A DISTRIBUTED PROTOCOL FOR THE NETWORK PRIMAL-DUAL METHOD AND SIMULATION ON A SHARED-MEMORY MULTIPROCESSOR

Parimala Thulasiraman

A Thesis

in

The Department

of

Electrical and Computer Engineering

Presented in Partial Fulfilment of the Requirements
for the Degree of Master of Applied Science at
Concordia University
Montreal, Quebec, Canada

November 1991

ISBN 0-315-73701-6

ABSTRACT

A Distributed Protocol for the Network Primal-Dual Method

and

Simulation on a Shared-Memory Multiprocessor

Parimala Thulasiraman

The theme of this work is distributed computing for network optimization problems. Network optimization refers to the class of optimization problems defined on graphs. For our study we select the transshipment problem (also known as the minimum cost flow problem) which generalizes several of the network optimization problems, and the primal-dual method for solving this problem. Thus the goal of our work is to design a distributed protocol for the network primal- dual method and to simulate the protocol on a shared memory multiprocessor.

The primal-dual method has the interesting property that its application involves repeated use (in an iterative loop) of the shortest path and maximum flow algorithms. We show that the primal-dual initialization problem can also be formulated as a shortest path problem. Therefore we first present synchronous distributed protocols for the shortest path and the maximum flow problems. For the shortest path protocol we adopt Chandy and Misra's termination detection scheme. For the max-flow protocol we develop a scheme for termination detection. We integrate these two distributed protocols and design a distributed protocol for the primal-dual method. To guarantee correct working of these protocols in an asynchronous environment, we incorporate appropriate synchronizer mechanisms into these protocols.

We carry out simulation of our protocols on the BBN Butterfly parallel machine. Several issues that do not arise in a truly message-passing environment are encountered while implementing the protocols on a shared-memory multiprocessor. We discuss these issues and our approach to resolve them. The thesis concludes with some experiences that we have gained in the course of the work, and some suggestions for furture work.

ACKNOWLEDGEMENTS

I thank my thesis supervisors Professors Marc Comeau and Anindya Das for their friendly guidance during the entire period of my research. The lively discussions I have had with them have enriched my understanding of distributed computing and network optimization.

I thank Professor M.O.Ahmad for the constant encouragement I have received from him during the last six years of my undergraduate and graduate studies at Concordia. His concern for my success and timely guidance at the appropriate moments are deeply appreciated.

I also thank Professor C. Tropper for giving permission to use the BBN Butterfly parallel machine at the McGill Computer Science department. But for this valuable help, the work in this thesis would not have been completed.

This research has been a part of the NSERC-BNR Co-operative Research and Development project being carried out at Concordia under a grant awarded to Professors K. Thulasiraman, M. Comeau and A.K. Elhakeem. The project provided an excellent environment for acquiring valuable academic and practical experience in distributed computing.

I extend my whole-hearted thanks to my father for his encouragement, help and constant drive through the entire period of the thesis, and to my mother for her warmth and care. Finally, I thank my husband, Thulasiram for his support, patience and under standing during this strenuous period.

Parimala Thulasiraman, Nov.1991.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER I
INTRODUCTION

In the recent past there has been considerable interest in the design of distributed and parallel algorithms. An early interest in distributed computing arose in the context of routing algorithms in computer networks. A computer network is a collection of geographically distributed computers interconnected by communication links. Routing in a network involves a rather complex collection of algorithms that work more or less independently and yet cooperate with each other by exchanging information. This interest in distributed routing algorithms has lead researchers to study distributed algorithms for a number of network problems such as the problem of constructing a minimum cost spanning tree of a network. While distributed network management continues to be an active area of research, distributed computing has also gained considerable importance because of applications involving distributed data bases, distributed operating systems, etc. A substantial introduction to distributed computing and distributed systems may be found in [Lyn88], [BeT89] and [Mul89]. Whereas the main emphasis in distributed computing is on solving a problem in a distributive manner using a collection of processors with limited memory and processing capabilities, in parallel computing the emphasis is on achieving speed-up over sequential algorithms. A detailed discussion of several parallel algorithms may be found in [Akl89].

Our interest in this thesis is in design of distributed algorithms for network optimization problems. Though our main concern is the design of efficient distributed algorithms, we find that these algorithms provide considerable insight into the structure of parallel (multiprocessor) algorithms aimed at achieving speed-up over uniprocessor algorithms.

We now present a brief review of current literature on distributed algorithms for network optimization problems. This will be followed by an outline of the scope of this thesis.

Network optimization refers to the class of optimization problems defined on graphs. These problems include the problem of constructing shortest paths, finding a maximum flow, constructing a minimum cost spanning tree, finding matchings in a network, etc. These problems occur in a variety of applications. While they are themselves significant in their fullness, they also occur as subproblems in several other applications. Kruskal [Kru56] and Prim's [Pri57] algorithms for the minimum cost tree problem, Dijkstra's [Dij59] and Bellman-Ford-Moore's algorithms [Bel58], [FoF62], [Moo57] for the shortest path problems, Ford-Fulkerson's algorithm [FoF56] and its several variants (for example, [Din70], [MPM78]) for the maximum flow problem are among the most fundamental algorithms in network optimization theory. They have also served as the basis for designing corresponding distributed algorithms. Some of these distributed network algorithms may be found in [ChM83], [GHS83], [Hum83] and [LTC89]. Issues relating to the synchronizer design problem have been discussed in [LaT87], [LTC89] and [SeS91].

In this thesis, our focus will be on the transshipment problem, which can be formulated as a linear programming problem. This problem generalizes several of the network optimization problems. A detailed discussion of this problem and algorithmic solutions may be found in [Chv83], [Roc84]. There are two basic approaches to this problem – the network simplex method and the primal-dual method. Recently, Goldberg [Gol87] presented a variant of the primal-dual method called the ε-relaxation method. References to other ε-relaxation methods may also be found in [Gol87]. The relaxation methods are primarily designed to obtain good complexity results for the transshipment problem. Goldberg [Gol87] and Goldberg and Tarjan [GoT80] also presented a novel algorithm for the maximum flow problem. This algorithm for the maximum flow problem and the primal-dual approach for the transshipment problem are quite elegant and amenable to distributed implementation. The primal-dual approach has the interesting property that its application involves repeated use of shortest path and maximum flow algorithms. Thus the main focus of our work is on designing and implementing distributed algorithms for the shortest path and maximum flow problems, and integrating them into a distributed algorithm for the primal-dual algorithm for the transshipment problem.

A brief outline of the scope of the thesis is as follows.

We define in Chapter II two models of distributed computation, namely, the asynchronous and synchronous models. We then examine the need for synchronizers in running synchronous algorithms on asynchronous networks. We also present certain difficulties one may encounter in designing the synchronizers as well as remedies for these difficulties. The synchronizers will be used in implementing asynchronous distributed protocols for the shortest path and maximum flow protocols discussed in Chapters III and IV. All the details of incorporating these synchronizer mechanisms will be discussed in Chapter VI where we discuss simulation of our distributed algorithms.

In Chapter III we discuss the Bellman-Ford-Moore shortest path algorithm and a variant of it which is amenable to distributed implementation. We then show how the shortest path algorithm can also be used to initialize the primal-dual method. Finally, we present a synchronous distributed protocol for the shortest path problem which incorporates a mechanism to detect infeasibility.

In Chapter IV we first present certain basic results in network flow theory and then discuss the details of Goldberg and Tarjan's maximum flow algorithm. We then present a synchronous distributed protocol for the maximum flow problem. We also discuss mechanisms for termination detection.

In Chapter V we discuss, in detail, the main steps in the primal-dual approach. We show that this approach involves repeated applications of the shortest path and maximum flow algorithms.

We present in Chapter VI our simulation of the distributed algorithms for the shortest path and the maximum flow problems, and their integration into a distributed algorithm for the primal-dual method. Our simulation is on the BBN Butterfly multiprocessor, which employs the shared memory paradigm. We discuss in this chapter various issues encountered relating to the simulation, in particular, synchronization and termination detection problems. We also give certain performance results.

The thesis has four Appendices. Appendix A gives an overview of the main programming features of the BBN multiprocessor. Appendices B, C and D give detailed descriptions of our distributed programs for the maximum flow, shortest path and primal-dual problems, respectively.

For graph-theoretic terminology used in the thesis we follow [SwT81]. We shall use the terms node and vertex interchangebly. Also, we shall follow the practice in distributing computing literature of using the terms node and distributed protocol in place of vertex and distributed algorithm, when the discussion is in the context of network problems.

CHAPTER II
MODELS OF DISTRIBUTED COMPUTATION AND SYNCHRONIZER DESIGN

This chapter is concerned with certain issues relating to distributed computing. First we introduce two models of computation, namely, the synchronous and asynchronous models. We then present the notion of a synchronizer. After briefly discussing the three synchronizers introduced by Awerbuch [Awe85] and [Awr85], we draw attention to certain problems one may encounter in the implementation of the synchronizer mechanism. We also discuss some of the remedies available in the literature [LaT87] and [SeS91].

2.1 Models of Distributed Computation

A *communication network* can be represented by a communication graph G=(V,E) where V is the set of nodes (vertices) corresponding to processors connected by bidirectional communication links represented by E. Let $|V| = n$ and $|E| = m$. It is assumed that processors have distinct identities and each processor only knows the identities of its neighbours. Also the processors share no common memory and have access only to their local information. Therefore, messages are frequently exchanged between processors, through communication links, for the purpose of computation.

A *distributed algorithm/protocol* consists of a collection of similar node algorithms residing at the processors. These node algorithms specify the actions to be taken in response to the messages that may arrive at a node.

Two network models - *asynchronous and synchronous* - are considered in dealing with distributed algorithms. An *asynchronous network* is a point-to-point (store and forward) communication network. Each processor receives messages from its neighbours, performs local computations and delivers messages to its neighbours. Each message sent by a processor arrives at its destination within a finite but undetermined time. Messages are of fixed length. The messages sent over a link follow a *first-in-first-out* rule. On the other hand, a global clock is associated with a *synchronous network*. Every processor in the network has access to this clock. Each processor sends messages only at integer times, or *pulses*, of the global clock. It is assumed that all messages sent at a previous clock pulse are received before the next pulse is generated. At most one message may be sent over a given link at a certain pulse. The delay of each link is at most one time unit of the global clock.

In both these models it is assumed that messages received at a processor are stamped with the identity of the sender and transferred to a single common queue before being processed one by one. Thus it is not permitted to delay the processing of a message once received.

For both network models considered above, it is assumed that the *message complexity* C of an algorithm is the total number of messages sent during the algorithm. Also in both models the actions performed by a processor are assumed to take negligible time. The *time complexity* T of a synchronous algorithm is the total number of pulses passed from its starting time to its termination. The *time complexity* of an asynchronous algorithm is the worst-case total number of time units from start to completion. When considering the performance evaluation of an asynchronous algorithm the propagation delay (difference between transmission time and arrival time) and inter-message delay (difference between transmission times of two consecutive messages) on each link are assumed to be at most one time unit. The algorithm must work for arbitrary delays also. The complexities of a distributed algorithm depend on the structure of the graph as well as parameters such as number of nodes, number of edges, etc. So while evaluating these algorithms only worst-case complexities are used.

2.2 Synchronizers

Distributed algorithms designed to run on a synchronous network are called *synchronous algorithms* and those designed to run on an asynchronous network are called *asynchronous algorithms*. Synchronous algorithms are usually easy to design and analyze for their complexities. This is not so in the case of asynchronous algorithms. Thus it is helpful to design a simulation technique that enables any synchronous algorithm to be executed on an asynchronous network. For this purpose, Awerbuch [Awe85] and [Awr85] proposed a simulation methodology called the *synchronizer* . A synchronizer is basically a distributed algorithm designed to convert programs written for synchronous networks into versions that can be used in asynchronous networks. It can be viewed as a layer of software, transparent to the user, placed on top of the asynchronous network to run synchronous algorithms.

The synchronizer is designed taking into account the assumption underlying a synchronous network model. Since each processor in a synchronous algorithm uses a global clock, a synchronizer similarly generates a sequence of clock pulses at each processor, such that a new pulse is generated only after it has received all messages of the synchronous algorithm sent to that processor by its neighbours at the previous pulse. But there is a synchronization problem associated with the synchronizer. A processor is not aware of which messages were sent to it by its neighbours. A solution to this problem is to allow the processor to wait. But, suppose one of its neighbours has no message to send. Then the processor may have to wait for an infinite amount of time since the link delay in an asynchronous network is unbounded. This problem can be eradicated by sending additional dummy messages for the purpose of synchronization.

The complexity analysis of a synchronous algorithm will be discussed now. The synchronizer v produces a certain overhead. Let C_v and T_v be the message and time complexities introduced by the synchronizer at each clock pulse, respectively. Since the algorithm may have an initialization phase, let $C_{v_{init}}$ and $T_{v_{init}}$ be the message and time complexities of the initialization phase, respectively. Let C_a, T_a and C_s, T_s denote the message and time complexities of the asynchronous algorithm A and synchronous algorithm S, respectively. Then the total message and time complexities introduced by the synchronizer are $C_v * T_s$ and $T_v * T_s$, respectively. Therefore the complexities of the asynchronous algorithm are:

$$C_a = C_{v_{init}} + C_s + C_v * T_s$$

$$T_a = T_{v_{init}} + T_s * T_v$$

Awerbuch proposed three types of synchronizers. Synchronizer α is time efficient, whereas synchronizer β is message efficient. The combination of α and β synchronizers results in the γ synchronizer, which is efficient both in messages and time.

Before describing synchronizers, a few concepts have to be explained. Awerbuch defines a node to be *safe* at a certain pulse if all the messages sent by the node at that pulse have arrived at their destinations. Using acknowledgement messages a node can detect that it is safe. That is, when acknowledgements for all messages sent by a node are received from its neighbours, then the node becomes safe. For proper computational purposes, it is also necessary for a particular node to know that all its neighbours are also safe. *Otherwise, there is no guarantee that messages sent at a previous pulse by a neighbour do not arrive in the future* . So each node has to somehow communicate its safety information to its neighbours at minimal cost.

A graph is said to be *acyclic* if it has no circuits. A *tree* is a connected acyclic graph. A *spanning tree* of a graph is a tree having all the vertices of the graph. Consider an unconnected graph G with k components. Then the collection of all the spanning trees of each component is a *spanning forest* .

2.2.1 α-Sychronizer

The α-synchronizer operates as follows. After a node receives an acknowledgement for each message sent, it declares itself safe and communicates this information to its neighbours. If all its neighbours are safe, the node generates the next pulse. Though there is a dependence between a node and its neighbours, the transmission of messages can be done in parallel.

As already mentioned, it is easy to see that this synchronizer is time efficient with a time complexity of O(1) per pulse. Since each link can hold at most one message, the message complexity per pulse is O(|E|).

2.2.2 β-Synchronizer

This synchronizer is essentially a variant of the α-synchronizer. As in the α-synchronizer, a node declares itself safe using the acknowledgement mechanism. However, the β-synchronizer employs a spanning tree to detect that all nodes have become safe. Thus an initialization phase is involved.

In the initialization phase, a rooted spanning tree is constructed from the communication graph G. Note that though G is undirected, directions are assumed for the purpose of creating a rooted spanning tree. A *leader*, usually the root of the tree, is then chosen. When a nonleaf node learns that it is safe and its children are also safe, the node, if it is not the leader of the tree, reports this information to its parent. Eventually the root receives this message and at this point knows that all the nodes in the tree are safe. The root then notifies this by broadcasting a message to all the nodes via the spanning tree allowing them to generate the next pulse.

It is obvious that initially the communication process is initiated by the leaves. Suppose the leaves are in level k. Then the leaves report to their parents at level k-1 that they are safe. The nodes in level k-1, after learning that they are safe and all their children are safe, convey this information to their respective parents. This process continues until the information reaches the root. As can be seen, all the operations at each level can be done in parallel and the β -synchronizer algorithm can be distributively implemented.

Since each message has to pass through the $|V|-1$ edges of the spanning tree, the message and time complexities per pulse are $O(|V|)$. This is in contrast to the α-synchronizer which requires only $O(1)$ additional time per pulse. Thus the β-synchronizer, as mentioned earlier, is not time efficient. But it is message efficient because it requires only $O(|V|)$ additional messages per pulse and $|V|-1 \leq |E|$.

2.2.3 γ-Synchronizer

This technique combines the methods of the α and β synchronizers. First the network is partitioned. A *cluster* is a spanning tree of one component. The combination of all these spanning trees is a spanning forest of the network. The clusters are connected by links. Between each pair of clusters one *preferred link* is chosen which will serve for communication between these clusters. As before a leader is selected for each cluster. A cluster is considered safe if all its nodes are safe.

The γ-synchronizer operates in two phases. In the first phase the β-synchronizer is applied separately in each cluster (using the tree). When the leader of a cluster detects that its cluster is safe, this information is broadcast to all the nodes in the cluster as well as the neighbouring clusters through preferred links. At this point, the nodes in the cluster enter the second phase. They wait in this phase until all the neighbouring clusters are safe and then generate the next pulse. Thus α-synchronizer scheme is used between clusters and the β-synchronizer scheme is used within a cluster.

A detailed complexity analysis of the γ-synchronizer may be found in [Awe85].

2.3 Limitations of Synchronizers

As mentioned before, for the correct working of a synchronous algorithm when combined with any synchronizer, it is necessary that before a node executes its k^{th} pulse, all messages sent to it by its neighbours at the $(k-1)^{th}$ pulse are received and processed. The α, β, and γ synchronizers ensure this. It is also necessary that before a node has started execution of its k^{th} pulse, any message that has been received from its neighbour executing its k^{th} pulse is not processed. This is not ensured by any of the three synchronizers described thus far.

Suppose after execution of its $(k-1)^{th}$ pulse, node i begins its k^{th} pulse and sends a message to its neighbour node j. It is possible for node j to receive this message before j has started its k^{th} pulse, thus modify some data structures maintained by node j, which it would probably have done during the execution of its k^{th} pulse. This means that some messages sent by the synchronous algorithm may not get sent in the simulated version. *Thus the simulated version may not behave exactly the same way as the corresponding synchronous algorithm.*

It is easy to see that the α-synchronizer suffers from the above problem.

In the β-synchronizer mechanism, the leader or the root sends a PULSE message down the tree to all the nodes allowing initiation of the next pulse. Since the tree is a rooted spanning tree, it is possible for a node i to receive the PULSE message before its neighbour j. Therefore as in the α-synchronizer methodology, it is possible for node i to complete actions of a particular pulse before node j starts the corresponding pulse.

Since the γ-synchronizer is a combination of the α and β synchronizers, the problem mentioned above persists in this case too.

We next illustrate the above problem using a simple synchronizer called the *GO-Synchronizer* described in [LaT87].

The process as usual is initiated by the leader of the nodes. The leader sends a WAKEUP message to all its nodes. Along with this, a GO message is also sent. The receipt of this message from all its neighbours notifies a node to start a new pulse. Every node sends an explicit GO message to its neighbour after sending a message of the synchronous algorithm. A node terminates its activity by sending a WINDUP message to the leader. The whole algorithm terminates when the leader receives WINDUP messages from all the nodes.

The synchronizer algorithm just described above requires a data structure called "go received". This data structure keeps track of the number of GO messages received from each of its neighbours. Note that at most two GO messages can be received from any neighbour. For example, suppose that at the $(k-1)^{th}$ pulse, node i and node j complete their computations and each sends a GO message to the other, permitting start of the next pulse. If node i receives the GO message earlier than node j then node i starts its k^{th} pulse. Suppose node i finishes the necessary computations of this pulse and sends another GO message to node j. Then node j will have received two GO messages from node i – one for the execution of the k^{th} pulse and for the $(k+1)^{th}$ execution. Thus the "go received" data structure is a multiset. Note that node i has to wait till it receives a GO message from node j before it starts its $(k+1)^{th}$ pulse. Thus at any point in time the number of pulses executed by two neighbouring nodes can differ by at most one. Also, the number of pulses completed by any two arbitrary nodes in the entire network can differ by at most the length of a shortest path between these nodes in the communicating graph.

A distributed synchronous algorithm for the Breadth-First-Search (BFS) tree of a communication network will now be explained and then an example will be given which will show that this algorithm combined with the synchronizer described above does not work correctly on an asynchronous network.

The BFS algorithm utilizes two types of messages – LABEL and ACK. The LABEL message carries a parameter and is sent by a node to its neighbouring nodes to inform them of its level number. This enables its neighbours to figure out their level numbers. An ACK message is sent in response to a LABEL message. The computation of the algorithm proceeds as follows. The leader at level 0 initiates the algorithm by sending a LABEL message to each of its neighbours during the first clock pulse. As soon as the neighbours receive this message, they figure out their level numbers and set their parent as the leader. Then they send a LABEL message to their neighbours. So in general, when any internal node receives a LABEL message, it figures out its level number and sets its parent as the node from which the LABEL message arrived. Note that the first LABEL message that arrives at a node determines its level number and its parent and cannot be updated later by the arrival of other LABEL messages. If a leaf receives a LABEL message, then its only neighbour is its parent and so an ACK message is sent to its parent. And if a node that has already been labelled receives a LABEL message, then an ACK message is sent to the corresponding node since as mentioned above, in this case no update is done. Each node waits until all ACK messages have been received for all LABEL messages sent. Then the node sends an ACK message to its parent from which it received the first LABEL message. Once the leader receives all the ACK messages, the algorithm terminates.

The BFS synchronous algorithm works properly because a message transmitted at each clock pulse arrives at its destination before the next clock pulse starts. This ensures that a node with label k receives its first LABEL message only during the k^{th} pulse. Clearly, this is not the case, in general, on an asynchronous network.

Now consider the execution of the above BFS synchronous algorithm combined with the GO synchronizer on a five node asynchronous network shown in Fig. 2.1. Note that GO messages of the synchronizer also serve the purpose of ACK messages in the

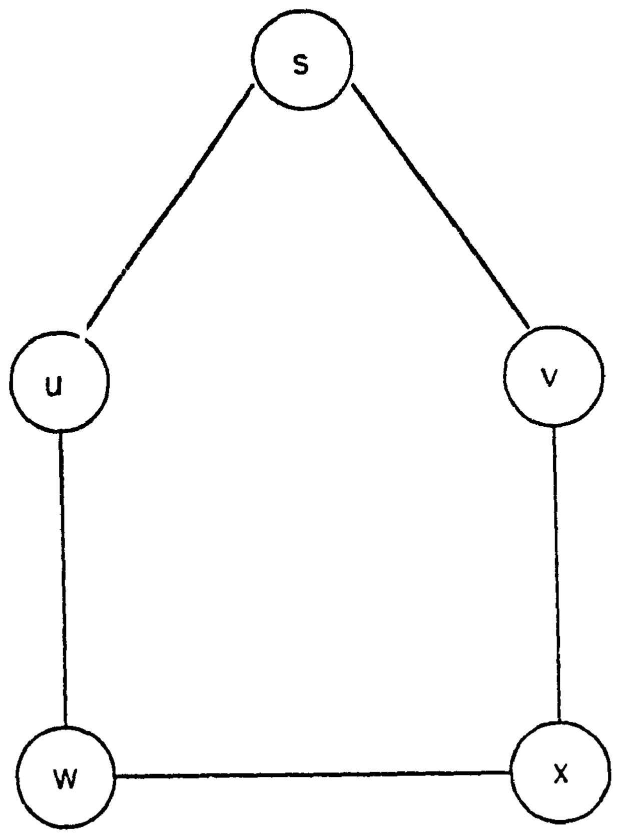


Fig. 2.1: Illustration of Go-Synchronizer.

asynchronous algorithm. This example will prove that sometimes simulating a synchronous algorithm on an asynchronous network using synchronizers may not produce the correct results. Let s be the leader. The leader initiates the algorithm by sending a WAKEUP message and then a GO message to u and v. The WAKEUP message awakes nodes u and v. Nodes w and x receive WAKEUP messages from u and v, respectively. Each of these nodes sends GO messages following the WAKEUP messages. Note that if a node has already woken up and another message is received, it is ignored. The initialization phase ends when each of the nodes receive GO messages from its neighbours. The nodes are then ready to execute their first pulse. The leader sends a LABEL(1) message to its neighbours u and v during the first pulse. Following this message, a GO message is sent to u and v by s. In the meantime, the nodes u, v, w and x send redundant GO messages to each of their neighbours for the purpose of synchronization. Suppose the nodes u and v receive the LABEL(1) messages and GO messages from s and GO messages from w and x respectively, without any delay. Then the nodes u and v figure out their level numbers as 1.

During the second pulse, nodes u and v send LABEL(2) messages to w and x, respectively. The node u sends a GO message to s and w. Similarly, node v sends a GO message to s and x. Meanwhile, nodes w and x on the other hand send GO messages to u and v, respectively, for synchronization. Now suppose node w receives the LABEL(2) message and a GO message from u and a GO message from x. But suppose the receipt of the LABEL(2) message from v to x is delayed. Then node w calculates its level number as 2 and sends a LABEL(3) and a GO message to node x and a GO message to u. During this pulse, node x is yet to receive the LABEL(2) message and when it receives the LABEL(3) message it determines its level number as 3. Recall that once a node that has already been labelled receives a label message, no update is done. Therefore, even if the LABEL(2) message from v arrives at x, no update is done to the level number of x. The algorithm has thus produced an incorrect result.

This algorithm can be modified as follows. Each LABEL message received from a node is allowed to update the level number of the destination node. This allows the early arrival of LABEL messages with higher level numbers to be corrected. This version will work correctly with the above synchronization. However such a simple modification may not be possible for arbitrary synchronous protocols.

The failure of the GO synchronizer to help exact simulation can thus cause incorrect implementation of an algorithm and hence may produce incorrect results. *Again the source of the problem is the fact that the messages sent by a node i during the k^{th} pulse may arrive at a neighbouring node j even before j has started its k^{th} pulse*. As we observed before this difficulty is present in other synchronizers too.

2.4 Possible Remedies

Summarizing the discussions of the previous section, we have the following. For the correct working of an arbitrary synchronous protocol when combined with any synchronizer, it is necessary that before a node executes its k^{th} pulse of activity, all messages sent to it by its neighbours during the $(k-1)^{th}$ pulse of activity must have been received and processed. All synchronizers proposed so far ensure this. It is also necessary that before a node executes its k^{th} pulse of activity, no message sent to it by its neighbours during the k^{th} pulse of activity should be processed, even if some have been received already. This is not ensured by any of the synchronizers discussed so far.

There are several ways of solving the above problem. Suppose while processing a message, it is valid to check upon some condition that determines whether to handle the message immediately or delay it by placing it in a queue. Then this would allow messages that have arrived earlier than expected to be placed in a queue to be processed later.

With the GO-synchronizer mechanism discussed above, it can be easily detected if a message is to be processed or delayed. Suppose a message has been received from node j and there already exists a pending unused GO message from j. Then this cautions the node to delay the processing of the current message. So, if two queues are kept for incoming messages, then the messages to be delayed can be placed in the second queue and then processed after generating a new pulse. α, β, and γ synchronizers can also be corrected by requiring each message to carry a pulse number, which can be used as a basis for deciding to delay the processing of messages that arrive early. Note that these corrections do not affect the complexities of the synchronizers. *But they will not be valid if the model of asynchronous computation does not permit a processor to delay processing of messages once received.*

We now present a modification to the β-synchronizer, which will not require delaying of processing of a message once received. Recall that in a β-synchronizer, an elected leader coordinates the pulse-by-pulse activity of all nodes in the network. Here, when the leader decides to send down the PULSE message, notifying all nodes to start a new pulse of activity, it is guaranteed that all nodes have completed the previous pulse, no messages of the synchronous algorithm are still in transit, and that all nodes are ready to start a new pulse. But the problem is that the notification to start a new pulse of activity is propagated via a rooted spanning tree and sometimes this notification may arrive at a node later than an useful message of the synchronous algorithm transmitted by a neighbouring node which has already completed its new pulse of activity. One way to remedy this will be to flood the network of this notification to start a new pulse, using the protocol of Segall [Seg83], as is done for WAKEUP and WINDUP messages of our synchronizer. In other words, each node, when it first receives the PULSE message, informs each of its neighbours of that fact before proceeding with the execution of the new pulse.

