MIT Document Services

Room 14-0551
77 Massachusetts Avenue
Cambridge, MA 02139
ph: 617/253-5668 | fx: 617/253-1690
email: docs@mit.edu
http://libraries.mit.edu/docs

DISCLAIMER OF QUALITY

Due to the condition of the original material, there are unavoidable flaws in this reproduction. We have made every effort to provide you with the best copy available. If you are dissatisfied with this product and find it unusable, please contact Document Services as soon as possible.

Thank you.

CHOPPY AND INCOMPLETE LETTERS AND TEXT.

ANALYSIS OF PARALLEL ASYNCHRONOUS SCHEMES FOR THE AUCTION SHORTEST PATH ALGORITHM

by

Lazaros Polymenakos

B.S.E.E., National Technical University of Athens (1989)

SUBMITTED TO THE DEPARTMENT OF
ELECTRICAL ENGINEERING AND COMPUTER SCIENCE
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 1991

©Massachussets Institute of Technology, 1991

Signature of Author __________
Department of Electrical Engineering and Computer Science
February 21, 1990

Certified by __________
Professor Dimitri P. Bertsekas

Accepted by __________
Arthur C. Smith, Chair, Department Committee on Graduate Students

ANALYSIS OF PARALLEL ASYNCHRONOUS SCHEMES FOR THE AUCTION SHORTEST PATH ALGORITHM

by

Lazaros Polymenakos

SUBMITTED TO THE DEPARTMENT OF
ELECTRICAL ENGINEERING AND COMPUTER SCIENCE
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

Abstract

An analysis of possible asynchronous parallel schemes for the auction shortest path algorithm for shared memory and message passing environments and a variety of schemes for one-sided and two-sided implementations are presented. This thesis addresses the problem of what conditions must be satisfied in order to parallelise efficiently the auction algorithm for shortest paths in an asynchronous environment, both shared memory and message passing. A new two-sided scheme with different price vectors for each side is developed and its performance is evaluated. Then we exploit the characteristics of the auction algorithm to derive various schemes for the one-sided algorithm and possible schemes for the two-sided algorithm. We also derive constraints on the extent of parallelisation. Finally, we make an assesssment of the efficiency of the parallel schemes and the speedup they may yield both theoretically (complexity) and by means of a simulation. The results show that the schemes are quite efficient and easily parallelisable.

Thesis supervisor: Dimitri P. Bertsekas
Title: Professor of Electrical Engineering

Contents

1. Introduction . . . . p. 5

2. Overview of the Auction Shortest Path Algorithm . . . . p. 9

3. Two-Sided Algorithms

3.1. Serial Two-Sided Algorithm with Common Price Vector . . . . . . p. 15

3.1.1. Convergence for Reals . . . . p. 17

3.1.2. Two-Sided Many Origin/Destination Auction with Common Prices . . . . p. 20

3.2. Serial Two-Sided Algorithm with Different Price Vectors . . . . . . p. 24

4. Asynchronous Shared Memory Forward Auction . . . . p. 35

4.1. The Totally Asynchronous Algorithmic Model . . . . p. 36

4.2. Shared Memory Model for the Asynchronous Auction . . . . p. 36

4.3. Validity and Convergence . . . . p. 40

4.4. Implementation Issues . . . . p. 45

Method 1 (waiting scheme) . . . . p. 47

Method 2 (no waiting scheme) . . . . p. 48

Method 3 (memory server scheme) . . . . p. 50

Method 4 (disjoint path scheme) . . . . p. 53

5. Forward Auction for Message Passing Environment

5.1. Message Passing Model for the Asynchronous Auction . . . . p. 58

5.2. Validity and Convergence . . . . p. 60

6. Asunchronous Two-Sided Auction with Common Price Vector . . . . . p. 65

Analysis . . . . p. 65

Implementation Issues . . . . p. 69

A Simpler Variation . . . . p. 72

7. Asynchronous Two-Sided Auction With Different Price Vectors . . . . p. 77

a) Shared Memory Scheme . . . . p. 77

b)Message Passing Scheme . . . . p. 78

8. Evaluation of Asynchronous Auction Schemes

8.1. Complexity Issues . . . . p. 81

8.2. Simulation and Results . . . . p. 83

8.2.1. Overview of Algorithms and Notation . . . . p. 84

8.2.2. Simulation Assumptions, Implementation and Results . . . . p. 89

Graphs . . . . p. 97

References . . . . p. 115

1. INTRODUCTION

The shortest path problem is undoubtedly one of the most fundamental problems in linear programming. Shortest path problems are usually underlying many combinatorial problems. Thus, as the size of network models is growing, it becomes important to have more efficient shortest path algorithms. If we denote by $\mathcal{N}$ the set of nodes in the network and by $\mathcal{A}$ the set of arcs, then the shortest path problem can be written in the minimum cost flow format:

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

subject to

$$\sum_{\{j|(i,j)\in\mathcal{A}\}} x_{ij} - \sum_{\{j|(j,i)\in\mathcal{A}\}} x_{ji} = f_i, \qquad \forall\, i \in \mathcal{N}, \tag{1.1}$$

$$0 \leq x_{ij}, \qquad \forall\,(i,j) \in \mathcal{A}, \tag{1.2}$$

where

$$f_s = 1, \qquad f_t = -1$$

$$f_i = 0, \qquad \forall\, i \neq s, t,$$

and t is the given destination.

The standard linear programming dual problem is

$$\begin{aligned} &\text{maximize } p_s - p_t \\ &\text{subject to } p_i - p_j \leq a_{ij}, \qquad \forall\,(i,j) \in \mathcal{A}. \end{aligned} \tag{1.3}$$

By the duality theorem [Chv83], [PaS82], the optimal primal cost is equal to the optimal dual cost.

1. Introduction

The shortest path problem has been analysed in the past by many researchers and excellent algorithms have been devised. Among the most well-known are the label setting algorithms (Dijkstra algorithm [Dijk59]), and label correcting algorithms (the Bellman algorithm [Bell58]). Variations of these two sets of generic shortest path algorithms are described in detail in [Pallotino]. The two approaches differ in the way the next node to be labelled is chosen (see also [Dia69], [DGK79], [AMO89], [GaP88], [GaP86], [HKS89]).

In this thesis we shall present various schemes for the asynchronous implementation of an auction algorithm finding the shortest path from many origins to a single destination in a directed graph $(\mathcal{N}, \mathcal{A})$. The algorithm for a serial environment is analysed in detail in [BER90]. One assumption we must make about the network which is important for the validity of the algorithm is that all arc lengths a_{ij} are nonnegative and all cycles have positive length. We shall present some additional assumptions about the network which will help simplify the notation we shall use in the presentation. These assumptions do not limit the generality of the algorithm in any way. Thus, we assume that each node except for the destination has at least one outgoing incident arc. This assumption does not change the problem or the algorithm since any node not satisfying this condition can be connected to the destination with a very high length arc. We also assume that two nodes are connected with at most one arc in each direction, and we refer to an arc (i, j) without ambiguity. Again, this assumption is made in order to simplify notation, since the algorithm can be extended unchanged, to the problem where multiple arcs connect a pair of nodes in the network. As described in [BER90], for the single origin-single destination case, the algorithm is very simple.

A single path starting at the origin is maintained. At each iteration, the path is either *extended* by adding a new node, or *contracted* by deleting its terminal node. The algorithm terminates when the destination becomes the terminal node of the path. It is evident that this algorithm actually solves the dual problem that we presented above (equations 1.3).

A key feature of this algorithm is that for the many-origin, one-destination problem the node prices, as set by paths starting from different origins, can be useful to each other if communicated in such a way as to maintain complementary slackness. We are going to exploit this feature in a variety of ways and thus develop various schemes for asynchronous implementation in two different parallel environments, shared memory or message passing. We shall attempt to make estimates of the efficiency of these schemes both theoretically and through simulation.

The thesis is organised as follows: In Section 2, we make an overview of the auction or dual coordinate ascent algorithm, the key points of which can also be found in greater detail in [BER90]. In Section 3, we discuss the two-sided implementation of the algorithm with a common price vector and we introduce another two-sided scheme with different price vectors. In Section 4, we analyse the asynchronous algorithm for a shared memory environment and we prove the validity of various schemes. In Section 5, we present the asynchronous implementation for the message passing environment. In Section 6, we analyse the case of the asynchronous double sided algorithm with a common price vector. In Section 7, we introduce a many-origin, one-destination two-sided algorithm, where the algorithm running from each side has a different price vector; we also show how this scheme can be extended for the many-origin, many-

destinations problem. In Section 8, we attempt to derive an overall estimate about the efficiency of the various parallel implementations of the auction algorithm. We do so by first exploring the complexity of the parallel schemes presented, i.e. we examine their behaviour for various simple networks. Then we describe the simulation that we made and present the results which are quite encouraging about the efficiency of the asynchronous parallel schemes.

2. OVERVIEW OF THE AUCTION SHORTEST PATH ALGORITHM

Although the auction shortest path algorithm is analysed in detail in [BER90], we shall describe it here for the simple case of the one-origin one-destination shortest path problem. The algorithm can be extended for the many-origin, one-destination shortest path problem as discussed in [BER90].

In the discussion that will follow we shall denote by s the origin node and by t be the destination node. We introduce also the following definitions. A (*directed*) *walk* is a sequence of nodes, $(i_1, i_2, \dots, i_k)$, and a corresponding sequence of arcs, such that $(i_m, i_{m+1}) \in \mathcal{A}$ for all $m = 1, \dots, k-1$. A *path* is a *walk* with the additional property that all nodes $i_1, i_2, \dots, i_k$ are distinct, i.e. a *path* does not contain any cycles. In a straightforward manner, the length of a walk is defined as the sum of its arc lengths.

As we said in the introduction, the algorithm maintains at all times a path $P = (s, i_1, i_2, \dots, i_k)$ starting at the origin. The last node on the path, i.e. node i_k, is called the *terminal* node of P. The degenerate path $P = (s)$ is usually the initialising path of the algorithm and may also be obtained in the course of the algorithm. If i_{k+1} is a node that does not belong to a path $P = (s, i_1, i_2, \dots, i_k)$ and $(i_k, i_{k+1}) \in \mathcal{A}$, we may, during the course of the algorithm, *extend P by i_{k+1}* that is replace P by the path $(s, i_1, i_2, \dots, i_k, i_{k+1})$. If P does not consist of just the origin node s -degenerate path- we may , during the course of the algorithm, *contract P* that is replace P by the path $(s, i_1, i_2, \dots, i_{k-1})$.

In addition to the path, the algorithm also maintains a set of *prices*, one for each node i in the network which we shall denote by p_i. These prices must satisfy the fol-

lowing conditions, which as we shall see, correspond to the dual problem we presented in the introduction (relations (1.3)):

$$p_i \leq a_{ij} + p_j, \qquad \forall\ (i,j) \in \mathcal{A}, \tag{2.1a}$$

$$p_i = a_{ij} + p_j, \qquad \text{for all pairs of successive nodes } i \text{ and } j \text{ of } P. \tag{2.1b}$$

We denote by p the vector of prices p_i. A path-price pair (P, p) satisfying the above conditions is referred to in linear programming literature as satisfying *complementary slackness* (or CS for short). It is easy to see the correspondence with the formulation of the shortest path problem as a linear minimum cost flow problem. The p_i are viewed as the *dual* variables, in the usual linear programming duality sense problem formulation. The complementary slackness conditions for optimality of the primal and dual variables can be shown to be equivalent to the conditions (2.1) for the path-price pair. In the analysis we shall ingnore the linear programming context. However, it is easy to prove that if a path-price pair (P, p) satisfies the CS conditions, 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 us consider a path $P = (s, i_1, i_2, \ldots, i_k)$. Then relation (2.1b) is satisfied for all nodes on the path, i.e.

$$p_s = a_{si_1} + p_{i_1},\ p_{i_1} = a_{1_1 i_2} + p_{i_2},\ \ldots,\ p_{i_{k-1}} = a_{i_{k-1} i_k} + p_{i_k}.$$

We observe that by summing up the first $n (< k)$ equations we get that $p_s - p_{i_n}$ is the length of the portion of P between s and i_n, and by Eq. (2.1a) every path connecting s and i_n must have length greater than or equal to $p_s - p_{i_n}$.

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

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

As discussed in [BER90], we may pick any price vector and then run a preprocessing algorithm in order to ensure that CS will hold for the resulting path-price pair. The auction shortest path algorithm algorithm maintains CS for the (P, p) path-price pair it generates iteratively. 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. The degenerate case in which the path consists of just the origin node s may occur; in this case 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

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

go to Step 1; else go to Step 2.

Step 1: (Contract path) Set

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

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

Step 2: (Extend path) Extend P by node i_x where

$$i_x = \arg \min_{(i,j) \in \mathcal{A}} \{a_{ij} + p_j\}. \quad (2.4)$$

If i_x is the destination t, stop; P is the desired shortest path.

The shortest path algorithm proceeds by perofrming such iterations till the terminal node t becomes endnode of the path. We would like to point out here that after an extension at Step 2, the resulting P is still a path from s to i_x, i.e. it contains no cycles. To see this, if by adding i_x to P we created a cycle, then the cycle would have zero length, since for every arc (i, j) of this cycle we would have $p_i = a_{ij} + p_j$. However, a cycle of zero length is ruled out by the assumptions stated in the introduction.

Figure 2.1 provides an example of the operation of the algorithm. In this example, the terminal nodes are visited in the order of proximity to the origin node s as long as their distance is less than that of the destination node. We will see that this behavior is typical when we choose a zero initial price vector.

We now present some properties of the algorithm. The proofs will not be presented here but can be found in [BER90].

Proposition 2.1: The path-price pairs (P, p) generated by the algorithm satisfy CS, and for every pair of nodes i and j, $p_i - p_j$ is an underestimate of the shortest distance from i to j throughout the algorithm.

Proposition 2.2: Each time a node becomes terminal node of the path P, the path is a shortest path from the origin to that node.

We present here another proposition with its proof as found in [BER90].

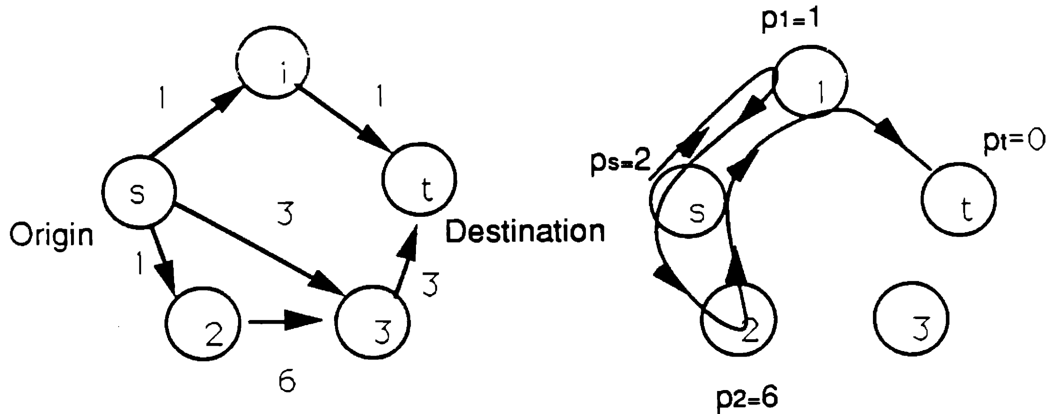


Figure 2.1: Trajectory of the terminal node and final prices generated by the algorithm starting with $P = (s)$ and $p = 0$. As we see node 3 is not visited by the algorithm at all since its distance from s is greater than the distance of the destination node d_n.

Proposition 2.3:

(a) For zero initial node prices, at the end of iteration k_i we have $p_s = d_i$.

(b) If $k_i < k_j$, then $d_i \leq d_j$.

Proof: (a) At the end of iteration k_i, P is a shortest path from s to i by Prop. 2, while the length of P is $p_s - p_i^0$.

(b) By part (a), at the end of iteration k_i, we have $p_1 s = d_i$, while at the end of iteration k_j, we have $p_s = d_j$. Since p_s is monotonically nondecreasing during the algorithm and $k_i < k_j$, the result follows. **Q.E.D.**

A direct conclusion drawn from proposition 2.3 is that for zero initial prices the nodes are visited in the order of proximity to the origin.

Proposition 2.4: Under the feasibility assumption that there exists at least one

path from the origin to the destination, the algorithm terminates in finite time and finds a shortest path from the origin to the destination. Otherwise, the algorithm never terminates and $p_s \to \infty$.

We will now present an estimate of the running time of the algorithm. We assume that all the arc lengths and initial prices are integers. We shall refer to the set of nodes

$$I = \{i \mid d_i \leq d_t\}, \tag{2.5}$$

and the set of arcs

$$\mathcal{R} = \{(i,j) \in \mathcal{A} \mid i \in I\} \tag{2.6}$$

and denote by R the number of arcs in $\mathcal{R}$.

Proposition 2.5: If there exists at least one path from the origin s to the destination t, and the arc lengths and initial prices are all integer then the running time of the algorithm is $O\left(R(D_t + p_t^0 - p_1^0)\right)$.

Proposition 2.5 says that the algorithm is pseudopolynomial since its running time depends on the shortest path lengths. However, under mild conditions, the algorithm can be turned into a polynomial one by employing the the usual trick in such cases, i.e. arc length scaling. The polynomial version of the algorithm is discussed in [BER90].

3. TWO-SIDED ALGORITHMS

3.1 Serial Two-Sided Algorithm with Common Price Vector

It is easy to see that the shortest path problem can also be formulated in reverse, i.e. find the shortest path from the destination to the origin by following incoming incident arcs. The auction algorithm can be easily changed to give such a destination-oriented algorithm that is similar to the origin-oriented, as discussed in [BER90]. In particular, it maintains a path R that *ends* at the destination and changes at each iteration by means of a contraction or an extension. This algorithm will be called the *reverse algorithm*, and is equivalent to the algorithm of Section 2, which will from now on be referred to as the ***forward algorithm***.

In the reverse algorithm, R is initially any path ending at the destination, and p is any price vector satisfying the CS conditions together with R; for example,

$$R = (t), \qquad p_i = 0, \quad \forall\, i.$$

As described in [BER90], an iteration of the algorithm is as follows:

Typical Iteration of the Reverse Algorithm

Step 0: (Scanning of predecessor nodes) Let j be the starting node of R. If

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

go to Step 1; else go to Step 2.

Step 1: (Contract path) Set

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

and if $i \neq t$, contract R, (that is, delete the starting node j of R).

Step 2: (Extend path) Extend R by node j_x, (that is, make j_x the starting node of R, preceding j), where

$$j_x = \arg \max_{(i,j) \in \mathcal{A}} \{p_i - a_{ij}\}.$$

If j_x is the origin s, stop; R is the desired shortest path. .

The reverse algorithm proceeds by performing iterations till the origin becomes endnode of the reverse path. It makes sense now to try to combine the forward and the reverse algorithms into one with the intuition that the resulting algorithm will be faster than the forward only. In this combined algorithm, we initialise, as before, the price vector p, and two paths P and R so as to satisfy CS, where P starts at the origin and R ends at the destination.

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 origin price p_s. 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 destination price p_t. Go to Step 1.

The algorithm is justified for integer data in [BER90] and the computation results presented there show that the combined algorithm runs faster that state of the art Dijkstra codes (even two-sided). This means that the common prices maintained help

the forward or the backward algorithm in its corresponding search for shortest path.

3.1.1 Convergence for Real Numbers

The forward backward scheme has been proved to converge for integer arc lengths only. The main problem in proving its convergence for real data has been so far the fact that after the two algorithms eventually visit the same nodes or nodes at a distance of one arc then a price increase would correspond to the difference between two directed walks (one forward and one backward) in the network. Since such walks can have loops we deduce that by following this idea we cannot reach a proof, since the difference might grow increasingly small and as a result the algorithm, although approaching the solution, would take infinite time to converge. The realisation that the two-sided algorithm would however approach the solution (which follows from the fact that the each sided iteration leads to a price change to the corresponding starting node and the prices are bounded) combined with the fact that the one-sided algorithms do converge for real data, leads us to the idea to try to ensure that price changes with real data will be above a threshhold before the algorithm switches to the opposite sided iterations. We shall call this the *threshold two-sided auction* and we shall describe its serial version and prove its convergence below.

Threshold Two-Sided Auction

Step 1: (Run forward algorithm) Execute several iterations of the forward algorithm (subject to the termination condition). If the total increase in the price of origin since the forward algorithm (re)started iterating is larger than the theshhold

h then Go to Step 2; Otherwise go to step 1.

Step 2: (Run reverse algorithm) Execute several iterations of the reverse algorithm (subject to the termination condition). If the total decrease in the price of destination since the backward algorithm (re)started iterating is larger than the theshhold h then Go to Step 1; Otherwise go to step 2.

To prove that the algorithm converges we think as follows: Let us consider the forward algorithm first. Let t be the time that we switched to the forward algorithm which is still iterating and let $p_s(t)$ denote the price of the origin when the forward algorithm started iterating. We must first prove that after a finite number of iterations the one-sided algorithm will have made a total increase in the price of the origin of at least h or that the termination condition has been satisfied. Indeed there exists some time $\bar{t} > t$ such that $p_s(\bar{t}) - p_s(t) \leq h$, or the algorithm terminates. This follows from the fact that the forward algorithm alone converges for real data. Thus if $d_{st} - p_s(t) + p_t(t) \geq h$ then, possibly after several contractions to the origin, the total price increase for the origin will be at least h. If instead $d_{st} - p_s(t) + p_t(t) < h$ then after several forward iterations the destination node will be reached and the algorithm will terminate. A similar argument can be used for the backward algorithm too. Thus in finite time each sided iteration will lead to an effective change in the price of the respective starting node of at least h, or terminate. Thus under the feasibility assumption and from the fact that the price of the origin (destination) only increases (decreases), we conclude that the threshold two-sided auction will converge in finite time.

As a conclusive remark we note that the threshold two-sided auction reduces to the two-sided auction for integer data if we choose $h = 1$. The asynchronous version of the algorithm developed here, is essentially the same as for the integer case, which will be analysed in §6, where the issue of a price increase corresponds to an effective price increase of at least h. Obviously, the speed of the two-sided algorithm will depend on the choice of the threshold h. If it is too big then the algorithm will not be very efficient. A possible choice for h would be the minimum strictly positive difference between the lengths of two arcs in the given graph, i.e.

$$h = \min_{\substack{\{\forall (i,j),(k,l) \in \mathcal{A}, \\ a_{ij} - a_{kl} > 0\}}} (a_{ij} - a_{kl}).$$

However, it is easy to see that the algorithm would be more efficient if we started it with h being a small number compared to the data thus allowing initially many switches between the two sides so that many nodes are visited from either side which results in improvement in complexity. Then as time elapses and the algorithm has not terminate and it takes many one-sided iterations to make an increase of h we realise that changing to the other side may result in a situation where even more one-sided iterations would be needed for the effective price increase of h. At such a point we would like to increase the threshold in order to make a larger change in the price of the starting node. Thus a dynamic change in the value of h might lead to a more efficient algorithm. Finally we note that the above discussion is mainly of theoretical interest because in practice a computer uses a truncated representation for reals thus h will be by default equal to the minimum positive real that can be represented and the algorithm will converge.

3.1.2 Two-Sided Many Origin/Destination Auction with Common Prices

We shall try to address here the issues of the auction algorithm for the many-origin many-destination problem with common prices. We know from the analysis in [BER90] that the two-sided auction with common prices solves efficiently the problem of many origins and one destination. It is evident that the two-sided auction cannot deal with the many-origin many-destination problem because convergence has been proven under the assumption that the price of the destination can only decrease since it cannot be part of a contraction of a forward path. The problem that arises in the many-origin many-destination case is that a destination may become part of a forward path since it may be an intermediate node so that some other destination can be reached. Then its price will increase and this may lead to oscillations. However, there is a certain advantage in a forward backward scheme with common prices for the many origin/destination problem. It comes from the fact that combined forward and backward iterations result in a great decrease in computational effort. As we shall see in examples in §8, the many-origin one-destination algorithm results in a reduction of complexity and in certain cases the algorithm becomes polynomial. Similarly the forward backward schemes with many origins and one destination result again in a dramatic improvement in complexity and even cases with one forward loop have polynomial complexity. If many destinations were alloed to iterate then even cases with two or more forward loops could be acommodated with a substantial reduction in their complexity. We shall develop below a possible scheme and prove its validity. We shall adopt the terminology that iterations are performed until a *suitable* price change

occurs to the corresponding starting node, in order to allow for both the integer data and the real data case. Let us assume that we have n_1 origins and n_2 destinations. The termination condition for the algorithm running from an origin (destination) is that the shortest distances to all destinations (origins) have been found. An origin can be in three states: iterating, stopped (if its termination condition is satisfied) or frozen. Similarly, a destination node can be iterating, stopped (if its termination condition is satisfied) or frozen. In the frozen mode a node does not iterate but may resume iteration if its termination condition is not satisfied and certain conditions, described below, hold. We shall denote by s_i $i \in \{1, \dots, n_1\}$ the starting nodes and by d_j $j \in \{1, \dots, n_2\}$ the destinations. The algorithm for the serial environment is described below:

Two-Sided Many Origin/Destination Auction

Forward Step 1: Pick an origin s_i, $i \in \{1, \dots, n_1\}$ which is in iterating mode. If all origins are in stop mode then the problem has been solved, thus terminate.

