Parallel and Distributed Algorithms

Prof. Dr. G. Schnitger
Johann Wolfgang Goethe-Universität
Institut für Informatik
Frankfurt am Main

December 15, 2009
If you find errors or if you have suggestions for improvements, please let us know!

poloczek@thi.informatik.uni-frankfurt.de
ionescu@thi.informatik.uni-frankfurt.de
Contents

1 Introduction 7

1.1 Asymptotic Notation ........................................... 10
1.2 Important Inequalities ......................................... 10
1.3 Graphs ....................................................... 12
1.4 Some Basics from Probability Theory .................... 14

I Distributed Memory 27

2 Communication Patterns 29

2.1 Trees ......................................................... 30
2.2 Meshes ...................................................... 35
2.3 Hypercube Architectures .................................... 42
  2.3.1 Simulating Trees and Meshes ......................... 45
  2.3.2 Butterflies ............................................ 47
  2.3.3 Routing on Hypercubes ............................... 49

3 The Message Passing Interface 53

3.1 Analyzing MPI Programs: The LogGP Model ........... 55
3.2 Work, Speedup and Efficiency ............................. 61
   3.2.1 Varying the Number of Processors ................ 63
3.3 Partitioning and Mapping .................................. 65

4 Parallel Linear Algebra 67

4.1 Matrix-Vector Multiplication ............................... 67
## CONTENTS

4.2 Matrix Multiplication .......................................................... 69
4.3 Solving Linear Systems .......................................................... 73
  4.3.1 Back Substitution .......................................................... 73
  4.3.2 Gaussian Elimination ....................................................... 75
  4.3.3 Iterative Methods ......................................................... 77
  4.3.4 Finite Difference Methods .............................................. 80
4.4 The Discrete Fourier Transform ............................................. 83
  4.4.1 The Binary Exchange Algorithm ...................................... 88
  4.4.2 The Transpose Algorithms .............................................. 89
4.5 Conclusion ................................................................. 91

5 Algorithms for Hard Problems ................................................. 93
  5.1 Monte Carlo Methods ....................................................... 93
    5.1.1 Markov Chain Monte Carlo Methods .................................. 95
    5.1.2 Pseudo Random Number Generators ................................ 100
  5.2 Backtracking ............................................................... 102
  5.3 Branch & Bound ............................................................ 103
  5.4 Alpha-Beta Pruning ........................................................ 104
  5.5 Conclusion ................................................................. 110

6 Load Balancing ................................................................. 111
  6.1 Work Stealing ............................................................... 113
  6.2 Work Sharing ............................................................... 116
  6.3 Conclusion ................................................................. 121

7 Termination Detection .......................................................... 123

8 Divide & Conquer and Dynamic Programming ............................... 129
  8.1 Sorting ................................................................. 129
    8.1.1 Parallelizing Quicksort .............................................. 130
    8.1.2 Sample Sort ........................................................... 132
    8.1.3 Parallelizing Mergesort ............................................. 133
  8.2 Dynamic Programming ..................................................... 135
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>12.3.2</td>
<td>Linear Programming</td>
<td>206</td>
</tr>
<tr>
<td>12.3.3</td>
<td>Network Flow</td>
<td>208</td>
</tr>
<tr>
<td>12.3.4</td>
<td>Parallelizing Sequential Programs</td>
<td>211</td>
</tr>
<tr>
<td>12.4</td>
<td>Conclusion</td>
<td>214</td>
</tr>
</tbody>
</table>

References | 215
Chapter 1

Introduction

We consider two models of parallel computing, multicomputers and multiprocessors. A *multicomputer* consists of processors (such as workstations or personal computers) and an interconnection network. Processors communicate by exchanging point-to-point mes-
sages. These messages are routed by an interconnection network composed of switches which cooperate according to various network technologies such as Gigabit Ethernet, Myrinet or InfiniBand.

Each processor has direct access only to its local memory and has to communicate any request for missing information. Thus multicomputers are *distributed memory* computers with the advantage of a large aggregate memory capacity and the disadvantage of (possibly) slow access time.

![Diagram of a distributed memory model](image)

$k$ processors in a distributed memory model.

The performance of a multicomputer program is determined by its *compute time* (the time spent by processors when computing on their local data) and its *communication time* (the time spent when sending, receiving or waiting for messages). Typically, the communication performance is the weak spot of a multicomputer. Fast Ethernet or Gigabit Ethernet have *latencies*\(^1\) of 100 µs, the latest generations of Myrinet and InfiniBand have latencies of

\(^1\)The latency of a link is defined as the time from the start of a transmission to the end of the reception for a short message.
as low as 2\(\mu s\) and 1.32 \(\mu s\) respectively, but still a simple compute step is by a factor of a few thousands faster than a simple communication step. Moreover, although the bandwidth of Fast Ethernet (100 Mbit/sec), Gigabit Ethernet (1 Gbit/sec), Myrinet (1.92 Gbit/sec) and InfiniBand (up to 10 Gbit/sec) seems considerable, current technologies do not allow to transport long message streams without interruption and thus slow down communication time even further. On the other hand, due to the inexpensive switching technology, multicomputers reach a superior performance/price ratio.

**Multiprocessors** form the second important parallel computer model. A multiprocessor consists of several CPUs which access their local cache directly and access a *shared memory* with special hardware to cut down on communication costs. However, in affordable multiprocessors, only few processors are supported.\(^2\) The production cost of multiprocessors has been reduced considerably and multiprocessors such as multicore Pentiums or multicore Opterons have reached the mass market.

![Diagram of shared memory model](image)

\(k\) processors in a shared memory model.

It is not inevitable that multicomputers and multiprocessors are competing parallel computer models. Just to the contrary, hybrid systems, i.e., clusters of multiprocessors such as the CSC cluster in Frankfurt, are becoming standard and receive special software support (MPI version 2).

We assume throughout that all processors work according to the same program. This is a reasonable restriction, since parallel programs should be portable to parallel systems of arbitrary sizes; moreover this approach is still flexible, since processors have different a priori information such as their ID and hence we can emulate multiple programs within the single-program restriction.

We allow asynchronous computing for multicomputers, but require multiprocessors to be fully synchronous. As a consequence, our algorithms for multicomputers are easily applicable for real life clusters, whereas our algorithms for multiprocessors, at least at this

\(^2\)This multiprocessor model is also called a *centralized multiprocessor*. A *distributed multiprocessor* uses an interconnection network to exchange messages between different processors and is capable to support more processors. Whereas a multiprocessor has a single global address space, a multicomputer has disjoint local address spaces.
moment in time, live in a “theoretical world” of synchronous computation with shared memory access and insignificant communication delays. This ideal setting is currently unrealistic, but it allows us to investigate fundamental questions such as a classification of computing problems into parallelizable respectively inherently sequential problems.

The mathematical prerequisites are described in the following sections. After introducing “communication patterns” we design and analyze parallel algorithms for the message passing paradigm in Part I. Algorithmic problems and their solutions for shared memory models are described in Part II. When is it possible to design superfast parallel algorithms? In Chapter 12 we encounter a large class of problems which in all likelihood do not possess superfast parallel algorithms: if one problem in this class is solvable superfast, then all problems in this class are. We conclude in Part ?? with a short survey of some important distributed algorithms.

We recommend the following references: [Q04, GGKK03] for message passing, [Ja92] and [KR90] for PRAMs, [GHR95] for Chapter 12 and [AW04, L96, T00] for distributed algorithms.

**Exercise 1**

We call a problem **parallelizable**, if it can be solved by a parallel algorithm with \(\text{poly}(n)\) processors in time \(\text{poly}(\log_2 n)\). A problem is **efficiently parallelizable**, if additionally the product of running time and the number of processors is by at most a factor of \(\text{poly}(\log_2 n)\) larger than the running time of the best sequential algorithm. Which of the following problems “seem” parallelizable, respectively efficiently parallelizable?

1. **Input:** \(n\) numbers \(z_1, \ldots, z_n\) with \(n\) bits each.  
   **Output:** The binary representation of the sum \(z_1 + \cdots + z_n\).

2. **Input:** Two numbers \(x\) and \(y\) with \(n\) bits each.  
   **Output:** The binary representation of \(\lfloor x/y \rfloor\).

3. **Input:** Two \(n \times n\) matrices \(A\) and \(B\) with entries from \(\{0, 1\}\).  
   **Output:** The product matrix \((c_{i,j})_{i,j}\) with \(c_{i,j} = \bigvee_{k=1}^{n} a_{i,k} \land b_{k,j}\).

4. **Input:** The \(n \times n\) matrix \(A\) whose entries are \(n\)-bit integers.  
   **Output:** The determinant of \(A\).

5. **Input:** A directed graph \(G\) and two distinguished nodes \(s\) and \(t\).  
   **Output:** Accept, if there is a path in \(G\) from \(s\) to \(t\).

6. **Input:** A directed graph \(G\) with weighted edges and two distinguished nodes \(s\) and \(t\).  
   **Output:** The length of a shortest path from \(s\) to \(t\).

7. **Input:** A directed graph \(G\).  
   **Output:** The ordering of nodes in which depth-first search visits the nodes.

8. **Input:** An undirected graph \(G\).  
   **Output:** A maximal independent set, i.e., an independent set which cannot be enlarged.
1.1 Asymptotic Notation

Some of the results are asymptotic, and we use the standard asymptotic notation: for two functions \( f, g : \mathbb{N} \to \mathbb{R} \), we write \( f = O(g) \) if \( f(n) \leq cg(n) \) for all \( n \geq N \), where \( N \) is a suitably large threshold and \( c > 0 \) is a suitably large constant. We write \( f = \Omega(g) \) if \( g = O(f) \), and \( f = \Theta(g) \) if \( f = O(g) \) and \( g = O(f) \). If the limit of the ratio \( f/g \) tends to 0 as the variables of the functions tend to infinity, we write \( f = o(g) \). Finally, \( f \sim g \) denotes that \( f = (1 + o(1))g \), i.e., that \( f/g \) tends to 1 when the variables tend to infinity. If \( x \) is a real number, then \( \lceil x \rceil \) denotes the smallest integer not less than \( x \), and \( \lfloor x \rfloor \) denotes the greatest integer not exceeding \( x \).

1.2 Important Inequalities

Recall that the inner product of two vectors \( x = (x_1, \ldots, x_n) \) and \( y = (y_1, \ldots, y_n) \) in \( \mathbb{R}^n \) is defined by:

\[
\langle x, y \rangle = x_1y_1 + \cdots + x_ny_n.
\]

The norm (or length) of a vector \( x = (x_1, \ldots, x_n) \) is the number

\[
\|x\| = \langle x, x \rangle^{1/2} = \left( \sum_{i=1}^{n} x_i^2 \right)^{1/2}.
\]

A real-valued function \( f(x) \) is convex (resp. concave) iff

\[
f(\lambda x + (1 - \lambda)y) \leq \lambda f(x) + (1 - \lambda)f(y) \quad \text{(resp. } f(\lambda x + (1 - \lambda)y) \geq \lambda f(x) + (1 - \lambda)f(y) \text{)}
\]

for any \( 0 \leq \lambda \leq 1 \). From a geometrical point of view, the convexity of \( f \) implies that if we draw a line \( l \) through points \( (x, f(x)) \) and \( (y, f(y)) \), then the graph of the curve \( f(z) \) must lie below that of \( l(z) \) for \( z \in [x, y] \). Thus, for a function \( f \) to be convex it is sufficient that its second derivative is nonnegative.