Subsequent PULSE messages received via other neighbouring nodes can always be ignored. To accomplish this, a PULSE message must carry one bit of information that alternates between 1 and 0, indicating whether it corresponds to an odd numbered or an even numbered pulse. Correspondingly, each node must maintain a one bit flag to keep track of the number of pulses of activity gone through. Based on this bit of information a node can decide to handle or ignore a PULSE message. As before, after completing a pulse of activity, a node may have to wait until it recognizes itself as safe before informing the leader so. This information can, of course, flow through the tree just as in a β-synchronizer. However, observe that this solution then means that the message and time complexity overheads of the modified β -synchronizer are C_{pulse} = O(m) and T_{pulse} = O(n), respectively.

The message complexity overhead of the modified β-synchronizer can now be reduced, if the PULSE message is forwarded by each node only to those neighbours to whom it intends to send a useful message of the synchronous algorithm, during that pulse of activity. But, in order to be sure that each node receives the PULSE message at least once, it may have to be propagated via the rooted spanning tree also, as in a normal β-synchronizer. Now, observe that the message complexity overhead resulting from the PULSE messages that flow along non-tree edges can be absorbed along with the message complexity of the synchronous algorithm, without affecting its asymptotic nature. Thus, this modified β-synchronizer, which ensures the correct working of an arbitrary synchronous protocol, also has only C_{pulse} = O(n) and T_{pulse} = O(n), overhead per pulse.

It can be shown that the α and γ synchronizers do not admit any modification without requiring considerable increase in their complexities. Whereas the remedies we have discussed (as presented in [LaT87]) do not require delaying of processing of a message, a recent paper [SeS91] presents remedies which permit delaying of processing of a message once received but do not require messages to carry pulse numbers.

2.5 Summary

In this chapter we have discussed the notion of a synchronizer and presented the α, β and γ synchronizers proposed by Awerbuch [Awe85]. We have also illustrated the difficulties which one may encounter in the implementation of the synchronizer and presented the approaches available to overcome these difficulties [LaT87]. In Chapter VI we shall explain how the synchronizers are implemented in a shared-memory model.

CHAPTER III

SHORTEST PATH AND PRIMAL-DUAL INITIALIZATION ALGORITHMS

Consider a connected directed graph G in which each directed edge is associated with a real number called the *length* of the edge. The length of an edge directed from a vertex v_i to a vertex v_j is denoted by $w(e) = w(v_i , v_j) = w_{ij}$. If there is no edge directed from v_i to v_j, then $w(v_i , v_j) = \infty$. The length of a directed path in G is the sum of the lengths of the edges in the path. A minimum length directed $s-t$ path is called a *shortest path* from s to t. The length of a shortest directed $s-t$ path, called the *distance* from s to t, is denoted as $d(s , t)$.

In this chapter we consider the following two problems:

(i) Single-Source Shortest Path Problem: Find the shortest paths from a specified vertex s to all the vertices in G.

(ii) Initialization of the Primal-Dual Method for the Transshipment Problem.

In the following chapter, we present an algorithm for the single-source shortest path problem. We then show how the primal-dual initialization problem can also be formulated as a single-source shortest path problem.

3.1 Single-Source Shortest Path Algorithm

Two efficient algorithms are available for the single-source shortest path problem. They are due to Dijkstra [Dij59], and Bellman, Ford and Moore [Bel58], [FoF62], [Moo57]. Though Dijkstra's algorithm has a better complexity than the Bellman-Ford-Moore (BFM) algorithm, it is not applicable when the graph has some negative length edges. Since the graph underlying a transshipment problem may have some negative length edges, Dijkstra's algorithm is not suitable for the application of interest to us. So we shall focus our discussion on the BFM algorithm.

The BFM algorithm is very elegant and easy to present. Initially it assigns a label, LABEL(s) = 0 to the vertex s and assigns LABEL(v) = ∞, for every other vertex v. The algorithm then repeatedly performs the following operation:

Select an edge $e = (v_i, v_j)$ such that LABEL(v_j) $>$ LABEL(v_i) $+$ $w(v_i, v_j)$ and set LABEL(v_j) $=$ LABEL(v_i) $+$ $w(v_i, v_j)$.

The algorithm terminates when the above operation is no longer applicable. At termination LABEL(v) gives the length of a shortest path from s to v.

A formal presentation of the BFM algorithm is given below:

Algorithm 3.1: THE BELLMAN-FORD-MOORE SHORTEST PATH ALGORITHM

S0. G is the given directed graph with lengths associated with edges. Shortest paths from vertex s to all the other vertices in G are required.

S1. (Initialize.) Set LABEL(s) = 0 and PRED(s) = s. For all $v \neq s$, set LABEL(v) = ∞ and PRED(v) = v.

S2. If there exists no edge $e = (v_i, v_j)$ for which LABEL(v_j) > LABEL(v_i) + $w(e)$, then HALT. (The current vertex label values represent the lengths of the shortest paths.)

S3. Select an edge $e = (v_i, v_j)$ for which LABEL(v_j) > LABEL(v_i) + $w(e)$. Set LABEL(v_j) = LABEL(v_i) + $w(e)$ and PRED(v_j) = v_i. Go to S2.

The PRED(v) used in the BFM algorithm serve one important purpose. PRED(v_j), at any step in the algorithm, denotes the vertex from which v_j has received its current label. At termination, PRED(v)'s can be used to trace shortest paths from s to v. For instance a shortest path from vertex s to vertex v would contain the sequence of vertices $s = v_0, v_1, v_2, \cdots, v_k = v$ such that PRED(v_i) = v_{i-1}, $1 \leq i \leq k$. At termination, the edges (PRED(v) , v) also constitute a spanning tree rooted at s.

For our purposes in the BFM algorithm ∞ is not greater than $\infty+k$ even if k is negative. Also, the algorithm will not terminate if there is a directed path from s to a vertex on a directed circuit of negative length because in such a case, going around this circuit will decrease label values, and the process can be repeated indefinitely. We now prove that in all other cases the algorithm will terminate in a finite number of steps and at termination, for every vertex v, LABEL(v) will give the length of a shortest directed path from s to v.

LEMMA 3.1 : If the graph G has no negative directed circuits of negative length, then, if at any stage of the BFM algorithm LABEL(v) is finite, there is a directed path from s to v whose length is LABEL(v).

Proof

We prove the result by displaying a directed path from s to v. The construction is backward from v.

Let u denote the vertex that gave v its present LABEL(v). If LABEL (u) represents the label of u at the time it gave v the label LABEL(v), then LABEL(v) = LABEL (u) $+w(e)$, where $e = (u, v)$. Now, continue from u to the vertex that gave it the label LABEL (u), and so on. In this process no vertex will be encountered more than once because each step in the process refers to an earlier time in the execution of the algorithm, and a vertex can decrease its own label only by going through a directed circuit of negative length. Thus there is a directed path from s to v such that the LABEL(v) is the sum of the lengths of the edges on this path. Thus the required result follows.

□

Recall that $d(s, v)$, the distance from s to v, denotes the length of a shortest directed path from s to v.

THEOREM 3.1 : For a directed graph G with no directed circuits of negative length, the Bellman-Ford-Moore Algorithm terminates in a finite number of steps, and upon termination LABEL(v) = $d(s, v)$ for all v.

Proof

Consider any vertex v in G By Lemma 3.1, at any stage of the Bellman-Ford-Moore algorithm, LABEL(v) represents the length of a directed path from s to v. Since there are only a finite number of such paths, it follows that the number of possible values for LABEL(v) is also finite.

Clearly, upon termination of the algorithm, LABEL(v) $\geq d(s, v)$. If LABEL(v) > $d(s, v)$, then let P : $s = v_0, v_1, \cdots, v_k = v$ be a shortest path from s to v and let e_i denote the edge (v_{i-1}, v_i) on this path. For every $i = 0, 1, 2, \cdots,$ k, we have

$$d(s, v_i) = \sum_{j=1}^{i} w(e_j)$$

Let v_i be the first vertex on this path for which, upon termination of the algorithm, LABEL(v_i) > $d(s, v_i)$. Since LABEL(v_{i-1}) = $d(s, v_{i-1})$ we have $d(s, v_i)$ = LABEL(v_{i-1}) + $w(e_i)$ and so upon termination LABEL(v_i) > LABEL(v_{i-1}) + $w(e_i)$. This is a contradiction because when the algorithm terminates there is no edge $e(u, w)$ with LABEL(w) > LABEL(u) + $w(e)$. Thus, for every vertex v_i on P, LABEL(v_i) = $d(s, v_i)$. In particular, LABEL(v_k) = LABEL(v) = $d(s, v)$ and the theorem follows.

□

Note that the number of directed paths from s to a vertex could be exponential in the number of vertices of the graph. So, if in the implementation of the BFM algorithm edges are seleced in an arbitrary manner, it is possible that the algorithm may perform an exponential number of operations before terminating. We can show that the following implementation of the algorithm will achieve a complexity of O(mn) where m and n are, respectively, the number of edges and the number of vertices of G.

Order the vertices as $v_1, v_2, \cdots, v_n$. Pick vertices in this order, and for each vertex v_i selected, examine all the edges directed out of v_i and perform step S3 on these edges whenever it is applicable. After one such sweep (examination) of all the vertices, perform additional sweeps until an entire sweep produces no changes in the vertex labels. If the number of edges in a shortest directed path from s to v is k, then it can be shown by induction that by the end of the k^{th} sweep, v will have its final label. Since $k \leq (n-1)$ and each sweep requires $O(m)$ operations, the implementation of the Bellman-Ford-Moore algorithm requires $O(mn)$ time.

A further variant of the BFM algorithm, which has the same complexity as the above implementation but yet is amenable for a distributed implementation is presented below.

```
S1: (Initialize) Set LABEL($s$) = 0, and PRED($s$) = $s$. For all $v \neq s$, set LABEL($v$)
    = TLABEL($v$) = $\infty$ and PRED($v$) = $v$.
S2: For each $i = 1, 2, \ldots, n$ and each edge $e = (v_i, v_j)$, do:
        If LABEL($v_j$) > LABEL($v_i$) + $w(e)$ and TLABEL($v_j$) > LABEL($v_i$) +
        $w(e)$, set TLABEL($v_j$) = LABEL($v_i$) + $w(e)$ and PRED($v_j$) = $v_i$.
S3: If for every $i = 1, 2, \ldots, n$, LABEL($v_i$) = TLABEL($v_i$), then HALT.
S4: For each $i = 1, 2, \ldots, n$, set LABEL($v_i$) = TLABEL($v_i$), and go to step
    S2.
```

Note that in step S2 of the above algorithm, one sweep of all the vertices is performed. After each sweep, label values are updated in step S4. These new label values are used in the next sweep.

As an example, consider the graph of Fig. 3.1 with edge lengths as shown. Shortest

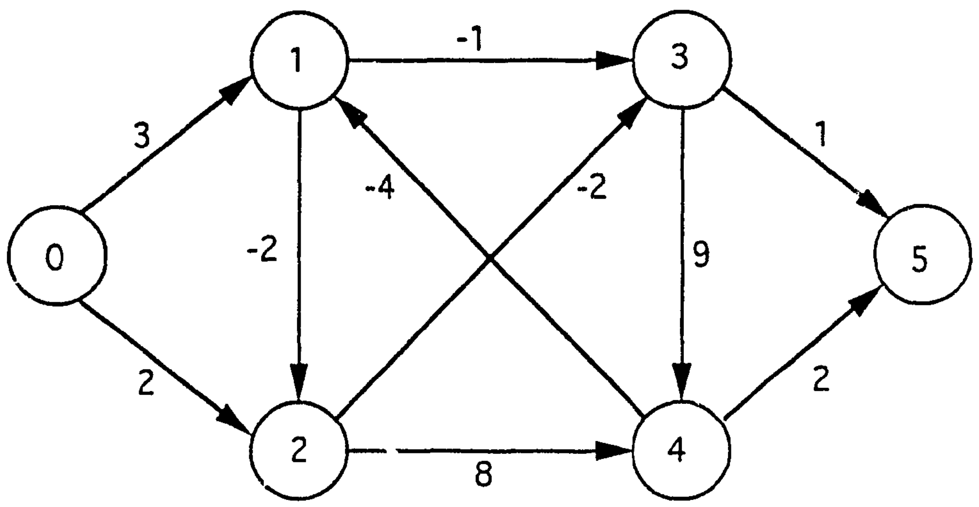


Fig. 3.1: A Directed Graph G with Edge Lengths.

paths from vertex 0 are required. Initially LABEL(0) = 0 and LABEL(v) = ∞ for all v. The label values at the end of the sweeps are :

Sweep	LABEL(0)	LABEL(1)	LABEL(2)	LABEL(3)	LABEL(4)	LABEL(5)
1	0	3	2	∞	∞	∞
2	0	3	1	0	10	∞
3	0	3	1	-1	9	1
4	0	3	1	-1	8	0
5	0	3	1	-1	8	0

Algorithm terminates at the end of sweep 5. It can be verified that vertices have the PRED values:

PRED(0) = 0

PRED(1) = 0

PRED(2) = 1

PRED(3) = 2

PRED(4) = 3

PRED(5) = 3

3.2 Initialization of the Primal-Dual Method

As we shall see in Chapter V, the problem of initializing the primal-dual method for the transshipment problem can be formulated as follows.

Given a graph G = (V,E) with cost w_{ij} associated with each edge $e = (v_i, v_j)$, and variable y_i associated with each vertex v_i. Then, the primal-dual initialization problem is: Determine y_i's such that for each edge (i, j), $y_i - y_j + w_{ij} \geq 0$.

If all w_{ij}'s are non-negative then setting every $y_i = 0$ will give a required solution. The problem becomes interesting when some w_{ij}'s are negative. In [CoT88] it is shown that y_i's selected as follows will give a required solution :

$$y_i = Max \left[0 \ , -\underset{j}{Min} \ \{ \ d_{ij} \ \} \right]. \tag{3.1}$$

where d_{ij} denotes the length of a shortest path from node v_i to v_j. Here, w_{ij} represents the length of edge (v_i , v_j). In other words y_i is equal to the negative of the length of the most negative shortest path originating at vertex v_i, if a negative length path exists; otherwise y_i is zero.

For instance, for the graph shown in Fig. 3.1, we have

$$y_0 = 1$$
$$y_1 = 4$$
$$y_2 = 2$$
$$y_3 = 0$$
$$y_4 = 8$$
$$y_5 = 0.$$

Calculating y_i's as given by (3.1) would require finding shortest paths between all pairs of vertices and then taking the appropriate maximum. However, we can reduce the problem to the single-source shortest path problem as follows.

1. First reverse the orientations of all the edges in the given graph G.
2. To this graph add a new vertex s and connect s to all the vertices with the edges directed away from s. Assign zero costs to these edges.

If G' denotes the graph constructed as above and d'_{ij} denotes the length of a shortest path in G' from vertex v_i to vertex v_j, then y_i's given by (3.1) can also be obtained from the following

$$y_i = -d'_{s,i}. \tag{3.2}$$

The fact that the y_i's computed using (3.2) indeed satisfies (3.1) can be shown from the following:

1. There is a one-to-one correspondence between the directed $i-j$ paths in G and the directed $j-i$ paths in G', for all $i,\ j \neq s$.

2. A shortest $i-j$ path in G has the same length as the shortest $s-i$ path in G'.

3. If $d_{i,k} = \underset{j}{Min}(d_{i,j}) < 0$ then $d_{s,i}' = d_{i,k}$. So by (3.1) in this case $y_i = -d_{i,k} = -d_{s,i}'$. If $d_{i,k} \geq 0$ then by (3.1) $y_i = 0$. But in this case $s-i$ path in G' also has length equal to zero. Thus in both cases (3.2) correctly computes y_i's.

As an example, the graph G' constructed from the graph G of Fig. 3.1 is shown in Fig. 3.2. We can now verify that (3.1) and (3.2) both yield the same values for y_i's.

3.3 A Distributed Protocol for the Single-Source Shortest Path Problem

We now return to the single-source shortest path problem considered in Section 3.1 and consider the design of a distributed protocol for this problem.

The variant of the BFM algorithm presented at the end of Section 3.1 is amenable to a distributed implementation. However, we need to add to this algorithm two mechanisms – one to detect termination and the other to detect the presence of a directed circuit of negative length. Recall that the algorithm terminates when the distances of all the vertices reach their final distance values. We adopt schemes given by Chandy and Misra [ChM82].

We now present the essential features of a synchronous distributed protocol

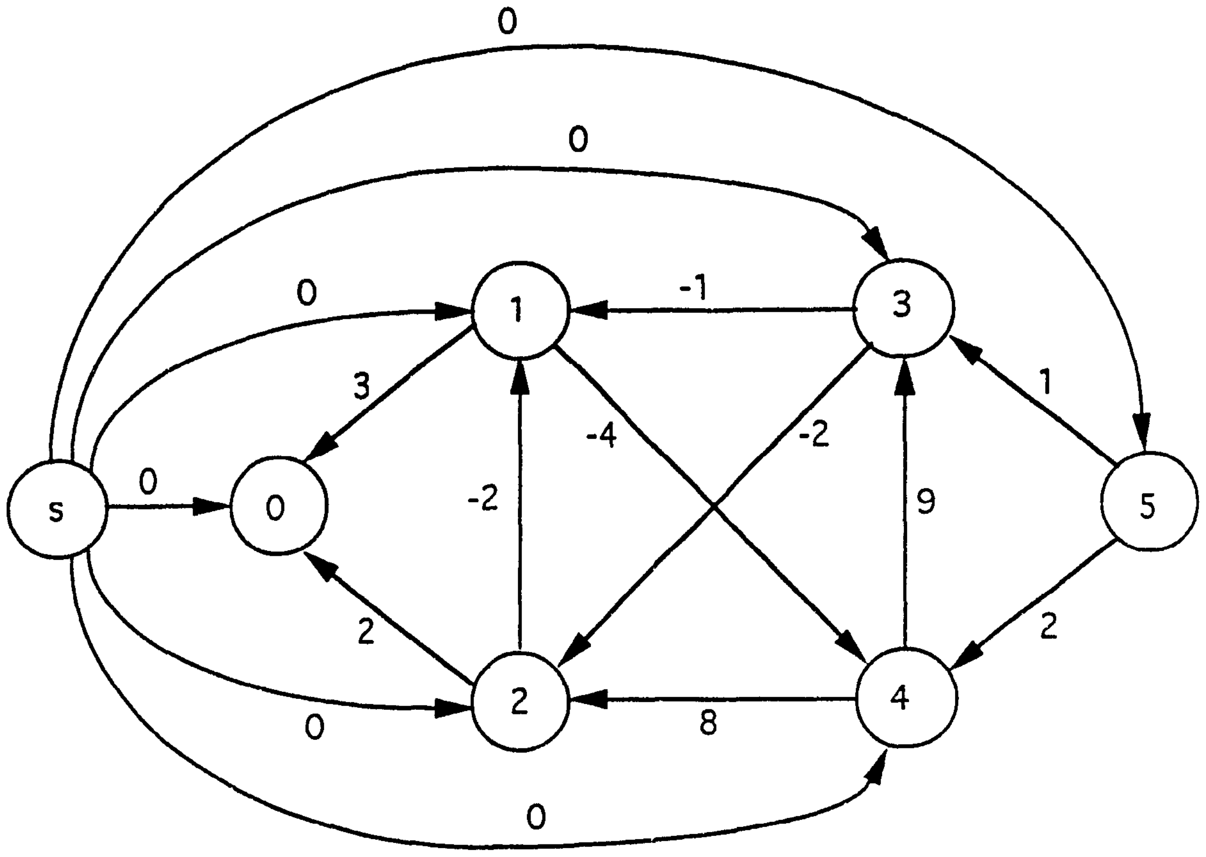


Fig. 3.2: Graph G' derived from Graph G of Fig. 3.1.

for the single-source shortest path problem. We are required to determine the shortest paths from a specified vertex s to all the other vertices in a graph G. The protocol uses four types of messages – DISTANCE, ACK, TERMINATE and INFEASIBLE messages .

The Distance message sent by a node carries the value of its current distance. A node sends an ACK message for every DISTANCE message received. The ACK messages help in detecting termination of the distributed protocol: the vertex s detects termination when it receives ACK messages for all the messages it has sent. Soon after detecting termination, vertex s sends TERMINATE messages to all vertices adjacent to it. These vertices then broadcast this information to adjacent vertices and so on. A vertex detects the presence of a directed circuit of negative length when it finds, at the time it receives a TERMINATE message, that an ACK message is yet to be received for a DISTANCE message it had sent earlier. In the following PRED(v) of vertex v has the same meaning as in Algorithm 3.1, and $d(v)$ is used in place of LABEL(v). We also assume that each vertex processor v is aware of the *lengths of the incoming edges at* v .

Algorithm 3.2 : A SYNCHRONOUS DISTRIBUTED PROTOCOL FOR THE SINGLE-SOURCE SHORTEST PATH PROBLEM

(1) During the first pulse, the following actions are performed by the different vertices.

(i) Vertex s sets $d(s) = 0$, PRED(s) $= s$ and sends DISTANCE messages to all adjacent vertices along all outgoing edges.

(ii) Each vertex $v \neq s$ sets $d(v) = \infty$ and PRED(v) $= v$.

(2) During each subsequent pulse, each vertex v examines the messages

received

and performs the following actions :

Vertex s

If it has received ACK messages for all the DISTANCE messages it has sent during the first pulse, then it detects termination and sends a TERMINATE message to each adjacent vertex.

Vertex $v \neq s$

(i) If v receives any ACK message, then it checks to see if ACK messages have been received for all the DISTANCE messages it had sent. If so, it sends an ACK message to PRED(v).

(ii)

a) Suppose v receives DISTANCE messages along edges $(i_1, v), (i_2, v), \ldots, (i_k, v)$. Then v computes

$$\text{MIN} = \underset{j}{Min} \left\{ d(i_j) + w(i_j, v) \right\}.$$

b) If $d(v) >$ MIN and MIN $= d(i_j) + w(i_j, v)$ then v does the following:

(i) Sends ACK messages to all the vertices i_r, $r \neq j$.

(ii) Sends an ACK message to PRED(v).

(iii) Sets $d(v)$ = MIN and sets PRED(v) $= i_j$.

(iv) Sends DISTANCE messages along outgoing edges.

(iii) If it receives a TERMINATE message, it checks to see if ACK messages have been received for all the DISTANCE messages it had sent earlier. If so, it sends TERMINATE messages to all adjacent vertices. If not, it sends INFEASIBLE messages to all adjacent vertices.

The above synchronous protocol can be converted into an asynchronous protocol by augmenting it with a synchronizer. In the case of the shortest path problem, the α-synchronizer or the simpler GO-synchronizer discussed in the previous chapter will be appropriate. All the details of incorporating the synchronizer mechanism will be discussed in Chapter VI, where we discuss simulation of our distributed algorithms.

3.4 Summary

In this chapter, we have presented the Bellman-Ford-Moore Algorithm for the single-source path problem and a variant of this algorithm that is amenable for distributed implementation. We have shown how the primal-dual initialization problem can be formulated as a single-source shortest path problem. Finally the main features of a synchronous distributed protocol for the shortest path problem have been discussed. Several issues relating to synchronization that will be required while running the protocol in an asynchronous environment will be discussed in Chapter VI.

CHAPTER IV
TRANSPORT NETWORK AND THE MAXIMUM FLOW PROBLEM

A *transport network* represents a model for transportation of a commodity from its production center to its market through communication routes. The network is thus a connected directed graph N=(V,E) with no self loops (cannot transport to itself). N has to satisfy the following conditions:

1. There is only one node with zero indegree; this is designated as the *source* (production center) and is denoted as s.

2. There is only one node with zero outdegree; this is designated as the *sink* (market) and is denoted as t.

3. Every directed edge $e = (i, j)$ in N is assigned a non-negative real number $c(i, j)$, the *capacity* of (i, j). $c(i, j) = 0$ if there is no edge directed from i to j.

4. Every directed edge $e = (i, j)$ in N is assigned a non-negative flow $f(i, j)$.

The capacity of an edge can be thought of as representing the maximum amount of some commodity that can be transported along the edge.

A flow f through a transport network N is an assignment of non-negative real numbers $f(i, j)$ to the edges (i, j) such that the following conditions are satisfied:

1. *capacity constraint:* The flow along an edge cannot exceed the capacity of an edge. Therefore,

$$0 \le f(i, j) \le c(i, j), \qquad \forall \quad (i, j) \in E.$$

2. *conservation constraint:* For each vertex i, except the source s and the sink t, the material transported into i is equal to the material transported out of i.

Therefore,

$$\sum_j f(i,j) - \sum_j f(j,i) = \begin{cases} \sum_j f(s,j) & \text{, if } i = s \text{ .} \\ -\sum_j f(j,t) & \text{, if } i = t \text{ .} \\ 0 & \text{, } otherwise \text{ .} \end{cases}$$

Fig. 4.1 shows a typical transport network.

The *value of a flow f* denoted by $val(f)$ is defined as

$$val(f) = \sum_j f(s,j) \text{ .}$$

In Fig. 4.1 $val(f) = 6$. Note that because of the conservation constraint, the total amount of material transported out of the source is equal to the total amount transported into the sink. This can be verified for the network in Fig. 4.1. Thus

$$val(f) = \sum_j f(s,j) = \sum_j f(j,t).$$

A flow f^* in a transport network N is said to be *maximum* if there is no flow f in N such that $val(f) > val(f^*)$.

The *maximum flow (in short, the max-flow) problem* is to find a maximum flow in a transport network.

4.1 Two Fundamental Theorems in Network Flow Theory

In this section we present two fundamental theorems in network flow theory , namely, the *augmentation path* theorem and the *max-flow min-cut* theorem. They form the basis of all algorithms for the max-flow problem.

Consider a connected graph N with vertex set V. Let S and $\bar{S} = V - S$ be two mutually disjoint subsets of V such that $V = S \cup \bar{S}$. That is, S and $\bar{S}$ have no

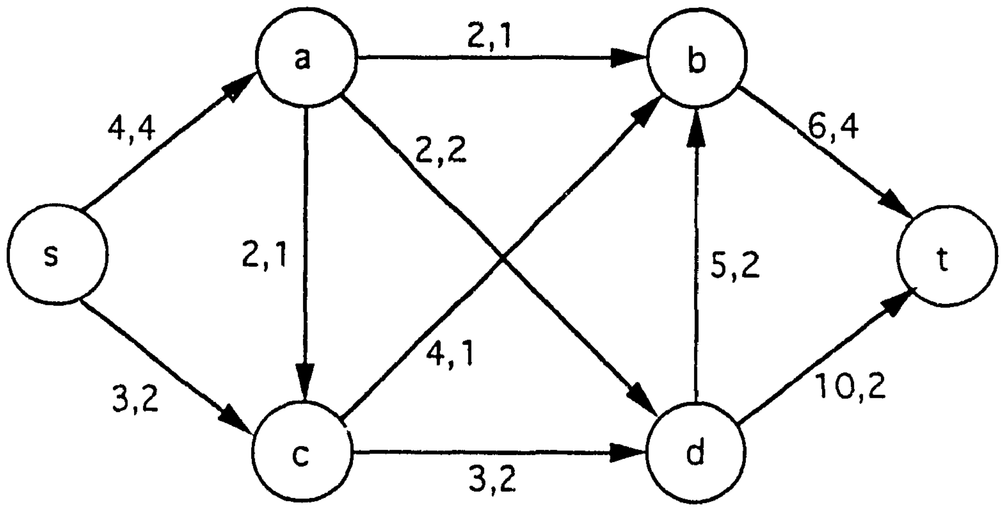


Fig. 4.1: A Transport Network.

common vertices and together contain all the vertices of V. Then the set of all those edges having one end vertex in S and another in $\bar{S}$ is called a *cut* of G, and is denoted by $<S, \bar{S}>$.

