ELSEVIER

Parallel Computing 20 (1994) 1221–1247

PARALLEL COMPUTING

Parallel shortest path auction algorithms †

L.C. Polymenakos, D.P. Bertsekas *

Laboratory for Information and Decision Systems, MIT, Cambridge, MA 02139-4307, USA

Received 11 December 1992; revised 10 May 1993, 1 November 1993

Abstract

In this paper we discuss the parallel implementation of the auction algorithm for shortest path problems. We show that both the one-sided and the two-sided versions of the algorithm admit asynchronous implementations. We implemented the parallel schemes for the algorithm on a shared memory machine and tested its efficiency under various degrees of synchronization and for different types of problems. We discuss the efficiency of the parallel implementation of the many origins-one destination problem, the all origins-one destination problem, and the many origins-many destinations problem.

Keywords: Shortest path problem; Auction algorithm; Shared memory multiprocessor; Parallel implementation; Performance results

1. Introduction

In this paper we consider the problem of finding the shortest path from an origin to a destination in a directed graph. Each arc has an associated length and the objective is to find a path connecting the origin to the destination with minimum total length. The auction algorithm for this problem was first proposed in [4,5] and was studied further for parallel implementations in [12]. The algorithm maintains a price for each node and a path starting at the origin. The terminal node of the path 'bids' for neighboring nodes based on their prices and the lengths of the connecting arcs. The path is then appropriately extended or contracted. This process continues until the destination becomes the terminal node of the path, in which case the shortest path to the destination has been found.

† Work supported by NSF under Grants No. DDM-8903385 and CCR-9108058. Thanks are due for comments and assistance with the parallel asynchronous codes to David Castanon.

* Corresponding author. Email: bertsekas@lids.mit.edu

SSDI 0167-8191(94)00029-A

The algorithm is well suited for parallel implementation. In particular, instead of keeping only one path, we may have many paths starting at different origins. Furthermore, the prices set by one path will be of use to the other paths for their extensions and contractions. This is a key feature of auction-like algorithms, which makes them appealing for parallel implementation. Actually, we will show that the algorithm admits a totally asynchronous implementation whereby the bidding may be done with out-of-date price information. Such an implementation minimizes the synchronization penalty, i.e. the delay incurred when several processors synchronize in order to base their bids on up-to-date information about the prices of the nodes.

In this paper we explore various parallelization schemes and prove their validity for the many origins-one destination, and the many origins-many destinations shortest path problems. We develop algorithms which run asynchronously from both the origins and the destinations. Extensions and contractions on the various paths can happen either synchronously or asynchronously. Finally, we present running time results from implementations on a shared memory, multiple instruction, multiple data stream parallel computer, the Encore Multimax.

Our results with randomly generated test problems with no discernible special structure are encouraging, showing that the parallel auction algorithm is capable of significant speedup. While there are several algorithms of the label setting and label correcting type, which can be parallelized, there are no published experimental results with parallel implementations of these methods, and there is considerable doubt regarding their potential.

The paper is organized as follows: In the next section we provide an overview of the serial auction shortest path algorithm and in Section 3 we develop and prove the validity of the parallel asynchronous one-sided auction algorithm (running from the origins only). The line of analysis bears similarity with a corresponding analysis of the auction algorithm for the assignment problem [1]. In Section 4 we extend the one-sided asynchronous scheme to two-sided asynchronous schemes for both the many origins-one destination and the many origins-many destinations problems. In Section 5 we discuss various implementations and we report computational results with different types of graphs.

2. The auction shortest path algorithm

We assume that we have a directed graph $(\mathcal{N}, \mathcal{A})$ where $\mathcal{N}$ is the set of nodes and $\mathcal{A}$ is the set of arcs. To simplify notation, we assume that for each pair of nodes i and j there is at most one arc starting at i and ending at j; such an arc is referred to as (i, j). For convenience in stating the algorithms, we also assume that there is at least one outgoing and at least one incoming arc to each node. Each arc $(i, j) \in \mathcal{A}$ has a length a_{ij}. We introduce two special nodes s and d, referred to as the *origin* and the *destination*. The shortest path problem is to find the path of smallest length among all paths that start at s and end at d. We introduce also the following definitions:

- A *walk* is a sequence of nodes, $(i_1, i_2, \ldots, i_k)$, and a corresponding sequence of arcs, such that $(i_m, i_{m+1}) \in \mathscr{A}$ for all $m = 1, \ldots, k-1$.
- A *cycle* is a walk whose initial and final node are the same.
- A *path* is a walk with the additional property that all nodes $i_1, i_2, \ldots, i_k$ are distinct, i.e. a path does not contain any cycles.

The length of a walk is the sum of the lengths of its arcs. We assume that all cycles have strictly positive length. We note, however, that the initialization of the algorithm may be difficult if some arcs have negative lengths (see [4,5]). The methodology of this paper is thus best suited for problems with nonnegative arc lengths and positive length cycles.

The algorithm maintains at all times a path $P = (s, i_1, i_2, \ldots, i_k)$ starting at the origin. The last node on the path, i_k, is called the *terminal* node of P. The *degenerate path* $P = (s)$ may also be obtained in the course of the algorithm. We define two operations that can be performed on a path:

- A path $P = (s, i_1, i_2, \ldots, i_k)$ can be *extended* by a node $i_{k+1} \notin P$ such that $(i_k, i_{k+1}) \in \mathscr{A}$, i.e. the path becomes $P = (s, i_1, i_2, \ldots, i_k, i_{k+1})$.
- A path $P = (s, i_1, i_2, \ldots, i_k)$ that is not degenerate can be *contracted*, i.e. it becomes $P = (s, i_1, i_2, \ldots, i_{k-1})$.

In addition to the path, the algorithm maintains a *price* for each node i in the network, which we shall denote by p_i. Let us denote by p the vector of prices p_i. We say that a path-price pair (P, p) satisfies *complementary slackness* (CS) if the following relations hold:

$$p_i \le a_{ij} + p_j, \quad \forall (i, j) \in \mathscr{A}, \tag{2.1a}$$

$$p_i = a_{ij} + p_j, \quad \forall (i, j) \in P. \tag{2.1b}$$

An important property is that if a path-price pair (P, p) satisfies CS, then the portion of P between node s and any node $i \in P$ is a shortest path from s to i. Furthermore, $p_s - p_i$ is the corresponding shortest distance. To see this, let the path be $P = (s, i_1, i_2, \ldots, i_k)$. Then from Eq. (2.1b) we have that

$$p_s = a_{si_1} + p_{i_1}, \quad p_{i_1} = a_{i_1 i_2} + p_{i_2}, \ldots, p_{i_{k-1}} = a_{i_{k-1} i_k} + p_{i_k}.$$

By summing the first n ($\le k$) equations, we obtain that $p_s - p_{i_n}$ is the length of the portion of P between s and i_n. For every other path $P = (s, i'_1, i'_2, \ldots, i'_{n-1}, i_n)$ connecting s and i_n, Eq. (2.1a) holds. Thus the length of the portion of P between s and i_n will be greater than or equal to $p_s - p_{i_n}$.

Let us now proceed to describe the algorithm. We initialize the algorithm by picking (P, p) to be any pair satisfying CS such as, for example,

$$P = (s), \quad p_i = 0, \quad \forall i$$

when all arc lengths are positive. As discussed in [4], we may pick any price vector and then run a preprocessing algorithm in order to ensure that CS will hold for the resulting price vector and the degenerate path $P = (s)$. The auction shortest path algorithm maintains a path-price pair (P, p) which satisfies CS. At each iteration, the path P is either extended by adding a new node or is contracted by deleting its terminal node. In the latter case, the price of the terminal node is strictly

increased. In the case of the degenerate path, $P = (s)$, the path is either extended, or left unchanged with the price p_s being strictly increased. The iteration is as follows:

Typical iteration. Let i be the terminal node of P.

- **Step 0 (Scanning of successor nodes).** If

$$p_i < \min_{\{j \mid (i,j) \in \mathscr{A}\}} \{a_{ij} + p_j\}, \tag{2.2}$$

go to Step 1; else go to Step 2.

- **Step 1 (Contract path).** Set

$$p_i := \min_{\{j \mid (i,j) \in \mathscr{A}\}} \{a_{ij} + p_j\}, \tag{2.3}$$

and if $i \neq s$, contract P.

- **Step 2 (Extend path).** Extend P by node j_i where

$$j_i = \arg \min_{\{j \mid (i,j) \in \mathscr{A}\}} \{a_{ij} + p_j\}. \tag{2.4}$$

If j_i is the destination d, stop; P is the desired shortest path.

The shortest path algorithm proceeds by performing such iterations until the destination node d becomes terminal node of the path. Note that after an extension at Step 2, P is still a path from s to j_i, that is, it contains no cycles. To see this, assume that by adding j_i to P we created a cycle. Then this cycle must have zero length, since for every arc (i, j) of this cycle we have by complementary slackness that $p_i = a_{ij} + p_j$. However, a cycle of zero length is ruled out by our assumptions about the graph. The validity of the algorithm and more details can be found in [4]. We shall refer to this algorithm as the *forward* auction algorithm to contrast it with the *two-sided* algorithm we shall develop in Section 4.

It is possible to weaken the positivity assumption on the cycle lengths to non-negativity by introducing the idea of *graph reduction* [11,2]. Here, through the use of certain upper bounds on the node distances, it is possible to ascertain that some of the arcs cannot participate in a shortest path to their endnode. Such arcs can be deleted from the graph or equivalently their arc lengths can be set to a very large number. In addition to allowing zero length cycles, it can be shown that graph reduction enhances the worst case running time of the auction algorithm and improves its practical performance for some difficult problems. In this paper, we will not discuss graph reduction further, but we note that the parallelization schemes to be presented, admit versions with graph reduction, where the lengths of the 'deleted' arcs are set to a very large number.