The following basic inequality, known as the Cauchy–Schwarz inequality, bounds the inner product of two vectors in terms of their norms.

**Lemma 1.1 (Cauchy–Schwarz Inequality)** If \( x \) and \( y \) are vectors in \( \mathbb{R}^n \), then

\[
|\langle x, y \rangle| \leq \|x\| \cdot \|y\|.
\]

That is, if \( x_1, \ldots, x_n \) and \( y_1, \ldots, y_n \) are real numbers then

\[
\left( \sum_{i=1}^{n} x_iy_i \right)^2 \leq \left( \sum_{i=1}^{n} x_i^2 \right) \left( \sum_{i=1}^{n} y_i^2 \right).
\]
1.2. IMPORTANT INEQUALITIES

Proof: We consider the inner product $\langle u, v \rangle$ of two vectors $u$ and $v$ of norm 1. The value of the inner product coincides with the cosine of the angle between $u$ and $v$ and is therefore at most one. Thus the claim follows, if we replace $u$ by $\frac{u}{\|u\|}$ and $v$ by $\frac{v}{\|v\|}$.

We also need an estimate of $1 - x$ in terms of the $e$-function.

**Lemma 1.2** For any $x > -1$:

$$e^{x/(1+x)} \leq 1 + x \leq e^x.$$  

Moreover, $1 + x \leq e^x$ holds for all $x \in \mathbb{R}$.

Proof: We first consider the case $x \geq 0$. The natural logarithm is a concave function and therefore the slope of the secant through $(1, 0)$ and $(1+x, \ln(1+x))$ is bounded from below by the slope of the tangent in $(1+x, \ln(1+x))$ and bounded from above by the slope of the tangent in $(1, 0)$. Thus we get the inequality

$$\frac{1}{1+x} \leq \frac{\ln(1+x) - \ln(1)}{(1+x) - 1} = \frac{\ln(1+x)}{x} \leq 1$$  

or equivalently

$$\frac{x}{1+x} \leq \ln(1+x) \leq x.$$  

Thus the first claim follows for $x \geq 0$ by exponentiation. For $x \in ]-1, 0]$ we only have to observe that this time the inequality

$$1 \leq \frac{\ln(1+x) - \ln(1)}{(1+x) - 1} \leq \frac{1}{1+x}$$  

holds: the argument is analogous to proof of inequality (1.1). We once again obtain inequality (1.2), and thus have shown the claim, if we multiply (1.1) by the negative number $x$.

But then we also obtain $1 + x \leq e^x$ for all real numbers, since $1 + x$ is nonpositive for $x \leq -1$ and $e^x$ is always positive. \qed

**Lemma 1.3 (Jensen’s Inequality)** If $0 \leq \lambda_i \leq 1$, $\sum_{i=1}^r \lambda_i = 1$ and $f$ is convex, then

$$f \left( \sum_{i=1}^r \lambda_ix_i \right) \leq \sum_{i=1}^r \lambda_if(x_i).$$

Proof: Easy induction on the number $r$ of summands. For $r = 2$ this is true, so assume the inequality holds for the number of summands up to $r$, and prove it for $r + 1$. For this
it is enough to replace the sum of the first two terms in $\lambda_1 x_1 + \lambda_2 x_2 + \ldots + \lambda_{r+1} x_{r+1}$ by the term
\[
(\lambda_1 + \lambda_2) \left( \frac{\lambda_1}{\lambda_1 + \lambda_2} x_1 + \frac{\lambda_2}{\lambda_1 + \lambda_2} x_2 \right)
\]
and apply the induction hypothesis.

Jensen’s inequality immediately yields the following useful inequality between the arithmetic and geometric means.

**Lemma 1.4** Let $a_1, \ldots, a_n$ be non-negative numbers. Then

\[
\frac{1}{n} \sum_{i=1}^{n} a_i \geq \left( \prod_{i=1}^{n} a_i \right)^{1/n}.
\]

**Proof:** Let $f(x) = 2^x$, $\lambda_1 = \ldots = \lambda_n = 1/n$ and $x_i = \log_2 a_i$, for all $i = 1, \ldots, n$. By Jensen’s inequality

\[
\frac{1}{n} \sum_{i=1}^{n} a_i = \sum_{i=1}^{n} \lambda_i f(x_i) \geq f\left( \sum_{i=1}^{n} \lambda_i x_i \right) = 2^{\left( \sum_{i=1}^{n} x_i \right)/n} = \left( \prod_{i=1}^{n} a_i \right)^{1/n}.
\]

\[\square\]

### 1.3 Graphs

An *undirected graph* is a pair $G = (V, E)$ consisting of a set $V$, whose members are called *nodes* (or *vertices*), and a family $E$ of 2-element subsets of $V$, whose members are called *edges*. A node $v$ is *incident* with an edge $e$ if $v \in e$. The two nodes incident with an edge are its *endpoints*, and the edge *joins* its endpoints. Two nodes $u, v$ of $G$ are *adjacent*, or *neighbors*, if $\{u, v\}$ is an edge of $G$. In a *directed* graph edges are not 2-element subsets $\{u, v\}$, but ordered pairs $(u, v)$ of nodes instead.

The number $d(u)$ of neighbors of a node $u$ is its *degree*. The degree of a graph is the maximal degree of its nodes.

A *walk* of length $k$ in $G$ is a sequence $v_0, e_1, v_1, \ldots, e_k, v_k$ of nodes and edges such that $e_i = \{v_{i-1}, v_i\}$. A walk without repeated nodes is a *path*. A *cycle* of length $k$ is a walk $v_0, \ldots, v_k$ such that $v_0 = v_k$ and $v_0, \ldots, v_{k-1}$ is a path. The *distance* between two nodes is the minimum length of a path between them.

A *Hamiltonian cycle* is a cycle containing all nodes of the graph. Note that a Hamiltonian cycle traverses every node exactly once. An *Euler cycle* is a walk which starts and ends at the same node and includes every edge exactly once.
1.3. GRAPHS

Figure 1.1: The first graph is a cycle of length 5. The second graph consists of two cycles of length 3.

A (connected) component in a graph is a set of its nodes such that there is a path between any two of them. A graph is connected if it consists of one component. A subgraph is obtained by deleting edges and nodes. A spanning subgraph is obtained by deleting edges only. An induced subgraph is obtained by deleting nodes (together with all the edges incident to them).

A forest is a graph without cycles. A tree is a connected graph without cycles.

In a rooted tree one node \( r \) is declared as root. Nodes of degree 1 are called leaves. The depth of a node \( u \) in such a tree is the distance between \( u \) and the root. Nodes of the same depth form a level of a tree. The depth of a tree is the maximum depth of its leaf. A tree is binary if the root has degree 2 and all other non-leaves have degree 3. A complete binary tree \( T_d \) of depth \( d \) is a binary tree such that all leaves have depth \( d \).

Figure 1.2: A forest consisting of three trees.

Figure 1.3: A rooted tree of depth 3; \( u \) and \( w \) are children of \( v \), and \( v \) is their parent; \( a \), \( b \) and \( v \) are on the same level.
A complete graph or clique is a graph in which every pair of nodes is adjacent. An independent set in a graph is a set of nodes with no edges between them. The greatest integer $r$ such that $G$ contains an independent set of size $r$ is the independence number of $G$, and is denoted by $\alpha(G)$. A graph is bipartite if its node set can be partitioned into two independent sets.

A legal coloring of $G = (V, E)$ is an assignment of colors to each node so that adjacent nodes receive different colors and hence it can be interpreted as a partition of the node set $V$ into independent sets. The minimum number of colors required for a legal coloring is the chromatic number $\chi(G)$ of $G$.

**Exercise 2**
(a) Show that every rooted binary tree with $N$ leaves has depth at least $\log_2 N$.
(b) Let $T$ be a rooted binary tree with $N$ leaves, and let $d_i$ be the depth of the $i$th leaf. Show that then the so called Kraft inequality holds: $\sum_{i=1}^{N} 2^{-d_i} \leq 1$. Hint: associate with each leaf a binary string.
(c) Show that always $\chi(G) \geq |V|/\alpha(G)$.

### 1.4 Some Basics from Probability Theory

A finite probability space consists of a finite set $\Omega$ (a sample space) and a function

$$\text{prob} : \Omega \to [0, 1]$$

(a probability distribution), such that $\sum_{x \in \Omega} \text{prob}[x] = 1$. A probability space is a representation of a random experiment, where we choose a member of $\Omega$ at random and $\text{prob}[x]$ is the probability that $x$ is chosen.

Elements $x \in \Omega$ are called sample points and subsets $A \subseteq \Omega$ are called events. The probability of an event is defined by

$$\text{prob}[A] = \sum_{x \in A} \text{prob}[x],$$

i.e., the probability that a member of $A$ is chosen.

Some elementary properties follow directly from the definitions. For any two events $A$ and $B$ (here and throughout $\bar{A} = \Omega \setminus A$ stands for the complement of $A$):

1. $\text{prob}[A \cup B] = \text{prob}[A] + \text{prob}[B] - \text{prob}[A \cap B]$;
2. $\text{prob}[\bar{A}] = 1 - \text{prob}[A]$;
3. $\text{prob}[A \cap B] \geq \text{prob}[A] - \text{prob}[\bar{B}]$;
4. If $B_1, \ldots, B_m$ is a partition of $\Omega$, then $\text{prob}[A] = \sum_{i=1}^{m} \text{prob}[A \cap B_i]$. 
For two events $A$ and $B$, the conditional probability of $A$ given $B$, denoted $\text{prob}[A|B]$, is the probability that $A$ occurs if one knows that $B$ has occurred. Formally, we define

$$\text{prob}[A|B] = \frac{\text{prob}[A \cap B]}{\text{prob}[B]},$$

when $\text{prob}[B] \neq 0$. For example, if we are choosing an integer uniformly from $\{1, \ldots, 6\}$, if $A$ is the event that the number is 2 and if $B$ is the event that the number is even, then $\text{prob}[A|B] = \frac{1}{3}$, whereas $\text{prob}[B|A] = 1$.

Two events $A$ and $B$ are independent $\text{prob}[A \cap B] = \text{prob}[A] \cdot \text{prob}[B]$. Note that this is equivalent to $\text{prob}[A|B] = \text{prob}[A]$ as well as to $\text{prob}[B|A] = \text{prob}[B]$. Hence, in the case of uniform distribution we have that the events $A$ and $B$ are independent iff the proportion $|A|/|\Omega|$ of the event $A$ in the whole sample space is equal to the proportion $|A \cap B|/|B|$ of the sub-event $A \cap B$ of $A$ in the event $B$.

