PARCO 683

Short communication

Distributed algorithms for the quickest path problem

Yung-Chen Hung and Gen-Huey Chen

Department of Computer Science and Information Engineering, National Taiwan University, Taipei, Taiwan

Received 26 March 1991

Revised 30 January 1992

Abstract

Hung, Y.-C. and G.-H. Chen, Distributed algorithms for the quickest path problem, *Parallel Computing* 18 (1992) 823–834.

Let $N=(V, A, C, L)$ be a network with node set V , arc set A , positive arc capacity function C , and nonnegative arc lead time function L . The quickest path problem is to find paths in N to transmit a given amount of data such that the transmission time is minimized. In this paper, distributed algorithms are developed for the quickest path problem in an asynchronous communication network. For the one-source quickest path problem, we present three algorithms that require $O(m^2)$ messages and $O(m^2)$ time, $O(rmn)$ messages and $O(m)$ time, and $O(rm^{1+\epsilon} \log w)$ messages and $O(rn^{1+\epsilon} \log w)$ time for any ϵ , $0 < \epsilon < 1$, respectively, where $m = |A|$, $n = |V|$, r is the number of distinct capacity values of N , and w is the maximal arc weight of N . For the all-pairs quickest path problem, we present an algorithm that requires $O(mn)$ messages and $O(m)$ time.

Keywords. Asynchronous networks; distributed algorithms; quickest path problem.

1. Introduction

An *asynchronous network* is a point-to-point (store-and-forward) communication network composed of processors and bidirectional non-interfering communication links. Each processor can be uniquely identified. Local memory is provided for each processor, but no common memory is shared between processors. Hence, processors can communicate with each other only by exchanging messages over the communication links. Each processor processes messages received from its neighbours, performs local computation, and sends messages to its neighbours, all in negligible time. Messages can be transmitted independently in both directions on a communication link and arrive after an unpredictable but finite delay, without

Correspondence to: Professor Gen-Huey Chen, Department of Computer Science and Information Engineering, National Taiwan University, Taipei, Taiwan.

error and in *FIFO (First-In-First-Out)* order. When a message is received by a processor, it is inserted into a queue that is maintained by the processor to store unprocessed messages. We assume that each message is of fixed length.

When a distributed algorithm is executed in an asynchronous network, some processors may be nominated to perform special roles and to execute special routines for this purpose. Nevertheless, it is assumed that at the start of the algorithm, each processor holds a copy of the entire code.

The performance of distributed algorithms can be measured according to two standards: message complexity and time complexity. (In some distributed algorithms, e.g. [3,7], only the message complexity is considered in performance analysis, since in an asynchronous distributed environment the number of messages transmitted is significantly more important than the number of computation steps performed.) The *message complexity* of a distributed algorithm is the total number of messages transmitted during its execution. The *time complexity* is the number of 'time units' taken by the algorithm, assuming that the propagation delay of any communication link is at most one unit of time. Since processors in an asynchronous network are far apart, the processing time for each message at a processor is small, in comparison with the message transmission time, and thus is negligible. In this paper, like most distributed algorithms, we only consider the message transmission time.

Let $N = (V, A, C, L)$ be a network with node set V , arc set A , positive arc capacity function C , and nonnegative arc lead time function L . The *quickest path problem*, which was originally proposed by Chen and Chin [5], is to find the quickest paths to transmit data in N . The quickest path problem is a variant of the shortest path problem. The *one-source quickest path problem* (1_QSP for short) is to find the quickest paths between one node and all other nodes. The *all-pairs quickest path problem* (A_QSP for short) is to find the quickest paths between every pair of nodes. Some centralized (single processor) algorithms have been proposed for the quickest path problem. For the 1_QSP , Chen and Chin [5] have presented an $O(m^2 + mn \log n)$ time algorithm, where n and m are the numbers of nodes and arcs, respectively, in N . For the A_QSP , Hung and Chen [8] have presented an $O(mn^2)$ time algorithm.

In this paper, distributed algorithms are presented for the quickest path problem in asynchronous networks. For the 1_QSP , we present three algorithms that require $O(rn^2)$ messages and $O(m^2)$ time, $O(rmn)$ messages and $O(m)$ time, and $O(rm^{1+\epsilon} \log w)$ messages and $O(m^{1+\epsilon} \log w)$ time for any ϵ , $0 < \epsilon < 1$, respectively, where r is the number of distinct capacity values of N and w is the maximal arc weight. For the A_QSP , we present an algorithm that requires $O(mn)$ messages and $O(m)$ time.