A cut $<S, \bar{S}>$ in a transport network N is said to separate the source s and the sink t, if $s \in S$ and $t \in \bar{S}$. Such a cut will be referred to as an $s-t$ cut. The capacity $c(S, \bar{S})$ of a cut $<S, \bar{S}>$ is defined as

$$c(S, \bar{S}) = \sum_{i \in S} \sum_{j \in \bar{S}} c(i, j)$$

Note that the capacities of the edges directed from $\bar{S}$ to S do not contribute to the capacity of the cut $<S, \bar{S}>$. Let,

$f(S, \bar{S})$ = the sum of the flows in the edges directed from S to $\bar{S}$.

$f(\bar{S}, S)$ = the sum of the flows in the edges directed from $\bar{S}$ to S.

An edge (i, j) is *f-saturated* if $f(i, j) = c(i, j)$; it is *f-unsaturated*, otherwise. An edge (i, j) is said to be *f-zero* if $f(i, j) = 0$; it is *f-positive*, otherwise.

Let us consider an example. Consider the cut $<S, \bar{S}>$ in the transport network of Fig. 4.1, where $S = \{s, a, b, c\}$ and $\bar{S} = \{d, t\}$. This cut is shown in Fig. 4.2.

$$\begin{aligned} c(S,\bar{S}) &= \sum_{i \in S} \sum_{j \in \bar{S}} c(i, j) \\ &= c(a, d) + c(b, t) + c(c, d) \\ &= 2 + 6 + 3 \\ &= 11, \end{aligned}$$

$$\begin{aligned} f(S,\bar{S}) &= f(a, d) + f(b, t) + f(c, d) \\ &= 2 + 4 + 2 = 8, \end{aligned}$$

and $f(\bar{S}, S) = 2$.

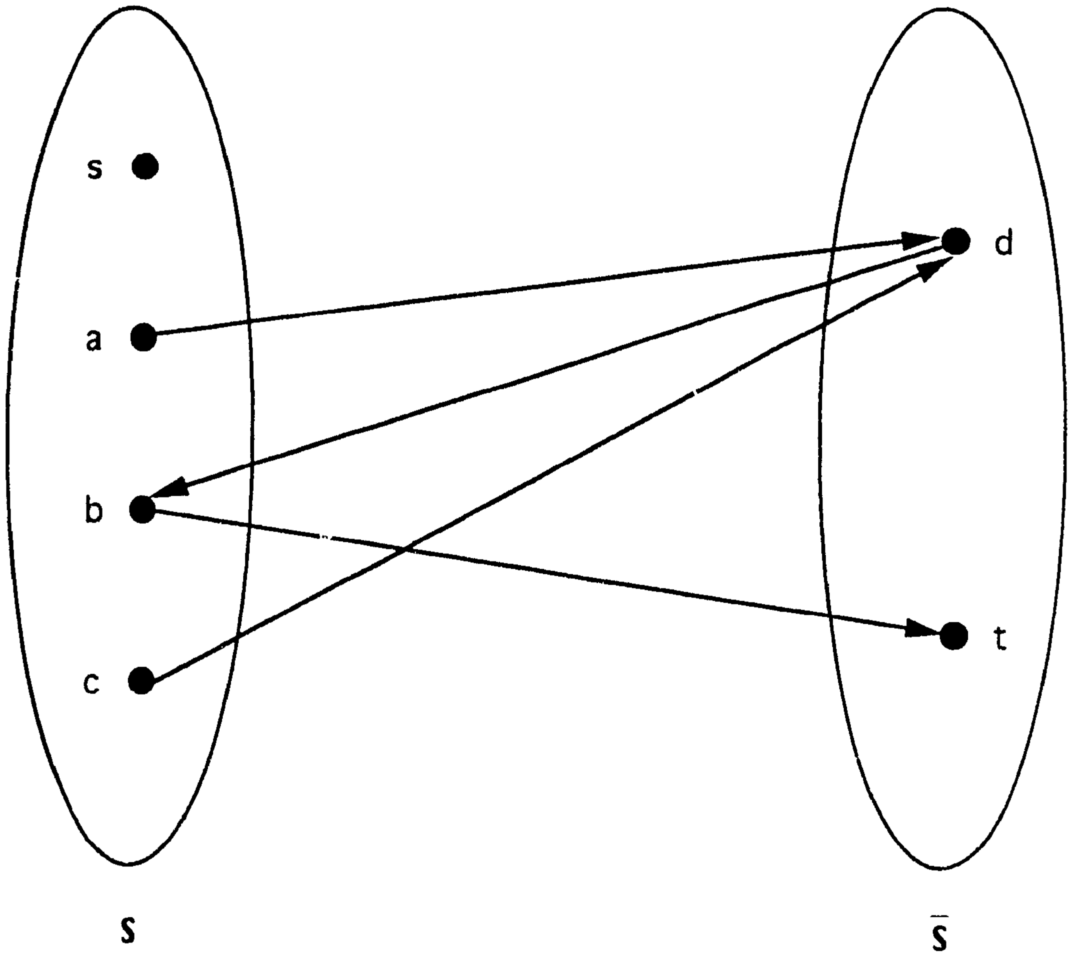


Fig. 4.2: s-t Cut $<S,\bar{S}>$.

The following theorem is a consequence of the conservation constraint.

THEOREM 4.1 : For any flow f and any $s-t$ cut $<S\ ,\bar{S}>$ in a transport network N,

$$val(f) = f(S\ ,\bar{S}) - f(\bar{S}\ ,S).$$

□

COROLLARY 4.1.1: For any flow f and any $s-t$ cut $<S\ ,\bar{S}>$ in a transport network N

$$val(f) \leq c(S\ ,\bar{S}).$$

Proof

We have,

$$val(f) = \sum_{i \in S} \sum_{j \in \bar{S}} f(i\ ,j) - \sum_{i \in S} \sum_{j \in \bar{S}} f(j\ ,i) = f(S\ ,\bar{S}) - f(\bar{S}\ ,S)$$

That is, the value of any flow is equal to the net flow through any cut. However, $0 \leq f(i\ ,j) \leq c(i\ ,j)$. Therefore,

$$\begin{aligned} val(f) &\leq f(S\ ,\bar{S}) \\ &= \sum_{i \in S} \sum_{j \in \bar{S}} f(i\ ,j) \\ &\leq \sum_{i \in S} \sum_{j \in \bar{S}} c(i\ ,j) \\ &= c(S\ ,\bar{S}) \end{aligned}$$

□

Note that $val(f) = c(S\ ,\bar{S})$ if $f(\bar{S}\ ,S) = 0$ and $f(S\ ,\bar{S}) = c(S\ ,\bar{S})$. In other words, $val(f) = c(S\ ,\bar{S})$ if all the edges directed from S to $\bar{S}$ are f-saturated and those directed from $\bar{S}$ to S are f-zero.

An $s-t$ cut $<S\ ,\bar{S}>$ in a transport network N is *minimum* if there is no cut $<K\ ,\bar{K}>$ in N such that $c(K\ ,\bar{K}) < c(S\ ,\bar{S})$.

COROLLARY 4.1.2: Let f be a flow and $<S\ ,\bar{S}>$ be an $s-t$ cut such that

$val(f) = c(S, \bar{S})$. Then f is a maximum flow and $<S, \bar{S}>$ is a minimum $s-t$ cut.

Proof

Let f be a flow and $<S, \bar{S}>$ be an $s-t$ cut such that $val(f) = c(S, \bar{S})$. Assume f^* is a maximum flow and $<K, \bar{K}>$ is a minimum $s-t$ cut. Then by Corollary 4.1.1,

$$val(f^*) \leq c(K, \bar{K}).$$

Also,

$$val(f) \leq val(f^*) \text{ and}$$

$$c(S, \bar{S}) \geq c(K, \bar{K}).$$

Therefore,

$$val(f) \leq val(f^*) \leq c(K, \bar{K}) \leq c(S, \bar{S}).$$

But by hypothesis, $val(f) = c(S, \bar{S})$. So,

$$val(f^*) = c(K, \bar{K}).$$

Thus,

$$val(f) = val(f^*) = c(K, \bar{K}) = c(S, \bar{S})$$

and f is a maximum flow and $<S, \bar{S}>$ is a minimum cut.

□

To understand and prove the maximum flow minimum cut theorem, we need the augmenting path theorem presented next.

Consider a transport network N with a flow f. Let P be a path (not necessarily directed) in N from the source s to some vertex v. Suppose that

$$P : s = u_0, u_1, u_2, \ldots, u_{i-1}, u_i, \ldots, u_k = v.$$

Let e_i denote the edge connecting vertices u_{i-1} and u_i. Then an edge e_i of P directed from u_{i-1} to u_i is called a *forward edge* of P. An edge e_i of P directed from u_i to u_{i-1} is called a *backward edge* of P. For each edge e_i in P, let

$$\varepsilon_i(P) = \begin{cases} c(e_i) - f(e_i) & \text{, if } e_i \textit{ is a forward edge.} \\ f(e_i) & \text{, if } e_i \textit{ is a backward edge.} \end{cases}$$

Define $\varepsilon(P) = \min\limits_{i} \{ \varepsilon_i(P) \}$. Note that $\varepsilon(P) \geq 0$.

A path is said to be f –unsaturated if the following two conditions are satisfied :

1. $f(i,j) \neq c(i,j)$ for every forward edge (i,j).

2. $f(i,j) > 0$ for every backward edge (i,j).

An $s-t$ path P is called an *f-augmenting path* if P is f –unsaturated. If P is f –augmenting then $\varepsilon(P) > 0$.

Given a network N, let P be an f –augmenting $s-t$ path. Then a new flow $\hat{f}$ can be obtained as follows:

$$\hat{f}(e) = \begin{cases} f(e) + \varepsilon(P) & \text{, if } e \textit{ is a forward edge of } P. \\ f(e) - \varepsilon(P) & \text{, if } e \textit{ is a backward edge of } P. \\ 0 & \textit{, otherwise.} \end{cases}$$

Thus $val(\hat{f}) = val(f) + \varepsilon(P)$ and $val(\hat{f}) > val(f)$. So if an f -augmenting path P exists, then the flow f is not maximum.

Consider the network N shown in Fig. 4.1. Let flow f be as shown in this figure. In this figure the numbers assigned to the edges indicate capacity and flow in this order. If the path P consists of edges (s,c),(c,a),(a,b) and (b,t), then (s,c),(a,b) and (b,t) are forward edges, and (c,a) is a backward edge. With respect to the flow f,

$$\begin{aligned} \varepsilon_1 &= \varepsilon_{(s,c)}(P) = 3 - 2 = 1 \\ \varepsilon_2 &= \varepsilon_{(c,a)}(P) = 1 \\ \varepsilon_3 &= \varepsilon_{(a,b)}(P) = 2 - 1 = 1 \\ \varepsilon_4 &= \varepsilon_{(b,t)}(P) = 6 - 4 = 2. \end{aligned}$$

Hence,

$$\varepsilon(P) = \min(\varepsilon_1(P), \varepsilon_2(P), \varepsilon_3(P), \varepsilon_4(P)) = 1$$

Since $\varepsilon(P) > 0$, P is f-augmenting. The revised flow $\hat{f}$ based on P is:

$$\hat{f}(s, c) = f(s, c) + \varepsilon(P) = 2 + 1 = 3$$

$$\hat{f}(c, a) = f(c, a) + \varepsilon(P) = 1 - 1 = 0$$

$$\hat{f}(a, b) = f(a, b) + \varepsilon(P) = 1 + 1 = 2$$

$$\hat{f}(b, t) = f(b, t) + \varepsilon(P) = 4 + 1 = 5$$

The flow $\hat{f}$ is shown in Fig. 4.3.

THEOREM 4.2 (AUGMENTING PATH THEOREM): A flow f in a transport network N is maximum if and only if there is no f-augmenting path.

Proof

Necessity : Suppose there exists an f- augmenting path. Then the revised flow $\hat{f}$ based on P has a larger value than f. Therefore, flow f in N is not maximum.

Sufficiency : Suppose N contains no f-augmenting path. Let S be the set of vertices of N such that they are reachable from the source s by f-unsaturated paths. Since there is no f-augmenting path, $s \in S$ and $t \in \bar{S}$.

Now we show that $val(f)=c(S, \bar{S})$, f-unsaturated path from s to v. So, if (v, w) is not f-saturated then there will be an f-unsaturated path (with (v, w) as a forward edge) from s to w, implying that $w \in S$. This is a contradiction because $w \in \bar{S}$. Thus each edge (v, w) with $v \in S$ and $w \in \bar{S}$ is f-saturated.

Consider an edge (v, w) with $v \in \bar{S}$ and $w \in S$. Then the edge is directed from $\bar{S}$ to S. If $f(v, w)$ is not f-zero then as before, there will be an f-unsaturated path from s to v, implying that $v \in S$. This is not possible since $v \in \bar{S}$. Thus for every edge

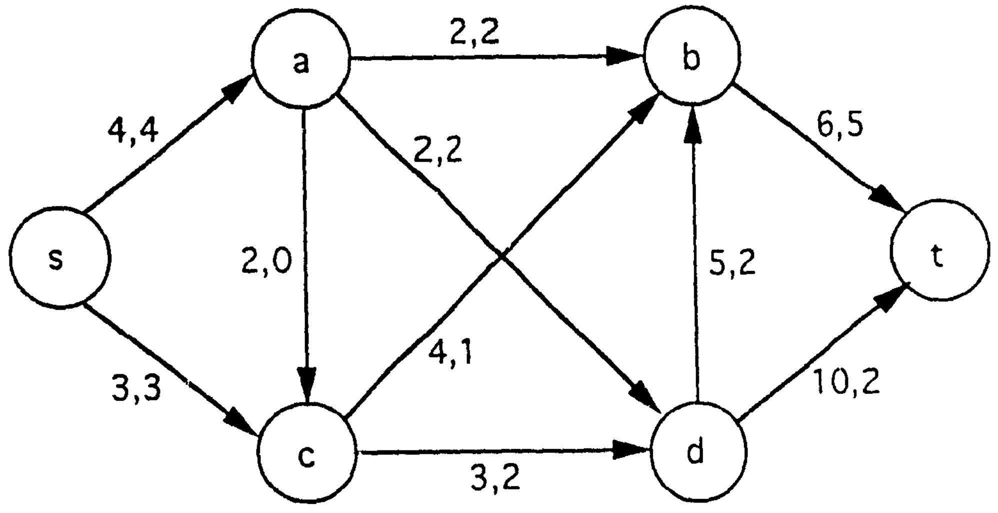


Fig. 4.3: Flow $\hat{f}$ in Transport Network of Fig. 4.1.

(v, w) with $v \in \bar{S}$ and $w \in S, f(v, w) = 0$.

Now,

$$val(f) = f(S, \bar{S}) - f(\bar{S}, S).$$

Since the edges directed from S to $\bar{S}$ are f-saturated ($f(i, j) = c(i, j), i \in S, j \in \bar{S}$) and the edges directed from $\bar{S}$ to S are f-zero ($f(i, j) = 0, j \in \bar{S}, i \in S$),

$$f(S, \bar{S}) = c(S, \bar{S}) \text{ and } f(\bar{S}, S) = 0.$$

Therefore,

$$val(f) = f(S, \bar{S}) = c(S, \bar{S})$$

Thus by Corollary 4.1.2, f is a maximum flow and $<S, \bar{S}>$ is a minimum cut.

□

Finally we have the maximum flow minimum cut theorem due to Ford and Fulkerson [FoF59] and [FoF62] stated and proved next.

THEOREM 4.3 (MAX FLOW MIN CUT THEOREM): In a transport network the value of a maximum flow is equal to the capacity of a minimum cut.

Proof

Consider a transport network N. Assume the present flow is maximum. Then as proved in Theorem 4.2, there is a cut $<S, \bar{S}>$ such that

$$val(f) = c(S, \bar{S})$$

Then from Corollary 4.1.2, $<S, \bar{S}>$ is a minimum cut.

□

4.2 The Push-Relabel Preflow Algorithm: Goldberg and Tarjan

Consider a transport network N with source s and sink t. Recall that a function f:E

$\rightarrow$ R , where R is the set of real numbers, is a maximum flow if the following conditions are satisfied:

1. $0 \le f(i,j) \le c(i,j)$.

2. $\sum_j f(i,j) - \sum_j f(j,i) = 0, \quad \forall\ i \in V - \{s,t\}$.

3. There is no augmenting path from s to t under f.

Conditions (1) and (2) ensure that f is a flow. Since condition (3) is also satisfied, it follows from the augmentation path theorem that f is a maximum flow.

The earliest algorithm for the maximum flow problem was due to Ford and Fulkerson [FoF62]. This algorithm starts with a flow f_0 that satisfies conditions (1) and (2) and constructs a sequence of flows f_1 , f_2 , $\cdots$ such that each f_i also satisfies these two conditions and that $val(f_{i+1}) > val(f_i)$. The algorithm terminates with a flow f_k that satisfies condition (3). Then f_k is a maximum flow because under f_k there is no augmenting path from s to t.

Several variations of the Ford-Fulkerson algorithm were subsequently presented resulting in algorithms with better complexities. One such algorithm was due to Dinic [Din70] which when combined with the MPM algorithm [MPM78] for constructing a maximal flow in an acyclic network has a complexity of $O(n^3)$ for a network with n vertices.

Recently Goldberg and Tarjan [Gol87],[GoT88] presented an approach that is fundamentally different from that of Ford and Fulkerson. This algorithm starts with an assignment $f(i,j)$ that satisfies conditions (1) and (3) and terminates with a flow that satisfies condition (2). This algorithm is quite elegant and is amenable to a distributed implementation.

We now proceed to present Goldberg and Tarjan's max-flow algorithm.

Let N =(V , E) be a network with each edge assigned a non-negative real capacity. Without loss of generality assume that N has no multiple edges. If there is an edge from a vertex v to a vertex w, this edge is unique by the assumption and is denoted by $(v\ , w)$. A *pseudoflow* is a function f :E $\rightarrow$ R that satisfies the following constraints :

$f(v\ , w) \leq c(v\ , w)$, $\forall\ (v\ , w) \in$ E (capacity constraint)

$f(w\ , v) = -f(v\ , w)$, $\forall\ (v\ , w) \in$ E (antisymmetry constraint)

We let $c(w\ , v) = 0$ if $(v\ , w) \in E$. Given a pseudoflow f, the excess function e_f : V $\rightarrow$ R is defined by,

$$e_f(v) = \sum_{u \in V} f(u,v),$$

Thus $e_f(v)$ is the net flow into v. A vertex v has *excess* if e_f(v) is positive. This indicates that some amount of flow can be pushed out from vertex v. A vertex v has *deficit* if $e_f(v)$ is negative.

Given a pseudoflow f, the *residual capacity* function c_f : E $\rightarrow$ R is defined by $c_f(v\ , w) = c(v\ , w) - f(v\ , w)$. The *residual graph* with respect to a pseudoflow f is given by G_f = (V, E_f), where $E_f = \{(v\ , w) \in$ E $\mid c_f\ (v\ , w) > 0\}$. Edge $(v\ , w)$ is a *residual edge* if $c_f(v\ , w) > 0$.

A *preflow* f is a pseudoflow f such that the $e_f(v) \geq 0$ for all vertices v other than s and t.

The push-relabel preflow algorithm of Goldberg and Tarjan starts with a preflow and a distance labeling, and uses two operations, *pushing* and *relabeling*, to update the preflow and the labeling, repeating them until a maximum flow is found.

For a given preflow f, a *valid distance labeling* is a function d from the vertices to the non-negative integers such that $d(s)=n$, $d(t)=0$ and $d(v) \leq d(w)+1$ for all residual edges (v,w). The intuition behind this is as follows.

$d(v)$ provides an estimate of the maximum distance from v to t. This is obvious for s and t because $d(s)=n$ and $d(t)=0$. Now consider a vertex v adjacent to vertices w_1 , w_2 , . . . , w_k. Let $d(w_i)$,$i = 1, 2, \ldots, k$ be the estimated distance from w_i to t. From among these vertices choose the one with the smallest distance. Let w_j be such a vertex. Then the estimated distance $d(v)$ will be at most $d(w_j) + 1$. So, during the relabel operation the minimum of $d(w)$, $(v, w) \in E_f$ will be used to update $d(v)$.

A vertex v is said to be *active* if $v \notin \{s, t\}$ and $e_f(v) > 0$. An edge (v,w) is *admissible* if $(v,w) \in E_f$ and $d(v)=d(w)+1$.

The push-relabel algorithm begins with an initialization phase. The flow on each edge leaving the source is set equal to the edge capacity, and all other edges not incident on the source have zero flow. Thus under this preflow, there is no augmenting path from s to t. For each vertex w, the excess $e_f(w)$ is calculated. It is clear that since some flow is pushed from the source, there exists at least one vertex with positive excess. So there exists at least one active vertex. Each vertex $w \in V - \{s\}$ is assigned an initial labeling $d(w)=0$. For vertex s, $d(s)=n$.

Then an update operation is selected and applied to an active vertex. This process continues until there are no more active vertices at which point the algorithm terminates, with a preflow f with no active vertices. Thus f satisfies conditions (1),(2) and (3) and is therefore a maximum flow. A maximum flow is thus found.

We next consider the update operations. The push operation modifies the preflow f and the relabel operation modifies the valid distance labeling d.

Consider a vertex v. The push operation is applicable if the following two conditions are satisfied :

1. *v is active: vertex v has positive excess flow* . This implies that some amount of flow can still be pushed out from v.

2. *There exists a residual edge, say (v,w), that is admissible: $(v,w) \in E_f$ and $d(v)=d(w)+1$*. Then path v to t via (v,w) is an estimated shortest path.

If these two conditions are satisfied, then a push along (v, w) can occur. A push from v to w increases the flow $f(v,w)$ and $e_f(w)$ by up to $\delta = \min\{e_f(v), c_f(v,w)\}$ and decreases $e_f(v)$ by the same amount.

After each such push, the new flows on edges and the residual graph are modified.

The relabel operation at vertex v is performed if the following conditions are satisfied:

1. Vertex v is active.

2. $d(v) \le d(w)$, for every residual edge (v, w).

Once these two conditions are satisfied, the updating of vertex v begins. Consider all outgoing edges at v in the residual graph. Then consider the labeling of all the terminal vertices of these outgoing edges. Pick the vertex with the minimum distance labeling. Let this vertex be w. Update the distance of v as $d(v) = d(w)+1$. $d(v)$ then gives the estimated shortest path from v to t. Thus the relabeling of v sets the label of v to the largest value allowed by the valid labeling constraints.

The push and relabel operations and the generic maximum flow algorithm are formally stated below.

Push(v,w).

Applicability

v is active, $c_f(v,w) > 0$ **and** $d(v) = d(w)+1$.

Action

Send $\delta = \min(e_f(v), c_f(v,w))$ units of flow from v to w;

$f(v,w) \leftarrow f(v,w) + \delta$; $f(w,v) \leftarrow f(w,v) - \delta$;

$e_f(v) \leftarrow e_f(v) - \delta$; $e_f(w) \leftarrow e_f(w) + \delta$;

Relabel(v).

Applicability

v is active **and** $\forall\, w \in V$, $c_f(v, w) > 0 \implies d(v) \leq d(w)$.

Action

$$d(v) \leftarrow \min_{(v,w) \in E_f} \{d(w)+1\}.$$

(If this minimum is over an empty set, $d(v) \leftarrow \infty$).

Algorithm 4.1 : GENERIC PUSH-RELABEL MAX-FLOW ALGORITHM (GOLDBERG AND TARJAN)

```
S1. (Initialization phase) :
    ∀ $(v,w) \in E$ set preflow $f$ as follows:
        $f(s, w) = c(s, w)$  for $w \in V$;
        $f(v, s) = -c(s, v)$ (by antisymmetry);
        $f(v, w) = 0$,  $\forall\ (v, w) \in E, v, w \neq s$;
    ∀ $w \in V$ set
        $e_f(w) = \sum_v f(v, w)$ ;  (compute the excess flow)
        $d(s) = n$ , $d(t) = 0$ , $d(w) = 0$ , $\forall\ w \in V - \{s\}$ ;
        (initialize the distance)
S2. While ∃ a basic operation that applies do
        select a basic operation and apply it;
    end.
```

We shall now explain an efficient implementation of the generic maximum flow algorithm. We shall start with a simple implementation and then refine it to improve efficiency. We need some data structures to represent the network and the preflow.

An unordered pair $\{v, w\}$ such that $(v, w) \in E$ or $(w, v) \in E$ is an *undirected edge* of G. Each undirected edge $\{v, w\}$ is associated with three values $c(v, w)$, $c(w, v)$ and $f(v, w)(=-f(w, v))$. Each vertex v has a list of the incident edges $\{v, w\}$, in fixed but arbitrary order. Thus each edge $\{v, w\}$ appears in exactly two lists, the one for v and the one for w. Each vertex v has a *current edge* $\{v, w\}$ which is the current candidate for a pushing operation from v. The refined max-flow algorithm repeats the *push/relabel* operation given below until there are no more active vertices.

```
Push/Relabel (v).
Applicability
    v is active.
Action
    Let {v, w} be the current edge of v.
    If push(v, w) is applicable then push (v, w)
    else
        If {v, w} is not the last edge on the edge list of v then
        replace {v, w} as the current edge of v the next edge on the edge list of v;
        else begin
            make the first edge on the edge list of v the current edge;
            relabel(v);
        end.
```

The *push/relabel* operation combines the basic push and relabel operations.

Initially, the current edge of v is the first edge on the edge list of v. When applied to an active vertex v the *push/relabel* operation tries to push excess along the current edge (v, w) if a pushing operation is applicable to this edge. If not, the operation replaces $\{v, w\}$ as the current edge of v by the next edge on the edge list of v; or if $\{v, w\}$ is the last edge on this list, it makes the first edge on the list the current one and relabels v.

The refined algorithm needs one additional data structure, a set Q containing all active vertices. Initially $Q=\{w \in V - \{s,t\} \mid c(s,w) > 0\}$. Maintaining Q takes only O(1) time per *push/relabel* operation.

By processing vertices in a more restricted order, we obtain improved performance. One such algorithm is the *first-in*, *first-out* (FIFO) algorithm. The algorithm maintains the set of active vertices Q as a queue. The FIFO algorithm consists of applying the *discharge* operation (described below) until Q is empty. The discharge operation terminates when the excess at vertex v is reduced to zero or v is relabeled.

```
Discharge.
Applicability
    Q ≠ 0.
Action
    Remove the vertex v on the front of Q.
    (Vertex v must be active)
    Repeat
        push/relabel(v);
        If w becomes active during the push/relabel operation then
            add w to the rear of Q;
    until $e_f(v) = 0$ or $d(v)$ increases.
If $v$ is still active then add $v$ to the rear of Q.
```

In the analysis of the FIFO algorithm, the concept of a *pass* is used. Pass one consists of the discharging operations applied to the vertices added to the queue during the initialization. Given that pass i is defined, pass $i+1$ consists of the discharging operations applied to vertices in the queue that were added during pass i. A key result is that the FIFO algorithm needs at most $4n^2$ passes. This results in complexity $O(n^3)$ for the Goldberg and Tarjan max-flow algorithm.

