all) nodes with positive surplus [1, 4]. The method is also known as the "sequential shortest path method," and it is closely related to an algorithm of Busaker and Gowen [11], which also involves augmentations along certain shortest paths.

In this paper we propose parallel asynchronous versions of the primal-dual method where several augmenting paths are simultaneously constructed, each starting from a different node. This is the first proposal for a parallel (synchronous or asynchronous) primal-dual method for the transshipment problem (other than the obvious suggestions of parallelizing the algebra of the serial version; see [7] for a recent survey of parallel algorithms for network optimization, which contains an extensive reference list). Our proposal has been motivated by the synchronous parallel sequential shortest path algorithm introduced by Balas et al. [2] for the case of an assignment problem. They have shown that if the augmenting paths are pairwise disjoint, they can all be used to modify the current flow; to preserve complementary slackness, the node prices should be modified according to the "max-rule," that is, they should be raised to the maximum of the levels that would result from each individual shortest path calculation. In [5], we have shown the validity of an asynchronous parallel implementation of the Hungarian method, which is an extension of the synchronous parallel Hungarian method of Balas et al. [2]. The potential advantage of asynchronous algorithms is that they often work faster than their synchronous counterparts because they are not penalized by synchronization delays (see [6] and also [9] for an extensive discussion of related issues). In particular, computational experiments with assignment problems on the Encore Multimax shared-memory multiprocessor [5] show that asynchronism often results in faster execution.

In addition to showing the finite termination of our parallel asynchronous primal-dual method to an optimal solution, we discuss combinations of the primal-dual method with single node relaxation (coordinate ascent) iterations, and we similarly show that the combined algorithms work correctly in a parallel asynchronous context. Our results can be used to develop parallel versions of efficient minimum cost network optimization codes such as the RELAX algorithm of [8].

Note that it is by no means obvious why the max-rule works in a synchronous setting and, *a fortiori,* in an asynchronous setting. For this reason the proofs of algorithmic validity of [2, 5] for the case of the assignment problem have been challenging and complicated. Similarly, our finite termination proof for the

minimum cost flow problem is long and nontrivial.

In the next section we describe synchronous and asynchronous parallel versions of the primal-dual algorithm and in Section 3 we prove their validity. The primal-dual method can be substantially accelerated by combining it with single node relaxation iterations of the type introduced in [3]. In Section 4 we show how such combinations can be implemented in a parallel asynchronous setting. Finally, in Section 5 we briefly discuss both synchronous and asynchronous implementations on shared-memory architectures, and discuss computational results obtained on an Encore MULTIMAX. The results illustrate the potential advantages of asynchronous computation for these methods.

2. The parallel asynchronous primal-dual method

We introduce some terminology and notation. We denote by f the vector with elements f_{ij}, $(i, j) \in \mathcal{A}$. We refer to b_{ij} and c_{ij}, and the interval $[b_{ij}, c_{ij}]$ as the *flow bounds* and the *feasible flow range* of arc (i, j), respectively. We refer to s_i as the *supply* of node i. We refer to the constraints (1) and (2) as the *conservation of flow constraints* and the *capacity constraints*, respectively. A flow vector satisfying both of these constraints is called *feasible*, and if it satisfies just the capacity constraints, it is called *capacity-feasible*. If there exists at least one feasible flow vector, problem (LNF) is called *feasible* and otherwise it is called *infeasible*. For a given flow vector f, we define the *surplus* of node i by

$$g_i = \sum_{\{j|(j,i)\in\mathcal{A}\}} f_{ji} - \sum_{\{j|(i,j)\in\mathcal{A}\}} f_{ij} + s_i. \tag{3}$$

We introduce a dual variable p_i for each node i, also referred to as the *price of node i*. A flow-price vector pair (f, p) is said to satisfy the *complementary slackness* conditions (CS for short) if f is capacity-feasible and

$$f_{ij} < c_{ij} \;\Rightarrow\; p_i \le a_{ij} + p_j \quad \forall\ (i, j) \in \mathcal{A}, \tag{4a}$$

$$b_{ij} < f_{ij} \;\Rightarrow\; p_i \ge a_{ij} + p_j \quad \forall\ (i, j) \in \mathcal{A}. \tag{4b}$$

For a pair (f, p), feasibility of f and CS are the necessary and sufficient conditions for f to be optimal and p to be an optimal solution of a certain dual problem (see e.g., [16] or [9]).

The primal-dual method maintains a pair (f, p) satisfying CS, such that f is capacity-feasible. The method makes progress towards optimality by reducing the total absolute surplus $\sum_{i\in\mathcal{N}} |g_i|$ by an integer amount at each iteration, as we now describe.

For a given capacity-feasible f, an *unblocked path* P (with respect to f) is a path $(i_1, i_2, \ldots, i_k)$ such that for each $m = 1, \ldots, k-1$, either (i_m, i_{m+1}) is an arc with $f_{i_m i_{m+1}} < c_{i_m i_{m+1}}$ (called a *forward* arc) or (i_{m+1}, i_m) is an arc with $b_{i_{m+1} i_m} < f_{i_{m+1}, i_m}$ (called a *backward* arc). We denote by P^+ and P^- the sets of

forward and backward arcs of P, respectively. The unblocked path P is said to be an *augmenting path* if

$$g_{i_1} > 0, \quad g_{i_k} < 0.$$

An *augmentation* along an augmenting path P consists of increasing the flow of the arcs in P^+ and decreasing the flow of the arcs in P^- by the common positive increment δ given by

$$\delta = \min\Big\{g_{i_1}, -g_{i_k}, \{c_{mn} - f_{mn} \mid (m, n) \in P^+\}, \\ \{f_{mn} - b_{mn} \mid (m, n) \in P^-\}\Big\}. \tag{5}$$

