NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilming. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylograpiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.
Distributed Algorithms for File Sorting and Shortest Path Finding

by
Buddy Kin-Hung Leung

A thesis submitted to the school of graduate studies in partial fulfillment of the requirements for the Master of Computer Science Degree

at the University of Ottawa

© Buddy Kin-Hung Leung, Ottawa, Ontario, Canada, 1990
The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.
Acknowledgements

I would like to thank my thesis supervisor, Dr. Jorge Urrutia, for his guidance and advice during the development of this work.

I am very grateful for the encouragement, love and support provided by my family.

Above all, my deepest thanks must go to my parents for their endless love, trust and support during these years.

Financial assistance in the form of teaching and research assistantships from the Department of Computer Science at the University of Ottawa are acknowledged with appreciation.
Preface

During the 1980s the marriage of computers and communications has played an increasingly important role in the development of automation technology. This new technology has allowed us to build more sophisticated and advanced systems ranging from banking to military defense systems. A common characteristic among these systems is to serve large networks with geographically distributed resources (e.g. computers and database). This property along with the two basic design objectives, cost-effective computing and system fault tolerance, form major concerns for these systems now generally called distributed systems.

In order to use these systems effectively and efficiently, some distributed algorithms are designed to solve various problems occurring in them. When we design a distributed algorithm, certain assumptions about the network (or computation) model must be made. These assumptions often create some tradeoffs which occur between the local memory and computation complexity, between the message capacity and computation complexity, and between the time and number of messages used by the algorithms. In this thesis we focus on design and analysis, in particular the tradeoffs, of our new distributed algorithms.

The thesis consists of four chapters. Chapter one lays out the foundations of distributed computing. Chapter two presents new decentralized algorithms for sorting files of integers stored in a distributed and asynchronous network. An optimal communication complexity $O(n \log_2 d)$ is obtained under the assumption that extra memory is available at each site in an almost balanced binary tree network, where $n$ is the file size and $d$ is the number of nodes in the network. When extra memory is removed from the nodes, our algorithm still manages to have a very good complexity of $O(n \log_2^2 d)$. Chapter three presents a new decentralized algorithm in shortest path finding in a distributed and synchronous network, in particular we focus on the termination detection problem. The analysis of this algorithm shows an improved complexity over previous methods. Moreover, in chapters two and three we discuss some tradeoffs in the design of these two algorithms. Finally in chapter four we summarize our results and present some unsolved problems arising from this work. Contents of different chapters are intended to be self-contained so that they can be studied independently of each other. This should ease the approach to the different problems.
# Table of Contents

Acknowledgements  
Preface  

Chapter 1  Background Knowledge  
1.1 Models of Distributed Computations ................................................. 1  
1.2 Distributed Computing and Parallel Computing ................................. 3  
1.3 Important Issues of Distributed Algorithms ...................................... 4  
1.3.1 Differences between Distributed and Non-Distributed Algorithms ..... 4  
1.3.2 Types of Messages ....................................................................... 5  
1.3.3 Issue of Synchronous and Asynchronous Networks ......................... 5  
1.3.4 Issue of Centralized and Decentralized Controls ......................... 5  
1.3.5 Issue of Parallelism .................................................................... 6  
1.3.6 Issue of Termination .................................................................. 7  
1.3.7 Performance Measurement .......................................................... 7  
1.4 Tradeoffs in Distributed Algorithms Design ..................................... 8  
1.5 Basic Computation Model and Notations Used in this Thesis ............ 9  

Chapter 2  Tradeoffs in Distributed File Sorting  
2.1 Introduction .................................................................................... 12  
2.2 Previous Results ............................................................................... 14  
2.3 Model and Notations ......................................................................... 16  
2.4 Sorting With Extra Memory .............................................................. 18  
2.4.1 Communication Complexity Analysis of the SDF Algorithm .......... 24  
2.4.2 Time Complexity in the Synchronous Case .................................. 24  
2.5 Sorting Without Extra Memory ......................................................... 26  
2.5.1 Communication Complexity Analysis of the SDFW Algorithm ...... 50  
2.5.2 Time Complexity in the Synchronous Case ................................. 50  
2.6 Tradeoffs between Extra Memory and Complexity .......................... 52  
2.7 Impacts of Local Computation Cost .................................................. 53  
2.8 Closing Remarks ............................................................................ 56
Table Of Contents

Chapter 3  Tradeoffs in Distributed Shortest Path Finding
3.1  Introduction .................................................................................58
3.2  Model and Notations .................................................................60
3.3  Finding the Shortest Path in Synchronous Networks ..................62
3.4  Complexity Analysis of the FSP Algorithm with the Longest Delay ... 64
   3.4.1 For Arbitrary Tree Networks ................................................. 64
   3.4.2 For Linear Networks ........................................................... 66
   3.4.3 For Ring Networks .............................................................. 67
3.5  Tradeoffs between Delay Function and Complexity .................. 69
3.6  Closing Remarks ....................................................................... 70

Chapter 4  Conclusions .................................................................. 71

Appendix
   A. A complete Algorithm for Sorting a Distributed File in Asynchronous
      Almost Balanced Q-ary Tree Networks with Extra Memory .......... 73
   B. A complete Algorithm for Sorting a Distributed File in Asynchronous
      Almost Balanced Q-ary Tree Networks Without Extra Memory ........ 77
   C. A complete Algorithm for Finding the Shortest Path in Arbitrary
      Synchronous Networks ............................................................... 85

References ..................................................................................... 87
Chapter 1

Background Knowledge

While distributed computing methodology is becoming an important research area, the concepts of distributed computing have never been clearly and sufficiently defined. Although the main objective of this thesis is to develop new algorithms, it seems necessary to provide some foundations of distributed computing before we describe them. In order to make our explanation as concise and precise as possible, we will use point-form in our description throughout this chapter.

In the following discussion we consider the process to be the execution of a program and the processor to be a functional unit by which a process can be carried out. A set of processors communicating to each other via a communication network is called a distributed system (or a multicomputer). Algorithms for distributed systems are known as distributed algorithms.

1.1 Models of Distributed Computations

Although many different models for distributed computations have been suggested in the literature, the practicality of most of these models still remains in question. To model a distributed system, the usual approach is to embed a communication network into a graph in such a way that the vertices of the graph represent the nodes (also called sites, stations or processors through the entire thesis) of the network and the arcs of the graph represent communication links (or channels). Different assumptions may be used in the model; they include:

a) Static or dynamic topology:
Some distributed system models assume a static topology which disallows node/link failures, insertions or recoveries during the course of a distributed computation. Obviously this restriction is not a fair assumption since a node/link in the network can go down at any arbitrary time. Systems allowing topological changes are called systems with dynamic topology. The dynamic environment causes difficulties in updating topological information, routing tables, etc. Currently most distributed algorithms assume a static topology, e.g. the ring, tree, or complete network. In this case, the
algorithm must be rerun whenever a topological change is detected.

b) Different types of communication channels:
Communication channels may be directed (i.e. messages can be transmitted in only one direction) or undirected (i.e. messages can be transmitted in both directions). Message transmission through a communication channel is either in fixed order or random order. In the case of fixed order, various strategies may be used. Two popular ones are:

i) First-In-First-Out (FIFO) - message transmission along a communication channel is serial; this implies that messages will arrive at the destination in the same order that they were sent from the originating processor.

ii) Priority queue - messages along a communication channel are transmitted according to their priorities, regardless of which one arrives first.

c) Synchronous or asynchronous networks:
In distributed systems, information exchanges are accomplished by sending messages from one node to another via communication channels. By message transmission delays, we mean the time needed to transmit a message from one node to a neighbouring node. In synchronous networks, the message transmission delay is uniform among the nodes and links (i.e. messages sent out by a node will arrive at its neighbor nodes at the same time). In asynchronous networks, the delay time is variable but finite.

d) Centralized or decentralized controls:
By the nature of distributed algorithms, an algorithm is executed in all sites concurrently. Each participant processor has a copy of the algorithm. Information updates are accomplished by exchanging messages among processors. If the process is controlled by a single processor, other processors will just follow instructions given by this controller. Such approach is called the centralized control. If the process is controlled jointly by a collection of processors, each processor will play the same role. Processors will take turns in controlling the process. This is called the decentralized control approach.

e) Local and global control information:
According to [Che83] control information is the combination of logical, topological, quantitative, and chronological information derived from the problem, the graph and the processing information. In general, no or limited global control information, i.e. knowledge about the network or graph, is available to each processor. What is available to each processor is the information related to its neighbor nodes, called local control information.

f) Processors' identification:

The most common assumption is that processors have unique identities and do not share common memory.

1.2 Distributed Computing and Parallel Computing

There has been some confusion in understanding distributed computing and parallel computing, even though they are different concepts and used for different purposes. First of all, let us define some terminology using in parallel computing. We call a set of processors which share a common memory parallel computers (or multiprocessors). Algorithms for parallel computers are known as parallel algorithms. The confusion between distributed and parallel computing is mainly caused by a few similar ideas used in these two types of computing models, namely

a) Ideas in centralized distributed systems and single-instruction multiple-data stream (SIMD) parallel computers:

With a SIMD, the central unit issues one stream of instructions which controls all the processors, each operating upon its own memory synchronously.

b) Ideas in decentralized distributed systems and multiple-instruction multiple-data stream (MIMD) parallel computers:

With a MIMD, the processors have independent instruction counters and operate asynchronously on shared memory.

Major differences between parallel computing and distributed computing are outlined below:

a) Processors of a distributed system are geographically distributed and connected by a
communication network, while in parallel computing, processors are usually installed in one physical site.

b) Only local data is allowed in distributed systems, whereas global data is allowed in classical parallel cases.

c) In distributed algorithms the problem is solved under the assumption that processors have no or limited knowledge about each other. Each processor executes the same algorithm and exchanges data and results with other processors by sending messages. In parallel algorithms the problem is divided into subsessions which will be computed in different processors separately. The final answer is obtained by combining the results of these subsessions once they are reported to the shared memory.

d) Due to the direct relation between parallel algorithms and parallel architectures (e.g. parallel sorting machines), the size of hardware is an important complexity measure for parallel algorithms. However this factor is irrelevant to distributed algorithms.

1.3 Important Issues of Distributed Algorithms

1.3.1 Differences between Distributed and Non-Distributed Algorithms

We will summarize the important differences between distributed algorithms and non-distributed algorithms (i.e. serial and parallel algorithms) as follows:

a) For non-distributed algorithms, complete control information is available at one site. For distributed algorithms, no single node in the network has complete information.

b) For non-distributed algorithms, the control of the process is carried out at one site. For typical distributed algorithms, this responsibility is shared by all processors.

c) For non-distributed algorithms, the CPU cost is the criterion for performance measurement. For distributed algorithms, the number of message transmissions is the
most important factor, due mainly to the transmission delays which are relatively large compared to the time spent in local computations. (More of this will be discussed later).

d) For parallel algorithms, processors share a common memory, which is not true for distributed algorithms.

1.3.2 Types of Messages

There are two types of messages: control messages and data messages. Control messages contain control information and serve purposes such as initiating a process, locating a network's component, requesting data, terminating a process, etc. Data messages contain data and serve for the purpose of data exchanges.

1.3.3 Issue of Synchronous and Asynchronous Networks

In reality, the time to execute the steps of a process is unpredictable due to the variations in computation time caused by different inputs, system scheduling policies, variations in the individual process speeds, etc. For these reasons, transmission delays are not uniform. This makes the assumption of asynchronous networks a natural choice. Unfortunately asynchronous behavior leads to a serious issue regarding the correctness of the distributed algorithm. The correctness issue arises because during the execution of an algorithm, operations from different processes may interleave in an unpredictable manner.

Nowadays more and more efforts are directed to the development of computation methods for asynchronous networks. The traditional approach is to simulate synchronous behavior in an asynchronous environment so that the synchronous algorithms can be used. Another approach is to design new algorithms to directly fit in an asynchronous environment. The first approach is the more popular one but the second approach may lead to a better solution.

1.3.4 Issue of Centralized and Decentralized Controls
Algorithms with centralized control are easier to design and involve fewer message transmissions, since useful information can be made available at the controller station. However, algorithms with decentralized control are preferable because of the following:

a) They avoid the 'bottleneck' situations occurring at the controller station, which appear in the centralized case when most of the control message transmissions are concentrated in one single node.

b) They are more reliable, since the whole process does not depend on any single mechanism or any single node, e.g. the controller.

c) They are more flexible since insertions/deletions of new nodes/links do not affect the overall view of the algorithm.

d) They reduce on-site technical expertise since the control task is shared by all processors.

In order to design a decentralized algorithm with a reasonable number of message transmissions, it is justifiable to use centralized techniques in some parts of the algorithm.

1.3.5 Issue of Parallelism

One of the advantages of distributed systems is their parallel computing capability. In order to reduce the computation time of a distributed algorithm, it is important to design the algorithm so as to maximize parallelism. Some interesting observations regarding this issue are listed below:

a) Algorithms with centralized control, in many cases, can normalize the sequence of operations taking place at different processors. This improves the parallelism of the process. A major drawback is the introduction of a longer waiting period from the time a processor finishes the current instruction to the time it receives the next instruction from the controller.

b) Algorithms designed for synchronous networks provide greater parallelism.
c) The network topology affects the parallelism of the computation. For example, finding the maximum of n integers distributed in n nodes takes $O(n)$ time in a linear topology but only $O(\log_2 n)$ time in a binary tree topology.

1.3.6 Issue of Termination

Consider a distributed system of n processors, each of which is either active or passive. An active processor is computing, sending or waiting for a message, and becomes inactive afterwards. A passive processor can become active only by receiving a message. A distributed computation is said to have terminated if all processors are passive, that is, no more computing and no more messages are in transit. Termination detection of a distributed computation is a hard and non-trivial problem because processors are aware of their local states (i.e. local information) but have only limited information about the states of other processes in the network (i.e. global information).

Many termination detection algorithms with various assumptions about the underlying model of distributed computation have been proposed in recent years ([Ma82],[To84],[Fr80],[MiCh82], [DiFeGa83]). The basic idea of these methods is to circulate one or more so-called tokens (i.e. a signal) around the network so that one of the processors (usually the token initiator) can realize if all processors are passive.