Forward Step 2: (Run forward algorithm) Execute several iterations of the forward algorithm starting at s_i (subject to the termination condition), at least one of which leads to a suitable increase in the price of origin s_i. If some iteration leads to an extention to a destination d_j, then the problem for the pair (s_i, d_j) has been solved, and in order to update the frozen and stopped lists go to step 3. Otherwise if a contraction to the origin s_i is performed with a suitable price change then we switch to the backward algorithm.

Step 3: (Updating of Stop and Frozen lists) We work on the pair (s_i, d_j) that resulted from a forward or a backward Step 2. First we check if the termination

condition for the origin s_i is satisfied and if so, we set the origin to stop mode and go to Step 1 of the opposite sided algorithm. Otherwise, if more than one destinations are iterating, then we set d_j in frozen mode and go to Step 2 of the same sided algorithm, unless it satisfies its termination condition in which case we set it to stop mode and go to Step 1 of the opposite sided algorithm. If, however, it is the only destination node iterating, then there are two possibilities: a) if its termination condition is not satisfied, then we set the origin s_i to frozen mode; b) if its termination condition is satisfied then we set d_j to stop mode, revive all frozen destination nodes and all frozen origins. We go to Step 1 of the opposite sided algorithm.

Backward Step 1: Pick a destination d_j, $j \in \{1, \ldots, n_2\}$ which is in iterating mode. If all destinations are in stop mode then the problem has been solved, thus terminate.

Backward 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 suitable decrease in the price of d_j. If some iteration leads to an extention to an origin s_i then the problem for the pair (s_i, d_j) has been solved and in order to update the frozen and stopped lists go to step 3. Otherwise, if a contraction to the destination d_j is performed with a suitable price change then we switch to the forward algorithm.

The scheme is demonstrated in an example in figure 3.1.

In order to prove the convergence of the proposed scheme we think as follows. First

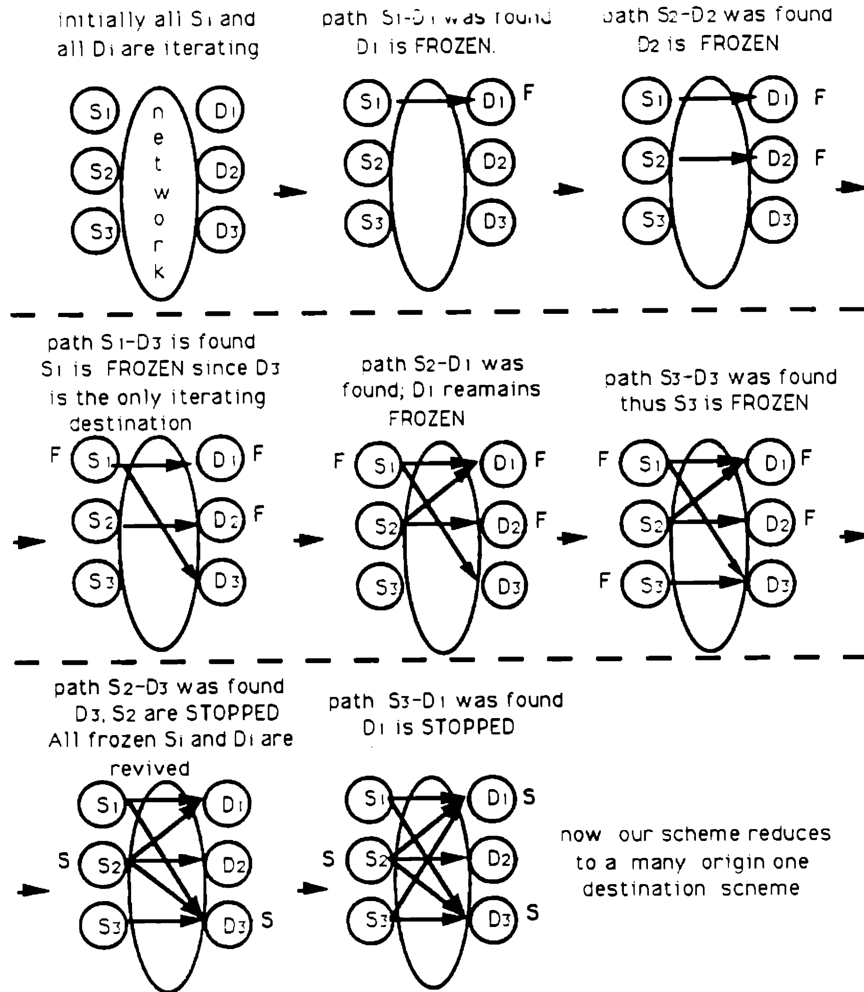


Figure 3.1: **A demonstration of how the many origin/destination auction scheme works.**

we must establish that the scheme does not oscillate indefinitely without any origin meeting any destination. This follows directly from the fact that iterations from a starting node continue till suitable price change to the starting node occurs. Also the prices of the origins (destinations) can only increase (decrease), since if an extension to a destination (origin) occurs the corresponding destination is frozen. Thus under

the feasibilty assumption in finite time only one destination will be left iterating and then we see that a scheme for the many-origins one-destination problem is followed which we know to converge. Once that destination meets its termination condition then again a many-origin one-destination scheme is followed. Thus we deduce that under the feasibility assumption the algorithm converges.

The many-origin many-destination problem could be solved by solving n_2 problems of many-origins and one-destination. However with the scheme that we developed above we expect to have faster convergence since the backward iterations from the many destinations will have a significant improvement in complexity. Thus we expect considerable speedup in the performance of the auction shortest paths algorithm. Actually, the computation results for the serial algorithm that we developed above give a considerable speedup of 2 or 3 over the algorithm where we break the problem up in n_2 problems of many-origins and one-destination (for cases upto five origins and five destinations). The running time results on the Macintosh for some problems are shown below. Up to five origins and five destinations are randomly selected on connected NETGEN graphs with dimensions (1000 nodes, 3000 arcs) and (3000 nodes, 15000 arcs). The running times are actual times measured with the internal clock of the Macintosh (resolution one sixtieth of a second).

3.2 Serial Two-Sided Algorithm with Different Price Vectors

Here we shall present a two-sided algorithm where we keep different price vectors for the forward and the backward algorithm. This technique was initially investigated for the case of the Dijkstra algorithm by [Nich66] and is referred to in bibliography

serial broken up	0.2333	0.3167	0.2	0.467	0.22
new scheme	0.1333	0.15	0.1833	0.35	0.23

serial broken up	0.4167	0.367	0.4167	0.4333	
new scheme	0.25	0.2867	0.15	0.267	

as the bi-directional search [Pohl69], or two-tree algorithm [HKS89]. The reason such an algorithm is of interest is that for certain types of graphs we may speed up the calculations, even if we are using a serial machine, by a factor of at least 2. The auction algorithm also can be implemented in such a way. The key uderlying property that a shortest path algorithm must have, in order to make such an implementation easy and relatively efficient, is the following:

Property 3.1 : The nodes of the graph are visited by the one-sided shortest path algorithm in the order of proximity to the starting node.

Algorithms like Dijkstra and dual coordinate ascent/auction (with zero initial prices as seen by proposition 2.3) possess this property. Let's see now why such a property may lead to an efficient forward and backward algorithm. We shall denote with V_s and V_t the sets of nodes *visited* by the algorithm as is running from the start node and the terminal node respectively. A node becomes *visited* the first time the node goes in the OUT list for the Dijkstra algorithm or the first time it becomes a terminal node on the path. Now let's suppose that no two nodes have the same distance from the start or the end node. This can be easily achieved by applying small perturbations to the arc lengths. Then we can prove the following proposition:

Proposition 3.1 : If $V_s \cap V_t = \{n\}$ then the shortest path from s to t consists only of nodes belonging to $V_s \cup V_t$.

Proof: Since the algorithm possesses property 3.1 and no two nodes have the same distance from either end, we have:

$$\forall\, i \in V_s \;:\; d_{ti} > d_{tn}$$

and

$$\forall\, j \in V_t \;:\; d_{sj} > d_{sn}$$

For the same reason:

$$\forall\, k \in \mathcal{N},\; k \notin V_s \cup V_t : \left.\begin{array}{l} d_{sk} > d_{sn} \\ d_{tk} > d_{tn} \end{array}\right\} \Rightarrow d_{sk} + d_{tk} > d_{sn} + d_{tn}$$

Thus nodes not in the union of the sets V_s and V_t cannot belong to the shortest path from s to t. **Q.E.D.**

Note 3.1 : Now let's suppose that $V_s \cap V_t = \{n\}$ and there exists a path different from (s, n, t) such that the length of (s, i, j, t) is less than $d_{sn} + d_{tn}$ where $i \in V_s$ and $j \in V_t$. Then:

$$d_{si} + a_{ij} + d_{tj} < d_{sn} + d_{tn} \quad \text{with}$$

$$\left\{\begin{array}{ll} d_{si} + a_{ij} > d_{sn} & (\text{since } j \notin V_s \Rightarrow d_{tj} < d_{tn}) \\ d_{tj} + a_{ij} > d_{tn} & (\text{since } i \notin V_t \Rightarrow d_{si} < d_{sn}) \end{array}\right.$$

So from the visited nodes we only have to consider those satisfying the relation:

$$\left\{\begin{array}{l} i \in V_s \;:\; d_{si} \le d_{sn} \\ j \in V_t \;:\; d_{tj} \le d_{tn} \end{array}\right\}$$

The significance of this relation is that if any of the one-sided algorithms has visited nodes more distant than n, they can be ignored in the remaining search of the shortest

path thus reducing computations. Algorithmically this can be achieved by keeping a list of visited nodes by each one-sided algorithm.

Note 3.2 : No path through n other than that giving d_{sn} and d_{tn} (when n became common node) needs to be considered as a candidate for the shortest path. This is because, if

$$\left.\begin{array}{l} \exists\ j \in V_t\ :\ d_{sn} + d_{nj} + d_{tj} < d_{sn} + d_{tn} \\ \text{or}\ \exists\ i \in V_s\ :\ d_{si} + d_{in} + d_{tn} < d_{sn} + d_{tn} \end{array}\right\} \Leftrightarrow \left\{\begin{array}{l} d_{nj} + d_{tj} < d_{tn} \\ d_{si} + d_{in} < d_{sn} \end{array}\right. \tag{3.1}$$

However this is impossible since the algorithm visits nodes in the order of proximity to the root node. In order to describe the two-sided algorithm we first present the reverse algorithm which keeps a different price vector from the forward. In the reverse algorithm, initially, R is any path ending at the destination, and p is any price vector satisfying the CS conditions (1) together with R; for example,

$$R = (t), \qquad p_i = 0, \quad \forall\ i.$$

Typical Iteration of the Reverse Algorithm

Let j be the starting node of R.

Step 0: (Scanning of the predecessor nodes) If

$$p_j < \min_{(i,j)\in\mathcal{A}} \{p_i + a_{ij}\},$$

go to Step 1; else go to Step 2.

Step 1: (Contract path) Set

$$p_j := \min_{(i,j)\in\mathcal{A}} \{p_i + a_{ij}\},$$

and if $i \neq t$, contract R. (that is, delete the starting node j of R).

Step 2: (Extend path) Extend R by node j_x, (that is, make j_x the starting node of R, preceding j), where

$$j_x = \arg \min_{(i,j) \in \mathcal{A}} \{p_i + a_{ij}\}.$$

If j_x is the origin s, stop; R is the desired shortest path.

The reverse algorithm proceeds by performing such iterations till the origin s becomes endnode of the reverse path. The combined forward backward algorithm with diferrent price vectors is as follows:

Combined Algorithm

Step 1: (Run forward algorithm) Execute an iteration of the forward algorithm subject to the following conditions: a) The termination condition of the forward algorithm, b) If a node i is visited for the first time (i.e., its price is zero for a zero initial price vector), add it in the set V_s. If $i \in V_t$ go to Step 3, else choose (randomly) to go either to Step 1 or to Step 2.

Step 2: (Run reverse algorithm) Execute an iteration of the reverse algorithm subject to the following conditions: a) The termination condition of the forward algorithm, b) If a node j is visited for the first time (i.e., its price is zero for a zero initial price vector), add it in the set V_t. If $j \in V_s$ go to Step 3, else choose (randomly) to go to either to Step 2 or to Step 1.

Step 3: (Final search) Once $V_s \cap V_t \neq \emptyset$, i.e. $\{n\} = V_s \cap V_t$, we find

$$d_{st} = \min_{(i,j) \in \mathcal{R}_{st}} (d_{si} + a_{ij} + d_{jt}),$$

where $\mathcal{R}_{st} = \{(i,j) \in \mathcal{A} \mid i \in V_s,\ j \in V_t\}$.

The validity of the algorithm described above follows directly from the validity of each one-sided algorithm, i.e. each side can find the shortest path based on its own price vector. Thus the two-sided algorithm with different price vectors converges in finite time for real arc costs.

Practical Issues : In the previous discussion we assumed that no two nodes have the same distance from the s or t nodes. However, in practice there are cases where many nodes have the same distance from the root node of our algorithm. The implication of this is that we cannot stop the computations on the first common node $\{n\}$. We must proceed to exhaust all nodes which have the same distance as $\{n\}$ on either side of our two-sided algorithm since nodes with the same distance from one root are visited in a random order by the algorithm starting from that root node. Thus, a node n' may have $d_{sn'} = d_{sn}$ but $d_{tn'} < d_{tn}$ and yet n may be visited by the algorithm running with s as a start node, before node n'.

We will now estimate the running time of the algorithm, assuming that all the arc lengths and initial prices are integer. Our estimate involves the sets of nodes

$$V_s = \{i \mid i \text{ was visited by forward algorithm}\},$$

$$V_t = \{i \mid i \text{ was visited by backward algorithm}\}$$

and the sets of arcs

$$\mathcal{R}_s = \{(i,j) \in \mathcal{A} \mid i \in V_s\},\ \mathcal{R}_t = \{(i,j) \in \mathcal{A} \mid j \in V_t\},$$

$$\text{and } \mathcal{R}_{st} = \{(i,j) \in \mathcal{A} \mid i \in V_s,\ j \in V_t\}.$$

We denote by R_* the number of arcs in $\mathcal{R}_*$ where $*$ is s, t, or st.

Proposition 3.2: Assume that there exists at least one path from the origin s to the destination t, and that the arc lengths and initial prices are all integer. The running time of the algorithm is $O\left((R_s + R_t)D_{st} + (R_s - R_t)(p_t^0 - p_s^0) + R_{st}\right)$.

Proof: Let us denote by p^f (p^b) the forward (backward) price vector and p^{f0} (p^{b0}) the corresponding initial price vectors. Let us first consider the forward algorithm. Each time a node i becomes terminal we know that $p_i^f = p_s^f - D_{si}$ (cf. Prop. 2.2). Since $p_s^f \leq D_{st} + p_t^{f0}$ (cf. Prop. 2.3), it follows that

$$p_i^f = p_s - D_{si} \leq D_{st} + p_t^{f0} - D_{si},$$

and using the definitions $d_t^f = D_{st} + p_t^{f0}$ and $d_i^f = D_{si} + p_i^{f0}$, we note that

$$p_i^f - p_i^{f0} \leq d_t^f - d_i^f$$

at the end of all iterations for which node i becomes a terminal node of the path. Therefore, since prices increase by integer amounts we have that $d_t^f - d_i^f + 1$ is an upper bound of the number of times that p_i^f increases. Each time i becomes terminal node of the forward path, the computation is proportional to the number of outgoing arcs from i, denoted n_i^f. Thus, the computation time for iterations where i is the terminal node is bounded by

$$M(d_t^f - d_i^f + 1)n_i^f,$$

where M is a constant independent of the problem data. Summing over the set V_s we see that the running time is bounded above by

$$M \sum_{i \in V_s} (d_t^f - d_i^f + 1)n_i^f.$$

Since by prop.2.3 we have $d_i^f \geq d_s^f$, the upper bound for the forward algorithm is less than or equal to

$$M(d_t^f - d_s^f + 1)\sum_{i \in V_s} n_i = M(d_t^f - d_s^f + 1)R_s = M(D_{st} + p_t^{f0} - p_s^{f0})R_s.$$

Similarly for the backward algorithm we derive the upper bound for the number of iterations is less than or equal to

$$M(D_{st} + p_s^{b0} - p_t^{b0})R_t.$$

Finally we have R_{st} iterations to find the shortest path once $V_s \cap V_t \neq \emptyset$.

Assuming for simplicity that the initial prices are the same for either one-sided algorithm, the result follows. **Q.E.D.**

The estimate of the running time that we found above is actually a loose upper bound to the actual running time. First of all we note that the price increases of any node for the forward algorithm is strictly less than $d_t^f - d_s^f + 1$, since the forward iterations will stop when a node n with $d_n^f < d_t^f$ becomes common node of the sets V_s, V_t. Actually the upper bound is close to $\frac{d_{st}}{2.}$ A similar argument holds for the backward algorithm too. Nonetheless, the above estimated worst case running time shows that for many types of graph we may have considerable speedup in finding the shortest path. In particular, V_s can be only slightly larger than $I_s = \{i \mid D_{si} \leq \frac{D_{st}}{2}\}$ and similarly V_t can only be a little bigger than $I_t = \{j \mid D_{jt} \leq \frac{D_{st}}{2}\}$. Thus the number of nodes that need to be visited may be much less than $I = \{i \mid D_{si} \leq D_{st}\}$, which was for the case of the one-sided algorithm; thus considerable speedup is expected. We shall give an example where the number of nodes visited may be reduced by as

much as a factor of two (figure 3.2). Let us consider the simple case where the start and end node are two points on a plane where only "adjacent" points are connected and distance means Euclidean distance. If the distance (Euclidean) between s and t is $2a$, then the one-sided algorithm starting at s would need to visit $4\pi a^2$ points (nodes) before it could find the shortest distance $s \leftrightarrow t$. However, the two-sided algorithm running on a serial machine would need to visit $2\pi a^2$ points (nodes), which means that the calculations are reduced by a factor of 2 on a serial machine. In general, when there are many nodes whose distance is less than D_{st} but greater than half that distance from either side, the two-sided implementation may be appealing even for a serial machine.

Compared to the two-sided algorithm with common prices, this algorithm is expected to run slower since the prices calculated by the one-sided algorithm are of no help to the other one-sided algorithm. However, an advantage of the two-sided algorithm that we presented here is that it is easily implemented on a parallel environment, synchronous or asynchronous. Each one-sided algorithm runs on a different processor with its own data and they only communicate the nodes each side has visited. This can also be easily done either with message passing or with a common set of flags. Then the expected speedup for a case as in the example above would be close to 4. By contrast, as we shall see in section 6 the two-sided algorithm with a common price vector is not easily parallelisable, whereas the two-sided algorithm with different price vectors will give rise to a number of asynchronous two-sided schemes discussed in section 7.

We present below some running time results to give a rough comparison of the

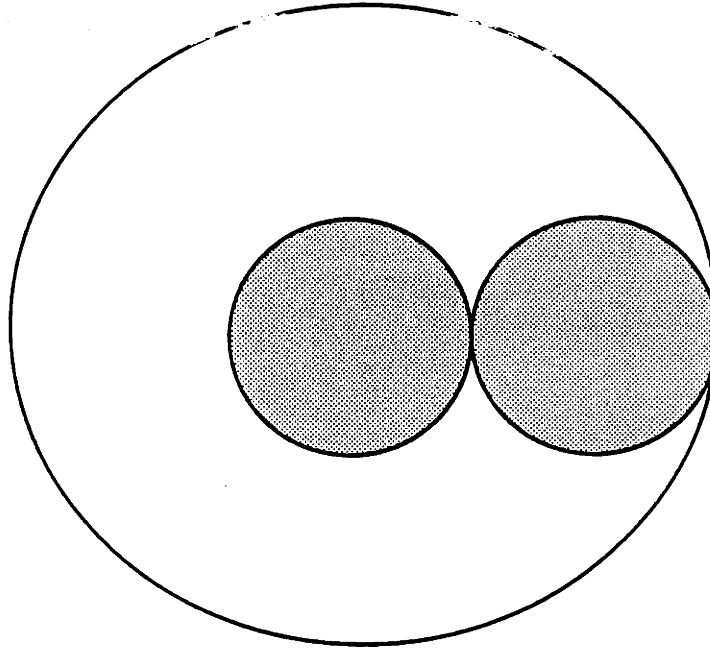


Figure 3.2: The example shows the difference in number of nodes (points) visited on the one-sided algorithm (unshaded circle) versus the two-sided algorithm (the two shaded circles. The origin is the common center of the unshaded and the shaded circles shown and the destination is the center of the second shaded circle. By a simple geometrical argument we conclude that the number of points needed to be visited by the two-sided algorithmb (area of the shaded circles) is half that for the forward only (area of the unshaded cicle).

running time of the two-sided algorithm compared to the one-sided auction algorithm and the two-sided algorithm with common prices. There are also running times of a very efficient forward backward implementation of binary heap Dijkstra (based on a forward code by Pallotino and Gallo) which offer a measure of the overall efficiency of the algorithms. The results represent time in seconds, as given by the clock of MacintoshII. The resolution of the clock is only 1/60 of a second and as a result the data we present have qualitative value only.

Problem #nod,#arc	Forward Auction	For/Back Auction w. com.pr.	For/Back Auction w. dif.pr.	For/Back Dijkstra bin.heap
Netgen 3000,10000	0.8833	0.0333	0.1	0.05
Netgen 2300,9500	0.6667	0.05	0.2667	0.15
Netgen 1000,11000	0.2167	0.0167	0.0667	0.0667
Netgen 2000,5000	0.8833	0.0333	0.1833	0.05
Grid 50X30	0.6333	0.05	0.1167	0.1
Grid 20X100	0.3333	0.0333	0.0667	0.05

4. ASYNCHRONOUS SHARED MEMORY FORWARD AUCTION

Solving Asynchronously the Many Origins - One Destination Problem

We shall present here various schemes for the asynchronous implementation of the auction method solving the problem of finding the shortest path from many origins to a common destination. In the discussion which will follow we assume that we have r origins, denoted $s_1, \ldots, s_r$. The algorithm that we will present in this section is to be executed on a shared memory environment. The exact number of processors or the way the common memory is addressed is of no concern to us as long as they satisfy the rules that will be set forth in the discussion that follows. It is vital to have a common memory which is accessible by all processors, and that each processor may have or be allocated a portion of memory which will be accessible by that processor only. We may, therefore, visualise the system as follows (fig. 4.1):

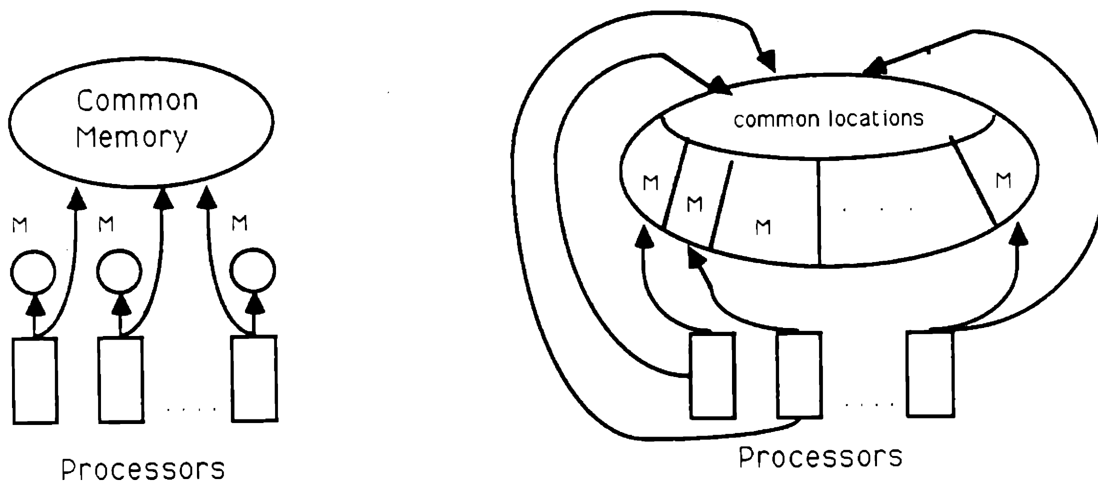


Figure 4.1: Possible shared memory environments

4.1 The Totally Asynchronous Algorithmic Model

We present here the totally asynchronous algorithmic model for general fixed point problems (see also [BER89]). We shall refer to it in the sections that follow so as to make comparisons with the models that will be put forth for the asynchronous auction, and thus draw conclusions on how efficient the proposed schemes are.