Given a price vector p, the *reduced cost* of arc (i, j) is given by

$$r_{ij} = a_{ij} + p_j - p_i. \tag{6}$$

If (f, p) is a pair satisfying the CS condition (4) and P is an unblocked path with respect to f, the *cost length* of P is defined by

$$C(P) = \sum_{(i,j)\in P^+} a_{ij} - \sum_{(i,j)\in P^-} a_{ij} \tag{7}$$

and the *reduced cost length* of P is defined by

$$R(p, P) = \sum_{(i,j)\in P^+} r_{ij} - \sum_{(i,j)\in P^-} r_{ij}. \tag{8}$$

Note that by CS, we have $r_{ij} \geq 0$ for all $(i, j) \in P^+$ and $r_{ij} \leq 0$ for all $(i, j) \in P^-$, so $R(p, P) \geq 0$. For a pair of nodes i and j, let $\mathcal{P}_{ij}(f)$ be the set of unblocked paths starting at i and ending at j, and let

$$v_{ij}(f, p) = \begin{cases} \min_{P\in\mathcal{P}_{ij}(f)} R(p, P) & \text{if } \mathcal{P}_{ij}(f) \text{ is nonempty} \\ \infty & \text{otherwise.} \end{cases} \tag{9}$$

If there exists at least one node j with $g_j < 0$, the *distance* of i is defined by

$$d_i = \begin{cases} \min_{\{j\mid g_j<0\}} v_{ij}(f, p) & \text{if } g_i \geq 0 \\ 0 & \text{otherwise,} \end{cases} \tag{10}$$

and, otherwise, the distance d_i is defined to be ∞. It is well known that if the problem is feasible, we have $d_i < \infty$ for all i with $g_i > 0$, that is, there exists an augmenting path starting at each node that has positive surplus.

The typical primal-dual iteration starts with a pair (f, p) satisfying CS and generates another pair $(\bar{f}, \bar{p})$ satisfying CS as follows.

Typical iteration of the serial primal-dual method

Choose a node i with $g_i > 0$. [If no such node can be found, the algorithm terminates. There are then two possibilities: (1) $g_i = 0$ for all i, in which case f is optimal since it is feasible and satisfies CS together with p; (2) $g_i < 0$ for some i, in which case problem (LNF) is infeasible.] If $d_i = \infty$ the algorithm terminates, since then there is no augmenting path from the positive surplus node i to any negative surplus node, and the problem is infeasible. If $d_i < \infty$, let $\overline{j}$ And $\overline{P}$ be the minimizing node with $g_{\overline{j}} < 0$ and corresponding augmenting path in the definition of the distance d_i [cf. (9), (10)], that is,

$$\overline{j} = \arg \min_{\{j \mid g_j < 0\}} v_{ij}(f, p), \tag{11}$$

$$\overline{P} = \arg \min_{P \in \mathcal{P}_{i\overline{j}}(f)} R(p, P). \tag{12}$$

Change the node prices according to

$$\overline{p}_j = p_j + \max\{0, d_i - v_{ij}(f, p)\}, \quad \forall\, j \in \mathcal{N}, \tag{13}$$

and perform an augmentation along the path $\overline{P}$, obtaining a new flow vector $\overline{f}$.

We note that the primal-dual iteration can be executed by a shortest path computation. To see this, consider the *residual graph,* obtained from the original by assigning length r_{ij} to each arc (i, j) with $f_{ij} = c_{ij}$, by replacing each arc (i, j) with $f_{ij} = c_{ij}$ by an arc (j, i) with length $-r_{ij}$, and by replacing each arc (i, j) with $b_{ij} < f_{ij} < c_{ij}$ with two arcs (i, j) and (j, i) with length zero [the reduced cost of (i, j), cf. the CS condition (4)]. Then the augmenting path $\overline{P}$ is a shortest path in the residual graph, over all paths starting at the node i and ending at a node j with $g_j < 0$. Note that by the CS condition, all arc lengths are nonnegative in the residual graph, so Dijkstra's method can be used for the shortest path computation.

The results of the following proposition are well known (see e.g. [1, 4, 15, 16]) and will be used in what follows:

PROPOSITION 1. *If problem (LNF) is feasible, then a node $\overline{j}$ and an augmenting path $\overline{P}$ satisfying* (11) *and* (12) *exist. Furthermore, if $(\overline{f}, \overline{p})$ is a pair obtained by executing a primal-dual iteration on a pair (f, p) satisfying CS, the following hold:*

(a) If f consists of integer flows, the same is true for $\overline{f}$.
(b) $(f, \overline{p})$ and $(\overline{f}, \overline{p})$ satisfy CS.
(c) Let $\overline{P}$ be the augmenting path of the iteration. Then

$$R(\overline{p}, \overline{P}) = 0,$$

that is, all arcs of $\overline{P}$ have zero reduced cost with respect to $\overline{p}$.

(d) $\bar{p}_j = p_j$ *for all* j *with* $g_j < 0$.

By Proposition 1, if initially f is integer and the pair (f, p) satisfies CS, the same is true after all subsequent iterations. Then at each iteration, the total absolute surplus $\sum_{i \in \mathcal{N}} |g_i|$ will be reduced by the positive integer 2δ, where δ is the augmentation increment given by (5). Thus only a finite number of reductions of $\sum_{i \in \mathcal{N}} |g_i|$ can occur, implying that the algorithm must terminate in a finite number of iterations if the problem is feasible.

We now introduce a parallel synchronous version of the primal-dual algorithm. To simplify the statement of this and the subsequent asynchronous algorithm, *we assume that the problem is feasible;* as in the serial version, infeasibility can be detected when no augmenting path can be constructed starting at some positive surplus node, or when there is no node with positive surplus, but there is a node with negative surplus.

The algorithm terminates when all nodes have zero surplus. Each iteration starts with a pair (f, p) satisfying CS. Several augmenting paths are constructed in parallel, and these paths are used to generate another pair $(\bar{f}, \bar{p})$ as follows.