1.3.7 Performance Measurement

For serial and parallel computations, the CPU processing cost is the main factor for measuring algorithm performance. However, this is no longer the case in distributed computing since the message transmission delay is variable and lengthy. As a result of this, message (or communication) complexity and traffic complexity are more important factors in the complexity evaluation of distributed algorithms. Message complexity is the number of messages (for both control and data) exchanged during the process. Traffic complexity not only concerns the number of messages, it also takes into the consideration the size of each message (e.g. number of bits). This often leads to local storage problems, i.e. is there enough space in each processor to store the incoming messages?
Finally, time complexity is the total execution time, which is calculated under the assumption that message transmission delay takes one unit of time. In other words, when we calculate the time complexity, an asynchronous network is always treated as a 'synchronous' network. Thus, time complexity is a less important factor in evaluating a distributed algorithm.

1.4 Tradeoffs in Distributed Algorithms Design

To design a distributed algorithm, we have to make certain assumptions about the computational model. These may include message capacity, memory storage at each node, etc. It is important for the designer to recognize the tradeoffs generated by these assumptions. The following example will illustrate how these assumptions could affect the algorithm performance.

Assume we have a distributed system of three nodes, S₁, S₂ and S₃. Any two nodes are connected by a communication link and exchange information by sending messages. We also assume that each message can carry up to \( x \) integers and that each site can store at most \( c \) integers in its local non-shared memory. Initially there are two integers in each node. Our goal is to rearrange these integers so that integers in S₁ are greater than integers in S₂ and S₃.

An easy solution is that S₁ will collect two integers from each of S₂ and S₃, compare them against its own integers, keep the two largest values, and return the smaller integers to S₂ and S₃. To achieve this, four messages are enough. However, one should not overlook the required conditions hidden behind this solution, namely, the message capacity \( x \) to be two integers and the local storage \( c \) to be 6 integers in S₁. With different conditions, the complexity for the same algorithm will vary dramatically. For the case of \( x=1 \) and \( c=4 \), the number of messages needed is 8.

In fact, tradeoffs do not only occur between the capacities and the message complexity, they also occur between the capacities and the time complexity. For the case of \( c \geq 4 \) in our example, S₁ may request data from S₂ and S₃ at the same time. However, if \( c=3 \), then S₁ can only send data requests to S₂ and S₃ one at a time. The given condition in the latter case allows lower parallelism.
of the algorithm and thus results in a longer process time.

The above example shows the effect that different assumptions can create in distributed algorithms design. Other types of tradeoffs exist. In particular, when local computation cost is taken into consideration (as we will show 'why' in Section 2.1), tradeoffs occur between the topological choice and time complexity. A good example of this is the file sorting problem which will be discussed in detail in Section 2.8. We should also point out that in most algorithms, these tradeoffs are often ignored by designers. As a result, they sometimes fail to take advantage of existing assumptions.

1.5 Basic Computational Model and Notation

In this section we will describe the basic computational model and notation used throughout this thesis. Additional assumptions for the model and notation for the sorting problem and shortest path problem will be given in Chapters 2 and 3.

A distributed communication network of size d and capacity c is a set \( T = \{S_1, ..., S_d\} \) of d sites where each site can store c records in its local non-shared memory. (Recall that site, station, node and processor are used interchangeably in this thesis.) We assume that each site has a unique identification number, \( \text{id} \). The underlying graph of \( T \) is an undirected and connected graph \( G(S,E) \) where \( S \) represents a set of nodes and \( E \) a set of edges. The distance between two adjacent nodes is one. Two sites \( S_i \) and \( S_j \) are neighbors if there is a communication link \( (S_i, S_j) \) connecting them. The communication links are undirected and FIFO. Two neighbor stations in the network communicate by sending messages via communication links. We will also assume that each message has capacity x, that is, each message carries a counter, an id or up to x integers. The diameter of the network \( T \), called \( \text{DIAM}(T) \), is the maximum of the distance over any pair of sites of the network.

An almost balanced binary tree network \( T \) (for short an AB binary tree network) is a distributed communication network whose underlying graph is an almost balanced binary tree (see Figure 1.1). In an AB binary tree network, each station \( S_j \) in \( T \) is aware of the id of its father \( F_j \) as well as the id's of left and right children. For every element \( S_j \) in \( T \) we denote \( T_j \) the subtree of \( T \).
induced by $S_j$ together with all its descendants. We will also denote by LN$_j$ the level number of the site $S_j$. For instance in the network shown in Figure 1.1, T$_2$ is the subtree with nodes $S_2$, $S_3$ and $S_5$, and LN$_4$=0, LN$_5$=LN$_2$=1, LN$_1$=LN$_6$=LN$_3$=2, etc.

We define AB $q$-ary tree networks, arbitrary tree networks, and any other rooted tree networks in the same way as described above.

An almost balanced tree network

Figure 1.1
Chapter 2
Tradeoffs in Distributed File Sorting

In this chapter we study the problem of sorting a file of n integers distributed over d stations in a communication network. We present new decentralized algorithms for almost balanced q-ary tree networks. These algorithms can easily be adapted to other topologies. The first algorithm requires extra memory space available at each node to store some temporary values. The second algorithm removes such a requirement but only works on tree networks with postorder labeling. Their message complexities are $O(n/x \log_q d)$ and $O(n/x \log_q^2 d)$ respectively, where $x$ is the message capacity. Tradeoffs between memory requirements and communication complexity are then discussed. Under similar conditions, our algorithms improve existing methods. Finally, we analyze the impact of local processing on time complexity for synchronous networks.
2.1 Introduction

Let \( F = \{a_1, a_2, \ldots, a_n\} \) a large file of \( n \) integers distributed over a communication network of \( d \) sites \( T = \{S_1, S_2, \ldots, S_d\} \). The file sorting problem on \( T \) consists of relocating the elements in \( F \) in such a way that at the end, any element of \( F \) stored in \( S_i \) is smaller than any other element of \( F \) stored in station \( S_j \), \( i < j \).

In this chapter we present two new distributed sorting algorithms based on almost balanced binary tree networks. In the first one, we assume that we have extra memory available at each node of our tree. Under this assumption, a simple Sorting Distributed File algorithm (from now on called SDF) with communication complexity \( O(n \log_2 d) \) is obtained. We will show that this result is optimal. (It is interesting to notice that most existing distributed sorting algorithms implicitly assume extra space available at each site, and yet they fail to take advantage of this fact). The SDF algorithm is similar to Zak's sorting algorithm [Za84], however his algorithm was designed under the assumption that only one integer is stored in each node.

Our second algorithm, called Sorting Distributed File Without extra memory space (SDFW), eliminates the need for extra storage space, and still manages to obtain a very good communication complexity \( O(n \log_2^2 d) \).

The two algorithms presented in this chapter can be easily adapted to work in an arbitrary tree network \( T \) in such a way that if its diameter \( \text{DIAM}(T) \) is \( k \), our SDF and SDFW algorithms have an \( O(k n/x) \) and \( O(k^2 n/x) \) communication complexity respectively, where \( x \) is the message capacity. Notice that in the average \( k \) is approximately equal to \( \log_2 d \). In particular, for almost balanced \( q \)-ary trees, the communication complexities for the SDF and SDFW algorithms are \( O(\log_q d n/x) \) and \( O(\log_q^2 d n/x) \) respectively.

It is customary in distributed computing to ignore the cost of local computations at each site in the network, due mainly to the relatively cheap cost of local computations with respect to the cost of communication activities. Nevertheless in some cases this could lead to misleading conclusions; for instance the communication complexity of an NP-complete problem is polynomial if enough storage space is available at each site.
In the last section of this chapter the impact of local computations in the time complexity of synchronous networks is studied. We will show that in some cases, we cannot ignore the cost of local computations. A specific example concerning sorting in complete synchronous networks is studied in which the cost of local processing has a strong effect in the time complexity of our algorithms. In the algorithms presented in this chapter for almost balanced q-ary trees, when the impact of local computations is taken into consideration, the total complexities of our algorithms (including local and communication costs) will be increased by a factor of $\log_2 q$. 
2.2 Previous Results

In the literature, several algorithms have been proposed to solve the distributed file sorting problem using different assumptions and approaches. In this section we assume that each station originally holds an equal number \( n/d \) of integers. The file sorting problem is to relocate these elements according to some distribution order \( P\{1,2,\ldots,d\} \) so that at the end, any element stored in \( S_{p(i)} \) is smaller than any other element stored in station \( S_{p(i+1)} \).

Wegner first presented an algorithm [We84] which uses centralized control and works only on linear networks. His algorithm is based on quicksort. To simplify its description, we assume that stations in the linear network as well as the distribution order are labeled in ascending order and \( S_d \) is the predetermined controller station.

In the first iteration, the medium of the file elements, called pivot value \( z \), is calculated by \( S_d \) and distributed to other stations. Upon received \( z \), station \( S_1 \) will partition its elements into two sets \( \text{LOW}_i \) and \( \text{HIGH}_i \) so that elements of \( \text{LOW}_i \) are smaller than or equal to \( z \) and elements of \( \text{HIGH}_i \) are greater than \( z \). When the local partition is finished, station \( S_1 \) will report the number of elements in \( \text{LOW}_i \) to the controller \( S_d \). Thus \( S_d \) can determine the split point \( S_p \) in such a way that the total number of elements in \( \{S_1,\ldots,S_p\} \) is equal to \( \sum_{i=1}^{d-1} |\text{LOW}_i| \). Let's denote \( \text{FL}=\{S_1,\ldots,S_p\} \) and \( \text{FR}=\{S_{p+1},\ldots,S_{d-1}\} \). Using \( S_p \) as a reference point, any two adjacent stations can exchange their elements in order to achieve the following: \( \text{FL} \) and \( \text{FR} \) will contain all \( \text{LOW} \)'s and all \( \text{HIGH} \)'s respectively. In other words at this point the file is partitioned into two halves, one half in \( \text{FL} \) and another in \( \text{FR} \). This ends the first cycle. Elements in \( \text{FL} \) and \( \text{FR} \) can be sorted in a recursive way using the same method.

A careful study of Wegner's algorithm shows that his method assumes \( 2d \) extra memory space at each station and \( n/d \) message capacity. The traffic complexity is \( O(n \log_2 d) \) in the average and \( O(n^2) \) in the worst.

Rotem, Santoro and Sidney have designed a decentralized algorithm [RoSaSi85] for completely connected networks based on the \( k \)th selection method. Their algorithm consists of two steps: (1) each station \( S_i \) finds its partition points \( e_{p(i)} \) which in fact is the \( (i+n/d) \)th element and broadcasts this information to other stations, and (2) each station inserts elements to their destinations according to the partition points, i.e. elements fall in between \( e_{p(i-1)} \) (for \( S_{i-1} \)) and \( e_{p(i)} \).
will go into $S_1$. Step one is itself a selection problem which can be solved by any of the existing selection algorithms [Fr83], [Ma83] and [RoSaSi831,2]. The element insertion in step two can be done in a straight forward manner.

This algorithm implicitly assumes an extra memory of $n/d$ and a message capacity of 1. The algorithm has the traffic complexity of $O(d^2 \log_2 d)$ in the average and $O(d^2 \log_2 n) + O(n)$ in the worst.

Cheung has developed a decentralized algorithm [Che88] for completely connected networks. His algorithm is an iterative algorithm in which each station takes turn as the controller according to $P$. Thus in the $i$th cycle of operation, processor $S_{p(i)}$ is the controller. Within one cycle there are a number of iterations. In each iteration, the controller $S_{p(i)}$ uses a mathematical formula to calculate the number of expected elements $u_i$ which has a least value of $n/d$ minus the number of elements in $S_{p(i)}$. Processor $S_{p(i)}$ then sends messages containing its largest element value $r_{p(i)}$ to each of other stations to request for $u_i/(d-1)$ elements whose values are below $r_{p(i)}$. If sufficient elements are collected, $S_{p(i)}$ will transfer the control to the next station $S_{p(i+1)}$ who subsequently starts the $(i+1)$th cycle's operation. Otherwise, $S_{p(i)}$ will recalculate the number of elements $u_i$ and repeat the iteration again. The algorithm terminates when processor $S_{p(n)}$ has collected its expected elements at the end of the $n$th cycle.

Cheung's algorithm is strictly decentralized in a sense that each processor shares the control responsibility and has an equal amount of workload. It has a traffic complexity of $O(n+d^2 \log_2 n)$ in the average and $O(d^2 n)$ in the worst. However in some cases his algorithm requires $n-d^2/2$ extra memory space at each station and $n/d$ message capacity. This happens, for instance, when elements of the file $F$ are initially stored in stations in the reverse order of $P$ such that any element in $S_{p(i)}$ is larger than any other element in station $S_{p(i+1)}$. 
2.3 Model and Notations

In addition to the assumptions described in section 1.5, the following requirements will be used in our sorting algorithms.

a) The networks under consideration are asynchronous networks.

b) When we describe our first algorithm in sections 2.4, we will assume that each message has capacity one ($x=1$), that is each message contains a single value which is a counter, an id, or an integer.

Our second algorithm, as described in sections 2.5, requires the message capacity to be two ($x=2$).

However, the above assumptions will be revised in section 2.6 when we discuss the tradeoffs in these algorithms. In this discussion, if we increase the message capacity to $1 \leq x \leq n/d$, the communication complexity will be substantially decreased.

c) For a rooted tree network $T$, we will assume that the elements $\{S_1, \ldots, S_d\}$ of $T$ have a postorder labeling (see Figure 2.1). Under this assumption, $S_d$ will always correspond to the root site of $T$.

An $\alpha$ distribution of $F$ over the network $T$ is a partition of $F$ into $d$ disjoint subsets $F_1, \ldots, F_d$ such that $F_i$ is stored in $S_i$ and $|F_i| \leq \alpha \leq c$, $i=1,\ldots,d$ and $c$ is the local storage at each site. The distributed file $F$ is said to be sorted in $T$ if for $a \in F_i$ and $a' \in F_j$, $a < a'$ iff $i < j$.

Our main objective in this chapter is to obtain distributed sorting algorithms for AB binary tree networks which produce postorder sorting of $\alpha$ distributed files. Two main cases are considered, in the first case $\alpha \leq c/2$ (i.e. extra space is required at each site), in the second case $\alpha = c-2x$, recall that $x$ is the message capacity. Although our algorithms will be designed for AB binary tree networks, they can easily be modified to work for general rooted tree networks and almost balanced q-ary tree networks.
A file $F=\{1,\ldots,21\}$ stored in a balanced binary tree network with sites labelled in post order form.

