Lecture Notes: Distributed Algorithms

Rashid Bin Muhammad, PhD.

This booklet includes lecture notes from parallel and distributed computing course I taught in Fall 07, Spring 08, and Fall 08 at the Kent State University, USA. I wrote a good part of these notes in Fall 2007; I revise them every time I teach the course. The problems in the notes have been used in class lectures, homework assignments, or exams. You can find a collection of homework assignments and exams from past semesters of my class online at http://www.personal.kent.edu/~rmuhamma/.

Introduction

As Tanenbaum put it, computer systems are undergoing a revolution. From 1945, when the modern computer era began, until about 1985, computers were large and expensive. However, starting in the mid-1980s, two advances in technology began to change that situation. The first was the development of powerful microprocessors. The second development was the invention of high-speed computer networks. The result of these technologies is that it is now not only feasible, but easy, to put together computing systems composed of large numbers of CPUs connected by a high-speed network. They are usually called distributed systems, in contrast to the previous centralized systems (or single-processor systems). There is only one fly in the ointment: software. Distributed systems need radically different software than centralized systems do. And this is the very reason to study the design and analysis of distributed algorithms.

Distributed algorithms are algorithms designed to execute on hardware consisting of several interconnected processors. Parts of a distributed algorithm execute concurrently and independently, each with only a limited amount of information. The algorithms are supposed to work correctly, even if the individual processors and communication channels operate at different speeds and even if some of the components fail. Distributed algorithms have a wide range of applications, including telecomunications, distributed information processing, scientific computing, and real-time process control. For example, today's telephone systems, airline reservation systems, weather prediction systems, and aircraft and nuclear power plant control systems all depend critically on distributed algorithms.

Because of the nature of the applications, it is important that the distributed algorithms run correctly and efficiently. However, because the setting in which distributed algorithms run are so complicated, the design of such algorithms can be extremely difficult task. So, here we give reasons for the study of distributed algorithms by briefly introducing the types of hardware and software systems for whcih distributed algorithms have been developed. By a distributed system we mean all computers applications where several computers or processors cooperate in some way. However, the main topic of this notes is not what these systems look like, or how they are used, but how they can be made to work. It is important to note that the entire structure and operation of a distributed system is not fully understood by a study of its algorithms alone. To understand such a system fully one must also study the complete architecture of its hardware and software includeing issues related to properties of the programming languages used to build the software of distributed systems.

1 What is a Distributed System?

The term "distributed system" means an interconnected collection of autonomous computers, processes, or processors. Note that to be qualified as "autonomous", the nodes (i.e., computers, processes, or processors) must at least be equipped

with their own private control; thus, a parallel computer of the single-instruction, multiple-data (SIMD) model does not qualify as a distributed system. To be qualified as "interconnected", the nodes must be able to exchange information.

For our purpose it is sufficient to give a Tanenbaum's loose characterization of a distributed system: *A distributed system is a collection of independent computers that appear to the users of the system as a single computer.*

Tanenbaum's definition has two aspects. The first one deals with hardware: the machines are autonomous. The second one deals with software: the users think of the system as a single computer. Both are essential.

1.1 Goals of Distributed Systems

Here we will discuss the motivation and goals of typical distributed systems and look at their advantages and disadvantages compared to centralized systems.

1.1.1 Advantages of Distributed Systems over Centralized Systems

- Economics: Microprocessors offer a better price/performance than mainframes.The real driving force behind the trend toward decentralization is economics. With Microprocessor technology, for a few handred dollars you can get a CPU chip that can execute more instructions per second than one of the largest 1980s mainframes. A slight variation on this theme is the obervation that a collection of microprocessors cannot only give a better price/performance ratio than a single mainframe, but may yield an absolute performance that no mainframe can achieve at any price.

- Speed: A distributed system may have more total computing power than a mainframe. The parallel machines may have more speed. But the distinction between distributed and parallel systems is fifficult to maintain because the design spectrum is really a continuum. Here, we prefer to use the term "distributed system" in the broadest sense to denote any system in which multiple interconnected CPUs work together.

- Inherent Distribution: Some applications involve spatially separated machines. Some applications are inherently distributed such as airline reservations, banking, computer-supported cooperative work, computer-supported cooperative games, etc.

- Reliability: If one machine crashes, the system as a whole can still survive. By distributing the workload over many machines, a single chip failure will bring down at most one machine, leaving the rest intact.

- Incremental growth: Computing power can be added in small increments. In contrast to centralized system, with a distributed systme, it may be possible simply to add more processors to the system, thus allowing it to expand gradually as the need arises.

Distributed Network Algorithms

Network algorithms are design to run on a computer network. These algorithms include how routers send data packet through the network. The computer network is modeled as a graph, $G = (V, E)$, in which the vertices represent processors or routers, and the edges represent communication channels. Network algorithms are unusual as compared to the traditional algorithms in the sense that not only the inputs are spread through out the network; the execution of algorithms is distributed across the processors in the network.

The major class of the network algorithms is the routing algorithms. Routing algorithms specify how to move about information packets among various computers in the network. A well designed routing algorithms should route packets to arrive at their destinations quickly and reliably and while also being "fair" to other packets in the network.

2 Complexity Measures and Models

Before we start the algorithmic study of network algorithm, we require a basic understanding of how interprocessor communications is performed.

2.1 The Network Protocol Stack

Diplomats learned a long time ago that when different cultures come together, you need rules for accurate transfer of information. The rules diplomats develop for communicating are called protocols. Network of today (e.g. Internet) consists of many millions of computers on tens of thousands of subnetworks. In fact, the Internet is arguably the most complex system ever assembled by humankind. Sooner or latter these computers are going to communicate with each other. Therefore, we a have a similar (to that of diplomats) function in networks, we use the same name for our rules. The most fundamental of these standards relate to a basic set of functions that has been defined collectively by the networking industry. At the core of these functions is a set of rules for exchanging information. These rules are known as *protocols*. Networking technology is highly modular: Its system is divided into "large pieces" of well-defined functions. Thus, we present the network protocols in the form of a five-layer model. The idea here is to separate the various functions of network communication into layers. Each layer has a different purpose. Note that one well-known stack with seven layers is called the Open Systems Interconnect (OSI) Reference Model. Since, the goal of this course is the algorithmic study of distributed/networking algorithms; the five-layer model is quite enough for this purpose. This simple five-layer model is associated with the internet. For algorithmic study, this stack is sometimes shown with only four layers, with the Data-link layer and Physical layers combined into a single host-to-network layer.

The functions of the five layers are:

1. Physical layer: This layer is responsible for passing information between two physical locations. In its simplest form, the physical layer is a wire where raw bits are transmitted across some medium, such as fiber optics cable or electrical wires. In most cases, design decisions based on the engineering techniques for representing 1's and 0's, bandwidth, and transmission noise.

2. Data-link layer: This layer is responsible for controlling operation of a single data communication link to move information efficiently from end to end. One important function of the layer is to handle algorithms or methods for breaking data into subunits called frames and send to other physical location through the physical layer.

3. Network Layer: This layer is responsible for putting the segment (different from frames) into "electronic envelopes" called packet and providing the organization necessary to get the packets from sender to receiver. In other words, this layer handles the algorithms and method to for breaking data into packets and routing them from one computer (source) to another computer (destination) through the communication links, routers, and gateways of the internet (network). The algorithmic issues concerned with this layer deal with methods for routing packets through network and for breaking and reassembling packets. The primary Internet protocol used in this layer is the Internet protocol, or IP.

4. Transport Layer: This layer is responsible for packaging data for host-to-host delivery. This layer also provides a means to identify how messages are to be used in the receiving host in that a set of messages is associated with a particular application. In particular, this layer accepts data from various applications, splits it up into smaller units if need be, and passes these to the network layer in a fashion that fulfills specific guarantees on reliability and congestion control. The two primary Internet protocols used in this layer are Transmission Control Protocol (TCP) and the User Datagram Protocol (UDP).

5. Application Layer: This layer is responsible for whatever the user wants the computer to do, such as interacting with a remote computer: virtual terminals (TELNET and SSH), electronic mail (SMTP), transferring files (FTP and SCP), the World Wide Web (HTTP), the Domain Name Service (DNS), etc. In other words, at this layer applications operate, using low-level protocols.