3. A forward parallel scheme

We now consider a multiple origins version of the shortest path problem, and a parallel version of the auction algorithm for its solution. The parallelization is

primarily suitable for a shared memory machine. It is based on the use and the simultaneous update of a different path for each origin. There are several possible schemes [12]. Here we shall restrict our analysis to a particular asynchronous scheme, which is easy to implement and can be extended to two-sided parallel algorithms (Section 4). According to this scheme the paths from different origins have no common nodes, i.e. they are *disjoint*.

There are r origins $s_k \in \mathcal{N}$, $k = 1, \ldots, r$, and we wish to find the shortest path from each origin to a common destination d. We will assume throughout this section that the problem is feasible, i.e. there exists at least one path from each origin to the destination. We maintain at each time t a price vector $p(t)$, to which all processors have access, and for each origin s_k we maintain a path $P^k(t)$ starting at s_k. In a synchronous implementation, the algorithm consists of phases that are separated by *synchronization points*, i.e. times at which processors have ended a phase but have not started a new one. In the asynchronous implementation, there is no notion of phases; processors proceed with computations with whatever data is accessible at the time.

We now describe the asynchronous auction algorithm in detail and prove its validity. The synchronous version is obtained as a special case by setting $R(t) = S(t)$ and $\tau_{i_k j}(t) = t$ in the following description. We assume that the price vector $p(t)$ and the paths $P^k(t)$ can only change at integer times and we let $T = \{0, 1, \ldots\}$ denote the set of these times. This is not a restrictive assumption since t may be considered as an index to physical times at which some interesting events occur.

We introduce the following notation:

$i_k(t)$: Terminal node of the path $P^k(t)$. In order to simplify the notation, we shall drop the argument t whenever $i_k(t)$ appears as a subscript and it is clear from the context.

$S(t)$: The set of *active* paths, i.e. paths $P^k(t)$, $k = 1, \ldots, r$, for which the destination node has not been reached yet, that is, $i_k(t) \neq d$.

$R(t)$: Subset of active paths for which there will be an iteration (an attempt to contract or extend) at time t as will be described below.

We assume that at time t each path $P^k(t) \in R(t)$ has calculated a scalar

$$v_k(t) = \min_{\{j \mid (i_k, j) \in \mathcal{A}\}} \left\{ a_{i_k j} + p_j\left(\tau_{i_k j}(t)\right) \right\} \tag{3.1}$$

and a corresponding node attaining the minimum above

$$j_k(t) = \arg \min_{\{j \mid (i_k, j) \in \mathcal{A}\}} \left\{ a_{i_k j} + p_j\left(\tau_{i_k j}(t)\right) \right\}, \tag{3.2}$$

called the *desired node of path* $P^k(t)$, by using prices $p_j(\tau_{i_k j}(t))$ from some earlier but otherwise arbitrary times $\tau_{i_k j}(t) \in [0, t]$.

We consider two subsets of $R(t)$:

$R_c(t)$: The subset of $R(t)$ consisting of paths that are *eligible to contract*, given by

$$R_c(t) = \left\{ P^k(t) \in R(t) \mid p_{i_k}(t) < v_k(t) \right\}. \tag{3.3}$$

$R_e(t)$: The subset of $R(t)$ consisting of paths that are *eligible to extend,* in the sense that their desired node at time t does not belong to any active path and $p_{i_k}(t) = a_{i_k j_k} + p_{j_k}(t)$, i.e.

$$R_e(t) = \left\{ P^k(t) \in R(t) \mid j_k(t) \notin \bigcup_{P^m(t) \in S(t)} P^m(t),\ p_{i_k}(t) = a_{i_k j_k} + p_{j_k}(t) \right\}. \tag{3.4}$$

We note that $R(t)$ may contain paths which are neither in $R_c(t)$ nor in $R_e(t)$. These are the paths $P^k(t)$ such that $p_{i_k}(t) = v_k(t)$ but either $j_k(t)$ belongs to another active path or $p_{i_k}(t) < a_{i_k j_k} + p_{j_k}(t)$. The latter case may occur because $v_k(t)$ was computed with out-of-date price information. We also define:

$E(t)$: The set of desired nodes corresponding to active paths eligible to extend, that is,

$$E(t) = \{ j_k(t) \mid P^k(t) \in R_e(t) \}. \tag{3.5}$$

We consider the following assumptions:

Assumption 1. *For all t, if a path P is active, then there will be an iteration for P at some time $t' \geq t$, i.e. for all t,*

$$P \in S(t) \Rightarrow P \in R(t') \text{ for some } t' \geq t. \tag{3.6}$$

Assumption 2. *For all i, j, t we have*

$$\lim_{t \to \infty} \tau_{ij}(t) \to \infty.$$

These assumptions are necessary because the problem cannot be solved if active paths stop iterating and if old information is not eventually discarded.

Initially, the common price vector $p(0)$ paired with any active path at time 0 must satisfy the CS conditions. Furthermore, we assume that the initial active paths are node-disjoint. Thus at time $t = 0$ we have for all $(i, j) \in \mathscr{A}$ and all active paths $P^k(0)$, $P^{k'}(0) \in S(0)$:

$$p_i(0) \leq a_{ij} + p_j(0),$$

$$p_i(0) = a_{ij} + p_j(0), \quad \text{if } (i, j) \in P^k(0) \text{ for some } k,$$

$$P^k(0) \cap P^{k'}(0) = \emptyset, \text{ if } k \neq k'.$$

If all arc lengths are nonnegative, one possible choice is to select all prices $p_i(0)$ to be zero and to select as initial paths the trivial paths $P^k(0) = (s_k)$, $k = 1, \ldots, r$. However, if some arc lengths are negative, a suitable initial choice of $p(0)$ and $P^k(0)$ may not be obvious, and may require considerable computation to obtain. It will be shown that the above properties are maintained during the algorithm.

At each time t, if $R(t)$ is empty, nothing happens. If $R(t)$ is nonempty then:

(a) For each $P^k(t) \in R_c(t)$ we set the price of node i_k to $v_k(t)$, and if $i_k(t) \neq s_k$, then $P^k(t)$ is contracted.

(b) For each $j \in E(t)$, we consider the subset of paths that are eligible to extend and for which j is the desired node, that is,

$$B_j(t) = \left\{ P^k(t) \in R_e(t) \mid p_{i_k}(t) = a_{i_k j} + p_j\left(\tau_{i_k j}(t)\right) \right\}.$$

A single path from $B_j(t)$, called the *desired path* of node j, and denoted by $P_j(t)$, is selected. Path $P_j(t)$ extends to j and we say that a *successful extension of* $P_j(t)$ *to* j was performed. If $j = d$, then $P_j(t)$ becomes an *inactive* path and never enters the set of active paths again.

Thus the following changes in the paths and the prices occur:

- If $P^k(t) \in R_c(t)$ then:

$$p_{i_{k(t)}}(t+1) = v_k(t), \quad P^k(t+1) = \begin{cases} P^k(t) \setminus \{i_k(t)\}, & \text{if } i_k(t) \neq s_k, \\ P^k(t), & \text{if } i_k(t) = s_k \end{cases}$$

- If $j \in E(t)$ and $P^k(t) = P_j(t)$, then:

$$P^k(t+1) = P^k(t) \cup \{j\}$$

and if in addition $j(t) = d$, then $S(t+1) = S(t) \setminus \{P^k(t)\}$.

- If $i \neq i_k(t)$ for all $P^k(t) \notin R_c(t)$, then $p_i(t+1) = p_i(t)$.
- If $P^k(t) \notin R_c(t)$ and $P^k(t) \neq P_j(t)$ for all $j \in E(t)$, then $P^k(t+1) = P^k(t)$.

The above formulation of the algorithm requires that the time index t takes an infinite number of values. This is a mathematical convenience, and in practice the algorithm can be terminated once there are no more active paths. We say that the algorithm *terminates at time* t if t is the first time such that $S(t)$ is empty, so that no prices or paths change after time t.

In the analysis that follows we prove the validity and termination of the parallel algorithm described. The issues that need to be addressed are the following:

- First we must prove that the active paths are node-disjoint, and when paired with the price vector, satisfy complementary slackness throughout the algorithm (Propositions 3.1 and 3.2). Complementary slackness will then guarantee that if a path extends to the destination, then it must be a shortest path from the corresponding origin to the destination.
- Next we must prove that following any time prior to termination, at least one successful extension will be performed (see the subsequent Lemma 3.4). Furthermore, the number of possible contractions is bounded from above (see the subsequent Lemma 3.3). Termination of the algorithm will then follow.

Proposition 3.1. *The active paths remain node-disjoint during the algorithm, i.e. for all* t *and* $P^k(t)$, $P^{k'}(t) \in S(t)$:

$$k \neq k' \Rightarrow P^k(t) \cap P^{k'}(t) = \emptyset.$$

Proof. We use induction. The active paths $P^k(0)$ are node-disjoint by assumption. Let us assume that the active paths $P^k(t)$ are node-disjoint. We shall prove that the active paths $P^k(t+1)$ are node disjoint.

From the inductive hypothesis it is clear that two active paths $P^k(t+1)$ and $P^{k'}(t+1)$ have a common node only if one of the following holds:

(a) At time t, one of the paths $P^k(t)$ or $P^{k'}(t)$ extended to a node belonging to the other.

(b) Both paths $P^k(t)$ and $P^{k'}(t)$ extended to the same node at time t.

From the description of the algorithm (cf. Eq. (3.4)) we see that, at time t, paths can only extend to nodes $j \notin \bigcup_{P^m(t) \in S(t)} P^m(t)$, making case (a) impossible. Furthermore, by the rules of the algorithm, only one path can extend at time t to a node of $E(t)$, making case (b) impossible. □

Proposition 3.2. *For all t, the active paths and the price vector satisfy the CS conditions.*

Proof. We use induction. The algorithm starts at time $t = 0$ with path-price pairs that satisfy CS. Let us assume that CS is maintained up to time t for all path-price pairs. We shall prove that CS is maintained at time $t+1$.