Figure 2.1
2.4 Sorting With Extra Memory

In this section, we present our first sorting algorithm (SDF algorithm) for sorting files in AB binary tree networks $T$ of $d$ sites each with capacity $c$. We will make the following assumption: at each site $S_i$ of our AB binary network $|F_i| = \alpha Sc/2$; that is at each site we have at least twice as much memory available as records stored in $S_i$, $i=1,...,d$. Notice that this is a very realistic assumption since in most real life problems extra space is usually available.

The SDF algorithm consists of two phases:

- **Phase 1:** Wake up and initialize.
- **Phase 2:** Sort the elements.

A brief description of each of these phases is now given:

---

**Phase 1:** Wake up and initialization

Initially, all stations in the network are sleeping. One or more stations wakes up. As soon as a station wakes up or is woken up by a WAKE_UP message, it sends a WAKE_UP message to its children and father.

**Phase 2:** Sort the elements

Each station will proceed as follows, depending on whether it is a leaf station, an intermediate station (i.e. neither a leaf or the root), or the root station.

**Leaf stations:**

As soon as a leaf station $S_i$ wakes up, it sends a message SEND_MIN($a_k$) to its father containing the minimum value $a_k$ of $F$ stored in $S_i$.

**Intermediate stations:**

Once a station $S_j$ (not a leaf station or the root station $S_d$) receives an SEND_MIN message from each of its children, $S_j$ will know the value of the smallest element (say $a_u$) stored in the subtree $T_j$ rooted at $S_j$. At this point, $S_j$ will send $a_u$ to its father $F_i$.

**Root station $S_d$:**
When the root station $S_d$ receives an SEND_MIN message from each of its children, it will know the value $a_1$ of the minimum element of $F$ stored in our communication network $T$. At this point $S_d$ will send $a_1$ down to the leftmost leaf station $S_1$ where $a_1$ will be permanently stored.

**Updating the minimum element in an arbitrary station $S_j$:**

As soon as a station $S_j$ sends SEND_MIN($a_u$) to its father $F_j$, it can proceed to calculate the new minimum element stored in $T_j$ (given that $S_j$ is not a leaf station) as follows:

Immediately after $S_j$ sends SEND_MIN($a_u$) to its father $F_j$, $S_j$ will issue a message REQ_MIN to request the next minimum value from the child $S_c$ where SEND_MIN($a_u$) came from. If all integers originally stored at $S_c$ have previously been sent out, $S_j$ will request such a minimum value from another child. Of course, no message exchange is involved if SEND_MIN($a_u$) originated from $S_j$ itself. As a result, the processes in all stations can be kept running in parallel.

Using the same strategy, the values of $a_2$, $a_3$, ... can be determined at the root station. As soon as the root station determines the $i^{th}$ element $a_i$, it directs it towards the node $S_{i/d}$, where $a_1$ will be stored in the extra memory available at $S_{i/d}$. 
(a) Leaf stations send min. to their fathers

(b) Intermediate stations $S_3$ & $S_6$ can now calculate minimums of their subtrees

(c) Sending min. of left subtree and min. of right subtree to the root

(d) Requesting the next min. from $S_1$ & $S_5$ where previous minimums came from

Figure 2.2 (a)-(d) Snapshots of the sorting algorithm SDF (extra memory is assumed)
(e) Root sends the smallest integer $a_i$ down

(f) Root requests next min. from its child $S_6$ while $a_i$ is approaching its destination $S_1$

(g) The next min. is being sent to the root and $a_i$ arrives at its destination $S_1$

(h) Determining the next smallest integer $a_2$ at the root

Figure 2.2 (e)-(h) Snapshots of the sorting algorithm SDF (extra memory is assumed)
(i) Root sends $a_2$ down and then requests the next min. from its child $S_6$

(j) The next min. is sent to the root while $a_2$ is approaching its destination $S_1$

(k) $a_2$ arrives at its destination while update of min. at each level is in process

(l) Root sends $a_3$ down and updating is still in process

Figure 2.2 (i)-(l) Snapshots of the sorting algorithm SDF (extra memory is assumed)
(m) Root requests the next min. from $S_6$ while $a_3$ is approaching $S_1$.

(n) $a_3$ arrives at its destination $S_1$ and the 4th smallest $a_4$ is updated at the root.

(o) The root sends $a_4$ down.

(p) $a_4$ approaches its destination, i.e., station $S_2$, since station $S_1$ is full.

Figure 2.2 (m)-(p) Snapshots of the sorting algorithm SDF (extra memory is assumed)
Termination conditions:

The termination of phase 2 is not hard to determine. We know that a station will terminate the local process after a) it has received all desired integers, and b) it has sent out all the integers which do not belong to itself. Globally, we can determine if the entire process is completed as follow: When a leaf station terminates its local process, it sends a termination message to its father. When a station has completed all sorting tasks and also collected termination notices from all its children, it sends a termination notice to its father. Therefore, once the root has received such notices from all its children, it declares the end of the computation.

2.4.1 Communication Complexity Analysis of the SDF Algorithm

In phase 1 each edge connecting a site $S_i$ to its father $F_j$ transmits at most 2 wake up messages. Thus the communication complexity of phase 1 is $2d$. The communication complexity in phase 2 is at most $3n \log_2 d$. This follows from the observation that before an element of $F$ can reach its destination say $S_j$, it has to go up from its original site $S_i$ to the root site $S_d$ and then come down to $S_j$. In the worst case, a value originating from the bottom level will have to go to the root station $S_d$ and down to the bottom level again. This requires $2 \log_2 d$ messages per element of $F$. Thus, for a file of $n$ records, the message complexity is $2n \log_2 d$. On top of this, before an element moves up one level from a station $S_i$ to its father station $F_j$, a request message has to be sent from $F_j$ to $S_i$. Thus the total communication complexity of the SDF algorithm is $2d + 3n \log_2 d$.

Since a message carries one value, the traffic complexity for SDF is also $2d + 3n \log_2 d$.

To prove optimality of our algorithm, suppose we have a balance tree with $d$ elements with a post order labeling. Assume that the value stored in each site is the label of its mirror image with respect to the root of our tree. It is easy to see in this example that since any value has to move from its original position to the mirror image position, the number of messages required is $O(n \log_2 d)$. 
2.4.2 Time Complexity in the Synchronous Case

In order to calculate the time complexity of the SDF algorithm for synchronous networks, we must assume that a message takes exactly one unit of time to transmit from one station to another adjacent station. Trivially phase 1 takes at most $2 \log_2 d$ units of time.

The time complexity of phase 2 is a bit harder to calculate. First we notice that the root site $S_d$ of our network will receive $a_1$ in exactly $\log_2 d$ time. However, the time needed to calculate each of $a_2, a_3, \ldots$ is constant, since once a site $S_j$ sends a message $\text{SEND}_\text{MIN}(a_j)$ to its father it will immediately request a new minimum from one of its children (as explained in the updating minimum values section of phase 2). It is now an easy matter to verify that after the integer $a_1$ has reached $S_d$, the amount of time needed for each of $a_2, a_3, \ldots$ to reach $S_d$ is at most 2 time units. Thus the time complexity of phase 2 is $\log_2 d + 2n$ and the total time complexity of SDF is $3\log_2 d + 2n$.

The SDF algorithm as presented here works in binary tree networks. In fact, the SDF algorithm can be easily modified to work in any AB q-ary tree network and any labeling of its nodes (not necessarily the post order labeling used in this section) using similar ideas to those presented in [Za84]'s paper. Thus we can obtain the following result:

**Theorem 2.1.** Distributed sorting in AB q-ary tree networks with message capacity of 1 can be accomplished in $2d + 3n \log_q d$ communication and traffic complexity and in $3\log_q d + 2n$ time complexity (in synchronous networks).
2.5 Sorting Without Extra Memory

In this section, we study the distributed sorting problem in a AB binary tree network in which the assumption of extra space is removed. More formally, we assume that each site $S_i$ has capacity $c$ and holds the same number $\alpha = n/d$ of elements of our distributed file $F$ where $\alpha = c - 2x$, $x$ is the message capacity. For the basic discussion of our method we consider $x$ to be 2, that is at each site the set of records $R_i$ stored in $S_i$ occupy most of the memory space available at $S_i$.

The algorithm presented in this section (called the SDFW algorithm) can be thought of as a recursive algorithm. During the first iteration of our algorithm, we will exchange elements of $F$ among the sites of the network in such a way that, at the end of the first iteration:

a) The $\alpha$ largest elements are stored in $S_4$.

b) All the elements of $F$ stored at sites in the left subtree of the root station are smaller than those elements stored in the right subtree. (See Figure 2.4(a)).

The procedure will then be repeated with the subtrees $T_i$ of $T$ with root stations $S_i$ at level 1, 2,... until we reach the leaf stations of $T$. At this point our algorithm terminates. (See Figure 2.4(b)).

(a) Configuration of Fig.2.1 after the first pass   (b) After the second pass the file is sorted

Figure 2.4
Our algorithm again consists of two phases:

Phase 1: Wake up and initialize
Phase 2: Sort the elements.

We proceed now to give description of Phases 1 and 2:

**Phase 1:** Wake up and initialize

Initially, all stations in the network are sleeping. One or more stations wake up. As soon as a station wakes up or is woken up by a WAKE_UP message, it sends a WAKE_UP message to its children and father. Thus all stations in the network will eventually become active. The root station \( S_d \), when it wakes up, will set a counter \( \text{Iterm} \) to 1 indicating the first iteration of the algorithm. This counter will be incremented by 1 in each iteration of computation.

**Phase 2:** Sort the elements

Each station, except the leaf stations, will issue a message REQ_MAX to its children. After this, each station will proceed as follows, depending on whether it is an intermediate station, a leaf station, or the root station.

**Intermediate stations:**

i) If \( S_j \) receives a request message REQ_MAX from its father \( F_j \), two possibilities arise:

a) \( S_j \) is waiting for an answer to a request message from one of its children. In this case, \( S_j \) has to wait until its own request is answered. Once it receives an answer to its request it will proceed as in b).

b) \( S_j \) has received a message SEND_MAX(\( a_l, S_d \)) from its left child and a message SEND_MAX(\( a_r, S_v \)) from its right child (or in the case that it has only one child, a message from its unique child). At this point two cases may occur:

1) If a message PUSH_REQ has arrived at \( S_j \), \( S_j \) will proceed as in ii) b).

2) If no PUSH_REQ has arrived at \( S_j \), \( S_j \) will proceed as in iii).
ii) A message PUSH_REQ\((a_b,S_b)\) from \(F_j\) to \(S_j\) carries an element \(a_b\) which is heading to its destination \(S_b\). \(S_j\) must also send the maximum available element stored in \(T_j\) to \(F_j\) when \(S_j\) receives this message.

If \(S_j\) receives a message PUSH_REQ, two possibilities arise:

a) \(S_j\) is waiting for an answer for a request message from one of its children. In this case, \(S_j\) has to wait until its own request is answered. Once it receives an answer to its request it will proceed as in b).

b) If \(S_j\) has got the maximum elements from its children, two cases may occur:

1) If \(S_j\) is the destination for element \(a_b\), i.e. \(S_b=S_j\), \(a_b\) will be stored in \(S_j\) and then proceed as in iii).

2) If \(S_j\) is not the destination for element \(a_b\), i.e. \(S_b\neq S_j\), \(S_j\) will determine if the element \(a_b\) should be transferred to the left child \(C_{j1}\) or right child \(C_{j2}\) in order to get to its destination \(S_b\). When left route is taken, \(S_j\) compares the left maximum element \((a_l,S_u)\) with \((a_b,S_b)\). If \(a_b\) is greater than \(a_l\), then \(a_b\) should really be the new left maximum. Thus we must update the left maximum by swapping \((a_l,S_u)\) and \((a_b,S_b)\). After this, \(S_j\) will proceed as in iv). The same principle applies to the case in which the right route is taken.

iii) \(S_j\) will determine \(a'=\max\{a_r,a_l,a_j\}\), where \(a_j\) is the maximum available element stored in \(S_j\). Notice that \(a'\) was first originated from the station \(S_w=(\text{either } S_u, S_v \text{ or } S_j)\) and is the largest available element of \(F\) stored in \(T_j\). \(S_j\) will then proceed as in vii). If all elements of \(T_j\) have already been processed, \(S_j\) notifies \(F_j\) that there are no more elements to be processed.

iv) \(S_j\) will determine \(a'=\max\{a_r,a_l,a_j\}\), where \(a_j\) is the maximum available element stored in \(S_j\). Notice that \(a'\) was first originated from the station \(S_w=(\text{either } S_u, S_v \text{ or } S_j)\) and is the largest available element of \(F\) stored in \(T_j\). If left route was taken in ii) b) 2), two cases may occur:
a) If $S_w = S_i$, we will like to push the element $a_b$ down and request the maximum from $C_{j1}$. To do this, $S_j$ will send a message `PUSH_REQ(a_b, S_b)` to $C_{j1}$.

b) If $S_w \neq S_i$, we will just want to push element $a_b$ down without requesting an update maximum from $C_{j1}$. To accomplish this, $S_j$ will send a message `PUSH(a_b, S_b)` to $C_{j1}$.

At this point, $S_j$ will proceed as in vi). If all elements of $T_j$ have already been processed, $S_j$ notifies $F_j$ that there are no more elements to be processed. The same principle applies to the case in which the right route was taken in ii) b) 2).

v) A message `PUSH(a_b, S_b)` from $F_j$ to $S_j$ carries an element $a_b$ which is heading to its destination $S_b$. If $S_j$ receives a message `PUSH_REQ`, two possibilities arise:

a) If $S_j$ is the destination for element $a_b$, i.e. $S_b = S_j$, $a_b$ will be stored in $S_j$.

b) If $S_j$ is not the destination for element $a_b$, i.e. $S_b \neq S_j$, $S_j$ will wait until it has got the maximum elements from both of its children. At this point, $S_j$ will determine if the element $a_b$ should be transferred to the left child $C_{j1}$ or right child $C_{j2}$ in order to get to its destination $S_b$. When the left route is taken, $S_j$ compares the left maximum element $(a_b, S_u)$ with $(a_b, S_b)$. Two cases arise:

1) If $a_b$ is not greater than $a_b$, $S_j$ will retransmit message `PUSH(a_b, S_b)` to $C_{j1}$.