The rest of this paper is organized as follows. In the next section, we introduce some notations and definitions that are used throughout this paper. The distributed algorithms for the 1_QSP and the A_QSP are presented in Section 3 and Section 4, respectively. In Section 5, we conclude this paper with some final remarks.

2. Notations and definitions

Let $N = (V, A, C, L)$ be a network, where $|V| = n$, $|A| = m$, and $G = (V, A)$ is an undirected graph without self loops and without multiple arcs. Also, let $C(u, v) > 0$ and $L(u, v) \geq 0$ denote the capacity and the lead time, respectively, of an arc $(u, v) \in A$. More specifically, $C(u, v)$ denotes the maximal amount of data that can be transmitted through arc (u, v) per unit time, and $L(u, v)$ denotes the lead time required to send data through arc (u, v) . Hence, if σ units of data are required to be transmitted through arc (u, v) , the required transmission time is computed as $L(u, v) + \sigma/C(u, v)$.

For example, suppose $C(u, v) = 2$ and $L(u, v) = 3$. If 8 units of data are required to be sent from node u to node v , no data reach node v during the first 3 time units. Then, at each of subsequent time units, 2 units of data reach node v until the transmission ends. Hence, the total transmission time is $3 + 8/2 = 7$ time units.

The following notations are used throughout this paper.

$L(P)$ is the lead time of a path $P = (u_1, u_2, \dots, u_k)$, which is defined as

$$L(P) = \sum_{i=1}^{k-1} L(u_i, u_{i+1}).$$

$C(P)$ is the capacity of a path $P = (u_1, u_2, \dots, u_k)$, which is defined as

$$C(P) = \min_{1 \leq i \leq k-1} C(u_i, u_{i+1}).$$

$T(P, \sigma)$ is the total transmission time to send σ units of data through a path P . Clearly,

$$T(P, \sigma) = L(P) + \sigma/C(P).$$

$QP(s, t, \sigma)$ is the quickest path to send σ units of data between node s and node t in N , which is defined as the path P satisfying

$$T(P, \sigma) = \min\{T(P_i, \sigma) \mid \forall P_i = (u_{i1}, u_{i2}, \dots, u_{ik}) \text{ in } N, \text{ where } s = u_{i1} \text{ and } t = u_{ik}\}.$$

$SP(s, t)$ is the shortest lead time path between node s and node t in N , which is defined as the path P satisfying

$$L(P) = \min\{L(P_i) \mid \forall P_i = (u_{i1}, u_{i2}, \dots, u_{ik}) \text{ in } N, \text{ where } s = u_{i1} \text{ and } t = u_{ik}\}.$$

$N(c_i)$ is a subnetwork of N containing only those arcs with capacities larger than or equal to c_i (without loss of generality, we assume that the distinct capacity values of N are $c_1 > c_2 > \dots > c_r$). That is, $N(c_i) = (V, A_i, C, L)$, where $(u, v) \in A_i$ if and only if $(u, v) \in A$ and $C(u, v) \geq c_i$.

$SP_i(s, t)$ is the shortest lead time path between node s and node t in $N(c_i)$.

As an illustrative example, let us consider the network N of Fig. 1(a), where $\sigma = 12$ and each arc is associated with a pair of values: the lead time and the capacity. The path $P_1 = (D, E, C)$ has $L(P_1) = L(D, E) + L(E, C) = 5$, $C(P_1) = \min\{C(D, E), C(E, C)\} = 2$, and $T(P_1, 12) = L(P_1) + 12/C(P_1) = 5 + 6 = 11$. The other paths between node D and node C are $P_2 = (D, C)$, $P_3 = (D, B, C)$, $P_4 = (D, B, E, C)$, $P_5 = (D, A, B, C)$, $P_6 = (D, A, B, E, C)$, and $P_7 = (D, E, B, C)$. The quickest path between node D and node C is $QP(D, C, 12) = P_1$, since $T(P_1, 12) = \min\{T(P_1, 12), T(P_2, 12), \dots, T(P_7, 12)\} = \min\{11, 16, 23, 26, 28, 31, 36\} = 11$. The shortest lead time path between node D and node C is $SP(D, C) = P_2$, since $L(P_2) = \min\{L(P_1), L(P_2), \dots, L(P_7)\} = \min\{5, 4, 11, 22, 16, 27, 24\} = 4$. The distinct capacity values of N are $c_1 = 6$, $c_2 = 4$, $c_3 = 3$, $c_4 = 2$, and $c_5 = 1$. Figure 1(b) shows the subnetwork $N(c_3)$ of N .