2.2 Computational Model

The reputation of distributed computing is tarnished for its excess of computational models. And, this is not it - those computational models do not translate exactly to real-life architectures. Therefore, we have chosen not to organize the course work around models, but rather around fundamental problems (as we

have done in our traditional algorithms course), indicating where the choice of model is crucial to the solvability or complexity of a problem.

In this course, we consider the basic message-passing model because this is one of the most fundamental paradigms for distributed applications. Another reason to chose the message-passing model to study network/distributed algorithms is that this model abstracts the functionality provided by the various layers in the network protocol stack

In a message-passing system, processors communicate by sending messages over communication channels. The communication channel is a bidirectional connection between any two processors. The pattern of connections provided by the channels describes the topology of the system. The topology is represented by an undirected graph in which each node represents a processor and an edge is present between two nodes if and only if there is a channel between the corresponding processors. For example, at the network layer of the internet, nodes correspond to specialized computers called IP routers, each identified by a unique numeric code called IP address, and the messages exchanged by them are basic data transmission units called IP packets.

The collection of channels is often referred to as the network. Each processor in the network is assigned a unique numeric identifier, id. For example, the identifier could be the IP address of a router or host on the internet. Moreover, we assume that each processor in the network knows its neighbor (i.e. adjacent node) and that it communicates directly only with its neighbors. Note that we will deal exclusively with connected topologies.

An algorithm for a message-passing system with a specific topology consists of a local program for each processor in the system. A processor's local program provides the ability for the processor to perform local computation and to send messages to and receive messages from each of its neighbors in the given topology.

Let us sum up the above discussion by defining the message-passing model as:

- Set of processes having only local memory.
- Processes communicate by sending and receiving messages. That is, a processor sends a message, often representing a request. The message is delivered to a receiver, which processes the message, and possibly sending a message in response. In turn, the reply may trigger a further request, which leads to a subsequent reply, and so forth.
- The transfer of data between processes requires "cooperative operations" to be performed by each process (a send operation must have a matching receive)

In the basic message-passing paradigm, the basic operations are *send*, and *receive*. For connection-oriented communication, the operations *connect* and *disconnect* are also required. With the abstraction provided by this model, the interconnected processes perform input and output to each other, in a manner similar to file I/O. The I/O operations encapsulate the details of network

communication at the operating-system level. Note that the socket application programming interface is based on this paradigm.

The message-passing model has gained wide acceptance in the fields of parallel and distributed computing (I think, due to its point-to-point communication routines for data transfer between two processors) that include:

- Hardware Matching: The message-passing model fits well with parallel and distributed paradigms. That is, this model is the choice of parallel supercomputers as well as cluster of workstations, which consists of separate processors connected by a communication network.
- Functionality: Message-passing model offers a full set of functions for expressing parallel and distributed algorithms.
- Performance: The CPUs require the management of their memory hierarchy, especially their cashes. Message-passing model achieves this by giving algorithm designer and programmer explicit control of data locality.

The principle drawback of the message-passing model is the responsibility it places on the algorithmic designer and especially, on the programmer. The algorithm designer must explicitly implement a data distribution scheme and all inter-processors communication and synchronization. In so doing, at the implementation level, it is the programmer's responsibility to resolve data dependencies and avoid deadlock and race conditions.

Now we present models for synchronous and asynchronous message-passing for systems with no failures.

- Synchronous model: In this timing model, processors execute in lockstep: The execution is partitioned into rounds, and in each round, every processor can send a message to each neighbor, the messages are delivered, and every processor computes based on the messages just received.This means that each processor in the network running the algorithm has an internal clock that times program execution and the clocks of all the processors are synchronized. The fundamental timing assumptions for this model consist of: The speeds of processors running the algorithm are uniform. Each processors take same number of clock ticks (i.e. same amount of time) to perform same operations. Communication between any two processors require same amount of time. That is, it takes the same number of clock ticks to send message through any communication channel in the network. Generally, this model is not realizable in practical distributed systems. Nevertheless, it is very convenient for designing algorithms, because an algorithm need not face much uncertainty. The good point is that once an algorithm has been design for this ideal timing model, it can be automatically simulated to work in other, more realistic, timing models.
- Asynchronous model: In this model, there is no fixed upper bound on how long it takes a message to reach at the destination node. In other words,

if there is no upper bound on how much time elapses between consecutive steps of a processor, a system is called asynchronous. For example, the Internet is an asynchronous system because a message (for instance, E-mail) can take days to arrive, although often it takes seconds. In particular, this model makes no assumptions about processors' internal clocks. Similarly, it does not assume that the processors' speeds are the same. In the absence of these assumptions, an algorithm's steps are determined by conditions or events, not by clock ticks. Still, to allow algorithms to perform their tasks effectively, there are some timing assumptions: Each communication channel is a first-in first-out (FIFO) queue that can buffer an arbitrary number of messages. That is, the messages that are sent on an edge are stored in a buffer for that edge so that the messages arrive in the same order they are sent. While processor speeds can vary, they are not arbitrarily different from on another. That is, there is a basic fairness assumption that guarantees that if a processor p has an event enabling p to perform a task, then p will eventually perform that task.

Even though we have said that in this system, there is no fixed upper bound on a message delivery. In reality, there is always an upper bound (even in our E-mail example), but sometimes these upper bound are very large, are only infrequently reached, and can change over time. Instead of designing an algorithm that depends on these bounds, it is often desirable to design an algorithm that is independent of any particular timing parameters, namely, an asynchronous algorithm.

The asynchronous message-passing model applies to loosely-coupled machines and to wide-area networks. The synchronous message-passing model is an idealization of message-passing systems in which some timing information is known, such as upper bound on message delay. Systems that are more realistic can simulate the synchronous message-passing model, for instance, by synchronizing the clocks. Thus, the synchronous message-passing model is a convenient model in which to design algorithms, and then the algorithms can be automatically translated to models that are more realistic.

2.3 Complexity Measures

We will be interested in four complexity measures, the computational rounds, space, the number of messages and the amount of time, required by distributed algorithms. To define these measures, we need a notion of the algorithm terminating. We say that the algorithm has terminated when all processors are in terminated states and no messages are in transit.

- Computational Rounds: The natural way to measure time in synchronous algorithms is simply to count the number of rounds until termination. Thus, the time complexity of a synchronous algorithm is the maximum number of rounds until the algorithm has terminated. Measuring time in an asynchronous system is less straightforward. In synchronous algorithm, clock ticks determine these rounds, whereas in asynchronous algorithms

these rounds are determined by propagating waves of events across the network.

- Space: Like traditional space complexity, we can define as the amount of space or memory needed for computation by an algorithm. However, unlike traditional algorithm, here we must specify whether it is a global bound on the total space used by all computers in the algorithm or a local bound on how much space is needed per computer involved.
- Local Running Time: In network algorithms, computations are carried out across many computers over the network, so it is difficult to analyze the global running of a network algorithm. In spite of, we can analyze the amount of local computing time required for a particular computer for computation in the network. The basic argument here is if all computers in an algorithm are performing the same type of function, then a single local running time bound can be sufficient for all. On the other hand, if there are many different classes of computers contributing in the computation in an algorithm, then we are required the running time for each class of computers.
- Message Complexity: The message complexity of an algorithm for either a synchronous or an asynchronous message-passing algorithm is the maximum of the total number of messages sent between all pairs of computers during the computation. For example, if a message M were routed through p edges to get from one computer to another, we would say that the message complexity of this communication is $p|M|$, where $|M|$ denotes the length of M (in words).

Note that to measure time of asynchronous algorithm a common approach is to assume that the maximum message delay in any execution is one unit of time and then calculate the running time until termination.

3 Fundamental Distributed Algorithms

The function of the size of the problem provides the complexity measures for network algorithm. For instance, the following parameters may express the size of a problem:

- The number of words used to express the input;
- The number of processor deployed; and
- The number of communication connection among processors.

Therefore, to analyze properly the complexity measures for a network algorithm one must define the size of the problem for that algorithm.

3.1 Leader Election

Problem. Given a network of n processors that are connected in a ring, that is, the graph of the network is a simple cycle of n vertices. The objective is to identify one of the n processors as the "leader" and have all the other processors agree on this selection.

Assumption. *Lets assume, for the sake of simplicity, the ring is a directed cycle.*

3.1.1 Synchronous Solution

The main idea of the algorithm is to select the leader processor with the smallest identifier. Since there is no obvious place to start the computation in the ring, so we start the computation everywhere.