2) If $a_b$ is greater than $a_b$ then $a_b$ should really be the new left maximum. Thus, $S_j$ should push $a_l$ down instead of to push $a_b$ down. To do this, $S_j$ will transmit a message `PUSH(a_l, S_b)` to $C_{j1}$. We must also update the left maximum by swapping $(a_l, S_u)$ and $(a_b, S_b)$.

vi) $S_j$ will send a message `SEND_MAX(a', S_w)` to its father. Immediately after $S_j$ sends $a'$ to its father, $S_j$ will proceed to find the next largest element (if any) stored in $T_j$ by issuing a message `REQ_MAX` to request the next maximum element (if it exists) from the child where $a'$ came from. At this point $S_j$ will wait until an answer (containing the next largest element stored in $T_w$ or a notification that all elements of $T_w$ have been already processed) from this child arrives. Once $S_j$ gets the answer it will be able to calculate the next largest element $a'$.
(if any) stored in $T_j$. At this point, $S_j$ will have to wait until it receives a request message $\text{REQ\_MAX}$ from its father and then proceed as in i).

vii) When $S_j$ receives a message $\text{ASSIGN}(a_p, S_p)$, $S_j$ will store this element in itself permanently if $S_j$ is the destination of $a_p$, i.e. $S_j=S_p$. Otherwise, $S_j$ will retransmit this message $\text{ASSIGN}(a_p, S_p)$ to its left child $C_l$ or right child $C_r$, depending on which route the element $a_p$ should take in order to get to $S_p$.

Leaf station:

viii) As soon as a leaf station $S_j$ receives a request message $\text{REQ\_MAX}$ from its father $F_l$, it will send $\text{SEND\_MAX}(a', S_j)$ containing its next largest element (if any) $a'$ and its own id $S_j$ to $F_l$.

ix) When receiving a message $\text{ASSIGN}(a_p, S_p)$, a leaf station $S_p$ will keep the element $a_p$ permanently. When receiving a message $\text{ASSIGN}(a_p, S_p)$, a leaf station $S_p$ will store the element $a_p$ temporarily.

Root station $S_d$:

x) The root station $S_d$ waits until it receives the largest element $(a_l, S_l)$ stored in its left subtree and the largest element $(a_r, S_r)$ stored in its right subtree together with the id's of the stations where they were originally stored. At this point $S_d$ will be able to determine the largest value $a_n$ of $F$ (it is either one of $a_l$, $a_r$ or the largest element say $a_m$ originally stored in $S_d$) (see Figure 2.5(d)). As in the SDF algorithm, this procedure will enable $S_d$ to determine the largest, second largest, ..., elements of $F$.

xi) For $k=1, 2, ..., n$, once $S_d$ has determined the $k^{th}$ largest element $a_{n-k+1}$, it will process as follows depending whether such element is an $\alpha$ element:

a) For the $\alpha$ elements $a_{n-k+1}, 1 \leq k \leq \alpha$:

Since the $\alpha$ largest elements of $F$ have to be stored in $S_d$, as soon as $S_d$ determines the $k^{th}$ largest element $a_{n-k+1}$ of $F$, $1 \leq k \leq \alpha$, and the site $S_l$ where it came from, $S_l \neq S_d$, it will keep $a_{n-k+1}$ permanently and direct the $k^{th}$ smallest element $a_l$ stored in $S_d$.
towards $S_i$ using a message PUSH_REQ($a_j, S_j$) (see Figure 2.5(e)).

b) For the non-$\alpha$ elements $a_{n-k+1}, \alpha < k \leq n$:

1) If $a_i > a_r$ then we will exchange the positions of $a_i$ and $a_r$ that is, $a_i$ will be directed towards $S_r$ where it will reside for the rest of this iteration (using a message ASSIGN($a_i, S_r$) as in Figure 2.5(h.2)) while $a_r$ will be moved into $S_d$ (using a message PUSH_REQ($a_r, S_r$)). Notice that the location of $a_r$ may change again during the current iteration (see Figure 2.5(h.1-4)).

2) If $a_i < a_r$ then $a_i$ and $a_r$ will return to their original positions (using a message ASSIGN($a_r, S_r$) and $a_r$ will remain in $S_r$ for the rest of the current iteration. $S_d$ will then proceed as in xii).

xii) $S_d$ requests the next largest element available (if any) at the child of $S_d$ where the $k^{th}$ largest element $a_{n-k+1}$ came from (using a message REQ_MAX). Once $S_d$ receives an answer to this request, it will be able to determine the $k+1^{th}$ largest element and the process x) will now be repeated again for $k+1$.

The above steps i) to xii) will be repeated until all elements stored in the right subtree are smaller than those stored in the left subtree, at which point the current iteration will be finished.

To determine the next largest element:

The root station gets two values, one $a_i$ from its left son and one $a_r$ from its right son. Notice that in both the left and right subtrees $a_i$ and $a_r$ determine each a winning path WP$_i$ and WP$_r$ (i.e. the paths from the original sites of $a_i$ and $a_r$ to the root station $S_d$). Using these paths we can now exchange $a_i$ and $a_r$ (if $a_i > a_r$) as described before.

Notice that to determine the second largest, third largest,..., element in the right and left subtrees, we need only to update values along the winning paths WP$_i$ and WP$_r$ of $a_i$ and $a_r$ in the left and right subtrees, thus the extra amount of work needed to obtain next largest element will be at most $2\log_2 d$. 
(a) Initial set up of the SDFW algorithm for the network shown in Figure 2.1

<table>
<thead>
<tr>
<th>j</th>
<th>Dj1</th>
<th>Dj2</th>
<th>Fj1</th>
<th>Fj2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>nil</td>
<td>nil</td>
<td>(←→)</td>
<td>(←→)</td>
</tr>
<tr>
<td>2</td>
<td>nil</td>
<td>nil</td>
<td>(←→)</td>
<td>(←→)</td>
</tr>
<tr>
<td>3</td>
<td>nil</td>
<td>nil</td>
<td>nil</td>
<td>nil</td>
</tr>
<tr>
<td>4</td>
<td>nil</td>
<td>nil</td>
<td>(←→)</td>
<td>(←→)</td>
</tr>
<tr>
<td>5</td>
<td>nil</td>
<td>nil</td>
<td>(←→)</td>
<td>(←→)</td>
</tr>
<tr>
<td>6</td>
<td>nil</td>
<td>nil</td>
<td>nil</td>
<td>nil</td>
</tr>
<tr>
<td>7</td>
<td>nil</td>
<td>nil</td>
<td>nil</td>
<td>nil</td>
</tr>
</tbody>
</table>

(b) When the condition $P_{ji} = D_{ji} = \text{nil}$ is fulfilled, station $S_j$ sends message $\text{REQ}$ to its child $C_{ji}$ to request the max. stored at $C_{ji}$.

Figure 2.5 (a)-(b) Snapshots of the sorting algorithm SDFW (NO extra memory)
(c) After station $S_3$ has received max. from all its children ($P_{31} \neq \text{nil} \& \ P_{32} \neq \text{nil}$), $S_3$ is now in the local process to determine its own max. The same thing occurs in station $S_5$.

(d) The root $S_7$ is in the local process to determine its max. while $S_3$ and $S_6$ are updating their maximums (Notice that no REQ is sent from $S_3$ to $S_1$ since $P_{31} = \text{nil}$)

**Figure 2.5 (c)-(d)** Snapshots of the sorting algorithm SDFW (NO extra memory)
(e.1) After keeping the \( \alpha \)-integer 21, root \( S_7 \) sends \( P_R \) to its left child \( S_3 \) to push its smallest integer 4 towards destination \( S_2 \) (where 21 came from) and also request next max. from \( S_3 \)

(e.2) \( S_3 \) pushes 4 one level down towards its destination (Message \( P_R \) is used here since \( P_{32} = \text{nil} \), i.e. next max. required). Upon received next max. 20 from left child, \( S_7 \) changes \( D_{71} \) to be \( \text{nil} \).

**Figure 2.5 (e)** Snapshots of the sorting algorithm SD FW (NO extra memory)
(f.1) Root \( S_7 \) keeps the second \( \alpha \)-integer 20 and then sends the smallest integer 7 toward its destination \( S_2 \) (where 20 came from).

(f.2) \( S_3 \) pushes integer 7 towards its destination \( S_2 \) (Message P is used here because \( P_{32} \neq \text{nil} \)).

Figure 2.5(f) Snapshots of the sorting algorithm SDFW (NO extra memory)
(g.1) Root $S_7$ keeps the last $\alpha$-integer 19 and then sends integer 9 to station $S_3$.

(g.2) $S_3$ keeps 9, sends current max. 18 up, and requests next max. from $S_1$. (since $D_{31}=P_{31}=\text{nil}$)

Figure 2.5 (g) Snapshots of the sorting algorithm SDFW (NO extra memory)
(h.1) $S_1$ returns the updated max. to $S_3$ while the root $S_7$, which has collected all $\alpha$-integers, is comparing $P_{71}$ and $P_{72}$ to determine the $(\alpha+1)^{th}$ largest integer.

(h.2) In root $S_7$, since left max. $P_{71}=(18,1)$ is greater than right max. $P_{72}=(17,4)$, these two integers are exchanged so that $18$ is sent (in message $A$) to $S_4$ and $17$ is pushed (in message $P_R$) towards $S_1$. (Note that $17$ will be compared to upcoming max. along the path from $S_7$ to $S_1$ in order to determine next left max.)

Figure 2.5 (h.1-2) Snapshots of the sorting algorithm SDFW (NO extra memory)
(h.3) The 4th largest integer 18 is approaching its destination $S_4$ while $S_7$ is requesting from its right child $S_5$ the next max. because $D_{72}=P_{72}=nil$

(h.4) Integer 17, which stopped at $S_3$ on the way to $S_1$, rebounds back because it is greater than the upcoming max., i.e. $D_{31}> (P_{31} \& P_{32})$. Now root $S_7$ has 17 from the left and 15 from the right. On the other hand, integer 18 has reached $S_4$ and will stay there permanently.

Figure 2.5 (h.3-4) Snapshots of the sorting algorithm SDFW (NO extra memory)
(i.1) In root $S_7$, since $P_{71}=(17,1) > P_{72}=(15,4)$, these two integers are exchanged so that 17 is sent (in message $A$) to $S_4$ and 15 is pushed (in message $P_R$) towards $S_1$.

(i.2) Integer 15, which stopped at $S_3$ on the way to $S_1$, replaces 10 as the max. of $S_1$ because $D_{31}>P_{31}$ and $D_{31}<P_{32}$. Integer 10 (instead of 15) will subsequently be pushed towards $S_1$. Meanwhile, the updated max. 16 of $S_3$ is sent to $S_7$.

**Figure 2.5 (i.1-2)** Snapshots of the sorting algorithm SDFW (NO extra memory)
(j.1) Integer 10 is pushed to $S_4$. (Message $P$ is used because $P_{31}$ has filled up so that no request for next max. from $S_1$ is needed). $(-\infty, \ )$ is sent from $S_4$ to $S_6$ since no more integers are available in $S_4$.

(j.2) The 6th largest integer 16 is directed to $S_4$ and integer 13 is pushed towards $S_3$.

Figure 2.5 (j.1-2) Snapshots of the sorting algorithm SDFW (NO extra memory)
(k.1) The 7th largest integer is being determined in S7

(k.2) The 7th largest integer 15 is directed to S6

Figure 2.5 (k.1-2) Snapshots of the sorting algorithm SDFW (NO extra memory)
(1.1) The 8th largest integer is being determined in $S_7$

(1.2) The 8th largest integer 14 is directed to $S_5$

Figure 2.5 (1.1-2) Snapshots of the sorting algorithm SDFW (NO extra memory)
(m.1) The 9th largest integer is being determined in \( S_7 \)

(m.2) The 9th largest integer 13 is directed to \( S_6 \)

Figure 2.5 (m.1-2) Snapshots of the sorting algorithm SDFW (NO extra memory)
(n.1) The 10th largest integer is being determined in $S_7$

Figure 2.5 (n.1-2) Snapshots of the sorting algorithm SDFW (NO extra memory)
(o.1) The 11\textsuperscript{th} largest integer is being determined in $S_7$

Figure 2.5 (o.1-2) Snapshots of the sorting algorithm SDFW (NO extra memory)
(p.1) The 12th largest integer is being determined in S7

(p.2) The 12th largest integer 10 is directed to S5

Figure 2.5 (p.1-2) Snapshots of the sorting algorithm SDFW (NO extra memory)
(q) The end of exchanging integers when all integers in $S_7$'s left subtree are smaller than integers stored in $S_7$'s right subtree, thus $P_{72}$ is set to $(-\infty, \infty)$.

(r) At this point, a station $S_j$ on the left side of $S_7$ will in turn send whatever is stored in $P_{ji}$ back to where it was sent from. Here $S_7$ sends $P_{71}$ (integer 9) back to $S_3$.

Figure 2.5 (q)-(r) Snapshots of the sorting algorithm SDFW (NO extra memory)
(s) $S_3$ sends $P_{31}$ (integer 8) back to $S_1$ and $P_{32}$ (integer 7) back to $S_2$

(t) End of the first iteration

Figure 2.5 (s)-(t) Snapshots of the sorting algorithm SDFW (NO extra memory)
To sort the whole file:

As we have seen, the first $n/2$ largest elements can be determined and placed in the right subtree by the end of the first iteration. Since the root station $S_4$ now contains the $\alpha$ largest elements, it can retire from further operations. Although the elements stored at sites in its left subtree are smaller than those elements stored in its right subtree, they still may not be in their final destinations yet.

We now proceed as follow until the counter $Iternum \geq$ the diameter of the tree $(\log_2 d) - 1$:

We first increment the counter $Iternum$ by 1. For the $Iternum^{th}$ iteration we repeat the same procedure described above on the subtrees whose root nodes are located on the $Iternum-1^{th}$ level. It is obvious that in the $Iternum^{th}$ iteration there are $2^{Iternum-1}$ subtrees (see Figure 2.6). Thus for these subtrees, when the current iteration ends the elements stored in their left sub-subtrees are smaller than those stored in their right sub-subtrees. In addition, the roots of the current iteration will retire from further operations.