The most important concept of the quickest path problem is that the selection of the quickest paths depends not only on the characteristics of the network but also on the amount of data to be transmitted. Taking these two factors into consideration will make the problem more reasonable. For example, when a certain amount of data are required to be transmitted between two nodes in a communication network, the baud rates of the transmission media between these two nodes can be treated as the capacities, and the lengths of different paths

Fig. 1. Example of (a) a network N , (b) a subnetwork $N(3)$.

between them can be treated as the lead times. For this example, if the amount of data is huge, the paths with larger capacities are preferred. On the other hand, if the amount of data is quite small, the paths with smaller lead times are preferred. Thus, including the amount of data into the selection of the quickest paths is practically significant.

In the next two sections, the network N is embedded in an asynchronous network in the sense that there is a 1-1 correspondence between the nodes of N and the processors of the asynchronous network, and between the arcs of N and the communication links of the asynchronous network. At the beginning of our distributed algorithms, each processor knows only the identities of its neighbours, and the capacities and lead times of its incident arcs. When the algorithms terminate, a routing table for traversing the quickest paths is established within each processor.

3. Distributed algorithms for 1-QSP

In this section, we first show that the quickest path problem can be solved by applying shortest path algorithms.

Theorem 1. $L(QP(s, t, \sigma)) + \sigma/C(QP(s, t, \sigma)) = \min_{1 \leq i \leq r} \{L(SP_i(s, t)) + \sigma/C(SP_i(s, t))\}$, where r is the number of distinct capacity values of N .

Proof. By definition, $L(SP_i(s, t)) + \sigma/C(SP_i(s, t)) \geq L(QP(s, t, \sigma)) + \sigma/C(QP(s, t, \sigma))$, for $i \in \{1, 2, \dots, r\}$. This theorem is proved by the fact that $QP(s, t, \sigma) = SP_j(s, t)$, where $j \in \{1, 2, \dots, r\}$ and $C(QP(s, t, \sigma)) = c_j$. $\square$

Based on Theorem 1, the 1_QSP can be solved by first finding all $SP_i(s, t)$ and then determining the minimum of $L(SP_i(s, t)) + \sigma/C(SP_i(s, t))$, $i = 1, 2, \dots, r$, which can be done by executing any existing one-source shortest path distributed algorithm. (Surely, some modifications are necessary. However, the extra messages and time are negligible.) The complexity of the 1_QSP distributed algorithm depends on the complexity of the adopted one-source shortest path distributed algorithm. Therefore, we have the following theorem.

Theorem 2. *The 1_QSP can be solved with $O(rM)$ messages and $O(rT)$ time in an asynchronous network N , where M and T stand for the message complexity and the time complexity, respectively, of the adopted one-source shortest path distributed algorithm.*

Since the one-source shortest path distributed algorithms proposed by Awerbuch [2], Frederickson [6], and Lakshmanan et al. [9] can find each $SP_i(s, t)$ and determine the values $L(SP_i(s, t))$ and $C(SP_i(s, t))$ using $O(m^{1+\epsilon} \log w)$, $O(n^2)$, $O(mn)$ messages in $O(n^{1-\epsilon} \log w)$, $O(n^2)$, $O(n)$ time, respectively, we have the following corollary.

Corollary 1. *The 1_QSP can be solved in an asynchronous network N using $O(rn^2)$ messages in $O(rn^2)$ time, or using $O(rmn)$ messages in $O(rn)$ time, or using $O(rm^{1+\epsilon} \log w)$ messages in $O(rm^{1+\epsilon} \log w)$ time for any ϵ , $0 < \epsilon < 1$, where r , n , m , and w are the number of distinct capacity values, the number of nodes, the number of arcs, and the maximal arc weight in N , respectively.*

If more efficient one-source shortest path distributed algorithm is adopted, the result of the paper is improved.

According to Theorem 1, we can also solve the A_QSP by repeating the one-source shortest path distributed algorithm nr times. However, this will result in a very high message complexity and time complexity. In the following section, we present a distributed algorithm for the A_QSP that requires only $O(mn)$ messages and $O(m)$ time.