It is worth to note that “independence” has nothing to do with “disjointness” of events: if, say, $\text{prob}[A] > 0$, $\text{prob}[B] > 0$ and $A \cap B = \emptyset$, then the events $A$ and $B$ are dependent!

Example 1.1 We describe the “Monty Hall problem”. In a game show a price is hidden behind one of three doors. A contestant guesses one of the three doors and the show master Monty Hall opens another door which does not hide the price. The contestant has the option to change her choice. What should she do? We consider the events

- $P_i$: the price is behind door $i$,
- $C_i$: the contestant’s first choice is door $i$,
- $M_i$: Monty Hall opens door $i$.

We assume without loss of generality that the contestant opens first door 1 and that Monty Hall opens door 2; moreover we assume that both choices are according to the uniform distribution. We determine the conditional probabilities $\text{prob}[P_1 \mid C_1, M_2]$ and
CHAPTER 1. INTRODUCTION

\[ \text{prob}[ P_3 \mid C_1, M_2 ] \text{ and observe that } \text{prob}[ P_1 \mid C_1, M_2 ] + \text{prob}[ P_3 \mid C_1, M_2 ] = 1 \text{ holds, since the price is not behind door 2. By definition of conditional probabilities we get} \]
\[ \text{prob}[ P_1 \mid C_1, M_2 ] \cdot \text{prob}[ C_1, M_2 ] = \text{prob}[ C_1, M_2 \mid P_1 ] \cdot \text{prob}[ P_1 ] \quad \text{and} \]
\[ \text{prob}[ P_3 \mid C_1, M_2 ] \cdot \text{prob}[ C_1, M_2 ] = \text{prob}[ C_1, M_2 \mid P_3 ] \cdot \text{prob}[ P_3 ]. \]

We determine the right hand sides and obtain
\[ \text{prob}[ P_1 \mid C_1, M_2 ] \cdot \text{prob}[ C_1, M_2 ] = \left( \frac{1}{3} \cdot \frac{1}{2} \right) \cdot \frac{1}{3}, \]
since Monty Hall can open doors 2 and 3. On the other hand we have
\[ \text{prob}[ P_3 \mid C_1, M_2 ] \cdot \text{prob}[ C_1, M_2 ] = \left( \frac{1}{3} \cdot 1 \right) \cdot \frac{1}{3}, \]
since Monty Hall can only open door 2. Thus we get
\[ \text{prob}[ P_3 \mid C_1, M_2 ] \cdot \text{prob}[ C_1, M_2 ] = 2 \cdot \text{prob}[ P_1 \mid C_1, M_2 ] \cdot \text{prob}[ C_1, M_2 ] \]
and obtain \( \text{prob}[ P_3 \mid C_1, M_2 ] = \frac{2}{3} \) and \( \text{prob}[ P_1 \mid C_1, M_2 ] = \frac{1}{3} \). The contestant should always change her choice!

A random variable is a function \( X : \Omega \rightarrow \mathbb{R} \) of the sample space. For example, if \( X \) is an integer uniformly chosen from \( \{1, \ldots, n\} \), then \( Y = 2X \) and \( Z = \text{"the number of prime divisors of } X \)" are both random variables, and so is \( X \) itself. The distribution of a random variable \( X \) is the function \( f : \Omega \rightarrow [0, 1] \) defined as \( f(i) = \text{prob}[X = i] \), where here and in what follows \( \text{prob}[X = i] \) denotes the probability of the event \( A = \{x \in \Omega : X(x) = i\} \).

An indicator random variable for the event \( A \) is a random 0-1 variable \( X \) with
\[ X(\omega) = \begin{cases} 
1 & \text{if } \omega \in A; \\
0 & \text{otherwise.}
\end{cases} \]

Example 1.2 The solution to the “Monty Hall problem” can be formulated concisely when using random variables. Let \( W \) be the random variable describing the first choice of the contestant and let \( T \) be the random variable describing the correct door. The Change-strategy finds the correct door, if \( W \) and \( T \) disagree and this event happens with probability \( \frac{2}{3} \).

Random variables \( X_1, \ldots, X_n \) are called independent, iff for all \( x_1, \ldots, x_n \in \mathbb{R} \)
\[ \text{prob}[ X_1 = x_1 \land \cdots \land X_n = x_n ] = \prod_{i=1}^{n} \text{prob}[ X_i = x_i ]. \]

The random variables are called \( k \)-wise independent, if \( X_{i_1}, \ldots, X_{i_k} \) is independent for any subset \( \{i_1, \ldots, i_k\} \subseteq \{1, \ldots, n\} \) of \( k \) elements.

Observe that repeated experiments correspond to independent random variables. Moreover independent random variables \( X_1, \ldots, X_n \) are \( k \)-wise independent for any \( k < n \). The reverse implication is false.
1.4. SOME BASICS FROM PROBABILITY THEORY

Exercise 3

Construct random variables $X_1, X_2, X_3$, which are pairwise independent, but not independent.

A fundamental probabilistic measure is the expected value of a random variable. The expectation is defined for any real-valued random variable $X$, and intuitively, it is the value that we would expect to obtain if we repeat a random experiment several times and determine the average of the outcomes of $X$. More formally, the expected value (or expectation or mean) of $X$ is defined as the sum

$$E[X] = \sum_i i \cdot \text{prob}[X = i]$$

over all numbers $i$ in the range of $X$. For example, if $X$ is the number of spots on the top face when we roll a fair die, then the expected number of spots is $E[X] = \sum_{i=1}^{6} \frac{i}{6} = 3.5$.

Figure 1.5: Interpretation of the expected value as the “mass-center”: the objects have weights $p_i = \text{prob}[X = a_i]$ on the $x$-axis in positions $a_i$ ($i = 1, \ldots, n$).

It is important to note that the expected value $E[X]$ does not have to lie in the range of $X$. For example, if the random variable $X$ assumes each value in the set $\{0, 1, 9, 10\}$ with probability $1/4$ then $E[X] = 0 \cdot \frac{1}{4} + 1 \cdot \frac{1}{4} + 9 \cdot \frac{1}{4} + 10 \cdot \frac{1}{4} = 5$, the number which lies far away from the possible values $0, 1, 9, 10$ of $X$. Do not always expect the “expected”! And indeed, in many cases the expected value $E[X]$ says not much about the behaviour of the random variable $X$. The expectation is only “reliable”, if one can show that the probability of $X$ being far away from $E[X]$ is small. The estimates of these probabilities are given by Markov, Chernoff and Chebyshev inequalities which we will prove soon.

The “distance” between a random variable $X$ and its expectation $E[X]$ is measured by its variance which is defined as

$$\text{Var}[X] = E[(X - E[X])^2].$$

A very useful property of the expectation is its linearity. The property is so useful because it applies to any random variables.

Theorem 1.5 (Linearity of Expectation) For any random variables $X, Y$ and real numbers $a, b$

$$E[a \cdot X + b \cdot Y] = a \cdot E[X] + b \cdot E[Y].$$
**Proof:** Let \( x_1, \ldots, x_n \) and \( y_1, \ldots, y_m \) be the elements in the range of \( X \) and \( Y \), respectively. Since for any real number \( a \) we have

\[
E[a \cdot X] = \sum_{i=1}^{n} a \cdot x_i \cdot \text{prob}[X = x_i] = a \cdot \sum_{i=1}^{n} x_i \cdot \text{prob}[X = x_i] = a \cdot E[X],
\]

it is enough to prove that \( E[X + Y] = E[X] + E[Y] \).

Since the events \( Y = y_j \) for different \( y_j \)’s are disjoint, we have that \( \text{prob}[X = x_i] = \sum_{j=1}^{m} \text{prob}[X = x_i, Y = y_j] \), and similarly for \( \text{prob}[Y = y_j] \). Hence

\[
E[X + Y] = \sum_{i=1}^{n} \sum_{j=1}^{m} (x_i + y_j) \text{prob}[X = x_i, Y = y_j] = \sum_{i=1}^{n} x_i \text{prob}[X = x_i, Y = y_j] + \sum_{j=1}^{m} y_j \text{prob}[X = x_i, Y = y_j] = \sum_{i=1}^{n} x_i \text{prob}[X = x_i] + \sum_{j=1}^{m} y_j \text{prob}[Y = y_j] = E[X] + E[Y].
\]

\( \Box \)

**Exercise 4**

(a) Show that the variance can be computed by the formula \( \text{Var}[X] = E[X^2] - (E[X])^2 \).

(b) Show that we always have \( \text{Var}[a \cdot X + b] = a^2 \cdot \text{Var}[X] \).

(c) Let \( X \) and \( Y \) be independent random variables. Show that then \( E[X \cdot Y] = E[X] \cdot E[Y] \) and \( \text{Var}[X + Y] = \text{Var}[X] + \text{Var}[Y] \).

(d) Let \( X \) be a non-negative integer-valued random variable. Show that \( E[X^2] \geq E[X] \), \( \text{prob}[X = 0] \geq 1 - E[X] \) and \( E[X] = \sum_{x=1}^{\infty} \text{prob}[X \geq x] \).

(e) Use the Cauchy–Schwarz inequality to show that, for any random variable \( X \), \( E[X]^2 \leq E[X^2] \).

(f) We have \( n \) letters going to \( n \) different persons and \( n \) envelopes with their addresses. We insert each letter into an envelope independently from each other at random (several letters may go in the same envelope). What is the expected number of correct matches?

(g) Recall that the length (or the norm) \( \|v\| \) of a vector \( v \in \mathbb{R}^n \) is the square root of the scalar product \( \langle v, v \rangle \). Prove that for any vectors \( v_1, \ldots, v_n \) in \( \{+1, -1\}^n \) there are scalars \( \epsilon_1, \ldots, \epsilon_n \in \{+1, -1\} \) such that

\[
\|\epsilon_1 v_1 + \cdots + \epsilon_n v_n\| \leq n.
\]

*Hint:* Choose the \( \epsilon_i \)’s independently at random to be \( +1 \) or \( -1 \) with probability \( 1/2 \), and use the linearity of expectation to evaluate the expected length of the vector \( \sum \epsilon_i v_i \) by computing the square of that quantity.
The method of generating functions There is a general method to compute the expectation $E[X]$ and the variance $\text{Var}[X]$ for an arbitrary discrete random variable $X : \Omega \rightarrow \mathbb{N}$. Set $p_k = \text{prob}[X = k]$ and define the generating function of $X$ by

$$F_X(x) := \sum p_k x^k,$$

where the sum is over all $k$’s in the range of $X$.

**Theorem 1.6** For every discrete random variable $X$ we have:

1. $F_X(1) = 1$ (1.4)
2. $E[X] = F_X'(1)$ (1.5)
3. $\text{Var}[X] = F_X''(1) + E[X] - E[X]^2$ (1.6)

**Proof:** The first equality is just the definition of the probability distribution: $F_X(1) = \sum p_k = 1$. Taking the first derivative $F_X'(x) = \sum k p_k x^{k-1}$ we obtain that $F_X'(1) = \sum k p_k = E[X]$. Finally, taking the second derivative $F_X''(x) = \sum k(k-1)p_k x^{k-2}$ we obtain that

$$F_X''(1) = \sum k(k-1)p_k = \sum k^2 p_k - \sum kp_k = E[X^2] - E[X].$$

Thus, if we add $E[X]$ and subtract $E[X]^2$ on the right hand, then we obtain precisely the variance $\text{Var}[X]$. \square

We mention some often used distributions of random variables.