If we repeat such an operation on all subtrees, level by level, the elements will eventually be sorted in the final order. The termination of SDFW algorithm can be determined in the same way as the termination of SDF algorithm described in previous section.

![Diagram](image)

Figure 2.6

(a) There is one tree at the beginning of the first iteration

(b) There are two subtrees at the beginning of the second iteration
2.5.1 Communication Complexity Analysis of the SDFW Algorithm

In Phase 1 each edge connecting a site $S_i$ to its father $F_i$ transmits at most two wake up messages. This requires $2d$ messages. The operation of each leaf station sending its maximum element to its father involves $d/2$ messages. Therefore the communication complexity of Phase 1 is $2^{1/2}d$.

Now let us look at the most important Phase 2 - 'Sorting the elements'. It is interesting to see that its message complexity is independent of the number of subtrees in the graph. In fact, the message complexity is determined by the number of elements being moved from one subtree to another subtree during the process.

In the first iteration, there are two subtrees. Each involves at most $n/2$ element movements so that the total element movements for the first iteration is $n$. In the second iteration, there are four subtrees. Each involves at most $n/4$ element movements so that the total element movements for the second iteration is still $n$. In general, in the $i^{th}$ iteration, there are $2^i$ subtrees with each involving at most $n/2^i$ element movements. Thus the total element movements for one iteration is always at most $n$. In reality, the element movement means to find the next largest element. Since finding one next largest element needs $2\log_2 d$ messages (as explained previously), it takes a total of $2n \log_2 d$ messages to complete $n$ element movements, i.e. one iteration. Since there are $\log_2 d$ iterations (levels), the total message complexity for phase 2 is $2n \log_2^2 d$.

A detailed mathematical calculation for the complexity of Phase 2 is

\[ \sum_{j=1}^{\log_2 d} \sum_{i=1}^{\log_2 (d/2^j)} ((n/2^i) \log(d/2^{j+i-1})). \]

Thus, the overall communication complexity of the SDFW is $2^{1/2}d + 2n \log_2^2 d$. Since a message carries one value, the traffic complexity for SDFW is also $2^{1/2}d + 2n \log_2^2 d$.

2.5.2 Time Complexity in the Synchronous Case

In synchronous networks a message takes exactly one unit of time to transmit from one station to another adjacent station. Trivially Phase 1 takes at most $2 \log_2 d + 1$ units of time.
We will now analyze the time complexity for Phase 2. As we mentioned before, Phase 2 can be thought of as a recursive procedure. When Phase 2 is executed, this procedure will be in turn applied on the subtrees rooted at the level 0, then on the subtrees rooted at the level 1, ..., and finally on the subtrees rooted at the highest level. During the execution, each level is considered as one iteration. Since the subtrees with roots located at the same level execute this procedure simultaneously, they take the same amount of time to complete this procedure. Thus, although there are \( d/2 \) subtrees in an AB binary tree network, the number of iterations is only \( \log_2 d \) (which equals the number of levels). Recall that the time complexity for one iteration in Phase 2 is \( 3\log_2 d + 2n \) (as previously explained in the time complexity for the SDF algorithm). Consequently, the time complexity for Phase 2 of SDFW is \( 3\log_2^2 d + 2n \log_2 d \) and the total time complexity for the SDFW algorithm is \( 3\log_2^2 d + 2(n+1) \log_2 d + 1 \).

The SDFW algorithm as presented here works in binary tree networks. In fact, the SDFW algorithm can be easily modified to work in any AB \( q \)-ary tree network with nodes in postorder labeling. Thus we can obtain the following result:

**Theorem 2.2.** Distributed sorting in AB \( q \)-ary tree networks without extra memory available at each site and with message capacity of 1 can be accomplished in \( 2^{1/2} d + 2n \log_q^2 d \) communication and traffic complexity and in \( 3\log_q^2 d + 2(n+1) \log_q d + 1 \) time complexity (in synchronous networks).
2.6 Tradeoffs between Extra Memory and Complexity

As we have seen in the previous sections, the SDF algorithm requires extra local memory at each site to store some temporary values, but in return it is \( \log_2 d \) times faster and needs \( \log_2 d \) times fewer messages than the SDFW algorithm. It is not hard to verify that, if one is willing to sacrifice more local storage and increases the message capacity, the performance of the SDF algorithm can be further improved. The tradeoffs between the memory storage and complexity, as we apply the SDF algorithm in an AB binary tree network, is summarized as follow:

<table>
<thead>
<tr>
<th>Message Capacity</th>
<th>Extra Storage</th>
<th>Message Complexity</th>
<th>Time Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>* 1 element</td>
<td>n/d+2 elements</td>
<td>O(n \log_2 d)</td>
<td>O(n + \log_2 d)</td>
</tr>
<tr>
<td>2 elements</td>
<td>n/d+4 elements</td>
<td>O(n/2 \log_2 d)</td>
<td>O(n/2 + \log_2 d)</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x elements</td>
<td>n/d+2x elements</td>
<td>O(n/x \log_2 d)</td>
<td>O(n/x + \log_2 d)</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>** n/d elements</td>
<td>2n/d elements</td>
<td>O(d \log_2 d)</td>
<td>O(d + \log_2 d)</td>
</tr>
</tbody>
</table>

This summary also shows that under the similar conditions our method manages to improve the existing algorithms. (Compared * to [RoSaSi85] and ** to [Che86].) Extra local memory storage is proved to be a useful strategic tool in distributed algorithms design.
2.7 Impacts of Local Computation Cost

Given a completely connected network, the distributed file sorting problem can be solved in three steps:

i) Elect a leader.
ii) Use the leader as the root to construct a q-ary tree.
iii) Apply the algorithm SDF or SDFW.

Since our algorithms SDF and SDFW work on AB q-ary tree networks, we are interested in knowing how the choice of q affects the computing performance. Let us take the two extreme cases for comparison.

Case 1: \( q = d - 1 \)

In this case our q-ary network is a star tree. For the SDF algorithm, this configuration has the advantage of minimizing the message complexity. The message complexity in this case is \( O(n) \). Furthermore, if we ignore the time spent performing local calculations (as customarily done in distributed computing) we will also get a linear time complexity for synchronous networks.

Case 2: \( q = 2 \)

In this case we obtain binary trees which have been fully discussed in previous sections. In this case the message complexity is \( O(n \log_2 d) \) and time complexity is \( O(n) \). Evidently the communication complexity is worse than that of the star network and the time complexity in this case is also linear.

This leads us to correctly believe that the best strategy to optimize the communication complexity of distributed sorting in complete networks will be achieved by choosing a star network. For the time complexity of synchronous networks, however, a subtle mistake in the previous conclusion would lead us to believe that both of the star networks and binary tree networks are equally fast.

A more careful analysis of the time complexity for synchronous networks will reveal an
obvious mistake. When the root station $S_d$ determines the largest, second largest, ... element, it has to select it from among $q$ candidates (one from each child). Clearly this cannot be achieved in constant time. Thus the real time complexity in this case is $O(n \log_2 d)$ as opposed to the linear time complexity which would result from the typical analysis for synchronous networks.

One fact hidden behind the file sorting problem and never before considered is the local processing time. This local processing time becomes a noticeable factor in our study of sorting file in the tree networks. In the above example, the root processor in the star network could have much longer local processing time (i.e. the queuing time at the message arrival port plus the local sorting time) than the one in the binary network if we are dealing with a very large network and/or a very large file. In more specific, when the SDF is applied, this long local processing time for the star network requires $\log_2(d-1)$ unit of local time to find its minimum while for the binary tree network it requires exactly two comparisons.

We are going to investigate the impact of local processing on the total cost of time in the case of SDF algorithm. Let the symbol $g$ represent a very tiny value such that 1 unit of local processing time is equal to 1 unit of communication time multiplied by $g$. Also let $t_1$ and $t_2$ be the total time cost for star and binary tree networks, respectively. We obtain the following results:

\[
q = d - 1 \quad \Rightarrow \quad t_1 = g(n - d) \log_2 d + n - d \\
q = 2 \quad \Rightarrow \quad t_2 = (g + 1) \log_2 d + (n - 3)(g + 1) + \log_2 d
\]

Thus, $t_1 < t_2$ iff $g < (2\log_2 d + d - 3) / ((n - d - 1) \log_2 d - n - 3)$.

This result indicates that it needs shorter time to sort a file in a star network than in a binary tree network if and only if the constant $g$ is smaller than some value. However, as the value of $n$ (the number of elements) and/or the value of $d$ (the number of stations) gets larger and larger, this condition becomes harder and harder to be satisfied. This shows that for sorting a very large file distributed over a very large network, the local processing could hurt the time performance.

In the general case of applying the SDF in the AB $q$-ary tree networks, the first element takes $\log_q d$ communication time and $\log_q d \times \log_2 q$ local processing time (for selecting the local minimum) to arrive at the top, while each of the remaining $(n-q-1)$ elements takes 1 unit of
communication time and $\log_2 q$ local processing time. The last element to come down from the root station to its destination requires $\log_q d$ communication time but no local processing time. Thus, the total time complexity is $g(\log_q d + n - q - 1) \log_2 q + (n - q - 1 + 2\log_q d)$. Using this general formula, we can actually calculate the total time complexity, including the external and internal processing, for different values of $q$. From these results we can choose a value of $q$ which meets the design requirement most closely.
2.8 Closing Remarks

Before closing the chapter, several final remarks should be made.

First, do our algorithms cause a bottleneck at the root node? Not really! The computation load is evenly distributed at each participating node in each cycle, i.e. no one specific node has heavier work load than the others, in spite of the fact that the root has to participate in all cycles during the entire process. Besides, our algorithms provide a great deal of parallelism. This compensates the possibly longer waiting time at the root node.

Second, if the message capacity $x$ is assumed to be greater than 1, our algorithms SDF and SDFW will be $x$ times faster and need $x$ times fewer messages. This has been shown in Section 2.6.

Third, our SDF algorithm (for sorting with extra memory) can be modified so that it will work on tree networks of any labeling. This modification requires that the root station $S_d$ has knowledge of the distribution order $P$. Therefore $S_d$ can send assigned elements directly to their destinations.

Fourth, the proposed methods use some blind operations which cannot recognize the already-sorted inputs. However, this shortcoming could be improved by sending both of the minimum and maximum elements in a station to its father. Thus, at the end, the root station can compare the maximum element of the left subtree and the minimum element of the right subtree. If the former is smaller, then all elements in the left subtree are smaller than those in the right subtree. The root keeps the largest elements and will retire from the further actions. The original tree is then divided into two subtrees, each being operated with the same algorithm. Applying this technique repeatedly in AB binary tree networks, already-sorted inputs can be detected in $O(\log_2^2 d)$ time with $O(d \log_2 d)$ messages.
Chapter 3
Tradeoffs in Distributed Shortest Path Finding

In recent years there has been a great deal of studies on the tradeoffs between the message and time complexities in distributed algorithm design. Research shows that the higher the number of messages used, the shorter the time complexity will be, and vice versa. For instance, election in a ring can be accomplished with $O(n)$ messages with a long waiting time. However, if we want to complete it in $O(n)$ time, $O(n^2)$ messages will be needed. On the other hand, one could also solve the election problem with $O(n \log n)$ messages and $O(n \log n)$ time complexity [Pe82].

In this chapter we study the tradeoffs between time and message complexities for the problem of finding a shortest path between two arbitrary stations $S_m$ (source) and $S_n$ (sink) in distributed and synchronous communication networks. In particular we focus on the termination detection problem. Based on Dijkstra's method our algorithm performs search and trail operations. During the search process search messages fan out from the source and start building a spanning tree until $S_n$ is reached. Once the sink has received the search message it starts the trail process, in which the sink sends a trail message backwards from $S_n$ to $S_m$ in order to stop all outgoing search messages as soon as possible. To accomplish this, before a full spanning tree is built, a delay is introduced at each node which has received a search message, that is after receiving a search message a node will retain the search message for some specified amount of time before sending it to its neighbours. Thus the search message does not always traverse the entire network like a straightforward implementation of Dijkstra's algorithm would do.

Our algorithm works in arbitrary synchronous networks. However, in this paper we will analyze its message and time complexities only on the tree, linear and ring networks. Further studies are needed to determine the complexities of our algorithm on other network topologies.
3.1 Introduction

Shortest-path problems are some of the most fundamental and commonly encountered problems in the study of communication networks. While there is a large class of shortest-path problems, in this chapter we will confine ourselves to the basic problem of how to determine a shortest path $SP_{m,n}$ between two specific nodes $S_m$ (source) and $S_n$ (sink) in a communication tree network $T$.

One of the solutions for the above problem in the non-distributed environment was due to Dijkstra in 1959. The basic idea behind Dijkstra's algorithm is to fan out from $S_m$ and proceed towards $S_n$ to construct a tree. The algorithm works as follows: All nodes are initially unlabeled. We start by labeling the source $S_m$. Then we label each immediate successor $S_v$ of source $S_m$. If none of these $S_v$'s is the sink $S_n$, the algorithm will proceed to label the immediate successor(s) of each $S_v$. However the labeling request to a node say $S_w$ will be rejected if $S_w$ has already been labeled (or visited). We continue processing in this fashion until the sink $S_n$ is finally reached. The actual shortest path $SP_{m,n}$ can be obtained by tracing backward from $S_n$ to $S_m$. It is quite easy to see that Dijkstra's algorithm generates a spanning tree rooted at the source $S_m$, lets call it source-rooted tree. Since the process is stopped as soon as the sink $S_n$ is reached, the tree constructed by Dijkstra's algorithm is not necessarily a spanning tree of the network.

Dijkstra's method can be adapted in a straight forward manner to solve the same problem in distributed networks in the following way: A labeling request here means a search message which carries the id of the sink $S_n$. The source $S_m$ broadcasts a search message. Upon receiving a search message, a node determines whether to broadcast this message or not, depending on if this node has already been labelled (or visited). Since in distributed networks there is not a central controller, no node can know what happens to other nodes. Thus the algorithm will not stop until all the nodes have been visited, i.e. a spanning tree of the network is generated.