We first consider the nodes whose price changes at time t. These are the terminal nodes $i_k(t)$ of the paths in $R_c(t)$. For these nodes we have:

$$p_{i_k(t)}(t+1) = v_k(t) = \min_{(i_k(t),j) \in \mathscr{A}} \left\{ a_{i_k(t)j} + p_j\left(\tau_{i_k(t)j}(t)\right) \right\}.$$

The structure of the algorithm [cf. the definition of $R_c(t)$] is such that prices can only increase. Thus in view of $\tau_{ij}(t) \le t < t+1$ for all $(i, j) \in \mathscr{A}$, we have

$$p_j\left(\tau_{ij}(t)\right) \le p_j(t) \le p_j(t+1), \quad \forall j \in \mathscr{N}.$$

Combining the last two relations, we see that

$$p_{i_k(t)}(t+1) \le \min_{(i_k,j) \in \mathscr{A}} \left\{ a_{i_k j} + p_j(t+1) \right\}, \quad \forall i_k(t) \text{ such that } P^k(t) \in R_c(t).$$

For all nodes i whose price does not change at time t, we have by the induction hypothesis

$$p_i(t+1) = p(t) \le a_{ij} + p_j(t) \le a_{ij} + p_j(t+1), \quad \forall j \text{ with } (i, j) \in \mathscr{A}.$$

Combining the last two relations we have

$$p_i(t+1) \le a_{ij} + p_j(t+1), \quad \text{for all arcs } (i, j).$$

There remains to prove that the condition

$$p_i(t+1) = a_{ij} + p_j(t+1) \tag{3.7}$$

holds for every arc (i, j) that belongs to some path $P^k(t+1) \in S(t+1)$. For such an arc (i, j) either $j \in E(t)$ in which case Eq. (3.7) holds by Eq. (3.4) and (3.5), or else (i, j) belongs to some path $P^k(t) \in S(t)$, in which case Eq. (3.7) holds by the induction hypothesis and the fact $p_i(t) = p_i(t+1)$ and $p_j(t) = p_j(t+1)$. [A node i changes price at time t if it is the terminal node of some path $P^k(t) \in R_c(t)$. Such a node will not belong to $P^k(t+1)$ since $P^k(t+1)$ is contracted at time t, and will not belong to $P^{k'}(t+1) \in S(t+1)$ for $k \neq k'$, since $P^k(t)$ and $P^{k'}(t)$ are node-disjoint and $P^{k'}(t)$ cannot extend to a node of $P^k(t)$.] The induction is complete. □

Based on Proposition 3.2 and the discussion following Eq. (2.1) we have:

Lemma 3.1. *If an active path $P^k(t)$ extends to a node i, then $P^k(t+1)$ is a shortest path from s_k to i.*

Proof. By Proposition 3.2 we conclude that if path P^k extends to a node i, then the CS condition (2.1a) holds with equality on that path. Thus from our analysis of the auction algorithm in Section 2, we conclude that a shortest path from origin s_k to node i has been found. □

An immediate conclusion from the above lemma is that if an active path $P^k(t)$ extends to the destination then $P^k(t+1)$ is a shortest path from s_k to d. We now prove some lemmas that we will need to prove that the algorithm terminates.

Lemma 3.2. *For the destination node d we have for all t:*
(a) $p_d(t) = p_d(0)$.
(b) If $j_k(t) = d$ for some active path $P^k(t)$, then $d \in E(t)$.

Proof. (a) Once a path extends to d, it becomes inactive. Therefore, no contraction is performed on node d and its price remains unchanged throughout the algorithm.
(b) If the desired node of path $P^k(t)$ is d, we have

$$p_{i_k}(t) = a_{i_k d} + p_d\big(\tau_{i_k d}(t)\big) = a_{i_k d} + p_d(0).$$

Since d cannot belong to any active path during the algorithm, we conclude, using the definition of $E(t)$ (cf. Eqs. (3.4) and (3.5)), that $d \in E(t)$. □

Lemma 3.2 allows us to refer to the price of the terminal node as p_d since it is invariant over time. Our next lemma establishes that the number of possible contractions on a node that becomes terminal node of any path is finite.

Lemma 3.3. *The number of contractions performed by the algorithm at any node is finite.*

Proof. By Proposition 3.2, we know that CS is maintained by the algorithm at all times. Therefore, for all origins s_k and times t, $p_{s_k}(t) - p_d$ is an underestimate of the shortest distance from s_k to the destination, which is finite by the feasibility assumption. Since the prices of the origins are monotonically nondecreasing and p_d remains unchanged throughout the algorithm, we conclude that $p_{s_k}(t)$ remains bounded for all origins s_k. We next claim that $p_i(t)$ remains bounded for all i. To see this, note that in order to have $p_i(t) \to \infty$, node i must become terminal node of at least one active path, say P^m, infinitely often, implying that $p_{s_m}(t) - p_i(t)$ is equal to the shortest distance from s_m to i infinitely often. This implies that $p_{s_m}(t) \to \infty$ contradicting our earlier assertion that $p_{s_m}(t)$ is bounded.

It is easy to see with an inductive argument that for every node i its price is the length of some walk starting at i plus the initial price of the final node of the walk; we call this the modified length of the walk. If a contraction occurs at a node, its price increases to a level corresponding to a strictly larger modified walk length. Since the number of distinct modified walk lengths within any bounded interval is bounded and $p_i(t)$ stays bounded, it follows that the total number of contractions that can be performed at a node is finite. □

Our next lemma establishes that a successful extension will be eventually performed if the algorithm has not terminated. Thus the algorithm cannot be deadlocked prior to termination.

Lemma 3.4.

$$S(t) \neq \emptyset \Rightarrow R_e(t') \neq \emptyset \quad \textit{for some } t' \geq t.$$

Proof. We assume that $R_e(t') = \emptyset$ for all $t' \geq t$ in order to reach a contradiction. Then after t no active path will extend. Therefore, there can be at most one contraction for each node in $\bigcup_{P^k(t) \in S(t)} P^k(t)$ except for the origins of the active paths. From Lemma 3.3 we know that the number of possible contractions at the origins of the active paths is finite. Thus there exists a time $\bar{t} \geq t$ after which the active paths remain unchanged and the prices of all nodes in the graph remain unchanged. Since $\lim_{t \to \infty} \tau_{ij}(t) \to \infty$ (Assumption 2), it follows that after some time $\tilde{t} \geq \bar{t} \geq t$, we have:

$$\begin{aligned}
&S(t') = S(\tilde{t}), && \forall t' \geq \tilde{t} \\
&R_c(t') = \emptyset, && \forall t' \geq \tilde{t} \\
&p_j(t') = p_j\big(\tau_{ij}(t')\big) = p_j(\tilde{t}), && \forall t' \geq \tilde{t}, \quad j \in \mathcal{N} \\
&v_k(t') = a_{i_k j_k} + p_{j_k}(t'), && \forall t' \geq \tilde{t}, \quad P^k(t') \in R(t')
\end{aligned}$$

These equations imply that

$$p_{i_k}(t') = a_{i_k j_k} + p_{j_k}(t'), \quad \forall t' \geq \tilde{t}, \quad P^k(t') \in R(t').$$

Since no successful extensions are performed, we conclude that the desired nodes of all active paths in $R(t')$ belong to other active paths, i.e.,

$$\big\{j_k(t') \mid P^k(t') \in R(t')\big\} \subset \bigcup_{P^m(t') \in S(t')} P^m(t'), \quad \forall t' \geq \tilde{t}.$$

From Assumption 1, we see that every active path $P^k \in S(\tilde{t})$ will iterate at some time $t_k \geq \tilde{t}$, that is, it will belong to some set $R(t_k)$, and will thus have a desired node $j_k(t_k)$. Let $\hat{P}^k$ be the path P^k extended by $j_k(t_k)$, and consider the subgraph $(\tilde{\mathcal{N}}, \tilde{\mathcal{A}})$ consisting of the set of paths $\{\hat{P}^k \mid P^k \in S(\tilde{t})\}$. Then after time $\tilde{t}$, the CS condition (2.1a) will hold with equality for all arcs of the subgraph, that is, $p_i = a_{ij} + p_j$ for all $(i, j) \in \tilde{\mathcal{A}}$. Since each node $j_k(t_k)$ belongs to the set

$$\bigcup_{P^m(\tilde{t}) \in S(\tilde{t}) \setminus \{P^k(\tilde{t})\}} P^m(\tilde{t}),$$

the subgraph $(\tilde{\mathcal{N}}, \tilde{\mathcal{A}})$ must contain a cycle. By summing up the CS condition $p_i = a_{ij} + p_j$ for all arcs (i, j) of the cycle, we conclude that this cycle has zero length, which contradicts our assumptions about arc lengths. □

Proposition 3.3. *The algorithm terminates.*

Proof. First we observe that between two extensions to a given node there must be an intervening contraction at that node. Therefore, each time a node i becomes terminal node of an active path, its price is strictly larger over the preceding time that i became terminal node of some path. Since the number of possible contractions at i is bounded (Lemma 3.3) we conclude that the number of times that any active path can extend to node i is bounded. Thus the number of extensions that the algorithm can perform is bounded. Since the algorithm will not stop performing extensions prior to termination (Lemma 3.4), we conclude that the algorithm terminates. □

We note an additional property that can be used to accelerate the algorithm. Assume that at some time t the shortest path P^k from origin s_k to the destination has been found. Then the prices of all the nodes on P^k remain unchanged during future iterations. Thus if an active path starting at another origin $s_{\bar{k}}$ extends to a node $j \in P^k$ at a time $\bar{t} > t$, then the path $P^{\bar{k}}(\bar{t})$ can become inactive and the shortest path from $s_{\bar{k}}$ to d can be found as the concatenation of $P^{\bar{k}}(\bar{t})$ and the part of P^k connecting node j and d.

4. Two-sided auction algorithms

In this section we discuss the two-sided auction algorithm and its parallelization. In Section 4.1 we present the serial two-sided algorithm for the one origin-one destination problem. In Section 4.2 we consider a naive parallel implementation and discuss the changes needed so that it terminates. In Section 4.3 we propose a scheme for the many origins-one destination problem and prove its validity. Finally in section 4.4 we introduce a new serial scheme solving the many origins-many destinations problem and discuss its parallelization.