**Uniform distribution** This is the simplest distribution for a random variable $X : \Omega \rightarrow \mathbb{R}$ and it is given by

$$p_k = \text{prob}[X = k] = \frac{1}{|\Omega|}.$$

In this case we also say that $X$ is *uniformly distributed*.

**Exercise 5**

What is are expectation and the variance of a uniformly distributed random variable?

**Binomial distribution** We say that a random variable $X$ is *binomially distributed* with parameters $n$ and $k$, provided

$$\text{prob}[X = k] = \binom{n}{k} p^k (1 - p)^{n-k}$$

for any $0 \leq k \leq n$. If $X_1, \ldots, X_n$ are $n$ independent random variables with $\text{prob}[X_i = 1] = p$ and $\text{prob}[X_i = 0] = 1 - p$, then $\text{prob}[X_1 + \cdots + X_n = k] = \binom{n}{k} p^k (1 - p)^{n-k}$. Hence we may set $X = X_1 + \cdots + X_n$ and $\text{prob}[X = k]$ is the probability that $k$ out of $n$ repeated
experiments succeed. The expectation of \( X \) is 
\[
E[X] = \sum_{i=1}^{n} E[X_i] = np.
\]
The variance is also easy to compute. By the linearity of expectation
\[
E[X^2] = E\left(\sum X_i\right)^2 = \sum_i E[X_i^2] + \sum_{i \neq j} E[X_i \cdot X_j].
\]
Since \( X_i, X_j \) are pairwise independent, \( E[X_i \cdot X_j] = E[X_i] \cdot E[X_j] \). Hence
\[
E[X^2] = np + 2 \cdot \left(\frac{n}{2}\right) \cdot p^2 = np + n(n-1)p^2 = (np)^2 + np(1-p),
\]
and the variance of \( X \) is
\[
\text{Var}[X] = E[X^2] - (E[X])^2 = np(1-p).
\]

**Geometric distribution** We toss a coin until we receive heads for the first time. Let \( X \) be the number of attempts. Hence, \( X \) is defined on the set \( \Omega = \{\text{Tail}^n \cdot \text{Head} \mid n \in \mathbb{N}\} \). (This time we have countably infinite sample points.) If \( p \) is the probability for Heads, then we get
\[
p_k = \text{prob}[X = k] = q^{k-1} \cdot p, \quad \text{where } q = 1-p.
\]
To compute the expectation and the variance for such a random variable we use the method of generating functions (see Theorem 1.6). The generating function of \( X \) is
\[
F_X(x) = \sum_{t=1}^{\infty} q^{t-1} p x^t = px \cdot \sum_{t=0}^{\infty} q^t x^t = \frac{px}{1-qx}
\]
The first and the second derivatives of \( F_X \) are
\[
F_X'(x) = \frac{(1-qx)p + pxq}{(1-qx)^2} = \frac{p}{(1-qx)^2}
\]
and
\[
F_X''(x) = \frac{2pq}{(1-qx)^3}
\]
Hence, by Theorem 1.6,
\[
E[X] = F_X'(1) = \frac{p}{(1-q)^2} = \frac{1}{p}
\]
and
\[
\text{Var}[X] = F_X''(1) + E[T] - E[Y]^2 = \frac{2pq}{p^2} + \frac{1}{p} - \frac{1}{p^2} = \frac{1-p}{p^2}.
\]

**Exercise 6**
A random source \( Q \) produces zeroes and ones, such that the probability for a one is \( p \) at any time. We have access to \( Q \), but do not know \( p \). Our task is to construct a perfect random source which emits a one with probability \( \frac{1}{2} \). Hence we can watch the stream of random bits of \( Q \) and have to construct perfect random bits.

Design a perfect random source from \( Q \), such that the expected number of \( Q \)-bits, required to produce one truly random bit, is at most \( \frac{1}{p(1-p)} \).

**Hint:** consider two successive bits of \( Q \).
Exercise 7
We have access to a random source $S$ which emits the bit 0 (resp. 1) with probability $\frac{1}{2}$.
- We want to produce the distribution $(p_1, \ldots, p_n)$. Design an algorithm which outputs $i$ with probability $p_i$. You should only use the expected number of $O(\log_2 n)$ bits from $S$. Observe that $p_1, \ldots, p_n \geq 0$ are arbitrary real numbers summing to one.
- Determine the worst-case number of bits from $S$, if we have to solve the above problem for $p_1 = p_2 = p_3 = \frac{1}{3}$.

Exercise 8
We are playing against an adversary. The adversary thinks of two numbers and writes each number, invisible to us, on a separate piece of paper. We randomly pick one piece of paper, read the number and then have the option, to either keep the number or exchange it against the other number. Let $x$ be the number we finally have and let $y$ be the other number. Then our possibly negative gain is the difference $x - y$.
- we consider strategies of the form “Return numbers $< t$ and keep numbers $\geq t$”: Analyze the expected gain $E_{x,y}(\text{gain}(S_t))$ of this strategy in dependence on $t, x$ and $y$.
- Design a randomized strategy, whose expected gain is positive for arbitrary $x \neq y$.

Exercise 9
We play a game with success probability $p = \frac{1}{2}$. If we win the game, we double our stake. If we lose, we lose our stake completely. We consider the following strategy.

\begin{verbatim}
i:=0
REPEAT
  put down the stake $2^i$
  i:=i+1
UNTIL(I win for the first time)
\end{verbatim}

Determine the expected gain of this strategy and the expected amount of money required to carry out the strategy.

Example 1.3 Determining the average salary without making the own salary public.
$n$ persons $1, \ldots, n$ want to determine their average salary, but no one wants to make their salary public. If all persons are honest, then the average salary should be computed correctly. If however some $k$ persons are liars, then they should not be able to gather any information, except for the total sum of salaries.

Algorithm 1.7 A protocol

(1) Let $M$ be sufficiently large number which is known to be larger than the sum of salaries.

Each person $i$ chooses numbers $X_{i,1}, \ldots, X_{i,i-1}, X_{i,i+1}, \ldots, X_{i,n} \in \{0, \ldots, M - 1\}$ at random and communicates $X_{i,j}$ to person $j$. 
(2) If $G_i$ is the salary of person $i$, then she determines the residue

$$S_i = G_i + \sum_{j, j \neq i}^n X_{j,i} - \sum_{j, j \neq i}^n X_{i,j} \mod M.$$ 

(3) Each person $i$ announces $S_i$ and then \( \sum_i S_i \mod M \) is determined.

Commentary: If all persons are honest, then we have \( \sum_i S_i \mod M \equiv \sum_i G_i + \sum_{j=1, j \neq i}^{n-k} X_{j,i} - \sum_{j=1, j \neq i}^{n-k} X_{i,j} \mod M \equiv \sum_i G_i \mod M = \sum_i G_i. \)

Thus \( \frac{1}{n} \cdot \sum_j S_j \) is the wanted average salary.

Why is this protocol safe? Suppose the last $k$ persons are liars. We set $S^*_i = G_i + \sum_{j=1, j \neq i}^{n-k} X_{j,i} - \sum_{j=1, j \neq i}^{n-k} X_{i,j} \mod M$. An honest person makes

$$S_i = S^*_i + \sum_{j=n-k+1}^n X_{j,i} - \sum_{j=n-k+1}^n X_{i,j} \mod M$$

public. Observe that \( \sum_{i=1}^{n-k} S^*_i = \sum_{i=1}^{n-k} G_i \) holds.

**Exercise 10**

Show: Any combination \((s^*_2, \ldots, s^*_{n-k})\) of values of the random variables $S^*_2, \ldots, S^*_{n-k}$ is produced with probability $M^{-(n-k-1)}$. Thus the random variables $S^*_2, \ldots, S^*_{n-k}$ are independent.

According to the problem, the random variables $S^*_2, \ldots, S^*_{n-k}$ are independent and uniformly distributed over \( \{0, \ldots, M - 1\} \). With the same argument this statement also holds for $S^*_2, \ldots, S^*_{n-k}$ and the liars learn nothing, since each sequence of $n - k - 1$ values occurs with probability \( \frac{1}{M^{n-k-1}}. \)

We investigate the probability of large deviations from the expected value. The following estimates are particularly helpful.

**Markov’s Inequality** Let $X$ be a non-negative random variable and $a > 0$ a real number. Then

$$\text{prob} \left[ X > a \right] \leq \frac{E[X]}{a}.$$ 

**Proof:** By the definition of the expected value we have

$$E[X] = \sum_x x \cdot \text{prob} [X = x] \geq \sum_{x > a} a \cdot \text{prob} [X = x] = a \cdot \text{prob} [X > a].$$

□
Chebycheff’s Inequality  Let $X$ be a random variable. Then for any real $t > 0$,

$$\text{prob} [ |X - E[X]| > t ] \leq \frac{\text{Var}[X]}{t^2}.$$  

Proof: We define the random variable $Y = (X - E[X])^2$, and apply Markov’s inequality. This yields

$$\text{prob} [ |X - E[X]| > t ] = \text{prob} [ Y > t^2 ] \leq \frac{\text{E}[Y]}{t^2} = \frac{\text{Var}[X]}{t^2}.$$

Chernoff’s Inequalities  These inequalities are special (and hence, more informative) cases of Markov’s inequality applied to sums of independent random 0-1 variables.

Theorem 1.8  Let $X_1, \ldots, X_n$ be arbitrary independent binary random variables, where $p_i = \text{prob} [X_i = 1]$ is the success probability of $X_i$. Then $N = \sum_{i=1}^{n} p_i$ is the expected number of successes and we get

$$\text{prob} \left[ \sum_{i=1}^{n} X_i > (1 + \beta) \cdot N \right] \leq \left( \frac{e^{\beta}}{(1 + \beta)\beta} \right)^{N} \leq e^{-N \cdot \beta^2 / 3}$$

$$\text{prob} \left[ \sum_{i=1}^{n} X_i < (1 - \beta) \cdot N \right] \leq \left( \frac{e^{-\beta}}{(1 - \beta)\beta} \right)^{N} \leq e^{-N \cdot \beta^2 / 2}$$

for any $\beta > 0$ (resp. $0 < \beta \leq 1$ in the second case).