Unfortunately, there is a major drawback of this approach: it always takes the same (maximum) amount of messages regardless of where the sink is located. This is caused by the fact that (in distributed environment) according to the algorithm a complete source-rooted tree of the network must be generated no matter how soon the shortest path is obtained. Such drawback is highly undesirable for the communication complexity (i.e. the number of messages exchanged) is our major concern.
Dijkstra's method for finding the shortest path between two nodes in distributed networks requires $O(d)$ units of time in the worst case (i.e. in linear topology) and $O(d^{1/2})$ units of time in the average case (i.e. in random tree topology, see [MeMo70]), where $d=|G|$ is the number of nodes. And it takes $O(|E|)$ number of messages.

In this chapter we will develop a new distributed algorithm to solve the shortest path problem. Our algorithm, called Finding Shortest Path (FSP), is based on Dijkstra's method but eliminates the burden of having maximum communication complexity at all times. The basic idea of the FSP algorithm is delay and trail. Once the sink node $S_n$ is reached, $S_n$ will issue a trail message tracing back from $S_n$ to $S_m$ to stop the outgoing search messages. Meanwhile, after receiving a search message, a node $S_w \neq S_n$ will wait for a certain amount of time (determined by a predefined delay function $W(S_w)$). When the delay time is up, $S_w$ will send the search message to its successors and then remain idle. If an idle node $S_w$ receives a trail message from another node say $S_v$, $S_w$ will immediately transmit the trail message to all its neighbours in the tree except $S_v$ and then terminate its own process. If $S_w$ receives a trail message during the delay time, it will terminate the process of $S_w$ at once.

Clearly, if the delay time is long enough, the entire FSP process could be terminated at the level where the sink $S_n$ is located. We will analyze this situation in section 3.4. This also leads us to a new observation: altering the delay function $W$ can change the amount of messages required in the process. Thus, the greater the delay time is, the smaller the amount of messages needed will be. In sections 3.5, the tradeoffs between the time and communication complexities of our algorithm will be studied.
3.2 Model and Notations

In addition to those as described in Section 1.6, the following model assumptions will be used in this chapter.

a) The networks under consideration are synchronous networks.
b) Each site has a timer which can be reset to some specific value at any arbitrary time. There is no mutual drift between timers in different sites. In synchronous networks, one unit of local time is equivalent to one unit of global time.
c) The network size $d$ is made public to all stations.
d) Each search message has fixed capacity for containing the id of the sink node. Each trail message has capacity of one bit for signaling that the sink is reached.
e) The stations $\{S_1, ..., S_d\}$ of $T$ have a random ordering.

During the execution of the FSP algorithm, $S_j$ will also keep track of two sets of information: the predecessor $\text{PRE}_j$ from where $S_j$ receives the search message, and a set of successors $\text{SUC}_j$ to where $S_j$ sends the search messages.

A source-rooted tree network of the network $T$ (for short called SR-T network) is an ordinary tree which is rooted at the source node $S_m$ and orients each edge away from the root (see Figure 3.2). As described in the introduction, this SR-T can be generated by an exhaustive execution of Dijkstra's algorithm (i.e. until every node in the network has been visited). For the sake of simplicity, we will conduct the analysis of our FSP algorithm directly on the SR-T network. Note that this approach is a very reasonable one since the SR-T not only represents the network $T$ but is actually generated during the execution of the algorithm.

For every element $S_j$ in the SR-T the father of $S_j$ is its predecessor $\text{PRE}_j$ and the children of $S_j$ are its successors $\text{SUC}_j$. In our algorithm, the $W(\text{LN}_j)$ function represents the amount of time for a search message to be delayed at a site $S_j$ located at the $\text{LN}_j$th level, before being sent to the adjacent sites. In this chapter we will choose $W(\text{LN}_j) = \omega(i)$ and assume the predetermined constant $\omega$ to be publicly known.
Figure 3.1  A connected network T

The source-oriented tree network SR-T of the network shown in Figure 3.1

Figure 3.2
3.3 Finding the Shortest Path in Synchronous Networks

In this section we present our search algorithm (FSP algorithm) for finding a shortest path $SP_{m,n}$ between a source station $S_m$ and a sink station $S_n$ in a synchronous network $T$ of $d$ sites.

Our algorithm starts by sending a search message, which contains the id of the sink node $S_n$, to each immediate successor $S_v$ of source $S_m$. A node which has received a search message is said to be 'visited'. If none of these $S_v$'s is the sink, the algorithm will proceed to search the immediate successor(s) of each $S_v$. However the search request to a node, say $S_w$, will be rejected if $S_w$ has already been visited. We continue processing in this fashion until the sink $S_n$ is finally reached. During the search process, message transmission delay occurs at visited nodes. Let $r$ be the distance between the source (root) $S_m$ and sink $S_n$; $r$ is called the target distance (see Figure 3.3). Once the sink $S_n$ is found, $S_n$ will issue a trail message tracing backward from $S_n$ to $S_m$ and propagating along the tree currently traversed by the algorithm. The trail message is intended to reach the leaves before more search messages are being sent out from other nodes. Let $t$ be the maximum trail distance, i.e. the distance between the level of the sink node and the level at which a search message is stopped. We will discuss in Section 3.5 how to define such $t$ in terms of the predetermined delay function $W$ and the level number of the sink $LN_n$. If we look at the FSP algorithm more carefully, we will find that for $r \geq DIAM(T) - t$ search messages must traverse the whole network. In this case the trail process will be unnecessary.

![Diagram](image)

Display of the target distance $r$ and trail distance $t$

**Figure 3.3**
A formal description of our FSP algorithm now follows:

i) Initially, all stations in the network $T$ are sleeping. The source station $S_m$ wakes up and sends a search message $\text{SEARCH}(\text{LN}_m, S_n)$ containing its level number $\text{LN}_m = 0$ and the id of the sink station $S_n$ to its adjacent stations. Upon receiving a $\text{SEARCH}(e, S_n)$ message from its adjacent station $S_b$, a sleeping station $S_j$ will wake up and proceed as in ii).

ii) If $S_j$ is a leaf it will terminate its own process. Otherwise, $S_j$ will obtain the information of its predecessor by $\text{PRE}_j = S_p$, its successors by $\text{SUC}_j = \{ \text{all } S_j \text{'s adjacent nodes excluding } S_i \}$ and its level number $\text{LN}_j = \epsilon + 1$. Then $S_j$ will compare its own id to $S_n$ and act according to the following:

a) If $S_j$ is not the sink node it will proceed as in iii).

b) If $S_j$ is the sink node but $\text{LN}_j < \text{DIAM}(T) - t$ it will proceed as in iv).

c) If $S_j$ is the sink node and $\text{LN}_j \geq \text{DIAM}(T) - t$ it will terminate its own process.

iii) $S_j$ will wait for $W(\text{LN}_j)$ units of time. During the delay if $S_j$ receives a trail message, $S_j$ will terminate its own process and perform no further action. Otherwise, when the delay timer expires, $S_j$ will perform the following:

a) $S_j$ will send $\text{SEARCH}(\text{LN}_j, S_n)$ to its successors $\text{SUC}_j$ and then remain idle. Upon receiving $\text{SEARCH}(e, S_n)$ each of $\text{SUC}_j$ will proceed as in ii).

b) Once the idle $S_j$ receives a trail message it will become active again and proceed as in iv).

iv) The trail message $\text{TRAIL}$ is used to inform other stations that the sink station has been reached and thus stop the outgoing search messages. Thus station $S_j$ will send a $\text{TRAIL}(1)$ message with a bit 1 to its predecessor $\text{PRE}_j$ and $\text{TRAIL}(0)$ messages with a bit 0 to its successors $\text{SUC}_j$. After sending message(s) $S_j$ will terminate its own process. Note that the path on which all nodes have received $\text{TRAIL}(1)$ is actually the shortest path $SP_{m,n}$.

The entire FSP algorithm will terminate when all participating sites in the network have terminated their processes. The termination detection can be done as described in Chapter 2.
3.4 Complexity Analysis of the FSP Algorithm with the Longest Delay

In this section we will analyze the message and time complexities of our FSP algorithm in particular for the delay function $W(LN_i)=2i+1$. Since in the $LN_i$ iteration of our algorithm, we build a tree $T$ of depth $LN_i$, the distance between two nodes in $T$ is at most $2i$; thus any node in $T$ can be reached from any other nodes in $T$ in at most $2i$ time using only edges in $T$. Then if we set our delay function $W(LN_i)=2i+1$, $S_n$ will have enough time to inform any other nodes in $T$ that that a shortest path has been established, and no further search message will be needed. In other words the trail distance $t$ is zero.

For every $S_i$ in $T$, let $T(S_i)$ be the number of messages (called total message) used to calculate the shortest path between $S_m$ and $S_i$, $i=1,2,...,d$, $i\neq m$. The average number of messages (called average message) needed to compute the shortest path between $S_m$ and a randomly chosen node $S_i$ in $T$ is $\sum_{i=1,d}^{d}T(S_i)/(d-1)$. We define total time and the average time in the similar manner.

Clearly as the delay function $W$ decreases, the time complexity of our algorithm will decrease while the message complexity increases. This relationship will be studied in Section 3.5.

3.4.1 For Random Tree Networks

In this section, we analyze our algorithm in the case when our communication network is a random tree.

According to [MeMo70] the expected average distance between two arbitrary points in a random tree $T$ is $E(H,T)=(\pi d/2)^{1/2}$ and the expected average degree of the nodes is $E(D,T)=2(1-1/d)$. It is clear that, for the delay function $W(LN_i)=2i+1$, the message complexity of our FSP algorithm is the number of nodes in $T$ at distance at most $E(H,T)$ from $S_m$. Since the number of edges between the level $0$ and level $1$ of the tree is $E(D,T)$ and that between levels $i$ and $(i+1)$ is $E(D,T)[E(D,T)-1]^i$, $i=1,2,...,E(H,T)$, we can calculate the average message complexity as follows:
Search Messages\textsubscript{random} = b \left[q + \frac{\sum_{i=1}^{e} q(q-1)^i}{1}ight]

= b \left[q \left(1 + \frac{\sum_{i=1}^{e} (q-1)^i}{1}\right)\right]

where \(e = E(H,T) = (\pi d/2)^{1/2}\) and \(q = E(D,T) = 2(1-1/d)\) and \(b = \) number of bits required to send a node id.

We will calculate the summation by substituting \(q=2(1-1/d)\):

\[
\sum_{i=1}^{e} (q-1)^i = \sum_{i=1}^{e} (2-2/d-1)^i
\]

\[
= \sum_{i=1}^{e} (1-2/d)^i
\]

\[
< \sum_{i=1}^{e} (1) = e = (\pi d/2)^{1/2}
\]

Thus the search message complexity is:

Search Messages\textsubscript{random} < 5 \left[2(1-1/d) \left(1 + (\pi d/2)^{1/2}\right)\right] = b \left[2+(2\pi d)^{1/2}-2/d-(2\pi/d)^{1/2}\right] \text{ bits}

Since the number of Trail messages is equal to the number of Search messages and each Trail message is of constant size, then the total communication complexity of the trail process is

Trail Messages\textsubscript{random} < 2+(2\pi d)^{1/2}-2/d-(2\pi/d)^{1/2} \text{ bits}

\[
\therefore \text{Average Messages\textsubscript{random} = Search Messages + Trail Messages}
\]

\[
< (b+1) \left[2+(2\pi d)^{1/2}-2/d-(2\pi/d)^{1/2}\right] \text{ bits}
\]

Then the total time taken by our algorithm equals the message transmission time plus the message delay time. Let us take an instance from Figure 3.2: the source \(S_9\) sent a search message to the sink \(S_{18}\) through node \(S_{20}\) and \(S_8\). In this case the total time includes 3 time units for transmission plus the delay time in nodes \(S_{20}, S_8\) and \(S_{18}\). In general, the time complexity of our algorithm is the height \(E(D,T)\) plus the delay time which is \(\sum_{i=1}^{e} E(H,T)(2i+1)\).

\[
\therefore \text{Average Time\textsubscript{random} = e + \sum_{i=1}^{e} (2i+1) where } e = E(H,T) = (\pi d/2)^{1/2}
\]

\[
= 2e + 2 \sum_{i=1}^{e} c_i
\]

\[
= 2e + e(e+1)
\]

\[
= e^2 + 3e
\]

\[
= \pi d/2 + 3(\pi d/2)^{1/2}
\]
**Theorem 3.1** The average communication and time complexities of finding a shortest path between two arbitrary points in distributed and synchronous random tree networks are \((b+1) [2+(2\pi d)^{1/2}-2/d-(2\pi/d)^{1/2}]\) bits and \(\pi d/2 + 3(\pi d/2)^{1/2}\) units of time respectively, with the delay function \(W(LN_i)=2i+1\), where \(b\) is the number of bits required to send a node id.

### 3.4.2 For Linear Networks

The linear network is the simplest topology for analysis. Since there is only one node on each level of the linear network, the number of messages needed to go from the source \(S_m\) to the sink \(S_i\) is \(\text{LEVEL}(i), i=m\); and it requires a time complexity of \(i+\Sigma_{i=1}^{\text{LEVEL}(i)}(2i+1)\) for the delay function \(W(LN_i)=2i+1\). In the following, we will calculate the average message and time complexities for our FSP algorithm on linear networks:

- **Total Search Messages**
  \[
  \text{Total Search Messages}_{\text{linear}} = b \sum_{i=1}^{d-1} i
  = b(1+2+3+...+d-1)
  = b(d(d-1)/2) \text{ bits}
  
  \text{where } b = \text{the number of bits required to send a node id}
  
  \]

- **Total Traill Messages**
  \[
  \text{Total Traill Messages}_{\text{linear}} = \sum_{i=1}^{d-1} i
  = d(d-1)/2 \text{ bits}
  
  \]

- **Total Messages**
  \[
  \text{Total Messages}_{\text{linear}} = (b+1) [d(d-1)/2] \text{ bits}
  
  \]

- **Average Messages**
  \[
  \text{Average Messages}_{\text{linear}} = (1/(d-1)) \text{ Total Messages}_{\text{linear}}
  = (b+1) d/2 \text{ bits}
  
  \]

- **Total Time**
  \[
  \text{Total Time}_{\text{linear}} = \sum_{i=1}^{d-1} [\Sigma_{j=1}^{i} (2j+1) + i]
  = \sum_{i=1}^{d-1} [i^2+3i]
  = d(d-1)(2d-1)/6+3d(d-1)/2
  = d(d-1)(d+4)/3
  
  \]

- **Average Time**
  \[
  \text{Average Time}_{\text{linear}} = (1/(d-1)) \text{ Total Time}_{\text{linear}} = d(d+4)/3
  
  \]
**Theorem 3.2.** The average communication and time complexities of finding a shortest path between two arbitrary points in distributed and synchronous linear networks are \((b+1)d/2\) bits and \(d(d+4)/3\) units of time respectively, with the delay function \(W(LN_i) = 2i + 1\), where \(b\) is the number of bits required to send a node id.