The algorithm starts by each processor sending its identifier to the next processor in the ring. In the subsequent rounds, each processor performs the following computation synchronously:

1. The processor receives an identifier i from its predecessor processor.
2. The receiving processor compares i with its own identifier, id.
3. The processor sends minimum of these two values, i and id, to its successor processor.

The processor will know that its own identifier is the smallest in the ring when it receives its own identifier from the predecessor. Hence, the processor announces itself the leader by sending a "leader is" message around the ring. The formulation of the above idea is represented in Algorithm 1: RingLeader.

Correctness Idea Suppose n be the number of processors in the ring network. In the first round, each processor starts by sending its identifier. Then, in the next $n-1$ rounds, each processor accepts a "candidate is" message, computes the minimum m of the identifier i in the message and its own identifier id, and transmits identifier m to the next processor with a "candidate is" message. Let l be the smallest identifier of a processor. The first message sent by processor l will traverse the entire ring and will come back unchanged to processor l. At that point, processor l will know that it is the leader. Now, the processor l will notify all the other processors with a "leader is" message. This "leader is" message traverses the entire ring over the next n rounds.

Note. In the above analysis, the number of processors, n, does not have to be known by the processors in the network.

Remark. Observe that there are no deadlocks in the algorithm, that is, no two processors are both waiting on messages that need to come from the other.

Algorithm 1 RingLeader(id):

Input: The unique identifier, id, for the processor running this algorithm.
Output: The smallest identifier of a processor in the ring.
$M \leftarrow$[Candidate is id]
Send message M to the successor processor in the ring.
done $\leftarrow$**false**
repeat
    Get message M from the predecessor processor in the ring.
    if $M =$ [Candidate is i] **then**
        if $i = id$ then
            $M \leftarrow$[Leader is id]
            $done \leftarrow$**true**
        else
            $m \leftarrow \min\{i, id\}$
            $M \leftarrow$[Candidate is m]
    else {M is a "Leader is" message}
        $done \leftarrow$**true**
    Send message M to the next processing in the ring.
until $done$
return M {M is a "Leader is" message}

Performance Analysis

- Number of Rounds: Observe that the first "candidate is" message from the leader takes n rounds to traverse the ring. In addition, the "leader is" message initiated by the leader takes n more rounds to reach all other processors. Thus, there are $2n$ rounds.
- Message Complexity: To analyze the message complexity, the algorithm can be divided into two distinct phases. The first phase consists of the first n rounds. In this phase, in each round each processor sends and receives one "candidate is" message. Thus, $O(n^2)$ messages are sent in the first phase. The second phase consists of the next n rounds where the "leader is" message traverses around the ring. In this phase, a processor continues sending "candidate is" messages until the "leader is" message reaches it. Therefore, in this phase, the leader will send one message, the successor of the leader will send two messages, its successor will send three messages, and so on. Thus, the total number of messages sent in the second phase is Sigma i, $\sum_{i=1}^{n} i = O(n^2)$.

Conclusion. We are given an n-node distributed directed ring network with distinct node identifiers but no distinguished leader. The "RingLeader" algorithm finds a leader in the network using a total number of $O(n^2)$ messages. Moreover, the total message complexity of the algorithm is $O(n^2)$.

3.1.2 Asynchronous Solution

The asynchronous algorithm does not process in "lock step." Instead, in asynchronous algorithm "events" (not "clock ticks") determine the processing. The above synchronous algorithm can easily be converted into asynchronous model by structuring each round so that it consists following steps:

1. A message receiving step.
2. A processing step.
3. A message sending step.

Note. The correctness of the synchronous algorithm only depend on each processor receiving messages from its predecessor in the same sequence as they were sent. This condition still holds in the asynchronous model.

3.1.3 Summary of the Leader Election in the Ring

Given a directed ring with n processors, algorithm RingLeader performs leader election with $O(n^2)$ message complexity. For each processor, the local computing time is $O(n)$ and the local space used is $O(1)$. Algorithm RingLeader works under both the synchronous models. In the synchronous model, its overall running time is $O(n)$.

3.2 Leader Election in a Tree

Unlike leader election in the ring, a tree has a natural starting place for the computation: the external nodes. So, in a sense, electing leader in a tree is simpler than in a ring.

3.2.1 Asynchronous Solution

Asynchronous leader election algorithm for a tree assumes that a processor can perform a constant-time message check on an incident edge to see if a message has arrived from that edge.

The main idea of the algorithm is to use two phases: the accumulation phase and the broadcast phase.

- Accumulation Phase: In this phase, identifiers flow in from the external nodes of the tree. Each node keeps track of the minimum identifier l received from its neighbors, and After the node receives identifiers from all but one neighbor, it sends identifier l to that neighbor. This node, called *accumulation node*, determines the leader.
- Broadcast Phase: After identifying the leader, the algorithm starts the broadcast phase. In this phase, the accumulation node transmits (or broadcasts) the identifier of the leader in the direction of the external nodes.

Note. A tie condition may occur where two adjacent nodes become accumulation nodes. In this case, accumulation nodes broadcast to their respective "halves" of the tree.

Remark. Any node in the tree, even an external node, can be an accumulation node, depending on the relative speed of the processors.

The formulation of the above idea is in Algorithm 2: TreeLeader.

Performance Analysis During the accumulation phase, each processor sends one "candidate is" message. During the broadcast phase, each processor sends at most one "leader is" message. Each message has $O(1)$ size. Thus, the message complexity is $O(n)$.

3.2.2 Synchronous Solution

In the synchronous TreeLeader algorithm, all the processors begin a round at the same time until they halt. Thus, the messages in the accumulation phase march in the center of the tree and in the broadcast phase messages march out from the accumulation node.

Recall from the introduction to graph theory, the diameter of a graph is the length of a longest path between any two nodes in the graph. For a tree, a path between two external nodes achieves the diameter. In the synchronous leader election in a tree algorithm, the number of rounds is exactly equal to the diameter of the tree.

Algorithm 2 TreeLeader(id)

Input: The unique identifier, id, for the processor running the algorithm.
Output: The smallest identifier of a processor in the tree.
{Accumulation Phase}
let $d \geq 1$ be the number of neighbors of processors id.
$m \leftarrow 0$ {counter for messages recieved}
$l \leftarrow id$ {tentative leader}
repeat {begin in new **round**}

for each neighbor j **do**

check if a message from processor j has arrived
if a message M = [Candidate is i] from j has arrived **then**

$l \leftarrow min\{i, l\}$
$m \leftarrow m + 1$

until $m \geq d - 1$
if $m = d$ **then**

$M \leftarrow$ [leader is l]
for each neighbor $j \neq k$ **do**

send message M to processor j

return M {M is a "leader is" message}

else

$M \leftarrow$ [candidate is l]
send M to the neighbor k that has not sent a message yet

{Broadcast Phase}
repeat {begin a new **round**}

check if a message from processor k has arrived
if a message from k has arrived **then**

$m \leftarrow m + 1$
if M=[Candidate is l] **then**

$l \leftarrow min\{i, l\}$
$M \leftarrow$[leader is l]
for each neighbor j **do**

send message M to processor j

else {M is a "leader is" message}

for each neighbor $j \neq k$ **do**

send message M to processor j

until $m = d$
return M {M is a "leader is" message}

Algorithm 3 SynchronousBFS(v,s)

Input: The identifier v of the node (processor) executing this algorithm and the identifier s of the start node of the BFS traversal
Output: For each node v, its parent in a BFS tree rooted at s
repeat
  {begin a new round}
  if $v = s$ or v has recieved a message from one of its neighbors **then**
    set parent(v) to be a node requesting v to become its child (or **null**, if $v = s$)
    for each node w adjacant to v that has not contacted v yet **do**
      send a message to w asking w to become a child of v
until $v = s$ or v has recieved a message

Performance Analysis To analyze the local running time, we ignore the time spent waiting to begin a round. The synchronous algorithm for processor i takes time $O(d_i D)$, where d_i is the number of neighbors of processor i, and D is the diameter of the tree. In addition, processor i uses space d_i to keep track of the neighbors that have sent messages.

3.2.3 Summary of the Leader Election in the Tree

Given a tree with n nodes and with diameter D, algorithm TreeLeader performs leader election with $O(n)$ message complexity. Algorithm TreeLeader works under both the synchronous and asynchronous models. In the synchronous model, for each processor i, the local running time is $O(d_i D)$ and the local space use is $O(d_i)$, where d_i is the number of neighbors of processor i.