4. A distributed algorithm for A_QSP

Since the amount of data to be transmitted is an important factor for the selection of the quickest paths, it is desirable to find the quickest paths for all possible values of σ . Let $(0, \infty)$ be the range of σ . If σ is very small, then the shortest lead time path is the quickest path. When σ increases, the shortest lead time path may no longer be the quickest path, and another path may become the new quickest path. Thus, we can divide the range of σ into intervals such that the quickest path remains unchanged during each interval of σ .

In our algorithm, an $n \times n \times r$ matrix M_v is provided in each node v to keep some information about the shortest lead time paths and intervals of σ . More specifically, the entry $M_v(s, t, i)$, $1 \leq s \leq n$, $1 \leq t \leq n$, $1 \leq i \leq r$, records four values: $M_v(s, t, i).LT$, $M_v(s, t, i).NEXT$, $M_v(s, t, i).LEFT$ and $M_v(s, t, i).RIGHT$. The value $M_v(s, t, i).LT$ represents the lead time of $SP_i(s, t)$. The value $M_v(s, t, i).NEXT$ represents the next node of v along $SP_i(s, t)$. If $SP_i(s, t)$ does not go through node v , then $M_v(s, t, i).NEXT$ is empty. When the value of σ falls into the interval $(M_v(s, t, i).LEFT, M_v(s, t, i).RIGHT)$, $SP_i(s, t)$ is the quickest path from node s to node t .

The algorithm consists of two stages. The first stage, referred to as *heap construction* stage, constructs a spanning tree of the network N , and then constructs a heap from the spanning tree according to the capacities of arcs. A *heap* is a tree, in which the value associated with each node is greater than or equal to the values associated with its children. An arc in the

heap is called a *tree arc* if it belongs to the heap, and a *nontree arc* otherwise. The root node of the spanning tree is assigned as the algorithm leader. In the second stage, which is referred to as *broadcasting* stage, the root node broadcasts all the arcs of N (including their end nodes, capacities, and lead times) over the tree, one by one and in a nonincreasing order of their capacities. Whenever a node v receives one such message, it must update the values $M_v(s, t, i).LT$, $M_v(s, t, i).NEXT$, $M_v(s, t, i).LEFT$ and $M_v(s, t, i).RIGHT$. The updating of the matrix M_v is discussed later in this section.

In the algorithm, each arc may be in one of the two statuses: *enable* and *disable*. Initially, all arcs are in status *enable*. During the execution of the algorithm, each node v must determine the largest capacity arc among those enabled arcs whose one or two end nodes are in the subtree of v , and then, as a reply to the request of the parent of v , the selected arc is sent out and its status is changed to *disable*.

Next, we give a summary of the main messages used in the algorithm. To simplify the description of the algorithm, various acknowledgement messages will not be described.

init

The message is broadcast over the spanning tree. The broadcasting starts from the root node and propagates toward leaf nodes. When a nonleaf node receives an *init* message from its parent, it forwards the message to its children.

echo($x, y, L(x, y), C(x, y)$)

When a nonroot node v receives an *init* message, it must return a message *echo*($x, y, L(x, y), C(x, y)$) to its parent, where (x, y) is the largest capacity arc among those enabled arcs whose one or two end nodes are in the subtree of v .

arc($u, v, L(u, v), C(u, v)$)

The root node broadcasts the message *arc*($u, v, L(u, v), C(u, v)$) for each arc (u, v) in N over the spanning tree. These *arc* messages are broadcast in a nonincreasing order of $C(u, v)$ s. Whenever a node w receives an *arc* message, it must update the matrix M_w .

The following is a formal description of the algorithm.

Algorithm A_QSP:

Stage 1: / Heap construction stage */*

- (1) Construct a spanning tree of N :
- (2) The status of each arc is set to *enable*.
- (3) The root node executes the following.
 - (3.1) Send an *init* message to each of its children.
 - (3.2) Wait for *echo* messages that are returned from its children.
- (4) Upon receiving an *init* message, a nonleaf node executes the following.
 - (4.1) Send an *init* message to each of its children.
 - (4.2) Wait for *echo* messages that are returned from its children.
 - (4.3) Select the largest capacity arc (assuming (x, y)) among the arcs contained in the returned *echo* messages, the arcs connecting the node with its children, and the enabled nontree arcs that are incident to the node.
 - (4.4) Return the message *echo*($x, y, L(x, y), C(x, y)$) to its parent.
 - (4.5) If (x, y) is a nontree arc or connects the node with one of its children, then the status of (x, y) is changed to *disable*.
 - (4.6) Otherwise, send an *init* message to its child from which the node received the message *echo*($x, y, L(x, y), C(x, y)$) and wait for a reply.