3.4.3 For Ring Networks

Ring networks have been the most well-studied topology in the literature of distributed computing. They are very similar to the linear networks except that in linear networks the source node and the last node on line are connected to only one other node, while in ring networks every node is connected to two other nodes. In other words, the diameter of the spanning tree generated from the ring networks is half of that generated from the linear networks. Because of this difference, when we apply the FSP algorithm on ring networks, we can obtain better message and time complexities than those for linear networks.

Total Search Messages\(_{\text{ring}}\) = \(b \sum_{i=1}^{d/2} 2i\)

= \(b \left[ 2(1+2+3+\ldots+d/2) \right]\)

= \(b \left[ d^2/4+d/2 \right]\) bits

where \(b\) = number of bits required to send a node id

Total Trail Messages\(_{\text{ring}}\) = \(2 \sum_{i=1}^{d/2} i\)

= \(d^2/4+d/2\) bits

Total Messages\(_{\text{ring}}\) = \((b+1)(d^2/4+d/2)\) bits

\[ \therefore \text{Average Messages}_{\text{ring}} = \frac{1}{(d-1)} \text{Total Messages}_{\text{ring}} \]

= \([b+1]/2(d-1)\) \((d^2/2+d)\) bits

Total Time\(_{\text{ring}}\) = \(\sum_{i=1}^{d/2} \sum_{j=1}^{i} (2j+1) + i\)

= \(\sum_{i=1}^{d/2} [i^2 + 3i]\)

= \((d/2)(d/2+1)(d+1)/6 + 3(d/2)(d/2+1)/2\)

= \((d/2)(d/2+1)(d/6+5/3)\)
\[ = \frac{d^3}{24} + \frac{d^2}{2} + \frac{5d}{6} \]

:. \quad \text{Average Time}_{\text{ring}} = \frac{1}{d-1} \text{ Total Time}_{\text{ring}}

\[ = \frac{d(d/2+1)(d/6+5/3)}{(d-1)} \]

\[ = \frac{d^3/24(d-1)+d^2/2(d-1)+5d/6(d-1)}{d-1} \]

**Theorem 3.3**  The average communication and time complexities of finding a shortest path between two arbitrary points in distributed and synchronous ring networks are \([(b+1)/2(d-1)](d^2/2+d)\) bits and \(\pi d/2 + \frac{d^3}{24}(d-1)+\frac{d^2}{2}(d-1)+\frac{5d}{6}(d-1)\) units of time respectively, with the delay function \(W(LN_j)=2i+1\), where \(b\) is the number of bits required to send a node id.
3.5 Tradeoffs Between Delay Function and Complexity

If we decrease the delay time, the time complexity of our algorithm FSP will decrease but its message complexity will increase. In this section we will analyze their tradeoffs.

Recall that, in Section 3.3, we called the target distance \( r \) and the trail distance \( t \). In Section 3.4 we chose the delay function \( W(LN_i) = 2i + 1 \) in order to enforce the trail distance \( t \) to be zero (i.e. all outgoing search messages are stopped at the level of the sink). Thus, in this section we will examine the situation when \( t \) is greater than zero.

Assume the delay function \( W(LN_j) = \omega(i) \) where \( \omega \) is a predetermined constant and \( 0 < \omega < 2 \). Once the sink node \( S_a \) has been reached, the trail message is sent back, from level \( r \) to level 0 and then from level 0 to level \( r+t \), to stop the search messages (refer to Figure 3.3). Thus the total length the trail message traveled is \( 2r+t \), which takes \( 2r+t \) time units as well. Meanwhile, the search message is being delayed and forwarded toward the level \( r+t \). Such a delay and forward process takes \( \sum_{i=r,t} F(LN_i) + t \) time units, in which the first term is for the delay time and the second term is for the transmission time. Using the following formula we can express the term \( \omega \) in \( r \) and \( t \).

\[
2r+t \leq \sum_{i=r,t} F(LN_i) + t
\]

\[
2r \leq \sum_{i=r,t} F(LN_i) = \sum_{i=0,r+t} \sum_{i=0,r-1} \omega(i) - \sum_{i=0,r-1} \omega(i)
\]

\[
\leq (\omega/2) [(r+t)(r+t+1) - r(r-1)] = (\omega/2) (t+2r)(t+1)
\]

\[
\therefore \quad \omega \geq 4r / (t+2r)(t+1)
\]

The function obtained shows that if the trail distance \( t \) increases, the delay time will decrease substantially. As the delay time gets shorter, more and more search messages cannot be stopped before they are retransmitted from one node to others. Thus the number of messages needed will increase.
3.6 Closing Remarks

Before closing the chapter, several final remarks should be made.

First, our FSP algorithm works on arbitrary synchronous networks. In this chapter we have analyzed its complexities on arbitrary tree, linear and ring networks in the case that the maximum delay time is chosen. Further study is needed in order to determine its complexities on other network topologies, in particular the random graphs.

Second, in Section 3.5 we stated that if the trail distance increases, the delay time will decrease substantially but the number of messages needed will increase. We have also shown this relationship in mathematical terms. Further study may help develop an exact function for this relation and thus enable us to determine the average complexities of our FSP algorithm in the case that an arbitrary delay time is chosen.
Chapter 4
Conclusions

In this thesis we study distributed algorithms used in communication networks. In particular we have designed and analyzed new algorithms for the file sorting and shortest path finding problems.

The file sorting problem is to sort a file of n integers distributed over d stations in a communication network. We have presented new decentralized algorithms for almost balanced q-ary tree networks. These algorithms can easily be adapted to other topologies. The first algorithm, called SDF algorithm, requires extra memory space available at each node to store some temporary values. The second algorithm, called SDFW algorithm, removes such a requirement but only w. rks on tree networks with postorder labeling. Their message complexities are \( O(n/x \log_q d) \) and \( O(n/x \log_q^2 d) \) respectively, where \( x \) is the message capacity.

Tradeoffs between memory requirements and communication complexity of the SDF algorithm have been discussed. If one is willing to sacrifice more local storage and larger message capacity, the performance of the SDF algorithm can be further improved. We have shown that, under similar conditions, our algorithm can achieve a better performance than existing file sorting algorithms. On the other hand, the impact of local processing on time complexity for synchronous networks is analyzed. We showed that the local processing time became a noticeable factor in our study of sorting file in the tree networks.

The two new algorithms have managed to evenly distribute the computation workload at each participating node in each cycle. They also provide a great deal of parallelism. If the message capacity \( x \) is assumed to be greater than 1, these algorithms will be \( x \) times faster and need \( x \) times fewer messages. Moreover, the SDF algorithm (for sorting with extra memory) can be modified so that it will work on tree networks of any labeling. This modification requires that the root station \( S_d \) has knowledge of the distribution order \( P \). Therefore \( S_d \) can send assigned elements directly to their destinations.

Although the proposed file sorting methods use some blind operations which cannot recognize the already-sorted inputs, this shortcoming could be improved by sending both of the
minimum and maximum elements in a station to its father. Thus, at the end, the root station can compare the maximum element of the left subtree and the minimum element of the right subtree. If the former is smaller, then all elements in the left subtree are smaller than those in the right subtree. The root keeps the largest elements and will retire from further actions. The original tree is then divided into two subtrees, each being operated on with the same algorithm. Applying this technique repeatedly in AB binary tree networks, an already-sorted inputs can be detected in $O(\log_2 d)$ time with $O(d \log_2 d)$ messages.

In this thesis we have also studied the problem of finding a shortest path between two arbitrary stations $S_m$ (source) and $S_n$ (sink) in distributed and synchronous communication networks. Based on Dijkstra's method our algorithm performs search and trail operations. During the search process search messages fan out from the source and start building a spanning tree until $S_n$ is reached. Once the sink has received the search message it starts the trail process, in which the sink sends a trail message backwards from $S_n$ to $S_m$ in order to stop the outgoing search messages as soon as possible. To accomplish this, before a full spanning tree is built, a delay is introduced at each node which has received a search message; that is after receiving a search message a node will retain the search message for some specified amount of time before sending it to its neighbours. Thus the search message does not always traverse the entire network as a straightforward implementation of Dijkstra's algorithm would do.

The average communication and time complexities of finding a shortest path between two arbitrary points in distributed and synchronous random tree networks are $(b+1)$ $[2+(2\pi d)^{1/2}-2d+(2\pi d)^{1/2}]$ bits and $\pi d/2 + 3(\pi d/2)^{1/2}$ units of time respectively, with the delay function $W(\ln_i)=2i+1$, where $b$ is the number of bits required to send a node id.

Complexities of the FSP algorithm have been analyzed for arbitrary tree, linear and ring networks for the case that the maximum delay time is chosen. Further study is needed in order to determine its complexities on other network topologies. We have also mathematically shown that if the trial distance increases, the delay time will decrease substantially but the number of messages needed will increase. Further study is needed to develop an exact function for this relation and thus enable us to determine the average complexities of our FSP algorithm in the case that an arbitrary delay time is chosen.
Appendix A

The Complete SDF Algorithm for AB q-ary Tree Networks

![Diagram of a station Sj]

- **Father** $F_j$
- **Rj** elements stored at $S_j$
- **Ej** extra memory available at $S_j$
- **T1, T2, …** $T_{ji}$ is total # elements in $T_i$
- **P1, P2, …** $P_{ji}$ is the min. element in $T_i$
- **Cj1, Cj2, …** $C_{ji}$ is the i-th child of $S_j$
- **Child 1, Child 2, …, Child i** $T_i$ is subtree rooted at $C_{ji}$

**Figure A** Local variables at a station $S_j$

During the execution of the SDF algorithm, the processor in a station $S_j$ changes its state from time to time. There are four states, $\text{STATE}_j = \{\text{'Sleep'}, \text{'Initialize'}, \text{'Sort_elements'}, \text{'Final'}\}$. The algorithm begins with all the stations in the state 'Sleep' and ends when all the stations reach the state 'Final'. Since processors communicate through exchange of messages, six types of messages are needed in our algorithms. They are:

- **WAKE_UP** - a message from $S_j$ to its children set $C_j$ to wake up each of $C_j$.
- **NUM(h)** - a message from $S_j$ to its father $F_j$ which contains the total number of
integers stored in the subtree T_j rooted at S_j, i.e. h = |R_j|+S_t where t \notin T_j.

**REQ_MIN** - a message from S_j to its child C_{ji} to request the minimum integer available in C_{ji}; here C_{ji} denotes the i-th child of C_j.

**SEND_MIN**(k) - a message from S_j to its father F_j which carries its minimum integer k.

**ASSIGN**(k) - a message from S_j to its child C_{ji} which carries the assigned element k; here C_{ji} is one of the intermediate stations through which the assigned element k could go to its destination. In our algorithm, C_{ji} is determined by taking the leftmost child of S_j whose \(T_{ji}\) is not equal to zero.

**TERMINATE** - a message from S_j to its father F_j to indicate termination of the process.

The complete SDF algorithm on AB q-ary tree networks is presented as following:

**In \(STATE_j = 'Sleep'\)**

While In \(STATE_j = 'Sleep'\) Do

A sleeping station can be woken up by: (a) itself, or (b) receiving a WAKE_UP message;

- set \(STATE_j = 'Initialize'\) after being woken up;

Endwhile;

**In \(STATE_j = 'Initialize'\)**

If (the father position of S_j is not connected to another station) Then

- set \(F_j = nil\);

Endif;

If (the i-th child position of S_j is not connected to another station) Then

- set \(C_{ji} = nil\) and \(P_{ji} = *\);

Else

- set \(P_{ji} = nil\);

Endif;

If (all \(C_{ji}'s = nil\) (where \(i=1,2,\ldots,q\)) Then \{ a leaf station \}
Send message NUM(|R_j|) to F_j;
set all T_{ij}’s = 0 (where i=1,2,...,q);
Else
  Send message WAKE_UP to F_j and all C_{ij}’s (where i=1,2,...,q);
Endif;

Wait Until (received message NUM(h_i) from every child C_{ij} (where i=1,2,...,q)) Do
  set T_{ij} = h_i;
  Send message NUM( |R_j|+\sum_{g=1}^{i} h_g ) to F_j;
Enddo;

set E_j = {};
set STATE_j = ‘Sort_elements’;

In STATE_j = ‘Sort_elements’
  While In STATE_j = ‘Sort_elements’ Do
    If (P_{ij} = nil) Then
      Send message REQ_MIN to C_{ij} to request for the min.;
    Endif;

    If (message REQ_MIN from F_j has arrived) And (no P_{ij} = nil) Then
      Send message SEND_MIN(k=Small(R_{ij},P_{ij})) to F_j;
      remove (k) from either R_j or P_j accordingly;
    Endif;

    If (received message SEND_MIN(k) from C_{ij}) Then
      store (k) in P_{ij};
    Endif;

    If (F_j = nil) And (no P_{ij} = nil) Then
      Send message ASSIGN(k=Small(R_{ij},P_{ij})) to the leftmost child C_{ij} whose T_{ij} ≠ 0;
      remove (k) from either R_j or P_j accordingly;
      T_{ij} = T_{ij} + 1;
    Endif;
  Endwhile;
If (received message ASSIGN(k)) Then
    If (all $T_{ji}$'s = 0 (where $i=1,2,...,q$)) Then
        add (k) into $E_j$;
    Else
        Retransmit this message ASSIGN(k) to the leftmost child $C_{ji}$ whose $T_{ji} \neq 0$;
    Endif;
    $T_{ji} = T_{ji} - 1$;
Endif;

If ($E_j$ is full) Or (all $T_{ji}$'s = 0 (where $i=1,2,...,q$)) Then
    set $P_{ji} = \ast$;
    If ($|R_j| = 0$) Then
        set $STATE_j = 'Final'$;
    Endif;
Endif;
Endwhile;

In $STATE_j = 'Final'$
While In $STATE_j = 'Final'$ Do
    If ($C_{ji} = \text{nil}$) Or (received message TERMINATE from $C_{ji}$) Then
        If ($F_j \neq \text{nil}$) Then
            send message TERMINATE to $F_j$ and then terminate the SDF process in $S_j$;
        Else
            declare the END OF ENTIRE PROCESS;
        Endif;
    Endif;
Endwhile;
Appendix B

The Complete SDFW Algorithm for AB q-ary Tree Networks

During the execution of the SDF algorithm, the processor in a station $S_j$ changes its state from time to time. There are four states, $\text{STATE}_j = \{\text{'Sleep'}, \text{'Initialize'}, \text{'Sort_elements'}, \text{'Final'}\}$. The algorithm begins with all the stations in the 'Sleep' state and ends when all the stations reach the state 'Final'.