4.1. The serial two-sided algorithm

It is easy to see that in the shortest path problem the role of the origin and the destination can be reversed. The auction algorithm that we developed in Section 2 is easily changed to run in reverse, from the destination that is.

The algorithm maintains a path ending at the destination, i.e. $P = (i_1, i_2, \ldots, i_k, d)$. In addition to the path, the algorithm also maintains a price vector p, which together with P satisfies CS. The operations of extension and contraction in the reverse algorithm are similar to those for the forward algorithm except that they are conducted at the starting node of the path P rather than at the end node (which is now the destination).

A typical iteration is as follows:

Typical iteration of the reverse algorithm.

- **Step 0 (Scanning of predecessor nodes).** Let j be the starting node of P. If

$$p_j > \max_{\{i \mid (i,j) \in \mathscr{A}\}} \{p_i - a_{ij}\},$$

go to Step 1; else go to Step 2.

- **Step 1 (Contract path).** Set

$$p_j := \max_{\{i \mid (i,j) \in \mathscr{A}\}} \{p_i - a_{ij}\},$$

and if $j \neq d$, contract P, that is, delete j from P.

- **Step 2 (Extend path).** Extend P by node i_j, where

$$i_j = \arg \max_{\{i \mid (i,j) \in \mathscr{A}\}} \{p_i - a_{ij}\},$$

that is, add i_j as the starting node of P. If i_j is the origin s, stop; P is the desired shortest path.

The reverse algorithm proceeds by performing iterations until the origin becomes the terminal node of the reverse path. Note that we can implement a parallel scheme for the one origin-many destinations problem by maintaining reverse paths starting at the destinations as we did in Section 3 for the forward algorithm.

It is possible to combine the forward and the reverse algorithms into a single algorithm. Computational experience [4] has shown that this speeds up the solution. The combined algorithm maintains a common price vector p, and two paths P_f and P_r satisfying CS. The forward path P_f starts at the origin and the reverse path P_r ends at the destination. The algorithm will terminate when the two paths have a common node.

Combined algorithm

- **Step 1 (Run forward algorithm).** Execute several iterations of the forward algorithm (subject to the termination condition), at least one of which leads to an increase of the price of the origin. Go to Step 2.
- **Step 2 (Run reverse algorithm).** Execute several iterations of the reverse algorithm (subject to the termination condition), at least one of which leads to a decrease of the price of the destination. Go to Step 1.

It is crucial for the validity of the combined algorithm that a price increase at the origin or a price decrease at the destination occurs before the algorithm is switched from one side to the other. Otherwise, examples can be constructed showing that the algorithm may never terminate. The algorithm as described, is justified for integer data in [4] and the computation results presented there show that the combined algorithm runs faster that state-of-the-art Dijkstra codes (even

two-sided). In the analysis that follows we assume that the arc lengths are integers, and that the problem is feasible.

4.2. Naive parallel implementation

Let us consider a naive implementation whereby the forward and the reverse algorithms are running on different processors and are updating a common price vector. For simplicity assume that the processors execute iterations simultaneously. It can be seen that a situation may arise where the terminal node of P_f and the starting node of P_r are connected by one arc. Under such conditions CS may be violated as shown in Fig. 1. A simple way to correct this problem is to use a priority rule whereby we allow only one side to iterate if the terminal nodes of the paths are one arc apart. In this way CS is guaranteed to be maintained throughout the algorithm.

Even with this modification, however, the algorithm may not terminate. The reason is that price increases at the origin and price decreases at the destination may fail to occur.

The difficulty is that some nodes, which had been visited by the forward algorithm and their price was increased, are then visited by the reverse and their price is decreased to the level it was before being visited by the forward. The prices of these nodes oscillate and the algorithm does not terminate.

We say that we have *degeneracy* if there are two nodes, $i \in P_f$ and $j \in P_r$, such that in all future iterations where i becomes the terminal node of P_f or j becomes the starting node of P_r an extension follows. Thus the prices of i and j remain unchanged and degeneracy occurs. Fig. 2 gives an example where we have degeneracy with the corresponding nodes being the origin and the destination. Degenerate iterations are what prevents the algorithm from terminating. Therefore, an easy modification to our naive implementation would be to stop the reverse side of the algorithm once we suspect degeneracy, until a contraction to the origin with a price increase for the origin occurs, in which case we restart the reverse side of the algorithm.

It is reasonable to suspect degeneracy if many successive extensions occur at certain nodes. We thus introduce two counters for each node, which we shall call the *forward* and the *reverse counter* respectively. The forward (reverse) counter records how many successive times a forward (respectively, reverse) extension was performed at a node without an intervening contraction. If a contraction is

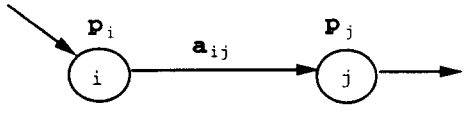
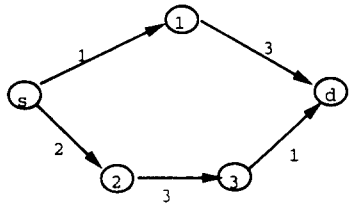


Fig. 1. This is an example where CS may not be maintained: Let i be the terminal node of the forward path and j be the starting node of the reverse path. Assume $p_i = 0$, $p_j = 0$, $a_{ij} > 0$, and (i, j) is the only arc outgoing from i and incoming to j. At time t both sides perform an iteration. The forward will set $p_i(t+1) = a_{ij}$ and the reverse will set $p_j(t+1) = -a_{ij}$. As a result, CS is violated.

performed at a node, then its corresponding counter is reset to zero. If both the forward counter of some node and the reverse counter of some (possibly different) node exceed a threshold, then we declare that degeneracy has been detected and we allow only one side of the algorithm to proceed as we explained above. It is also possible to have different thresholds for each counter.

We may wish to detect degeneracy as soon as possible, so that time is not wasted with the algorithm cycling. This can be done by having a low threshold for the counters. However, iterations with a few consecutive extensions on the same node are not necessarily degenerate. In particular, we may have more than one paths originating from the same node and having the same length. In such a case the algorithm is not cycling since new paths are being explored. Therefore, the trade-off here is that we may allow a number of degenerate iterations with the algorithm cycling, which is time-consuming, or else stop one side of the algorithm before it is necessary. However, for all practical purposes the number of allowed degenerate iterations can be set to be large (for example 10% of the number of arcs), since the situation of the example in Fig. 2 is rare.



						Terminal node		
T	s	1	2	3	d	FOR	REV	
1	2	2	0	0	-1	s	3	
2	2	2	0	-3	-1	2	d	
3	2	2	0	-3	-1	3	1	
4	2	1	0	0	-1	2	d	
5	2	1	3	0	-1	s	3	
6	2	1	3	0	-1	1	2	
7	2	2	0	0	-1	s	3	Steps 1 and 7 are identical
								the algorithm cycles

Fig. 2. An example of cycling of the naive two-sided algorithm. We record the following: in the column labeled T the time index, in the columns labeled s, 1, 2, 3, t, the corresponding prices, and in the columns labeled FOR and REV the terminal and starting nodes of the forward and reverse paths, respectively. One time unit corresponds to a full iteration of the forward and the reverse algorithms in parallel. The sequence of events are recorded at the leftmost column. Step 7 is the same as step 1, so the algorithm cycles. A similar but longer example can be constructed for the same graph when the initial prices are all zero (see [12]).

4.3. The many origins-one destination algorithm

We proceed now to discuss the model for the parallel two-sided algorithm. We assume that we have r origins $s_k \in \mathcal{N}$, $k = 1, \ldots, r$ and one destination d. We shall denote by $P_f^k(t)$ the path starting at origin s_k, $(k = 1, \ldots, r)$, and by $P_r(t)$ the path ending at d. In the sequel, we will use the term *forward paths* to refer to paths that start at an origin, and the term *reverse path* to refer to the path ending at d. We also have, as before, a common price vector $p(t)$, to which all paths have access. We assume that the price vector $p(t)$ and the paths $P_f^k(t)$, $k \in \{1, \ldots, r\}$ and $P_r(t)$ can only change at integer times. Let $T = \{0, 1, \ldots\}$ denote the set of these times.

Throughout the analysis, we assume that the arc lengths are integers and that the problem is feasible, that is, there exists at least one path from each origin to the destination. If the shortest path from an origin s_k to d has not been found, we say that the path P_f^k is *active* and otherwise we say that P_f^k is *inactive*. The reverse path is active throughout the algorithm. Let $S(t)$ denote the set of active forward paths. A forward path may also become *idle* if degeneracy has been detected. An idle path does not iterate but may become active again if certain conditions apply.

Let t_k^f, (t^r) be the largest time prior to t that the kth forward path (the reverse path) started its auction iteration at its current terminal node $i_k^f(t)$, (starting node $i_r(t)$). Then the calculations are based on price information $p(\tau_{i_k^f j}(t))$ and $p(\tau_{i_r j}(t))$ of some earlier times $\tau_{i_k^f j}(t)$ and $\tau_{i_r j}(t)$ where:

$$t_f^k \leq \tau_{i_k^f j}(t) \leq t, \quad \text{and} \quad t^r \leq \tau_{i_r j}(t) \leq t \tag{4.1}$$

Let $R(t)$ denote the subset of active paths (forward or reverse) for which there will be an iteration (an attempt for an extension or a contraction) at time t. We introduce counters of successive forward and reverse extensions.

$FC_i(t)$: The forward counter of node i at time t.

$RC_i(t)$: The reverse counter of node i at time t.

We also introduce a threshold parameter *NUM* for the allowed number of successive extensions per node. All counters are initialized to zero. Analogously to Assumptions 1 and 2 we assume the following:

Assumption 3. *For all t, if $S(t)$ is not empty, then at least one of the following holds*:
(1) $P_r \in R(t')$, *for some* $t' \geq t$.
(2) *Each path* $P_f^k \in S(t)$ *will be in* $R(t_k)$ *for some* $t_k \geq t$.