- (5) Upon receiving an *init* message, a leaf node executes the following.
 - (5.1) Select the largest capacity arc (assuming (x, y)) among those enabled nontree arcs that are incident to the node.
 - (5.2) If no such arc exists, then return the message $echo(0, 0, \infty, 0)$ to its parent.
 - (5.3) Otherwise, return the message $echo(x, y, L(x, y), C(x, y))$ to its parent, and change the status of (x, y) to *disable*.

Stage 2: /* Broadcasting stage */

- (1) The root node (assuming q) executes the following.

repeat

 - (1.1) Select the largest capacity arc (assuming (u, v)) among the arcs contained in the returned *echo* messages and the enabled arcs that are incident to the root node.
 - (1.2) If $(u, v) = (0, 0)$, then terminate the algorithm (by broadcasting a termination message), and exit the loop.
 - (1.3) Otherwise, send the message $arc(u, v, L(u, v), C(u, v))$ to each of its children.
 - (1.4) If (u, v) is incident to the root node, then change the status of (u, v) to *disable*.
 - (1.5) Otherwise, send an *init* message to the child from which the root node received the message $echo(u, v, L(u, v), C(u, v))$, and wait for a reply.
 - (1.6) Update $M_q(s, t, i).LT$, $M_q(s, t, i).NEXT$, $M_q(s, t, i).LEFT$ and $M_q(s, t, i).RIGHT$, where $1 \leq s \leq n$, $1 \leq t \leq n$, and $C(u, v) = c_i$.

forever
- (2) Upon receiving a message $arc(u, v, L(u, v), C(u, v))$, a node (assume w) executes the following.
 - (2.1) Send the *arc* message to each of its children.
 - (2.2) Update $M_w(s, t, i).LT$, $M_w(s, t, i).NEXT$, $M_w(s, t, i).LEFT$ and $M_w(s, t, i).RIGHT$, where $1 \leq s \leq n$, $1 \leq t \leq n$, and $C(u, v) = c_i$.
- (3) Upon receiving an *init* message, a node executes the following.
 - (3.1) Maintain the heap and return an *echo* message to its parent (as we have described in steps 4 and 5 of Stage 1).

In order to construct a heap from the spanning tree, the root node broadcasts an *init* message over the tree. Initially, let the status of each arc be *enable*. When a nonleaf node receives an *init* message from its parent, it forwards the message to its children. Thus, the *init* message is propagated toward leaf nodes. When a leaf node x receives an *init* message, it returns a message $echo(x, y, L(x, y), C(x, y))$ to its parent, where (x, y) is the largest capacity arc among those enabled nontree arcs that are incident to x . Then the status of (x, y) is set to *disable* (node x will inform node y of this change). After a nonleaf node has received all the *echo* messages that are returned from its children, it selects the largest capacity arc (assuming (x, y)) among the arcs contained in the returned *echo* messages, the arcs connecting the node with its children, and the enabled nontree arcs that are incident to the nonleaf node. Then, the nonleaf node returns the message $echo(x, y, L(x, y), C(x, y))$ to its parent. If the arc (x, y) is a nontree arc or connects the nonleaf node with one of its children, then its status is changed to *disable*. The heap construction stage terminates when the root node has received all the *echo* messages returned from its children. Note that in the above execution, each node returns to its parent the largest capacity arc that it holds. After this, the heap property may be violated. So, in order to maintain the heap, the node has to require its one child to send the *echo* message again (see step 4.6 in Stage 1).

The message and time complexities of the heap construction stage are analyzed as follows. The distributed algorithm proposed by Awerbach [1], which requires $O(m + n \log n)$ messages and $O(n)$ message transmission time, can be used to find a spanning tree of N . There

are at most two *init* and *echo* messages transmitted via each tree arc. Thus, the total number of messages transmitted are $O(m + n \log n)$, where n and m are the numbers of nodes and arcs in N , respectively. The time required to transmit the *init* and *echo* messages is proportional to the depth of the tree. Thus, the total message transmission time is $O(n)$.