Let $X = X_1 \times X_2 \times \ldots \times X_n$ be a set of n-tuples, that is, for $x \in X$ we have:

$$x = (x_1, x_2, \ldots, x_n),$$

where $x_i \in X_i$, $i = 1, \ldots, n$. Let $f_i : X \longmapsto X_i$ be given functions, and define:

$$f(x) = (f_1(x), \ldots, f_n(x)), \quad \forall\, x \in X.$$

Let $x_i(t)$ denote the value of the ith component at time t, and T^i be the set of times at which x_i is updated. We assume that there is a set of times $T = \{0, 1, 2, \ldots\}$ at which one or more components x_i of x are udated by some processor of a distributed computing system (thus $T^i \subset T$). The updating of x_i may be done based on older values of the components of x, i.e

$$x_i(t+1) = f_i(x_1(\tau_1^i(t)), \ldots, x_n^i(t))), \quad \forall\, t \in T^i.$$

where $\tau_j^i(t)$ are times satisfying

$$0 \leq \tau_i^i(t) \leq t, \quad \forall\, t \in T.$$

At all times $t \notin T^i$, the component x_i remains unchanged i.e., $x_i(t+1) = x_i(t)$.

4.2 Shared Memory Model for the Asynchronous Auction

We have, as before, r origins $s_k \in \mathcal{N}$, $k \in \{1, \ldots, r\}$, and we wish to find the shortest paths to the common destination. For the algorithm to operate, we maintain

a common price vector $p(t)$. For each origin s_k we maintain a pair $(P^k(t), p^k(t))$ where $P^k(t)$ is the path starting from s_k and $p^k(t)$ is the price vector associated with the path $P^k(t)$. We also define by $T = \{0, 1, 2, \ldots\}$ asthe set of times that the common price vector is updated, and $T^k \subset T$ the set of times at which an update of the common price vector is based on the path-price pair of origin s_k. We assume that the sets T^k are disjoint, i.e. $T^k \cap T^{k'} = \emptyset$ for $k \neq k'$ and $\bigcup_{k=1}^{r} T^k = T$. The conditions that must be satisfied by the above variables are:

a) The common price vector satisfies the duality conditions (DC for short), i.e:

$$p_i(t) \leq \min_{\{j:(i,j)\in\mathcal{A}\}} (a_{ij} + p_j(t)), \quad \forall\, t, \text{ and } \forall\, i \in \mathcal{N}.$$

b) The pair $(P^k(t), p^k(t))$ satisfies the complementary slackness conditions, i.e:

$$p_i^k(t) \leq \min_{\{j:(i,j)\in\mathcal{A}\}} (a_{ij} + p_j^k(t)), \quad \forall\, t, \text{ and } \forall\, i \in \mathcal{N}.$$

and

$$p_i^k(t) = \min_{\{j:(i,j)\in\mathcal{AA}\}} (a_{ij} + p_j^k(t)), \quad \forall\, t, \text{ and } \forall\, i \in P^k(t).$$

Let us now see how the algorithm works. Let t and $\bar{t}$ be two successive elements of T^k, $(\bar{t} > t)$. Then the pair $(P^k(\bar{t}), p^k(\bar{t}))$ is obtained by several iterations on the pair $(P^k(t), p^k(t))$ where $p^k(t) \equiv p(t)$. An issue that we must refer to is initialisation. Any path-price pairs satisfying the CS conditions are accepatble along with a common price vector satisfying the duality conditions. However an initial common price vector which is just larger by a scale factor from all local price vectors would result in an initial waste of iterations, so it makes sense to initialise all price vectors with the same values. A usual choice is to set all price vectors (local and common) equal to the zero

vector and the paths consisting only of the corresponding origins, i.e. $P^k(0) = \{s_k\}$.

A typical iteration involves several algorithmic steps which we proceed to describe :

Algorithmic step:

Let i^k be the terminal node of P^k.

If:

$$p^k_{i^k} < \min_{(i^k,j)\in\mathcal{A}} (a_{i^k j} + p^k_j)$$

then set

$$p^k_{i^k} = \min_{(i^k,j)\in\mathcal{A}} (a_{i^k j} + p^k_j)$$

and if $i^k \neq s_k$, contract P^k, i.e. $P^k = P^k - \{i^k\}$.

Otherwise: extend P^k by node

$$j^k_z = \arg \min_{(i^k,j)\in\mathcal{A}} (a_{i^k j} + p^k_j),$$

i.e.

$$P^k = P^k \cup \{j^k_z\}.$$

If j^k_z = common destination, do nothing.

Let us suppose that at time $t \in T^k$ we update the common price vector, and we produce an update on the pair (P^k, p^k) as follows:

Common Price Vector Update: $\forall\, i \in \mathcal{N}$,

$$p_i(t) = \max(p_i(t-1), p^k_i(t))$$

Update of the path-price pair of s_k: Let $J = \{j \in P^k(t) :\ p_j(t) > p^k_j(t)\}$.

Then we set:

$$P^k = \{P^k(t) - J\} \cup \{s_k\}.$$

Also we set

$$p^k = p(t).$$

As it will be shown later, P^k is a connected path and the path-price pair generated this way satisfies CS.

Before we proceed to prove the validity and the convergence of the above scheme, we would like to point out the differences with the totally asynchronous model, described in section 4.1. These differences can be viewed as synchronisation restrictions. First we note that delays in information refer to the common price vector as a whole, i.e. $p^k(t) \equiv p(\tau^k(t))$, and not to individual components. Another restriction is that the update of the price vector is based only on one pair $(P^k(t), p^k(t))$, $k \in \{1, 2, \ldots, r\}$ at a time $t \in T$. The reason for this is to maintain CS as seen in the figure below. The reason for error in the example shown is that the prices are not updated in the order they were generated. Yet a similar problem may arise even if the prices are updated in the order they are generated. The problem will occur in the generation of the local price vector since we may have a half copied price vector and prices being updated by other processors. To remedy the problem we must allow all updating processors to finish with their updating of the common prices and then proceed to generate the local path-price pairs. The implemetation issues for such a scheme will be discussed in section 4.4.

A third restriction is that the updating of the common price vector is based not only on the past information but also on the current common price vector. Again tthis is imposed in order to maintain CS of the common price vector.

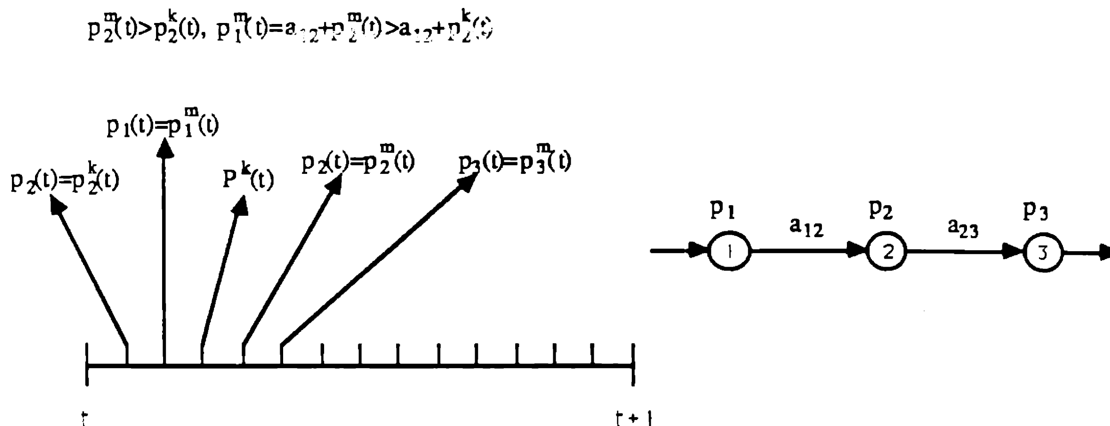


Figure 4.2: We may view time $t \in T$ as a series of events, i.e. price updates and path generation. For the case shown above, we allow processor m to make a common price update before the path of processor k has been generated. If $p_2^m(t) > p_2^k(t)$, we may have that $p_1^m(t) = a_{12} + p_2^m(t) > a_{12} + p_2^k(t)$ as shown in the figure. Then complementary slackness may not hold for the path price pair of origin s_k.

4.3 Validity and Convergence

Here we shall prove that the asynchronous auction on a shared memory environment that we described in section 1.2 will find the shortest paths from the origins s_k to the common destination in finite time under quite mild assumptions that we will put forth.

Assumption 1: The information used for an iteration on a path-price pair for origin k is increasingly recent, i.e.:

$$\lim_{t \to \infty} \tau^k(t) = \infty$$

Assumption 2: All origins for which the path has not been found will continue to iterate and perform common price updates.

Assumption 3 (feasibility): There exists at least one path from each origin to the common destination.

The purpose of introducing assumption 1 is to ensure that as time elapses old information is eventually discarded from the system as obsolete in the sense that no origin is using it to find the shortest path to the common destination. Assumption 2 ensures that no origin will be led to a "starvation" or "indefinite sleeping" before a shortest path to the common destination has been found from that origin. Based on these assumptions we prove:

Lemma 4.1: Let $t \in T^k$ for some $k \in \{1, 2, \ldots, r\}$. If $p(t-1)$ satisfies DC and the path-price pair $(P^k(t), p^k(t))$ satisfies CS, then after the common price update the resulting common price vector $p(t)$ satisfies DC and the generated path-price pair for processor k, i.e. (P^k, p^k) satuisfies CS.

Proof: We have by assumption that both the vectors $p(t-1)$, $p^k(t)$ satisfy DC, i.e.

$$p_i(t-1) \le a_{ij} + p_j(t-1), \;\; p_i^k(t) \le a_{ij} + p_j^k(t), \;\; \forall\, (i,j) \in \mathcal{A}.$$

Thus

$$\max[p_i(t-1), p_i^k(t)] \le a_{ij} + \max[p_j(t-1), p_j^k(t)], \;\; \forall\, (i,j) \in \mathcal{A}.$$

Therefore, after a common price update, the resulting common price vector, i.e.

$$p(t) = \{p_i(t) \mid p_i(t) = \max[p_i(t-1), p_i^k(t)], \;\; i \in \mathcal{N}\}$$

satisfies DC. We know that $p^k = p(t)$ thus p^k satisfies DC. Finally the path is updated in such a way that DC holds with equality. To see this we note that after a path update, there does not exist a node j in the path $P^k(t)$ such that for $(i,j) \in P^k(t)$ the following hold true simultaneously:

$$p_i(t) > p_i^k(t) \text{ and } p_j(t) = p_j^k(t),$$

because then

$$p_i(t) - p_j(t) > p_i^k(t) - p_j^k(t) = a_{ij},$$

which implies that the duality conditions do not hold for the common price vector – contradiction. Thus DC holds on the updated path with equality, which means that the generated path-price pair (P^k, p^k) satisfies CS. **Q.E.D.**

Proposition 4.1: The common price vector satisfies the duality conditions, i.e.

$$p_i(t) \leq \min_{\{j:(i,j)\in\mathcal{A}\}} (a_{ij} + p_j(t)), \quad \forall\, t, \text{ and } \forall\, i \in \mathcal{N}.$$

Furthermore, for every pair of nodes $i, j \in \mathcal{N}$, $p_i - p_j$ is an underestimate of the distance D_{ij}.

Proof: We shall employ induction on $t \in T$ since the set T is an integer set (defined by the finite resolution of the memory clock).

At time 0 all price vectors satisfy the duality conditions (for example, all of them equal zero).

We assume that up to time $t \in T$, the common price vector satisfies the duality conditions.

We shall prove that at time $t + 1$ the common price vector satisfies the duality conditions as direct conclusion of Lemma 4.1. To see this we only need to note that the common vector update is based on taking the maximum of the components of two price vectors that satisfy the duality conditions. For example, at $t + 1 \in T^k, k \in \{1, 2, \ldots\}$ then, $p(t)$ satisfies the duality conditions by assumption, and $p^k(t + 1)$ satisfies DC by construction (processor k performed auction iterations on a price vector satisfying duality conditions, since processor k got a copy of the common price vector at some

time before $t+1$). Since the duality conditions hold we see that the length of any path starting at node i and ending at node j is at least $p_i - p_j$, which completes the proof of the proposition. **Q.E.D.**

Proposition 4.2: For every $k \in \{1, 2, \ldots, r\}$ the pair $(P^k(t), p^k(t))$ maintains CS for all t.

Proof: We only need to prove this proposition for times $t \in T^k(\subset T)$ for $k \in \{1, 2, \ldots, r\}$ since at all other times the pair by construction (based on a typical auction iteration) satisfies the CS conditions. We use a simple induction argument. Initially CS is satisfied (initialisation). Let CS be satisfied at time $t \in T^k \subset T$. Then after the common price vector update, the price vector $p^k \equiv p(t)$, and the path is generated by the rule we have put forth. Thus since by proposition 4.1 the common price vector satisfies the duality conditions, the proof follows from Lemma 4.1. **Q.E.D.**

Note: Each time that we perform a common price update we need not take the maximum over all the elements of the two price vectors. We only need to make the update the nodes whose price in the updating path-price pair has increased due to contractions. The reason is that for any t we have that for all $i \in \mathcal{N}$ whose price was not changed by iterations on the path price pair:

$$p_i(t) \geq p_i^k(t)$$

and both $p(t)$ and $p^k(t)$ satisfy the duality conditions.

Proposition 4.3: The price of the common destination node remains unchanged throughout the algorithm.

Proof : Obviously if the common destination node is attached to a path, say $P^k(t)$, then a shortest path from origin s_k has been found. Thus by the description

of an algorithmic step, the algorithm running from origin s_k stops. Thus no change in price is made. Furthermore, at all other times the price of the common destination node remains unchanged. **Q.E.D.**

Proposition 4.4: The algorithm presented solves the many origin - one destination shortest path problem in finite time.

Proof : Let us suppose, in order to reach a contradiction, that the algorithm does not terminate, i.e. there exists a set S of origins for which the shortest path has not been found in finite time. Then by assumption 2 these origins continue to iterate on their price-path pairs. If some of these origins update the common price vector a finite number of times, then they will find the shortest path to the common destination in finite time. This is because after the last common price update they iterate on a path-price pair satisfying CS and from the validity of the dual auction algorithm, the path will be found in finite time. Thus we conclude that the set of origins S update the common price vector an infinite number of times which implies that there is a set of nodes J^∞ visited an infinite number of times. From proposition 4.1 we conclude that for all origins s_k in S, $p_{s_k} - p_{dest}$ is an underestimate of the finite (by assumption 3) shortest distance from s_k to t. Since the p_{s_k} are monotonically nondecreasing and p_t is fixed throughout the algorithm (Prop.4.2), we conclude that p_{s_k} for all s_k in S remain bounded. We next claim that p_i must stay bounded for all i. To see this we note that, in order to have $p_i \to \infty$, node i must become the terminal node of one or more paths P^k an infinite number of times, which implies that for one or more origins s_k, $p_{s_k} - p_i$ must be equal to the shortest distance from s_k to i an infinite number of times, which is a contradiction since for all s_k, p_{s_k} 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. From the way the common price vector is updated we infer that after a finite number of times that a node $i \in J^\infty$ is visited, its price will be strictly larger over the preceding time i became the terminal node of a path P^k 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 stays bounded, it follows that the number of times i can become a terminal node by extensions of one or more paths P^k is bounded. In addition, the number of iterations between two consecutive path extensions to node i is bounded. We note also that the waiting time for a common price update is finite and the local price vector generated is greater or equal to the one before the price update. This implies that after a common price update the number of distinct walk lengths starting from any i is bounded and less than or equal to the number before the price updateand in turn implies that the algorithm will terminate finitely. Thus a contradiction has been reached. **Q.E.D.**

4.4 Implementation Issues

We now discuss several implementation issues. The first two show the tradeoff between the need for updated information for all path-price pairs and the delay incurred by frequent common price updating. Then we shall present a variation of the asynchronous model presented in the preceding sections in order to allow more processors to make a common price update simultaneously. In order to implement such a scheme we need to make certain assumptions about the way the common memory locations

are updated. We assume also that the algorithms are implemented on systems where the number of processors is equal to the number of our origins nodes. Thus each origin actually corresponds to a processor.

Another issue, that needs to be discussed here, is the amount of work needed for a common price update and local path-price pair generation. First we note that we do not need to take the maximum over all elements of the local and common price vectors when common price updating is performed. We only need to update the common prices of the nodes whose local prices at processor say k changed since the last common price update of that processor. This can be achieved by keeping a **local** list at every processor k which empties after a common price update and is created as follows: If a contraction is performed at a node i enter i in the list. In addition it is not really necesssary to copy the whole common price vector in order to generate the local price vector. Only the nodes whose price changed since the last time that the processor performed a common price update need to be copied. CS will still hold since it holds for the common price vector and all the new information will be correctly communicated. This can be achieved by having a **global** list for each processor k which contains the nodes whose price changed since the last update of processor k. This list empties each time processor k performs a common price update. The list, i.e. for processor k is generated by the other processors when they lock the common memeory to perform their common price update. Then the nodes whose common prices changed during the update are added to the list. This scheme may lead to longer common price updating but the copying of prices needed per processor is much less than the number of nodes in the network being analysed. Thus we have a considerable speedup in the copying of

the common prices since we expect that (number of updates per processor)×(number of processors)<<(number of nodes in network).

Method 1 : (waiting scheme) The iteration on a path-price pair ends when a small number of contractions is made. As in the general scheme, if the common price vector is free then it is *locked* in the sense that only one path-price pair can make changes on it. Only the price of the nodes on which the contractions ware made are updated by the *max* rule described. The local price vector is then generated and the path is updated as described before. This is the new path-price pair for that origin. Obviously CS holds for the common price vector. If the common memory is not free (i.e. is locked by some other origin) the processor waits until it is released. By the description of the algorithm we also conclude that complementary slackness holds for the path-price pairs as before. Thus the algorithm ends in finite time as stated in proposition 4 of §1.4 above.

Evaluation of method 1: The advantage of method 1 is that the common price vector is updated as soon as a few price changes occur which means that almost every processor is working with a price vector very close to the common price vector. Thus new prices become known to processors relatively fast. The main drawback, however, is that this locking of the common price vector and copying are very time consuming. Frequent updates result in overhead in the calculations as such operations are performed at the end of a small number of price changes. Figure 4.3 shows the activity in the common memory as time (absolute time) elapses. The example is a rather obvious one but it provides a basis for comparison with other methods that will be put forth later in this section.

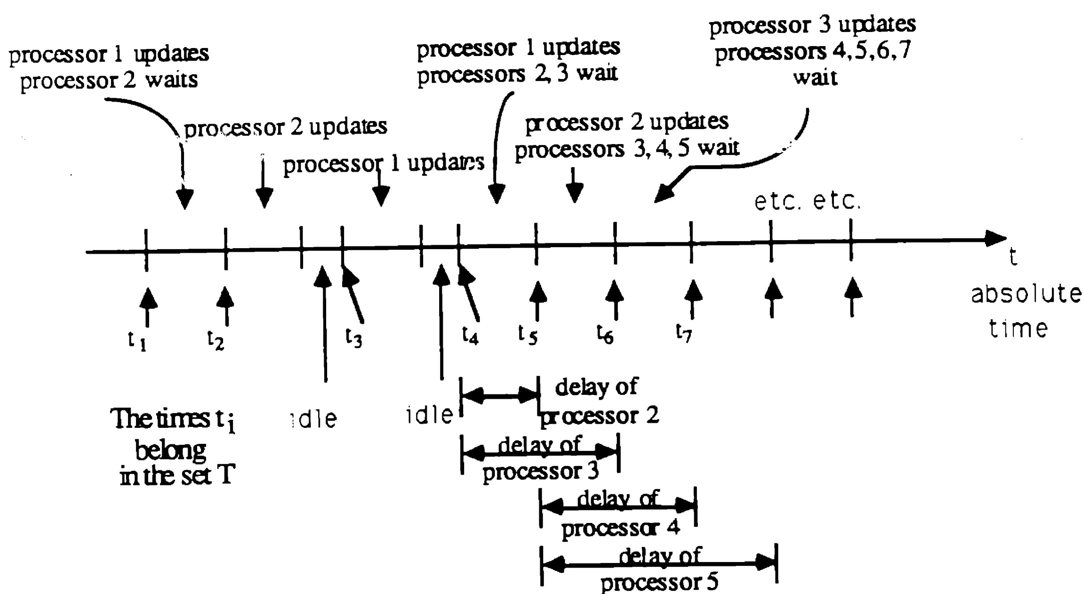


Figure 4.3: An example illustrating the overhead created by frequent updates and costly memory copying.

Method 2 : In method 1 above one main drawback is that if a processor has locked the common memory locations, it is impossible for some other processor to proceed if it wanted to make a price update, too. In the method that we present here, an iteration corresponds to many extensions and contractions (many basic algorithmic steps). At the end of such an iteration, a processor requests to lock the common memory locations in order to make a common price update. If this request cannot be granted at that time, the processor proceeds to make one or more iterations with its current data, then makes a request for a common price update and so on, until the destination node is reached. When access to the common price vector is granted, then only the prices of those nodes for which contraction(s) were performed are updated by the usual rule. Then the common price vector is copied as in method 1 and the path is updated as in the general scheme described in §4.2. Thus this method can be viewed as buffering the updates until access to the common price vector is granted. We assume

that no price update a processor does is lost, i.e. they are all buffered. It follows directly from the proof for the general scheme that complementary slackness holds for the common price vector. By the description of the algorithm we also conclude that complementary slackness holds for the path-price pairs as before. Thus the algorithm ends in finite time as stated in proposition 4 of §1.4 above.

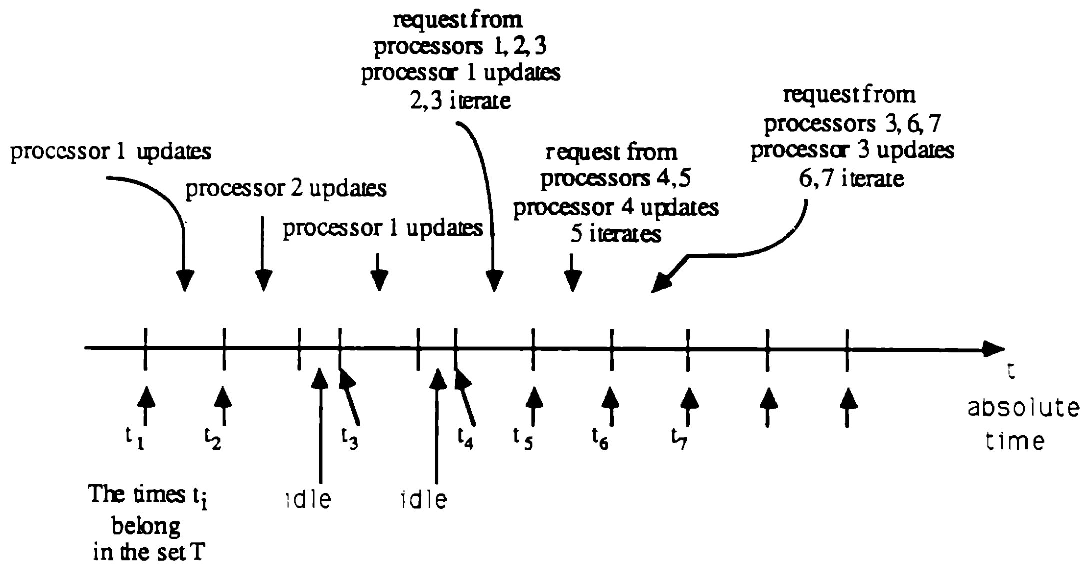


Figure 4.4: The example of figure 3 modified to illustrate a possible senario of updating the common prices for method 2.

Evaluation of method 2: In this method we have improved the overhead resulting from the frequent addressing of the common memory locations. However, the price vector of an origin contains information more outdated than in case 1. In the worst of cases a processor may be led to some kind of starvation and may never be able to access the common memory locations in order to make an update, thus this processor is executing the dual auction on its own without getting information from the other processors in the system. We may take measures to ensure some sort of "fairness,"

i.e. assign priority based on the number of successful common price updates and select among those processors that make a request the one with the least number of successful common price updates .