4.3 A Distributed Protocol for the Maximum Flow Problem

We now present a distributed protocol for the maximum flow problem. This protocol is a distributed version of Goldberg and Tarjan's push-relabel preflow algorithm discussed in the previous section. We first present the synchronous protocol. As usual, each vertex in the network represents a processor with a certain amount of memory. The vertices have access only to local information that is, capacities and flows on incident edges. Therefore, communication between vertices is by exchange of messages over the edges. Since messages can be received and transmitted by a vertex along an edge, each edge in the network is required to be bidirectional.

The algorithm proceeds in pulses. All the vertices perform the algorithm in parallel. During each pulse, the active vertices go through the following four stages.

In the first stage, all the flows received by the vertex from its neighbours are added to the excess and the edge flows are also updated. Pushing of flow is done during the second stage and if necessary relabelling of vertices in the third stage. In the final stage, the current distance label is broadcast to all its neighbours.

The algorithm terminates when there are no more active vertices. But in a distributed implementation, termination detection is difficult because a vertex which becomes inactive at a particular pulse may become active again at a later pulse. We now present a simple scheme for termination detection. This is based on the following theorem.

THEOREM 4.4 : If at any pulse, the total flow out of the source is equal to the total flow into the sink, then at that pulse and all subsequent pulses there will be no active vertices.

Proof

Consider the following flow equations at all the vertices.

$$\sum_{v} f(s, v) = \Delta_1 \tag{4.1}$$

$$\sum_{v} f(u, v) - \sum_{v} f(v, u) = -e_f(u), \quad u \neq s, t \tag{4.2}$$

$$-\sum_{u} f(u, t) = -\Delta_2 \tag{4.3}$$

In the above equations, term $f(u,v)$ appears exactly once as $f(u,v)$ and exactly once as $-f(u,v)$. So, summing the R.H.S. and L.H.S. of the above equations, we get

$$0 = \Delta_1 - \Delta_2 - \sum_{u} e_f(u)$$

If flow out of the source is equal to flow into the sink, that is $\Delta_1 = \Delta_2$, then we get

$$\sum_{u} e_f(u) = 0$$

But $e_f(u) \geq 0$.

So $e_f(u) = 0, \quad \forall u \neq s, t$

In other words, all vertices are inactive at the pulse when $\Delta_1 = \Delta_2$. This will be true at all subsequent pulses also, because no update operations are performed once all the vertices become inactive.

□

The above termination detection scheme can be implemented as follows. First, we select a path P from the source to the sink. This is done initially at the start of the algorithm using any distributed Breadth-First-Search Algorithm. At each pulse, the sink will send a message to its neighbour in the path P giving information about the flow into itself. In turn, each vertex in P conveys this information to its neighbours. When the source receives this information, it will check to see if the flow out of itself is equal to the flow into the sink. If so, it conveys to all other vertices that termination has occurred. This can be done through any broadcast scheme. Thus the source will detect termination in no later than n pulses after it has occurred because P has at most n-1 edges, and all the vertices will be informed about termination in no later than 2n pulses after it has occurred.

One can show that the distributed protocol performs $O(n^2)$ pulses and its message complexity is $O(n^3)$. The proof will follow along the same lines as that for the sequential time complexity [Gol87].

To implement the synchronous protocol on an asynchronous network, a synchronizer has to be incorporated. Consider first the $\alpha-$ synchronizer. It requires $O(m)$ messages per pulse. This means that using the $\alpha-$ synchronizer would result in a message complexity of $O(mn^2)$ for the asynchronous protocol. On the other hand, using the *beta* – synchronizer will not increase the message complexity because it requires only $O(n)$ messages per pulse. Issues relating to the choice of the synchronizer, when implemented in a shared-memory model, will be discussed in Chapter VI.

We now present the essential features of the synchronous protocol outlined above.

The protocol uses four types of messages : TFLOW, TERMINATE, δ and d-messages. A TFLOW message carries the value of the flow into the sink at an earlier pulse. When the source observes that the flow out of it is equal to the flow into the sink, it broadcasts a TERMINATE message to all its neighbours. These nodes will then broadcast TERMINATE message to all other nodes and so on. A δ-message from node i to node j carries the value of $\delta(i,j)$, which is the amount by which the flow on the edge (i,j) is to be increased. A d-message from node i carries the current value of node i's distance label.

Each node i is associated with an edge list. An edge (i,j) in this list represents the undirected edge connecting nodes i and j. $c(i,j)$ gives the capacity of this edge if, in the network, it is directed from node i to node j; otherwise $c(i,j) = 0$. We assume that each node is aware of the capacities of all incident nodes. Finally, we assume that a path P connecting the source and sink has been selected before activating the distributed protocol. Each node in the path is aware of its predecessor node as one traverses P from source to the sink. The variables $f(v,w)$, $c_f(v,w)$ and $e_f(v)$ denote the flow on edge (v,w), the residual capacity of (v,w) and the excess at node v, respectively.

Algorithm 4.2 : SYNCHRONOUS DISTRIBUTED MAX-FLOW PROTOCOL

1. (Initialization).

   During the first pulse at each node the following actions are performed.

   Node s

   (i) Set $d(s) = n$, $e_f(s) = 0$.

   (ii) For each edge (s,j) do:

   set

   $f(s,j) = c(s,j)$,

   $\delta(s,j) = c(s,j)$,

   $e_f(s) = e_f(s) + c(s,j)$.

(iii) Send $d(s)$ and $\delta(s, j)$ to node j.

Node $v \neq s$

(i) Set $d(v) = 0$.

(ii) For each edge (v, j)

set $f(v, j) = 0$, and

send $d(v)$ to node j.

(iii) Set the first edge in the adjacency list of v as current edge.

2. During each subsequent pulse, the actions to be performed at a node are :

Node s

Examine messages received from adjacent nodes and do :

a) If the message is a TFLOW message containing the value of flow into sink t, check to see if this value is equal to $e_f(s)$. If so, send TERMINATE messages to all adjacent nodes and HALT.

b) If the message is a δ-message received from node j, then set

$$f(s, j) = f(s, j) - \delta(j, s),$$
$$e_f(s) = e_f(s) - \delta(j, s).$$

Node $v \neq s$

S1: Examine messages received from adjacent nodes and do :

a) If the message is a TFLOW message, send this message to the predecessor of v in the path P.

b) If the message is a d-message from node j, update the value of $d(j)$.

c) If the message is a TERMINATE message, send TERMINATE messages to all adjacent nodes and HALT.

d) If the message is a δ-message received from node j, then set

$$f(v,j) = f(v,j) - \delta(j,v),$$
$$e_f(v) = e_f(v) + \delta(j,v).$$

S2: If $e_f(v) > 0$, then repeat the following until $e_f(v) = 0$ or $d(v)$ increases:

(i) Let (v, w) be the current edge in the edge list of v. Set $c_f(v,w) = c(v,w) - f(v,w)$. If $c_f(v,w) > 0$ and $d(v) = d(w) + 1$ then set

$$\delta(v,w) = \min\left\{ c_f(v,w) \quad , e_f(v) \right\},$$

$$f(v,w) = f(v,w) + \delta(v,w),$$
$$e_f(v) = e_f(v) - \delta(v,w).$$

Send $\delta(v,w)$ to node w.

(ii) If (v, w) is not the last edge in the edge list of v, then make the next edge on the list, the current edge.

(iii) If (v, w) is the last edge in the edge list of v, then make the first edge on the edge list of v, the current edge and relabel v as follows:

$$d(v) = \min\left\{ d(w) + 1 \mid c_f(v,w) > 0 \right\}.$$

S3: Send $d(v)$ to all adjacent nodes.

4.4 Summary

In this chapter, we first presented a review of certain basic results in network flow theory. We also presented, in detail, the push-relabel preflow algorithm of Goldberg and Tarjan [GoT88]. We then showed how to design a synchronous distributed maximum flow protocol starting from the push-relabel preflow algorithm. Issues relating to synchronization that will be required while implementing the protocol in an asynchronous environment will be discussed in Chapter VI.

CHAPTER V

THE TRANSSHIPMENT PROBLEM: PRIMAL-DUAL APPROACH

In this chapter, we discuss the transshipment problem which is also known as the minimum cost flow problem. We present the details of the primal-dual approach for solving this problem. This approach uses the shortest path and the maximum flow algorithms of the previous chapters as building blocks and involves repeated applications of these two algorithms.

5.1 The Transshipment Problem

Consider a network N with the underlying graph G = (V,E). Some of the vertices in N represent sources and are called *supply* vertices. Some of the others represent demand centers called *sinks*. There may be vertices which are neither supply nor demand. These are called *neutral* vertices. The supply or demand at a vertex v_i is denoted by b_i. For a neutral vertex, $b_i=0$. Each edge (v_i, v_j) is associated with a cost w_{ij}, which represents the cost of transporting a commodity along the edge. Each edge (v_i, v_j) is also associated with a capacity $c(i, j)$ representing the maximum amount of the commodity that the edge can accomodate. Given the supplies available at the sources and the demands at the sinks, the *transshipment problem* is to arrive at a routing pattern for a given commodity so that the demands are satisfied at minimum cost.

The transshipment problem is a linear programming problem and can be formulated as follows :

minimize : WX

subject to :

$$A X = b \tag{5.1}$$

$$0 \leq X \leq C \tag{5.2}$$

where

W= Row vector of edge costs w_{ij}

X= Column vector of edge flows x_{ij}.

A= Incidence matrix of the graph G underlying network N.

C= Row vector of edge capacities $c(i, j)$.

The incidence matrix $A = [a_{ij}]$ has one row for each vertex in G and one column for each edge in G. The elements of A are given by :

$$a_{ij} = \begin{cases} 1 & , \text{ if } \textit{edge } (v_i, v_j) \textit{ is incident into vertex } v_i. \\ -1 & , \text{ if } \textit{edge } (v_i, v_j) \textit{ is incident out of vertex } v_i. \\ 0 & , \textit{ otherwise.} \end{cases}$$

With a_{ij}'s defined as above, we get from (5.1),

b_i = sum of the flows into vertex v_i − sum of the flows out of vertex i.

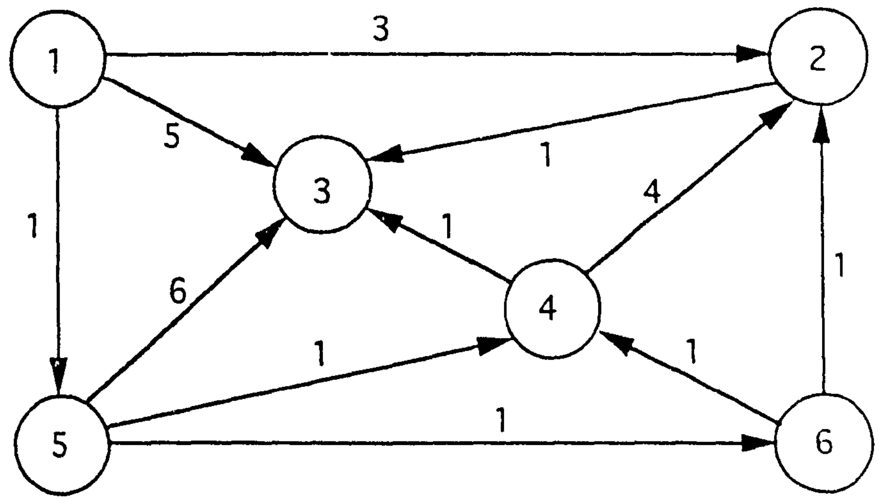
From this we can see that b_i is negative, if v_i is a supply vertex, and b_i is positive, if v_i is a demand vertex. We assume without loss of generality that the total supply available is equal to the total demand. Thus,

$$\sum_i b_i = 0.$$

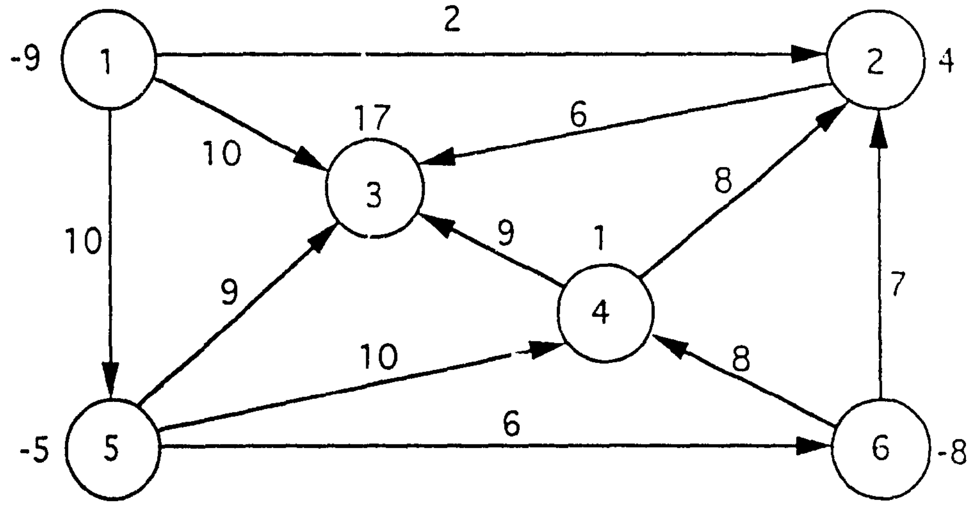
See Fig. 5.1 for an example.

Associated with any linear programming problem there is a *dual problem.* The original problem is then called the *primal problem.* The dual of the transshipment problem has n dual variables $y_1, y_2, \ldots, y_n$. The optimum values for y_i's would maximize the sum $\sum_i b_i y_i$. An important result in linear prramming theory is stated next.

If x_{ij}'s and y_i's represent optimum solutions for the primal and dual problems, respectively, then



(a) Node Names and Edge Costs.



(b) Node Demands and Edge Capacities.

Fig. 5.1: A Network N for the Transshipment Problem.

$$x_{ij} = 0\ , \quad \text{if } y_i - y_j + w_{ij} > 0$$
$$x_{ij} = c(i\ ,j)\ , \quad \text{if } y_i - y_j + w_{ij} < 0. \tag{5.3}$$

The above conditions are called the *complementary slackness* conditions.

We can now say that for any optimum solution X for the transshipment problem the following are true :

(i) $A X = b$.

(ii) $0 \le X \le C$.

(iii) There exist y_i's such that (5.3) is true.

At this point, it will be useful to explain the significance of the complementary slackness conditions.

Consider an edge $(v_i\ ,v_j)$ of N with flow x_{ij}. In linear programming theory, the quantity $y_i - y_j + w_{ij}$ is called the *relative cost coefficient* of $(v_i\ ,v_j)$. This quantity has an important role to play. If the flows in all the edges except $(v_i\ ,v_j)$ are kept unchanged and x_{ij} is changed to $x_{ij} + \Delta_{ij}$, then the objective function $\sum_{i,j} w_{ij}\, x_{ij}$ will change by $(y_i - y_j + w_{ij})\, \Delta_{ij}$. Since no further decrease in the value of the objective function is possible once an optimum is reached, it means that at that point

$$x_{ij} = 0\ , \quad \text{if } y_i - y_j + w_{ij} > 0$$
$$x_{ij} = c_{ij}\ , \quad \text{if } y_i - y_j + w_{ij} < 0.$$

There are two distinct approaches to the transshipment problem – the primal and primal-dual approaches[Chv83]. The primal approach, called the *network simplex* method, starts with an X satisfying (5.1) and (5.2) and repeatedly updates X (without violating (5.1) and (5.2)) until (5.3) is satisfied. The update of X is achieved through what is called a *pivot operation* . Note that each new X leads to a value for the objective function WX that is not greater that the previous value. As a sequential algorithm, the primal approach is known to be a very efficient one. However, unfortunately, it is not suited for a distributed or parallel implementation. The bottleneck here is that the pivot operation is inherently sequential in nature.

The primal-dual approach starts with an X and Y satisfying (5.2) and (5.3). It then updates X and Y (without violating (5.2) and (5.3)) until X satisfies (5.1). This approach is quite amenable to distributed and parallel implementations, since it uses the maximum flow and shortest path algorithms of the previous chapters, as building blocks

In the following section we present the primal-dual approach.

5.2 Primal-Dual Approach

The primal-dual approach consists of three main steps :

(i) Initialization.

(ii) Updating Y.

(iii) Updating X.

5.2.1 Initialization of the Primal-Dual Method

In this step, a pair of vectors X and Y such that

(i) $0 \le x_{ij} \le c(i\ ,j)$

(ii) $$x_{ij} = \begin{cases} 0 \quad , & \textit{whenever } y_i - y_j + w_{ij} \ge 0 \\ c(i\ ,j), & \textit{whenever } y_i - y_j + w_{ij} < 0 \end{cases}$$

are to be selected. The following cases arise.

Case 1 : All w_{ij}'s are positive.

In this case, selecting all x_{ij}'s and all y_i's equal to zero will result in a required pair of X and Y.

Case 2 : $c(i, j) = \infty$ for all edges (v_i, v_j).

In this case, we need to find y_i's such that $y_i - y_j + w_{ij} \geq 0$ for all edges (v_i, v_j). As we have shown in Section 3.2, finding such y_i's (whenever they exist) reduces to the shortest path problem on a graph G' constructed from the graph G of the given network.

Note that if there exists in G a directed circuit of total negative cost, the shortest path problem is infeasible. In such a case, it can be shown that the solution to the transshipment problem is unbounded.

Case 3 : $c(i, j)$ finite for some edges (v_i, v_j).

In this case, we need to find y_i's such that for all edges (v_i, v_j) with $c(i, j) = \infty$, $y_i - y_j + w_{ij} \geq 0$. Such y_i's can be obtained by applying the shortest path algorithm after removing all those edges (v_i, v_j) with $c(i, j)$ finite from the graph G' (see case 2).

Note that the traditional approach to the initialization problem is to apply the primal-dual approach to a new network constructed from the given network.

Once y_i's have been formed, as described in all the three cases considered above, we can select x_{ij}'s as in (5.3).

5.2.2 Updating the Dual Vector Y

Given a pair of X and Y vectors satisfying (5.2) and (5.3), we now show how we can update X and Y without violating these conditions.

From X and b we first calculate the new demand vector b' as follows.

$$\begin{aligned} b'_i &= b_i - net\ flow\ into\ vertex\ i \\ &= b_i - \sum_j a_{ji} x_{ji} + \sum_j a_{ij} x_{ij} \end{aligned}$$

Each b'_i then gives the current supply available or the current demand at vertex v_i. We shall call a vertex v_i *wet, balanced* or *dry* if b'_i is negative, zero or positive, respectively.

Using b' and X we construct an auxiliary network N' from the original network as follows :

(i) N' has an edge (v_i, v_j) of cost $y_i - y_j + w_{ij}$ for each original edge (v_i, v_j) with $x_{ij} < c(i, j)$.

(ii) N' has an edge (v_j, v_i) of cost of $y_j - y_i - w_{ij}$ for each original edge (v_i, v_j) with $x_{ij} > 0$.

(iii) Add a new vertex s and new edges (s, v_i) with zero cost for every wet vertex v_i.

For example, for the network in Fig. 5.1 with flows as shown in Fig. 5.2(a) and the following y_i's,

$$Y = [0\ 1\ 2\ 1\ 0\ 0]$$

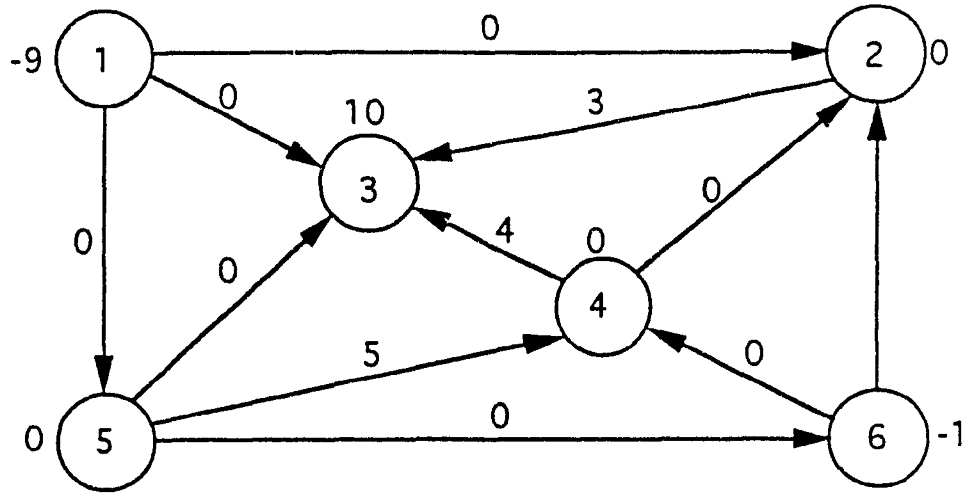
the corresponding auxiliary network with vertex demands and edge costs is shown in Fig. 5.2(b).

To update Y, we apply the single-source shortest path algorithm of Chapter III to N' and determine the shortest paths and distances from s to all the vertices in N', in particular, the demand nodes. The new N' is given by

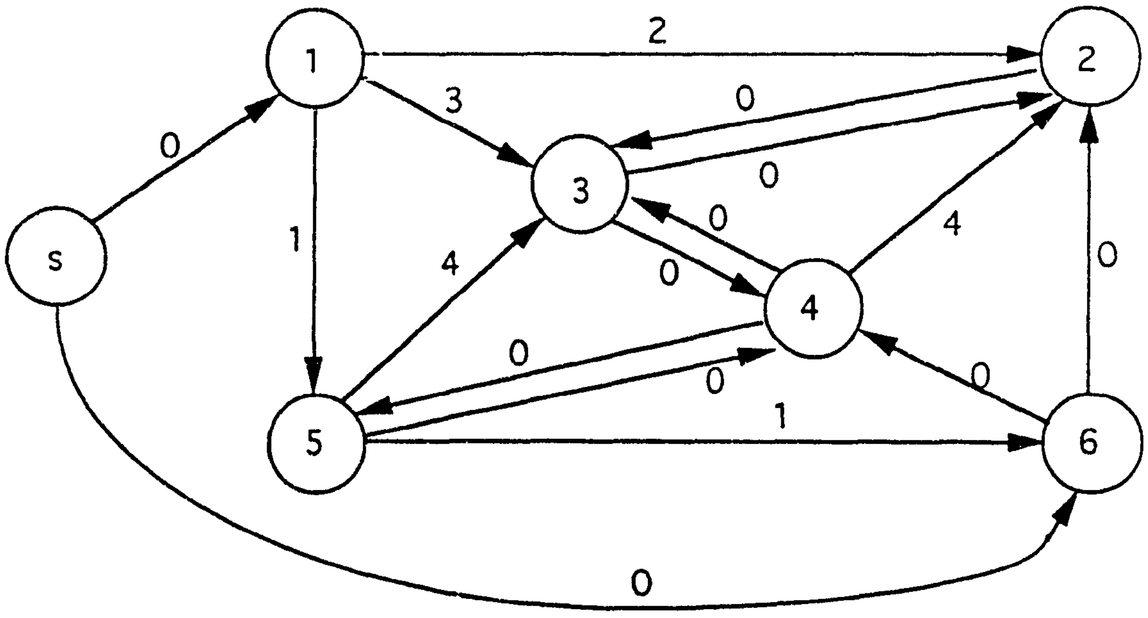
$$y'_i = y_i + d'_{s,i} \tag{5.4}$$

where $d'_{s,i}$ is the length of a shortest path from s to v_i in N'.

It can be shown that the new vector Y' and X satisfy condition (5.3). In other words, the update of Y has been achieved without violating the complementary slackness conditions.



(a) Network N of Fig. 5.1 with a Flow and New Demands.



(b) Auxiliary Network N' with Edge Costs.

Fig. 5.2: Construction of Auxiliary Network.

5.2.3 Updating the Flow Vector X

Given X and Y', we now show how X can be updated to a new X' such that both X' and Y' satisfy complementary slackness conditions.

Our aim really is to update X so that we make as much progress as possible towards satisfying (5.1), namely, the equation $A\ X = b$. This requires that we push as much flow as possible from the current wet nodes. But this should be accomplished without violating (5.3). Thus, we can modify the flows only on those edges (v_i, v_j) for which $y'_i - y'_j + w_{ij} = 0$. Interestingly, all the edges on a shortest path from s to vertex v_i satisfy this requirement. Thus while pushing the flows from the wet to the dry nodes, we should use only these edges.

This suggests the use of the following network N'' for modifying the X vector :

(i) N'' is a subnetwork of N'.

(ii) N'' has an edge (v_i, v_j) of capacity $c(i, j) - x_{ij}$ for each original edge (v_i, v_j) for which $y_i - y_j + w_{ij} = 0$ and $x_{ij} < c(i, j)$.

(iii) N'' has an edge (v_j, v_i) of capacity x_{ij} for each original edge (v_i, v_j) for which $y_i - y_j + w_{ij} = 0$ and $x_{ij} > 0$.

(iv) For each wet vertex v_i, N'' has an edge (s, v_i) with capacity $-b'_i$.

(v) N'' has a new vertex t and an edge (v_i, t) of capacity b_i for each dry node.

Thus pushing as much as possible from the wet nodes to the dry nodes reduces to pushing a maximum flow from s to t in N''.

As an example, the network N'' constructed from N' of Fig. 5.2 is shown in Fig. 5.3. Here the edge capacities are shown next to the edges.

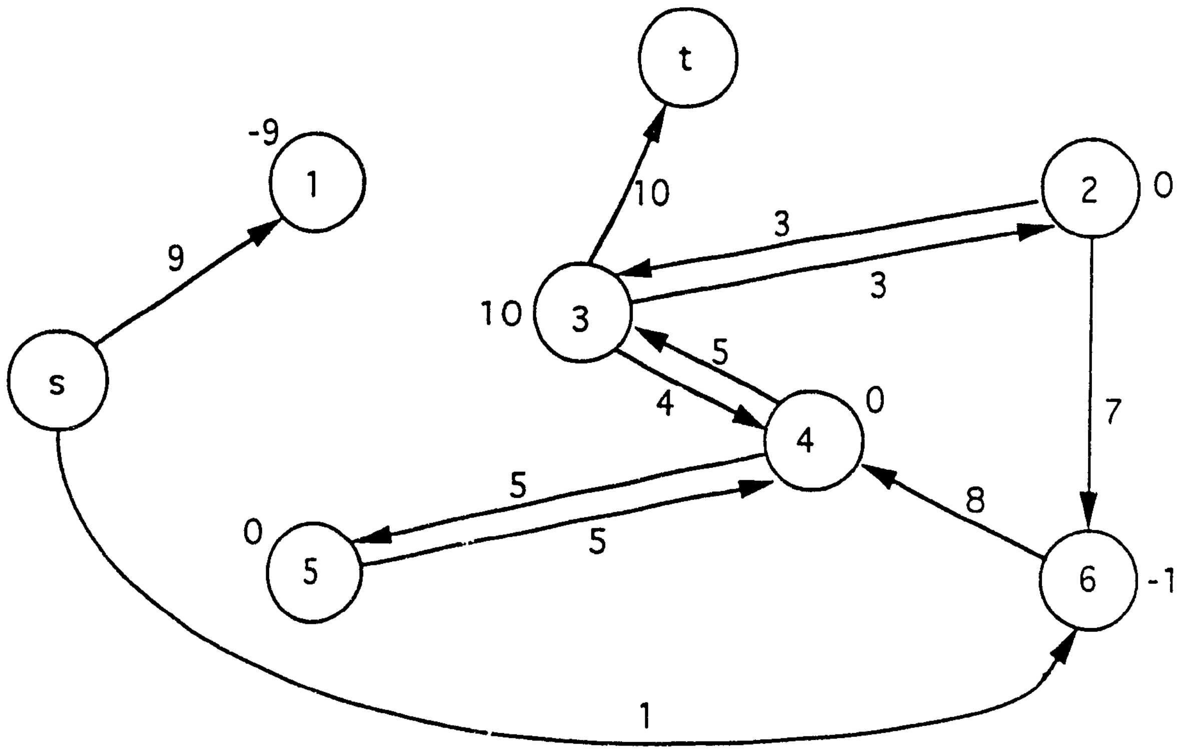


Fig. 5.3: Network N'' constructed from Network N' of Fig. 5.2 with given Flows and y values.