In the broadcasting stage, the root node repeatedly selects the largest capacity arc (assuming (u, v)) among the arcs contained in the returned *echo* messages and the enabled arcs that are incident to the root node, and then broadcasts the message $arc(u, v, L(u, v), C(u, v))$ over the tree. These *arc* messages are broadcast in a nonincreasing order of $C(u, v)$ s. In the meantime, whenever a node w receives an *arc* message, the matrix M_w must be updated. Also, the heap property must be maintained, as discussed in the heap construction stage. The total number of messages transmitted is $O(mn)$. Since the *arc* messages may be sent from the root node in m consecutive time steps (the heap maintenance is performed simultaneously), the total message transmission time is $O(m)$.

Now, the remaining problem is how to update the matrix M_w for each node w whenever it receives an *arc* message. At first, the network N is regarded as empty (containing no arcs). Since new paths may be generated from a node s to another node t when an arc (u, v) is added to N , the quickest path from s to t may be altered. Thus, whenever a node w receives a message $arc(u, v, L(u, v), C(u, v))$, it must update the values $M_w(s, t, i).LT$, $M_w(s, t, i).NEXT$, $M_w(s, t, j).LEFT$ and $M_w(s, t, i).RIGHT$ for every pair of nodes s and t , where $C(u, v) = c_i$ is assumed.

Although many new paths from node s to node t may be generated after adding an arc (u, v) to N , only two paths are needed to be considered. One contains the shortest lead time path from s to u (without passing the arc (u, v)), the arc (u, v) , and the shortest lead time path from v to t (without passing (u, v)), and the other contains the shortest lead time path from s to v (without passing (u, v)), the arc (u, v) , and the shortest lead time path from u to t (without passing (u, v)). Initially, let $M_w(s, t, 0).LT = \infty$ and $M_w(s, t, 0).NEXT$ be empty. The updating of $M_w(s, t, i).LT$ and $M_w(s, t, i).NEXT$ is shown below as the procedure *ComputeSP*($u_k, v_k, L(u_k, v_k), C(u_k, v_k)$). Without loss of generality, we assume that the arcs of N are broadcast in the sequence of $(u_1, v_1), (u_2, v_2), \dots, (u_m, v_m)$, where $C(u_1, v_1) \geq C(u_2, v_2) \geq \dots \geq C(u_m, v_m)$.

```

ComputeSP( $u_k, v_k, L(u_k, v_k), C(u_k, v_k)$ );
/* Assume  $C(u_{k-1}, v_{k-1}) = c_j$  and  $C(u_k, v_k) = c_i$ . */
1   for each  $s \in V$ 
2     for each  $t \in V$ 
3       begin
4          $T_1 \leftarrow M_w(s, u_k, j).LT + L(u_k, v_k) + M_w(v_k, t, j).LT$ ;
5          $T_2 \leftarrow M_w(s, v_k, j).LT + L(u_k, v_k) + M_w(u_k, t, j).LT$ ;
6          $M_w(s, t, i).LT \leftarrow \min\{T_1, T_2, M_w(s, t, j).LT\}$ ;
7         if  $T_1 = M_w(s, t, i).LT$ 
8           then if  $w = u_k$  then  $M_w(s, t, i).NEXT \leftarrow v_k$ 
9             else if  $M_w(s, u_k, j).NEXT$  is not empty
10               then  $M_w(s, t, i).NEXT \leftarrow M_w(s, u_k, j).NEXT$ 
11             else if  $M_w(v_k, t, j).NEXT$  is not empty
12               then  $M_w(s, t, i).NEXT \leftarrow M_w(v_k, t, j).NEXT$ 
13             else let  $M_w(s, t, i).NEXT$  be empty;
14         if  $T_2 = M_w(s, t, i).LT$  and  $T_1 \neq T_2$ 
15           then if  $w = v_k$  then  $M_w(s, t, i).NEXT \leftarrow u_k$ 
16             else if  $M_w(s, v_k, j).NEXT$  is not empty
              then  $M_w(s, t, i).NEXT \leftarrow M_w(s, v_k, j).NEXT$ 

```

Fig. 2. The minimal transmission time is the lower portion of $SP_i(s, t)$, $i = 1, \dots, 6$.

```

17                                     else if  $M_w(u_k, t, j).NEXT$  is not empty
18                                     then  $M_w(s, t, i).NEXT \leftarrow M_w(u_k, t, j).NEXT$ 
19                                     else let  $M_w(s, t, i).NEXT$  be empty;

end.
```

The intervals of σ , which is the amount of data to be transmitted, are determined as follows. Since each $SP_i(s, t)$, $1 \leq i \leq r$, has the shortest lead time among those paths from s to t whose capacities are larger than or equal to c_i , the transmission time of each path from s to t with capacity equal to c_i is greater than or equal to the transmission time of $SP_i(s, t)$ for all values of σ . Thus, only the paths $SP_1(s, t), SP_2(s, t), \dots, SP_r(s, t)$ are necessary to be considered in constructing intervals of σ for a pair of nodes s and t .