Assumption 4. *For all i, j, t we have*:

$$\lim_{t \to \infty} \tau_{ij}(t) \to \infty.$$

Initially, the common price vector paired with any of the initial paths must satisfy the CS conditions. Furthermore, all initial paths are node disjoint. The algorithm proceeds as follows:

- The forward active paths perform auction iterations according to a parallel scheme similar to the one we described in Section 3 (maintaining the forward active and idle paths disjoint) and using the prices from some earlier time. If the terminal node of a forward path is one arc apart from the terminal node of the reverse path, then the forward path restarts its auction iteration at its current terminal node. Each time a node i is terminal node of a path and a contraction is made, its forward counter FC_i is reset to zero. Each time a node i is terminal node of a path and an extension is made, its forward counter FC_i is incremented; if $FC_i > NUM$ and there is some node j such that $RC_j > NUM$, then the forward path containing node i becomes idle. If a forward path extends to a node belonging to the reverse path, then the forward path becomes inactive.
- The reverse path performs auction iterations. It never becomes idle. Each time a node i is the starting node of the reverse path and a contraction is made, its reverse counter RC_i is reset to zero; if the contraction is made at the destination, then all idle forward paths become active and the forward counters of their nodes are reset to zero. Each time a node i is the starting node of the reverse path and an extension is made, its reverse counter RC_i is incremented. If the reverse path extends to a node belonging to an active or idle forward path, then the forward path becomes inactive.

We say that the algorithm terminates when all forward paths become inactive. Note that the reverse path is always active; only forward paths can become idle during the algorithm. The termination of the algorithm follows similar lines as the one of the forward algorithm of Section 3, with appropriate modifications to deal with degeneracy and idle paths. In summary:

First we establish that the active and idle paths paired with the price vector satisfy CS throughout the algorithm. This will be a result of the forward paths being node-disjoint, the constraints on how outdated the prices can be, and the way the price vector is updated. The proof is similar to the one of Section 3. CS guarantees that each time a forward path becomes inactive, the shortest path from the corresponding origin to the destination has been found. To prove that the algorithm terminates we need to establish that the number of extensions performed, before a contraction at the origin or a contraction at the destination occurs is finite. This is evident from the fact that we have set an upper bound on the number of successive extensions that can be performed when any node of the graph is the terminal node of a forward path or the starting node of the reverse path. Since the arc lengths are integers and the prices of the origin and the destination can be shown to be bounded, the algorithm terminates.

4.4. The many origins – many destinations problem

In this section we present a serial scheme for the many origins-many destinations problem and then parallelize it.

4.4.1. The serial algorithm

Let us assume that we have r_1 origins and r_2 destinations. For each origin node we maintain a path starting at that origin and for each destination we maintain a path ending at that destination. We also maintain, as before, a common price vector. In the sequel, we will use the term *forward paths* to refer to paths starting at an origin and the term *reverse paths* to refer to paths ending at a destination.

A forward path is *active* if the shortest paths from the corresponding origin to all destinations have not been found. Similarly a reverse path is *active* if the shortest paths from the corresponding destination to all origins have not been found. Otherwise, a path is *inactive*. An active reverse path may also become *idle* in the course of the algorithm. An idle path does not perform any auction iterations, but may become active again if certain conditions apply. We use idle paths in order to be able to concentrate once in a while on one of the many origins-one destination problems. In this way, we guarantee that the algorithm makes some irreversible progress once in a while. The criterion for a path to become idle is as follows:

> **Idle reverse path criterion.** *Each time a shortest path from an origin to a destination is found, the corresponding reverse path becomes idle unless it is the only reverse path that is active.*

We consider an algorithm which proceeds in phases. At the beginning of each phase idle paths become active again. Thus at the beginning or each phase paths are either active or inactive. Each phase consists of two subphases:

- **Subphase 1.** All active paths are allowed to perform auction iterations in any sequence subject to the following constraint: We perform auction iterations with the same forward or reverse path until either the price of the corresponding origin or destination, respectively, changes, or some new shortest path is obtained. Reverse paths become idle according to the criterion we stated above. Paths may also become inactive. When there is only one active reverse path left, the algorithm enters subphase 2.
- **Subphase 2.** The corresponding many origins-one destination problem is solved to completion. All active paths are allowed to perform auction iterations in any sequence subject to the following constraint: We perform auction iterations with the same forward or reverse path until either the price of the corresponding origin or destination, respectively, changes, or a new shortest path between one of the origins and the destination is obtained. When the reverse path becomes inactive, subphase 2 ends and we proceed to a new phase of the algorithm.

The scheme is demonstrated in Fig. 3. In order to prove the termination of the scheme we reason as follows. First we must establish that the scheme does not oscillate indefinitely without any origin finding the path to any destination. This follows directly from the fact a forward or reverse path continues to perform auction iterations until a price change to the corresponding origin or destination,

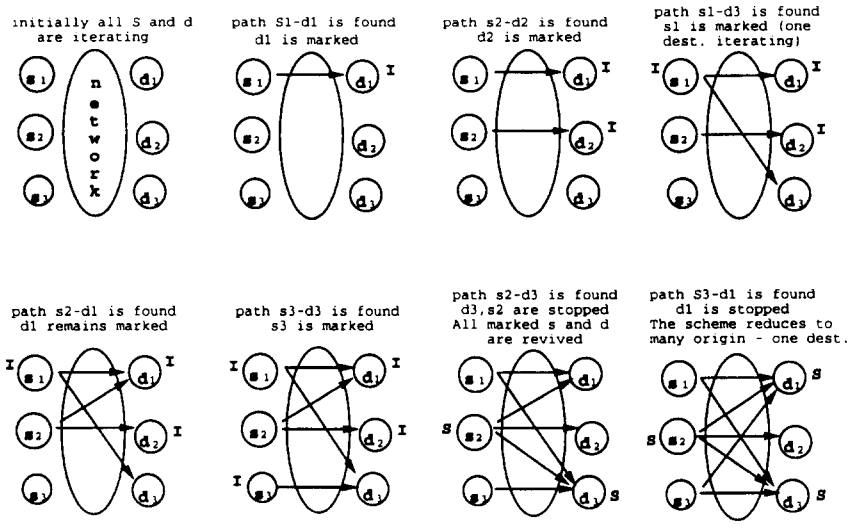


Fig. 3. A demonstration of the many origin-many destination serial auction. A node with I on its side is marked idle. A node with S on its side has become inactive.

respectively, occurs. Also the prices of the origins (destinations) corresponding to active paths can only increase (decrease). Thus under the feasibility assumption (i.e. shortest paths for all origin-destination pairs exist), in finite time only one destination will be left iterating and then our scheme is equivalent to the one for the many-origins one-destination problem, which is known to terminate. Once that destination becomes inactive, again a many origins-many destinations scheme is obtained with at least one destination less. Thus we deduce that under the feasibility assumption the algorithm terminates. Further discussion of the proposed scheme can be found in [12].

4.4.2. Parallel many origins – many destination problems

The many origins-one destination scheme that we developed in Section 4.3 can be easily extended to the many origins – many destinations problem. Assume as in Section 4.4.1 that we have r_1 origins and r_2 destinations. A straightforward parallel scheme is to break up the problem into r_2 many origins – one destination problems. We pick a destination from which we run the reverse algorithm. Once all paths to that destination have been found then we pick a new destination. The advantage is that we can use the same price vector generated by the previous run with a different destination.

A more interesting scheme arises when we parallelize the serial scheme of Section 4.4.1. The parallel scheme is similar to the one in Section 4.3. The only differences are that now we maintain many disjoint reverse paths, we switch between subphases 1 and 2 as described in Section 4.4.1, and we allow only one side to set the price of its terminal (starting) node when an active forward and an active reverse path are one arc apart (as discussed in Section 4.1). In particular:

- We have r_1 forward paths, each starting at one of the r_1 origins, which change according to the scheme in Section 4.3. Furthermore, if a reverse path is intersected by a forward path then the reverse path is marked idle unless it is the only active reverse path. A path becomes inactive if the shortest paths from its origin to all destination nodes have been found.
- We have r_2 reverse paths, each ending at one of the r_2 destinations, which change according to the equivalent reverse parallel auction with disjoint reverse paths. Furthermore, if a reverse path is intersected by a forward path then the reverse path is marked idle unless it is the only active reverse path. A reverse path becomes inactive if the shortest paths between its destination and all origin nodes have been found.
- When there is only one active reverse path, the corresponding many origins-one destination problem is solved (in parallel). Upon completion, the idle reverse paths become active again. The algorithm terminates when all paths are inactive.

We observe that this scheme is a composite of all the schemes that we have analyzed so far. Its validity follows from the validity of the constituent schemes and the fact that the corresponding serial algorithm terminates.

5. Implementations and computational results

In this section we describe the implementation and performance of the schemes developed above on the Encore Multimax which is a shared memory machine. We used a maximum of 20 processors. We implemented the following algorithms:

(1) Asynchronous Forward Algorithm for the all origins – one destination problem.
(2) Asynchronous Two-Sided Algorithm for the all origins – one destination problem.
(3) Asynchronous Two-Sided Algorithm for the many origins – many destinations problem.

We express special thanks to Professor David Castanon of Boston University for providing a sample parallel auction/assignment code for the Encore Multimax, which became the basis for the development of the parallel implementation of our algorithms, as well as for his valuable insights and assistance in our implementations.