When we discuss about the overall efficiency of the asynchronous algorithm in section 8 (simulations), we shall see that there is always a tradeoff between obtaining recent information and reducing the overhead. Figure 4.4 shows memory activity in terms of time if method 2 is applied to the example of figure 4.3.

Method 3 : We shall develop now a variation of the original asynchronous model in order to allow more processors to make an update. However, for this method to be executed in a shared memory environment we shall need some kind of memory server, i.e. a processor designated to make the memory updating according to the following scheme: at the beginning of time $t \in T$ all processors that wish to make a common price update have made a request to the memory server. Let S_t be the set of these processors. Then the buffered price updates of each processor (i.e. $B_k(t)$, $k \in S_t$) are sent to the memory server. The common prices are then updated according to the usual rule. When all $B_k(t)$, $k \in S_t$ have been exhausted, then the corresponding paths $P^k(t)$ $k \in S_t$ are calculated, and then the common price vector is copied and sent to every processor $k \in S_t$. We see here that a scenario like that of figure 2 can not longer occur. The validity of this scheme follows directly from the fact that the rule of common price updating maintains CS, and the path-price pairs are generated after all common price updates have been made.

If a memory server is not available the scheme can be implemented as follows. The first condition will be that price updating is done in the order the prices are generated

at each processor. However, after a processor finishes its common price update, the path and local price generation is done in one step, i.e. no other processor is allowed to perform price updating once some processor starts generating its path-price pair. That is we allow many processors perform updates at will (subject to the condition that the order prices are generated is kept) and once a processor starts its path-price generation no changes in prices are allowed. Since the order prices are generated is kept, duality conditions hold for the common prices. In addition, since the path-price pair is generated while no change in common prices is performed, we have the same situation as before. Thus this variation of method 3 is valid. A special case arises when no path needs to be generated. This can be achieved if the processor attempts a common price update only when a contraction to its corresponding origin has occurred. Then any processor can update the common memory without waiting at all by making the update according to the maximum rule and in the order the prices were generated. Evidently, no memory server is needed in this case and the scheme achieves the best possible parallelisation for a forward scheme.

Evaluation of method 3: This method attempts to optimize the tradeoff between overhead for updates and acquisition of recent price vectors. The effect is that more processors are able to perform a common price update and acquire an updated price vector. Also according to this method the maximun waiting is only the time it takes for one common price update. However common price updates are now lengthier and the worst would be for them to be equal to the sum of the times for common price update as in method 1. Figure 4.5 shows an example illustrating how method 3 works.

Before we proceed to analyse another scheme we would like to discuss here the

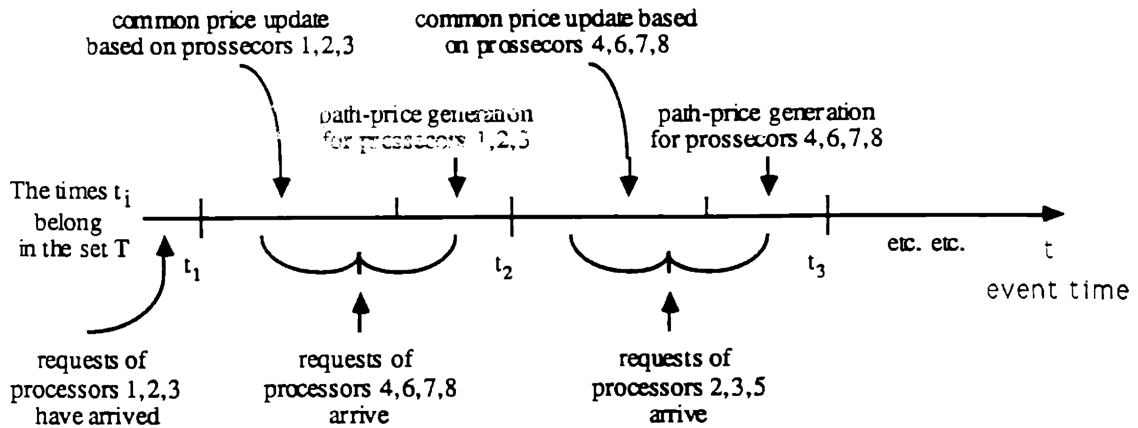


Figure 4.5: The example illustrates a possible scenario of updating the common prices for method 3.

effect of an increasing number of processors to the schemes presented above. It is fairly easy to see that for the first case the delays will start being really humongous since the last processor will have to wait for all r of the processors ahead of it. For the second case the prospects are a lot better but problems may arise. There is, of course, no delay incurred from waiting for other processors to perform an update but some processor(s) may be led to *starvation* in the sense that it may never be allowed to perform a common price update. Thus it will not be able to benefit from the data that other processors generated. Finally, for the third method we see that the server must be increasingly fast so as to be able to serve r processors simultaneously at a relatively high rate (so that long delays are not incurred) and therefore hardware limitations become important. The simpler variation where no path needs to be generated and which, concequently, can do without a memory server is better but new problems arise in regulating the updating of common memory locations by the processors. Of course, copying only the nodes whose prices have changed is very efficient but the generation of the individual lists becomes increasingly time consuming as the number of processors

increases. Thus, given a network to be analysed, there will be an upper bound on the number of processors which can lead to a speed-up in the calculations. Beyond that we may not have considerable speedup or even have a slower algorithm than the serial one with common prices. The situation, however, can be remedied if we want to use a big number of processors (in order to find the distance from a great many origins to the common destination) by breaking up the processors in groups with a different common memory per group. Then we can satisfy the above restriction regarding the number of processors per common memory, and at random times we can perform a maximization over the elements of all different common memories, thus enabling communication of the prices to all processors. Another idea along the same lines would be to have processors which would act as mercenaries. Say we have m processors and q common memory pools and we want to find the shortest distance from n origins. Then each processor is assigned to a task which may be a path-price pair to perform iterations on followed by a common price update to some memory pool, or a maximization over the node prices of two memory pools and generation of a new common price vector for both memories. Such schemes give much greater flexibility to the ways that the asynchronous auction algorithm can be implemented.

Method 4 : We consider now another variation of the original scheme. The model for the asynchronous auction that we introduce here remedies the time consuming copying of a common price vector. However as we shall see in the discussion, new sources of delay arise. In this model, there is only one price vector, the common price vector. Each origin maintains one path, i.e. $P^k(t), k \in \{1, 2, \ldots, r\}$ and an iteration is performed directly on the common price vector. The nodes on such a path are

captured or *locked* in the sense that no other path can make an extension to them. A direct conclusion that we draw from this is that all paths are disjoint at all times, i.e.

$$P^k(t) \cap P^{\bar{k}}(t) = \emptyset \text{ for } k \neq \bar{k}, \ \forall\, t.$$

In order to implement this scheme, we have along with the common price vector, a common set of flags, each taking two values: "C(aptured)" or "F(ree)". We now proceed to describe a typical iteration performed by some path. A vital point is the set of times T^k that a path P^k attempts an extension or a contraction. Let $\bar{t}, t$ be two successive elements of T^k and we denote by $\tau(t)$ times such that $\bar{t} \leq \tau(t) \leq t$ time. Then:

Algorithmic step:

Let $i^k(\bar{t})$ be the terminal node of $P^k(\bar{t})$. We calculate the quantities:

$$Z = \min_{(i,j)\in A} (a_{ij} + p_j(\tau(t))) \text{ and } j^k = \arg\min_{(i,j)\in A} (a_{ij} + p_j(\tau(t)))$$

IF { $p_{i^k}(t) < Z$ then set $p_{i^k}(t) = Z$ and perform a contraction, i.e. $P^k(t) = P^k(\bar{t}) - \{i^k\}$ or if $i^k = s^k$ perform a degenerate contraction. Set the flag of i^k to F(ree).}

ELSE

{*Critical Part:*

IF $p_{i^k}(t) = a_{i^k j^k} + p_{j^k}(t)$ then check flag of j^k : IF F(ree) then change it to Captured and make an extension of the path, i.e. $P^k(t) = P^k(\bar{t}) + \{j^k\}$. If j^k is the common destination node then the shortest path has been found. Free the flags of all the nodes on the current (shortest) path and do nothing.

ELSE the path remains unchanged.}

The part of an algorithmic step that we called *Critical* is the only part of the algorithm that a memory location can be read or changed by one path only. In particular the flag and price of j^k can only be read or changed by the algorithm running on the path starting from s_k. The purpose of this assumption is to make sure that CS will hold with equality after the extension and that an extension to a certain node is allowed for this path in the sense that paths will remain disjoint. In essence this scheme is the closest to the generic one origin-one destination dual coordinate ascent. CS obviously holds for the common price vector and it holds with equality for all paths. In comparison to the previous schemes, here we only lock one memory location. Thus the sets T^k are not disjoint but their subsets, i.e. times of (attempted) extensions to a certain node, are disjoint. Furthermore, we note from the nature of the described algorithm, that the price of a node belonging to a path P^k can be changed only by the algorithm running on path P^k but it can be read by an algorithm running on a different path. We also make the following assumption:

Assuption 4: Every path which is still iterating (i.e. it has not entered a do-nothing mode) that wants to enter the critical part of the algorithmic step, will enter it in finite time. This can be easily ensured by some fairness scheme in the *locking* of a memory location (as we assumed in method 2).

At this point we would like to point out that no situation like the one shown in figure 4.6 can occur. This is because if all the paths shown want to make an extension then since CS holds with equality on all paths, we would have a cycle of zero length

which is contrary to our original assumptions about the validity of the dual auction method. Thus there is at least one path which changes at any time $t \in \bigcup_{k=1}^{r} T^k$.

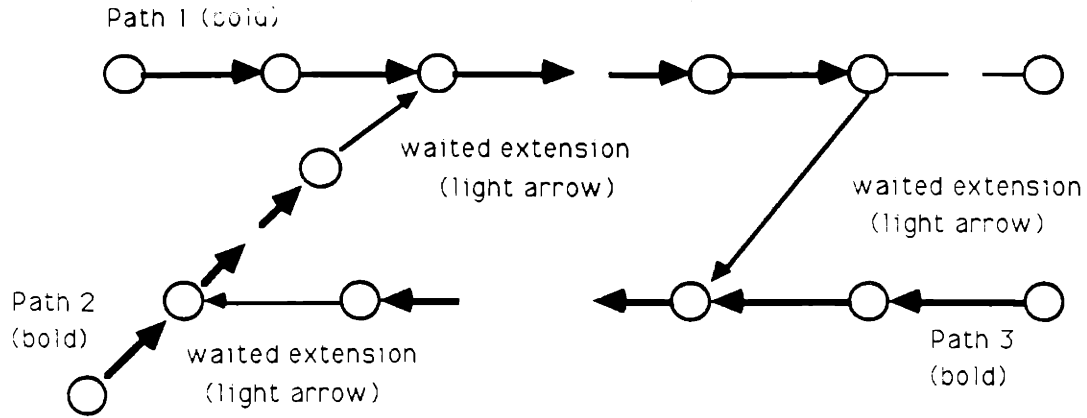


Figure 4.6: Such a deadlock situation is impossible to occur in our scheme since the complementary slackness condition is satisfied with equality on the paths which implies that we would have a cycle (shown) of zero length.

We proceed to show the connection between this scheme and the scheme presented in §1.2.

Assumption 1 is satisfied by the nature of the algorithm since each iteration is done directly on the common price vector. Assumption 2 corresponds to assumption 4 here. Also no infinite waiting of one path for a node belonging to some other path is possible: If a path were to wait indefinetly to make an extension to a certain node then this means that the wanted node is part of some other path for infinite time (otherwise the price of the wanted node would have risen due to a contraction and CS would not hold with equality any more at the path that wanted to make an extension to it). Since no deadlocking situation can occur, at least one path will be changing, thus either the destination node will be found and all nodes on that path will be freed or, due to a contraction, some nodes on the path will be freed which will give the opportunity for other paths to extend to them. Thus we may consider this scheme as a variation of

the original scheme were times $t \in T$ have been selected appropriately so that paths are disjoint.

Evaluation of method 4: In this method the updating of the common memory and the passing of recent information to all paths has been achieved without locking and copying the whole common memory locations. Thus the scheme seems more versatile than the original scheme. However, big delays arise when some path is waiting for some other path to free a certain node for an extension. The delay incurred may be considerable, especially if there are many nodes of the same proximity to the desired node.

Before we end this chapter we would like to note that the assumption of having one processor per origin may not necessarily be fulfilled. In particular the algorithmic model for the auction algorithm presented gives the flexibility to create schemes where we have processors each performing a serial algorithm from a number of origins to the common destination on its own price vector (such a serial algorithm is described in [BER.90]). After a number of iterations a common price update according to the rules presented in this chapter may be executed therby allowing all processors to share the prices of the nodes visited.

5. FORWARD AUCTION FOR MESSAGE PASSING ENVIRONMENT

The asynchronous algorithm that we will present here is to be executed on a network of processors. Each processor is totally autonomous with its own memory and has communication links through which it can send messages to any (all) other processors in the network. The exact nature of the communications will not matter to us. A general scheme of buffered communications independent of the processors that perform the algorithm is assumed. More assumptions about the reliability of the communications is put forth in the discussion that will follow. We also assume that the number of processors is equal to the number of origins for which we want to find the shortest path to the common destination, i.e. processor k will find the shortest path from origin s_k.

5.1 Message Passing Model for the Asynchronous auction

We proceed to describe the message passing model for the asynchronous implementation of the auction. Each processor k maintains a pair $(P^k(t), p^k(t))$ where $P^k(t)$ is the path starting at s_k and $p^k(t)$ is the price vector associated with the path $P^k(t)$. The pair $(P^k(t), p^k(t))$ must satisfy the CS conditions, i.e:

$$p_i^k(t) \leq \min_{\{j:(i,j)\in\mathcal{A}\}} (a_{ij} + p_j^k(t)), \quad \forall\ t, \text{ and } \forall\ i \in \mathcal{N}.$$

and

$$p_i^k(t) = \min_{\{j:(i,j)\in\mathcal{A}\}} (a_{ij} + p_j^k(t)), \quad \forall\ t, \text{ and } \forall\ i \in P^k(t).$$

An algorithmic step is defined here the same way as for the generic one-origin one-destination algorithm. For the initialisation of the algorithm we refer to the same

issues as in §4.2, i.e. any path-price pairs satisfying CS can be used. However, if a processor has an initial local price vector which is just larger by a scale factor from all other local price vectors, this would result in an initial waste of iterations. Therefore, it makes sense to initialise all price vectors with the same values. A usual choice is to set all price vectors (local and common) equal to the zero vector and the paths consisting only of the corresponding origins, i.e. $P^k(0) = \{s_k\}$. We now procede to describe an algorithm step:

Algorithmic step:

Let i^k be the terminal node of P^k.

If:

$$p^k_{i^k} < \min_{(i^k,j)\in\mathcal{A}} (a_{i^k j} + p^k_j)$$

then set

$$p^k_{i^k} = \min_{(i^k,j)\in\mathcal{A}} (a_{i^k j} + p^k_j)$$

and if $i^k \neq s_k$, contract P^k, i.e. $P^k = P^k - \{i^k\}$.

Otherwise: extend P^k by node

$$j^k_x = \arg \min_{(i^k,j)\in\mathcal{A}} (a_{i^k j} + p^k_j),$$

i.e.

$$P^k = P^k \cup \{j^k_x\}.$$

If j^k_x = common destination, do nothing.

A typical iteration of the algorithm running on processor k consists of several algorithmic steps. So far, the scheme looks similar to the shared memory model. The

difference is that now, instead of updating a common price vector, each processor k sends messages to all other processors about the prices of the nodes that changed (increased) due to contraction on the path P^k. In addition, at unspecified times, processor k accepts such messages from other processors. At the end of an iteration, processor k performs an update of its local path-price pair based on the messages it has received. In particular, let us denote by T^k the set of times that processor k performs such an update. Then for $\bar{t}, t$ successive elements of T^k we denote by $B_k^m(t)$ the set of messages that processor k received from procesor m in the time interval $(\bar{t}, t]$. Let $B_k(t) = \bigcup_{m=1, m \neq k}^{r} B_k^m(t)$ be the whole collection of messages that processor k accepted in the time interval $(\bar{t}, t]$. Then the path-price pair update is as follows:

Update of the local path-price pair of processor k:

Forall $i \in \mathcal{N}$, such that a message p_i exists in $B_k(t)$ we set

$$p_i^k(t) = \max(p_i, p_i^k(t))$$

Let $J = \{j \in P^k(t)$: such that their local price changed in the update performed above$\}$.Then we set:

$$P^k = \{P^k(t) \; - \; J\} \cup \{s_k\}.$$

As it will be shown later, P^k is a connected path and the path-price pair generated this way satisfies CS.

5.2 Validity and Convergence

In order to ensure the validity of the algorithm described above, we make the following assumptions:

Assumption 1 (feasibility): There exists at least one path from each origin to the common destination.

Assumption 2 (communication reliability): Messages are not lost and arrive at the recipient in the order they were sent. The necessity for this assumption is to assure that CS will hold after a local path-price pair update. In order to see this we note that if a message for a change in the price of a node is received and an update is performed before the change in the price of its successor node is received, then the update will result in violation of the CS condition on the local path-price pair.

We now proceed to prove that CS holds throughout the algorithm for all processors in the network and that the algorithm terminates in finite time.

Proposition 5.1: CS holds for the path-price pair of every processor.

Proof: We note that we need to prove the proposition only for the times of a local price update at some processor k, since at other times, the auction algorithmic iteration maintains CS. We use an induction argument. Obviously at the beginning all local price vectors are the same and satisfy CS. We assume that upto time $\bar{t}$ all price vectors satisfy CS. We shall prove that at time

$$t = \min_{t' \in \bigcup_{j=1}^{r} T^j} \{t' > \bar{t}\},$$

say $t \in T^k$, CS holds at processor k.

Let us suppose that at time $t \in T^k$ we begin to make an update on the local price vector of processor k. We assume that the update will be performed in the following manner (we shall relax this assumption later in the proof): We pick a buffered collection of messages, say from processsor m, i.e. $B_k^m(t)$ and we assume that the messages are in the order they were generated by processsor m. The price updating

based on the first message in this buffer is, because of assumption 2, the same as taking the maximum over two price vectors: the current local price vector of processor k and the price vector of processor m at the time the message was generated. Both these price vectors satisfy CS since this is the first price update performed after time $\bar{t}$. A similar argument holds for every element of $B_k^m(t)$ since they are in the order they were generated at processor m. Similarly we proceed to exhaust all available buffered messages in $B_k(t)$. Thus at the end of the local price update CS holds for the price vector of processor k, and therefore for all processors at time t. Similarly for all update times greater that t of any processor we can repeat the above argument and prove that CS holds for their local price vectors. Now we can relax the assumption that the local update is performed based on each $B_k^m(t)$ and that the messages are taken during the update in the order they were generated/received since the overall result is the same.

In addition, the path is updated in such a way that CS holds with equality. To see this we note that after a path update, there does not exist a node j in the path $P^k(t)$ such that $(i,j) \in P^k(t)$ the price $p_i^k(t)$ increased during the update and the price $p_j^k(t)$ did not increase, since by assumption 2 the messages are received in the order they are generated and CS does holds for all the local price vectors. Thus CS holds on the updated path of any processor with equality. **Q.E.D.**

The above proof has given us the chance to see the connection of the message passing scheme for the auction and the corresponding shared memory scheme. In particular, for a processor k let $\bar{t}, t$ be two successive elements of T^k. We may consider its price vector at a time t as a common price vector produced by the maximization over the elements of the price vector at processor k and an imaginary common price

vector at time t. This imaginary common price vector at time t has resulted from the maximization over the elements of the imaginary common price vector at time $\bar{t}$, and the price vectors of other processors at the times that the corresponding received messages were generated. In the light of this equivalence, we deduce that the message passing asynchronous auction algorithm described ends in finite time. We also note that a number of price update messages could be grouped together and sent to other processors instead of sending each price update individually, so long that packets continue to arrive to the recepients in the order they were sent.

A key parameter which greatly affects the time it will take for the problem to be solved on a message passing environment is the overall delay on the links that connect the processors. This includes the reliability factor, since errors result in retransmissions thus delays. Low delays mean fast exchange of information (node prices) thus resulting to fast termination of the algorithm solving the problem. Large delays make the algorithm more slow and in the worst of cases, no information is exchanged thus each processor is running only on its own price vector and the prices it generates. Another issue is also the communication load that the network ought to be able to cope with. A rough estimate of the number of messages that the network may have to trasport at a time is order of r^2. The number is derived if we think that we have r processors each needing to send its messages to $r-1$ processors. Let us now assume that each processor k generates messages each time there is a contraction to the origin s_k it is assigned to. Each message contains the origins and their corresponding prices when a contraction was performed at them; their number being less than $\mathcal{N}$. Then a crude upper bound on the number of messages that a processor may need to generate is

$O(MD_{s_k t}L)$, where L is the maximum arc length in the network. We see that the number of messages that need to be communicated depends on the maximum arc length in the network in a pseudopolynomial way.

Finally we would like to note that the assumption of having one processor per origin may not necessarily be fulfilled. In particular the algorithmic model for the auction algorithm presented gives the flexibility to create schemes where we have processors each performing a serial algorithm from a number of origins to the common destination on its own price vector (such a serial algorithm is described in [BER90]). After a number of iterations messages for the nodes whose prices changed are exchanged thus allowing all processors share the prices of the nodes visited.

6. ASYNCHRONOUS TWO-SIDED AUCTION WITH A COMMON PRICE VECTOR.

We shall analyse here the forward - backward algorithm running asynchronously in a shared memory environment. In particular, we shall first consider the problem of one-origin, one-destination and we will assume that there are two processors one running the forward and one the backward algorithm with a common price vector for both sides of the algorithm. We shall also refer to the nodes *visited* by the forward (backward) algorithm. We say that a node has been *visited* by the forward (backward) algorithm if it has become terminal node of some forward (backward) path. Let us call V_s (V_t) the set of nodes visited by the forward (backward) algorithm. Then if the problem is feasible and CS is kept on the path-price pairs (we shall discuss how this can be done), then we can easily prove the following for integer data:

Proposition 6.1: After a finite number of forward - backward iterations, the corresponding sets of visited nodes have a non-empty intersection, i.e.

$$V_s \cap V_t \neq \emptyset.$$

Proof: Let's suppose, in order to reach a contradiction, that as time tends to infinity, the intersection of the corresponding sets of visited nodes remains empty, i.e. $V_s \cap V_t = \emptyset$. This means that the nodes in V_s (V_t) are visited by the forward (backward) algorithm an infinite number of times, and as a result an infinite number of contractions is performed by either sided algorithm. Every contraction corresponds to a price increase (decrease) of some node of at least one (for integer data). Thus the prices of the nodes in V_s (V_t) go to infinity (minus infinity). For a feasible shortest path problem and a price vector that satisfies complementary slackness, we know that

the prices are bounded. Thus, a contradiction has been reached and the proposition holds. **Q.E.D.**

The above argument can easily be extended for the threshold auction algorithm fo real data where price increase of 1 corresponds to price increase of at least h.

A direct consequence of the above proposition is that after a finite number of iterations there will be at least one arc such that one of its endnodes (say i) has its price set by the forward algorithm, and its other node (say j) has its price set by the backward algorithm.

One issue that arises directly from the above discussion is how can we maintain complementary slackness at the common price vector for the asynchronous forward-backward algorithm. We give an example to show the difficulty that arises.

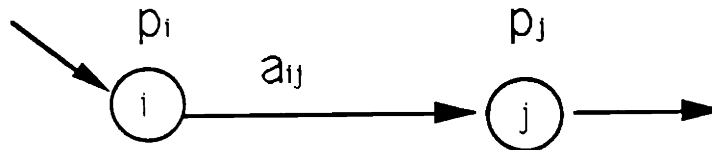


Figure 6.1: A possible situation of that can arise in the asynchronous forward backward algorithm with common prices and which results in a violation of the complementary slackness conditions on the common price vector and the forward/backward paths.