Clearly, the transmission time of each $SP_i(s, t)$ is a function of σ . That is,

$$y = M_w(s, t, i).LT + x/c_i,$$

where $y = T(SP_i(s, t), \sigma)$ and $x = \sigma$. It is easy to see that the minimal transmission time for each value of σ can be determined as $\min\{M_w(s, t, i).LT + x/c_i \mid i = 1, \dots, r\}$. Equivalently, the lower portion of these r lines $y = M_w(s, t, i).LT + x/c_i$, $i = 1, \dots, r$, denotes the minimal transmission time for all values of σ (see Fig. 2), and the intersection points (s_1 and s_2 in Fig. 2) along the lower portion separate the intervals.

The intersection points can be found by adding the lines $y = M_w(s, t, i).LT + x/c_i$, $i = 1, \dots, r$, to the plane, one by one and in increasing sequence of i . The line $y = M_w(s, t, 1).LT + x/c_1$ is first added to the plane. Then, let us consider the situation of adding a line $y = M_w(s, t, i).LT + x/c_i$, $1 < i \leq r$, to the plane. The newly added line intersects the current lower portion at one point z (since the slopes of the lines are positive and increasing in their order), which can be determined by scanning the lower portion from left to right. The point z is then stored as a new intersection point, and the intersection points (s_2 in Fig. 3) on the left of z are discarded.

The values $M_w(s, t, i).LEFT$ and $M_w(s, t, i).RIGHT$, $i = 1, \dots, r$, are determined as follows. Initially, set $M_w(s, t, 1).LEFT = 0$ and $M_w(s, t, 1).RIGHT = \infty$. For each newly generated intersection point z of two lines $y = M_w(s, t, i).LT + x/c_i$ and $y = M_w(s, t, j).LT + x/c_j$, where $j < i$, set $M_w(s, t, i).RIGHT = M_w(s, t, j).LEFT = z$, and $M_w(s, t, k).LEFT = M_w(s, t, k).RIGHT = \infty$, for $j < k < i$. In the following, the procedure *ComputeInt*

Fig. 3. After a line $y = M_i(s, t, i).LT + x/c_i$ is added to the plane, a new intersection point z is generated and the intersection point s_2 is discarded.

$(C(u_k, v_k))$ shows the necessary changes of the intervals of σ whenever a node w receives an *arc* message in the broadcasting stage.

```

ComputeInt( $C(u_k, v_k)$ );
/* Assume  $C(u_k, v_k) = c_i$ . */
1   if  $i > 1$  then
2     for each  $s \in V$ 
3       for each  $t \in V$ 
4         begin
5            $j \leftarrow i - 1$ ;
6           While ( $M_w(s, t, i).LT + M_w(s, t, j).LEFT/c_i \geq M_w(s, t, j).LT$ 
7              $+ M_w(s, t, j).LEFT/c_j$ )
8             or  $M_w(s, t, j).LEFT = \infty$ 
9             begin
10               $M_w(s, t, j + 1).LEFT \leftarrow \infty$ ;
11               $M_w(s, t, j + 1).RIGHT \leftarrow \infty$ ;
12               $j \leftarrow j + 1$ 
13            end;
14             $z \leftarrow (M_w(s, t, j).LT - M_w(s, t, i).LT) * c_i * c_j / (c_j - c_i)$ ;
15             $M_w(s, t, j).LEFT \leftarrow z$ ;
16             $M_w(s, t, i).RIGHT \leftarrow z$ ;
17             $M_w(s, t, i).LEFT \leftarrow 0$ ;
18          end.

```

Theorem 3. The A_QSP can be solved in an asynchronous network using $O(mn)$ messages, in $O(m)$ time.

It is easy to see that the matrix M_w can serve as a routing table for node w . Whenever node w receives an amount σ of data that are required to be transmitted from a node s to another node t , it knows how to route the data so as to minimize the total transmission time by looking up the matrix M_w . First, it determines an interval $(M_w(s, t, i).LEFT,$

$M_w(s, t, i).RIGHT)$ such that $M_w(s, t, i).LEFT \leq \sigma \leq M_w(s, t, i).RIGHT$. Then, it determines $M_w(s, t, i).NEXT$ as the next node to route the data, since the arc $(w, M_w(s, t, i).NEXT)$ belongs to the quickest path from s to t .