3.3 Breath-First Search

Suppose a general connected network of processors is given and we have identified a specific vertex, s, in this network as a source vertex.

3.3.1 Synchronous Solution

To construct the BFS tree, the algorithm proceeds in "waves". The "waves" propagate outward from the source vertex, s, and constructs the tree layer-by-layer from top down. The algorithm begins by identifying s as an external node, which is just s at the beginning. Then, in each round, each external node v sends a message to all its adjacent nodes (neighbors) that have not contacted v earlier, requesting that v wants to make them its children. These nodes respond by picking v as their parent, only if they have not already chosen parent.

Note. This algorithm takes an extraordinary advantage of the synchronization model. Specifically, to make the propagation process completely coordinated.

The formulation of the above idea is in Algorithm 3: SynchronousBFS.

Analysis In each round, the algorithm propagates out another level of the BFS tree. Thus, the running time of this algorithm is proportional to the depth of the BFS tree. In addition, it sends at most one message on each edge in each direction over the entire computation. Thus, the number of messages sent in a network of n nodes and m edges is $O(m+n)$.

3.3.2 Asynchronous Solution

At the expense of addition message complexity, the synchronous BFS algorithm can be transform into an asynchronous algorithm. In addition, in the asynchronous algorithm, each processor must know the total number of processor in the network. The asynchronous algorithm uses a so-called pulse technique. The pulse technique is a process in which a signal is passed down the BFS tree from the root s (pulse-down phase), and a signal is combined from the external nodes back up the tree to the root s (pulse-up phase).

In the asynchronous algorithm, the source node send out a "pulse" message that triggers the other processors to perform the next round in the computation. (As opposed to the synchronous version in which clock ticks determine each round in the computation.) By using pulse technique, the algorithm propagates the computation out from the source towards external nodes and builds up a BFS tree level by level.

When the processors at the external nodes receive a new pulse-down signal, only then the external nodes move to the next round. This holds at the source node too. The root s will not issue a new pulse-down signal until it has received all the pulse-up signals from its children. In this fashion, the algorithm ensures that the processors operate in "rough" synchronization.

The formulation of the above idea is in Algorithm 4: AsynchronousBFS.

Performance Analysis In each round, the root s sends "pulse-down" messages down the BFS tree constructed so far. When these messages reach the external level of the tree, the current external nodes try to extend the tree one more level, by issuing "make-child" messages to its children. When these children respond, (either by accepting the invitation to become a child or by rejecting it), the processors send a "pulse-up" message that propagates back up to s. The root s then starts the whole process all over again. As the result, The BFS tree grows level-by-level.

The node s repeats the pulsing process $n-1$ times to make sure that each node in the network has been included in the BFS tree. (Recall from graph theory, the BFS tree can have at most $n-1$ height.) Thus, unlike synchronous version, this algorithm achieves the synchronization from round to round by means of *message passing*. (Hence, the number of message needed to perform processors coordination is more than that needed in the synchronous algorithm.) In the network, each edge m has at most one message in each direction. (One "make-child" message in the direction of children and one response to a "make-child" message in the direction of parents.) Thus, the total message complexity for accepting or rejecting make-child message is $O(m)$, where m is the number

Algorithm 4 AsynchronousBFS(v,s,n)

Input: Identifier v of processor running this algorithm, Identifier s of the start node of BFS traversal, and the number n of netwwork nodes
Output: For each node v, its parent in the BFS tree rooted as s

```
C ← ∅        {verified BFS children for v}
Set A to be the set of neighbors of v        {candidate BFS children for v}
repeat
  {begin a new round}
  if parent(v)is defined or v = s then
    if parent(v) is defined then
      wait for a pulse-down message from parent(v)
    if C is not empty then
      {v is an internal node in the BFS tree}
      send a pulse down message to all nodes in C
      wait for a pulse up message from all nodes in C
    else
      {v is an external node in the BFS tree}
       for each nodeu ∈ A do
        send a make-child message to u
       for each node u ∈ A do
        get a message M from u and remove u from A
          if M is an accept-child message then
           add u to C
    send a pulse-up message to parent(v)
else
  {v ≠ s has no parent yet.}
  for each node w ∈ A do
    if w has sent v a make-child message then
      remove w from A        {w is no longer a candidate child for v}
       if parent(v) is undefined then
        parent(v)← w
        send an accept-child message to w
       else
        send a reject-child message to w
until (v has recieved message done) or (v = s and has pulsed-down n−1 times)
send a done message to all the nodes in C
```

of edges in the network. But, in each round, there are at most $O(n)$ pulse-up and pulse-down messages. Given that the source s runs $n-1$ rounds, there are at most $O(n^2)$ messages that are issued to coordinate the pulse. Therefore, the total message complexity of the algorithm in a network of m number of edges and n number of nodes is $O(n^2+m)$. Since we know that m is $O(n^2)$, this bound can be further simplified to $O(n^2+n^2) = O(2n^2) = O(n^2)$.

3.3.3 Summary of the Breadth-First Search

Given a network G with n vertices and m edges, one can compute a breath-first spanning tree in G in the asynchronous model using $O(n^2)$ messages.

3.4 Minimum Spanning Trees

A minimum spanning tree (MST) of a weighted graph is a spanning subgraph that is a tree and that has minimum weight.

Problem. Given a weighted undirected graph G, we are interested in finding a tree T that contains all the vertices in G and minimizes the sum of weights of the edges of T, that is $w = \sum_{e \in T} w(e)$.

3.4.1 Sequential Baruvka

Before we give an efficient distributed algorithm for the MST problem, let us first look into the oldest known minimum spanning tree algorithm - Baruvka's algorithm.

Remark. The Jarnik and Kruskal's algorithms have achieved their efficient running time by using a priority queue, which could be implemented using a heap. Surprisingly, the Baruvka's algorithm does not use the priority queue.

The pseudo-code for sequential Baruvka's algorithm is presented in Algorithm 5.

Analysis Each round performs the exhaustive searches to find the minimum-weight out going edge from cluster. These searches can be done in $O(m)$ time by going through the adjacency list of each vertex in each cluster. Note that the time $O(m)$ involves examine each edge (v,u) in G twice: once for node v and once for node u. In the while-loop, relabeling the vertices takes $O(n)$ time and traversing all edges in takes $O(n)$ time. Thus, since $n \leq m$, each round takes $O(m)$ time. In each round, the algorithm picks one out-going edge of each cluster, and then merges each new connected component of T into a new cluster. Thus, each old cluster must merge with at least one other old cluster. That is, in each round of the algorithm, the total number of clusters in T reduces by half. Therefore, the total number of rounds is $O(logn)$; hence, the total running time of Barurvka's algorithm is $O(mlogn)$.

Algorithm 5 BaruvkaMST(G)

Input: A weighted connected graph $G = (V, E)$, where $|V| = n$ and $|E| = m$
Output: A minimum spanning tree T for G.
Let T be a subgraph of G initially containing the vertices in V.
while T has fewer than $n - 1$ edges {T is not yet an MST} **do**

for each connected component C_i of T **do**

{Perform the MST edge addition procedure for cluster C_i}
Find the smallest-weight edge $e = (u, v)$ in E with $v \in C_i$ and $u \notin C_i$.
Add e to T (unless e is already in T)

return T

3.4.2 Synchronous Solution

In each round, the algorithm performs two critical computations:

1. Determine all the connected components.
2. Determine the minimum outgoing edge from each connected component.

Assumption. *For each node v, v stores a list of edges of T that are incident on v.*

Note that due to the assumption, each node v belongs to a tree, but v only stores information about its neighbors in T. In each round, the Synchronous Baruvka uses the algorithm TreeLeader twice:

1. To identify each connected component.
2. To find, for each connected component, the minimum-weight edge joining the components.

A round works as follows.

- Using the identifier of each node, it performs Leader-election computation to identify the connected components. Thus, the smallest identifier, *id*, of its nodes identifies each component.
- Next, each node v computes the minimum-weight edge e incident on v such that the endpoints of e are in different connected components. Note that if there is no such edge, the algorithm uses a fictitious edge with infinite weight. Then, the algorithm performs a leader election again. This time using the edges associated with each vertex and their weights. The second leader election computation yields, for each connected component C, the minimum-weight edge connecting C to another component.