Proof: We only show the first inequality. We obtain with the Markov inequality for an arbitrary $\alpha > 0$

$$\text{prob} \left[ \sum_{i=1}^{n} X_i > t \right] = \text{prob} \left[ e^{\alpha \sum_{i=1}^{n} X_i} > e^{\alpha t} \right] \leq e^{-\alpha t} \cdot \text{E} \left[ e^{\alpha \sum_{i=1}^{n} X_i} \right] = e^{-\alpha t} \cdot \prod_{i=1}^{n} \text{E} \left[ e^{\alpha X_i} \right].$$

In the last equality we have utilized that $\text{E} [Y_1 \cdot Y_2] = \text{E} [Y_1] \cdot \text{E} [Y_2]$ holds, if $Y_1$ and $Y_2$ are independent random variables. We replace $t$ by $(1 + \beta) \cdot N$, $\alpha$ by $\ln(1 + \beta)$ and obtain

$$\text{prob} \left[ \sum_{i=1}^{n} X_i > (1 + \beta) \cdot N \right] \leq e^{-\alpha (1 + \beta) \cdot N} \cdot \prod_{i=1}^{n} \text{E} \left[ e^{\alpha X_i} \right] = (1 + \beta)^{-(1 + \beta) \cdot N} \cdot \prod_{i=1}^{n} \text{E} \left[ (1 + \beta)^{X_i} \right].$$
The claim follows, since $E[(1 + \beta)^X_i] = p_i(1 + \beta) + (1 - p_i) = 1 + \beta \cdot p_i \leq e^{\beta p_i}$.

**Exercise 11**
Let $x \leq 1$ be arbitrary. Show

$$\frac{e^x}{(1 + x)^{1+x}} = e^{x - (1 + x) \ln(1 + x)} \leq e^{-x^2/3}.$$

**Exercise 12**
Let $X_1, \ldots, X_n$ be binary, independent random variables with $p_i = \text{prob}[X_i = 1]$. Assume that $N^* \geq \sum_{i=1}^n p_i$. Show that

$$\text{prob}\left[\sum_{i=1}^n X_i > (1 + \beta) \cdot N^*\right] \leq \left(\frac{e^\beta}{(1 + \beta)^{1+\beta}}\right)^{N^*} \leq e^{-\beta \cdot N^*/3}$$
holds for any $\beta > 0$.

**Exercise 13**
Show the second Chernoff inequality.

**Example 1.4** We would like to obtain the expected value of an experiment $X$. The experiment however is quite unstable, i.e., its variance is quite large. We boost $X$, that is we repeat the experiment $k$ times. If $X_i$ is the result of the $i$th repetition, then we set $Y = \frac{1}{k} \cdot \sum_{i=1}^k X_i$ and observe that the random variables $X_1, \ldots, X_k$ are independent and in particular

$$\text{Var}[Y] = \frac{1}{k^2} \cdot \sum_{i=1}^k \text{Var}[X_i] = \frac{1}{k^2} \cdot \sum_{i=1}^k \text{Var}[X] = \frac{1}{k} \cdot \text{Var}[X]$$
holds. We have reduced variance by the factor $k$ and the expected value remains stable, since $E[Y] = E\left[\frac{1}{k} \cdot \sum_{i=1}^k X_i\right] = \frac{1}{k} \cdot \sum_{i=1}^k E[X] = E[X]$. The Chebycheff inequality implies the inequality

$$\text{prob} [|Y - E[X]| > t] = \text{prob} [|Y - E[Y]| > t] \leq \frac{\text{Var}[Y]}{t^2} = \frac{\text{Var}[X]}{k \cdot t^2}$$
and large deviations from the expected value have become more improbable.

Assume we know that $Y$ lies, with probability at least $p = 1 - \varepsilon$, within the “confidence interval” $T = [E[X] - \delta, E[X] + \delta]$. Is it possible to drive $p$ fast towards one? This time we repeat the experiment $Y$ $m$ times and obtain again independent random variables $Y_1, \ldots, Y_m$. We use the median $M$ of $Y_1, \ldots, Y_m$ to obtain an estimate of the expected value. How large is the probability, that $M$ does not belong to the confidence interval $T$?

$Y$ belongs to $T$ with probability at least $p$. If the median lies outside of $T$, then at least $m \cdot 2 - m = \varepsilon \cdot m$ $Y_i$s are expected to
lie outside of $T$. We apply the Chernoff inequality and observe that $(1 + \frac{1-2\varepsilon}{2\varepsilon}) \cdot \varepsilon \cdot m = \frac{m}{2}$ holds. We set $\lambda = \frac{1-2\varepsilon}{2\varepsilon}$ and get
\[
\text{prob } [M \notin T] \leq e^{-\varepsilon \cdot m \cdot \lambda^2 / 3} = e^{-(1-2\varepsilon)^2 m / (12\varepsilon)}.
\]
Thus the probability of error decreases negative exponentially, provided $\varepsilon < \frac{1}{2}$. 
Part I

Distributed Memory
Chapter 2

Communication Patterns

How should we distribute information efficiently? Of course the answer depends on the algorithmic context, but a few basic “communication patterns” such as trees, meshes and hypercube architectures are distinguished.

We model a communication pattern by an undirected graph of processes where we introduce an edge between processes $p$ and $q$, whenever $p$ and $q$ communicate. How do we determine whether a given undirected graph $G = (V, E)$ is a “good” communication pattern? Important parameters are its degree, diameter and bisection width.

- The degree of $G$ is the maximal degree of a node.

- The diameter of $G$ is the maximal distance between any two nodes. Whenever possible we should keep the diameter as small as possible, since the time to route information between distant nodes is governed by the diameter.

   However the diameter is not the whole story behind a fast computation. Important is also the “bisection width” of the network.

- The bisection width of $G$ is the minimal number of edges, whose removal disconnects the graph into two halves with sizes differing by at most one.

   Observe that bisection width $b$ implies that $b$ edges have to carry all information from one half to the other and hence a large bisection width is of utmost importance to avoid bottlenecks. Moreover a large bisection width also provides robustness against link- and process-failures.

   However a large bisection width comes with the price of complex communications.

We discuss trees, meshes and hypercube architectures in the following sections and evaluate these most successful communication patterns with respect to degree, diameter and bisection width. Certainly a further fundamental component when evaluating architectures is to determine how well an architecture is suited to implement important algorithmic design principles such as for instance the Divide & Conquer method. Hypercube architectures
turn out to be particularly well suited for this task. Finally we should also mention that hypercube architectures are good choices to implement interconnection networks. We return to this subject in Section 2.3.3, where we describe randomized routing on the hypercube.

**Exercise 14**
A permutation network is a directed acyclic graph with \( n \) sources \( s_1, \ldots, s_n \) and \( n \) sinks \( t_1, \ldots, t_n \). We call its edges *links* and its nodes *switches*. Each switch has two inputs \( x_0 \) and \( x_1 \), two outputs \( y_0 \) and \( y_1 \) and two states *neutral* and *exchange*. In the exchange state the switch exchanges \( x_0 \) and \( x_1 \) (i.e., \( y_i = x_{1-i} \)), whereas inputs are left unchanged in the neutral state (i.e., \( y_i = x_i \)). We demand that the network can implement any permutation \( \pi \) of \( \{1, \ldots, n\} \) by properly setting switches such that \( s_i \) reaches \( t_{\pi(i)} \).

![Figure 2.1: The networks \( A_n \) and \( B_n \)](image)

(a) Prove or disprove: \( A_n \) is a family of permutation networks.
(b) Prove or disprove: \( B_n \) is a family of permutation networks.
(c) Show that \( A_n \) and \( B_n \) are composed of \( O(n \cdot \log_2 n) \) switches.
(d) Show that any permutation network with \( n \) sources and sinks requires \( \Omega(n \cdot \log_2 n) \) switches.

### 2.1 Trees

The broadcast problem is possibly the most fundamental problem with trees as communication pattern.

**Example 2.1** In broadcasting a process \( P \) sends a message to the remaining \( p-1 \) processes. Let \( T \) be a balanced binary tree with root \( P \) and depth at most \( \lfloor \log_2 p \rfloor \). The broadcast is started by root \( P \) who sends its message to its two children. Then the message is passed from one generation to the next and reaches all processes in \( \lfloor \log_2 p \rfloor \) steps.

**Exercise 15**
Let \( p \) be a power of two. Assume that a node has to broadcast a message to its \( p-1 \) colleagues. Is it a good idea to organize the broadcast by a binary tree or are more general tree shapes better?
Let $T_k$ be an ordered, complete binary tree of depth $k$. Complete binary trees have the big advantage of the small diameter $2^k$ allowing algorithms to use at most $O(k)$ communication steps. The big disadvantage is the bottleneck structure of trees: if nodes outside of the subtree $T_k(v)$ with root $v$ require information held by nodes of $T_k(v)$, then this information has to travel across bottleneck $v$.

**Example 2.2** A further prominent example of a tree algorithm is the tournament approach of determining the minimum among $n$ keys. We assign the keys to the leaves of $T_k$ such that each leaf obtains $\frac{n}{2^k}$ keys. First we let all leaves determine their minimum in parallel. Then the respective minimal keys are passed upwards from the children to their parents, who determine the minimum and pass it upwards themselves. We set $p = 2^{k+1} - 1$ and we have computed the minimum of $n$ keys with $p$ processors in $O(\log_2 p + \frac{n}{p})$ compute steps and $\log_2 p$ communication steps. Is it possible to obtain the same performance with just $p = 2^k$ processors?

We considerably generalize the class of problems solvable on binary trees by considering the important class of prefix problems.

**Definition 2.1** A set $U$ and an operation $*: U \times U \rightarrow U$ is given. We demand that operation $*$ is associative, i.e., we require that

$$x * (y * z) = (x * y) * z$$

holds for all $x, y, z \in U$. The prefix problem for the input sequence $x_1, x_2, \ldots, x_n \in U$ is to determine all prefixes

$$x_1, x_1 * x_2, x_1 * x_2 * x_3, \ldots, x_1 * x_2 * \cdots * x_n$$

The prefix problem has many applications. For instance

- An array $A$ of $m$ cells is given and we assume that a cell is either empty or stores a value. We would like to remove all empty cells. We set $U = \mathbb{N}$ and

$$x_i := \begin{cases} 1 & A[i] \text{ is non-empty,} \\ 0 & \text{otherwise.} \end{cases}$$

Finally we choose addition as our operation $*$. If cell $i$ is not empty, then $A[i]$ has to be moved to cell $x_1 + x_2 + \cdots + x_i$. 