At the completion of the maximum flow algorithm, we update the flow vector X to a new vector X' as follows.

For each (v_i, v_j) in the original network, let x_{ij}'' and x_{ji}'' be the corresponding flows in N''. Then the new flow x_{ij}' of the vector X' is given by

$$x_{ij}' = x_{ij} + x_{ij}'' - x_{ji}''$$

Note that some of the edges in N'' may not lie on any path from s to t. These edges would not play any role while performing the maximum flow algorithm. So, for an efficient implementation, it will be worthwhile to remove these edges before applying the maximum flow algorithm.

The primal-dual approach discussed in this section in summarized in the flow chart of Fig. 5.4.

In the sequential case, like the network simplex method, the primal-dual is also known to be very efficient. Variants of the primal-dual method called ε-relaxation method are also available in [Gol87] [GoT88].

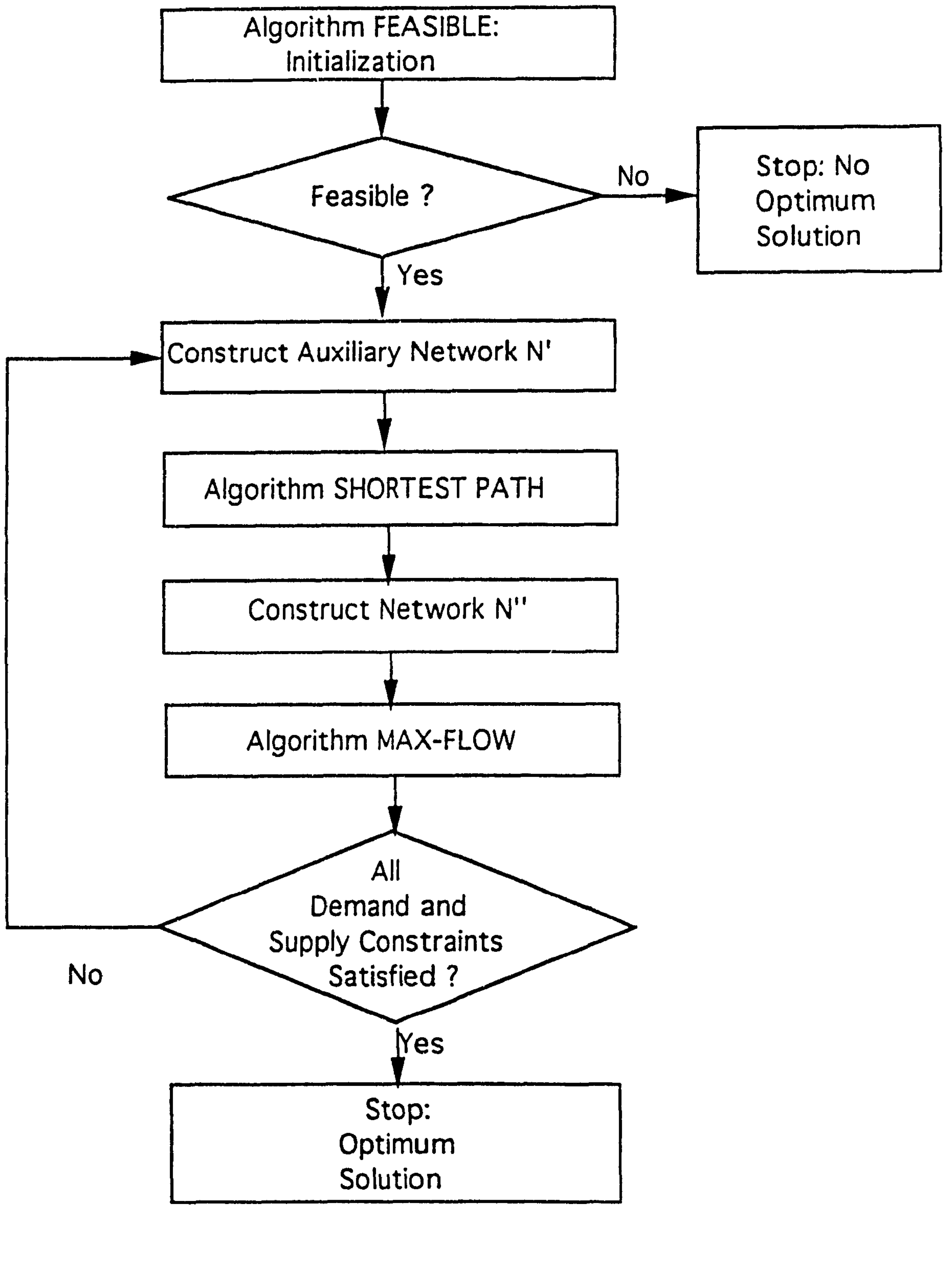

Algorithm FEASIBLE: Initialization
Feasible ?
No
Stop: No Optimum Solution
Yes
Construct Auxiliary Network N'
Algorithm SHORTEST PATH
Construct Network N''
Algorithm MAX-FLOW
All Demand and Supply Constraints Satisfied ?
No
Yes
Stop: Optimum Solution

Fig. 5.4: Primal-Dual Method.

5.3 Summary

In this chapter, we presented the transshipment problem and discussed the details of the primal-dual approach to this problem. The approach is elegant and can be easily implemented in a distributed manner, because it involves repeated applications of the max- flow and shortest path algorithms discussed in the previous chapters. In Chapter VI, we shall present the details of our distributed protocol for the primal-dual method.

CHAPTER VI

SIMULATION OF DISTRIBUTED PROTOCOLS ON A SHARED-MEMORY MULTIPROCESSOR

In this chapter we discuss the main features of our simulation (in a shared-memory environment) of our distributed protocols for the maximum flow and the shortest path problems and their integration into a shared-memory implementation of a distributed protocol for the primal-dual method. Detailed descriptions of these shared-memory protocols are given in the Appendices B, C and D. A brief description of those features of the BBN Butterfly shared-memory machine that we have used in our simulation work is given in Appendix A.

6.1 Shared-Memory Simulation of the Distributed Protocol for the Maximum-Flow Problem

We first briefly recall some aspects of Algorithm 4.2, a synchronous distributed protocol for the maximum flow problem. We then proceed to a discussion of its shared memory implementation.

Given a transport network with underlying graph $G = (V , E)$, in the distributed protocol each node is assigned to a processor with a certain amount of memory. In the following, the term node will also be used to refer to the corresponding processor. Each node has access only to local information, that is, capacities and flows on its incident edges. Therefore, communication between nodes is by exchange of messages over the edges.

The protocol proceeds in pulses. All the nodes perform the same algorithm in parallel. During each pulse, the active nodes go through the following four stages. In the first stage, all the flows received by a node from its neighbours are added to compute the excess. The edge flows are also updated. Pushing of flows is done during the second stage and, if necessary, relabelling of nodes is done in the third stage. In the final stage, the current distance labels are broadcast to all the neighbours of each node.

The *push/relabel* algorithm of Chapter IV is used for the pushing and relabelling operation of the max-flow algorithm.

The protocol terminates when there are no more active nodes.

In the shared-memory implementation, the four stages described above are performed by a node during three phases – read, compute, write. During the read phase, all the flows received by the node from its neighbours are added to calculate the current excess and the edge flows are also updated. It also reads the distance information of all its neighbours. This is required to correctly perform the push operation. During the compute phase of the pulse, the node computes the flows to be pushed along its incident edges, and relabels itself (if necessary). The amounts of flow pushed along the incident edges are then transmitted to the neighbours in the write phase. Finally, the new distance value is transmitted to the neighbours through write operations. Note that read and write operations essentially , lay the roles of send and receive operations in the message -passing model.

Each node i needs two adjacency lists pertaining to incoming and outgoing edges. Two data structures *a1(i)* and *a0(i)* are therefore declared, respectively. The node must also have access to the capacities of its outgoing edges. A data structure called *cap(i)* is defined for this purpose.

The degree of a node i is the number of nodes except s and t, adjacent to i.

The edges incident on a node, the capacities of the edges and the degree information are all input to the algorithm. Without loss of generality, we assume that at most two edges may be connected between any two nodes. For this reason, two data structures *delta0(i)* and *delta1(i)* are declared in the shared-memory for storing messages. These data structures store two types of information: the amount of flow(δ) pushed along an edge and the terminate information. *delta0(i)* receives messages pushed along its outgoing edges and *delta1(i)* receives information pushed along its incoming edges. Consider an edge (i, j) in graph G directed from i to j. Then (i, j) is an incoming edge at j. Suppose j pushes an amount δ of flow along (i, j). According to the antisymmetry rule, there exists a corresponding virtual edge (j, i). Node j then updates $f(j, i)$ and stores $-\delta$ in *delta0(i)(j)* so that node i can update $f(i, j)$ in the next pulse. Similarly, for an edge (j, i) directed from j to i, there exists a corresponding virtual edge (i, j). Node j then updates $f(j, i)$ and stores $-\delta$ in *delta1(i)(j)* so that node i can update $f(i, j)$ in the next pulse.

In the shared-memory model, care has to be taken to ensure write conflicts do not occur when processors try to access the same memory location. It is also possible for concurrent read and writes to occur. So, without proper synchronization, the coherence of the data may be destroyed.

During a pulse, a node reads information written in the previous pulse. In a distributed implementation, depending on the speed of the processors, a processor can read out-of-date information or read data to be received in the next pulse. This could result in an incorrect implementation of the synchronous protocol. In other words, the shared memory environment in which our protocol is expected to work is actually asynchronous. To overcome this problem, read and write operations have to be properly synchronized.

For the data structures *delta0(i)* and *delta1(i)* declared previously, two boolean data structures *check0(i)* and *check1(i)* are, respectively, defined for synchronizing the read and write operations.

For a node i to perform a read or a write operation, it first determines if it is safe to do so using the value in the entry, say (i, j), of the boolean data structure. FALSE in this entry indicates a read operation is possible. And TRUE indicates a write operation is possible. After every read and write operation, FALSE is set back to TRUE and TRUE is set back to FALSE, respectively.

In the initialization phase (pulse 0), the source performs the generic maximum flow algorithm, while all other processes only communicate dummy messages.

For computational purposes, a node i needs to be informed of all its neighbours' distances. A node i updates its distance information after it ensures that all its neighbours have read its old distance value. One technique is to allow the neighbours to decrement the degree of node i after reading node i's distance information. Eventually, the degree of node i will decrease to zero, indicating that it is safe for a node i to update the distance value. It is clear that the distance and degree variables permit concurrent read and concurrent write, respectively. So, again synchronization is necessary to avoid concurrent writes. Lock variables are used for this purpose. Since only one processor at a time can gain access to a lock variable, others must "busy wait" until the lock is released.

Note that in the definition of the degree of a node, the nodes s and t are excluded. This is a departure from the conventional definition of a degree. The reason for this is as follows. The source s and the sink t never modify their distance values. Also they do not need distances of their neighbours to perform any pulse. So they do not have to read the distance variables of their neighbours. Thus they are excluded while defining the degree.

Note that during a pulse, an active node performs read operations because it needs distance and new flow information to perform the *push/relabel* operation. It also has to perform write operations because it has to communicate the updated flow and distance values to its neighbours. In the case of an inactive node, which is not required to transmit any flow information to neighbours, dummy messages $\delta = 0$ have to be sent to neighbours. This is vitally important in a shared-memory asynchronous environment, for otherwise, the nodes may not know whether updated flow values are due from their neighbours. Interestingly, these dummy messages also help in the implicit implementation of the α-synchronizer.

6.1.1 Termination Detection

The termination of the maximum flow protocol can be detected using the scheme employed in Algorithm 4.2. In this scheme, a path P from source s to sink t is first determined. At each pulse the sink transmits the total flow it has received to its neighbour in P. In the next pulse, the neighbour transmits this information to its predecessor in P and so on. Thus at each pulse, each node in P will perform write and read operations to transmit and receive the sink flow information. Since these operations also use synchronization primitives, this scheme will cause considerable slowdown in the progress of the protocol. Also the complete termination of the protocol will occur only 2k pulses after the termination has occurred, where k is the length of P. In order to overcome these problems, we have used the following scheme to detect termination and propagate this information.

A virtual edge (t, s) is connected from the sink to the source. During each pulse, the sink calculates the total flow it has received and sends this message to the source. When this information reaches the source, the source determines the total flow out of it and compares it with the information received from the sink. If both are equal then the protocol terminates. The source then sends a TERMINATE message in the next pulse to all its neighbours including the sink.

When a process i receives a TERMINATE message, it propagates this information in the next pulse to all those neighbours from whom a TERMINATE message has not been received. In the subsequent pulse, it will read and write to those neighbours from whom a TERMINATE message has not been received. In the current pulse, it will read and write to all neighbours regardless of which neighbour sent a TERMINATE message. Receiving a TERMINATE message from a neighbour indicates that this neighbour is no longer active. Notice that sending a TERMINATE message is basically the same as writing this message in the globally-shared memory. A neighbour that has already terminated will never acknowledge this information by reading it and thus process i will wait endlessly. This would result in deadlock. This explains the purpose of sending the TERMINATE message to only those from whom this message has not been received.

There is still one more point to be mentioned. After propagation of the TERMINATE message, a processor performs one more pulse for synchronization purposes and then terminates.

Suppose process i received a TERMINATE message during pulse k. It then broadcasts this information to its neighbour j in pulse $k + 1$. Neighbour j will read this information only in pulse $k + 2$. Assume process i terminates in pulse $k + 1$ and performs no more pulse. Since process j will notify all its neighbours (from whom a TERMINATE message has not been received) only in pulse $k + 3$, it will wait in pulse $k + 2$ to perform a write operation to process i. Remember that in pulse $k + 2$, j will not write to i until i has read the information delivered to it in the previous pulse. But since process i already terminated in pulse $k + 1$, this will never occur, and process j will deadlock.

Therefore, every process has to go through one more pulse after broadcasting TERMINATE information. Then in the above example, process j will be able to write to process i in pulse $k + 2$. In pulse $k + 3$, process j will notify all its neighbours (i.e., read/write to only those that have not sent a TERMINATE message) to terminate. So, process i will not at all be considered in the read/write operations in pulse $k + 3$.

Notice that since the TERMINATE message travels in both directions (from sink to neighbours, and from source to neighbours), the rate at which this information is conveyed to all processors is faster than just the rate of traversal of the message in one direction, source to neighbours. Thus the TERMINATE information will arrive at all processors in at most $k/2$ pulses, where k is the diameter of the network.

6.1.2 Use of a Synchronizer

In a parallel computer the relative speeds of processors play a significant role. It is possible for a processor to be more than one pulse ahead of its neighbour. This could cause a processor to read out–of–date information or read information to be received in a later pulse. Thus, our implementation of the synchronous protocol Algorithm 4.2 should take into account the asynchronous nature of the environment. So, for a correct working version of our synchronous protocol it has to be augmented with a synchronizer.

We may recall (Section 2.3) that for an exact simulation of a synchronous protocol in an asynchronous environment, the following requirements have to be met:

1. Before a node completes its k^{th} pulse of activity, all the messages sent to it by its neighbours at the $(k-1)^{th}$ pulse are received and processed.

2. Before a node completes its k^{th} pulse of activity, no messages received from a neighbour executing its k^{th} pulse are processed.

We now discuss how our shared memory implementation meets these two requirements.

First, we note that when a node completes its pulse, it will have read and also processed all the information sent by its neighbours in the previous pulse. Thus requirement (1) is met. Also, the synchronization primitives used with read and write operations guarantee that all the information transmitted by a node i during a pulse will have reached its neighbours before i completes its present pulse. Thus the read and write operations provide considerable help in the synchronization process.

To meet requirement (2) we proceed as follows.

We associate with each node i a variable *safeset(i)*. At the beginning of a pulse, this variable is set equal to the degree of node i. At the end of the pulse, node i decrements *safeset(j)* for each neighbour j. Node i initiates its next pulse when *safeset(i)* becomes zero. In other words, node i initiates its $(k+1)^{th}$ pulse only after it has completed its k^{th} pulse and after it has become aware that all its neighbours have completed their k^{th} pulse activities. This guarantees that during the $(k+1)^{th}$ pulse, node i will process only messages sent during the k^{th} pulse. Thus our implementation using the variable *safeset* meets requirement (2).

Essentially, our implementation uses the α-synchronizer mechanism remedied as suggested in Section 2.4 (but without using ithe pulse number). Though we mentioned in Section 4.3 that the β-synchronizer is appropriate when the implementation is in a message-passing environment, in a shared-memory environment this synchronizer will require considerable overhead in terms of messages required for synchronization and will result in a slowdown of the progress towards termination.

6.2 Shared-Memory Simulation of the Distributed Shortest Path Protocol

In this section we present the main features of our shared-memory implementation of Algorithm 3.2, a synchronous distributed protocol for the shortest path problem. To make our presentation easier, we first explain Algorithm 3.2 using the data structures we have employed in our shared- memory implementation.

Let G = (V,E) be the graph of the network in which we have to find shortest paths from a specified node s to all the other nodes. Each edge (i, j) is associated with a length $w(i, j)$. Some of the lengths could be negative. But the graph is assumed to have no directed circuits of negative length. Let i be a node in G and let (i, j) be an edge directed from i to j. Then i is called a *dominator* of j and j is called a *successor* of i. A *predecessor* j of i is defined as the dominator node that caused the most recent update in the distance estimate of i.

All nodes in the distributed protocol (Algorithm 3.2) perform the same algorithm. They all receive distance information from their dominators and whenever a node discovers a path from s that is shorter than that currently known it updates its distance and sends appropriate length messages to all its successors so that they can update their distances, if necessary. If there exists no shorter distance from s than that currently known then the node sends no length messages to its successors. A node acknowledges each length message it receives by sending an acknowledgement message.

When a length message arrives at node i from a dominator causing no update in distance $d(i)$, then an acknowledgement is immediately transmitted to its dominator. Every node has a predecessor. Initially the predecessor of a node is itself. An acknowledgement will not be sent to the predecessor of a node if the node has not received all acknowledgement messages for all the length messages it transmitted to its successors. Suppose j is a predecessor of i. During the course of the algorithm, suppose a message arrives from k that causes an update in node i's distance value. Then an acknowledgement is sent back to the predecessor j and k becomes the current predecessor of i. Predecessor k will not be acknowledged until all length messages sent by i have been acknowledged by its successors or is replaced by another node as node i's predecessor. These are the different cases for which a length message arriving at a node will be acknowledged.

The protocol terminates when the source node has received all acknowledgement messages for all the length messages it sent to its neighbours.

The above protocol works correctly even in an asynchronous environment.

The synchronous version of the protocol also uses two messages: LENGTH and ACK. The protocol works in pulses.

Messages are transmitted only when a clock pulse is generated. So, during a pulse the processing of messages does not necessarily trigger the dispatch of further messages. But data structures are updated during the current pulse and information regarding messages to be transmitted in the next pulse is recorded.

If a LENGTH message arrives at a node i and causes an update of its distance estimate, appropriate LENGTH messages need to be transmitted in the next pulse. For a node to recognize that an update has occurred in the previous pulse, a boolean variable called *change(i)* is set to TRUE. After sending the appropriate messages, *change(i)* is set back to FALSE. In order to keep track of the identities of the predecessors in the previously known shortest paths for which ACK messages have to be sent, a data structure called *ackset(i)* is needed. And another variable *num*(i) is needed to keep track of the number of messages propagated by node i that are yet to be acknowledged. These data structures will be updated during a given pulse.

We now proceed to a discussion of our implementation of this protocol using a shared-memory model.

During a pulse a node goes through three stages. In the first stage, a node receives information from all its dominators. In the second stage, it updates its current distance, if necessary and in the third stage it propagates appropriate LENGTH messages if an update of its distance has occurred.

Recall that in the shared-memory model, read and write operations serve the role of receiving and transmitting messages, respectively. These operations have to be properly synchronized, as explained in the previous section.

Two data structures – *distinfo(i)* and *mesgterm(i)* – are globally declared to store messages. The *distinfo* data structure is used for storing messages received from the dominators and the *mesgterm* data structure is used for storing messages received from successors.

Suppose j is a dominator of i. *distinfo(i)(j)* consists of three types of information: distance estimate, terminate information and change value. The *mesgterm(i)(j)*, on the other hand, consists only of the terminate information since the successors do not send any distance value. Associated with these two data structures are two boolean data structures *checkdist* (i) and *checkterm(i)* for synchronizing the read and write operations.

Before any read operation, a node i checks *checkdist(i)(j)* to determine if it is set to FALSE. If so, it indicates that dominator j has already written to i in *distinfo(i)(j)*. This then allows i to proceed with reading the information delivered to it by j. After a read operation *checkdist(i)(j)* is set to TRUE to inform j that node i has acquired the message delivered to it. This allows j to write to i in *distinfo(i)(j)*. After each write operation *checkdist(i)(j)* is set back to FALSE. In a similar way, *mesgterm(i)* and *checkterm(i)* are handled.

For every LENGTH message transmitted to a successor, a corresponding acknowlegement has to be received. The number of acknowledgements yet to be received from the successors for all the LENGTH messages propagated to them is noted in a variable called *num(i)* declared globally. Remember that a node does not acknowledge it's predecessor until the node receives all acknowledgement from its successors. In our implementation *num(i)* = 0, indicates that acknowledgement messages have been received for all messages sent by i. *num(i)* is a concurrent-write and exclusive-read variable. It is possible for *num(i)* to assume a negative value if *num(i)* is not updated at the appropriate time. *num* (i) is incremented by the number of successors of i, every time LENGTH messages are propagated to the successors. This is done at the end of a pulse. However, before this is done, suppose a successor node decrements *num* (i) when it equals zero. Then *num(i)* will become negative. Such a situation is possible due to the difference in the speeds of the processors. To avoid any modification to occur in the variable *num(i)* before any update has been performed on it, some sort of synchronization has to be utilized. One scheme is to associate a one bit boolean variable to each node that is globally declared. Assume that after a node i modifies *num*(i) the boolean variable is set to FALSE. But the problem in this scheme is that node i may not be aware if all its neighbours have modified the data structure so that it can update or change the value in *num(i)* and reinitialize the boolean variable to FALSE. It might be the case that *num*(i) never gets decremented by node i's neighbours during a given pulse, causing the variables to remain TRUE without being reinitialized to FALSE.

In our implementation, we follow the scheme described next. A global variable called *pulsenum(i)* is used. This variable contains the current pulse number of node i. A node j does not decrement *num(i)* until the pulse numbers of i and j are equal. This would indicate that both nodes i and j are in the same pulse and therefore, the necessary updates to *num(i)* have taken place in the previous pulse.

In the shared-memory model, a message has to always to be exchanged between nodes during each pulse for proper synchronization. Because of this requirement one may encounter certain problems as explained next.

Note that a node's distance value may not change during a pulse. As was mentioned earlier, whenever a node i finds a path shorter than that currently known, it sends appropriate LENGTH messages to all its successors and increments *num(i)*. Suppose LENGTH messages are transmitted regardless of whether an update has occurred or not in the distance value. *num*(i) is still incremented by the number of successors of i. Since *num(i)* is a concurrent-write and exclusive-read variable, lock variables are necessary to access it. Assume an update has not occurred in the distance value of i. After the successors receive the information, they immediately send back acknowledgements to i since the information received is not new. Sending acknowledgement basically means accessing the lock variable to decrement *num(i)*. So, if LENGTH messages are sent even if there is no change in the distance value of a node, *num(i)* and the corresponding lock variables will have to be accessed several times during a pulse. But, accessing lock variables is a time-consuming operation. The only way to solve the problem is to send LENGTH messages only if an update has occurred in the node's distance value. This is implemented as follows.

We introduce a variable called *change(i)* . Whenever an update takes place *change(i)* is set to TRUE and *num(i)* is incremented by the number of successors. This change information is propagated to the successor along with the LENGTH information. When a node j receives this message it first checks *change(i)* . If *change(i)* is FALSE then the distance information of i is ignored. *change(i)* = FALSE indicates no update has taken place in the distance value *d(i)* since the previous pulse. Otherwise, node j considers the distance information of i and updates *d(j),* if necessary. This avoids the repeated and unnecessary updates of *num(i),* resulting from transmission of LENGTH messages even when no change has occurred in the distance value of a node.

Consider next the following situation. Suppose during a pulse p, *change(i)* = FALSE and at the end of the pulse *num(i)* = 0. If j is a predecessor of i, then at pulse $p+1$, j receives an acknowledgement from i. But, suppose at $p+1$, *change(i)* is still FALSE. This means that *num(i)* will remain zero at the end of this pulse, also. Then at pulse $p+2$, j would receive an acknowledgement message again. *change(i)* could continuously remain FALSE and the predecessor j could go on getting acknowledgements indefinitely. This is a very serious situation because this may cause the protocol to terminate prematurely! The problem here is that node i is not aware that its predecessor has already been sent a required acknowledgement. We next discuss how, in our implementation we prevent a situation such as this from occurring.

Recall that a node i sends an acknowledgement to j in either of three cases:

(i) If the LENGTH message received from j does not cause an update of $d(i)$.

(ii) If j is the predecessor and j is replaced by k as i's new predecessor,

(iii) If j is the predecessor and all acknowledgements from the successors for all the LENGTH messages propagated to them have been received.

In cases (i) and (ii), it is possible to send an acknowledgement immediately to the corresponding nodes. However, as we have seen just now, case (iii) can occur in a number of consecutive pulses resulting in more than one acknowledgement to be sent to j. This cannot be avoided using *num(i)* and *change(i)* alone.

Let us call a node *active* if it has a predecessor to whom an acknowledgement has not yet been sent. We use a new variable *active(i)* . If *active(i)* = TRUE, it means that the predecessor has not yet received an acknowledgement. Thus we send an acknowledgement to the predecessor if *active(i)* = TRUE and *num(i)* = 0. We then set *active(i)* = FALSE.

Suppose during pulse p, *change(i)* becomes TRUE and j becomes the predecessor of i. Then at this pulse *active* = TRUE. At pulse $p+1$, let *change(i)* be FALSE. Thus at $p+1$, *active(i)* continues to remain TRUE. Assume *num(i)* = 0 at the end of pulse $p+1$. Then during pulse $p+2$, predecessor j receives an acknowledgement and *active(i)* is set to FALSE. Thus an acknowledgement is sent back to the predecessor if *active(i)* = TRUE and *num(i)* = 0.

Suppose *num(i)* = 0 and *change(i)* = 0 for several pulses. Note that during these pulses *active(i)* continues to be FALSE and no additional acknowledgements will be sent to j. Let, during pulse $p+q$, $q \geq 2$, *change(i)* become TRUE and a new node k replace j as the predecessor of i. At this point, no acknowledgement will be sent to j because *active(i)* = FALSE. We make *active* (i) = TRUE after k becomes the predecessor of i. Now note that at the end of pulse $p+q$, *num*(i) becomes non-zero again.