Typical iteration of parallel synchronous primal-dual method

Choose a subset $I = \{i_1, \ldots, i_m\}$ of nodes with positive surplus. For each i_n, $n = 1, \ldots m$, let $\bar{p}(n)$ and $\bar{P}(n)$ be the price vector and augmenting path obtained by executing a primal-dual iteration starting at i_n, and using the pair (f, p). Then generate sequentially the pairs $(f(n), p(n))$, $n = 1, \ldots, m$, as follows, starting with $(f(0), p(0)) = (f, p)$:
For $n = 0, \ldots, m-1$, if $\bar{P}(n+1)$ is an augmenting path with respect to $f(n)$, obtain $f(n+1)$ by augmenting $f(n)$ along $\bar{P}(n+1)$, and set

$$p_j(n+1) = \max\{p_j(n), \bar{p}_j(n)\}, \quad \forall\, j \in \mathcal{N}.$$

Otherwise set

$$f(n+1) = f(n), \quad p(n+1) = p(n).$$

The pair $(\bar{f}, \bar{p})$ generated by the iteration is

$$\bar{f} = f(m), \quad \bar{p} = p(m).$$

The preceding algorithm can be parallelized by using multiple processors to compute the augmenting paths of an iteration in parallel. On the other hand the algorithm is synchronous in that iterations have clear "boundaries." In particular, all augmenting paths generated in the same iteration are computed on the basis of the same pair (f, p). Thus, it is necessary to synchronize the parallel processors at the beginning of each iteration, with an attendant synchronization penalty.

The parallel asynchronous primal-dual algorithm tries to reduce the synchronization penalty by "blurring" the boundaries between iterations and by allowing processors to compute augmenting paths using pairs (f, p) which are out of date.

To describe the parallel asynchronous algorithm, let us denote the flow-price pair at the times

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

by $(f(k), p(k))$. [In a practical setting, the times k represent "event times," that is, times at which an attempt is made to modify the pair (f, p) through an iteration.] We require that the initial pair $(f(1), p(1))$ satisfies CS. The algorithm terminates when during an iteration, a feasible flow is obtained.

kth iteration of parallel asynchronous primal-dual method

A primal-dual iteration is performed on a pair $(f(\tau_k), p(\tau_k))$, where τ_k is a positive integer with $\tau_k \leq k$, to produce a pair $(\bar{f}(k), \bar{p}(k))$ and an augmenting path $\bar{P}_k$. The iteration (and the path $\bar{P}_k$) is said to be *incompatible* if $\bar{P}_k$ is not an augmenting path with respect to $f(k)$; in this case we discard the results of the iteration, that is, we set

$$f(k+1) = f(k), \quad p(k+1) = p(k).$$

Otherwise, we say that the iteration (and the path $\bar{P}_k$) is *compatible,* we obtain $f(k+1)$ from $f(k)$ by augmenting $f(k)$ along $\bar{P}_k$, and we set

$$p_j(k+1) = \max\{p_j(k), \bar{p}_j(k)\}, \quad \forall\, j \in \mathcal{N}. \tag{14}$$

We note that the definition of the asynchronous algorithm is not yet rigorous, because we have not yet proved that $(f(k), p(k))$ satisfies CS at all times prior to termination, so that a primal-dual iteration can be performed. This will be shown in the next section.

The implementation of the asynchronous algorithm in a parallel shared memory machine is quite straightforward. The main idea is to maintain a "master" copy of the current flow-price pair in the shared memory; this is the pair $(f(k), p(k))$ in the preceding mathematical description of the algorithm. To execute an iteration, a processor copies from the shared memory the current master flow-price pair; at the start of this copy operation the master pair is locked, so no other processor can modify it, and at the end of the operation the master pair is unlocked. The processor performs a primal-dual iteration using the copy obtained, and then locks again 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 flow-price pair. The processor then unlocks the master pair, possibly after retaining a copy to use at a subsequent iteration. The times

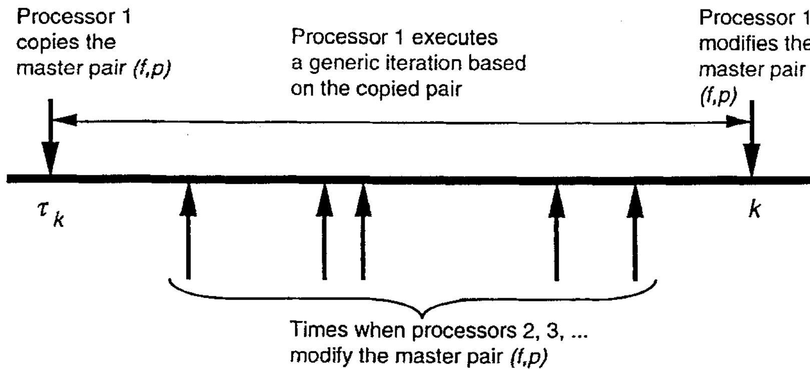

Figure 1. Operation of the asynchronous algorithm in a shared-memory machine. A processor copies the master flow-price pair at time τ_k, executes between times τ_k and k a generic iteration using the copy, and modifies accordingly the master flow-price pair at time k. Other processors may have modified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

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. This implementation is similar to the one of our asynchronous Hungarian algorithm for the assignment problem described in [5].

We finally note that any sequence of flow-price pairs generated by the synchronous parallel algorithm can also be viewed as a sequence $(f(k), p(k))$ generated by the asynchronous version. In particular, in a synchronous algorithm, suppose that m processors participate in a given iteration, copy the current flow-price pair (f, p) at a common time corresponding to a synchronization point, and update the master copy of the flow-price pair to $(\overline{f}, \overline{p})$ at a subsequent common time corresponding to another synchronization point. Let $(f(k+n), p(k+n))$, $n = 1, \ldots, m$, be the successive updates of the master copy resulting from this synchronous iteration. We may view these updates as also generated by the asynchronous algorithm, with

$$(f(k), p(k)) = (f, p), \quad (f(k+m), p(k+m)) = (\overline{f}, \overline{p})$$
$$\tau_{k+n} = k, \quad \forall n = 0, \ldots, m-1.$$

Thus, our subsequent proof of validity of the asynchronous algorithm applies also to the synchronous version.

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.