Let i be the terminal path of the forward algorithm and J be the terminal path of the backward algorithm at some time t. Let $p_i(t) = 0$, $p_j(t) = 0$. At time $t_1 > t$ the forward algorithm reads the price of node j and the backward reads the price of node i. At time $t_2 > t_1$ the forward algorithm sets $p_i(t_2) = a_{ij} + p_j(t_1) = a_{ij}$ and the backward algorithm sets $p_j(t_2) = p_i(t_1) - a_{ij} = -a_{ij}$. It is immediately evident that at time t_2 complementary slackness does not hold for the common price vector. In the

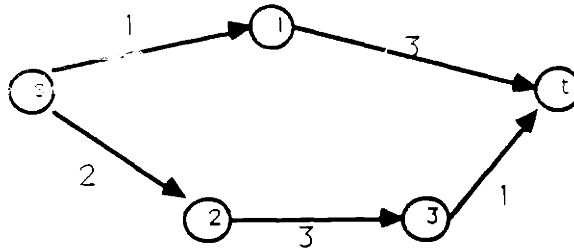
example we just presented there is no conflict arising from reading or writing on the common price vector, i.e. the problem does not have to do with the addressing of the common memory. Wet make the following modification to the algorithm:

Modification: If the terminal nodes of the forward (say i) and the backward (say j) algorithm are at a distance of one arc (i, j), then only one of the two terminal nodes sets its price and the other performs annew an auction algorithmic step on its terminal node (which has not changed since the previous iteration). For instance, in the above example we would have only i setting its price at t_2, then j would have to make a new reading of the price of i at $t_3 > t_2$ and finally perform an extension to i.

The modification that we presented and the fact that CS is otherwise maintained by each sided auction algorithm, guarantees that CS will hold at all times on the common price vector. However, this does not guarantee termination in finite time. To see this, let us consider the example of figure 6.2 where we assume that at each time increment a typical auction iteration is performed, i.e. the successors of the terminal node are scanned, and an extension or a contraction is performed. If need for the modification arises, then either a forward or a backward iteration is performed.

What causes the algorithm not to terminate, is the fact that the prices of the intermediate nodes increase and decrease in such a way that the prices of p_s and p_t do not change. In fact, a node visited by the forward algorithm is then visited by the backward algorithm and the price oscillates. To ensure that the algorithm will terminate in finite time, we propose the following scheme:

We run the forward-backward algorithm with the modification we stated in order to maintain CS. Let us assume that at some time t the backward algorithm makes an



T	s	1	2	3	t	Terminal node For	Terminal node Bac	
0	0	0	0	0	0	s	t	Bac. has not started yet
1	1	0	0	0	0	1	t	Bac. started but priority to For.
2	1	3	0	0	0	s	t	Bac restarts here
3	2	3	0	0	0	2	1	
4	2	1	3	0	0	s	t	
5	2	1	3	0	-1	1	3	
6	2	2	3	0	-1	s	2	Priority given to Bac.
7	2	2	0	0	-1	s	3	For. restarts here
8	2	2	0	-3	-1	2	t	
9	2	2	0	-3	-1	3	1	
10	2	1	0	0	-1	2	t	
11	2	1	3	0	-1	s	3	
12	2	1	3	0	-1	1	2	
13	2	2	0	0	-1	s	3	Step 13 is the same as step 7 so steps 7 => 13 will be repaeted indefinitely

Figure 6.2: A possible situation that can arise in the asynchronous forward backward algorithm with common prices is shown in this figure. We record the following: under T: the time, under s,1,2,3,t: the prices of the corresponding nodes, and finally the terminal nodes opf the forward and the backward algorithm respectively. One time unit corresponds to one scanning of the successor/predecessor nodes of the terminal node of either sided algorithm, and a correcponding extention or a contraction. The series of events and how one side is stopped in order to satisfy the modification we have put forth, are shown at the side. We note that step 13 is the same as step 7 which shows that the algorithm cycles.

extension to a node $i \in V_s(t)$ which means that $V_s(t) \cap V_t(t) \neq \emptyset$. Then the backward algorithm stops. The forward algorithm is allowed to proceed without checking if its terminal node belongs to V_t until one of the following holds:

i) a shortest path to the destination is found,

ii) a contraction to the origin with a price increase for the origin occurs. If (i) occurs then the algorithm terminates.

If (ii) occurs, say at time $\bar{t}$, then $V_s(\bar{t})$ has been extended and contains all nodes visited by the forward algorithm in the interval $(t, \bar{t}]$ thus we must set $V_t(\bar{t}) = V_t(t) - \{V_s(\bar{t}) \cap V_t(t)\}$, and we restart the algorithm from both ends. A symmetric scheme is followed if the forward algorithm makes an extension to some node $j \in V_t(t)$.

Proposition 6.2: The algorithm described above terminates in finite time.

Proof: Let us assume, in order to reach a contradiction. that the algorithm does not terminate. This however cannot happen if $V_s \cap V_t \neq \emptyset$ an onfinite number of times because then either algorithm will have performed an infinite number of contractions to the origin and/or destination with corresponding price increase/decrease. Since the data is integer the prices of the origin and the destination have a difference which tends to infinity. This is not possible since CS holds for the price vector and prices are bounded. Thus the event $V_s \cap V_t \neq \emptyset$ can happen a finite number of times. However, this implies that the algorithm will terminate in finite time because from proposition 1 in finite time after the last event $V_s \cap V_t \neq \emptyset$ we are going to have another such event if the shortest path has not been found. However, such an event leads to a price increase to the origin or a price decrease of the destination. Since the data are integer, we conclude that the event $V_s \cap V_t \neq \emptyset$ can only happen a finite number of times. **Q.E.D.**

Implementation Issues: The asynchronous two-sided scheme with common

prices needs more complex implementation than the forward only schemes. This is because the modification of the auction algorithm and the scheme whereby we perform only one-sided iterations if $V_s(t) \cap V_t(t) \neq \emptyset$, and then resume after a price increase (decrease) of the origin (destination) has ocurred, involve a lot more synchronisation than before. An idea which would lead to an efficient implementation is to keep a common set of flags (f) one for each node which can take the following values: F(orward), B(ackward), F(or.)T(erminal), B(ack.)T(erminal), and N(ot)S(et). If node i becomes terminal node of the forward path, we lock the flag of i in the sense that it can be changed only by the forward algorithm. If f_i =B then the flag is released and an iterrupt is generated so that only one side is allowed to proceed without checking the flags as described in detail above. The interrupt may be a flag or some message indicating which processor will proceed and which will stop. The processor which is allowed to proceed is responsible to signal the other processor to resume execution. If f_i =F or NS, we change it to FT, and we release the flag location (which was previously locked). We scan the successor nodes both for their prices and their flags. If there exists a successor node with flag equal to BT we perform an interrupt similar to the one before. The two interrupts, however, are different because in the former, we resume iterations from the corresponding start node, whereas in the latter we resume iterations from the node we were when the interrupt occurred. If a contraction or an extension occurs, we change the flag of node i to F. A corresponding procedure is followed for the backward algorithm.

The above scheme can be extended for solving the problem of finding the shortest path from many origins to one destination. In particular, we have many processors

executing the asynchronous forward scheme number 4 (dijoint forward paths) and a processor executing the reverse algorithm of §3.1 on the common price vector. We keep a set of flags as discussed above and each forward path we make the checking described above. Similarly we perform a corresponding checking for the backward algorithm. In particular, if node i becomes terminal node of some forward path, we lock the flag of i in the sense that it can be changed only by the forward algorithm. If f_i =B then the flag is released and an iterrupt is generated so that only one side (the asynchronous forward considered as a whole) is allowed to proceed without checking the flags as described in detail above. The side which is allowed to proceed is responsible to signal the other side to resume execution. If f_i =F or NS, we change it to FT, and we release the flag location (which was previously locked). We scan the successor nodes both for their prices and their flags. If there exists a successor node with flag equal to BT we perform an interrupt whereby only the forward or the backward path that collided continues to iterate. We would also like to add here that the three first asynchronous schemes of forward auction with common prices could be used too, but there is additional overhead incurred from copying the common price vector to all forward algorithm running processors after halting one side. Also the simulation results give for scheme 4 a speedup comparable to the other schemes (about 3 to 4) and the fact that the common price vector is updated dirctly makes it the best choice for the two-sided algorithm with common prices.

From the discussion above, it is easy to see that the two-sided algorithm with common prices involves a lot of synchronisation and thus is difficult to implement. In addition, the checking of flags, the locking of memory locations and the interrupts

result in considerable delays, which may make the scheme not very efficient even though the sharing of the forward and backward generated prices leads to a fast serial algorithm.

A Simpler Variation :

The implementation considered above for the two-sided algorithm with common price vectors. mainly stresses on the fact that no *degenerate* iterations are allowed - degenerate in the sense that both the prices of the origin and the destiantion do not increase/decrease respectively after a contraction to the origin/destination. If instead we allow such iterations to be performed we have a more easily implemented scheme for the two-sided algorithm with common price vectors. In order to ensure termination however, we must have a way of knowing that degenerate iterations occurred and then allow only one side to proceed. Here we shall not keep any set of visited nodes or flags per node. The idea will be to allow the two-sided algorithm to iterate and if both sides perform iterations with contractions back to the origin/destination with no price increase/decrease then we stop one side and allow only the other side to proceed till a contraction to the starting node of the running algorithm is reached with a price change. This can be achieved as follows: We have two flags initialised to FALSE called F, B. If, say, the forward algorithm performs a contraction to the origin with no price increase then flag F is set to TRUE (similarly it is set back to FALSE if a contraction to the origin with a price increase is performed). Flag B is read and if TRUE too, then the forward algorithm stops sets F to FALSE and waits till B is set to FALSE; otherwise the forward algorithm continues to iterate. A symmetric scheme

is followed by the backward algorithm. In order to avoid a synchronous setting of the corresponding flags and then reading of the symmetric flags -which would result to deadlocking- locking of both flags is performed so that only one side can acquire them. If a side performs a contraction to its corresponding starting node with a price change, the corresponding flag is set to FALSE. Then the algorithm restarts from both sides.

The method, as described above, is deceptively simple. This is because we were eager to assume that contractions to the origin/destination will eventually occur. However, it is easy to see an example were contractions to the origin/destination will fail to occur. Actually the example is the same as before with the difference that the nodes where oscilation occurs are now considered intermediate and not the origin and the destination.

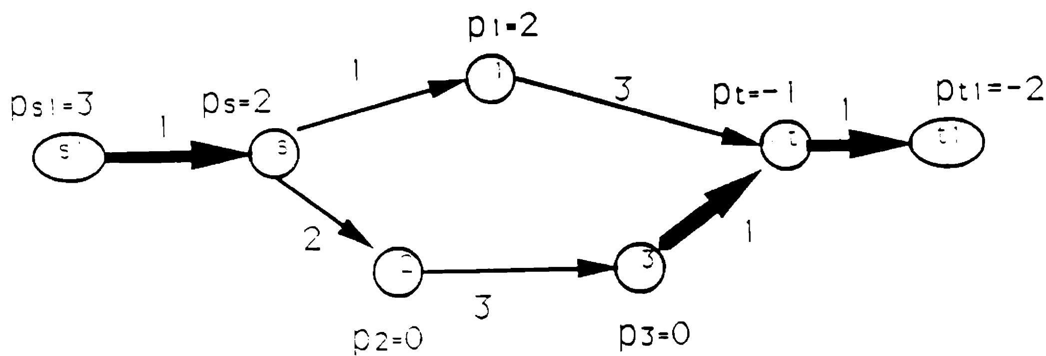


Figure 6.2.a: This figure is actually the same as 6.2 augmented with one origin s_1 and a destination t_1. It is easy to see that if we start with the prices and the paths indicated, we shall follow the same steps as in figure 6.2 and never contract to the origin s_1 or the destination t_1.

Yet the situation is easily remedied, without adding additional complexity to the problem, by doing at each node what we proposed for the origin/destination nodes.

In particular if a node is visited by the forward algorithm a number of times with no price increase and/or some other node is visited by the backward algorithm a number of times without price decrease the we set the corresponding flags the same way we described above for the origin and the destination nodes, allowing eventually one side only to proceed till the price of the origin (destination) is increased (decreased).

The validity of the algorithm follows from the fact that after one degenerate iteration a price change of the origin or the destination is guaranteed. Since deadlocking has been ruled out, and the data are integer, we conclude that the scheme ends in finite time.

The tradeoff here is that we may allow a number of degenerate iterations which may be time consuming or else stop one side of the algorithm even before it is necessary: If both sides performed a contraction to their respective starting node without a price change it may not have been a degenerate contraction, i.e. if two or more of the successor and predecessor nodes of the starting node of each side are connected to the starting node with arcs of the same length (figure 6.3). However we expect to gain a lot from the fact that we do not keep lists of visited nodes nor do we update any common memory locations that require memory locking other than the common prices. The simulation indicates a speedup of two for the one-origin one -destiantion problem.

The scheme that we just analysed is easily extended to the many-origins one-destination problem. Here we shall use again the asynchronous forward scheme 4 for the algorithm starting from the corresponding origins and the backward running on an additional processor. The simple scheme that we analysed can be implemented

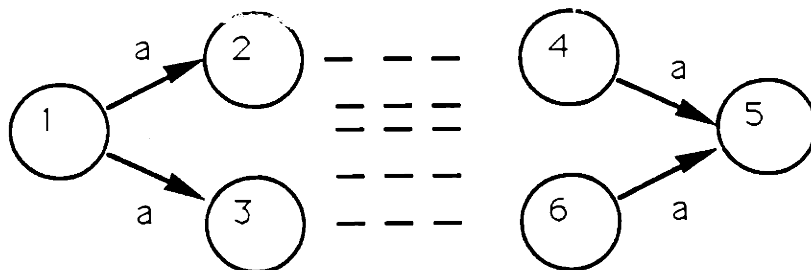


Figure 6.3: A situation that may cause unnecessary stopping of one side of the two sided algorithm.

by checking for degenerate iterations for each origin-destination pair and if so halting the forward algorithm from that origin. The backward algorithm is always allowed to proceed since it is evident that price decrease to the destination results to "progress" in finding the solution, for all the origins. The simulations indicate remarkable speedup of a factor of over 40 for the five-origin one destination problem (compared to the serial forward version) and about 3 to 4 over the serial two-sided version. This scheme is actually the most efficient by far from any other developed so far and is the one expected to have actual running time much better than any other known asynchronous algorithm. It can also be used to solve the many-origins many-destinations problem as follows: We run the many-origins one-destination two-sided algorithm for one of the destinations. When this problem is solved we can start the backward algorithm at a new origin and solve the new problem on the existing price vector calculated so far. This results in an extremely efficient scheme where all past calculations are of use

6. Asynchronous Two-Sided Auction with a Common Price Vector for each new problem.

Finally this scheme can easilv be extended for the many-origin many-destinations case where we implement a scheme like the one presentea in §3.1.2. The origins run scheme 4 and the destinations run a reverse version of scheme 4. If a destination is reached by some origin and it is not the only one iterating then it is frozen as in §3.1.2. Once only one destination is left and its termination condition is satisfied then all origins and destinations which are frozen and their termination conditions have not been satisfied, are revived. We saw from the running time examples that the serial version of this scheme was faster than the version of solving many (many-origin one-destination) problems by a factor of 2 to 3 (for up to five origin five destinations problem). If we take into account that the many-origin one-destination auction is about 3 times faster than the two-sided Dijkstra, we conclude that the serial scheme is at least 6 to 9 times faster than the serial two-sided Dijkstra. Thus our parallel scheme will be unbeatable even by the parallel Dijkstra since the latter's highest speed up cannot exceed 5 (for the 5 origin 5 destination problem). Thus the parallel version is expected to be even faster. Therefore we conclude that we have found an efficient way of solving the many-origin many-destination problem.

7. ASYNCHRONOUS TWO-SIDED AUCTION WITH DIFFERENT PRICE VECTORS.

We shall employ the two-sided algorithm with different price vectors, developed in §3.2, in order to have forward-backward schemes solving the many-origin one-destination problem both for shared memory and message passing environments. The motivation to do this comes from the fact that the two-sided algorithm presented in §3.2 is faster than the forward algorithm and thus we may expect a speed up in the asynchronous schemes for the many-origin one-destination problem.

We assume as before that we have r origins each corresponding to a processor and we wish to find the shortest path to a common destination t. There is an additional processor which runs the reverse algorithm described in §3.2 from the common destination t. All price vectors have been initialised to zero so that we can take advantage of the key property whereby nodes are visited from the starting node in their order of proximity. There are various schemes that can be implemented this way:

a) Shared Memory scheme: We have a common memory which can be addressed by the $r+1$ processors present. Each of the r processors runs the forward auction on its own price vector, whereas the $r+1$ processor runs the reverse algorithm from t with its own price vector. In addition, we keep r lists of visited nodes, denoted by V_{s_k}, $k \in \{1,\ldots,r\}$, one for each origin and one for the destination, denoted by V_t. Each time a new node is visited from some origin s_k – in the sense that it becomes the endnode of the path starting from s_k– the node is added in the visited list V_{s_k}. The easiest way to implement such a list is to keep a set of flags which will take the value TRUE if the node is visited and stay at the value FALSE otherwise. For

the reverse iteration we update in a similar fashion the set of visited nodes V_t. At times τ, i.e. each time a new node is visited or after a certain number of nodes are visited, we check whether $V_{s_k}(\tau) \cap V_t(\tau) \neq \emptyset$. If this condition is satisfied, the forward auction iterations from origin s_k can stop and processor k, corresponding to origin s_k, performs an iteration of Step 3 of the algorithm described in §3.2. In other words, if $\{n_k\} = V_{s_k}(\tau) \cap V_t(\tau)$ processor k calculates:

$$d_{s_k t} = \min_{(i,j) \in \mathcal{K}_{s_k t}} (d_{s_k i} + a_{ij} + d_{jt}),$$

where $\mathcal{K}_{s_k t} = \{(i,j) \in \mathcal{A} \mid i \in V_{s_k} \ : \ d_{s_k i} \leq d_{s_k n_k},\ j \in V_t \ : \ d_{jt} \leq d_{n_k t}\}$, and we have assumed that the distances from s_k and t are kept by the forward and backward algorithm, respectively, at corresponding arrays.

b) Message Passing scheme: The situation is similar to the one before. Here the forward iterations are performed by the r processors again each operating on its own price vector. The backward iteration is run by the $r+1$ processor which sends the nodes it has visited and their corresponding distance from t to all the other processors. It is significant that messages arrive in the order they were sent and that they are not lost because, otherwise, the calculation of the shortest path may be wrong (because not all the data from the backward iteration were received). Each of the r processsors performing the forward iterations keeps its own V_{s_k} and V_t^k. The former is created the same way in the shared memory scheme. The latter is created based on the messages that arrive to processor k from the reverse algorithm running at processot $r+1$. At times τ_k processor k checks whether $V_{s_k}(\tau_k) \cap V_t^k(\tau_k) \neq \emptyset$. If this condition is satisfied, the forward auction iterations from origin s_k can stop and processor k, corresponding to origin s_k, performs an iteration of Step 3 of the algorithm described in §3.2. In

other words, if $\{n_k\} = V_{s_k}(\tau_k) \cap V_t^k(\tau_k)$ processor k calculates:

$$d_{s_k t} = \min_{(i,j)\in\mathcal{K}_{s_k t}} (d_{s_k i} + a_{ij} + d_{jt}),$$

where $\mathcal{K}_{s_k t} = \{(i,j) \in \mathcal{A} \mid i \in V_{s_k} \ : \ d_{s_k i} \leq d_{s_k n_k},\ j \in V_t^k \ : \ d_{jt} \leq d_{n_k t}\}$, and we have assumed that the distances from s_k and t are kept by the forward and backward algorithm, respectively, at corresponding arrays.

The validity and convergence of the above schemes can be seen easily from the fact that each-sided algorithm converges and that the two-sided algorithm with different price vectors was shown to converge in §3.2. We also expect the combined algorithm to be faster than the forward-only algorithm where each processor is operating alone, since keeping and updating the lists of visited nodes incurs negligible delays, and the two-sided scheme presented in §3.2 was faster than the forward-only algorithm.

The schemes that we presented above can be trivially extended to solve the many-origin many-destination problem. In this case we have r_1 origins and r_2 destinations and each of them corresponds to a processor (total number of processors is $r_1 + r_2$). The r_1 processors run the forward auction each on its own price vector and the r_2 processors run the reverse auction of §3.2 each on its own price vector. Similar to the schemes presented in this section, we have the sets of visited nodes $V_{s_k},\ k \in \{1,\ldots,r_1\}$ and $V_{t_{\bar{k}}},\ \bar{k} \in \{1,\ldots,r_2\}$. At times τ_k processor k checks whether

$$\exists\ \bar{k} \in \{1,\ldots,r_2\} : V_{s_k}(\tau_k) \cap V_{t_{\bar{k}}}(\tau_k) \neq \emptyset.$$

If this codition is satisfied for some (one or more) $\bar{k}$, then processor k performs a Step 3 as before (for all $\bar{k}$ satisfying the condition) and then continues to perform forward iterations until the shortest paths to all destinations have been found. Each time a

path is found from s_k to some destination $\bar{k}$, we need not check again whether a newly visited node from s_k belongs to V_{t_k}. A similar checking can be likewise performed by the reverse asynchronous algorithm.

The decoupling of the forward and backward price vectors results to a scheme where we can solve the problem with as many processors as the total number of origins and destinations involved (order of n processors) without any of the sychronisations involved in the two-sided scheme with common prices. This, however does not mean it is faster because the two-sided auction with common prices improves dramatically the complexity of the problem as we shall see in §8.

In conclusion, we see that the two-sided algorithm of §3.2 gives a lot of flexibility in developing asynchronous forward-backward schemes. The latter algorithm can also be implemented in a serial environment. We also note that these two-sided schemes are easy to implement and do not need the synchronisations of the scheme in §6 but of course lack the sharing of forward and backward generated prices which is a unique feature of the two-sided auction with a single price vector. We also note that these schemes can be implemented with any shortest path algorithm that visits nodes in the order of their proximity to the starting node (i.e., the Dijkstra algorithm).

8. EVALUATION OF ASYNCHRONOUS AUCTION SCHEMES.

8.1 Complexity Issues.

Here we shall give various examples exploring the performance of the asynchronous schemes that where discussed in earlier chapters. Our aim is to see whether the parallel implementations result to algorithms with better complexity in soving certain problems. We start off with the simple case of the network shown in figure 8.1. It is a chain network. The serial algorithm starting from node s will take $\frac{n(n+1)}{2}$ iterations (contractions and extentions) to find the shortest path to t. For the forward only parallel algorithm it is rather difficult to visualise what is happening, so for simplicity let us have in mind method 4 of §4.4 where each origin keeps a disjoint path. Then it is easy to see that for n processors each corresponding to an origin the algorithm will take only n iterations. Our example results in a polynomial algorithm even for the auction algorithm. Parallelism resulted in a decrease of the order of the polynomial complexity.

A second example considers the case where the auction algorithm displays a pseudopolynomial behaviour. As shown in figure 8.2 the graph contains a loop of legth 1 (for simplicity). We denote by n_1 the number of nodes from s upto the first node of the cycle, and by n_2 the number of nodes that create the cycle. The forward auction algorithm starting from s will need $\frac{(n_1+n_2)(n_1+n_2+1)}{2}+2(n_1+n_2)(R-1)$ iterations before the destination t is reached. The asynchronous parallel auction with disjoint paths starting from every node will take n_2R+n_1 iterations to find the shortest path. We see that

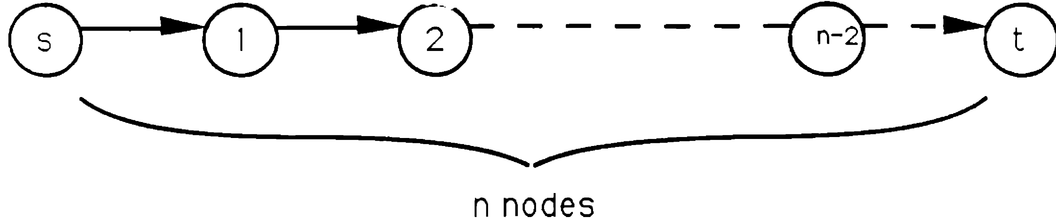


Figure 8.1: A chain network with n nodes.

the algorithm is still pseudopolynomial but the coefficient before the pseudopolynomial factor is considerably smaller. For the two-sided algorithm with common prices, where we have $n_1 + n_2$ processors running the forward algorithm with disjoint paths and one processor running the backward algorithm from t we see that the complexity is polynomial. The backward algorithm immediately finds the shortest path over the arc of length r so the algorithm ends in no more than $n_1 + n_2 + 1$ iterations. The same behaviour is displayed by the two-sided algorithm with different price vectors for the same parallel machine as above. Thus for the network shown the two-sided algorithm reduces the complexity to polynomial of degree 1 from pseudopolynomial, which is considerable improvement in complexity (at the cost of a big number of processors).