5. Concluding remarks

The quickest path problem is a variant of the shortest path problem. Unlike the shortest paths, the selection of the quickest paths depends on not only the characteristics of the network but also the amount of data to be transmitted. In this paper, we have proposed distributed algorithms for the quickest path problem. For the 1_QSP, three algorithms are proposed, which require $O(rn^2)$ messages and $O(rn^2)$ time, $O(rmn)$ messages and $O(rn)$ time, and $O(rm^{1+\epsilon} \log w)$ messages and $O(rm^{1+\epsilon} \log w)$ time for any ϵ , $0 < \epsilon < 1$, respectively, where m , n , r , and w are the number of arcs, the number of nodes, the number of distinct capacity values, and the maximal arc weight, respectively, in the network N . For the A_QSP, a distributed algorithm, which requires $O(mn)$ messages and $O(m)$ time, is proposed. The quickest path problem has not been solved before in distributed environments.

It is known that for any graph problem, it is always possible to obtain a trivial distributed solution by broadcasting all topological information to all processors, and then solve the problem sequentially in each processor. Using the trivial approach, the quickest path problem can be solved in four main stages: constructing a spanning tree of N , broadcasting all topological information of N , sorting the arcs of N nonincreasingly according to their capacities, and computing the quickest paths. The most difficult problem in the trivial approach is how to broadcast in a simple and efficient fashion.

Essentially, the proposed A_QSP algorithm is a refinement of the trivial approach. The use of the heap makes the broadcasting simpler and more rhythmical. The main advantages gained due to the use of the heap are as follows. First, the proposed A_QSP algorithm becomes very simple and can be easily implemented. Only three main types of messages are needed; the *init* and *arc* messages are sent downwards, and the *echo* message is sent upwards. Secondly, the arcs are broadcast in a nonincreasing order of their capacities. Thus, the sorting becomes unnecessary. Thirdly, since the arcs are broadcast in a sorted sequence, the message transmission can be overlapped with the message processing (computing the quickest paths). Thus, more parallelism is obtained. Also, note that the proposed approach for computing the quickest paths fully utilizes the advantages of a sorted sequence of arcs.

Finally, since maintaining the heap will cause the arcs with greater capacities moved upwards, the induced cost is made up by the gainings due to data moving. Moreover, the maintenance of the heap is overlapped with the data broadcasting.

Acknowledgement

The authors are pleased to thank the anonymous referees for their valuable suggestions and comments. They have improved the paper a lot and made the paper more readable. Also, thanks are due to Professor Jau-Hsiung Huang, who has spent much time in reading this paper very carefully and providing many helpful comments.

References

- [1] B. Awerbuch, Optimal distributed algorithm for minimum weight spanning tree, counting, leader election and related problems, *Proc. 19th Annual ACM Symp. on Theory of Computing* (May 1987) 230–240.

- [2] B. Awerbuch, Distributed shortest paths algorithms, *Proc. 21st STOC* (1989) 490–500.
- [3] B. Awerbuch and R. Gallager, A new distributed algorithm to find breadth first search trees, *IEEE Trans. Inform. Theory* 33 (3) (May 1987) 315–322.
- [4] L.D. Bodin, B.L. Golden, A.A. Assad and M.O. Ball, Routing and scheduling of vehicles and crews: the state of the art, *Comput. Operat. Res.* 10 (1982) 63–211.
- [5] Y.L. Chen and Y.H. Chin, The quickest path problem, *Comput. Operat. Res.* 17 (1989) 153–161.
- [6] G.N. Frederickson, A distributed shortest path algorithm for a planar network, *Inform. and Comput.* 86 (1990) 140–159.
- [7] M.L. Fredman and R.E. Tarjan, Fibonacci heaps and their uses in improved network optimization algorithms, *J. ACM* 34 (1987) 596–615.
- [8] Y.C. Hung and G.H. Chen, On the quickest path problem, *Lecture Notes in Comput. Sci.* 497 (May 1991) 44–46.
- [9] K.B. Lakshmanan, K. Thulasiraman and M.A. Comeau, An efficient distributed protocol for finding shortest paths in networks with negative weights, *IEEE Trans. Software Engrg.* 15 (5) (May 1989).