LEMMA 1. *Assume that* (f, p) *satisfies CS. Let* $P = (i_1, i_2, \ldots, i_k)$ *be an unblocked path with respect to* f. *Then*

$$p_{i_k} = p_{i_1} + R(p, P) - C(P).$$

Proof. Using (7) and (8), we have

$$\begin{aligned} R(p, P) &= \sum_{(i_m, i_{m+1}) \in P^+} (a_{i_m i_{m+1}} + p_{i_{m+1}} - p_{i_m}) - \sum_{(i_{m+1}, i_m) \in P^-} (a_{i_{m+1} i_m} + p_{i_m} - p_{i_{m+1}}) \\ &= C(P) + \sum_{m=1}^{k-1} (p_{i_{m+1}} - p_{i_m}) \\ &= C(P) + p_{i_k} - p_{i_1}, \end{aligned}$$

which yields the desired result. □

LEMMA 2. *Let* $g_j(k)$ *denote the surplus of node* j *corresponding to* $f(k)$. *For all nodes* j *such that* $g_j(k) < 0$, *we have* $p_j(k+1) = p_j(t)$ *for all* $t \leq k$.

Proof. By the nature of augmentations, we have $g_j(t) \leq g_j(t+1) \leq 0$ if $g_j(t) < 0$. Therefore, the hypothesis implies that $g_j(t) < 0$ for all $t \leq k$ and the result follows from (14) and Proposition 1(d). □

LEMMA 3. *Let* $k \geq 1$ *be given and assume that* $(f(t), p(t))$ *satisfies CS for all* $t \leq k$. *Then:*

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

$$\overline{p}_j(t) \leq p_j(\tau_t) + d_j(\tau_t). \tag{15}$$

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

$$p_j(t) + d_j(t) = \overline{p}_j(t) = p_j(t+1). \tag{16}$$

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

$$p_j(t) + d_j(t) \leq p_j(t+1) + d_j(t+1). \tag{17}$$

Proof. (a) If j is such that $g_j(\tau_t) < 0$, by Proposition 1(d), we have $\overline{p}_j(t) = p_j(\tau_t)$ and $d_j(t) = 0$, so the result holds. Thus, assume that $g_j(\tau_t) \geq 0$. Consider any unblocked path P [with respect to $f(\tau_t)$] from j to a node $\overline{j}$ with $g_{\overline{j}}(\tau_t) < 0$. By Lemma 1, we have

$$p_{\overline{j}}(\tau_t) = p_j(\tau_t) + R(p(\tau_t), P) - C(P),$$
$$\overline{p}_{\overline{j}}(t) = \overline{p}_j(t) + R(\overline{p}(t), P) - C(P),$$

where the second equality holds because by Proposition 1(b), the pair $(f(\tau_t), \bar{p})$ satisfies CS and Lemma 1 applies. Since $g_{\bar{j}}(\tau_t) < 0$, we have $p_{\bar{j}}(\tau_t) = \bar{p}_{\bar{j}}(t)$ and it follows that

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

Taking the minimum of $R(p(\tau_t), P)$ over all unblocked paths P, starting at j and ending at nodes $\bar{j}$ with $g_{\bar{j}}(\tau_t) < 0$, the result follows.

(b), (c) We prove parts (b) and (c) simultaneously, by first proving a weaker version of part (b) [see (18) below], then proving part (c), and then completing the proof of part (b). Specifically, we will first show that for $t \leq k$, if $f(t+1) \neq f(t)$ and j is a node which belongs to the corresponding augmenting path, then we have

$$p_j(t) + d_j(t) \leq \bar{p}_j(t) = p_j(t+1). \tag{18}$$

Indeed, if $g_j(t) < 0$, (18) holds since, by Lemma 2, we have $p_j(t) = \bar{p}_j(t)$ and $d_j(t) = 0$. Assume that $g_j(t) \geq 0$. Let the augmenting path of iteration t end at node $\bar{j}$, and let P be the portion of this path that starts at j and ends at $\bar{j}$. We have, using Lemma 1, and Propositions 1(b) and 1(c),

$$\bar{p}_{\bar{j}}(t) = \bar{p}_j(t) - C(P),$$
$$p_{\bar{j}}(t) = p_j(t) - C(P) + R(p(t), P).$$

Since $g_{\bar{j}}(t) < 0$, by Lemma 2, we have $\bar{p}_{\bar{j}}(t) = p_{\bar{j}}(t)$, and we obtain

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

showing the left-hand side of (18). Since $d_j(t) \geq 0$, this yields $p_j(t) \leq \bar{p}_j(t)$, so $\bar{p}_j(t) = \max\{p_j(t), \bar{p}_j(t)\} = p_j(t+1)$, completing the proof of (18).

We now prove part (c), making use of (18). Let us fix node j and assume without loss of generality that iteration t is compatible [otherwise (16) and (17) hold trivially]. If $g_j(t+1) < 0$, we have $p_j(t) = p_j(t+1)$ and $d_j(t) = d_j(t+1) = 0$, so the desired relation (17) holds. Thus, assume that $g_j(t+1) \geq 0$, and let $P = (j, j_1, \ldots, j_k, \bar{j})$ be an unblocked path with respect to $f(t+1)$, which is such that $g_{\bar{j}}(t+1) < 0$ and

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

Let $\bar{P}$ denote the augmenting path of iteration t. Then there are three possibilities: (1) $P \cap \bar{P} = \emptyset$; (2) $j \in \bar{P}$; or (3) $P \cap \bar{P} \neq \emptyset$ and $j \notin \bar{P}$. We prove (17) separately for each of these cases:

(1) In this case, the nodes $j, j_1, \ldots, j_k$ do not belong to $\bar{P}$, and the path P is also unblocked with respect to $f(t)$. By using Lemma 1, it follows that

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

and

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

Since $g_{\bar{j}}(t+1) < 0$, we have $p_{\bar{j}}(t+1) = p_{\bar{j}}(t)$, so the preceding equations yield

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

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

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

and (17) is proved.

(2) In this case, by (18), we have

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

and (17) is proved.