- The problem of determining the minimum is a prefix problem, if we define operation \( \ast \) by
  \[ x \ast y = \min\{x, y\}. \]
  Observe that \( x_1 \cdots \ast x_n \) is the minimum of \( x_1, \ldots, x_n \).
- The reals with addition or multiplication define a prefix problem.
- Let \( U \) be a class of functions which is closed under composition, i.e., if \( f, g \in U \), then \( f \circ g \in U \). We choose the operation \( \ast \) as the composition of functions and observe that \( \ast \) is associative.
- A Mealey machine is a deterministic finite automaton, which produces an output whenever reading a letter. Let \( M \) be a Mealey machine with state set \( Q \), program \( \delta \) and input alphabet \( \Sigma \). Observe that \( M \) defines a function
  \[ f_a : Q \rightarrow Q \quad \text{with} \quad f_a(q) := \delta(q, a) \]
for each letter \( a \in \Sigma \). Therefore, with initial state \( q_0 \) and input word \( w = w_1w_2\ldots w_n \), we obtain the sequence of states

\[
\begin{align*}
q_0, \\
f_{w_1} \circ f_{w_2} \circ \cdots \circ f_{w_n} (q_0)
\end{align*}
\]

Thus we have reduced the problem of determining the sequence of states to the prefix problem with \( U = \{ f \mid f : Q \rightarrow Q \} \) and \( \ast \) as the composition of functions, provided we work on the sequence \( f_{w_1}, \ldots, f_{w_n} \) instead of on input \( w \). Observe however that in this case the evaluation of \( \ast \) takes time proportional to \( |Q| \).
- In particular, we have reduced the problem of adding two \( n \)-bit numbers \( x_1x_2\ldots x_n \) and \( y_1y_2\ldots y_n \) to a prefix problem, since the Mealey machine

A Mealey machine for addition.
performs addition, if we choose \( \Sigma = \{0, 1\}^2 \) as input alphabet. \( q_0 \) is the starting state and we move to \( q_1 \), whenever a carry has to be remembered. The last letter has to be 00. Retrieving the output bits is now easy, since the \( i \)th output bit is determined by \( x_{n-i+1}y_{n-i+1} \) and the \( i \)th state \( q_{i-1} \).

Exercise 16
We would like to solve a bidiagonal system \( A \cdot x = b \) of linear equations, that is all entries of the \( n \times n \) matrix \( A \) are zero with the exception of entries of the form \((i, i-1)\) or \((i, i)\). We also require that all entries of the main diagonal are non-zero since otherwise the system is not be solvable. Define a prefix problem whose solution determines \( x \).

Exercise 17
We recursively define the language of a well-formed brackets:

- \( () \) is well formed,
- if \( K_1 \) and \( K_2 \) are well-formed, then so is \((K_1)\) and \(K_1K_2\).

Determine how to decide in parallel whether a given expression of brackets is well-formed.

Exercise 18
In the segmented prefix problem, besides an array \( A \) and an associative operation \( \ast \), an additional 0-1 array \( B \) is given. The array \( B \) partitions \( A \) into segments, where a segment starts in position \( i \) whenever \( B[i] = 1 \). The task is to determine all prefix sums, but restricted to the segments defined by \( B \).

For instance, if we choose the operation addition:

\[
\begin{array}{cccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
\\
A & 4 & 2 & 6 & -2 & 6 & 7 & 0 & -3 \\
B & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\
1 & 4 & 6 & 12 & -2 & 4 & 7 & 7 & 4 \\
\end{array}
\]

Define a new operation \( \ast_{B,\ast} \), which depends on \( B \) and \( \ast \) only, such that \( \ast_{B,\ast} \) determines the “segment sums” as prescribed by \( B \).

We describe a fast evaluation of the prefix problem, assuming that the operation \( \ast \) can be quickly evaluated. Our communication pattern \( T^k_{\ast} \) is an extension of the complete binary \( T_k \): we introduce diagonal edges linking a left child to its leftmost nephew.
We first make the unrealistic assumption that we have \( p = 2n - 1 \) processors (with \( n = 2^k \)) and assign input \( x_i \) to the \( i \)th leaf. The computation proceeds in phases. In Phase 1 the input is moved up the tree and each interior node, upon receiving \( x \) and \( y \) from its children, computes \( z = x \cdot y \), stores the result \( z \) and hands it over to its parent.

In the second phase the computation is moving down again.

- The root broadcasts a “pay attention” signal down the tree. Moreover the result of the root is passed down to its right child.

- Assume that \( v \) receives the pay attention signal from the root and that it currently stores the result \( z_v \). \( v \) computes the new value \( z'_v \).

**Case 1:** If \( v \) is a left child, which does “sit” on the leftmost path, then it sets \( z'_v = z_v \) and sends \( z'_v \) to its nephew and its right child.

**Case 2:** If \( v \) is a left child, which does not “sit” on the leftmost path, then it reads the data \( x \) it receives from its uncle and computes \( z'_v = x \cdot z_v \). It sends \( z'_v \) to its nephew and its right child as well as \( x \) to its left child.

**Case 3:** If \( v \) is a right child, then it reads the data \( x \) it receives from its parent, sets \( z'_v = x \) and sends \( z'_v \) to its right child.
Observe that we have to provide a priori information to nodes. In particular, the root should know its prominent position, leaves should know that they have inputs and each inner node should be able to differentiate between its children, nephews and parent.

Exercise 19
(a) Show how to solve the prefix problem on the complete binary tree $T_k$.
(b) Show how to solve the prefix problem with $n = 2^k$ processors with no loss in running time.
(c) Assume that we only have $p = 2^k$ processors available with $p < n$. Each evaluation of $*$ may cost time $O(1)$. Show how to solve the prefix problem with $p$ processors in $O\left(\frac{n}{p} + \log_2 p\right)$ compute steps and in $O(\log_2 p)$ communication steps in which elements of $U$ are communicated.

Theorem 2.2 Assume that we can evaluate a $*$-product in time $\ell$. Then the prefix problem for $n$ elements can be solved with $p$ processors in $O(\ell \cdot \left(\frac{n}{p} + \log_2 p\right))$ compute steps and in $O(\log_2 p)$ communication steps involving elements from $U$. In particular, two $n$-bit numbers can be added with $p$ processors in $O\left(\frac{n}{p} + \log_2 p\right)$ compute steps.

Exercise 20
Assume that $p$ is a power of two. Show how to sort $n$ keys (with $p < n$) on $T_{\log_2 p}$ in time $O(n \cdot \frac{n}{p} \cdot \log_2 \frac{n}{p})$. We demand that each leaf stores at the beginning and at the end $\frac{n}{p}$ keys. Moreover we require that at most one key can be exchanged between neighbors. Is it possible to sort faster, if we can work on $T_s$ for arbitrarily large $s$ and sort by comparing keys?

2.2 Meshes

Definition 2.3 The $d$-dimensional mesh $M_d(m)$ is an undirected graph whose nodes are all $m^d$ possible vectors $x = (x_1, \ldots, x_d)$ with $x_i \in \mathbb{Z}_m := \{0, 1, \ldots, m - 1\}$. Two nodes $x$ and $y$ are connected by an edge iff $\sum_{i=1}^{d} |x_i - y_i| = 1$. Hence, $x$ and $y$ are connected iff there is a coordinate $i$ such that $y = (x_1, \ldots, x_{i-1}, x_i \pm 1, x_{i+1}, \ldots, x_d)$.
CHAPTER 2. COMMUNICATION PATTERNS

We also call $M_1(m)$ a one-dimensional array. Further important meshes are the two- and three-dimensional meshes $M_2(m)$ and $M_3(m)$ as well as the $d$-dimensional hypercube $M_d(2)$ which we investigate in the next section. Observe that $M_d(m)$ has degree at most $2d$ and diameter $d \cdot (m - 1)$.

To obtain an upper bound on the bisection width, we fix the first component per vector and remove all $m^{d-1}$ edges connecting the nodes with vectors $(m/2 - 1, x_2, \ldots, x_d)$ and $(m/2, x_2, \ldots, x_d)$. We have disconnected the node set into the two components $V_1 = \{i, x_2, \ldots, x_d) \mid 0 \leq i < \frac{m}{2}, 0 \leq x_2, \ldots, x_d < m\}$ and $V_2 = \{i, x_2, \ldots, x_d) \mid \frac{m}{2} \leq i < m, 0 \leq x_2, \ldots, x_d < m\}$. We show in the next section, that our upper bound is tight for the $d$-dimensional hypercube. The moderate bisection width however has its price: if a problem requires us to exchange information between $V_1$ and $V_2$, then this information has to be carried across the bottleneck of $m^{d-1}$ edges.

**Exercise 21**

A finite impulse response (FIR) filter of order $p$ receives a data stream $x_0, x_1, \ldots$ and outputs the data stream $y_0, y_1, \ldots$ with

$$y_t = \sum_{k=0}^{p-1} a_k \cdot x_{t-k}$$

for $t \geq p - 1$. Hence FIR produces the output stream

$$y_p = a_0 \cdot x_{p-1} + a_1 \cdot x_{p-2} + \cdots + a_{p-1} \cdot x_0$$
$$y_{p+1} = a_0 \cdot x_p + a_1 \cdot x_{p-1} + \cdots + a_{p-1} \cdot x_1$$
$$y_{p+2} = a_0 \cdot x_{p+1} + a_1 \cdot x_p + \cdots + a_{p-1} \cdot x_2$$

Show how to construct a FIR filter with the help of a linear array with $p$ processors so that $x_t$ is input at step $t$ and $y_t$ is output at step at most $t + 2p$.

**Example 2.3** We consider the linear array with $p$ processors.
We have two distinguished nodes 0 and \( p - 1 \) which do not have a left, respectively no right neighbor. All other nodes \( i \) have a left neighbor \( i - 1 \) and a right neighbor \( i + 1 \). We assume that \( p \) is odd and show how to parallelize Bubblesort.

**Algorithm 2.4 Odd-even transposition sort.**

// Each node receives \( \frac{n}{p} \) elements.

1. Each node sorts its subsequence with Quicksort;
2. for \( k = 1 \) to \( p \) do
   if \( k \) is odd then
      for \( i = 1 \) to \( \lfloor \frac{p}{2} \rfloor \) pardo
         node \( 2 \cdot i \) sends its sorted sequence to left neighbor \( j = 2 \cdot i - 1 \).
         \( j \) merges the two sorted sequences,
         keeps the smaller half and returns the larger half to neighbor \( 2 \cdot i \);
   else
      for \( i = 1 \) to \( \lfloor \frac{p}{2} \rfloor \) pardo
         node \( 2 \cdot i \) sends its sorted sequence to right neighbor \( j = 2 \cdot i + 1 \).
         \( j \) merges the two sorted sequences,
         keeps the larger half and returns the smaller half to neighbor \( 2 \cdot i \);

We have used pseudo-code, which uses standard “sequential statements” such as the for-do loop, but also has “parallel statements” such as the for-pardo loop which dictates a parallel execution. In particular, Algorithm 2.4 runs sequentially for exactly \( p \) steps, where in an odd (resp. even) step all even nodes transmit their sequences to their left (resp. right) neighbor in parallel. The odd nodes compare and return the larger (resp. smaller) half in parallel.

\[
\begin{array}{cccccc}
0 & 1 & 2 & 3 & 4 \\
\text{phase 1} & \cdot & \circ & \circ & \circ \\
\text{phase 2} & \circ & \circ & \cdot & \circ \\
\text{phase 3} & \cdot & \circ & \circ & \cdot \\
\text{phase 4} & \circ & \circ & \cdot & \circ \\
\text{phase 5} & \cdot & \circ & \circ & \circ \\
\end{array}
\]

Pattern of comparisons
**Theorem 2.5** Odd-even transposition sort (OETS) sorts $n$ keys on a linear array with $p$ processors. OETS executes $O\left(\frac{n}{p} \cdot \log_2 \frac{n}{p} + n\right)$ compute steps and $p$ communication steps involving messages of length $n/p$.