5.1. Asynchronous forward algorithm

We start with a list containing all the origins and our goal is to find their distance to a particular node. Origins are taken out of that list as their paths become inactive. The algorithm is implemented so as to reduce the synchronization overhead by allowing each processor to compute prices without waiting for other processors to complete their price updates. Some synchronization is needed to ensure that prices are monotonically increasing and CS is maintained. Synchronization occurs when a processor attempts an extension. In order to ensure that CS

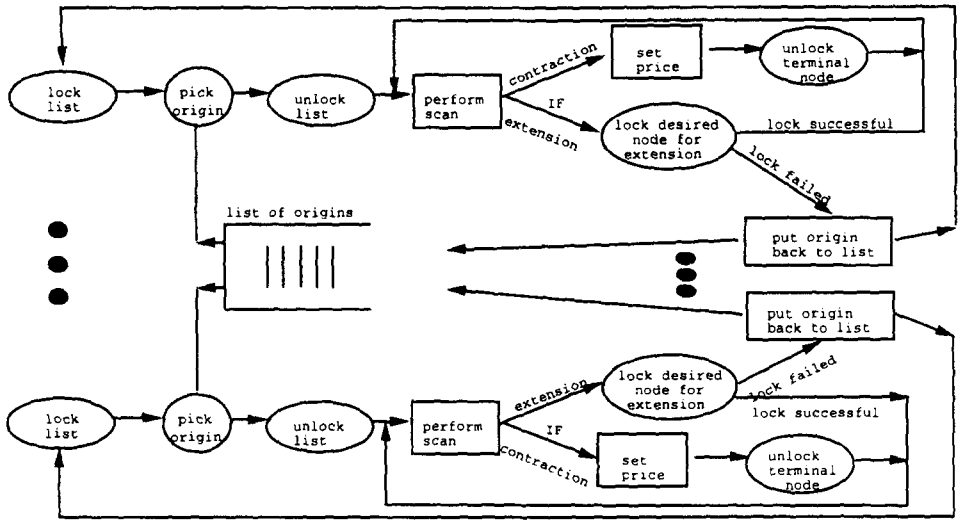


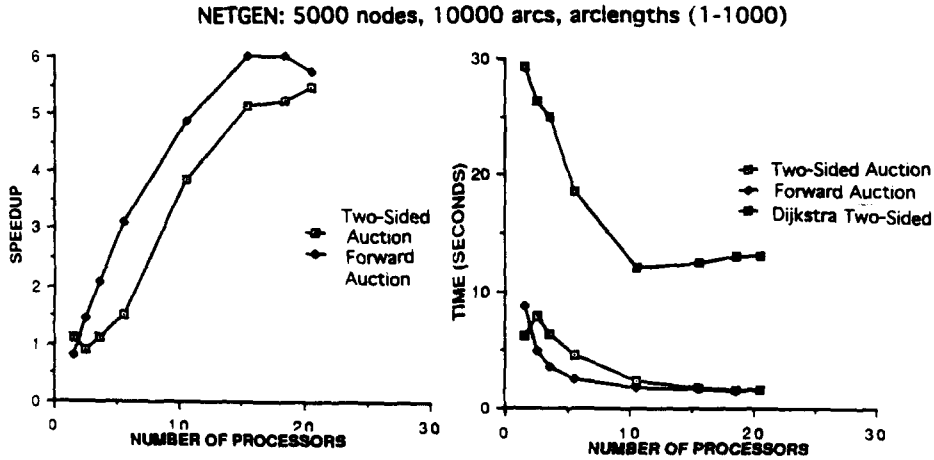
Fig. 4. Flow diagram of the parallel forward-only auction shortest paths algorithm for the all origin-one destination problem.

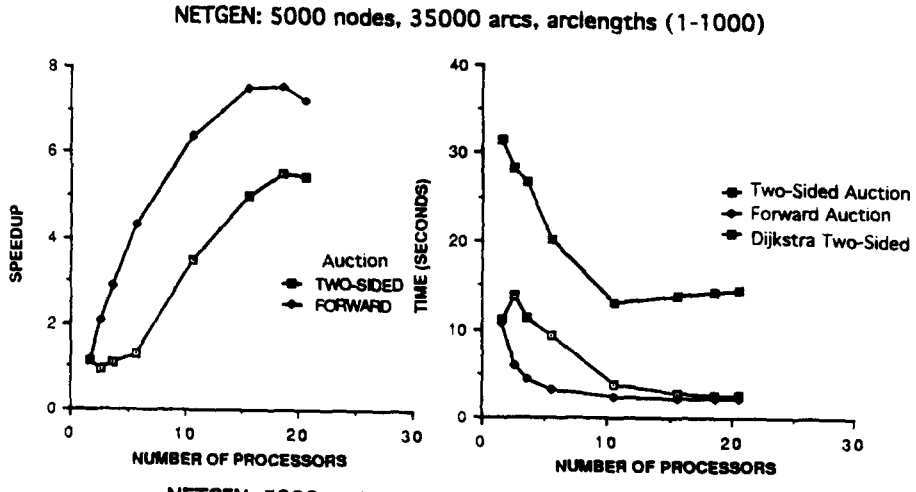
is maintained, we must check whether the desired node is not part of another path and that CS holds with equality with the current prices (rather than the out-of-date prices that the bid was calculated with). This is done with the use of a lock on the memory location containing the price of the node. When a path P extends to a node i, the memory location of the price of i is locked by the path. Thus no other path can extend to i or change the contents of that memory location. To reduce contention for the locks, if a processor is unsuccessful in locking of the price of a node, while working with a path P, it resets P to consist of just the origin, adds the origin to the bottom of the list of active origins and picks another origin from the list to work on. If the shortest distance from an origin to the destination is found, the origin is taken out of the list of active origins permanently. Furthermore, a processor switches to a new origin after a certain number of iterations has been performed on the current origin and the shortest path has not been found. This is done because, heuristically, it is better to allow short paths to be found first, facilitating the search for longer paths. The design of the asynchronous forward algorithm is illustrated in Fig. 4.

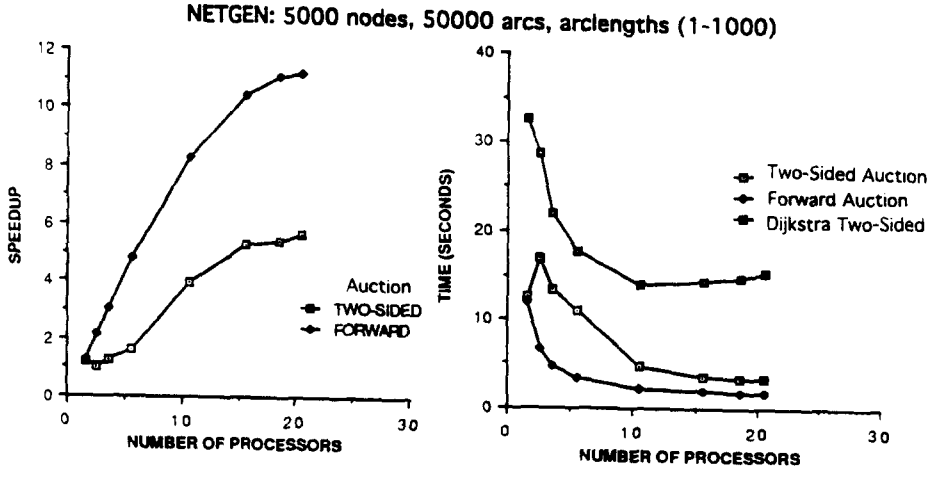
5.2. *Asynchronous two-sided algorithms*

The implementation here also attempts to minimize the synchronization overhead. The forward part of the algorithm is similar to the one above. However, we

Fig. 5 (*facing page*). Computational results for NETGEN graphs with 5000 nodes and 10000, 35000 and 50000 arcs respectively. The arc lengths range from 1 to 1000. We see that as the density of the graph increases the factor of superiority of the forward-only auction over the two-sided algorithm increases. The auction schemes are much faster than the parallel two-sided Dijkstra algorithm.


NETGEN: 5000 nodes, 10000 arcs, arclengths (1-1000)
SPEEDUP
NUMBER OF PROCESSORS
Two-Sided Auction
Forward Auction
TIME (SECONDS)
Two-Sided Auction
Forward Auction
Dijkstra Two-Sided


NETGEN: 5000 nodes, 35000 arcs, arclengths (1-1000)
SPEEDUP
NUMBER OF PROCESSORS
Auction
TWO-SIDED
FORWARD
TIME (SECONDS)
Two-Sided Auction
Forward Auction
Dijkstra Two-Sided


NETGEN: 5000 nodes, 50000 arcs, arclengths (1-1000)
SPEEDUP
NUMBER OF PROCESSORS
Auction
TWO-SIDED
FORWARD
TIME (SECONDS)
Two-Sided Auction
Forward Auction
Dijkstra Two-Sided

have additional synchronization, which arises from checking if the forward path is one arc apart from the starting node of the reverse path. This is achieved by locking the memory location that contains the price of the starting node of the reverse path. When a forward path performs an auction iteration, it locks successively the prices of the nodes neighboring its terminal node. If the price of some node is already locked, by the reverse path, the node enters a left-over list and its price is checked once the reverse path has released the locked memory location.