Finally let us consider the following example which corresponds to a more complicated network where there are two loops on the origin-destination path (figure 8.3). Let us denote by n_1, n_2 the number of nodes in each of the loops. The auction

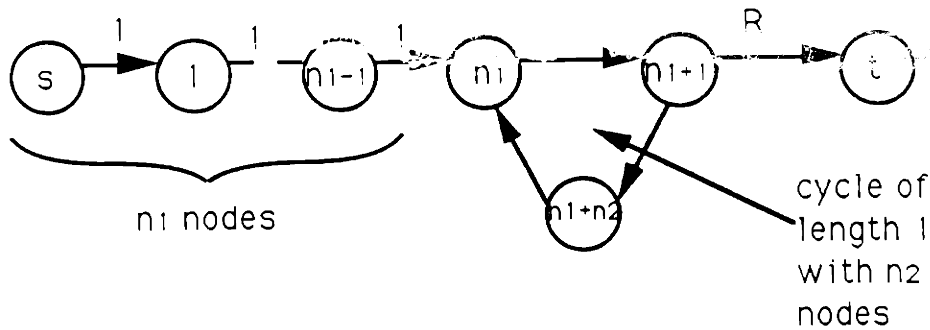


Figure 8.2: A network with one loop for which the auction algorithm is pseudopolynomial.

algorithm starting from s will take $O(2(n_1+n_2)R)$ iterations. The parallel forward algorithm will take $O((n_1+n_2)R)$ iterations and the two-sided algorithm with common prices will take $O(\frac{(n_1+n_2)R}{2})$ iterations. In all cases the complexity is pseudopolynomial. However, it took a much more complicated graph to display such behaviour for the two-sided algorithm and even then the parallelism in all cases resulted to a considerable reduction to the number of iterations.

8.2 Simulation and Results.

In this section we will refer to the assumptions and implementation of the simulations of the various schemes. We have tried to take into account all the issues that may result in overhead in calculations based on the features of each of the schemes. However, most of our comparisons do not involve actual running times. Instead they are based on the number of iterations needed to solve the problem. Although this may seem

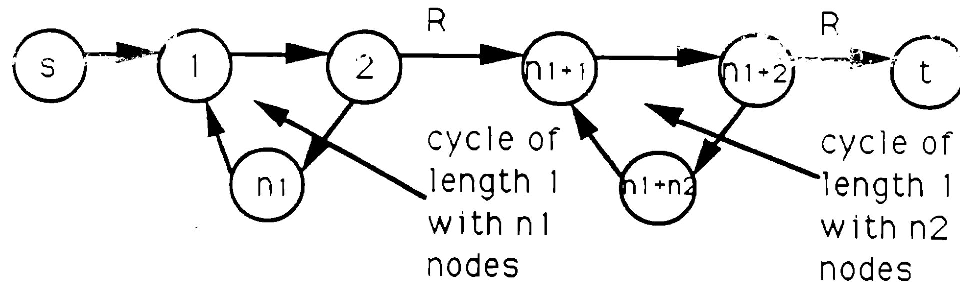


Figure 8.3: A network with an with two loops with n_1, n_2 nodes respectively for which pseudopolynomial behaviour is displayed for the auction and the asynchronous one-sided and two-sided auction.

(and is) a rough estimate of the performace of the schemes, we are sure that it reflects their actual performance on a parallel machine since running time is proportional to the number of iterations. Furthermore, the speedup we estimate from our simulations is so big in some cases, that we expect the performance of the algorithm on a parallel machine to be undoubtedly remarkable.

8.2.1 Overview of the Algorithms and Notation.

Before we proceed in the presentation of the simulations, we shall make an overview of the algorithms and give some descriptive abbreviations that will appear in the graphs and their captions.

Forward Schemes (analysed in §4, §2)

Asynchronous Scheme 1: We have r processors, each corresponding to an origin, which are performing forward auction iterations on their own local price vector. Af-

ter a certain number of iterations some processor updates the common price vector and copies it to the corresponding local price vector according to the Maximum price rule. If the common memory is busy the processors that want to perform a common price update and copy enter a first-in-first-out (FIFO) list and wait. Only one processor is allowed to perform a common price update and copying at a time. We may refer to this scheme either as scheme 1 or as ASynchronous 1 UPdate Wait (AS 1UP W, AS 1UP WAIT), sometimes followed by a number showing how many node prices can be updated/copied during an iteration.

Asynchronous Scheme 2: Similar to scheme 1, we have r processors, each corresponding to an origin, which are performing forward auction iterations on their own local price vector. After a certain number of iterations some processor attempts to update the common price vector and copies it to the corresponding local price vector according to the Maximum price rule. However, if the common memory is busy the processors that want to perform a common price update and copy do not wait but go back and perform more forward iterations and attempt again a common price update and copying later. Again, only one processor is allowed to perform a common price update and copying at a time. We may refer to this scheme either as scheme 2 or as ASynchronous 1 UPdate NO Wait (AS 1UP NW, AS 1UP NWAIT), sometimes followed by a number showing how many node prices can be updated/copied during an iteration.

Asynchronous Scheme 3: Similar to schemes 1 and 2, we have r processors, each corresponding to an origin, which are performing forward auction iterations on their own local price vector. After a certain number of iterations some processor attempts

to update the common price vector and copies it to the corresponding local price vector according to the Maximum price rule. Here we assume the existence of a memory server which can serve many processors simultaneously for their common price updating and copying. The server first updates the common price vector according to the Maximum rule then generates the paths and finally performs the copying of the common prices to the local price vector. Each processor is served with an a priori guaranteed rate no matter how many processors are served. We may refer to this scheme either as scheme 3 or as ASynchronous Many UPdate (AS MUP), sometimes followed by a number showing how many node prices can be updated/copied during an iteration per processor served.

Asynchronous Scheme 4: For this scheme we have r processors, each corresponding to an origin, which are performing forward auction iterations by updating directly a common price vector. In order to maintain complementary slackness and ensure the validity of the scheme, we recall that the paths generated by the processors at some time instant must be disjoint and this property should hold throughout the algorithm. We shall not use any special abbreviation for this scheme so we will refer to it as scheme 4.

Asynchronous Alone: We have r processors, each corresponding to an origin, which are performing forward auction iterations on their own local price vector. No information is communicated between the processors. This is not a special asynchronous scheme since it can be implemented for any shortest path algortihm. We shall refer to it for comparison with the performance of the asynchronous schemes that we will be testing.

Serial: This is the scheme for many origins and one destiantion developed in [BER90] and is run on a serial environment. We shall refer to it as Serial or Forward Serial and use it just for comparison with the performance of the asynchronous schemes that we will be testing.

Two-Sided Schemes (analysed in §6, §9)

Asynchronous Scheme 1: We have one origin and one destination and we run the simple variation of the two-sided scheme developed in §6 whereby we are carefull in price setting when the two sides are one arc apart and only a certain number of degenerate iterations is allowed. We refer to it as asynchronous two-sided or asynchronous forward backward scheme for one origin and one destination.

Asynchronous Scheme 2: We have many origins and one destination. The origins are performing forward scheme 4 above while the destination is performing the reverse algorithm. We again implement the simple variation of the two-sided scheme developed in §6 whereby we are careful in price setting when the two sides are one arc apart and only a certain number of degenerate iterations is allowed. Also according to forward scheme 4, the paths from the origins are disjoint. We shall refer to this scheme as asynchronous two-sided (or forward backward) many-origin one-destination.

Asynchronous Scheme 3: We have many origins and many destinations each corresponding to a processor. The origins are running a forward scheme 4 and the destinations are running scheme 4 for the reverse algorithm. Apart from the issues of the simple variation of the two-sided scheme developed in §6, we have to take

into consideration the issues involved for the many origins/destinations scheme of §3.2.2. No simulation was done for this scheme.

Serial algorithms: These are the two-sided serial algorithms for one origin/one destination and for many origins/one destination which are developed in [BER90]. We just refer to them as serial algorithm and it is evident from the context (i.e., if the rest of the algorithms tested are many origin many destination algorithms) the one to which we are refering. For many origins and many destinations we split the problem to many (many origins one destination) problems and we still call it the serial algorithm.

Serial algorithm many origin/destinations: This algorithm is the one presented in §3.2.2 where we freeze destinations until we are left with a many origin one destination problem. Then following the rules of §3.2.2 we revive all frozen nodes and we start with the reduced problem. We shall refer to it as SERial Many Origins/Destinations (SER. MO/D).

Finally we note that in the thesis we have developed two-sided schemes for message passing and for common memory using the two-sided scheme with different price vectors of §3.3. Also we we have developed forward only schemes for message passing. We did not test these schemes because, as it has become evident from the theoretical analysis, they cannot outperform the asynchronous two-sided schemes with common prices that we developed.

8.2.2 Simulation Assumptions, Implementations and Results.

Simulation 1: We shall present below the issues involved in the simulation of the 3 initial schemes of the asynchronous auction algorithm solving the many origin - one destination problem. Initially, we shall describe the parameters of the simulation so as to ensure that the results that we will get are realistic and represent a measure of the performance of the algorithm. First of all the clock resolution of the Macintosh II is very low so we have to define a clock tick. In the simulation that we implemented, a clock tick corresponds to a path extension or a contraction of the generic auction shortest path algorithm. In particular, in clock tick the generic auction shortest path algorithm scans the successors of the terminal node of the path, and performs an extention or a contraction. In order to provide a good measure of the delays incurred from updating and copying the common price vector, we must specify the maximum number of prices that can be updated or copied at a clock tick. Moreover in order to give the simulation the flexibility to provide results for different schemes we introduced the following parameters:

a) NUMIT : is the minimum number of price changes of the local price vector at a processor that must be made, before the processor is allowed to make a common price update. In order to keep the code simple a common price update is performed when the path kept on that processor has contracted to the origin (or it has found the destination), thus no path update is needed after a common price update. In this setting, NUMIT just provides a lower bound on the generic auction iterations performed by some processor. The number of iterations before a common price update

is otherwise random. Sometimes it appears on the figure captions as (ITER).

b) COPY : is an array whose elements are the maximum number of common price updates or copies a processor can make in a clock tick. In order to allow for different processor needs at time and different memory speeds, COPY is calculated at the end of each price updating cylcle of a processor, to be random in the range of some lower bound value BASE and some upper bound (BASE+FACTOR*BASE) where the parameters presented are set by the user of the simulation. The elements of copy can be set to be constant by choosing FACTOR = 0.

The number of common price updates or copies that can be performed in a clock tick is, in essence, a hardware problem reflecting how fast or slow the memory addressing is. Thus the delay incurred if many processors want to perform a common price update depends on how fast or slow the memory is. If for example the copying of memory can be performed in blocks, then the delay is minimal. Our simulation allows the user to make the allowed number of updates big or small thus giving a sense of the performance of the algorithm in a variety of possible systems. We would like to note however, that the delay incurred from common memory updating and copying must be related to the amount of time for an auction iteration. More specifically, for a relatively sparse network where there only 2 outgoing arcs from each node, in one clock tick the auction algorithm in the simple form described, performs about 50 memory addresses and updates without including the comparisons. Thus it would make sense to assume that for such a network, about 50 common prices are updated or copied in a clock tick. The simulation that we have implemented gives the possibility of having this number constant or varying the above number randomly with around

certain mean we want. By increasing or decreasing the range (deviation) around the mean value we can get a measure of the effect of such varying on the performance of the algorithm.

In our simulation a processor can be in four states: iterating (state 0), performing a common price update (state 2), waiting to perform a common price update (state 2 in a FIFO wait list), or finished (state 5). We have implemented the simulation of the first three forward schemes that we summarised in §8.2.1. We describe briefly how the simulation treats each problem. The first one is the waiting scheme. Each processor corresponds to an origin and performs iterations on its own price vector. After a number of iterations (determined in the simulation by the variable ITER) a processor attempts to make a common price updating and copying. In the simulation if more that one processors want to perform a common price update simultaneously, only one is picked and the rest enter a FIFO list. Similarly if the common memory is being updated by some processor all other processors that attempt common price updating and copying enter the FIFO waiting list. The second scheme we simulate is the non-waiting scheme. When a processor wishes to perform a common price updating and copying (determined again by the variable ITER) then it checks whether the memory is free and if not it continues to iterate, then check again some time later and so on. Finally we implemented the third scheme with the memory server. For that scheme processors enter a waiting list if the memory is busy but all the waiting processors are served at the same time once the memory is free. The copying-updating rate (nodes per clock tick) is the same for each processor and determined by the parameter COPY. Whenever there is waiting, the delays are recorded and added to the total time till

the corresponding shortest path is found.

At this point we would also like to note that in our simulation a processor performs a common price updating (but no copying) after the shortest path from its corresponding origin has been found and the delay incurred is included in the running time of that processor. This is done first because the computations of other processors will be facilitated by this communication of information, and because, in a sense, the common price memory is the summary of the results of the asynchronous algorithm since monitoring each processor individually in a real asynchronous environment is tedious or even impossible. Another feature of the simulation is that if a processor is the last to iterate still, it need not make more than one common price update since such an update would offer no new information to speed up the calculations.

Results of Simulation 1: We shall summarise the results that we derived from simulation 1 and are evident from the figures that follow. As a general note, even for the simulations to follow, the results presented in the figures correspond to randomly generated problems the details of which are described for each one separately. Also as a rule in the presentation the times of the serial algorithm are put in ascending order to simplify the figures. Figure 8.4 shows the results of the simulation for 10 randomly generated problems of finding the shortest paths from 5 origins in a connected network of 1000 nodes and two thousand arcs. The network was generated by the NETGEN program which creates connected graphs for general minimum cost problems. This is done by first creating a tree of forward arcs from node 1 to the final node (highest numbered node) and then arcs are added randomly to meet the specified number required. The figure compares the running time (in number of clock ticks, i.e. auc-

tion itertions) the running time of the serial forward algorithm with common prices (denoted as serial) with the running time of the waiting scheme (ASYNC) as we vary the number of node prices updated or copied in a clock tick. Also as a measure of efficiency of the sharing of information via the common price vector, we plot for each problem the maximum running time of the processors if they iterate without copying or updating the common price vector (described in the figure as ALONE). As we see we have considerable speed up of a factor of 2.2 over the serial algorithm with common prices and a factor of 2 over the algorithm running on each processor alone without exchange of information. This speed up is when we have 50 or 100 (constant) number of prices being updated or copied at a clock tick. In the next two figures we make similar plots where we vary the number of allowed updates and copies per clock tick randomly around 50, we note that as the variance increases the performance of the algorithm deteriorates. In figures 8.6 and on, we make similar plots. Scheme 1 is usually denoted as AS(ynchronous) 1 processor UP(dating) with W(aiting) followed by the number of allowed prices updated or copied in a clock tick (variable COPY). Scheme 2 is denoted as AS(ynchronous) 1 processor UPdating with No W(aiting) followed by the number of allowed prices updated or copied in a clock tick (variable COPY). Scheme 3 is denoted by AS(ynchronous) M(any processors) UP(dating), again followed by the variable COPY.

There are more results on different networks as described by the captions, and for more than five processors (figures 8.8, 8.9, 8.12, etc.). We see that by increasing the number of processors we have faster convergence (speed up factor between 6 and 8). However, as the processsors increase cases of "starvation" for the second scheme

appear (as described in §4.4) and it is evident that after a certain number (depending on the problem) further increase in the number of the processors deteriorates the performance of the algorithm (scheme 1).

We would like now to make some general comments about the results that we got from the simulation for the first three schemes. The effect of small updating-copying rates results to huge delays as predicted theoretically. Furthermore, the cases of starvation because of increased number of processors for scheme 2 where as expected from the theoretical analysis. Also the speedup that we got was considerable and quite encouraging about the performance of the algorithm in a real serial environment. Actually the speedup shown by our results is rather conservative because it is in terms of iterations and not in actual processor time. Also the common price vector was copied as a whole and not by keeping lists of nodes for which the common price vector changed since the last update as described in §4.4.

Simulation 2 and Results: The fourth scheme of asynchronous forward auction with a common price vector, where the paths starting at different origins are disjoint, was also implemented. The simulation did not involve any assumptions about the speed of memory addressing since the common memory is updated directly. The results show that this scheme is equally good to scheme 2 which as we said achieved the maximum speedup in the above simulation. This is because the delay from waiting for a node belonging to an other path is compensated by the fact that the change in price for that node is actually "progress" for all paths that wanted to extend to it (figures 8.15, 8.16). Again as a rule in the presentation, the times of the serial algorithm are put in ascending order to simplify the figures.

Simulation 3 and Results: The two-sided auction with common prices solving the one origin one destination problem was also simulated for the simple variation described in §6, whereby we allow a few degenerate iterations and then we stop one side till contraction to the corresponding starting node with price change is performed. This scheme, as expected from the theoretical analysis, was sensitive to how tolerant or intolerant we were about degenerate iterations. The simulation showed that if we do not allow more than one or 2 degenerate iterations, the resulting scheme achieves no speedup, in fact it is as slow or even worse than the serial. However if we allow upto 5 degenerate iterations for the intermediate nodes (possibly corresponding to 5 different paths of the same length) and upto 3 degenerate iterations back to the origin/destination, the scheme becomes very fast achieving a speedup of 2 over the serial algorithm (figures 8.17, 8.18, 8.19). Also as a rule in the presentation the times of the serial algorithm are put in ascending order to simplify the figures.

Simulation 4 and Results: We also implemented the many-origin one-destination two-sided scheme with common prices by combining the simulations of scheme 4 of §4 and the algorithm of §6 as described briefly at the end of chapter 6. Now the speedup was remarkable. The combined efficiency of the forward scheme and of the two-sided scheme gave an extremely fast algorithm having a speedup over the FORWARD serial of over 40 for the 5 origins case and a speedup of about 3 to 4 over the serial two-sided auction (figures 8.20, 8.21). This is the best algorithm that resulted from all the analysis of asynchronous schemes for the auction shotest path algorithm. Also as a rule in the presentation the times of the serial algorithm are put in ascending order to simplify the figures.

Thus as a conclusion we would say that the asynchronous schemes developed in this master's thesis are quite efficient and easily implemented. The running time of the asynchronous two-sided schemes is by far the best and the actual performance on a parallel machine is expected to be at least equally good. Further testing, however, is needed in order to arrive at conclusive results about the efficiency of the schemes in comparison to existing parallel algorithms. Yet, we are quite confident that the two-sided schemes developed in this thesis will have comparable or even better performance since the serial algorithm seems to outperform its fastest rivals ([BER90]).

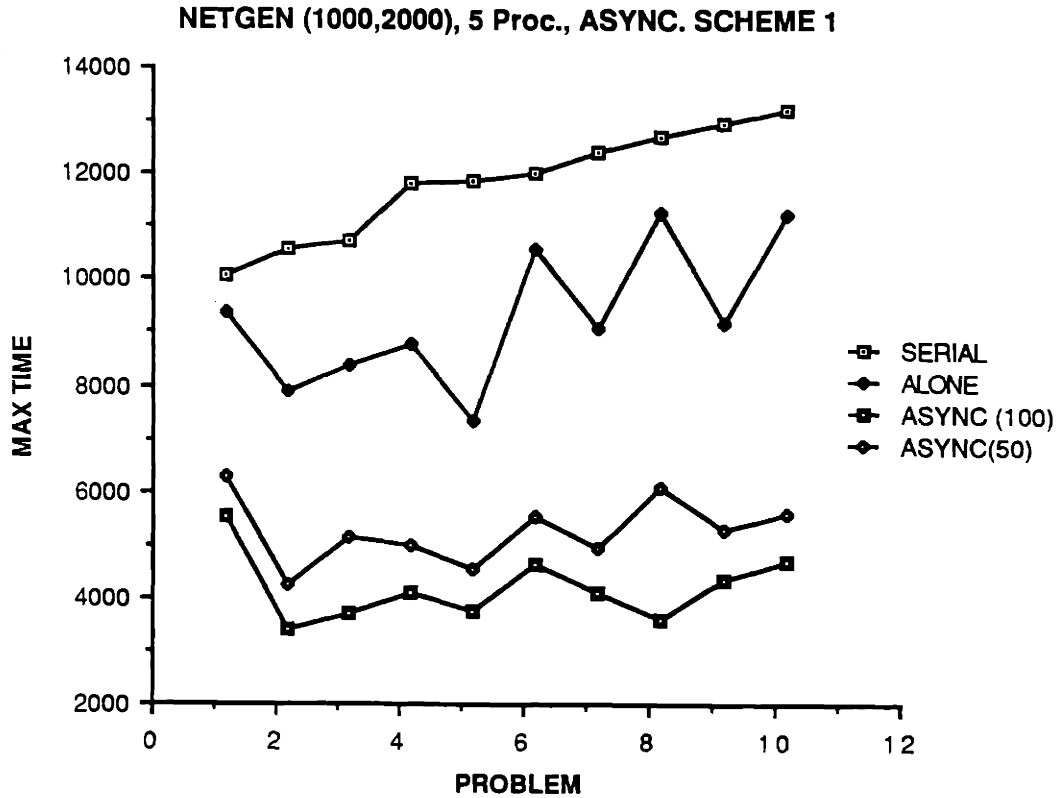


Figure 8.4: Comparison of the serial common memory running time with the running time if each processor is alone and the running time for the asynchronous algorithm in a common memory environment if the maximum allowed memory references per clock tick are 100 or 50 (constant). The results of the simulation refer to 10 randomly generated problems of finding the shotest paths from 5 origins in a NETGEN network of 1000 nodes and two thousand arcs.

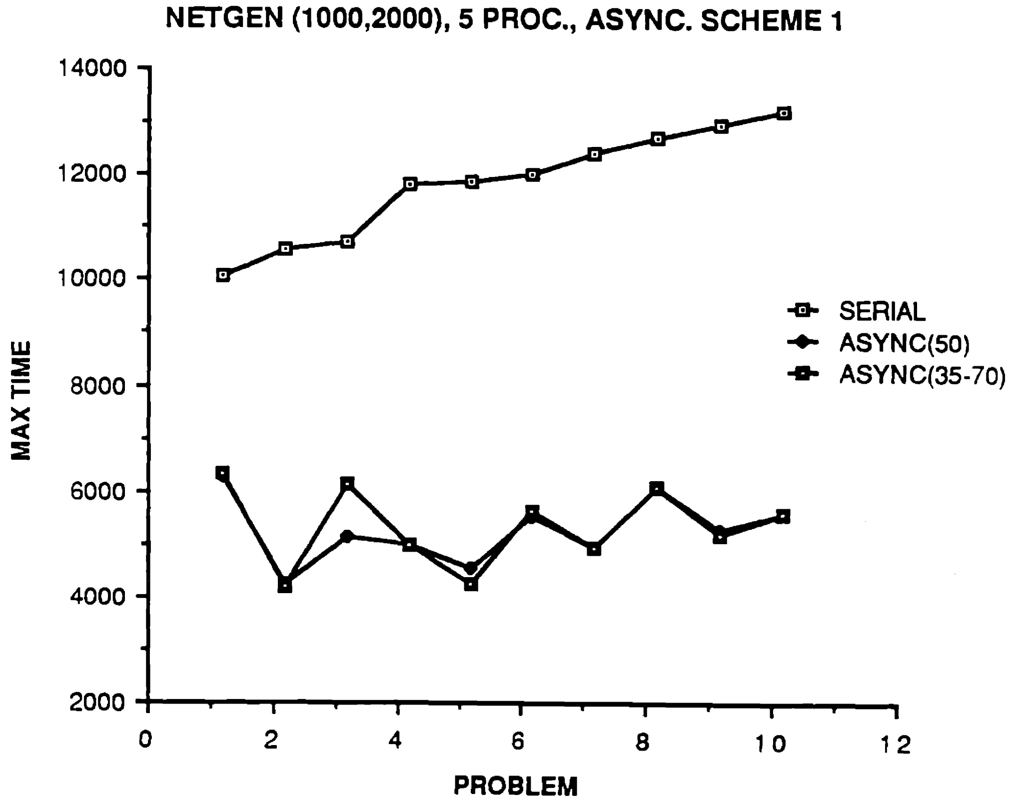


Figure 8.5: Comparison of the running times for the asynchronous algorithm where the number of updates is 50 constant and when it varies with mean 50 between 35 and 70. It is evident that the deviations from the mean increase the maximum running time of the asynchronous algorithm. Yet, the deviations from the mean are still small and the running times are still about half those of the serial. The results of the simulation refer to 10 randomly generated problems of finding the shotest paths from 5 origins in a NETGEN network of 1000 nodes and two thousand arcs.