Remark. The end of the algorithm is the round that computes the weight of the minimum-weight edge to be infinity.

Message Complexity There are $O(m)$ constant-size messages sent at each round. Since, there are $O(logn)$ rounds therefore, the overall message complexity is $O(mlogn)$.

4 Distributed Broadcast Routing

Broadcast routing is a message, which is sent from one router (processor) to all other routers (processors) in a network. When a router wants to share its information with all other routers in the network, this form of communication i.e. broadcasting, is used.

Assumption. *The network is fixed and does not change over time.*

The assumption implies that we must restrict ourselves to the study of static broadcasting algorithms.

Assumption. *The messages exchanged by the routers have constant size.*

The assumption implies that the message complexity is proportional to the total number of messages exchanged.

4.1 The Flooding Algorithm

Problem. A router s would like to send a message M to all other routers in the network.

1. A router s sends the message M to all its neighbors.
2. When a router $v \neq s$ receives an M from an adjacent router u, a router v simply rebroadcast M to all its neighbors, except for u itself.

Remark. The above flooding algorithm for braodcast routing is simple and requires no setup. However, its routing cost is high.

The setback with the above algorithm is that if we left it unmodified, it will cause an "infinite loop" of messages on the network, which contains a cycle. The two possibilities to avoid infinite loop problem are as follows.

4.1.1 Flooding with Hop Count Heuristic

To avoid infinite loop problem, one possibility is to associate memory or state with nodes running this algorithm.

- Add a hop counter to each message M.
- Decrement the hop counter for M each time a router processes M.

- **If** a hop counter in M ever reaches 0, **then** discard the message M.
- **Otherwise**, process the message M and send M to all neighbors.

We initialize the hop counter for a message M to the diameter of the network so that we can reach all routers while avoiding the creation of an infinite number of messages.

Analysis There is no additional space needed at the routers. The expected time required by a router to process a message is proportional to the number of neighbors of the router, since search and insertion in a hash table have $O(1)$ expected running time. The message complexity is $O((d_{max}-1)^D)$ in the worst case, where d_{max} is the maximum degree of the routers in the network.

4.1.2 Flooding with Sequence Number Heuristic

To avoid infinite loop problem, second possibility is that at each router, store a hash table (or other dictionary data structure). The hash table keeps track of which messages the router has already processed.

- When a router x creates a broadcast message M, it assigns a unique sequence number k to it. In other words, at the time of creation of a flooding message M, a router x tags a message M with the pair (x, k).
- When a router y (say) receives an M with tag (x, k), the router y checks whether the tag *(x, k)* is in its hash table.
- **If** the tag (x, k) is in router y's table, **then** the router y will discard the M.
- **Otherwise**, a router y adds the tag (x, k) in its table and rebroadcast the M to all its neighbors.

This approach will certainly solve the infinite loop problem, but is not space efficient. The common solution to this problem is to have each router keep only the latest sequence number. This solution is based on the following assumption.

Assumption. *If a router receives a message originated by x with sequence number k, it has probably already processed all the messages sent by x with sequence numbers less than k.*

Analysis The space needed by each router is $O(n)$ in the worst case. The expected time required by a router to process a message is proportional to the number of neighbors of the router, since search and insertion in a hash table have O(1) expected running time. The message complexity is $O(m)$, since the sequential number heuristic ends up sending a message along every edge in the network. Since m is usually much smaller than $O((d_{max}-1)^D)$, sequence numbers are generally preferable to hop counters.

5 Distributed Unicast Routing

Unicast routing involves setting up data structures in a network. The idea behind setting up data structures is to support point-to-point communication, where a single router has a message that it wants to have relayed to another router.

Problem. The unicast routing problem is to set up data structures at the nodes of the network that support the efficient routing of a message from an origin node to a destination node.

In unicast routing, our assumptions are the same as that of broadcast routing.

Assumption. *The network is fixed and the messages exchanged by the routers have constant size.*

In addition, we assume that networks have a positive weight $w(e)$ assigned to each edge e of the network, which represents the cost of sending a message through e.

5.1 The Distance Vector Algorithm

The distance vector algorithm is a distributed version of the Bellman and Ford algorithm for finding shortest paths in a graph. The *distance vector* is simply a bucket array implemented at node x as an adjacency list or adjacency matrix, which stores the length of the best-known path from router x to every other router y in the network and the first edge of such a path.

Notation. Notation $D_x[y]$ denotes the best-known path from router x to every other router y in the network. In addition, $C_x[y]$ denotes the first edge of such a path.

The distance vector algorithm always routes along shortest paths. The main idea of the algorithm is as follows.

1. For each router x store the distance vector. That is, $D_x[y]$ and the first edge $C_x[y]$.
2. Initially, for each edge (x, y), assign $D_x[y] = D_y[x] = w(x, y)$ and $C_x[y] = C_y[x] = (x, y)$.
3. All other D_x entries are set equal to $+\infty$.
4. Iteratively perform a series of rounds to *refine* each distance vector to find possibly better path until every distance vector stores the actual distance to all other routers.

The algorithm uses the technique called relaxation to refine the distance vector at each router. Recall from our Algorithms course, the term *relaxation* is used for an operation that tightens an upper bound.

Distributed Relaxation This is the setup phase of the algorithm, which is the characteristic property of the unicast routing algorithm. This setup consists of series of relaxation steps in each round. The setup accomplished as follows.

- At the beginning of a round, each router sends its distance vector to all of its neighbors in the network.
- After a router x has received the current distance vectors from each of its neighbors, it performs the following local computation for $n-1$ rounds:

for each router w connected to x **do**
 for each router y in the network **do**
  {relaxation}
  if $D_w[y] + w(x,w) < D_x[y]$ **then**
   {a better route from x to y through w has been found}
   $D_x[y] = D_w[y] + w(x,y)$
   $C_x[y] = (x,w)$.

Performance Note that in each round, every vertex x sends a vector of size n to each of its neighbors. Therefore, the time and message complexity for x to complete a round is $O(d_x n)$, where d_x is the degree of x and n is the number of routers in the network.

Actually, using the following fact, one can improve the performance of the distance vector algorithm.

Fact. *The algorithm can be iterated D number of rounds, where D is the diameter of the network.*

Instead of $n-1$ rounds, one can iterates this algorithm (particularly, relaxation) for D rounds. The reason is that the distance vector will not change after D rounds.

Routing Algorithm After the setup, the distance vector of a router x contains two objects: (1) the actual distances of x to the other routers in the network and, (2) the first edge of such path. Therefore, once the setup of the data structure is completed, the routing algorithm is straightforward:

if a router x receives a message M intended for router y **then**
 Router x sends M to router y along the edge $C_x[y]$.

The following inductive argument establishes the correctness of the distance vector algorithm.

Lemma. *At the end of round i, each distance vector stores the shortest path to every other router restricted to visit at most i other router along the way.*

This fact is true at the beginning of the algorithm, and the relaxations done in each round ensure that it will be true after each round as well.

Remark. Recall from the CLRS, we must show that this loop invariant holds prior to the first iteration of the round, that each iteration of the round maintains the invariant, and that the invariant provides a useful property to show correctness when the round terminates.

Analysis It is easy to see that this algorithm has a significant setup cost as compared to flooding algorithm. The total number of messages passed in each round is proportional to n times the sum of all the degrees in the network. This is because the number of messages sent and received at a router x is $O(d_x n)$, where d_x is the degree of a router x. Thus, there are $O(nm)$ messages per round. Hence, the total message complexity for the setup of the distance vector algorithm is $O(Dnm)$.

Note. In the worst case the total message complexity is $O(n^2 m)$ for the distance vector algorithm.

After the setup is completed, each router stores a bucket array or distance vector with $n-1$ elements, which takes $O(n)$ space, and processes a messages in $O(1)$ expected time.

5.2 The Link-State Algorithm

The link-state algorithm is a distributed version of the Dijkstra's algorithm for finding shortest paths in a graph. For the link-State algorithm, our assumptions are the same as that of distance vector algorithm.

Assumption. *The network is fixed and that a positive weight w(e) assigned to each edge e of the network.*

The major difference between link-state algorithm and the distance vector algorithm is that the link-state algorithm computes a single communication round that requires lots of global communication throughout the entire network, whereas the distance vector algorithm performs its setup in a series of rounds that each requires only local communication between adjacent routers.