Suppose at the end of pulse $p+r$, $r \geq q+1$, $num(i)$ becomes zero. Then at the beginning of the pulse $p+r+1$, an acknowledgement will be sent to the current predecessor of i because at this point, *active(i)* = TRUE.

We have used the above ideas in our shared-memory implementation of the shortest path protocol. In our implementation, *active(i)* is set to TRUE at the beginning of the pulse that follows the one where the new predecessor is selected. This is correctly done by using a variable *prevchange* (i) that indicates whether *change(i)* was TRUE or FALSE in the previous pulse and also a variable *received(i)* which is TRUE if *active(i)* is TRUE and *num(i)* = 0. After an acknowledgement is sent *received(i)* is set to TRUE, and *active(i)* is set to FALSE. *received(i)* is set to FALSE if the current predecessor is replaced by a new predecessor. This scheme helps to avoid the problem of the same predecessor receiving more than one acknowledgement during consecutive pulses.

The protocol terminates when the source node has received all acknowledgements from its neighbours. This happens if the neighbours of the source have received all the required acknowledgements and so on.

As in the case of the max-flow protocol, the source when it detects termination sends TERMINATE messages to all its neighbours. The nodes keep track of all its neighbours from whom TERMINATE messages have been received and are cautious not to send any message to those already terminated. Important features of our termination detection scheme may be found in Section 6.1.1. When a process receives a TERMINATE message and finds that *num(i)* $\neq$ 0, then it detects the presence of negative length circuits and terminates, signalling that the shortest path problem is infeasible.

The processes are synchronized using a modified form of $\alpha-$ synchronizer technique, as described in Section 6.1.2. This prevents processes from running several pulses ahead of the others.

6.3 Shared-Memory Simulation of the Distributed Protocol for the Primal-Dual Method

In this final part of the thesis, we integrate the distributed protocols for the max-flow and shortest path problems, described in the previous sections, and construct a distributed protocol for the primal-dual approach. Since the shortest path and max-flow protocols are, by themselves, very useful in several applications, we ensure, during our integration, that no modifications are done to these protocols.

$A0(i)$ and $A1(i)$ are, respectively, the adjacency lists containing information about outgoing and incoming edges at node i. $cost(i)$ and $C(i)$ are the data structures representing the cost and capacities of edges at node i. $Demprime(i)$ and $Y(i)$ denote the demand (or supply) and y-value (dual variable) at node i. Associated with the adjacency lists are the data structures $f0prime(i)$ and $f1prime(i)$ representing the flows in the edges represented in $A0(i)$ and $A1(i)$. All these data structures are declared globally.

Before the main body of the primal-dual protocol is initiated, the transshipment problem is tested for feasibility (see Section 3.2). If the problem is not feasible, the protocol terminates with no solution. If the problem passes the feasibility test, then the remaining phases of the protocol are executed.

In the first phase the y-values and edge flows are initialized to satisfy the complementary slackness condition (Section 5.2.1). The y-values are obtained as in Section 3.2. From the initial flows, the new (residual) demands and supplies at all nodes are calculated.

In the second phase, network N' is constructed as in Section 5.2.2. The adjacency lists $a0(i)$ and $a1(i)$ of N' are created. This phase is carried out sequentially by one processor, say, processor 0. The reason for this sequential computation is as follows. Suppose edges (i, j) and (k, j) directed into node j are to be included in N'. Then j

has to be included in the adjacency lists $a0(i)$ and $a0(k)$. This can be done in parallel. Also i and k have to be included in $a1(j)$. Thus both nodes i and k need to acces $a1(j)$. This could result in write conflicts. It is for this reason, the second phase is performed sequentially by one processor.

In the third phase.the shortest path protocol is invoked and applied on N'. The y-values are also updated using the distance values obtained at the completion of the shortest path protocol.

In the fourth phase, the network N'' is constructed as in Section 5.2.3. This is done in two subphases. In the first subphase, each node i checks an outgoing edge (i, j) of the original network. If condition (ii) or (iii) of Section 5.2.3 is satisfied then the appropriate entries are included in $a0(i)$ and $a1(i)$. This can be performed in parallel by all the processors. Once this has been done, in the next subphase for each entry j in $a0(i)$ and each entry k in $a1(i)$ (included in the first subphase), the node i should be included in $a1(j)$ and $a0(k)$, respectively. The adjacency lists $a0(s)$ and $a1(t)$ are also created in the second subphase. The reason for performing the fourth phase in two subphases is explained next.

Suppose there exists an original edge (i, j) and processor i created in N'' an edge (j, i). Then at processor j, $a0(j)$ would contain i. But (j, i) is not an original edge. If, while updating the edge flows, we examine only outgoing edges at each node then the edge (j, i) should not be considered. Thus we need to partition the lists $a0(i)$ and $a1(i)$ so that only those added during the first subphase are used while updating the flows.

To avoid write conflicts, as in the third phase, we perform the second subphase of the fourth phase sequentially.

In the fifth phase, the max-flow protocol is invoked. The processors update the flows in the sixth phase. New demands and supplies are also calculated. If not all the demands are equal to zero, phases 2 – 6 are repeated.

The protocol terminates when all the demands and supplies become equal to zero.

For this purpose a variable *dzero* is used. This is initially set to the number of nodes in the network. When *dzero* = 0, it signals the completion of the primal-dual protocol. At this point, all the processors terminate.

Note that all the phases discussed above should be done in seauence. Thus in addition to synchronization between pulses in a phase, synchronization is also required between phases. These synchronization issues are handled as explained in the first two sections of this chapter.

A description of the primal-dual distributed protocol may be found in Appendix D.

6.4 Performance Results

The protocols discussed in the previous sections have been implemented in the C language and tested on several networks. Since the BBN Butterfly machine available to us has only 20 processors, we tested the protocols on networks containing up to 20 nodes. Since our main interest is in the correct implementation of the distributed protocols, we counted the average number of messages per processor. We also counted the average number of steps performed by a processor.

We also designed sequential implementations of the max-flow and shortest path algorithms. In each case, we counted the total number of steps performed by the uniprocessor. The above results are summarized in the following tables, where n and m refer, to the number of nodes and the number of edges in the graphs tested, respectively.

n	m	Max.# steps per processor	Max.# messages per processor	# steps by uniprocessor
20	190	10278	7050	387839
15	105	5327	4007	176802
12	66	3392	2315	84624
10	45	2666	1241	38652
20	190	7931	4727	265480
15	105	3573	1183	73121
12	66	2390	796	35273
10	45	203	290	23134
20	190	367	620	186490
15	105	272	455	7844

Table 6.1: Results obtained for the Maximum Flow problem

n	m	Max.# steps per processor	Max.# messages per processor	# steps by uniprocessor
20	190	124	441	962
15	105	113	331	612
12	66	110	263	385
10	45	106	218	289
20	190	176	445	2049
15	105	159	331	1307
12	66	145	264	702
10	45	141	218	654
20	190	165	442	2132
15	105	159	331	1233

Table 6.2: Results obtained for the Shortest Path problem

As one would expect, the number of steps per processor is much smaller than the total number of steps performed by a uniprocessor.

A good portion of the messages broadcast by the processors are mainly for synchronization purposes. In a truly distributed environment, a number of these messages will not be necessary.

The protocols can be modified to run on a parallel machine using the message passing paradigm. They can also be incorporated in applications involving the network optimization problems.

6.5 Summary

In this chapter, we presented in detail several features of our simulation of the distributed protocols for the shortest path and max-flow protocols as well as their integration into a distributed protocol for the primal-dual approach to the transshipment problem. The simulation has been carried out on the BBN Butterfly machine, which employs the shared-memory model. Our simulations make an effective use of the synchronizer mechanisms in designing correct implementations of distributed protocols running in an asynchronous environment. Though synchronizers are proposed mainly in the context of synchronous distributed protocols running on asynchronous networks, our work and discussions underscore the importance and need for synchronizers in the implementation of parallel algorithms on commercially available multiprocessors.

CHAPTER VII

SUMMARY AND FURTHER RESEARCH

In this chapter, we give a summary of the work presented in the thesis and highlight some of our experiences. We also point to certain directions for future work.

7.1 Summary

Our goal has been to design and implement a distributed protocol for the transshipment problem (also known as the minimum cost flow problem). Two distinct approaches are available to solve this problem, namely, the network simplex method and the primal-dual method.

In the sequential case, both these approaches are known to lead to very efficient algorithms for solving the transshipment problem. So, to start with, we studied them and examined their suitability for distributed/parallel implementation. The network simplex method suffers from two shortcomings. First, the algorithm is inherently sequential in nature, moving (through pivots) from one basic tree solution to another. The pivot operation does not have much parallelism in itself. This is because during a pivot operation not all the nodes in a network would participate in the process. Added to this shortcoming, there is also the problem of determining, in a distributive way, an initial basic feasible solution. To our knowledge, no distributive algorithm is available for this process. This is because the traditional approach in linear programming is to use the simplex method itself to obtain an initial basic feasible solution. On the other hand, the primal-dual method has several attractive features, which make it an excellent candidate for distributed/parallel implementation. First, the application of this method involves repeated applications (in an iterative loop) of two simpler but well known algorithms, namely, the shortest path and maximum flow algorithms. In other words, the primal-dual method uses these two algorithms as building blocks. If we follow the traditional approach to initialization of the primal-dual method , it also suffers from the same problem as the network simplex method. But, as we have shown in the thesis, this initialization problem can also be solved using the shortest path algorithm. We have also examined variants of the primal-dual method such as the ε-relaxation method. Though they help in obtaining better theoretical complexity results, they are not easy to implement and are not known to be better than the primal-dual method even in the sequential case. These considerations led us to focus our work on the primal-dual method

Essentially, our work involved designing distributed protocols for the shortest path and the maximum flow problems and then integrating them into a distributed protocol for the primal-dual method. We also needed to incorporate synchronizer mechanisms for a correct working of these protocols in an asynchronous environment.

With the above in view, we first examined algorithms for the shortest path problem. Though Dijkstra's algorithm for this problem is known to be very efficient in the sequential case, it suffers from two shortcomings which make it unsuitable for use in our work. First, it is not applicable when there are negative length edges. Such edges could be present in a typical transshipment problem. Also, Dijkstra's algorithm passes through different phases with each phase requiring computations of the minimum of all the node labels generated during that phase. In a distributed or parallel implementation this could result in considerable slow down. On the other hand the Bellman-Ford-Moore algorithm does not suffer from these shortcomings. So we selected this for our work. We first presented in Chapter III a variant of this algorithm, which is elegant for distributed implementation. Our distributed protocol for the shortest path problem, given in Chapter III, is synchronous in nature and also incorporates mechanisms for termination detection and for detecting the presence of negative length circuits. For these purposes, we use Chandy and Misra's approach[ChM82]. We have also shown, in Chapter III, how the shortest path algorithm can be used to solve the primal-dual initialization problem. This became possible because of a result in [CoT88]. But for this, we would not have been able to achieve an elegant distributed protocol for the primal-dual method.

We then examined different approaches to the maximum flow problem with regard to their suitability for distributed /parallel implementation. We found that Dinic's algorithm [Din70] combined with the MPM algorithm [MPM78], though known to be very efficient, suffers from a very serious shortcoming. Like Dijkstra's algorithm it passes through different phases with each phase requiring the computation of the minimum of what are called the potentials of all the nodes. On the other hand, the recent algorithm due to Goldberg and Tarjan [Gol87], [GoT88] is quite attractive from the point-of-view of distributed implementation. However, no work on the distributed implementation of this algorithm is available. We have presented, in Chapter IV, a distributed version of the Goldber-Tarjan max-flow algorithm. We also incorporated in this protocol a mechanism for termination detection. This is based on a theorem that we proved in Chapter IV.

The shortest path and the max-flow protocols presented in Chapter III and IV are synchronous in nature. For them to work correctly in an asynchronous environment, we need to incorporate synchronizers into these protocols. For this reason, we examined in Chapter II the synchronizer design approaches available in the literature and the difficulties one may encounter in implementing them. We found that the α-synchronizer is appropriate for the shortest path protocol and β-synchronizer is appropriate for the max-flow protocol. We showed, in Chapter V, how the primal-dual protocol can be constructed using the protocols of the previous chapters.

We presented, in Chapter VII, our simulation of the different distributed protocols. Our simulation was carried out on the BBN Butterfly parallel computer, which employs the shared-memory model. Several problems that one does not encounter in a truly distributed (that is, message-passing) environment were encountered during our shared-memory implementations. Many of these problems relate to synchronization issues. For instance, the scheme used for termination detection for the shortest path protocol raised several problems while we implemented it in the shared-memory model. Without proper care to details, one might end up with a protocol that could terminate prematurely. Interesting questions also arose while integrating the max-flow and shortest path protocols to construct the primal-dual protocol. These different issues and our approaches are discussed in sufficient detail in Chapter VI.

We have presented, in the appendices, detailed descriptions of our distributed programs written in the C language. These programs can be modified to run on a message passing computer such as the hypercube. Appropriate optimization heuristics for mapping nodes onto the hypercube nodes would then result in efficient parallel programs for the transshipment problem. Such programs will be very valuable in solving practical problems. One such problem is the layout compaction and wire-length minimization problem. Interestingly in this application, the corresponding problem can be formulated as a transshipment problem [TCC90],[LoV90] and [Yos85]. The distributed/parallel protocols of this thesis can also be used to design protocols for certain important network optimization problems, because either they can be formulated as a shortest path or as a maximum flow problem or they can be formulated using these two problems as some subproblems. Two such optimization problems are: the maximum matching problem in bipartite graphs and the Chinese postman problem.

Summarizing, our contributions are as follows. It has been shown that the primal dual initialization problem can be solved using the shortest path protocol. A scheme has been devised to detect termination and properly broadcast the TERMINATE message to prevent deadlock of the protocol. While implementing the shortest path protocol we encountered that acknowledgements have to be sent appropriately at the appropriate time to prevent the protocol from the following: premature termination, endless looping or incorrectly signalling infeasibility. During the implementation of the maximum flow protocol we showed that though the β-synchronizer is appropriate for this problem in a message-passing environment, in a shared-memory environment this synchronizer will require considerable overhead in terms of messages required for synchronization and so will result in a slowdown of the progress towards termination. So, we used the α-synchronizer with modifications. We showed that *no old information read before new information written and no new information written before old information read* guarantees that the protocol do not suffer from the shortcomings of the α-synchronizer discussed in Chapter II. In this scheme pulse numbers are not necessary. Also, we showed that synchronization is necessary between pulses in a phase (creating N', shortest path, etc.) and between phases of the primal-dual protocol. This is achieved using the α-synchronizer mechanism. But, in this case pulse numbers (alternately 0 and 1) are necessary because *read-write* requirement is not there. And finally while constructing the network N'' the adjacency lists must be partitioned appropriately to avoid computing flows incorrectly. Parallel algorithms for several implementation applications can be designed through easy modifications to the distributed programs presented in the thesis.

7.2 Further Research

Our experiences during this work point to two possible directions for further research.

Almost all the parallel or distributed algorithms presented in the literature follow from the corresponding sequential algorithms. This approach necessarily restricts algorithms to be of fine grain nature – each node/ edge requiring a processor. Thus these algorithms assume the availability of a very large number of processors, though with limited processing capabilities. But fine grain parallel algorithms impose exclusive synchronization requirements, which could cause considerable slowdown and, hence, lower the speed-up which one would like to achieve. An interesting problem would be to design parallel graph algorithms which are of a coarse grain nature. No works of this nature for graph or network problems are available in the literature.

In designing asynchronous algorithms, we have proceeded in two steps: first designing a synchronous algorithm and then embedding a synchronizer. There are two reasons for this. First it is difficult to design and prove correctness of an asynchronous algorithm. Secondly, asynchronous algorithms may require an excessive number of message exchanges. But this would result only if the transmission delays are unpredictable. This may not be so in a practical situation. On the other hand, asynchronous algorithms may not require much synchronization. Therefore another problem for further research is the design of synchronous algorithms that run correctly even in an asynchronous environment. Note the synchronous shortest path protocol of Chapter III is in fact one such algorithm, which will work correctly even in an asynchronous network. However, the synchronous max-flow protocol of Chapter V does not have this property.

REFERENCES

[Akl89] Akl, S., *The Design and Analysis of Parallel Algorithms* , Prentice Hall, New Jersey, 1989.

[Awe85] Awerbuch,B., "Complexity of Network Synchronization", *J. ACM,* Vol.32, 804-823, 1985.

[Awr85] Awerbuch, B., "Reducing Complexities in the Distributed Max-Flow and Breadth-First-Search Algorithms by means of Network Synchronization", *Networks,* Vol.15, 425-437, 1985.

[BBN90] *Getting Started with the Mach 1000 Operating System,* BBN Advanced Computers Inc., Cambridge, Massachusetts, 1990.

[BeG87] Bertsekas, D., and R. Gallagher, *Data Networks* , Prentice Hall, Englewood Cliffs, New Hersey, 1987.

[Bel58] Bellman, R.E., " On a Routing Problem", *Quart. Appl. Math.,* , Vol.16, 87-90, 1958.

[BeT89] Bertsekas, D.P., and J. Tsitsiklis, *Parallel and Distributed Computations: Numerical Methods*. Prentice Hall, New Jersey. 1989.

[ChM82] Chandy, K.M., and J. Misra, " Distributed Computation on Graphs", *CACM* , Vol.25, 833-837, 1982.

[ChM89] Cheriyan and S.N. Maheswari, "Analysis of Preflow Push Algorithms for Maximum Network Flows", *SIAM J. Comp.,* , Vol.18, 939-954, 1989.

[Chv83] Chvatal, V., *Linear Programming* , Freeman Company, Potomac, Maryland., 1983.

[CoT88] Comeau, M., and K. Thulasiraman, "Structure of the Submarking Reachability Problem and Network Programming", *IEEE Trans. Circuits and Systems,* Vol.CAS-35, 89-100, 1988.

[Dij59] Dijkstra, E.W., " A Note on Two Problems in Connexion with Graphs", *Numerische Math.,* Vol.1, 269-271, 1959.

[Din70] Dinic, E.A., "Algorithm for the Solution of a Problem of Maximum Flow in a Network with Power Estimation", *Soviet Math., Dokl.,* Vol.11, 1277-1280, 1970.

[FoF56] Ford, L.R., and D.R. Fulkerson, "Maximal Flow through a Network", *Canadian J. Math.,* Vol.8, 399-404, 1956.

[FoF62] Ford, L.R., and D.R. Fulkerson, *Flows in Networks,* Princeton University Press, Princeton, New Jersey, 1962.

[GHS83] Gallagher, P. Humblet and P.A. Spira, "A Distributed Algorithm for Minimum-Weight Spanning Trees", *ACM Trans. Programming Languages and Systems* , Vol.5, 66-77, 1983.

[Gol87] Goldberg, A.V., " Efficient Graph Algorithms for Sequential and Parallel Complexity", Ph.D., Thesis, Lab. for Comp. Science, M.I.T., Cambridge, Massachusetts, 1987.

[GoT88] Goldberg, A.V., and R.E. Tarjan, "A New Approach to the Maximum Flow Problem", *J. ACM,* Vol.35, 921-940, 1988.

[Hum83] Humblet, P.A., " A Distributed Algorithm for Minimum – Weight Directed Spanning Trees", *IEEE Trans. Communications,* COM-34, 345-347, 1983

[Kru56] Kruskal, J.B., "On the Shortest Spanning Subtree of a Graph and the Travelling Salesman Problem", *Proc. Am. Math. Soc.,* Vol.7, 48-50, 1956.

[LaT87] Lakshmanan, K.B., and K. Thulasiraman, " On the Use of Synchronizers for Asynchronous Communication Networks", *Proc. 25 th Allerton Conf. on Communication, Control and Computing,* Univeristy of Illinois, Urbana-

Champaign, 1987.

[LoV90] Lo, C.Y., and R. Varadarajan, " An $O(n^{1.5}\log n)$ 1-d compaction Algorithm", *Proc. 27th ACM/IEEE Design Automation Conf.,* 382-387, 1990.

[LTC89] Lakshmanan, K.B., K. Thulasiraman and M.A. Comeau , "An Efficient Distributed Protocol for the Shortest Path Problem in Networks with Negative Weights", *IEEE Trans. Software Engg.* , Vol. SE-15, 639-644, 1989.

[Lyn88] Lynch, N., "Distributed Algorithms", *Lecture Notes* , Lab for Comp. Science, M.I.T., Cambridge, Massachusetts, 1988.

[Mar90] Marple, D., "A Hierarchical Preserving Hierarchical Compactor", *Proc. ACM/IEEE Design Automation Conference,* 375-381, 1990.

[Moo57] Moore, E.F., "The Shortest Path through a Maze", *Proc. Intl. Symp. Theory of Switching, Part II,* Univ. Press, Cambridge, Massachusetts, 285-292, 1957.

[MPM78] Malhotra, V.M., M. Pramodh Kumar and S.N. Maheswari, " An $O(v^{3})$ Algorithm for Maximum Flows in Networks", *Information Proc. Letters,* Vol.7, 277-278, 1978.

[Mul89] Mullender, S., *Distributed Systems* , Addison-Wesley, Reading, Massachusetts, 1989.

[Pri57] Prim, R.C., "Shortest Connection Networks and Some Generalizations", *Bell Syst. Tech. J.,* Vol.36, 1389-1401, 1957.

[Roc84] Rockfeller, R.T., *Network Flows and Monotropic Optimization,* Wiley-Interscience, New York, 1984.

[Seg83] Segall, A., "Distributed Network Protocols", *IEEE Trans. Information Theory* , Vol.IT-29, 23-25, 1983.

[SeS91] Segall, A., and L. Shabtay, "Message Delaying Synchronizers", *Proc. Workshop on Data Structures and Algorithms,* Spain, 1991.

[SwT81] Swamy, M.N.S., and K. Thulasiraman, *Graphs, Networks and Algorithms,* Wiley-Interscience, New York, 1981.

[TCC90] Thulasiraman, K., M. Comeau, R.P. Chalasani, A.Das and J.W. Atwood, "On the Design of Parallel Algorithm for VLSI Compaction", *Proc. Intl. Symp. Circuits and Systems* , 1990.

[Yos85] Yoshimura, T., "A Graph-theoretic Compaction Algorithm", *Proc. Intl. Symp. Circuits and Systems,* 1445-1458, 1985.

APPENDIX A

THE BBN BUTTERFLY MULTIPROCESSOR

The BBN Butterfly multiprocessor uses the Mach 1000 operating system which is a Berkeley 4.3 BSD – compatible version of the UNIX operating system. It uses the GP1000 hardware. Either C or Fortran can be used to write programs to run on the GP1000 multiprocessor.

A.1 Programming with the Mach 1000 Operating System

This section describes clusters, how to use them and commands that can be used to manage them.

A *cluster* is a set of GP1000 nodes grouped together. These sets of nodes can be used to run a particular program. The programmer has the flexibility to divide the system into any number of clusters, each containing one or more nodes, depending on the number of nodes available in the system. Each cluster then becomes a separate computing resource with its own characteristics, specified according to the programming needs.

The cluster configuration can be changed to better suit the program needs. The size of the clusters can be varied. Also, nodes to be included in a cluster can be specified. Several other characteristics, such as who can access the cluster and how nodes of a cluster are selected can be defined.

The way the clusters are used depends on several factors:

(i) Number of nodes in the system.

(ii) Characteristics of programs run.

(iii) Number of users on the system.

(iv) Characteristics of programs currently run by other users.

The cluster mechanism has several commands which allow a programmer to:

(i) Put additional nodes into an existing cluster with *addnodes* .

(ii) Execute a command in a particular cluster with *runincluster* .

(iii) Obtain information about clusters with *whichcluster* and *clusters* .

(iv) Remove nodes with *freenodes* .

(v) Disband clusters altogether with *removecluster* .

(vi) Modify cluster attributes explicitly with *clusterctl* .

For more information on the use of cluster commands and other features of clusters, refer to [BBN90].

A.2 The Uniform System Approach

The Uniform System is a library of subroutines that can be used with C or Fortran programs. Here, we explain the system with reference to the C language.

There are two main considerations of the GP1000 parallel processor: storage management and processor management. The goal of storage management is to utilize the full memory bandwidth of the machine. All the memories in the machine are kept equally busy. This increases speed up, since processors are allowed to access different parts of the memory, reducing memory contention. Slow down and memory conflicts may occur if all processors attempt to access a single memory. The goal of processor management is to utilize the full processor bandwidth of the machine. The purpose is to keep all processors equally busy, thereby preventing overloading of some processors while others sit idle.

A.2.1 Memory Management

The GP1000 switch provides low delay and high bandwidth access to all the memory.

The Mach 1000 operating system provides *virtual memory*. It enables the processes to manage their own address spaces, thereby preventing memory contention.

The GP1000 hardware and Mach 1000 operating system are a foundation on which a variety of software structures are built.

The Uniform System allows the processes to share a single large block of virtual memory. The application program is implemented on this single large address space. This frees the application programmer from the need to manipulate memory space maps, thereby simplifying programming. The application data is scattered across all memories of the machine, thus reducing memory contention. Stacks and local variables of the processors are kept locally. So one processor does not have access to the other processors' local memories.

The collection of the memories of the GP1000 nodes form the shared memory of the machine (See Fig. A.1). In other words, each processor of the machine has a block of memory local to the processor. The combination of these memories is the shared memory. This means that the large shared memory the application program sees is implemented by a collection of separate memories.

If the data is located in the memory of just one processor then accessing the data causes memory conflicts, forcing one processor to wait until the other is finished if they are both trying to access the same memory location. Therefore, to avoid memory contention, the data is distributed across the different memories. By so doing, the full memory bandwidth of the machine is utilized.

As one would expect, there is a cost associated with the memory management

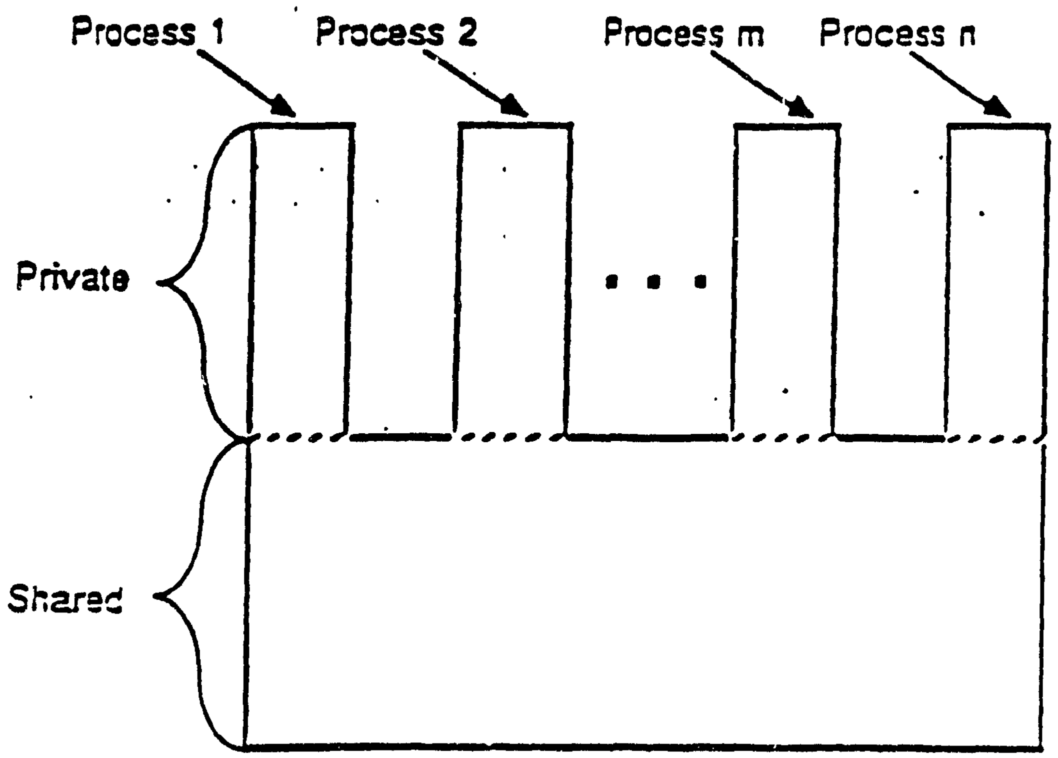


Fig. A.1. Processes Share Much of their Address Space.

strategy. The memories of the processors are connected by an interconnection network to form the shared memory. Therefore, a processor has to follow a path along this interconnection network to access the data which may be located in some other physical memory other than its own. So, there might be memory contention problems and slowdown in accessing the data. But the execution time is increased from 4% to 8% mainly due to less contention.