(3) In this case, there is a node j_m, $m \in \{1, \ldots, k\}$, which belongs to $\overline{P}$, and is such that j and $j_1, \ldots, j_{m-1}$ do not belong to $\overline{P}$. Consider the following paths, which are unblocked with respect to $f(t+1)$:

$$P' = (j, j_1, \ldots, j_{m-1}, j_m),$$
$$P'' = (j_m, j_{m+1}, \ldots, j_k, \bar{j}).$$

By using Lemma 1, we have

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

and since by (18), $p_{j_m}(t+1) - p_{j_m}(t) \geq d_{j_m}(t)$, we obtain

$$R(p(t+1), P') + p_j(t+1) \geq R(p(t), P') + p_j(t) + d_{j_m}(t). \tag{19}$$

On the other hand, we have

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

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

$$R(p(t+1), P) \geq R(p(t+1), P'). \tag{20}$$

Combining (19) and (20), we see that

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

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

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

and the proof of (17) is complete.

To complete the proof of part (b), we note that by using (15) and (17), we obtain

$$\bar{p}_j(t) \le p_j(\tau_t) + d_j(\tau_t) \le p_j(t) + d_j(t),$$

which combined with (18) yields the desired (16). □

We can now prove that the asynchronous algorithm preserves CS.

PROPOSITION 2. *All pairs $(f(k), p(k))$ generated by the asynchronous algorithm satisfy CS.*

Proof. By induction. Suppose all iterations up to the kth maintain CS, let the kth iteration be compatible, and let $\overline{P}_k$ be the corresponding augmenting path. We will show that the pair $(f(k+1), p(k+1))$ satisfies CS. For any arc (i, j) there are four possibilities:

(1) $f_{ij}(k+1) \neq f_{ij}(k)$. In this case by Proposition 1(c), we have $\bar{p}_i(k) = a_{ij} + \bar{p}_j(k)$. Since i and j belong to $\overline{P}_k$, by Lemma 3(b), we have $p_i(k+1) = \bar{p}_i(k)$ and $p_j(k+1) = \bar{p}_j(k)$, implying that $p_i(k+1) = a_{ij} + p_j(k+1)$, so the CS condition is satisfied for arc (i, j).

(2) $f_{ij}(k+1) = f_{ij}(k) < c_{ij}$. In this case, by the CS property (cf. the induction hypothesis), we have $p_i(k) \le a_{ij} + p_j(k)$. If $p_i(k) \ge \bar{p}_i(k)$, it follows from (14) that

$$p_i(k+1) = p_i(k) \le a_{ij} + p_j(k) \le a_{ij} + p_j(k+1),$$

so the CS condition is satisfied for arc (i, j). Assume therefore that $p_i(k) < \bar{p}_i(k)$. If $f_{ij}(\tau_k) < c_{ij}$, then since by Proposition 1(b), $(f, \bar{p})$ satisfies CS, we have $\bar{p}_i(k) \le a_{ij} + \bar{p}_j(k)$, from which $p_i(k+1) \le a_{ij} + \bar{p}_j(k) \le a_{ij} + p_j(k+1)$, and again the CS condition is satisfied for arc (i, j). The last remaining possibility [under the assumption $f_{ij}(k+1) = f_{ij}(k) < c_{ij}$] is that $f_{ij}(\tau_k) = c_{ij}$ and $p_i(k) < \bar{p}_i(k)$. We will show that this can't happen by assuming that it does and then arriving at a contradiction. Let t_1 be the first time index such that $\tau_k < t_1 \le k$ and $f_{ij}(t_1) < c_{ij}$. Then by Lemmas 3(a) and 3(c), we have

$$\bar{p}_i(k) \le p_i(\tau_k) + d_i(\tau_k) \le p_i(t_1 - 1) + d_i(t_1 - 1),$$

while from Lemma 3(b),

$$p_i(t_1 - 1) + d_i(t_1 - 1) = p_i(t_1) \le p_i(k),$$

[since $f_{ij}(t_1) \neq f_{ij}(t_1 - 1)$ and node i belongs to the augmenting path of iteration $t_1 - 1$]. It follows that $\bar{p}_i(k) \le p_i(k)$, which contradicts the assumption $p_i(k) < \bar{p}_i(k)$, as desired. We have thus shown that the CS condition holds for arc (i, j) in case (2).

(3) $f_{ij}(k+1) = f_{ij}(k) > b_{ij}$. The proof that the CS condition is satisfied for arc (i, j) is similar as for the preceding case (2).
(4) $f_{ij}(k+1) = f_{ij}(k) = b_{ij} = c_{ij}$. In this case, the CS conditions (4) are trivially satisfied. □

Proposition 2 shows that if the asynchronous algorithm terminates, the flow-price pair obtained satisfies CS. Since the flow obtained at termination is feasible, 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 3. *If* $\lim_{k\to\infty} \tau_k = \infty$, *the asynchronous algorithm terminates. If the problem is feasible, the flow obtained at termination is optimal.*

Proof. There can be at most a finite number of compatible iterations, so if the algorithm does not terminate, all iterations after some index $\bar{k}$ are incompatible, and $f(k) = f(\bar{k})$ 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 $f(\tau_k) = f(k)$ for all k with $\tau_k \geq \bar{k}$. This contradicts the incompatibility of the kth iteration. □

4. Combination with single node relaxation iterations

Computational experiments show that in a serial setting, primal-dual methods are greatly speeded up by mixing shortest path augmentations with single node relaxation (or coordinate ascent) iterations of the type introduced in [3]. The typical single node iteration starts with a pair (f, p) satisfying CS and produces another pair $(\bar{f}, \bar{p})$ satisfying CS. It has the following form.

Single node relaxation iteration

Choose a node i with $g_i > 0$ (if no such node can be found, the algorithm terminates). Let

$$B_i^+ = \{j \mid (i, j) \in \mathcal{A},\ r_{ij} = 0,\ f_{ij} < c_{ij}\}, \tag{21}$$

$$B_i^- = \{j \mid (j, i) \in \mathcal{A},\ r_{ji} = 0,\ f_{ji} > b_{ji}\}. \tag{22}$$

Step 1: If

$$g_i \geq \sum_{j \in B_i^+} (c_{ij} - f_{ij}) + \sum_{j \in B_i^-} (f_{ji} - b_{ji}),$$

go to step 4. Otherwise, if $g_i > 0$, choose a node $j \in B_i^+$ with $g_j < 0$ and go to step 2, or choose a node $j \in B_i^-$ with $g_j < 0$ and go to step 3; if no such node can be found or if $g_i = 0$, set $\overline{f} = f$ and $\overline{p} = p$, and go to the next iteration.