The main idea of the link-state algorithm consists of the broadcast phase and the computation phase and is as follows:

- The setup of data structure begins with each router x broadcasting its *status* to all other routers in the network using a flooding routing algorithm with sequential number heuristic.

Here, the *status* of x means the weights of its incident edges. The reason we used the sequential number heuristic is that it requires no prior setup. After the broadcast phase, each router knows the entire network.

- For each router x, run Dijkstra's shortest path algorithm to determine the shortest path from x to every other router y and the first edge $C_x[y]$ of such path.

Analysis The internal computation takes $O(mlogn)$ times using standard implementations of Dijkstra's algorithm, or $O(nlogn+m)$ using more sophisticated data structures. The data structure constructed by the setup at each router has $O(n)$ space and supports the processing of message in $O(1)$ expected time. As far as message complexity is concern, a total of m constant-size messages are broadcast, each of which causes m messages to be sent by the flooding algorithm in turn. Thus, the overall message complexity of the setup is $O(m^2)$.

6 Multicast Routing

Broadcast routing and unicast routing can be viewed as being at the two extremes of a communication spectrum. Somewhere in the middle of this spectrum is multicast routing. Multicast routing involves communication with a subset of hosts on a network, called a multicast group.

6.1 Reverse Path Forwarding Algorithm

The reverse path forwarding (RPF) algorithm adapts and fine-tunes the flooding algorithms for broadcast routing to multicast routing. To broadcast a multicast message along the shortest path from a source, the RPF designed to work in combination with existing shortest-path routing tables available at every router.

Main Idea of RPF The algorithm starts with some host that wants to send a message to a group g.

- The host sends that message to its local router s so that s can send the message to all of its neighboring routers.
- When a router x receives a multicast message that originated at the router s from one of its neighbors y, x checks its local routing table to see if y is on x's shortest path to s.
- If y is not on x's shortest path to s, then x discards the packet sent from y and sends back to y a special prune message that tells y to stop sending multicast messages from s intended for group g. For, in this case, the link from y to x is not in the shortest path tree from s. Note that the prune message includes the name of the source s and the group g.
- On the other hand, If y is on x's shortest path to s, then the router x replicates the message and sends it out to all of its neighboring routers, except for y itself. For, in this case, the link from y to x is on the shortest path tree from s.

This mode of broadcast communication extends outward from s along the shortest path tree T from s until it floods the entire network.

Fact. *If only a small number of the routers have clients, who want to receive multicast messages that are sent to the group g, then clearly broadcasting a multicast message to every router in the network is wasteful.*

To deal with this waste, the RPF algorithm provides an additional type of message pruning. In particular, if a router x, at an external node of T, determines that it has no clients on its local network that are interested in receiving messages for the group g, then x issues a prune message to its parent y in T telling y to stop sending it messages from s to the group g.

Note. This message in effect tells y to remove the external node x from the tree T.

Pruning In parallel, many external nodes of T issue prune messages; as a result there can be many external nodes removed at the same time. It is easy to see that the removal of some external nodes of T may create new external nodes. Therefore, the RPF algorithm has each router x continually test if it has become an external node in T. And, if x has become an external node, then x should test if it has any client hosts on its local network that are interested in receiving multicast messages for the group g. Again, if there are no such clients (in x's local network), then x sends its parent in T a prune message for multicast messages from s to the group g. This external-node pruning continues until all remaining external nodes in T have clients wishing to receive multicast messages to the group g. At this point, the RPF algorithm has reached a steady state.

However, this steady state cannot be locked in forever, because some client hosts may want to start receiving messages for the group g. Formally, we say at least one client h wants to join the group g. Since RPF provides no explicit way for clients to join a group, the only way h can start receiving multicasts to group g is if h's router x is receiving those packets. But if x has been removed from T, then this will not happen.

Thus, one additional component of the RPF algorithm is that the prunes that are stored at a node in the Internet are timed out after a certain amount of time. When such a time-out occurs, say for a prune coming from a router z to the router x holding z's previous prune message, then x resumes sending multicast packets for the group g to z. Therefore, if a router really intends not to receives or process certain types of multicast messages, then it must continually inform its upstream neighbors of this desire by using prune messages.

Performance Initially, the RPF sends out $O(m)$ messages, where m is the number of connections in the network. However, after the first wave of prune propagate through the network, the complexity of multicasting reduces to $O(n)$ messages per multicast message, where n is the number of routers.

In terms of additional storage at the router, RPF requires that each router store every prune message it receives until that prune message times out. Then, in the worst case a router x may have to store as many as $O(|S| \cdot |\mathcal{G}| d_x)$ prune messages, where S is the set of sources, $\mathcal{G}$ is the set of groups, and d_x is the degrees of x in the network. So, in terms of additional storage at the router, the RPF algorithm is not too efficient.

6.2 Center-Based Trees Algorithm

The message efficiency of the center-based trees algorithm is better than that of the reverse path forwarding algorithm. The main idea of the center-based trees method is as follows:

- For each group, choose a single router z on the network that will act as the "center" or "rendezvous" node for the group g.

- The router z forms the root node of the multicast tree T for g. The algorithm uses z for sending messages to routers that are a part of this group.
- Any source wants to send a multicast message to the group g, it first sends the message toward the center z. In the simplest form of the algorithm, this message moves all the way to z, and once z receives the message, it then broadcast it to all the nodes in T.
- Thus, each router x in T knows that if x receives a multicast message for g from its parent in T, then x replicates this message and sends it to all of its neighbors that correspond to its children in T.
- Likewise, in the simplest form of the center-based tree algorithm, if x receives a multicast message from any other neighbor different than its parent in T, then it sends this message up the tree to z. Such a message is coming from some source and should be sent up to z before being multicast to T.

Since, the algorithm must explicitly maintain the multicast tree T for each group. Therefore, we must provide a way for routers to join the group g.

Join Operation A *join* operation for a router x starts by having router x send a *join* message toward the center node z.

- Any other router y receiving a *join* message from a neighbor t looks up to see which of y's neighbors u is on the shortest path to z, and then y creates and stores an internal record showing that y now has a child t and a parent u in the tree T.
- If y was already in the tree T (that is, there was already a record saying that u was y's parent in the tree T), then this completes the *join* operation. This means that the router x is now a connected external node in the tree T.
- (Otherwise) If y was not already in the tree T, then y propagates the *join* message up to its (new) parent u in T.

Leave Operation For routers wanting to leave group g, either through explicit leave messages or because a join record times out, we essentially reverse the action we performed for a *join* operation.

Performance This algorithm only sends messages along the multicast tree T for group g, so its message complexity is $O(\mid T \mid)$for each multicast. In addition, since each router only stores the local structure of T, the maximum number of records that need to be stored at any router x is $O(\mid \mathcal{G} \mid d_x)$, where $\mathcal{G}$ is the set of groups and d_x is the degree of router x in the network.

Parallel Algorithms

This section is about the most fundamental aspect of parallelism, namely, parallel algorithms. In order to properly design such algorithms, one needs to have clear understanding of the model of computation underlying the parallel computer.

7 Models of Parallel Computation

Any computer operates by executing instructions on data. A stream of instructions (the algorithm) tells the computer what to do at each step. A stream of data (the input to the algorithm) is affected by these instructions. There are four classes of computers depending upon how many streams are there:

1. Single Instruction stream, Single Data stream (SISD)
2. Multiple Instruction stream, Single Data stream (MISD)
3. Single Instruction stream, Multiple Data steam (MIMD)
4. Multiple Instruction stream, Multiple Data stream (MIMD)

7.1 SISD Computers

A computer in this class consists of a single processing unit receiving a single stream of instructions that operates on a single stream of data. At each step during the computation, the control unit emits one instruction that operates on a data obtained from the memory unit. For example, such an instruction may tell the processor to perform some arithmetic or logic operation on the data and then put it back in memory.

Problem 1. It is required to compute the sum of n numbers.

7.2 MIMD Computers

A computer in this class consists of N processors each with its own control unit share a common memory unit where data reside. There are N streams of instructions and one stream of data. At each step, all the processors operate upon one data received from memory simultaneously, each according to the instruction it receives from its control. Thus, parallelism is achieved by letting the processors do different things at the same time on the same data.

Problem 2. It is required to determine whether a given positive integer z has no divisors except 1 and itself.