In this algorithm, we use a function called $\text{DIRECTION}(j,i)$ to solve the following problem: In an q-ary tree network with postorder labeling, a station $S_j$ which has children $(C_{j1},C_{j2},...,C_{jq})$ wants to send a message to station $S_i$. The function $\text{DIRECTION}$ determines to which child $S_j$ should transmit the message so that this message will be in the correct route towards
its final destination $S_j$. Since the network is postorder labeled, this function is not hard to implement.

In a communication network, processors communicate through exchange of messages, six types of messages are needed in our algorithms. They are:

**WAKE_UP** - a message from $S_j$ to its children set $C_j$ to wake up each of $C_j$.

**REQ_MAX** - a message from $S_j$ to its child $C_{ji}$ to request for the maximum integer available in $C_{ji}$, here $C_{ji}$ denotes the $i^{th}$ child of $C_j$.

**SEND_MAX**(k,t) - a message from $S_j$ to its father $F_j$ to carry the maximum integer $k$ which is currently available at $S_j$ but originally from $S_t$.

**ASSIGN**(k,u) - a message from $S_j$ to its child $C_{ji}$ to carry the assigned element $k$ that is heading to its destination $S_u$.

**PUSH**(k,v) - a message from $S_j$ to its child $C_{ji}$ to push the element $k$ to its destination $S_v$, here $C_{ji}$ is the $i^{th}$ child of $S_j$ and the value of $i$ is determined according to the value of $v$ and the postorder labelling of the tree network.

**PUSH_REQ**(k,v) - a message from $S_j$ to its child $C_{ji}$ to push the element $k$ to its destination $S_v$ and to request for the maximum integer available in $C_{ji}$.

**TERMINATE** - a message from $S_j$ to its children $C_j$ to indicate the end of the current iteration. Station $S_j$ will retire from the SDFW process.

The complete SDFW algorithm on AB q-ary tree networks is presented now:

While **In_STATE**$_j$ = 'Sleep' Do

A sleeping station can be woken up by: (a) itself, or (b) a WAKE_UP message;

If (the father position of $S_j$ is not connected to another station) Then
set ITERNUM = 0 and F_j = nil;
Endif;
set STATE_j = 'Initialize' after being woken up;
Endwhile;

In STATE_j = 'Initialize'
If (F_j = nil ) Then
    ITERNUM = ITERNUM+1;
Endif;

If (the i^{th} child position of S_j is not connected to another station) Then
    set C_{ji} = nil and P_{ji} = (-\infty) and D_{ji} = (nil,nil);
Else
    set P_{ji} = (nil,nil) and D_{ji} = (nil,nil);
Endif;

set R_j = |R_j|; {normalize the integer elements stored in S_j}
Send message WAKE_UP to F_j and all C_{ji} 's (where i=1,2,...,q);
set STATE_j = 'Sort_elements';

While In STATE_j = 'Sort_elements' Do
    If (P_j = D_{ji} = (nil,nil)) Then
        Send message REQ_MAX to C_{ji} to request for the max.;
    Endif;

    If (message REQ_MAX from F_j has arrived) And (no P_{ji} = (nil,nil)) Then
        let R_{j}' = \{all positive integer elements in R_j\};
        let (b', v') = Large(R_{j}' \cup P_j); {where v = j for set R_j}

    Endif;
Endwhile;
Send message SEND_MAX(b', v') to F_j;
remove (b', v') from either R_j or P_j accordingly;
Endif;

If (received message SEND_MAX(b,v) from C_{ji}) Then
    P_{ji} = (b, v);
    D_{ji} = (nil,nil);
Endif;

{if all P_{ji}'s are (-\infty , ) then F_j will retire from further operations, otherwise do the following}

If (F_j = nil) And (no P_{ji} = (nil,nil)) And (more than one P_{ji} \neq (-\infty, )) Then
    let R_j' = {all positive integer elements in R_j};
If R_j' \neq {} Then {the root station is collecting the \alpha-integer}s
    let (b,v) = Large(R_j \cup P_j); {where v = j for set R_j}
    If (any P_{ji} = (b,v)) Then
        remove k = Small(R_j') from R_j;
        l = DIRECTION(j,v);
        Send message PUSH_REQ(k,v) to S_i (towards final destination S_o) ;
        P_{ji} = (nil,nil);
        D_{ji} = (k,v);
    Else
        remove b from R_j;
    Endif;
    add (-b) into R_j;
Else {the root will compare left and right max.}
    For i = 1, q Do
        let (b_i, v_i) = P_{ji};
    Endfor;
    l = q;
While ($P_{ji} = (-\infty, j))$ Do
  $l = l-1$;
Endwhile;
remove ($b', v'$) = Large($b_l, v_l$) from $P_j$;
Send message ASSIGN($b', v'$) to $C_{ji}$ (towards final destination $S_{v_l}$);
For $l' = 1, l-1$ Do
  remove ($b', v'$) = the $l'$-th Small($b_l, v_l$) from $P_j$;
  Send message PUSH_REQ($b', v'$) to $C_{ji}$ (towards destination $S_{v_l}$);
  $D_{ji} = (b_l, v_l)$;
Endfor;
Endif;
Endif;

If (message PUSH_REQ($b,v$) has arrived) And (no $P_{ji} = (\text{nil}, \text{nil})$) And ($v \neq j$) Then
  $l = \text{DIRECTION}(j,v)$;
  $D_{ji} = (b,v)$;
  let ($b_l, v_l$) = $P_{ji}$;
  If ($b > b_l$) Then
    {swap $D_{ji}$ and $P_{ji}$}
    $D_{ji} = (b_l, v_l)$;
    $P_{ji} = (b,v)$;
  Endif;
  let $R_j' = \{\text{all positive integer elements in } R_j\}$;
remove ($b', v'$)=Large($R_j' \cup P_j$) from either $R_j$ or $P_j$ according; {where $v=j$ for set $R_j$}
Send message SEND_MAX($b', v'$) to $P_j$;
If ($v' = l$) Then
  $l = \text{DIRECTION}(j,l)$;
  Send message PUSH_REQ($D_{ji}$) to $C_{ji}$;
Else
  Send message PUSH($D_{ji}$) to $C_{ji}$;
  If ($v' \neq j$) Then
    $l = \text{DIRECTION}(j,v')$;
Send message REQ\_MAX to C_{ji}; \{towards final destination S_v'\}

Endif;
Endif;

D_{ji} = (nil,nil);

If (message PUSH\_REQ(b,v) has arrived) And (no P_{ji} = (nil,nil)) And (v = j) Then

add (b) into P_j;

let R_j' = \{all positive integer elements in R_j\};

remove (b', v')=Large(R_j' \cup P_j) from either R_j or P_j according; \{where v=j for set R_j\}

Send message SEND\_MAX(b', v') to P_j;

If (v' \neq j) Then

l = DIRECTION(j,v');

Send message REQ\_MAX to C_{ji};

Endif;
Endif;

If (message PUSH(b,v) has arrived) Then

l = DIRECTION(j,v);

D_{ji} = (b,v);

If (P_{ji} \neq (nil,nil)) Then

let (b', v') = P_{ji} ;

If (b > b') Then \{swap D_{ji} and P_{ji}\}

D_{ji} = (b', v');

P_{ji} = (b,v');

l' = DIRECTION(j,v');

Send message PUSH(D_{ji}) to C_{ji}';

Else

l' = DIRECTION(j,v');

Send message PUSH(D_{ji}) to C_{ji}';

Endif;
D_{j} = (nil, nil);
Endif;

If (received message ASSIGN(b,v)) Then
  If (j = v) Then
    remove (b) from R_{j};
    add (-b) into R_{j};
  Else
    1 = DIRECTION(j,v);
    Retransmit this message ASSIGN(b,v) to the child C_{j}; \{towards final destination S_{v}\}
  Endif;
Endif;

If (F_{j} = nil) And (no P_{j} = (nil,nil)) And (exactly one P_{j} ≠ (-∞, )) Then
  For each (b,v) in \{P_{j}∪D_{j}\} Do
    1 = DIRECTION(j,v);
    Send message PUSH(b,v) to C_{j};
  Endfor;
  Send message TERMINATE to each C_{j};
  set STATE_{j} = 'Final';
Endif;

If (received message TERMINATE) Then
  If (all C_{j} 's = nil) Then \{the leaf station\}
    set STATE_{j} = 'Final';
  Else
    While ( \{P_{j}∪D_{j}\} ≠ \{\}) Do
      remove an element (b,v) from \{P_{j}∪D_{j}\};
      If ((b,v) ≠ (nil,nil)) And (b,v) ≠ (-∞, )) Then
        1 = DIRECTION(j,v);
  Endwhile;
Endif;
Send message PUSH(b,v) to C_j;

Endif;
Endwhile;
set F_j = nil;
set STATE_j = 'Initialize';
Endif;
Endif;
Endwhile;

In STATE_j = 'Final'

declare the end of the ITERNUM\textsuperscript{th} iteration;
terminate the SDFW process in S_j;
Appendix C

The Complete FSP Algorithm on Synchronous Networks

The FSP algorithm begins with all stations in the 'Sleep' state. The execution for a station \( S_j \) ends in the 'Find' state when it receives (and retransmits in some cases) TRAIL message(s). In the algorithm described below, \( \text{PRE}_j, \text{SUC}_j, \text{LN}_j \) and \( \text{TIMER}_j \) are four local variables in \( S_j \). The \( \text{PRE}_j \) and \( \text{SUC}_j \) contain the predecessor and successors of \( S_j \). The \( \text{LN}_j \) is the level number of the source-rooted tree of \( T \) at which \( S_j \) is located. The \( \text{TIMER}_j \) monitors the delay time in \( S_j \). (Note: one unit of local time is equal to one unit of global time in synchronous networks.) Let \( r \) be the target distance from the source to sink and \( t \) be the predetermined trail distance from the sink to where all outgoing search messages are stopped. Thus the delay function is equal to \( W(\text{LN}_j) = \omega(j) \) where \( \omega \) is a constant to satisfy \( \omega \geq 4r/(t+2r)(t+1) \). Three types of messages are:

- \( \text{SEARCH}(e, S_n) \) - from \( S_j \) to its successors \( \text{SUC}_j \) carries an integer \( e \) which is the level number \( \text{LN}_j \) and id of the sink station \( S_n \).

- \( \text{TRAIL}(\text{bit 0}) \) - from \( S_j \) to its successors \( \text{SUC}_j \) stops the outgoing \( \text{SEARCH} \) message.

- \( \text{TRAIL}(\text{bit 1}) \) - from \( S_j \) to its predecessor \( \text{PRE}_j \) stops the outgoing \( \text{SEARCH} \) message; it also marks the receiving node \( \text{PRE}_j \) to be a node on the selected shortest path \( \text{SP}_{m,n} \).

The complete FSP algorithm on AB q-ary tree networks is presented as follows:

**In STATE \( S_j = 'Sleep' \)**

A sleeping station can be woken up by:

(a) itself for it is the source station \( S_m \), or

(b) receiving a \( \text{SEARCH}(e, S_n) \);

After the source station \( S_m \) wakes up, it does the following:

(a) set its predecessor \( \text{PRE}_m = \{ \} \) and successors \( \text{SUC}_m = \{ \text{all its adjacent stations} \} \);

(b) set \( \text{LN}_m = 0 \);

(c) send \( \text{SEARCH}(\text{LN}_m, S_n) \) to its successors \( \text{SUC}_j \);
Set $\text{STATE}_j = '\text{Find}'$;

**In $\text{STATE}_j = '\text{Find}'$**

If $(S_j \text{ received } \text{SEARCH}(e, S_o) \text{ from } S_k)$ Then

Set $S_j$'s predecessor $\text{PRE}_j = S_k$ and

$S_j$'s successors $\text{SUC}_j = \{\text{all adjacent stations of } S_j \text{ excluding } S_k\}$;

Set $\text{LN}_j = e+1$;

If $(S_j \text{ is the sink station})$ Then

$r = \text{LN}_j$;

declare the SINK STATION HAS BEEN REACHED;

send TRAIL(1) to its predecessor $\text{PRE}_j$;

declares the END OF EXECUTION FOR STATION $S_j$;

Else

set $\text{TIMER}_j = W(\text{LN}_j)$ units of time;

wait until either $\text{TIMER}_j$ is expired or receiving TRAIL or TRAIL$^+$;

If (received TRAIL(0) or TRAIL(1) during waiting) Then

declares the END OF EXECUTION FOR STATION $S_j$;

Else

send SEARCH(\text{LN}_j, S_o) to its successors $\text{SUC}_j$;

Endif;

Endif;

Endif;

If $(S_j \text{ received } \text{TRAIL}(1) \text{ from } S_k)$ Then

$S_j$ marks itself as a node on the resulting shortest path $SP_{m,n}$;

send TRAIL(1) to its predecessor $\text{PRE}_j$;

send TRAIL(0) to each of its successors $\text{SUC}_j$;

declares the END OF EXECUTION FOR STATION $S_j$;

Endif;
References


[LeSaUr87] Van Leeuwen, Nicola Santoro and Jorge Urrutia, "Guessing Games and Distributed Computations in Synchronous Networks", Lecture Notes in Computer Science 267, ICALP, pp.347-356.


[McSc86] Robert McCurley and Fred B.Schneider, "Derivation of a Distributed Algorithm


[RoSaSi85] "Distributed Sorting", *IEEE Transactions on Computers*, Vol.c-34, No.4,
pp.372-376.