Step 2 (Flow adjustment on outgoing arc): Let

$$\delta = \min\{g_i, -g_j, c_{ij} - f_{ij}\}.$$

Set

$$f_{ij} := f_{ij} + \delta, \quad g_i := g_i - \delta, \quad g_j := g_j + \delta$$

delete j from B_i^+, and go to step 1.

Step 3 (Flow adjustment on incoming arc): Let

$$\delta = \min\{g_i, -g_j, f_{ji} - b_{ji}\}.$$

Set

$$f_{ji} := f_{ji} - \delta, \quad g_i := g_i - \delta, \quad g_j := g_j + \delta$$

delete j from B_i^-, and go to step 1.

Step 4 (Increase price of i): Set

$$\begin{aligned} g_i &:= g_i - \sum_{j \in B_i^+} (c_{ij} - f_{ij}) - \sum_{j \in B_i^-} (f_{ji} - b_{ji}), \\ & \quad f_{ij} = c_{ij}, \quad \forall\, j \in B_i^+, \\ & \quad f_{ji} = b_{ji}, \quad \forall\, j \in B_i^-, \\ p_i &:= \min\{\min\{p_j + a_{ij} \mid (i, j) \in \mathcal{A},\, p_i < p_j + a_{ij}\}, \\ & \quad \min\{p_j - a_{ji} \mid (j, i) \in \mathcal{A},\, p_i < p_j - a_{ji}\}\}. \end{aligned} \tag{23}$$

If following these changes, $g_i > 0$, recalculate the sets B_i^+ and B_i^- using (21) and (22), and go to step 1; else, set $\overline{f} = f$ and $\overline{p} = p$, and go to the next iteration. [Note: If the set of arcs over which the minimum in (23) is calculated is empty, there are two possibilities: (a) $g_i > 0$, in which case it can be shown that the dual cost increases without bound along p_i, and the primal problem is infeasible; or (b) $g_i = 0$, in which case the cost stays constant along p_i; in this case, we set $\overline{f} = f$, $\overline{p} = p$, and go to the next iteration.]

It can be seen that the flow changes of the above iteration are such that the condition $g_i \geq 0$ is maintained. Furthermore, it can be shown that the pair $(\overline{f}, \overline{p})$ generated by the iteration satisfies CS. To see this, first note that steps 2 and 3 can only change flows of arcs with zero reduced cost; then observe that the flow changes in step 4 are designed to maintain CS of the arcs whose reduced

cost changes from zero to nonzero, and the price change is such that the sign of the reduced costs of all other arcs does not change from positive to negative or reversely.

A combined primal-dual/single node relaxation iteration can now be constructed. It starts with a pair (f, p) satisfying CS and produces another pair $(\bar{f}, \bar{p})$ as follows:

Combined primal-dual/relaxation iteration

Choose a node i with $g_i > 0$ (if no such node can be found, stop the algorithm). Perform a single node relaxation iteration. If as a result (f, p) is changed, terminate the iteration; otherwise, perform a primal-dual iteration starting from (f, p).

A synchronous parallel combined method can be constructed based on the above iteration. To this end, we must modify the definition of compatibility for the case where the pair $(\bar{f}(n), \bar{p}(n))$ (refer to the description of the synchronous parallel iteration in Section 2) is produced by the single node relaxation iteration. In this case, we discard the results of the iteration if

$$\bar{p}_{i_n}(n) < p_{i_n}(n),$$

where i_n is the node i used in the single node iteration. Otherwise, we say that the iteration is *compatible*, we set

$$p_i(n+1) = \begin{cases} \bar{p}_{i_n} & \text{if } i = i_n, \\ p_i(n) & \text{otherwise,} \end{cases}$$

and for all arcs (i, j), we set

$$f_{ij}(n+1) = \begin{cases} f_{ij}(n) & \text{if } i \neq i_n \text{ and } j \neq i_n, \\ \bar{f}_{ij}(n) & \text{if } i = i_n \text{ or } j = i_n, \text{ and } r_{ij}(n+1) = 0, \\ b_{ij} & \text{if } i = i_n \text{ or } j = i_n, \text{ and } r_{ij}(n+1) > 0, \\ c_{ij} & \text{if } i = i_n \text{ or } j = i_n, \text{ and } r_{ij}(n+1) < 0, \end{cases}$$

where $r_{ij}(n+1)$ is the reduced cost of arc (i, j) with respect to the price vector $p(n+1)$.

The definition of compatibility is such that the above synchronous parallel iteration preserves CS. Using this property and the monotonic increase of the node prices, it can be seen that the associated algorithm terminates finitely, assuming the problem is feasible (see [3]). A similar result can be shown for the corresponding asynchronous version of the parallel iteration.

5. Computational results

In order to illustrate the expected performance of the above parallel primal-dual minimum cost network flow algorithms, we designed a synchronous and two asynchronous parallel versions of one of the primal-dual codes developed by Bertsekas and Tseng for comparison with the RELAX code (see [8] for a description). We implemented these parallel primal-dual algorithms on a shared-memory Encore MULTIMAX and evaluated the parallel computation time for two transshipment problems as a function of the number of processors used. In this section, we briefly overview the parallel implementations and discuss the numerical results obtained.

The synchronous algorithm (SPD) operates as follows: The current flow-price pair (f, p) satisfying CS is kept in shared memory. Each iteration starts synchronously with each processor copying the current set of node prices into local memory. Each processor $n = 1, \ldots, m$ selects a different node i_n with positive surplus, and performs a primal-dual iteration to compute a shortest augmenting path (in terms of the reduced cost lengths) from node i_n to the set of nodes with negative surplus. Let $\overline{p}(n)$ and $\overline{P}(n)$ be the modified price vector (in local memory) and augmenting path obtained by processor n.