7.3 SIMD Computers

In this class, a parallel computer consists of N identical processors. Each of the N processors possesses its own local memory where it can store both program and data. All processors operate under the control of a single instruction stream issued by a central control unit. The processors operate synchronously: At each step, all processors execute the same instruction, each on a different data. Sometimes, it may be necessay to have only a subset of the processors execute an instruction. This information can be encoded in the instruction itself, thereby telling a processor whether it should be *active* (and execute the instruction) or *inactive* (and wait for the next instruction). There is a mechanism, such as a global clock, that ensures lock-step operaton. Thus, processors that are inactive during an instruction or those that complete execution of the instruction before others may stay idle until the next instruction is issued.

On SIMD computer, it is desirable for the processors to be able communicate among themselves during the computation in order to exchange data or intermediate results. This can be achieved in two ways, giving rise to two subclasses: SIMD computers where communication is through *shared memory* and those where it is done via an *interconnectionnetwork*.

7.3.1 Shared-Memory (SM) SIMD Computers

In the literature, this class is also known as the Parallel-Access Machine (PRAM) model. Here, when two processors wish to communicate, they do so through the shared memory. For example, suppose processor i wishes to pass a number to processor j. This is done in two steps. First, processor i writes the number in the shared memory at a given location known to processor j. Then, processor j reads the number from that location.

The class of shared-memory SIMD computes can be further divided into four subclasses, according to whether two or more processors can gain access to the same memory location simultaneously:

1. **Exclusive-Read, Exclusive-Write (EREW) SM SIMD Computers**. Access to memory locations is exclusive. In other words, no two processors are allowed simultaneously to read from or write into the same memory location.

2. **Concurrent-Read, Exclusive-Write (CREW) SM SIMD Computers**. Multiple processors are allowed to read from the same memory location but the right to write is still exclusive.

3. **Exclusive-Read, Concurrent-Write (ERCW) SM SIMD Computers**. Multiple processors are allowed to write into the same memory location but read accesses remain exclusive.

4. **Concurrent-Read, Concurrent-Write (CRCW) SM SIMD Computers**. Both multiple-read and multiple-write privileges are granted.

Problem 3. Consider a very large computer file consisting of n distinct entries. We shall assume for simplicity that the file is not sorted in any order. It is required to determine whether a given item x is present in the file in order to perform standard database operation, such as *read*, *update*, or *delete*.

7.3.2 Interconnection-Network SIMD Computers

Here, the M locations of the shared memory are distributed among the N processors, each receiving M/N locations. In addition, every pair of processors are connected by a two-way line. At any step during the computation, processor P_i can receive a data from P_j and send another one to P_k (or to P_j). Consequently, each processor must contain (i) a circuit of cost $f(N-1)$ capable of decoding a $log(N-1)$-bit address. This allows the processor to select one of the other $N-1$ processors for communicating; and (ii) a circuit of cost $f(M/N)$ capable of decoding a $log(M/N)$-bit address provided by another processors.

Simple Networks for SIMD Computers In most applications a small subset of all pairwise connections is usually sufficient to obtain a good performance. The most popular of these networks are briefly outlined as follows.

- **Linear Array**. The simplest way to interconnect N processors is in the form of a one-dimensional array. Here, processor P_i is linked to its two neighbors P_{i-1} and P_{i+1} through two-way communication line. Each of the end processors, namely, P_1 and P_N, has only one neighbor.

- **Two-Dimensional Array**. A two-dimensional network is obtained by arranging the N processors into an $m \times m$ array, where $m = \sqrt{N}$. The processor in row j and column k is denoted by $P(j,k)$, where $0 \leq j \leq m-1$ and $0 \leq k \leq m-1$. A two-way communication line links $P(j,k)$ to its neighbors $P(j+1,k)$, $P(j-1,k)$, $P(j,k+1)$, and $P(j,k-1)$. Processors on boundary rows and columns have fewer than four neighbors and hence fewer connections. This network is also known as the *mesh*.

- **Tree Connection or tree-connected computer**. In this network, the processors form a *complete binary tree*. Such a tree has d levels, numbered 0 to $d-1$ and $N = 2^d - 1$ nodes each of which is a processor. Each processor at level i is connected by a two-way line to its parent at level $i+1$ and to its two children at level $i-1$. The root processor (at level $d-1$) has no parent and the leaves (all of which are at level 0) have no children.

- **Perfect Shuffle Connection**. Let N processors $P_0, P_1, \ldots, P_{N-1}$ be available where N is a power of 2. In the *perfect shuffle* interconnection a one-way links P_i to P_j, where $j = \begin{cases} 2i & for\ 0 \leq i \leq N/2-1 \\ 2i+1-N & for\ N/2 \leq i \leq N-1 \end{cases}$. In addition to these *shuffle* links, two-way lines connecting every even-numbered processor to its successor are sometimes added to the network. These connections are called the *exchange* links. In this case, the network is known as the *shuffle-exchange* connection.

- Cube Connection. Assume that $N = 2^q$ for some $q \geq 1$ and let N processors be avaiable $P_0, P_1, \ldots, P_{N-1}$. A *q-dimensional cube* (or *hypercube*) is obtained by connecting each processor to q neighbors. The q neighbors P_j of P_i are defined as follows: The binary representation of j is obtained from that of i by complementing a single bit.

There are quite a few other interconnection networks besides the ones just pointed out. The decision about which of these to use largely depends on the application and in particular on such factors as the kinds of computations to be performed, the desired speed of execution, and the number of processors available.

Problem 4. It is required to compute the sum of n numbers $x_1, x_2, \ldots, x_n$ on tree-connected SIMD computer.

7.4 MIMD Computers

Here we have N processors, N stream of instructions, and N stream of data. Each processor possesses its own control unit in addition to its local memory and arithmetic and logic unit. This makes these processors more powerful than the ones used for SIMD computers. Each processor operates under the control of an instruction stream issued by its control unit. Thus, the processors are potentially all executing difference programs on different data while solving different subproblems of a single problem. This means that the processors typically operate asynchronously.

As with SIMD computers, communication between processors is performed through a shared memory or an interconnection network. MIMD computers sharing a common memory are often referred to as *multiprocessors* (or *tightly coupled machines*) while those with an interconnection network are known as *multicomputers* (or *loosely coupled machines*). Multicomputers are sometimes referred to as *distributed systems*. The distinction is usually based on the physical distance separating the processors and is therefore often subjective. A general rule is if all the processors are in close proximity of one another (they all are in the same room, say), then they are a multicomputer; otherwise (they are in different cities, say) they are a distributed system. As we have seem in the study of distributed algorithms, because processors in a distributed system are so far apart, the number of data exchanges among them is significantly more important than the number of computational steps performed by any of them.

Problem 5. Generating and searching game trees. For instance, a computer program that play chess do so by using these techniques.

8 Analyzing Algorithms

This part is concerned with two aspects of parallel algorithms: their design and their analysis. A number of algorithm design techniques were illustrated, in the

problems at the end of each Section, in connection with our description of the different models of parallel computation. The *algorithm analysis* refers to the process of derermining how good an algorithm is i.e., how fast, how expensive to run, and how efficient it is in its use of the available resources.

8.1 Running Time

The running time is one of the most important measures in evaluating parallel algorithms. The running time of the algorithm is defined as the time taken by the algorithm to solve a problem on a parallel computer, that is, the time elapsed from the moment the algorithm starts to the moment it terminates. If the various processors do not all begin and end their computation simultaneously, then the running time is equal to the time elapsed between the moment the first processor to begin computing starts and the moment the last processor to end computing terminates.

8.2 Counting Steps

As we have learnt in the CLRS, before actually implementing an algorithm on a computer, it is customary to perform a theoretical analysis of the time it will take to solve the computational problem at hand. This is usually done by counting the number of basic operation or *steps*, executed by the algorithm in the worst case. This gives an expression describing the number of such steps as a function of the input size. Of course, the definition of what constitutes a step varies from one theoretical model of computation to another. For details, see CLRS, Chapter 3. The running time of a parallel algorithm is usually obtained by counting two kinds of steps: *computational steps* and *routing steps*. A computational step is an arithmetic or logic operation performed on a data within processor. On the other hand, in a routing step, data travels from one processor to anther via the shared memory or through the communication network. For a problem of size n, the parallel worst-case running time of an algorithm, a function of n, will be denoted by $t(n)$.