The purpose of the memory management strategy is to allow the programmer to treat all processors as identical workers, each processor having access to all the application data and, hence, being able to do any application task independently.

The Uniform System also provides atomic operations such as locks to avoid conflicts in certain operations (for instance, concurrent write operations at the same location) using the same memory locations.

A.2.2 Processor Management

The goal of processor management, as mentioned earlier, is to allow all processors to be busy, thereby using the full processor bandwidth. In other words, the goal is to minimize processor idle time. Without proper processor management, it may so happen that one processor sits idle after completion of its computation, while others are still working.

The GP1000 parallel processor uses two strategies, dynamic and static strategies, to minimize the idle time of processors. The *static approach* uses exactly as many concurrent tasks as there are processors. The programmer, in this method, has to apportion the work so that all processors finish at approximately the same time. The *dynamic approach* , on the other hand, uses many tasks per processor. Tasks are dynamically allocated to the processor. When a processor finishes with a task it is assigned the next task for execution. This method balances the load in the system. Though there might be some wait at the end of the program, it is generally small compared to the total program execution time. The advantage of the approach is that it is not necessary to know in advance how long an individual task takes to complete. It also adapts to the varying numbers of processors and sizes of problems.

The Uniform System encourages the dynamic approach but also supports the static approach.

Once the programmer determines what processing will occur in parallel, these tasks have to be scheduled for parallel execution. To use the Uniform System, the programmer must structure the application into two parts:

a) A set of subroutines that perform various application tasks ; and

b) One or more subroutines called *task generators* that identify the task for execution.

The task generators will be explained later.

A.3 Using the Uniform System (Us)

In this section, we explain some of the Us library routines used in the programs implemented in this thesis. For more details on various other routines, see [BBN90].

Include File

Any Us program must include the header file *us.h* at the beginning of the program:

```
#include <us.h>
```

Initializing the Us

The program (or process) is loaded into all the processors in the cluster. The routine

```
InitializeUs();
```

initializes the Uniform System. This routine starts and creates a Us process on every available processor in the cluster, sets up the memory that is globally shared among all Us processes in the cluster, and initializes the Us storage allocator. This routine should be called before any other Us routines except *SetUsConfig* , or *ConfigureUs*. It should be called only once in a program.

Obtaining Configuration Information

Sometimes it is necessary to refer to processors by numbers. There are two separate numbering schemes for processors, and routines for converting them.

In the firs' scheme, a process is assigned the number of the hardware processor on which it is running. The number assigned to a processor depends on the size of the GP1000 switch and the way the processor is connected to the switch. The hardward processor numbers used can range from 0 to 255.

In the second numbering scheme, a virtual number is assigned to each of the processors. The virtual processor numbers are consecutively numbered from 0 to P-1, where P is the number of processors available to the program. The Us uses this numbering scheme because it affords considerable flexibility to the programmer.

Routines are available to determine a processor's hardware or virtual number. And mapping routines for conversion between hardware and virtual processor number also exist. Routines are also available to get more information on the number of processors and memory available to a program.

For more information on different routines, see [BBN90].

Memory Classes

A process has access to two classes of memory (See Fig. A.2). Memories local to a

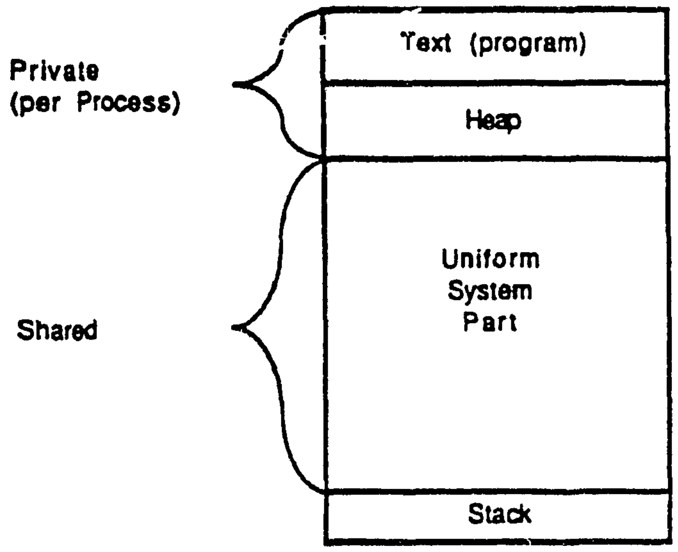


Fig. A.2. Address Space of a Uniform System Process.

process p are called *process private memory* . Only process p can access this memory. The other class of memory is the *globally-shared memory* . Data to be shared by two or more processors are located in this memory. All processors have access to this memory.

Within these two memory management classes, several different types of storage are available to C programs.

Local variables declared within a process p are local to this process. The process p can access these local variables and modify them. The changes made to these variables within the process are not seen by the other processes. The local variables are stored in stacks.

Within a process there may be several subroutines, and one subroutine might need access to some variables local to the process. By passing these variables to the subroutines by means of pointers, any changes made within the subroutine can be seen by the process after completion of the subroutine. So, variables can be shared by subroutine calls within the same process, but are hidden from all other processes.

Then there is the dynamic storage obtained by *malloc* and other related routines. They are local to the process. Variables created by these routines can be accessed by subroutines within the same process, but are hidden from all processes.

The last type of storage is data that are globally shared. Data can be allocated in the shared-memory by using routines of the Us. In particular, the *UsAlloc* routine allocates space for data to be shared globally. Pointers to these variables are valid on all processors and can be passed freely among them for communication purpose.

The Us allocator creates and manages the globally-shared memory region of the process address space. A program can ask the allocator for space within a particular processor node or for space that is scattered across the machine. Once space has been allocated for a program for sharing, the program is free to pass pointers to variables in the space from one process to another.

Memory Allocator

The Us provides a variety of memory allocators that allocate storage in globally-shared memory. The allocators return a pointer to the block of memory allocated. If an allocator is unable to obtain the requested amount of memory, it returns the null pointer (i.e., zero).

In this section, only allocators that have been used in the programs will be explained. For more information on other allocators, see [BBN90].

To allocate a block of storage in globally-shared memory,

UsAlloc(SizeInBytes);

is used.

The Us library provides storage allocation routines for arrays and matrices. These routines scatter data across the memories of the machine to reduce memory contention.

The routine

UsAllocScatterMatrix(nrows,ncols,element_size);

allocates a matrix that is scattered by rows over the memories of the machine.

It does this by allocating a vector of pointers *nrows* long, and *nrows* separate vectors, each containing *ncols* items of size *element_size* bytes. The row vectors are allocated in separate memories. *UsAllocScatterMatrix* returns a pointer to the vector of pointers. The vector of pointers is itself filled in with pointers to the scattered row vectors (See Fig. A.3). Elements of an array A allocated in this way can be referenced using standard C notation as follows:

A[row][col]

Internally, *UsAllocScatterMatrix* uses the routine *UsAllocAndReportC* to scatter

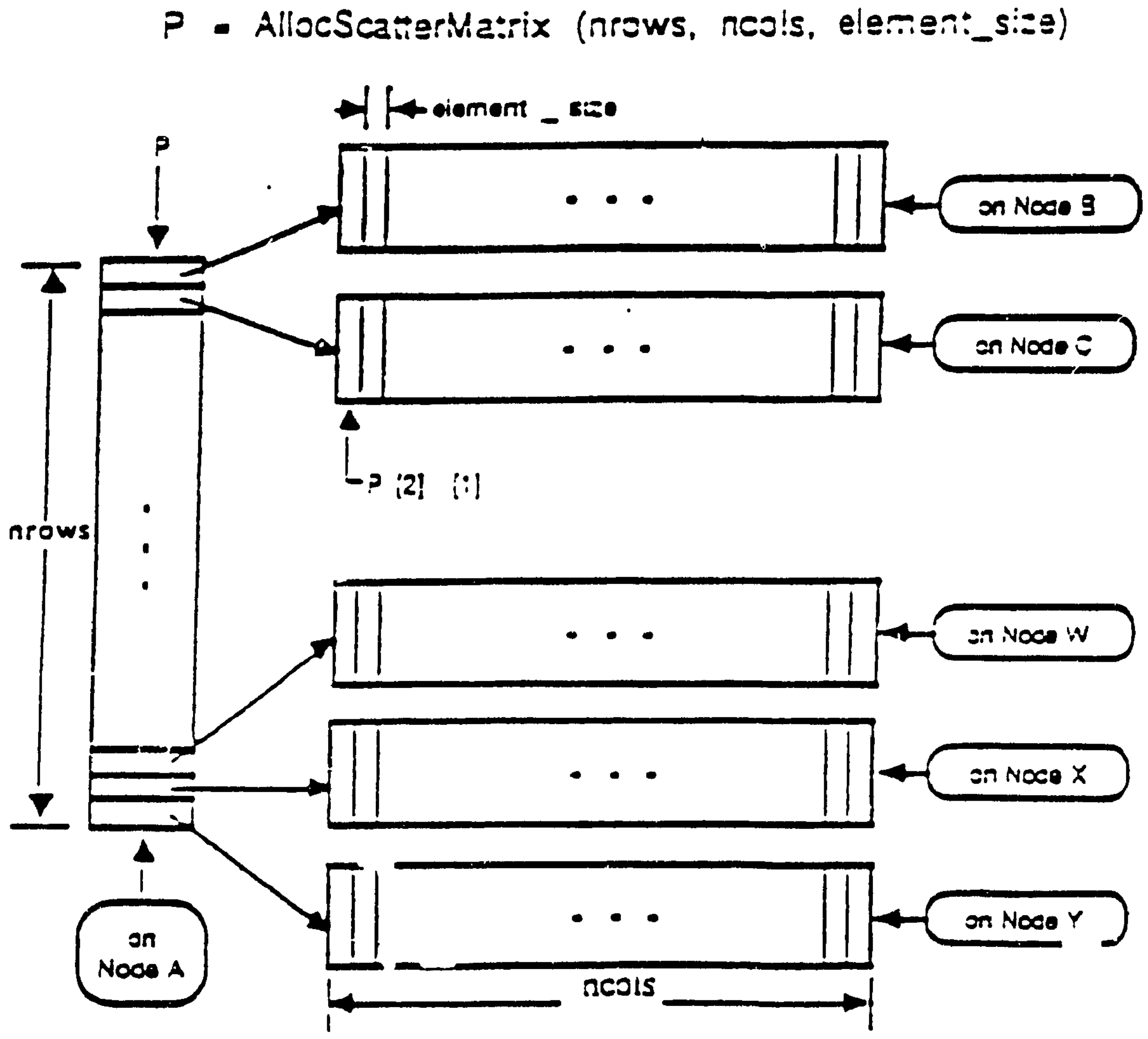


Fig. A.3. A Scattered Matrix Created by UsAllocScatterMatrix.

the row vectors.

There are other allocators which define space only in local processors or specified processors. For more details on allocators see [BBN90].

Once the space has been allocated in globally-shared memory for certain data structures, the processors must be able to access them. This is done through the *Share* mechanism of the Us.

Assume X is an integer variable declared as global or static.

Share(&X);

copies the value of X into each processor that performs tasks generated by subsequent task generators. The value copied is the value X has when *Share* is involved. X is set prior to the first call of the task worker routine on that processor. The effect of share is illustrated schematically in Fig. A.4.

When processor P executes *Share* (&X), the following happens:

1. Processor P allocates a *Share* block in shared memory to hold both the address of X and the current value of X.
2. The *Share* block, is linked together with other *Share* blocks and they can all be found when needed.

When processor Pa begins working on a task generator for the first time, the following happens.

1. Processor Pa finds all of the *Share* blocks that have been linked together.
2. For each *Share* block, processor Pa copies the value of X saved in *Share* block to the address of X, which was also saved in the *Share* block.

Allocating X in static or global process private memory ensures that the address of X is at the same location in all processes. If X was a local variable (i.e., allocated on the stack), processors P and Pa could have X located at different addresses.

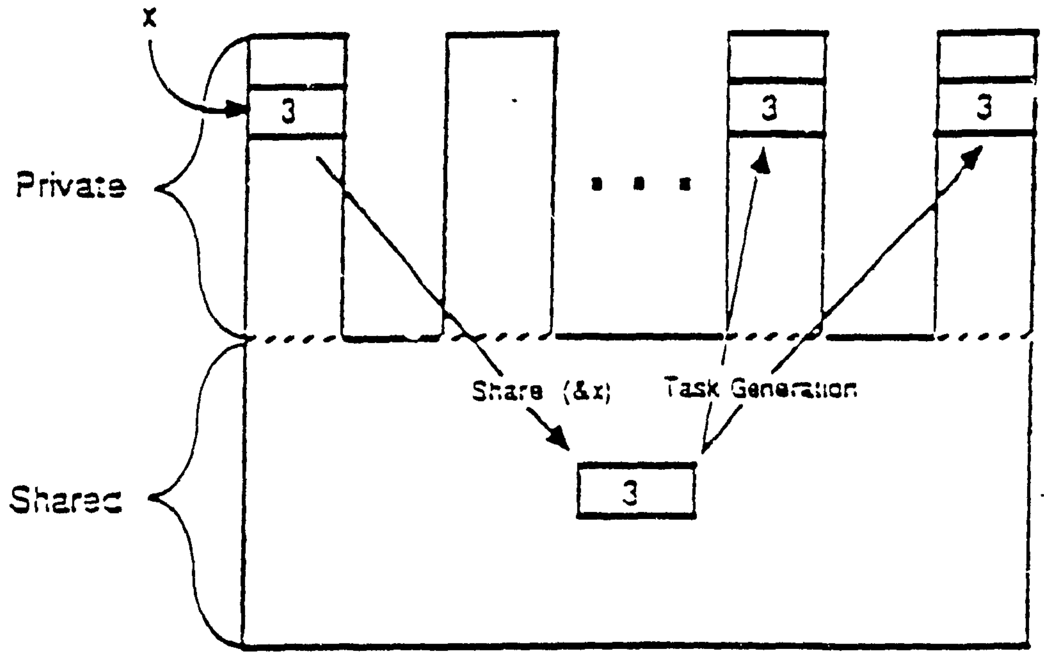


Fig. A.4. Share Passes Copies of Process Private Variables.

When many processors make frequent references to many elements of an array allocated by *UsAllocScatterMatrix* , it is often desirable for each processor to have its own copy of the vector of pointers created by *UsAllocScatterMatrix* . This reduces contention for the pointers which are all stored in a single memory and must be referenced to access the array elements. The routine

```
ShareScatterMatrix(&p,nrows);
```

where p is

```
p =   UsAllocScatterMatrix(nrows, ncols, element_size);
```

causes such copies to be made. p is globally shared. Each processor that performs tasks generated by task generators called after the call to *ShareScatterMatrix* will have its p set to point to a local copy of the vector of pointers (the local copy is allocated in globally-shared memory).

Task Generators

The Us processor management is accomplished using task generators. A task is a basic unit of computation; a Uniform task is a subroutine call. At any instant, there is a set of runnable tasks that must be mapped on to the available set of processors. The Us takes the view that both the set itself and the priority of items within the set are dynamically changing; as a result, a simple queue is not an adequate model of the task structure. Instead, the Us requires a user-supplied task generation procedure that can answer the question, "What is the current most important task at this instant?"

It is good practice to make the tasks themselves small. The responsiveness of the system to changes in priorities depends on the size of a test, because once a task is started, the system runs it to completion. Also, even if the priorities are not changing, there will come a point towards the end of a task generator when all the tasks have been generated by the task generation procedure. When that happens and if there are no other active generators, some processors will sit idle while others finish the last task. If the tasks are small in size, the idle time will not have much impact on program efficiency.

The Us supports two generator control disciplines. Synchronous generators return to the caller after all of the generated tasks have been processed. Furthermore, the processor that calls a synchronous generator always works on the tasks that are generated. Asynchronous generators return to the caller as soon as the generator has been activated. This enables the calling process to do other work. The calling process can later work on generated tasks if it so chooses.

The Us matches available processors to the generated tasks and keeps track of active generators. Whenever a processor has no work to do, it obtains a task using the task generation procedure for one of the active generators. When a Us program begins execution all the processors, except the one used to start the program, are idle. As long as there are active generators with tasks to be done, there are no idle processors.

Deadlocks are possible using generators. For synchronous generators, since there is always at least one processor working on each generator, (perhaps recursively), progress should be made unless that processor hangs. With asynchronous generators, more care needs to be taken to avoid race and deadlock conditions.

The Us Library includes a collection of generator activator procedures that embody various commonly-used task generation procedures. Here only one family of the synchronous generator activator procedures will be described. For more details on other synchronous and asynchronous activator procedures, see [BBN90].

The index family of the synchronous generator activator procedure are next discussed. Each family of generators has a simple, abortable, limited or full version of the call to the generator. Our main interest is in the simple version of the call to the generator.

Consider a subroutine Worker(Arg, index) which is to be called for all values of index from 0 through Range − 1. A call of the form

```
code    =    GenOnI(Worker, Range);
```

generates tasks of the form

```
Worker(0, index);
```

Note that the worker routine is passed a dummy Arg parameter.

A.4 Synchronization and Atomic Operations

Sometimes two processors need to work on the same data at the same time. If the order of work does not matter (e.g., incrementing a counter), the principal concern is that the processors do not interfere with one another (i.e., one finishes before the other starts). If the order of work does matter (e.g., task A is writing and task B is reading), the program logic may be flawed in the sense that task B is really not ready to run, and should not have been generated until A finished.

Mach 1000 supports atomic operations for both 32-bits and 16- bit quantities. For example, Atomic-add, Atomic-and, Atomic-or etc.

Some cases may require more than a simple atomic operation. In these cases, it may be necessary to construct a lock around the code as follows:

```
lock
    Operations that must be atomic
unlock;
```

The Us provides the following lock and unlock operations:

```
Uslock(lock, n);
UsUnlock(lock);
```

The Uslock operation is a "busy wait" type of lock, where *lock* is a pointer to a short variable used as the lock (assumed to have been initialized in the unlocked state with value zero), and n is an integer that specifies the time to wait in tens of microseconds between attempts to get the lock. Using zero for n forces use of a default value, which is about one millisecond. If a program simply needs to wait until something occurs, and if "busy" waiting is acceptable, it can use LockWait as follows:

```
While (something has not occurred)
    LockWait(n);
```

where n is an integer that specifies the time to wait in ten of microseconds. As usual, using n = 0 forces use of a default value of about one millisecond.

There are various other routines for atomic operations. For more details see [BBN90].

APPENDIX B

SHARED-MEMORY SIMULATION OF THE DISTRIBUTED PROTOCOL FOR THE MAX – FLOW PROBLEM

Global Shared Variables

For a given node i:

$a0(i)$	Adjacency list of node i. Node j is included in this list if it is connected to node i by an edge $(i\ ,j)$ directed from i to j.
$a1(i)$	Adjacency list of node i. Node j is included in this list, if it is connected to node i by an edge $(j\ ,i)$ directed from j to i.
$cap\,(i)$	j^{th} element of $cap\,(i)$ contains the capacity of edge $(i\ ,j)$ directed from i to j.
$degree\,(i)$	The number of edges incident on node i except those connected to s and t.
$safeset\,(i)$	At the end of each pulse, the node determines it is safe to start the next pulse. The next pulse begins only if all its neighbours are safe. This is indicated as $safeset(i) = 0$. At the beginning of a new pulse $safeset(i)$ is initialized to the number of nodes adjacent to i.
$d\,(i)$	Contains two types of information; degree of node i as defined above and distance of the node from the sink.
$delta\,0(i)$	If $(i\ ,j)$ is an outgoing edge at i, then $delta\,0(i)(j)$ will store two types of information:

delta: The amount of flow pushed along $(j\ ,i)$. Set to 0 initially.

terminate : TERMINATE information, set to FALSE initially.

delta 1(i)	If (j, i) is an incoming edge of i, then *delta* 1$(i)(j)$ will store two types of information arriving from j. *terminate* : TERMINATE information, set to FALSE initially. *delta*: The amount of flow pushed along (j, i). Set to zero initially.
check 0(i)	A boolean variable corresponding to *delta* 0(i). If the *jth* element of *check* 0(i) is TRUE then j can write in the j^{th} element of *delta* 0(i). If it is FALSE, i can read from the j^{th} element of *delta* 0(i). It is used mainly for synchronizing read/write operations. Initially all entries are TRUE.
check 1(i)	A boolean variable corresponding to *delta* 1(i). If the j^{th} element of *check* 0(i) is TRUE then j can write in the j^{th} element of *delta* 1(i). If it is FALSE, i can read from the j^{th} element of *delta* 1(i). Used mainly for synchronizing read/write operations. Initially all entries are TRUE.
size	The number of nodes in the graph.

Local variables at node *i*

excess	Excess at node i.
tempdel 0(i),*tempdel* 1(i)	A vector which stores the amount of flow pushed along outgoing and incoming edges, respectively, at node i.
distance (i)	A vector containing the minimum distance estimate of all the neighbours of node i.
termset (i)	A boolean vector indicating the neighbour from whom TERMINATE messages have been received. *termset*$(i)(j)$ = TRUE, if i has received a TERMINATE message from j.

flow 0(*i*)	A vector containing the total amount of flow along all outgoing edges at node *i*.
flow 1(*i*)	A vector containing the total amount of flow along all incoming edges at node *i*.
totaloutflow (0)	The total flow out of *s*.
totalinflow (*size* −1)	The total flow into *t*
check (*s*)	A one-bit variable. If it is TRUE, then it means that some flow was pushed during the procedure check − push − applicability.

Note: Nodes 0 and (size-1) represent the source and the sink respectively. A data structure x which contains more than one piece of information will be addressed as follows