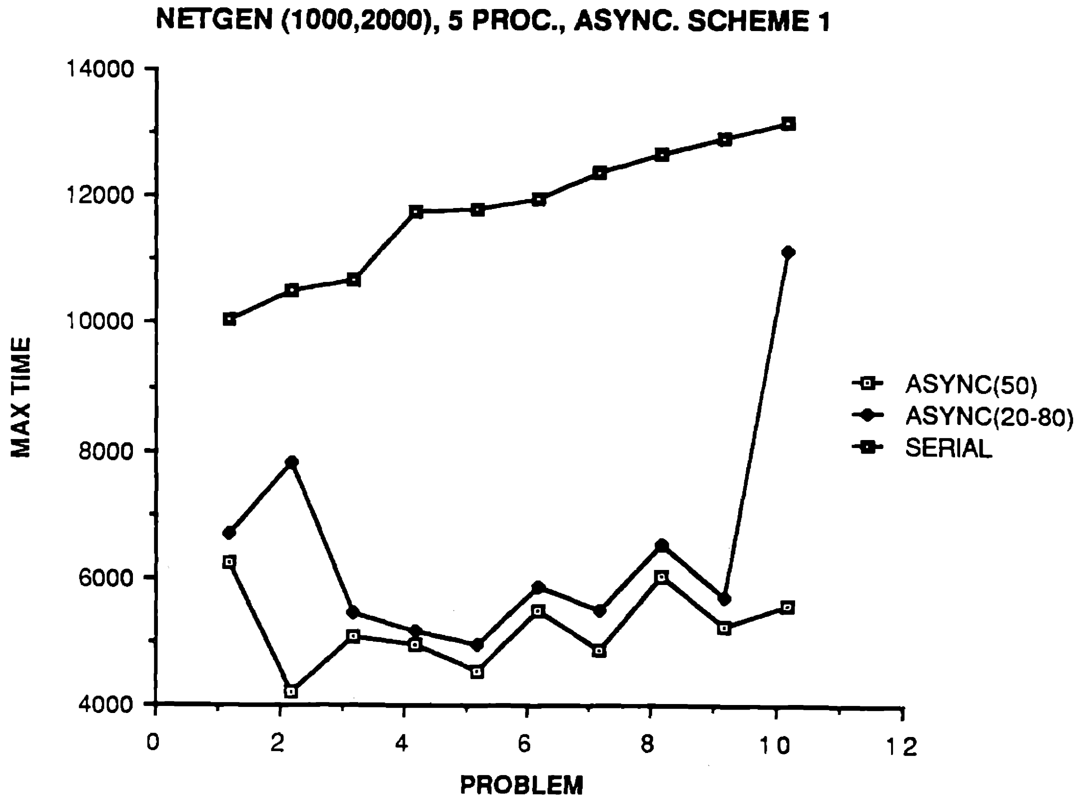


Figure 8.6: Comparison of the running times for the asynchronous algorithm where the number of updates is 50 constant and when it varies with mean 50 between 20 and 80. It is evident that the deviations from the mean increase the maximum running time of the asynchronous algorithm.Now, the deviations from the mean are big and the running times deteriorate a lot. In a sense what we see is that really low copying rate (20 per clock tick) results in such a long cummulative delay that it is not remedied by a fast copying rate at a later time. This is because a long updating-copying period of some processor results in a having many (even all) processors in the waiting list.The results of the simulation refer to 10 randomly generated problems of finding the shotest paths from 5 origins in a NETGEN network of 1000 nodes and two thousand arcs.

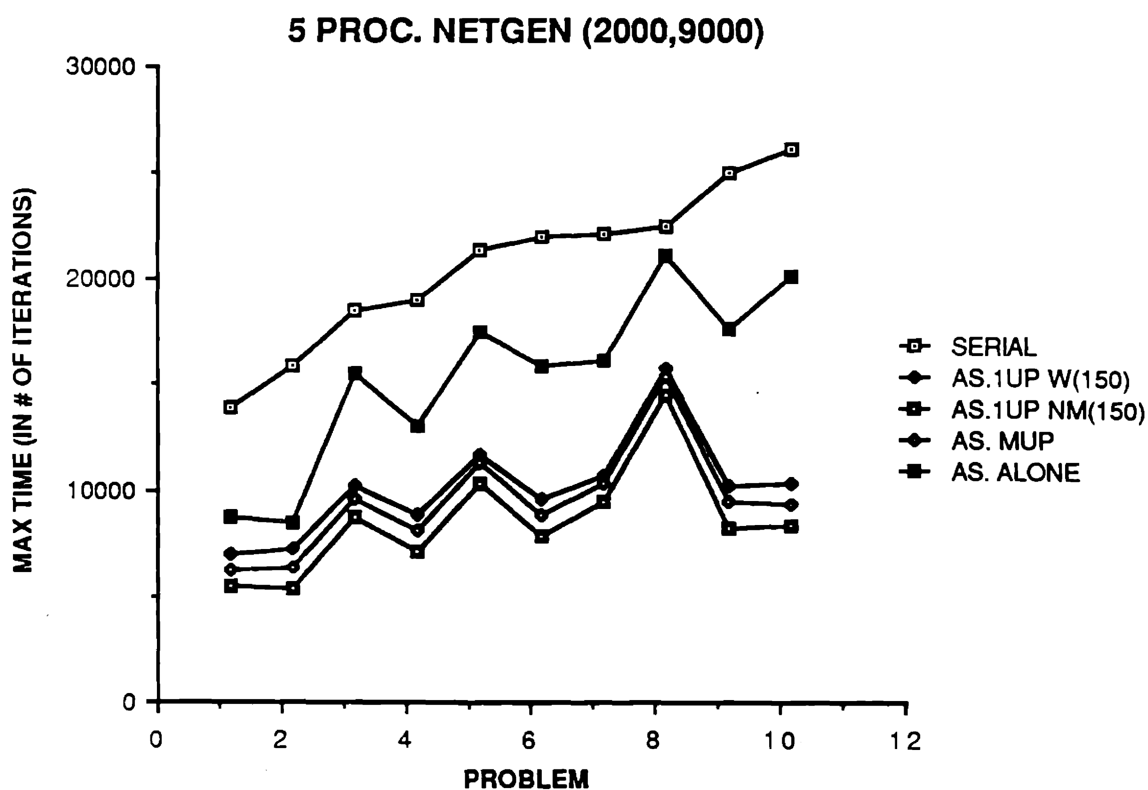


Figure 8.7: This figure shows the results obtained from the simulation when we have five origins. Ten different problems (i.e. 5 different origins each time) on the same network generated by NETGEN with 2000 nodes and 9000 arcs were simulated. Since there are between 4 to 5 arcs on the average originating from each node we may assume that 150 nodes are copyied/updated per clock tick. The speedup we got from the best scheme (scheme 2) was 3 to 4.

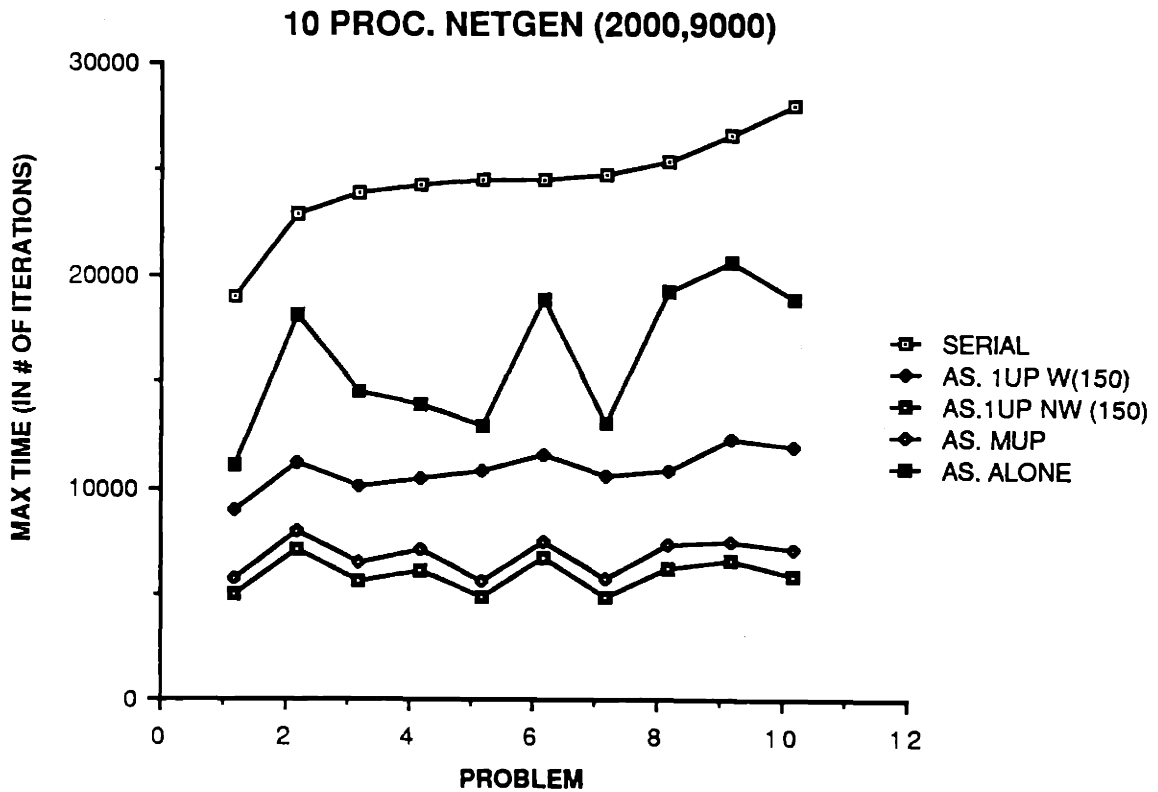


Figure 8.8: This figure shows the results of the simulation when we have 10 origins each assigned to one processor. Ten randomly generated problems were simulated on the same NETGEN network as the previous figure. As we see the speedup is greater. The best scheme (scheme 2: 1 updating but no waiting) achieves a speedup of about 5 to 6. The number of nodes copied/updated per clock tick is the same as in the previous figure since we are dealing with the same network.

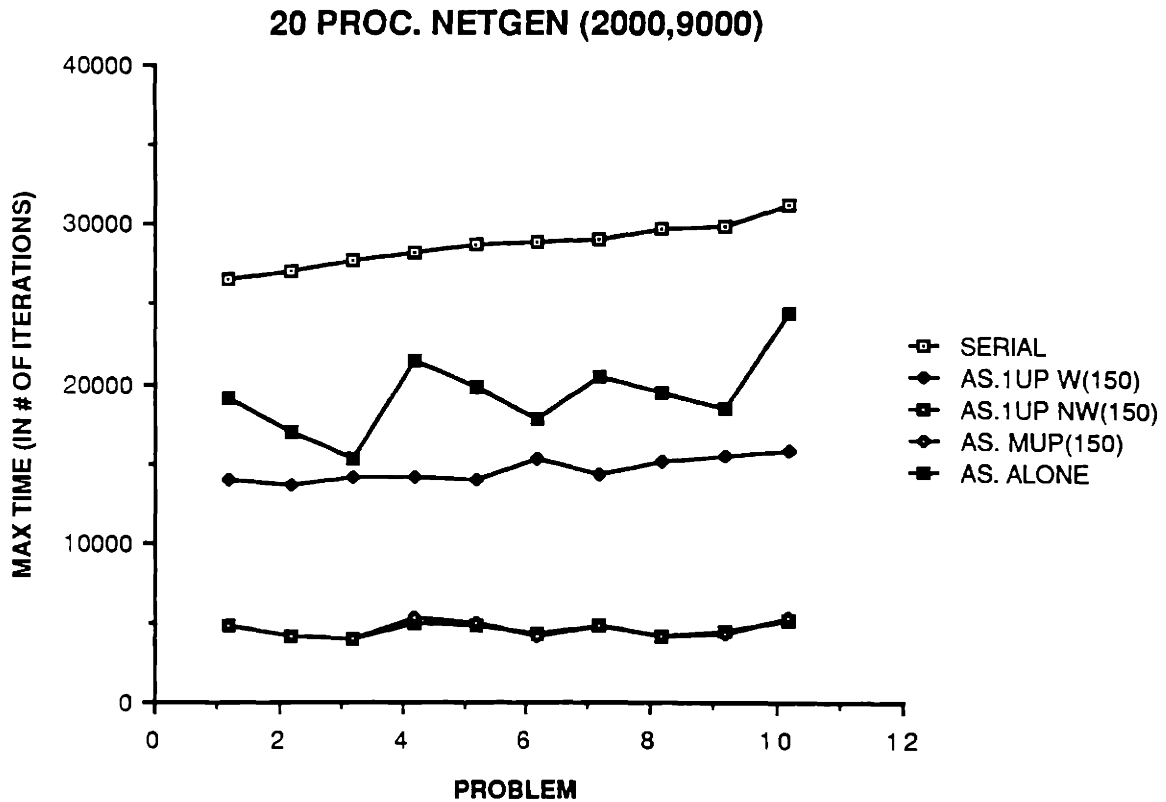


Figure 8.9: The results of the simulation when we have 20 origins each assigned to one processor are shown here. Ten randomly generated problems were simulated on the same NEGEN network as the two previous figures in oder to be able to make comparisons. The speedup here is greater now for scheme two, about 7. However the difficulties that schemes 1 runs in are evident. The delays incurred from waiting in a list with possibly 10-20 processors ahead lead to long running times and the speedup for scheme 1 is comparable to having the processors running alone without exchange of information. Also scheme 3 (with the memory server) gives good results but the memory server is assumed to be very efficient serving upto 20 processors at a time with a guaranteed rate for each one of them (150 nodes copied/updated per clock tick).

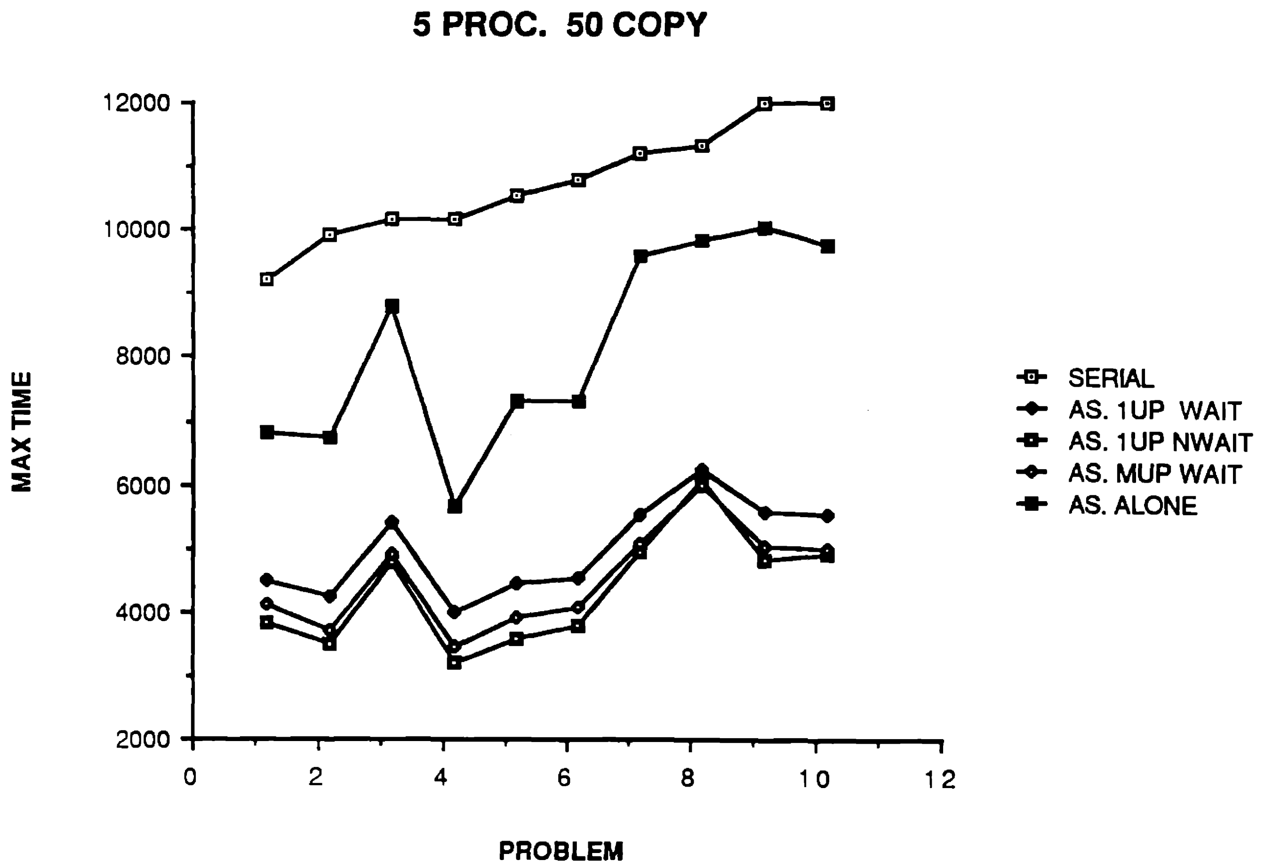


Figure 8.10: Results of similar simulations as before on another NETGEN network (1000,3000) for five origins. The speedup achieved is the same as before (between 3 and 4).

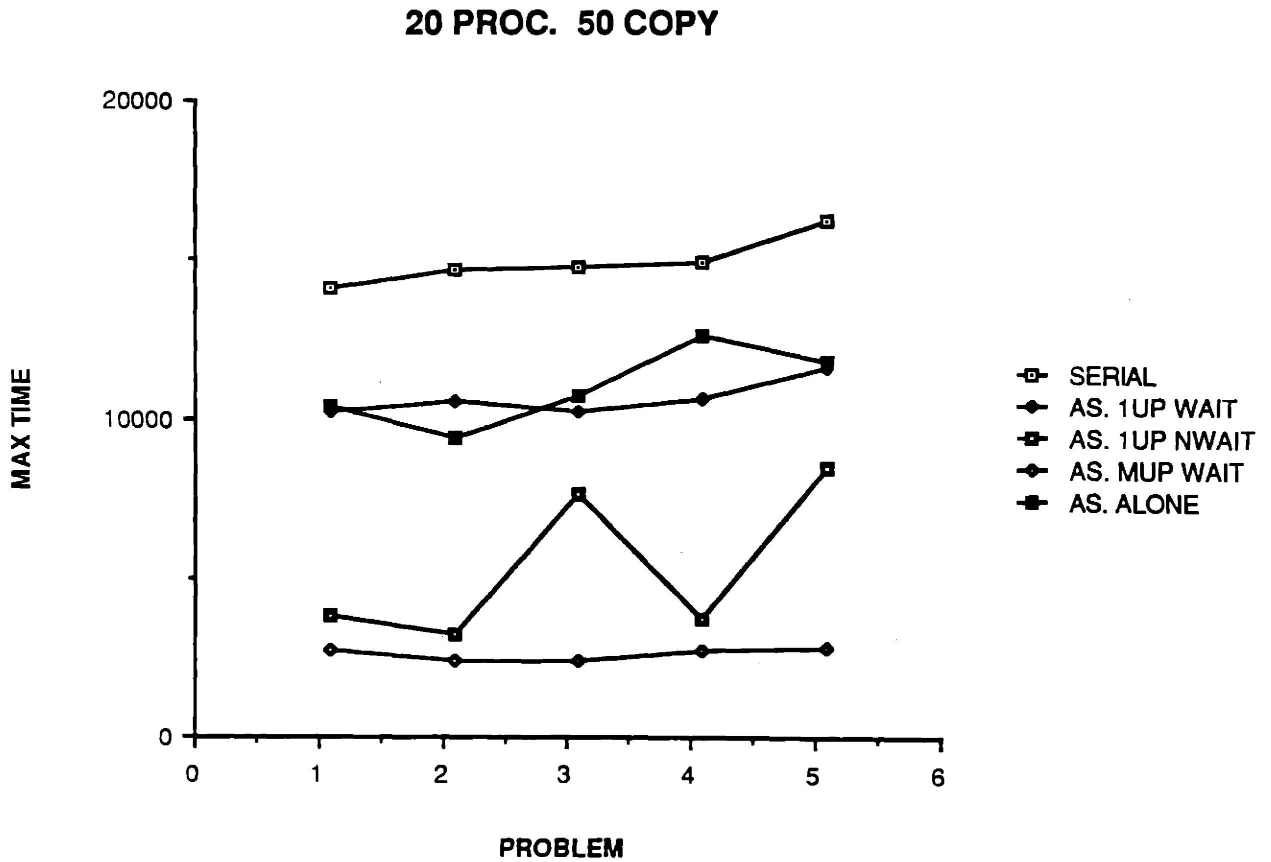


Figure 8.11: Results of similar simulations as before on another NETGEN network (1000,3000) for twenty origins. The speedup achieved is the same as before (about 7).

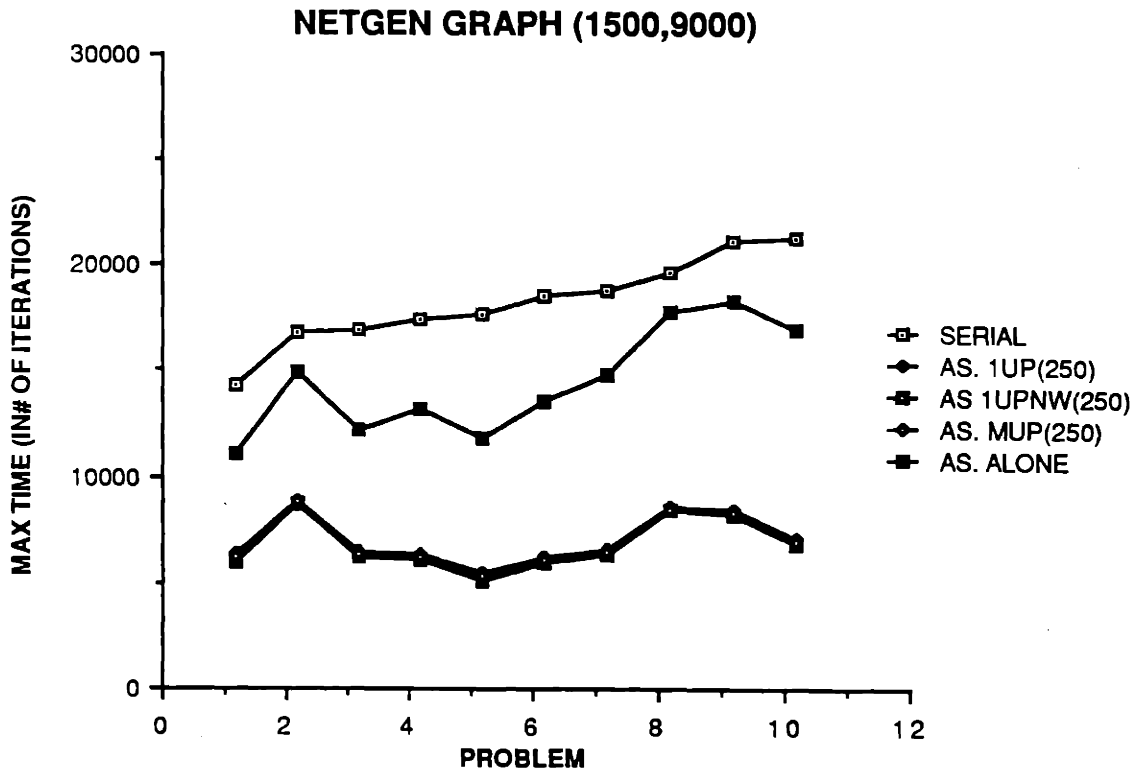


Figure 8.12: This figure and the next show the effect of the rate of nodes copied/updated per clock tick on the efficiency of the schemes. We see that for large rate (250 for this network), the three schemes have identical performance. When the rate is smaller differences appear and the non-waiting scheme (scheme2) has better performance.