We first investigate the resources required by odd-even transposition sort. In the first step each node runs Quicksort for a sequence of length $n/p$ and hence time $O\left(\frac{n}{p} \cdot \log_2 \frac{n}{p}\right)$ is sufficient. In every of the subsequent $p$ steps a merging step is executed which runs in linear time $O(n/p)$. Obviously we have $p$ communication steps in which sequences of length $n/p$ are exchanged.

To show that OETS sorts correctly we first verify the important 0-1 principle which applies to oblivious comparison-exchange algorithms. The only operation of a comparison-exchange algorithm is to compare two keys in positions $i$ and $j$ and to swap dependent on the outcome of the comparison. For the algorithm to be oblivious the choice of $i$ and $j$ is pre-specified and does not depend on the outcome of previous comparisons.

**Lemma 2.6** The 0-1 principle.

An oblivious comparison-exchange algorithm sorts correctly iff it sorts 0-1 sequences correctly.

**Proof:** It suffices to show that an oblivious comparison-exchange algorithm $A$ sorts correctly, if it sorts 0-1 sequences correctly. Assume otherwise and there will be a sequence $x = (x_1, \ldots, x_n)$ which is incorrectly sorted. Let $\pi$ be an order type of $x$, i.e., we have

$$x_{\pi(1)} \leq x_{\pi(2)} \leq \cdots \leq x_{\pi(n)},$$

and assume that $A$ determines the wrong order type $\sigma$. We determine the smallest $k$ such that $x_{\sigma(k)}$ is “out of order”, that is $k$ is minimal with $x_{\pi(k)} < x_{\sigma(k)}$. Hence there is $l > k$ with $\sigma(l) = \pi(k)$. We concentrate on $x_{\pi(k)}$ and define the 0-1 sequence

$$y_i = \begin{cases} 0 & x_i \leq x_{\pi(k)}, \\ 1 & x_i > x_{\pi(k)}. \end{cases}$$

Observe that $y_{\pi(k)} = 1$ and $y_{\sigma(l)} = 0$. Since $x_i \leq x_j \Rightarrow y_i \leq y_j$, the sequence $y$ produces the same outcome of comparisons as sequence $x$. Thus algorithms $A$ “claims” the sequence

$$y_{\sigma(1)} \leq \cdots \leq y_{\sigma(k)} \leq \cdots \leq y_{\sigma(l)} \cdots$$

of inequalities. Bad luck, since $y_{\pi(k)} = 1$ and $y_{\sigma(l)} = 0$, and $A$ does not sort 0-1 sequences after all.

**Proof of Theorem 2.5.** We apply the 0-1 principle and have to show that Algorithm 2.4 sorts an arbitrary sequence $x$ of zeroes and ones. We first observe that, with the possible exception of the first step, the $n/p$ rightmost ones move right in every step until node $p - 1$.
is reached. (One of the $n/p$ rightmost ones is not moved in the first step, if it initially occupies an even node.)

What happens to a one among the second block of $n/p$ rightmost ones? Any one belonging to the rightmost block is moving right from step two onwards and hence will not delay a one from the “runner-up block” after step 1. Thus, if a one from the runner-up block is unlucky, then it is blocked in the first step by ones from the rightmost block and hence, sitting on an odd node, wins the next comparison against bits from the left neighbor and stays put as well. Thus the worst that can happen is that ones from the runner-up block move right after step 2 until they reach node $p − 2$.

An inductive argument shows that ones from the $i$th rightmost block move right after step $i$ until they reach node $p − i$. But then their distance is at most $(p − i) − 0 = p − i$ and they will reach their destination in the $p − i$ steps $i + 1, \ldots, p$. □

Let us analyze the running time of Algorithm 2.4 in dependence on the number $p$ of nodes. If $p ≤ \log n$, then Quicksort dominates and Algorithm 2.4 runs in time proportional to $\frac{n}{p} \cdot \log \frac{n}{p} = O\left(\frac{n}{p} \log n\right)$. We have reached a perfect parallelization in this case, since the best sequential sorting algorithm runs in time $O(n \cdot \log n)$. For $p > \log n$ we have $n = \Omega\left(\frac{n}{p} \log \frac{n}{p} + n\right)$ and the $p$ merging steps dominate and force Algorithm 2.4 to run in linear time $O(n)$. Thus there is no gain in running time when working with $\omega(\log n)$ nodes. Is Algorithm 2.4 a good sorting algorithm? The answer is certainly “yes”, if we have to compute on a linear array.

Exercise 22
(a) Let $G$ be an undirected graph with bisection width $w$ and $p ≤ n$ nodes. Show that any sorting algorithm requires $\Omega(n/w)$ compute steps to sort $n$ keys on $G$, provided only keys can be communicated.

(b) Show that $\Omega(n)$ compute steps are required, when $n$ keys are sorted on the linear array and only keys can be communicated. This claim holds for any number $p$ of nodes.

(c) Show that any sorting algorithm on the two-dimensional $\sqrt{p} \times \sqrt{p}$ mesh requires $\Omega\left(\frac{n}{\sqrt{p}}\right)$ compute steps to sort $n$ keys, when only keys can be communicated.

We define the work of a parallel algorithm as the product of the worst-case running time and the number of processors and observe that the work of Algorithm 2.4 is bounded by $O\left(p \cdot \frac{n}{p} \log \frac{n}{p} + p \cdot n\right) = O(n \cdot \log \frac{n}{p} + p \cdot n)$: for $p = O(\log n)$ the work of Algorithm 2.4 is therefore bounded by $O(n \cdot \log n)$. However each communication step involves sending/receiving $n/p$ keys and it dominates the compute step of merging two sorted sequences of length $n/p$.

If we have the option of designing a communication pattern tailor-made for sorting, then we can achieve work $O(n \log n)$ and running time $O\left(\frac{n}{p} \log \frac{n}{p}\right)$ for larger choices of $p$. For instance let us have a look at the two-dimensional mesh.

We use the odd-even transposition sort to describe shear sort, a sorting algorithm to sort $n^2$ keys on the two-dimensional $n \times n$ mesh $M_2(n)$. Shear sort works in $2 \cdot \log n$ phases, if $n$ is a power of two. In odd phases it sorts all rows in parallel and in even phases all columns
are sorted in parallel. To sort odd rows all smaller keys move left, to sort even rows all smaller keys move right and to sort columns all smaller keys move up. All sorting steps utilize the odd-even transposition sort for linear arrays. We show with the 0-1 principle that the final sequence determined by shearsort is sorted in snakelike order.

\[ \text{Lemma 2.7} \quad \text{Shearsort is correct.} \]

\[ \text{Proof:} \quad \text{We say that a row is clean, if it consists only of zeroes or only of ones, and dirty otherwise. Assume that} \ n \ \text{is a power of two. We claim that at any time we have clean 0-rows, followed by at most} \ n/2^k \ \text{dirty rows and finally only clean 1-rows appear. The claim is correct for} \ k = 0, \ \text{since we have initially at most} \ n \ \text{dirty rows. To show the inductive step for} \ k + 1, \ \text{we consider two rows in positions} \ 2i - 1 \ \text{and} \ 2i. \ \text{First both rows are sorted individually in snakelike order.} \]

\[ \text{Case 1:} \quad \text{the number of zeroes of row} \ 2i \ \text{is at least as large as the number of ones of row} \ 2i - 1. \ \text{After the first step of column sorting, row} \ 2i - 1 \ \text{turns into a clean 0-row. Subsequent steps of the column sort move the new 0-row upwards until it hits another 0-row.} \]

\[ \text{Case 2:} \quad \text{the number of zeroes of row} \ 2i \ \text{is less than the number of ones of row} \ 2i - 1. \ \text{After the first step of column sorting, row} \ 2i \ \text{turns into a clean 1-row. Subsequent steps of the column sort move the new 1-row downwards until it hits another 1-row.} \]

Now assume that we work with \( p \) instead of \( n^2 \) processors which we imagine to be arranged in a \( \sqrt{p} \times \sqrt{p} \) mesh of processors. We use the checkerboard decomposition and assign all keys \( x_{k,l} \) with \( k \in \{(i - 1) \cdot \sqrt{n/p} + 1, \ldots, i \cdot \sqrt{n/p}\} \) and \( l \in \{(j - 1) \cdot \sqrt{n/p} + 1, \ldots, j \cdot \sqrt{n/p}\} \) to processor \((i, j)\). Each processor uses quicksort to sort its \( n/p \) keys in time \( O(\frac{n}{p} \cdot \log_2 \frac{n}{p}) \). Any application of odd-even transposition sort involves \( \sqrt{p} \) processors and hence runs in time \( O(\sqrt{p} \cdot \frac{n}{p}) = O(\frac{n \sqrt{p}}{p}) \). Since shearsort applies odd-even transposition sort \( 2 \log_2 p \) times we obtain the following result.

\[ \text{Theorem 2.8} \quad \text{If} \ p \leq n, \ \text{then shearsort sorts} \ n \ \text{keys in time} \ O(\frac{n}{p} \cdot \log_2 \frac{n}{p} + \frac{n}{\sqrt{p}} \cdot \log_2 p) \ \text{on} \ M_2(\sqrt{p}). \ \text{It uses} \ O(\sqrt{p} \cdot \log_2 p) \ \text{point-to-point communication steps involving} \ \frac{n}{p} \ \text{keys.} \]
Observe that the work of shearsort is bounded by $O(n \cdot \log_2 \frac{n}{p} + \sqrt{p} \cdot n \cdot \log_2 p)$. Hence its work is bounded by $O(n \cdot \log_2 n \cdot \log_2 \log_2 n)$, whenever $p \leq \log^2_2 n$. However we can do better. Our next approach sorts $n$ keys recursively on the two-dimensional mesh $M_{\sqrt{n}}$: we recursively sort the four quadrants of $M_{\sqrt{n}}$ in snakelike order. Then we sort the rows in snakelike order followed by sorting the columns. Finally we sort the “snake” with $2 \cdot \sqrt{n}$ steps of odd-even transposition sort.

Exercise 23
Let $n$ be a power of two. Show that the recursive algorithm sorts $n$ keys in $O(\sqrt{n})$ compute steps on the two-dimensional $\sqrt{n} \times \sqrt{n}$ mesh.

Exercise 24
Show how to implement the recursive sorting procedure on $M_{2(\log_2 n)}$ such that $n$ keys are sorted in $O(\frac{n}{\log_2 n})$ compute steps with $p = \log^2_2 n$ processors.
Moreover show that $n$ keys can be sorted optimally (i.e., within work $O(n \cdot \log_2 n)$), whenever at most $p \leq \log^2_2 n$ processors are available. Observe that the odd-even transposition sort can sort $n$ keys optimally whenever $p \leq \log_2 n$.

Exercise 25
(a) The $n$-cell ring results from the $(n + 1)$-cell linear array by identifying the first and the last cell.

Show that an $n$-cell linear array can simulate an $n$-cell ring with a slowdown of at most 2. (You may assume that $n$ is even.)