x. item.

```
Procedure Read_message()
begin
    for all j ∈ a0[i] do
    begin
        while (check0[i][j] ≠ FALSE) Wait();
       excess[i] = excess[i] - delta0[i][j].value;
       flow0[i][j] =flow0[i][j]+delta0[i][j].value;
        if (delta0[i][j].end = TRUE) then
        begin
            termset[i][i]= TRUE;
            termset[i][j]=TRUE;
        end
       check0[i][j]=TRUE;
        if ((j ≠ (0)) AND (j ≠(size-1))) then
           while (d[j].deg =0) Wait();
       Lock(lock);
           distance[j]=d[j].dist;
            if ((j ≠ 0) AND (j ≠ (size-1))) then
               d[j].deg =d[j].deg-1;
       Unlock(lock);
    end
    for all j ∈ a1[i] do
    begin
        while (check1[i][j] ≠ FALSE) Wait();
       excess[i] =excess[i] - delta1[i][j].value;
       flow1[i][j]=flow1[i][j]+delta1[i][j].value;
```

```
            if (delta1[i][j].end = TRUE) then
            begin
                termset[i][i]=TRUE;
                termset[i][j]=TRUE;
            end
            check1[i][j]=TRUE;
            if ((j ≠ 0) AND (j ≠ (size-1))) then
                while (d[j].deg = 0) Wait();
            Lock(lock);
               distance[i][j]=d[j].dist;
               if ((j ≠ 0) AND (j ≠ (size-1))) then
                  d[j].deg =d[j].deg-1;
            Unlock(lock);
        end
    end
    Procedure  Check_Push_applicability ()
    begin
        for all j ∈ a0[i] do
        begin
            residualcap[i]=cap[i][j]- flow0[i][j];
            if ((excess[i] > 0) AND (residualcap[i] > 0)
              AND (d[i].dist = (distance[i][j]+1))) then
            begin
                if (excess[i] < residualcap[i]) then
                    tempdel0[i][j]=excess[i];
                else
```

```
                    tempdel0[i][j]=residualcap[i];
                excess[i] =excess[i] - tempdel0[i][j];
                flow0[i][j] =flow0[i][j]+tempdel0[i][j];
            end
            else
               tempdel0[i][j]=0;
        end
        for all j ∈ a1[i] do
        begin
            residualcap[i] = (-flow1[i][j]);
            if ((excess[i] > 0) AND (residualcap[i] > 0)
              AND (d[i].dist = (distance[i][j]+1))) then
            begin
                if (excess[i] < residualcap) then
                    tempdel1[i][j]= excess[i];
                else
                    tempdel[i][j]=residualcap[i];
                excess[i] =excess[i]-tempdel1[i][j];
                flow1[i][j] =flow1[i][j]+tempdel1[i][j];
            end
            else
               tempdel1[i][j]=0;
        end
    end
    Procedure Relabel ()
    begin
```

```
distance(i)(i) = $\min_{\forall j \in \{a0 \cap a1\}} \left\{ distance(i)(j) \right\} + 1$;
end
Procedure Write_message()
begin
    for all j ∈ a0[i] do
    begin
        while (check1[j][i] ≠ TRUE) Wait();
        if (check[i] = TRUE ) then
          delta1[j][i].value= (-temdel0[i][j]);
        else
          delta1[j][i]=0;
        check1[j][i]=FALSE;
    end

    for all j ∈ a1[i] do
    begin
        while (check0[j][i] ≠ TRUE) Wait();
        if (check[i] = TRUE ) then
          delta0[j][i].value= (-tempdel1[i][j]);
        else
          delta0[j][i]=0;
        check0[j][i]=FALSE;
    end
end
Procedure Store_new_label ()
begin
```

```
    while (d[i].deg $\neq$ 0) Wait();
    Lock(lock);
        d[i].dist=distance[i][i];
        d[i].deg=degree[i];
    Unlock(lock);
end
Procedure Set_pulse ()
begin
    if (p = 0) then
    begin
        for all j $\in$ a0[0] do
        begin
            while (safeset[j]= 0) Wait();
            Lock(key);
                safeset[j] = safeset[j] - 1;
            Unlock(key);
        end
        while (safeset[size-1] = 0) Wait();
        Lock(key);
            safeset[size-1] =safeset[size-1]-1;
        Unlock(key);
    end
    if (i = (size-1)) then
    begin
        for all j $\in$ a1[size-1] do
        begin
        **if** (termset[i][j] $\neq$ TRUE) **then**
        **begin**
            **while** (safeset[j] = 0) Wait()
            Lock(key);
              safeset[j] = safeset[j]-1;
            Unlock(key);
        **end**
    **end**
    **if** (termset[i][0] $\neq$ TRUE) **then**
    **begin**
        **while** (safeset[0] = 0) Wait();
        Lock(key);
          safeset[0]=safeset[0]-1;
        Unlock(key);
    **end**
    **if** i $\neq$ s or t **then**
    **begin**
        **for** all j $\in$ a0[i] AND a1[i] **do**
        **begin**
            **if** (termset[i][j] $\neq$ TRUE) **then**
            **begin**
                **while** (safeset[j] = 0) Wait();
                Lock(key);
                  safeset[j] =safeset[j]-1;
                Unlock(key);
            **end**
            end
        end
    end

    Procedure  Source_read_write ()
    begin
        for all j ∈ a0[0] do
        begin
            while (check0[0][j] ≠ FALSE) Wait();
            flow0[0][j] = flow0[0][j] + delta0[0][j].value;
            check0[0][j]=TRUE;
            totaloutflow[0]=totaloutflow[0] + flow0[0][j];
        end
        while (check1[0][j] ≠ FALSE) Wait();
        flow1[0][j]=delta1[0][size-1].value+ flow0[0][j];
        totalinflow[size-1] = delta1[0][size-1].value;
        check1[0][size-1]=TRUE;
        for all j ∈ a0[0] do
        begin
            while (check1[j][0] ≠ TRUE) Wait();
            delta1[j][0].value=0;
            check1[j][0]=FALSE;
        end
        while (check1[size-1][0] ≠ TRUE) Wait();
        delta1[size-1][0].value=0;
        check1[size-1][0]=FALSE;
    end
```

```
Procedure  Source_finish()
begin
    for all j ∈ a0[0] do
    begin
        while (check0[0][j] ≠ FALSE) Wait();
        check0[0][j]=TRUE;
    end
    while (check1[0][size-1] ≠ FALSE) Wait();
    check1[0][size-1]=TRUE;
    for all j ∈ a0[0] do
    begin
        while (check1[j][0] ≠ TRUE) Wait();
        delta1[j][0].value=0;
        delta1[j][0].end=TRUE;
        check1[j][0]=FALSE;
    end
    while (check0[size-1][0] ≠ TRUE) Wait(0);
    delta0[size-1][0].value=0;
    delta0[size-1][0].end=TRUE;
    check0[size-1][0]=FALSE;
end
Procedure  Process_source()
begin
    d(0).dist=size-1;
    totaloutflow[0]=0;
    totalinflow[0]=0;
```

```
for all (j) ∈ a(0) do
begin
    flow0[0][j]=cap[0][j];
    delta1[j][0].value= -cap[0][j];
    check1[j][0]=FALSE;
    totaloutflow[0] = totaloutflow[0] + flow0[0][j];
end
delta0[size-1][0].value=0;
check0[size-1][0]=FALSE;
repeat
begin
    Lock(key);
        reset safeset(0);
    Unlock(key);
    Source_read_write();
    if (-totaloutflow[0] = totalinflow[size-1]) then
        termset[0][0]=TRUE;
    Set_pulse();
    while (safeset[0] ≠ 0) Wait();
    if (termset[0][0] = TRUE) then
    begin
        Lock(key);
            reset safeset[0];
        Unlock(key);
        Source_finish();
        Set_pulse();
```

```
                **while** (safeset[0] $\neq$ 0) Wait();
                Lock(key);
                    reset safeset[0];
                Unlock(key);
                *Source_finish();*
                *Set_pulse();*
            **end**
        **end**
        **until** (termset[0][0] = TRUE)
    **end**
    **Procedure** *Sink_read_write()*
    **begin**
        **for** all j $\in$ a1[size-1] **do**
        **begin**
            **while** (check1[size-1][j] $\neq$ FALSE) Wait();
            flow1[size-1][j] =flow1[size-1][j] + delta1[size-1][j].value;
            totalinflow[size-1] =totalinflow[size-1]+flow1[size-1][j];
            **if** (delta1[size-1][j].end = TRUE) **then**
            **begin**
                termset[size-1][size-1]=TRUE;
                termset[size-1][j]=TRUE;
                check1[size-1][j]=TRUE;
            **end**
        **end**
        **while** (check0[size-1][0] $\neq$ FALSE) Wait();
        flow0[size-1][0] = flow0[size-1][0]+delta0[size-1][0].value;
    if (delta0[size-1][0].end = TRUE) then
    begin
        termset[size-1][size-1]=TRUE;
        termset[size-1][0]=TRUE;
    end
    check0[size-1][0]=TRUE;
    for all j ∈ a1[size-1] do
    begin
        while (check0[j][size-1] ≠ TRUE) Wait();
        delta0[j][size-1].value=0;
        check0[j][size-1]=FALSE;
    end
    while (check1[0][size-1] ≠ TRUE) Wait();
    delta1[0][size-1].value= totalinflow[size-1];
    check1[0][size-1]=FALSE;
end

Procedure Sink_finish()
begin
    for all j ∈ a1[size-1] do
    begin
        if (termset[size-1](j) ≠ TRUE) then
        begin
            while (check1[size-1][j] ≠ FALSE) Wait();
            check1[size-1][j]=TRUE;
        end
    end
```

```
        if (termset[size-1][0] ≠ TRUE) then
        begin
            while (check0[size-1][0] ≠ FALSE) Wait();
            check0[size-1][0]=TRUE;
        end

        for all j ∈ a1[size-1] do
        begin
            if (termset[size-1][j] ≠ TRUE) then
            begin
                while (check0[j][size-1] ≠ TRUE) Wait();
                delta0[j][size-1].value=0;
                delta[0][j][size-1].end=TRUE;
                check[0][j][size-1]=FALSE;
            end
        end
        if (termset[size-1][0] ≠ TRUE) then
        begin
            while (check1[0][size-1] ≠ TRUE) Wait();
            delta1[0][size-1].value= 0;
            delta1[0][size-1].end=TRUE;
            check1[0][size-1]=FALSE;
        end
    end
    Procedure Process_sink()
    begin
        d[size-1].dist=0;
```

```
for all j ∈ a1[size-1] do
begin
    delta0[j][size-1].value=0;
    check0[j][size-1]=FALSE;
end
delta1[0][size-1].value=0;
check1[0][size-1]=FALSE;
repeat
begin
    Lock(key);
       reset safeset[size-1];
    Unlock(key);
    Sink_read_write() ;
    Set_pulse();
    while (safeset[size-1] ≠ (0)) Wait();
    if (termset[size-1][size-1] = TRUE) then
    begin
        decrement safeset(size-1) by the number of neighbours who sent TER-
        MINATE message;
        Sink_finish ();
        Set_pulse();
        while (safeset[size-1] ≠ 0) Wait();
        Lock(key);
           reset safeset[size-1];
        Unlock(key);
        Sink_finish();
```

```
                *Set_pulse();*
            **end**
        **end**
        **until** termset[size-1][size-1] = TRUE;
**end**

**Procedure** *Intermediate_finish()*
**begin**
        **for** all j $\in$ a0[i] **do**
        **begin**
            **if** (termset[i][j] $\neq$ TRUE) **then**
            **begin**
                **while** (check0[i][j] $\neq$ FALSE) Wait();
                check0[i][j]:=TRUE;
                **if** (j $\neq$0) AND (j $\neq$ (size-1)
                **begin**
                    **while** (d[j].deg = 0) Wait();
                    Lock(lock);
                        d[j].deg = d[j].deg-1;
                    Unlock(lock);
                **end**
            **end**
        **end**
        **for** all j $\in$ a1[i] **do**
        **begin**
            **if** (termset[i][j] $\neq$ TRUE) **then**
            **begin**
            while (check1[i][j] ≠ FALSE) Wait();
            check1[i][j]=TRUE;
            if ((j ≠ (0)) AND (j != (size-1))) then
            begin
                while (d[j].deg = (0)) Wait();
                Lock(lock);
                   d[j].deg = d[j].deg - 1;
                   Unlock(lock);
            end
        end
end
for all j ∈ a0[i] do
begin
        if (termset[i][j] ≠ TRUE)
        begin
            while (check1[j][i] ≠ TRUE)Wait();
            delta1[j][i].value= 0;
            delta1[j][i].end=TRUE;
            check1[j][i]=FALSE;
        end
end
for all j ∈ a1[i] do
begin
        if (termset[i][j] ≠ TRUE) then
        begin
            while (check(0)[j][i] ≠ TRUE) Wait();
```

```
                delta0[j][i].value= 0;
                delta0[j][i].end=TRUE;
                check0[j][i]=FALSE;
            end
        end
    end

Procedure Process_intermediate ()
begin
    for all j ∈ a0[i] do
    begin
        delta1[j][i].value=0;
        check1[j][i]=FALSE;
    end
    for all j ∈ a1[i] do
    begin
        delta0[j][i].value=0;
        check1[j][i]=FALSE;
    end
    repeat
    begin
    Lock(key);
        reset safeset[i];
    Unlock(key);
    Read_message();
    if excess[i] > 0 then
```

```
begin
    Check_push_applicability ();
    check[i]=TRUE;
end
if (excess[i] > 0) then
begin
    Relabel();
    relabel[i]=TRUE;
end
Write_message ();
check[i]=FALSE;
if (relabel = TRUE ) then
   Store_new_label();
else
begin
     if (termset[i][i] = TRUE) then
    begin
         while (d[i].deg ≠ (0)) Wait();
        Lock(lock);
           d[pid].deg=# neighbours from whom a TERMINATE message is
           not received;
        Unlock(lock);
    end
    else
    begin
         while (d[i].deg ≠ (0)) Wait();
```

```
            Lock(lock);
                d[i].deg=degree[i];
            Unlock(lock);
        end
    end
    Set_pulse ();
    while (safeset[i] ≠ [0]) Wait();
    if (termset[i][i] = TRUE) then
    begin
        Lock(key);
            decrement safeset by the # neighbours who sent TERMINATE message;
        Unlock(key);
        Intermediate_finish ();
        while (d[i].deg ≠ (0)) Wait();
        Lock(lock);
            update d[i].deg;
        Unlock(lock);
        Set_pulse();
        while (safeset[i] ≠ 0) Wait();
        Lock(key);
            update safeset[i];
        Unlock(key);
        Inte.mediate_finish();
        while (d[i].deg ≠0) Wait();
        Lock(lock);
            update d[i].deg
            Unlock(lock);
            Set_pulse();
        end
    end
end
```

APPENDIX C

SHARED MEMORY SIMULATION OF THE DISTRIBUTED PROTOCOL FOR THE SHORTEST PATH PROBLEM

Globally Shared Variables

For a given node i:

$a0(i)$	Adjacency list of node i – a set containing all the successors of i. (Note : j is a successor of node i if (i, j) is directed from i to j.)
$a1(i)$	Adjacency list of node i – a set containing all the dominators of i. (Note : j is a dominator of node i if (j, i) is directed from j to i.)
degree (i)	The number of edges incident on node i.
size	The number of nodes in a graph.
length (i)	*length* $(i)(j)$ indicates the weight of edge (i, j).
num (i)	Contains two types of information. *ack* : number of acknowledgements yet to be received. *pulsenum* : the number of the pulse the node is currently creating.
distinfo (i)	*distinfo* $(i)(j)$ for an edge (i, j) stores three types of information received from dominator j. *distance* : the calculated distance estimate of j. *terminate* : If it is to terminate then *terminate* = TRUE otherwise, terminate = FALSE. *change* : Indicates if an update has occurrred in the current $d(i)$

	feasible :	Determines the feasibility of the algorithm. Initially TRUE.

mesgterm (i)	*mesgterm* $(i)(j)$ stores information received from successors. Receives only terminate and feasible information.
checkdist (i)	Boolean data structure. *checkdist* $(i)(j)$ = TRUE if a write has occurred and FALSE if a read has occurred in *distinfo* $(i)(j)$. Mainly used for synchronizing read and write operations.
checkterm (i)	Boolean data structure. *checkterm* $(i)(j)$ = FALSE if a read has occurred in mesgterm(i)(j). Mainly used for synchronizing read and write operations.
safeset (i)	At the end of each pulse, the node indicates to the negihbours its completion of the pulse. A node does not start the next pulse until all its neighbours are safe and it is safe also. This is indicated as *safeset* (i) = 0. *safeset* (i) = *degree* (i) at the beginning of each pulse.

Local variables at node *i*

received	Indicates that the predeessor has received an *ack*.
pred (i)	predecessor of node i.
ackset (i)	A list containing all dominators from whom updated distance information has been received. This is indicated as *change* (i) = TRUE in the received message (*distinfo* data structure).
termset (i)	Boolean variable indicating the neighbours from whom TERMINATE message is received. *terminate* $(i)(j)$ = TRUE if j has sent a TERMINATE message.

dmin (i)	The j^{th} element contains $d(j)$ if j belongs to *ackset* (i).
change (i)	A boolean variable which determines if an update has occurred in $d(i)$. This information is propagated to all successors. *change* (i) = TRUE if an update has occurred, otherwise FALSE. Initially it is FALSE.
prevchange (i)	The value of *change* in the previous pulse.
active (i)	Indicates if there exists a predecessor of i which is yet to receive an acknowledgement from i. *active* (i) = TRUE, if there is such a predecessor. After sending an acknowledgement back to the predecessor *active* (i) is set to FALSE.
$d(i)$	The current distance of i from the source.

```
Procedure  Source_initialization
begin
    for all j ∈ a0[0] do
    begin
        distinfo[j][0].distance=(*d)+length[0][j];
        distinfo[j][0].change=TRUE;
        checkdist[j][0]=FALSE;
    end
end
Procedure  Source_read()
begin
    for all j ∈ a0[0] do
    begin
        while (checkterm[0][j] ≠ FALSE) Wait();
        checkterm[0][j]=TRUE;
    end
end
Procedure  Source_write()
begin
    for all j ∈ a0[0] do
    begin
        while (checkdist[j][0] ≠ TRUE) Wait();
        distinfo[j][0].change=FALSE;
        if (termset[0][0] = TRUE) then
        begin
            distinfo[j][0].terminate=TRUE;
```

```
                checkdist[j][0]=FALSE;
            end
        end
    end
    Procedure  Source_node()
    begin
        Source_initialization();
        Lock(key);
          num[0].ack=num[0].ack+# successors of source;
          num[0].pulsenum=num[0].pulsenum+1;
        Unlock(key);
        repeat
        begin
            if  (# ACK received ≠ 0)  then
            begin
                Lock(key);
                  Reset safeset[0];
                Unlock(key);
                Source_read();
                Source_write();
                Pulse_finish();
                while  (safeset[0] ≠ 0) Wait();
                UsLock(lock);
                  increment pulse number;
                UsUnlock(lock);
            end
            else
            begin
                termset[0][0]=TRUE;
                Reset safeset[0][0];
                Source_read();
                Source_write();
                Pulse_finish();
                while (safeset[0] ≠ 0) Wait();
                Reset safeset[0][0];
                Source_read();
                Source_write();
                Pulse_finish();
            end
        end
        until (termset[0][0] = TRUE)
    end
    Procedure Intermediate_read()
    begin
        for all j ∈ al[i] do
        begin
            while (checkdist[i][j] ≠ FALSE) Wait();
            if (distinfo[i][j].change = TRUE) then
            begin
                insert j in ackset[i];
                dmin[j]=distinfo[i][j].distance;
            end
            if (distinfo[i][j].terminate=TRUE) then
            begin
                termset[i][i]=TRUE;
                if distinfo[i][j].feasible=FALSE then infeasible[i]=TRUE;
            end
            checkdist[i][j]=TRUE;
        end
        for all j ∈ a0[i] do
        begin
            while (checkterm[i][j] ≠ FALSE) Wait();
            if (mesgterm[i][j] = TRUE) then
            begin
                termset[i][i]=TRUE;
                if mesgterm[i][j].feasible=FALSE then infeasible[i]=TRUE;
            end
            checkterm[i][j]=TRUE;
        end
    end
    Procedure Compute_minimum_distance()
    begin
        if ackset[i] not empty then
        begin
            determine the new minimum distance and new predecessor;
            place the minimum distance in dmin[i][i]i;
            if dmin[i][i] < d(i) then
            begin
        d[i]=dmin[i];
        change[i]=TRUE;
        while num[pred[i]].pulsenum ≠ num[i].pulsenum then
        if (pred[i] ≠ i)
        begin
            if (received[i] ≠TRUE) then
                send ACK to the old predecessor.
            else
                received[i]=FALSE;
        end replace old predecessor by new predecessor;
        for all j ∈ ackset[i] j ≠ i do
        begin
            while num[j].pulsenum ≠ num[i].pulsenum Wait();
           send ACK;
        end
    end
    else
    begin
        change[i]=FALSE;
        for all j ∈ ackset[i] do
        begin
            while (num[j].pulsenum ≠ num[i].pulsenum) Wait();
           send ACK;
        end
    end
end
```

```
end
Procedure Intermediate_write()
begin
    for all j ∈ a0[0] do
    begin
        while (checkdist[j][i] ≠ TRUE) Wait();
        distinfo[j][i].distance=d[i]+length[i][j];
        dmin[j]=distinfo[j][i].distance;
        distinfo[j][i].change=change[i];
        checkdist[j][i]=FALSE;
    end
    for all j ∈ a1[i] do
    begin
        while (checkterm[j][i] ≠ TRUE) Wait();
        checkterm[j][i]=FALSE;
    end
end
Procedure Intermediate_finish_SP ()
begin
    for all j ∈ a0[0] do
    begin
        if termset[i][j] ≠ TRUE then
        begin
            while (checkterm[i][j] ≠ FALSE) Wait();
            checkterm[i][j]=TRUE;
        end
```

```
end
for all j ∈ a1[i] do
begin
     if termset[i][j] = TRUE then
     begin
          while (checkdist[i][j] ≠ FALSE) Wait();
          checkdist[i][j]=TRUE;
     end
end
for all j ∈ a0[i] do
begin
     while (checkdist[j][i] ≠ TRUE) Wait();
     distinfo[j][i].distance=infinity;
     distinfo[j][i].terminate=TRUE;
     distinfo[j][i].terminate=infeasible;
     distinfo[j][i].change=FALSE;
     checkdist[j][i]=FALSE;
end
for all j ∈ a1[i] do
begin
     if termset[i][j] ≠ TRUE then
     begin
          while (checkterm[j][i] ≠ TRUE) Wait();
          mesgterm[j][i]=TRUE;
          mesnterm[j][i]=infeasible[i];
          checkterm[j][i]=FALSE;
```

```
            end
        end
    end
    Procedure  Pulse_finish()
    begin
        if  (p=0)  then
        begin
            while  (safeset[j] = 0) Wait();
            Lock(key,0);
                safeset[j] =safeset[j]-1;
            Unlock(key);
        end
        else
        begin
            for  all j ∈ a0[i] AND a1[i]  do
            begin
                if  (termset[i][j] ≠ TRUE)  then
                begin
                    while  (safeset[i][j] = 0) Wait();
                    Lock(key,0);
                        safeset[i][j] =safeset[i][j]-1;
                    Unlock(key);
                end
            end
        end
    end
```

```
Procedure  Intermediate_node()
begin
    Process_initialization();
    repeat
    begin
        Reset safeset[i];
        if (pred[i] ≠ i ) then
        begin
            if ((active= TRUE) AND (# ACK's received=0)) then
            begin
                while (num[pred[i]].pulsenum ≠ num[i].pulsenum) Wait();
                send an ACK to the predecessor;
                active=FALSE;
                received=TRUE;
            end
        end
        prevchange[i]=change[i];
        change[i]=FALSE;
        Intermediate_read ();
        Compute_minimum_distance();
        Intermediate_write();
        if ((prevchange[i] = TRUE) AND (change[i] =FALSE)) then
            active=TRUE;
        Pulse_finish ();
        while ( safeset[i] ≠ 0) Wait();
        if (change[i]= TRUE) then
```

```
            increment num[i].ack  by the number of successors;
        increment pulse number;
        if (termset[i][i] = TRUE) then
        begin
            if (still an ACK is pending ) then
                set infeasible=TRUE;
            decrement safeset[i] by the # neighbours who sent TERMINATE message;
            Intermediate_finish_SP();
            Pulse_finish(pid,termset);
            while (safeset[i] ≠ 0) Wait();
            Reset safeset[i];
            Intermediate_finish_SP();
            Pulse_finish();
        end
    end
    until (termset[i][i] ≠ TRUE)
end
```

APPENDIX D

SHARED-MEMORY SIMULATION OF THE DISTRIBUTED PROTOCOL FOR THE PRIMAL – DUAL

Globally Shared Variables

For a given node i:

$A0(i)$	Adjacency list of node i. Node j is included in this list if it is connected to node i by an edge (i, j) directed from i to j.
$A1(i)$	Adjacency list of node i. Node j is included in this list, if it is connected to node i by an edge (j, i) directed from j to i.
cost (i)	*cost* $(i)(j)$ indicates the weight of edge (i, j).
nodecount (i)	Number of nodes in the network – for synchronizin_ ɔetween tasks.
tasknum (i)	The number of the pulse node i is currently executing.
$C(i)$	*jth* element of $C(i)$ contains the capacity of edge (i, j) directed from i to j.
demprime (i)	Indicates the demand or supply of each node.
dzero	Initially contains size-2 nodes. A processor decrements the value in *dzero* when its demand equals 0.
$Y(i)$	Contains $Y'(i)+d(i)$ in the current pulse, where $Y'(i)$ is the value in the previous pulse.
f 0prime (i)	Vector containing the total amount flow along all outgoing edges of i.
f 1prime (i)	Vector containing the total amount of flow along all incoming edges of i.

Local variables at node *i*

pdquit (*i*)	Contains the TERMINATE value.
setdemand (*i*)	Indicates whether a processor has already decremented the value in *dzero* when the demand was equal to 0. This variable ensures that *dzero* is not decremented more than once when the demand equals 0.
demand (*i*)	Indicates the residual demand or supply of a node.

```
Procedure  Reduce_nodecount()
begin
    for all nodes j except i do
    begin
        while (tasknum[j] ≠ tasknum[i]) Wait();
        Lock(nodekey);
            nodecount[i] =nodecount[i]-1;
        UsUnlock(nodekey);
    end
    while (nodecount[i] ≠ 0) Wait();
end
Procedure  Update_nodecount()
begin
    Lock(nodekey);
        nodecount[i] =size;
        tasknum[i] =tasknum[i]+1;
    Unlock(nodekey);
end
Procedure  Initialize_PD()
begin
    if (i ≠ size-1) then
        Y[i] = 0;
    Reduce_nodecount();
    Update_nodecount();
    if ((i ≠ size-1) AND (i ≠0)) then
    begin
```

```
        for j ∈ A0[i] do
        begin
            if (Y[i]+cost[i][j] ≥ Y[j]) then
                f0prime[i][j]=0;
            else
                f0prime[i][j]=C[i][j];
            f1prime[j][i]= (-f0prime[i][j]);
        end
    end
    Reduce_nodecount();
    Update_nodecount();
    if ((i≠size-1) AND (i≠0)) then
    begin
        for all j ∈ A1[i] do
            demand[i] =f1prime[i][j]+demand[i];
        for all j ∈ A0[i] do
            demand[i] =f0prime[i][j]+demand[i];
        if (demand[i] = 0) then
        begin
            Lock(nodekey);
                dzero=dzero-1;
                setdemand[i] =TRUE;
            Unlock(nodekey);
        end
    end
    Reduce_nodecount();
    Update_nodecount();
end

Procedure   Terminate_PD ()
begin
    if dzer()=0 then pdquit[i]=TRUE;
    Reduce_nodecount();
    Update_nodecount() ;
end
Procedure   Create_newadjcost()
begin
    if (i = 0) then
    begin
        for all nodes j except sink do
                for all k ∈ A0[j] do
                    begin
                        if (f0prime[j][k] < C[j][k]) then
                        begin
                            Add k in a0[j];
                            Add j in a1[k];
                            length[j][k]=Y[j]-Y[k]+cost[j][k];
                        end
                        if (f0prime[j][k] > 0) then
                        begin
                            Add k in a1[j];
                            Add j in a0[k];
                            length[k][j]=Y[k]-Y[j]-cost[k][j];
                    end
                end
            for  all node j with negative demand  do
            begin
                Add j in a0[0];
                Add 0 in a1[j];
                length[0][j]=0;
            end
        end
        Reduce_nodecount();
        Update_nodecount();
    end
    Procedure  Create_t()
    begin
        if  (i ≠ size-1)
            Y[i]=Y[i]+pathdist[i];
        Reduce_nodecount();
        Update_nodecount();
        if  ((i ≠ size-1)  (i ≠ 0))
        begin
            for  all j ∈ A0[i]  do
            begin
                if  (Y[j] = Y[i]+cost[i][j])  then
                begin
                    if  (f0prime[i][j] < C[i][j])  then
                    begin
                Add j in a0[i];
                cap[i][j]=C[i][j]-f0prime[i][j];
            end
            if (f0prime[i][j] > 0) then
            begin
                Add j in a1[i];
                cap[j][i]=f0prime[i][j];
            end
        end
    end
end
Reduce_nodecount();
Update_nodecount();
if (i = 0) then
begin
    for all intermediate nodes i except sink do
    begin
      count0[i]=a0[i][0];
      count1[i]=a1[i][0];
    end
    for all intermediate nodes i except sink do
            for all j ∈ A0[i] do
              begin
                  if (Y[j] = Y[i] + cost[i][j]) then
                  begin
                      if (f0prime[i][j] < C[i][j]) then
                  begin
                     Add i in a1[j];
                  end
                  if (f0prime[i][j] > 0) then
                  begin
                     Add i in a0[j];
                  end
              end
          end
end
*Reduce_nodecount();*
*Update_nodecount();*
if (i = 0) then
begin
    for all j ∈ A0[0] do
    begin
        for all nodes j with negative demand do
        begin
            Add 0 in a1[j];
            cap[0][j]=(-demand[j] );
        end
    end
    for all nodes j with positive demand do
    begin
        Add t in a0[j];
        Add j in a1[t];
```

```
                cap[j][t]=demand[j];
            end
        end
        *Reduce_nodecount();*
        *Update_nodecount();*
    end
    **Procedure** *PDupdate_flowdemand* ()
    **begin**
        **if** ((i ≠ 0) AND (i ≠ size-1)) **then**
        **begin**
            bound0=count0[i];
            bound1=count1[i];
            **for** ali j ∈ count0[i] **do**
            **begin**
                f0prime[i][j] =f0prime[i][j]+flow0[j];
                f1prime[j][i]= (-f0prime[i][j]);
            **end**
            **for** all j ∈ count1[i] **do**
            **begin**
                f0prime[i][j] =flow1[j]+f0prime[i][j];
                f1prime[j][i]=(-f0prime[i][j]);
            **end**
        **end**
        *Reduce_nodecount();*
        *Update_nodecount();*
        **if** ((i ≠ size-1) AND (i ≠ 0)) **then**
        begin
            for all j $\in$ A1[i] do
                Update demand of i;
            for all j $\in$ A0[i] do
                Update demand of i;
            if ((demand[i] = 0) AND (setdemand $\neq$ TRUE)) then
            begin
                Lock(nodekey);
                    dzero=dzero-1; setdemand:= TRUE;
                Unlock(nodekey);
            end
        end
        Reduce_nodecount();
        Update_nodecount();
    end
    Procedure  Primal_dual()
    begin
        Update_nodecount():
        Enter Shortest Path Algorithm;
        Reduce_nodecount();
        Update_nodecount();
        if (Algorithm feasible) then
        begin
            Initialize_PD();
            Terminate_PD();
            begin
```

```
                while (pdquit $\neq$ TRUE)
                begin
                    Create_newadjcost();
                    Enter Shortest Path Algorithm;
                    Reduce_nodecount();
                    Update_nodecount();
                    Create_t();
                    Enter Max-Flow Algorithm
                    Reduce_nodecount();
                    Update_nodecount();
                    PDupdate_flowdemand();
                    Terminate_PD();
                end
            end
        end
    end
```