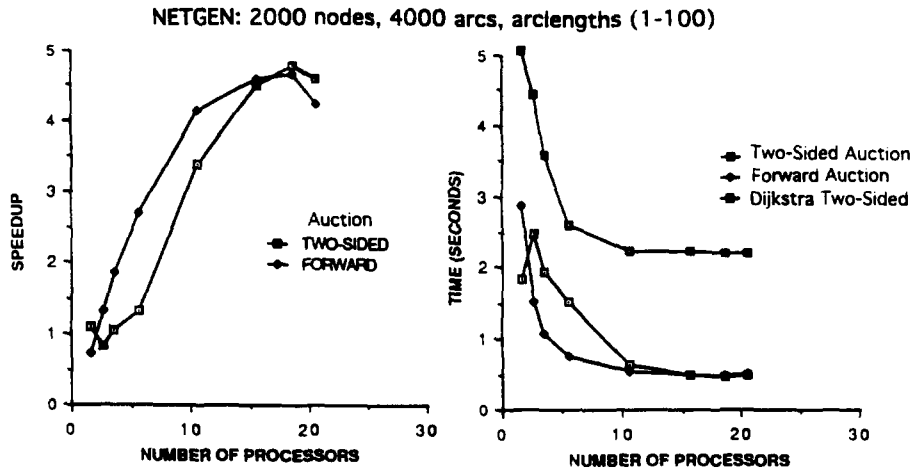
5.3. Computational results

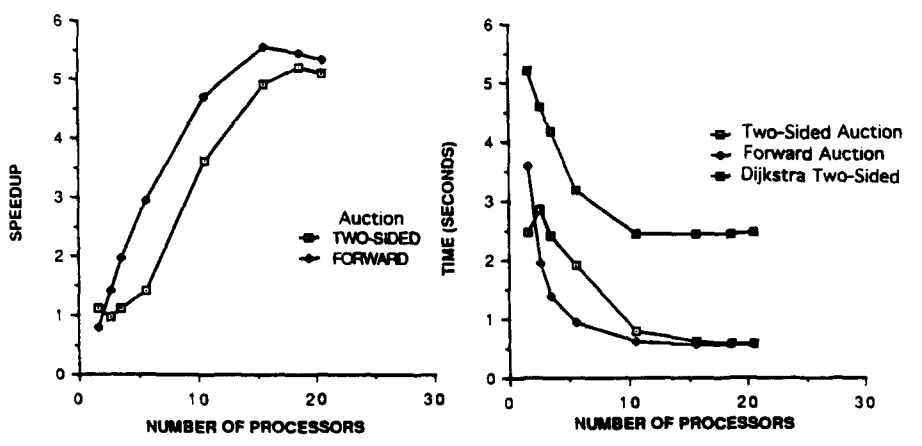
The algorithms were implemented and tested extensively. Three types of graphs were used: Connected randomly generated graphs, pure grid graphs, and hybrid graphs with a grid structure and additional randomly generated arcs satisfying the Euclidean triangle inequality. The connected randomly generated graphs were created with a public domain program called NETGEN [8]. The speedup was measured against the running time of the fastest serial auction algorithm. We also compared the running times of our algorithms with those of a parallel two-sided binary heap Dijkstra algorithm which we also implemented. The serial two-sided Dijkstra algorithm due to [9] is described in [5], p. 86, and the one-sided binary heap implementation of the Dijkstra algorithm is described in [6] and [7]. We implemented the two-sided Dijkstra algorithm based on the state-of-the-art binary heap shortest path code given in [7]. Our implementation is straightforward: We have r processors each running the two-sided serial binary heap Dijkstra algorithm from a different origin. We also maintain a common list of origins as we did for the action algorithms. Each time a shortest path is found, a new origin is picked and the labels set by the reverse side of the Dijkstra algorithm are used to initialize the algorithm. This is done in order to use information from previous runs of the algorithm thus improving its efficiency.

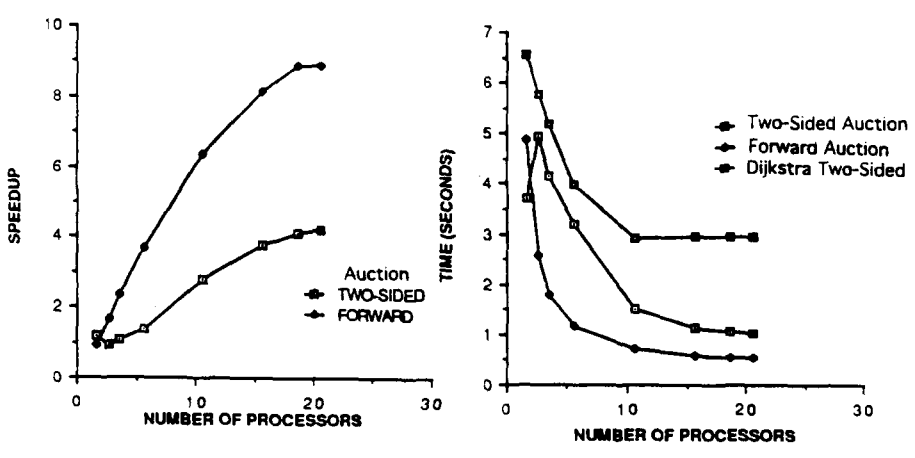
The results of our testing are summarized as follows:

- For the randomly generated (NETGEN) graphs, the forward-only parallel algorithm achieves an average maximum speedup of 5.5 for sparse graphs (see the top graphs in Fig. 5 and 6), which progressively increases with the density of the graph (middle graphs of Fig. 5 and 6) to reach an average speedup of 10 for dense graphs (bottom graphs of Figs. 5 and 6). The maximum speedup is achieved for 15 to 18 processors. The running time deteriorates as the number of processors increases beyond a certain number, because of increased synchronization overhead (more paths compete to acquire locks for the nodes). The two-sided auction algorithm has an average maximum speedup of 5.3, which varies little with the density of the graph and is achieved for 20 processors (Figs.

Fig. 6 (*facing page*). Computational results for NETGEN graphs with 2000 nodes and 4000, 7000 and 20000 arcs respectively. The arc lengths range from 1 to 100. We see that as the density of the graph increases the factor of superiority of the forward-only auction over the two-sided algorithm increases. The auction schemes are much faster than the parallel two-sided Dijkstra algorithm.


NETGEN: 2000 nodes, 4000 arcs, arclengths (1-100)
SPEEDUP
NUMBER OF PROCESSORS
Auction
TWO-SIDED
FORWARD
TIME (SECONDS)
Two-Sided Auction
Forward Auction
Dijkstra Two-Sided


NETGEN: 2000 nodes, 7000 arcs, arclengths (1-100)
SPEEDUP
NUMBER OF PROCESSORS
Auction
TWO-SIDED
FORWARD
TIME (SECONDS)
Two-Sided Auction
Forward Auction
Dijkstra Two-Sided


NETGEN: 2000 nodes, 20000 arcs, arclengths (1-100)
SPEEDUP
NUMBER OF PROCESSORS
Auction
TWO-SIDED
FORWARD
TIME (SECONDS)
Two-Sided Auction
Forward Auction
Dijkstra Two-Sided

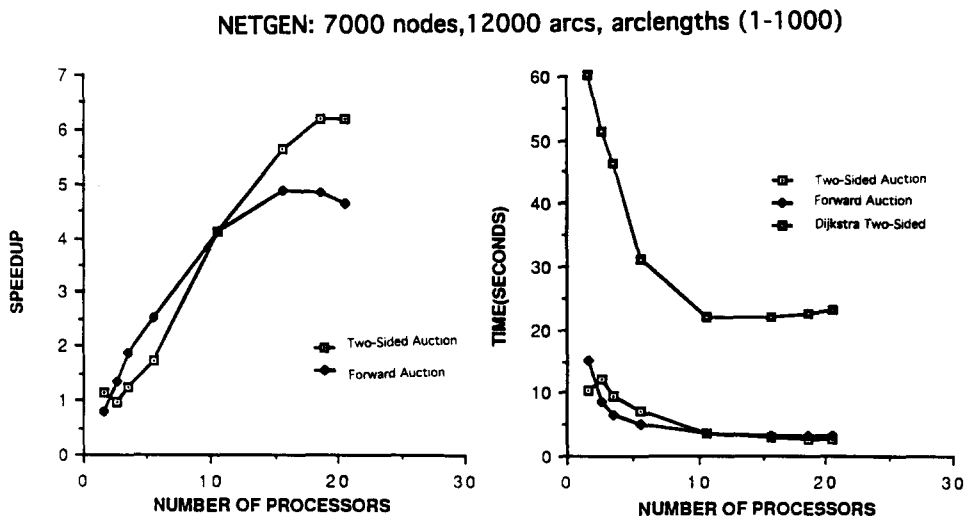


Fig. 7. Computational results for a sparse NETGEN graph with 7000 nodes and 12000 arcs. The arc-lengths range from 1 to 1000. This example shows clearly that for sparse graphs the two-sided auction can outperform the forward-only. The auction schemes are much faster than the parallel two-sided Dijkstra algorithm.

5 and 6). The synchronization overhead for the two-sided algorithm increases for dense graphs since now the forward and the reverse paths are one arc apart more often. This is why the forward-only algorithm has the edge for dense graphs. For very sparse graphs, however, the two-sided algorithm performs equally well or better (see the top graphs of Figs. 5 and 6; also Fig. 7). In all cases the auction algorithms are about 5 times faster than the two-sided Dijkstra algorithm.

- The pure grid graphs are among the most unfavorable for the auction algorithm because of their large diameter. Parallelization, however, leads to great improvement in performance. The forward-only parallel algorithm achieved maximum speedup of around 9 and the two-sided parallel algorithm achieved maximum speedup of around 7 (see Fig. 8). The two-sided Dijkstra algorithm is faster than the parallel algorithms in all cases by a factor of 2 to 3 (Fig. 8, for 20 processors). Parallelization does not seem to improve the running time of the Dijkstra algorithm considerably whereas parallelization is very effective for the auction algorithms.
- For the hybrid graphs we note that as we increase the number of additional randomly generated arcs, the performance of the auction algorithms improves (see Fig. 9). We also observe that although the serial algorithms have a poor performance when compared to the two-sided Dijkstra algorithm, parallelization is very effective and for 20 processors the auction schemes are 2 to 3 times faster than the parallel two-sided Dijkstra.