Assume without loss of generality that the m processors find their shortest augmenting paths in the order $n = 1, \ldots, m$, and let $(f(n), p(n))$ denote the flow-price pair resulting from incorporation of the results of the processor n [note that $(f(0), p(0)) = (f, p)$]. As described in Section 2, once a processor computes $\overline{p}(n)$ and $\overline{P}(n)$, it checks to see whether $\overline{P}(n)$ is a compatible augmentation based on the most recent network prices and flows $(f(n-1), p(n-1))$. During this operation, the network is locked so that only one processor (at a time) can verify the compatibility of an augmentation or modify the flow-price pair. If the augmentation is compatible, the arc flows are modified accordingly and the node prices are adjusted as described in Section 2. The processor then waits for all other processors to complete their computations before starting the next cycle of augmentations.

In our implementation on the Encore MULTIMAX, the most recent flow-price pair $(f(n), p(n))$ is also kept in shared memory. The set of nodes with positive surplus is maintained in a queue in shared memory; a lock on this queue is used in order to guarantee that a given node can be selected by only one processor. A synchronization lock on the flow-price pair $(f(n), p(n))$ is used to restrict modifications of flows of prices by more than one processor simultaneously, and a synchronization barrier is used at the end of each iteration to synchronize the next iteration.

The principal drawback of our implementation of the synchronous algorithm is the idle time spent by the processors waiting while other processors are still computing augmenting paths or modifying the pair $(f(n), p(n))$ that is kept in shared memory. Figure 2 illustrates the processor idle times in a typical iteration.

In order to reduce the idle time spent by the processors, asynchronous algo-

rithms allow processors which have finished their computations to proceed with further computation. In our asynchronous algorithms, the current flow-price pair (f, p) satisfying CS and a queue of nodes with positive surplus are also kept in shared memory. The first asynchronous algorithm (ASPD1) operates as follows: Each processor starts its computation by extracting a node from the queue of nodes with positive surplus. It then copies the flow-price pair (f, p) into local memory, and performs a primal-dual iteration to compute a shortest augmenting path and modified price vector $\overline{P}$ and $\overline{p}$. The node then checks whether this augmentation is compatible with the possibly modified flow-price pair (f, p). If the augmentation is compatible, the flows and prices are modified as described in Section 2. The processor then repeats the cycle without waiting for other processors to complete their computations.

In our implementation of ASPD1 and the Encore MULTIMAX, a lock is used to allow only one processor to either read or modify the flow-price pair (f, p) at a time. A second lock is used to allow only one processor to access the queue of positive surplus nodes at a time. The first lock can create a critical slowdown when several processors are used because a processor must wait until another processor has completely copied (f, p) before it can begin its own copy. In order to reduce this potential bottleneck, we developed a different asynchronous implementation ASPD2 using a monitor [10] instead of locks to allow several processors to copy (f, p) simultaneously, but to exclude any processors from either reading or writing (f, p) whenever another processor is currently modifying (f,p).

Table 1 shows the performance of the algorithm on the Encore MULTIMAX for two uncapacitated transshipment problems generated using the widely used NETGEN program of [14]; these problems correspond to problems NG31 and NG35 in [14]. Problem NG31 has 1,000 nodes and 4,800 arcs, with 50 sources and 50 sinks, while problem NG35 has 1,500 nodes and 5,730 arcs, with 75 sources and 75 sinks. The Encore MULTIMAX's individual processors are rated at roughly 1 MIPS each. The table contains the average time obtained over 11 different runs, as a function of the number of processors used; the standard deviation is enclosed in parenthesis. The variability of the run times for different runs is due to randomness in the order of completion of the computations of the individual processors, which can lead to differences as to which augmentations are found compatible.

Table 1 clearly illustrates the superiority of the asynchronous implementations over the synchronous implementations, even on a shared-memory multiprocessor where synchronization is easily achieved. The ASPD2 implementation is superior for a larger number of processors because it allows simultaneous reading of the flow-price pair (f, p); for a small number of processors, the ASPD1 algorithm is slightly faster because of its simpler synchronization logic. Note also that the speedups achieved are larger for the larger NG35 problem, because of the greater difficulty in computing augmenting paths, which increases the ratio of

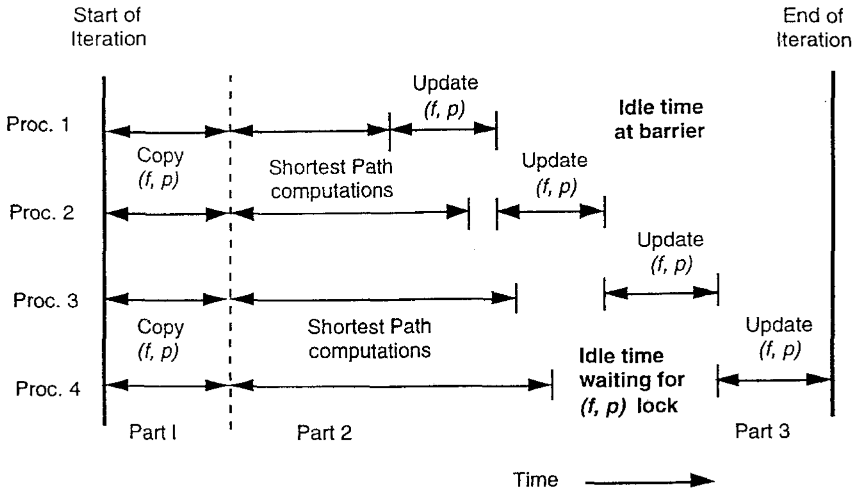

Figure 2. Timing diagram of an iteration. The computation of each processor consists of three parts, possibly separated by idle time. In the first part, all processors copy (in parallel) the master pair (f, p). In the second part, the processors calculate (in parallel) their shortest augmenting paths. In the third part, the processors update (one at a time) the master pair (f, p). The next iteration does not begin until all processors have finished all three parts.

computation time to synchronization overhead.