Problem 6. In problem 2 we studied a parallel algorithm that searches a file with n entries on an N-processor EREW SM SIMD computer. Compute the parallel worst-case running time the problem.

8.3 Lower and Upper Bounds

Recall from the CLRS, given a computational problem for which a new sequential algorithm has just been designed, it is common practice among algorithm designers to ask the following questions:

1. Is it the fastest possible algorithm for the problem?
2. If not, how does it compare with other existing algorithms for the same problem?

On Question 1 The answer to the first question is usually obtained by comparing the number of steps executed by the algorithm to a known lower bound on the number of steps required to solve the problem in the worst case.

Example. Say that we want to compute the product of two $n \times n$ matrices. Since the resulting matrix has n^2 entries, at least this many steps are needed by any matrix multiplication algorithm simply to produce the output.

Lower bounds, such as the one in the above example, are usually known as *trivial* lower bounds, as they are obtained by counting the number of steps needed during input and/or output. A more sophisticated lower bound is derived in the next example.

Example. For the problem of sorting (see the Author's lecture notes on design and analysis of algorithms), there are $n!$ possible permutations of the input and $logn!$ (that is, on the order of $nlogn$) bits are needed to distinguished among them. Therefore, in the worst case, any algorithm for sorting requires on the order of $nlogn$ steps to recognize a particular input.

If the number of steps an algorithm executes in the worst case is *of the same order as* the lower bound, then the algorithm is the fastest possible and is said to be *optimal*. Otherwise, a faster algorithm may have to be invented, or it may be possible to improve the lower bound. In any case, if the new algorithm is faster than all known algorithms for the problem, then we say that it has established a new *upper bound* on the number of steps required to solve that problem in the worst case.

On Question 2 Once we settled the question 1, the question 2 is therefore settled by comparing the running time of the new algorithm with the existing upper bound for the problem (established by the fastest previously known algorithm).

Example. To date, no algorithm is known for multiplying two $n \times n$ matrices in n^2 steps. The standard textbook such as CLRS requires on the order of n^3 operations. However, the upper bound on this problem is established by an algorithm requiring on the order of n^x operations at most, where $x < 2.5$. By contrast, several sorting algorithms exists that require on the order of at most $nlogn$ operations and hence are optimal.

Finally, let us review the notion of *on the order of* to express lower and upper bounds. For details, see the Author's lecture notes on Algorithms. Let $f(n)$ and $g(n)$ be functions from the positive integers to the positive reals:

- The function $g(n)$ is said to be *of order at least* $f(n)$, denoted $\Omega(f(n))$, if there are positive constants c and n_0 such that $g(n) \geq cf(n)$ for all $n \geq n_0$.
- The function $g(n)$ is said to be *of order at most* $f(n)$, denoted $O(f(n))$, if there are positive constants c and n_0 such that $g(n) \leq cf(n)$ for all $n \geq n_0$.

This notation allows us to concentrate on the dominating term in an expression describing a lower or upper bound and to ignore any multiplicative constants.

For matrix multiplication, the lower bound is $\Omega(n^2)$ and the upper bound $O(n^{2.5})$. For sorting, the lower bound is $\Omega(nlogn)$ and the upper bound $O(nlogn)$.

The treatment of lower and upper bounds in this section has so far concentrated on sequential algorithms. The same general ideas also apply to parallel algorithms while taking two additional factors into consideration:

- the model of parallel computation used and
- the number of processors involved.

Example. An $n \times n$ mesh-connected SIMD computer is used to compute the sum of n^2 numbers. Initially, there is one number per processor. Processor $P(n-1, n-1)$ is to produce the output. Since the number initially in $P(0,0)$ has to be part of the sum, it must somehow find its way to $P(n-1, n-1)$. This requires at least $2(n-1)$ routing steps. Thus the lower bound on computing the sum is $\Omega(n)$ steps.

8.4 Speedup

A good indication of the quality of a parallel algorithm is the speedup it produces with respect to the best available sequential algorithm for that problem.

$$Speedup = \frac{worst-case-running-time-of-fastest-known-sequential-algorithm-for-problem}{worst-case-running-time-of-parallel-algorithm}$$

Example. Determine the speedup in the problem 3.

8.5 Number of Processors

Another important criterion in evaluating a parallel algorithm is the number of processors it requires to solve a problem. Clearly, the large number of processors an algorithm uses to solve a problem, the more expensive the solution becomes to obtain. For a problem of size n, the number of prcessors required by an algorithm, a function of n, will be denoted by $p(n)$.

Remark. Sometimes the number of processors is a constant independent of n.

Determine the number of processors used in problem 3 and problem 4.

8.6 Cost

The cost of a parallel algorithm equals the number of steps executed collectively by all processors in solving a problem in the worst-case. Hence Cost = Parallel running time $\times$ Number of processors used. This definition assumes that all processors execute the same number of steps. If this is not the case, then cost is an upper bound on the total number of steps executed. For a problem of size n, the cost of a parallel algorithm, a funciton of n, will be denoted by $c(n)$. Thus, $c(n) = p(n) \times t(n)$.

Assume that a lower bound is knwon on the number of sequential operations required in the worst case to solve a problem. If the cost of a parallel algorithm for that problem matches this lower bound to within a constant multiplicative factor, then the algorithm is said to be *cost optimal*.

Problem. Determine whether cost is optimal or not in problem 3 and in problem 4.

A method for obtaining model-independent lower bounds on parallel algorithms is as follows. Let $\Omega(T(n))$ be a lower bound on the number of sequential steps required to solve a problem of size n. Then $\Omega(T(n)/N)$ is a lower bound on the running time of any parallel algorithm that uses N processors to solve that problem.

Problem. Determine the lower bound on a parallel sorting algorithm.

When no optimal sequential algorithm is known for solving a problem, the efficiency of a parallel algorithm for that problem is used to evaluate its cost. This is defined as follows:

$$Efficiency = \frac{worst-case-running-time-of-fastest-known-sequential-algorithm-for-problem}{cost-of-parallel-algorithm}$$

Usually, $efficiency \leq 1$;

Problem. Determine the efficency of a parallel algorithm for multiplying two $n \times n$ matrices.

9 Two Useful Procedures

In the EREW SM SIMD model, no two processors can gain access to the same memory location simultaneously. However, two situations may arise in a typical parallel algorithm.

1. All processors need to read a datum held in a particular location of the common memory.
2. Each processor has to compute a function of data held by other processors and there needs to receive these data.

Clearly, a way must be found to efficiently simulate these two operations that cannot be performed in one step on the EREW model. In this section, we present two procedures for performing these simulations.

9.1 Broadcasting a Data

Assume that N processors $P_1, P_2, \ldots, P_N$ are available on a EREW SM SIMD computer. Let D be a location in memory holding a data that all N processors need at a given moment during the execution of an algorithm. We present this process formally as procedure BROADCAST. The procedure assumes that

Algorithm 6 Broadcastin a data

procedure BROADCAST (D, N, A)
Step 1. Processor P_1
(1a) reads the value in D,
(1b) store it in its own memory, and
(1c) writes it in $A(1)$
Step 2. for $i = 0$ to $(logN - 1)$ do
for $j = 2^i + 1$ to 2^{i+1} do in parallel
Processor P_j
(a) reads the value in $A(j - 2^i)$
(b) stores it in its won memory, and
(c) writes it in $A(j)$.

presence of an array A of length N in memory. The array is initially empty and is used by the procedure as a working space to distribute the contents of D to the processor. Its ith position is denoted by $A(i)$.

Analysis Since the number of processors having read D doubles in each iteration, the procedure terminates in $O(logN)$ time. The memory requirement of BROADCAST is an array of length N.

9.2 Computing All Sums

References

[1] Hagit Attiya and Jennifer Welch, Distributed Computing: Fundamentals, Simulations and Advanced Topics, John Wiley and Sons, 2004.

[2] Michael T. Goodrich and Roberto Tamassia, Algorithm Design: Foundations, Analysis, and Internet Examples, John Wiley and Sons, 2002.

[3] Nancy A. Lynch, Distributed Algorithms, Morgan Kaufmann Publishers, Inc., 1996.

[4] Gerard Tel, Introduction to Distributed Algorithms, 2nd Edition, Cambridge University Press, 2000.

[5] Cormen, Leiserson, Riverst, and Stein (CLRS), Introduction to Algorithm, MIT press, Cambridge, MA, 2009.