The above results are illustrated in the figures that follow. Additional testing was performed over different graphs of the same size in order to ensure that the

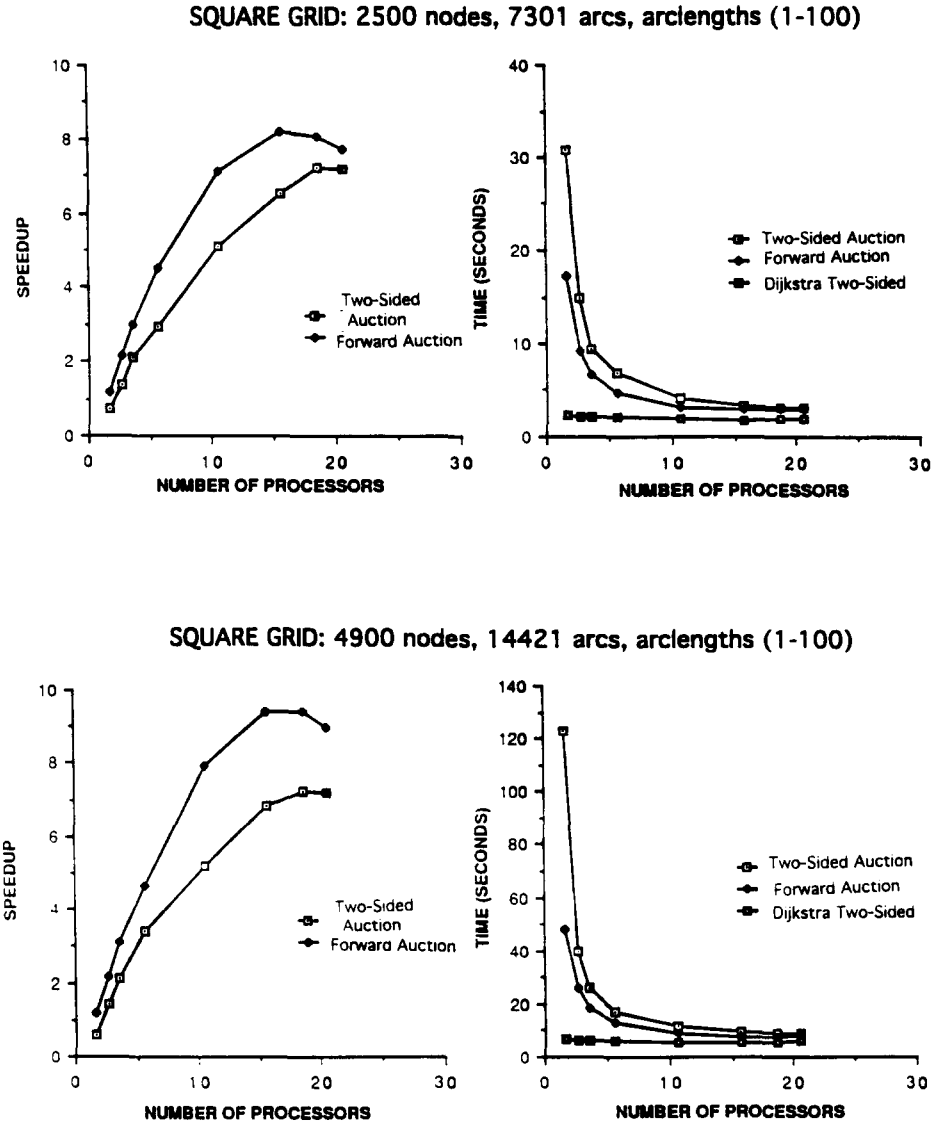


Fig. 8. Computational results for purely grid graphs with 50 nodes in each dimension (top) and 70 nodes in each dimension (bottom). Each node has at most 3 outgoing arcs: one to each of the neighboring nodes on the grid in each dimension and one across the diagonal on the grid. The total number of arcs is 7301 (top) and 14421 (bottom) and the arc lengths range between 1 and 100. We see that parallelization improves dramatically the performance of the auction algorithms.

speedups depend only on the size of the graph and not on a particular instance. Furthermore the times recorded are the average of 3 runs since asynchronism leads to variations in the running times. These variations were less than 10% of the average running times.

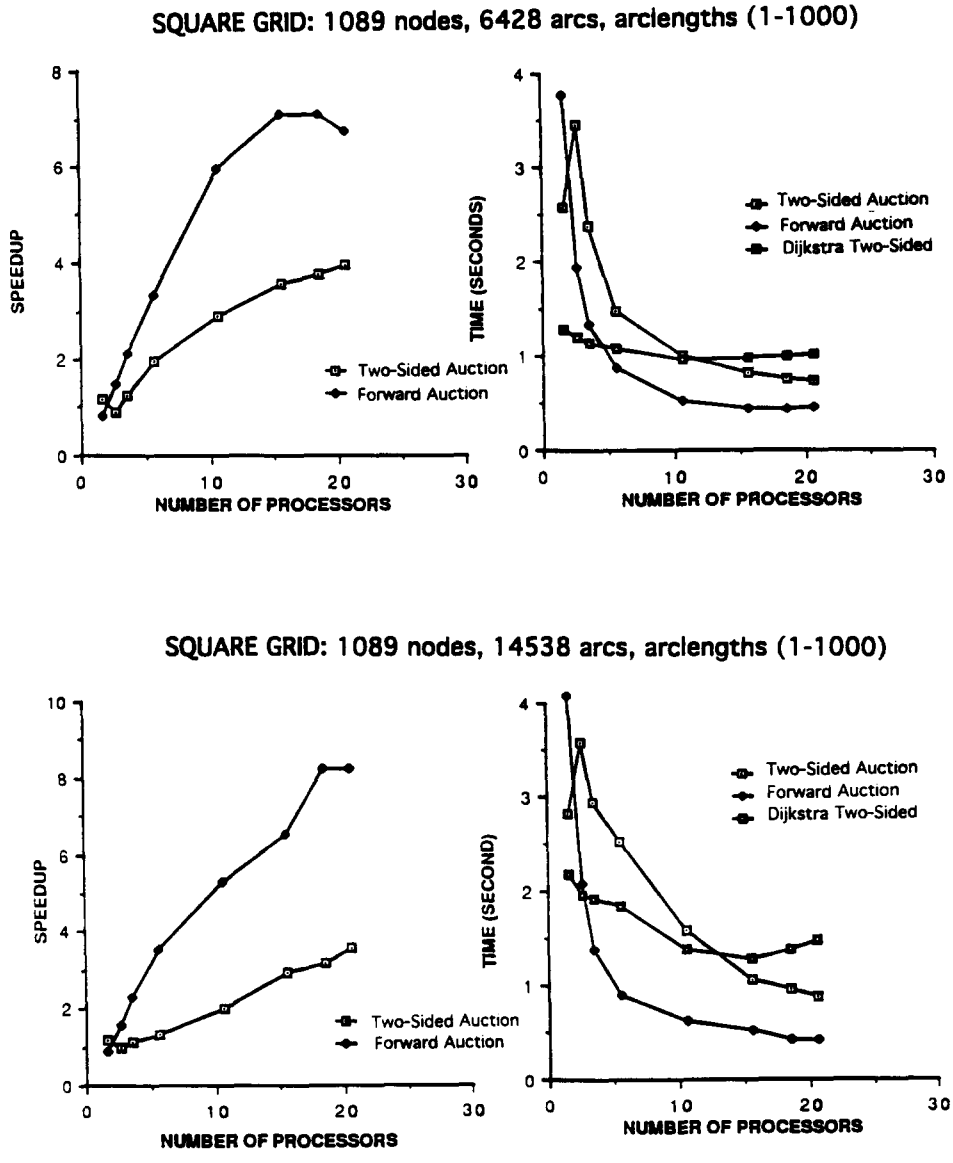


Fig. 9. Computational results for Euclidean grid graphs with additional arcs. For the problem on top we have 2000 randomly generated additional arcs whereas for the problem on the bottom we have 10000 additional arcs. We see that parallelization improves dramatically the performance of the auction algorithms and for 20 processors they outperform the two-sided parallel Dijkstra algorithm.

Finally we tested the many origin-many destinations scheme. We compared it to the serial algorithm where we break up the problem in many single destination subproblems maintaining the same price vector. For 10 origins – 10 destinations problems, we achieved a maximum speedup of about 3 for NETGEN graphs. The

reason the observed speedup is small is that the number of origin-destination pairs in our tests was small, which did not allow taking full advantage of the parallelization.

In conclusion, the parallel versions of the auction shortest path algorithms proved very effective and seem to outperform efficient label setting algorithms. The schemes that we developed are asynchronous and easy to implement. The many origin – many destinations problem achieved reasonably good speedup when parallelized.

References

[1] D.P. Bertsekas and D.A. Castanon, Parallel asynchronous implementations of the auction algorithm, *Parallel Comput.* 1 (1991) 707–732.

[2] D.P. Bertsekas, S. Pallotino and M.G. Scutella, Polynomial auction algorithms for shortest paths, Lab. for Information and Decision Systems Report LIDS-P-2107, May 1992, Mass. Inst. of Tech., Computat. Optimization and Applications, to appear.

[3] D.P. Bertsekas and J.N. Tsitsiklis, *Parallel and Distributed Computation: Numerical Methods* (Prentice-Hall, Englewood Cliffs, NJ, 1989).

[4] D.P. Bertsekas, The auction algorithm for shortest paths, *SIAM J. Optimization* 1 (1991) 425–447.

[5] D.P. Bertsekas, *Linear Network Optimization: Algorithms and Codes* (M.I.T. Press, Cambridge, MA, 1991).

[6] G. Gallo and S. Pallotino, Shortest path methods: A unified approach, *Math. Programming Study* 26 (1986) 38.

[7] G. Gallo and S. Pallotino, Shortest path algorithms, *Annals Operat. Res.* 7 (1988) 3–79.

[8] D. Klingman, A. Napier and J. Stutz, NETGEN – A program for generating large scale (un)capacitated assignment, transportation and minimum cost flow network problems, *Management Sci.* 20 (1974) 814–822.

[9] T. Nicholson, Finding the shortest route between two points in a network, *Comput. J.* 9 (1966) 275–280.

[10] C.H. Papadimitriou and K. Steiglitz, *Combinatorial Optimization: Algorithms and Complexity* (Prentice-Hall, Englewood Cliffs, NJ, 1982).

[11] S. Pallotino and M.G. Scutella', Strongly polynomial algorithms for shortest paths, Dipartimento di Informatica Report TR-19/91, University of Pisa, Italy, 1991.

[12] L.C. Polymenakos, Analysis of parallel asynchronous schemes for the auction shortest path algorithm, Master Thesis, Lab. for Information and Decision Systems Report LIDS-TH-2048, July 1991, Mass. Inst. of Tech.