The results of Table 1 also indicate that the speedups achieved are limited as the number of processors increase. There are two primary reasons for this: Even in the asynchronous algorithm, there is some synchronization overhead associated with maintaining the integrity of the queue of positive surplus nodes and the flow-price pair (f, p); this overhead increases with the number of processors. Furthermore, when the algorithms are near convergence, there are very few nodes with positive surplus, so that there isn't enough parallel work for the available processors. These last few iterations are nearly sequential, and often consist of the longest augmenting paths. Similar limitations were observed in [5] in the context of parallel Hungarian algorithms for the assignment problems. For a more detailed discussion of these limiting factors, the reader is referred to [5], which reports extensive numerical experiments quantifying both the synchronization overhead and the sequential part of the computation.

Alternative parallel algorithms which significantly reduce the synchronization overhead can be designed using the theory described in Sections 2 and 3. One approach is to have each processor search for multiple augmenting paths (from different starting nodes with positive surplus) during each iteration. In this manner, the number of iterations is considerably reduced, thereby reducing the overall synchronization overhead. To make this approach efficient, the assignment of positive surplus nodes to each processor should be adaptive, depending on

Table 1. Average run times and standard deviations (in parenthesis) in seconds over 11 runs on the Encore MULTIMAX for problems NG31 and NG35 of [14]. SPD is a synchronous version, while ASPD1 and ASPD2 are asynchronous versions.

Problem	# Processors	SPD	ASPD1	ASPD2
NG31	1	23.51 (0.16)	23.15 (0.15)	24.00 (0.22)
	2	16.20 (0.15)	14.84 (0.28)	15.24 (0.66)
	3	13.94 (0.86)	13.11 (0.66)	13.45 (0.68)
	4	14.07 (0.57)	11.59 (0.50)	11.81 (0.55)
	5	14.15 (0.56)	11.74 (0.96)	11.29 (0.49)
	6	14.79 (0.92)	11.00 (0.75)	11.38 (0.36)
	7	13.35 (0.79)	11.54 (0.60)	10.19 (0.85)
	8	14.74 (0.40)	11.76 (0.60)	9.65 (0.53)
NG35	1	55.90 (0.50)	54.23 (0.71)	55.64 (0.65)
	2	40.45 (1.30)	33.15 (1.83)	33.72 (1.00)
	3	33.72 (1.56)	26.96 (0.95)	28.05 (1.56)
	4	32.21 (1.87)	24.52 (1.33)	24.29 (1.09)
	5	25.45 (1.39)	22.64 (1.22)	21.82 (0.94)
	6	25.34 (2.19)	21.46 (1.64)	20.22 (1.74)
	7	26.86 (2.03)	20.97 (0.86)	19.16 (1.34)
	8	23.70 (2.13)	20.48 (1.82)	18.40 (1.59)

the time required to find the previous augmentations. Such an algorithm was implemented and evaluated in [5] in the context of the assignment problem, yielding significant reductions in synchronization overhead.

References

1. R. K. Ahuja, T. L. Magnanti, and J. B. Orlin, "Network flows," Handbooks in Operations Research and Management Science, vol. 1, Optimization, G. L. Nemhauser, A. H. G. Rinnooy-Kan, and M. J. Todd, (eds.), North-Holland: Amsterdam, 1989, pp. 211–369.
2. E. Balas, D. Miller, J. Pekny, and P. Toth, "A parallel shortest part algorithm for the assignment problem," J. ACM, Vol. 34, pp. 985–1004, 1991.
3. D. P. Bertsekas, "A unified framework for minimum cost network flow problems," Math. Programming, pp. 125–145, 1985.
4. D. P. Bertsekas, Linear Network Optimization: Algorithms and Codes, MIT Press: Cambridge, MA, 1991.
5. D. P. Bertsekas, and D. A. Castañon, "Parallel asynchronous Hungarian methods for the assignment problem," Alphatech Report, Feb. 1990, (to appear in ORSA J. of Comput).
6. D. P. Bertsekas, and D. A. Castañon, "Parallel synchronous and asynchronous implementations of the auction algorithm," Parallel Computing, vol. 17, pp. 707–732, 1991.

7. D. P. Bertsekas, D. A. Castañon, J. Eckstein, and S. Zenios, "A survey of parallel algorithms for network optimization," Lab. for Information and Decision Systems Report P-1606, Massachusetts Institute of Technology, November 1991; to appear in Handbooks in Operations Research and Management Science, vol. 4, North-Holland: Amsterdam.
8. D. P. Bertsekas, and P. Tseng, "Relaxation methods for minimum cost ordinary and generalized network flow problems," Oper. Res. J., vol. 36, pp. 93–114, 1988.
9. D. P. Bertsekas, and J. N. Tsitsiklis, Parallel and Distributed Computation: Numerical Methods, Prentice-Hall: Englewood Cliffs, NJ, 1989.
10. J. Boyle, R. Butler, T. Disz, B. Glickfield, E. Lusk, R. Overbeek, J. Patterson, and R. Stevens, Portable Programs for Parallel Processors, Holt, Rinehart, and Winston: New York, NY, 1987.
11. R. G. Busaker, and P. J. Gowen, "A procedure for determining a family of minimal cost network flow patterns," O.R.O. Technical Report No. 15, Operational Research Office, Johns Hopkins University, Baltimore, MD, 1961.
12. L. R. Ford, Jr. and D. R. Fulkerson, "A primal-dual algorithm for the capacitated Hitchcock problem," Naval Res. Log. Q., vol. 4, pp. 47–54, 1957.
13. L. R. Ford, Jr. and D. R. Fulkerson, Flows in Networks, Princeton Univ. Press: Princeton, NJ, 1962.
14. D. Klingman, A. Napier, and J. Stutz, "NETGEN– A program for generating large scale (un) capacitated assignment, transportation, and minimum cost flow network problems," Mgmt. Sci., vol. 20, pp. 814–822, 1974.
15. C. H. Papadimitriou, and K. Steiglitz, Combinatorial Optimization: Algorithms and Complexity, Prentice-Hall: Englewood Cliffs, NJ, 1982.
16. R. T. Rockafellar, Network Flows and Monotropic Programming, Wiley-Interscience: NY, New York, 1984.