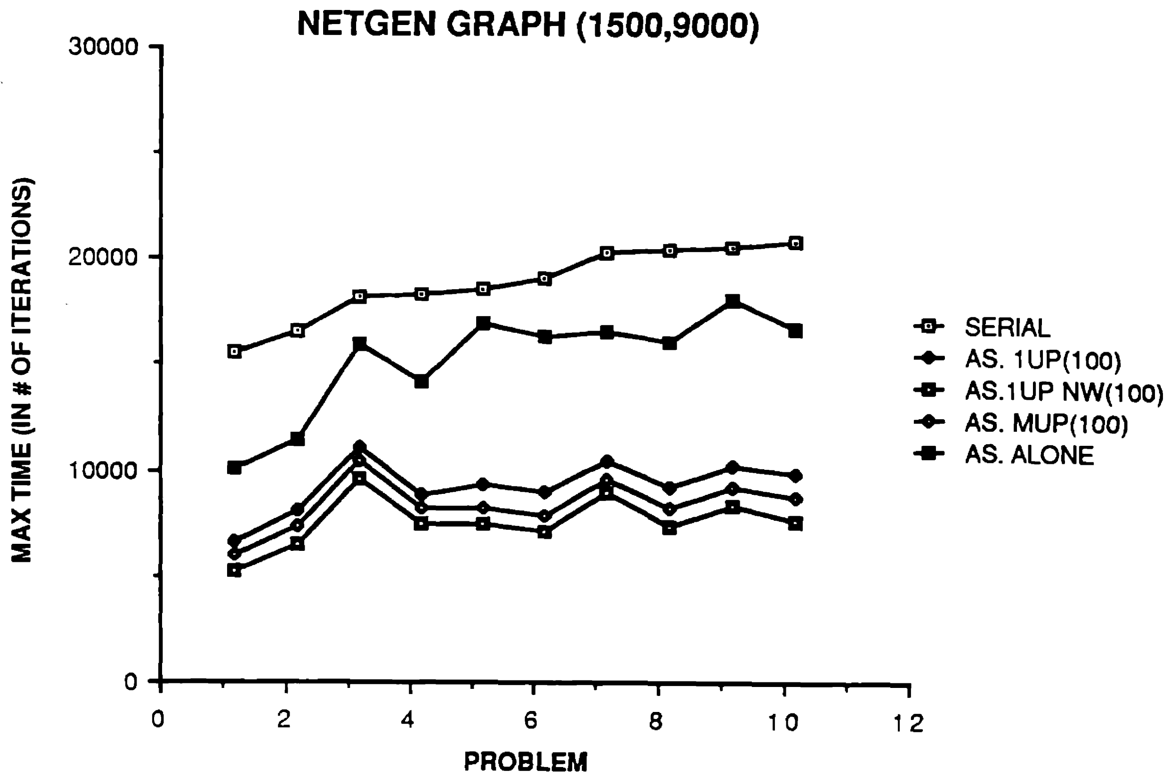


Figure 8.13: This figure and the previous one show the effect of the rate of nodes copied/updated per clock tick on the efficiency of the schemes. We see that for large rate (250 for this network), the three schemes have identical performance. When the rate is smaller differences appear and the non-waiting scheme (scheme2) has better performance.

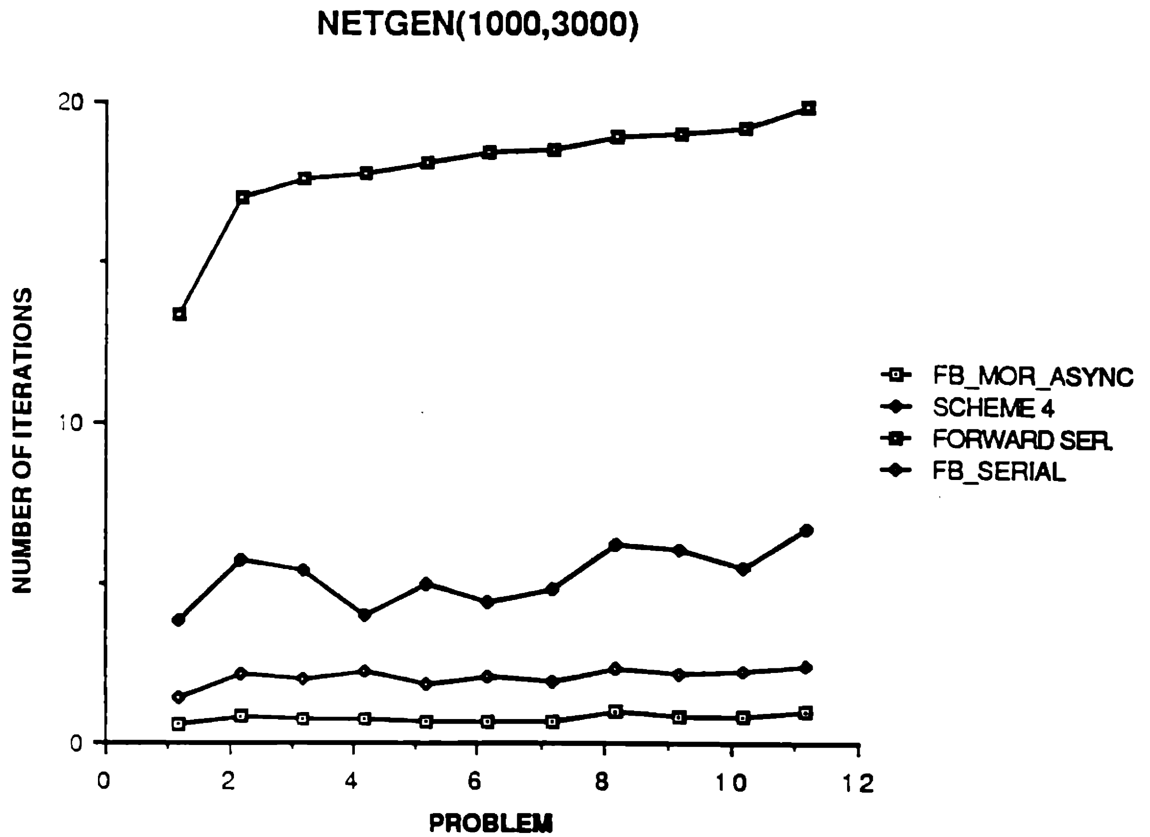


Figure 8.14: This figure intends to show the performance of scheme 4 for the forward auction, which is comparable to that of scheme 2 for five origins. It also shows the excellent performance of the parallel two-sided auction with common prices for 5 origins and 1 destination where the forward algorithm is scheme 4. What this figure (and the next) indicates is that such performance cannot be matched by any other parallel shortest path algorithm. The problem is NETGEN with 1000 nodes and 3000 arcs.

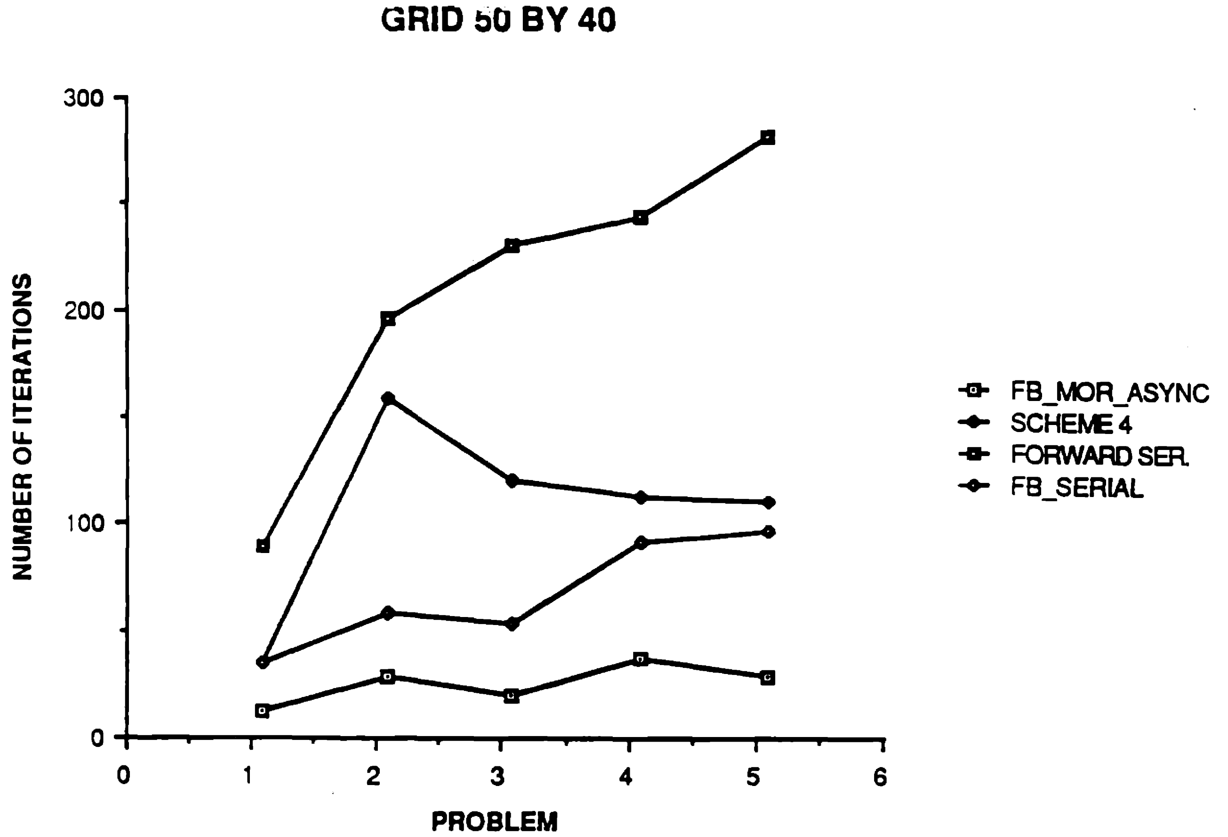


Figure 8.15: This figure intends to show the performance of scheme 4 for the forward auction, which is comparable to that of scheme 2 for five origins. It also shows the excellent performance of the parallel two-sided auction with common prices for 5 origins and 1 destination where the forward algorithm is scheme 4. What this figure (and the previous) indicates is that such performance cannot be matched by any other parallel shortest path algorithm. The problem is GRID 50 by 40.

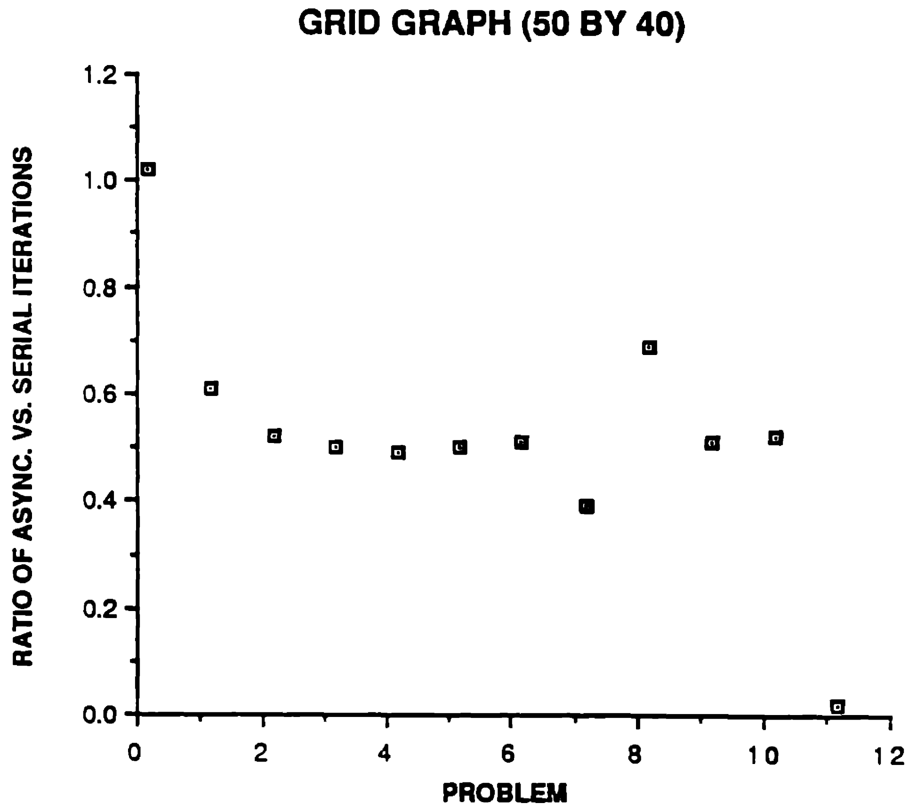


Figure 8.16: This figure (and the two that follow) makes a comparison of the serial two-sided auction with common prices to the parallel two-sided auction with common prices where each side is running on a different processor. A speedup of about 1.8 to 2 is indicated. The algorithms were run on a GRID graph 50 by 40.

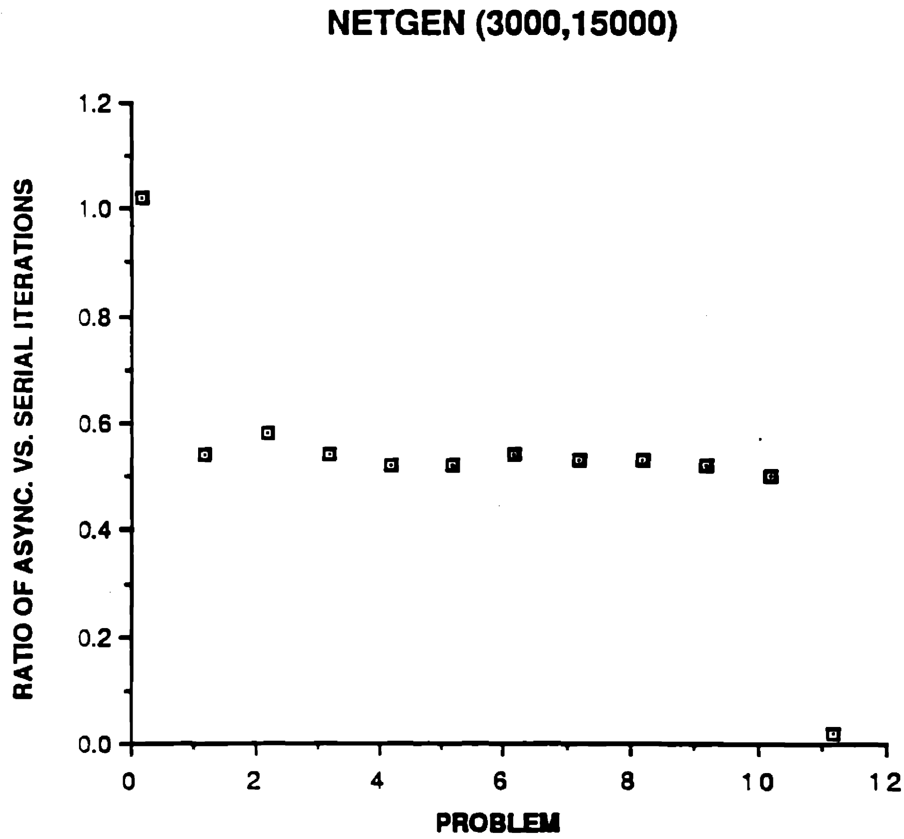


Figure 8.17: This figure makes a comparison of the serial two-sided auction with common prices to the parallel two-sided auction with common prices where each side is running on a different processor. A speedup of about 1.8 to 2 is indicated. The algorithms were run on a NETGEN graph of 3000 nodes and 15000 arcs.

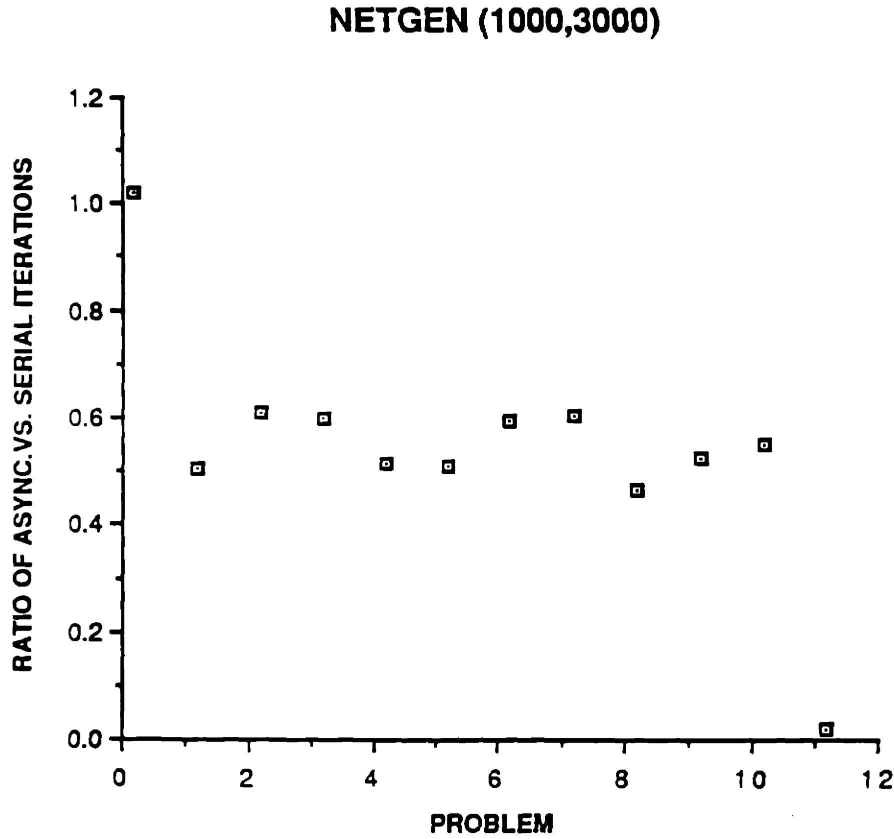


Figure 8.18: This figure makes a comparison of the serial two-sided auction with common prices to the parallel two-sided auction with common prices where each side is running on a different processor. A speedup of about 1.8 to 2 is indicated. The algorithms were run on a NETGEN graph of 1000 nodes and 3000 arcs.

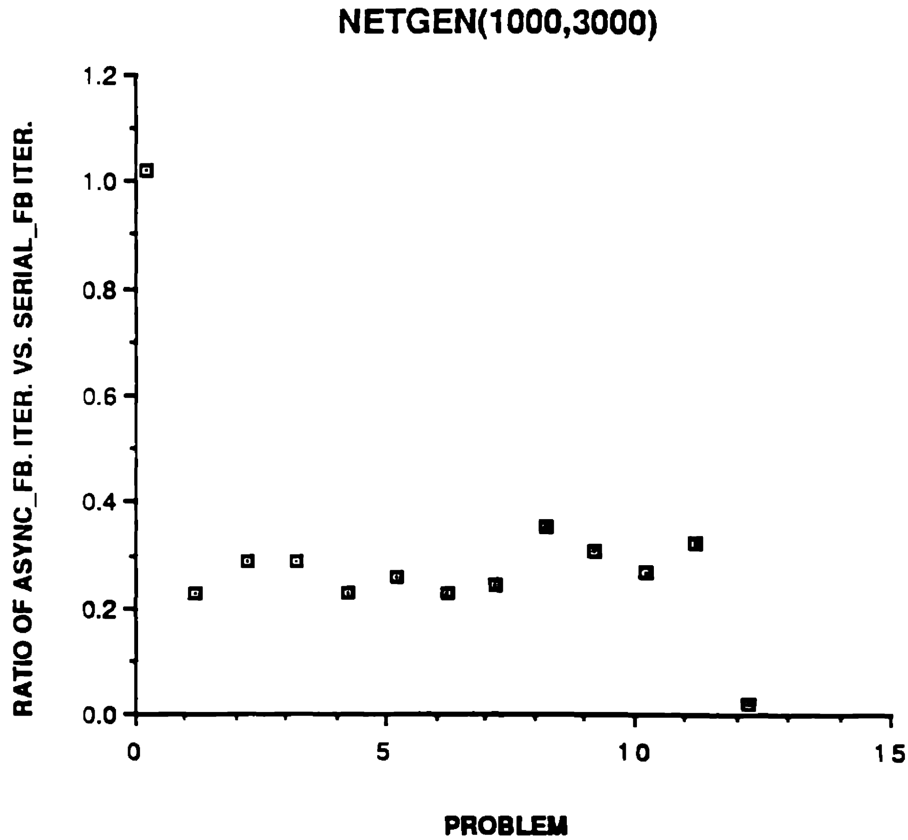


Figure 8.19: This figure makes a comparison of the serial two-sided auction with common prices to the parallel two-sided auction with common prices for 5 origins and 1 destination. The origins are running the parallel scheme 4 and the destination is running the backward algorithm. A speedup of about 4 is indicated. The algorithms were run on a NETGEN graph of 1000 nodes and 3000 arcs.

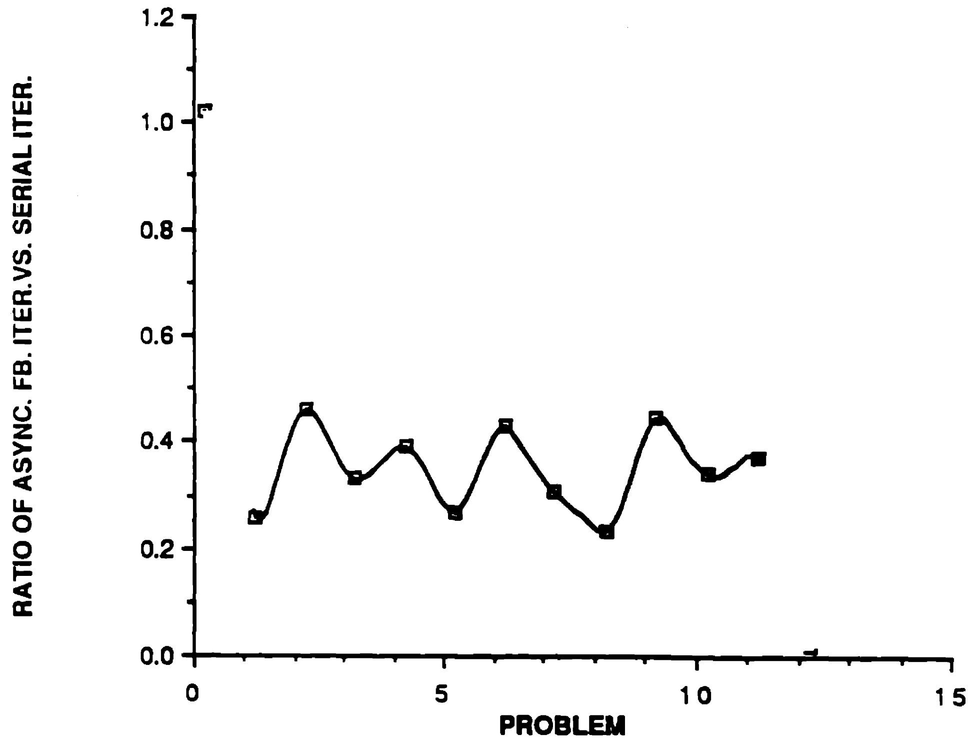


Figure 8.20: This figure makes a comparison of the serial two-sided auction with common prices to the parallel two-sided auction with common prices for 5 origins and 1 destiantion. The origins are running the parallel scheme 4 and the destination is running the backward algorithm. An average speedup of about 3 is indicated. The algorithms were run on a GRID graph 50 by 40.

REFERENCES

[AMO89] Ahuja, R. K., Magnanti, T. L., and Orlin, J. B., "Network Flows", Sloan W. P. No. 2059-88, M.I.T., Cambridge, MA, March 1989

[Bell58] Bellman, R., "On a Routing Problem", Quarterly of Applied Mathematics, Vol. 16, 1958, pp. 88-90.

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

[Ber90] Bertsekas, D. P., "An Auction Algorithm for Shortest Paths", Lab. for Information and Decision Systems-P-2000, M.I.T., October 1990.

[Chv83] Chvatal, V., Linear Programming, W. H. Freeman and Co., N.Y., 1983.

[Dia69] Dial, R. B., "Algorithm 360: Shortest Path Forest with Topological Ordering", Commun. A.C.M., Vol. 12, 1969, pp. 632.

[Dijk59] Dijkstra, e. w., "A Note on Two Problems in Connection With Graphs", Numerische Mathematik, Vol. 1, 1959, pp. 269-271.

[DGK79] Dial, R., Glover, F., Karney, D., and Klingman, D., "A Computational Analysis of Alternative Algorithms and Labeling Techniques for Finding Shortest Path Trees", Networks, Vol. 9, 1979, pp. 215-248.

[GaP86] Gallo, G., and Pallotino, S., "Shortest Path Methods: A Unified Approach", Math. Programming Study, Vol. 26, 1986, p. 38.

[GaP88] Gallo, G., and Pallotino, S., "Shortest Path Algorithms", Annals of Operations Research, Vol. 7, 1988.

[HKS89] Helgason, R. V., Kennington, J. L., Stewart, D. B., "Computational Comparison of Sequential and Parallel Algorithms for the One-To-One Shortest Path Problem", Technical Report 89-CSE-32, 1989.

[Nich66] Nicholson, T., "Finding the Shortest Route Between Two Points in a Network", The Computer Journal, Vol. 9, 1966, pp. 275-280.

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

[Pohl69], Pohl, I., "Bi-directional Search", Machine Intelligence, Vol. 6, B. Meltzer and D. Michie, eds., Edinburgh University Press, Edinburgh 1971, pp. 127-140.

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