(b) The $n \times n$ torus results from the $n \times n$ two-dimensional mesh after identifying the first and last column as well as the first and last row.
Show that an $n \times n$ torus and an $n \times n$ mesh have equivalent computational power up to constant factors. In particular, show that an $n \times n$ mesh can simulate an $n \times n$ torus with a slowdown of at most two.

**Exercise 26**

(a) Assume that $m$ divides $n$. Show how to simulate a mesh $M_2(n)$ on a mesh $M_2(m)$ with slowdown at most $O((n/m)^2)$. Can you do better?

(b) Show how to multiply two $n \times n$ matrices on a $p$-processor two-dimensional mesh in $O(n^3/p)$ steps for $p \leq n^2$. How efficient is this algorithm? (We assume that each processor receives $O(n^2/p)$ entries.)

### 2.3 Hypercube Architectures

We begin with another perspective on hypercubes.

**Definition 2.9** The $d$-dimensional hypercube $Q_d = (V_d, E_d)$ has node set $V_d = \{0, 1\}^d$. The edges of $Q_d$ are partitioned into the sets

\[
\{ \{w, w \oplus e_i\} \mid w \in \{0, 1\}^d \}
\]

of edges of dimension $i$ (for $i = 1, \ldots, d$) where $e_i = (0, \ldots, 0, 1, 0, \ldots, 0)$ is a binary vector with a one in position $i$.

Observe that $M_d(2) = Q_d$. Nodes of $Q_d$ are binary strings $u \in \{0, 1\}^d$ of length $d$, and two nodes $u, v$ are connected by an edge iff $v$ can be obtained from $u$ by flipping exactly one bit. Edges of dimension $i$ correspond to flipping the $i$th bit.

Note that $Q_d$ has $2^d$ nodes, $d \cdot 2^{d-1}$ edges and the (high) degree $d$. There is another equivalent recursive definition of the hypercube. The $d$-dimensional hypercube consists of two copies of the $(d - 1)$-dimensional hypercube (the 0-hypercube and the 1-hypercube) with edges between corresponding nodes in the two subcubes: there is an edge between node $0x$ in the 0-hypercube and node $1x$ in the 1-hypercube for any $x \in \{0, 1\}^{d-1}$.

We show that the $d$-dimensional hypercube $Q_d = (V_d, E_d)$ is a very robust graph in the following sense: consider how many edges must be removed to separate a subset $S \subseteq V_d$
2.3. HYPERCUBE ARCHITECTURES

![Figure 2.3: The 3-dimensional hypercube.](image)

![Figure 2.4: A 4-dimensional hypercube.](image)

of nodes from the remaining nodes \( \tilde{S} = V_d - S \). Assume that \( S \) is the smaller set, i.e. \(|S| \leq |\tilde{S}|\). A cut \( C(S, \tilde{S}) \) between \( S \) and \( \tilde{S} \) is a set of edges whose removal disconnects \( S \) and \( \tilde{S} \).

**Theorem 2.10** For every subset of nodes \( S \) with \(|S| \leq |\tilde{S}|\) we have \(|C(S, \tilde{S})| \geq |S|\).

**Proof:** By induction on \( d \). The base case \( d = 1 \) is trivial. For the induction step, take an arbitrary subset \( S \subseteq V_d \) such that \(|S| \leq |\tilde{S}|\); hence, \(|S| \leq 2^{d-1}\). Let \( S_0 \subseteq S \) be the set of nodes of \( S \) in the 0-hypercube, and \( S_1 \subseteq S \) be the set of nodes of \( S \) in the 1-hypercube.

**Case 1:** If \(|S_0| \leq 2^{d-1}/2 \) and \(|S_1| \leq 2^{d-1}/2 \), then applying the induction hypothesis to each of the subcubes shows that the number of edges between \( S \) and \( \tilde{S} \), even without considering edges between the 0-subcube and the 1-subcube, already exceeds \(|S_0| + |S_1| = |S|\).

**Case 2:** Suppose \(|S_0| > 2^{d-1}/2 \). Then \(|S_1| \leq 2^{d-1}/2 \). By the induction hypothesis, the number of edges in \( C(S, \tilde{S}) \) within the 0-subcube is at least \(|\tilde{S}_0| = 2^{d-1} - |S_0|\), and those within the 1-subcube is at least \(|S_1|\). But now there must be at least \(|S_0| - |S_1|\) edges in \( C(S, \tilde{S}) \) between the two subcubes, since every node \( 0x \in S_0 \) is connected with the unique node \( 1x \) in the 1-subcube and only \(|S_1|\) edges connect nodes in \( S \). Thus, the entire cut consists of at least \((2^{d-1} - |S_0|) + |S_1| + (|S_0| - |S_1|) = 2^{d-1} \) edges and we are done since \(|S| \leq 2^{d-1}\). \(\square\)
Corollary 2.11 The bisection width of \( Q_d \) is \( 2^{d-1} \).

**Proof:** We already know that the bisection width of \( Q_d \) is at most \( 2^{d-1} \). But Theorem 2.10 implies that at least \( 2^{d-1} \) edges must be removed to disconnect a subset of size \( 2^{d-1} \) and hence \( 2^{d-1} \) is a lower bound for the bisection width. \( \square \)

**Exercise 27**

Let \( G = (V,E) \) be an undirected graph. We say that a mapping \( f : V \rightarrow V \) is an automorphism of \( G \) iff \( f \) is bijective and

\[
\{u, v\} \in E \iff \{f(u), f(v)\} \in E
\]

holds for all \( u, v \in V \). Show that

\[
(x_1, \ldots, x_d) \mapsto (x_{\pi(1)}, \ldots, x_{\pi(d)})
\]

is an automorphism of \( Q_d \) for any permutation \( \pi \) of \( \{1, \ldots, d\} \).

Why does the hypercube play a prominent role among communication patterns? There are several reasons.

- The diameter of \( Q_d \) is \( d \) and hence logarithmic in the number of nodes. Thus any two nodes can communicate after logarithmically many steps.

- The previous property also applies to the complete binary tree \( T_d \). However \( T_d \) has severe bottlenecks, or, in other words, its bisection width is one: we partition \( T_d \) into two halves (of almost same size) when removing an edge that is incident to the root. But the bisection width of \( Q_d \) is \( 2^{d-1} \).

- \( Q_d \) has good routing algorithms. A first algorithm is the bit-fixing strategy: when moving a packet from \( u \) to \( v \), “fix” bits successively by moving the packet along those dimensions in which \( u \) and \( v \) differ. Bit-fixing is good for many routing problems, but may degenerate severely. It turns out that a randomized routing scheme (see Section 2.3.3) always does the job with high probability.

- A good network should be able able to simulate other important networks without significant congestion and delay. We show in this section that \( Q_d \) can simulate trees and meshes with ease.

- \( Q_d \) is tailor-made to simulate divide & conquer algorithms through its recursive structure.

**Exercise 28**

Show how to transpose a \( 2^d \times 2^d \) matrix on the hypercube \( Q_{2d} \) in time \( O(d) \).

Hint: bit fixing does not work. Use a divide & conquer algorithm instead.

**Exercise 29**

Let \( n = 2^d \). Show how to solve a solve a parallel prefix problem on the elements \( x_0, x_1, \ldots, x_{n-1} \) in time \( O(d) \) on \( Q_d \). We assume that processor \( u \) receives the element \( x_{|u|} \) as input, where \( |u| \) is the number with binary representation \( u \), and outputs \( x_0 \ast \cdots \ast x_{|u|} \).
2.3. HYPERCUBE ARCHITECTURES

2.3.1 Simulating Trees and Meshes

We first worry about simulating trees. We have apparently bad news, since the complete binary tree $T_d$ is not a subgraph of $Q_{d+1}$. Why? We observe that hypercube edges flip the parity of nodes and hence all nodes of $Q_{d+1}$ corresponding to tree nodes of fixed depth have the same parity. But then the $2^{d-2} + 2^d > 2^d$ hypercube nodes corresponding to tree nodes of depth $d - 2$ and $d$ have the same parity and this is impossible since there are exactly $2^d$ hypercube nodes of parity zero respectively one in $Q_{d+1}$.

However we can simulate normal tree algorithms easily by normal algorithms on the hypercube.

Definition 2.12 (a) An algorithm with communication pattern $T_d$ is a normal tree algorithm, if at any time only nodes of identical depth are used. Moreover, if only nodes of depth $i$ are used at time $t$, then only nodes of depth $i - 1$ or only nodes of depth $i + 1$ are used at time $t + 1$.

(b) We say that an edge $e = \{u, v\}$ of the hypercube $Q_d$ is active at time $t$ iff $u$ and $v$ exchange information at time $t$.

An algorithm with communication pattern $Q_d$ is a normal hypercube algorithm iff at any time, edges of only one dimension are active and if edges of dimension $i$ are active at time $t$, then at time $t + 1$ edges of dimension $\begin{cases} 1 & i = d, \\ i + 1 & \text{otherwise} \end{cases}$ or $\begin{cases} d & i = 1, \\ i - 1 & \text{otherwise} \end{cases}$ are active.

![Figure 2.5: Three steps of a normal algorithm on $Q_3$.](image)

Theorem 2.13 Any normal tree algorithm on $T_d$ can be simulated by a normal hypercube algorithms on $Q_d$ without any slowdown. In particular, the hypercube algorithm sends no more messages than the normal tree algorithm.

Proof: We label an edge of $T_d$ leading to a left child by zero and an edge leading to a right child by one. Thus in particular, if $\text{label}(v)$ is the 0-1 sequence of labels on the path from the root to $v$, then we can identify $v$ by $\text{label}(v)$. We simulate the tree node $v$ by the hypercube node $v^* = \text{label}(\text{lm}(v))$, where $\text{lm}(v)$ is the leftmost leaf in the subtree with root $v$. 


CHAPTER 2. COMMUNICATION PATTERNS

The mapping \( v \mapsto v^* \) is not injective, since the \( d + 1 \) tree nodes \( \varepsilon, 0, 0^2, \ldots, 0^d \) for instance are all mapped to the hypercube node \( 0^d \), but we also have the following positive properties:

- Tree nodes of same depth \( t \) are simulated by distinct hypercube nodes from the set \( \{0, 1\}^t \cdot 0^{d-t} \). Thus a communication with children is easy on hypercubes using the edges \( \{w0^{d-t}, w10^{d-t-1}\} \) and a tree node and its left child are simulated by the same hypercube node.

- Edges to right children, originating in tree nodes from the same layer, are therefore mapped into hypercube edges of identical dimension.

Thus we may simulate any normal tree algorithm without slowdown.

We show next how to embed a generalized class of meshes.

**Definition 2.14** We define the generalized mesh \( M_d(n_1, \ldots, n_d) \), for numbers \( n_1, \ldots, n_d \in \mathbb{N} \), as the undirected graph with node set

\[ V = \times_{i=1}^d \{1, \ldots, n_i\}. \]

As for conventional meshes we connect two nodes \( u \) and \( v \) iff their distance \( \sum_{i=1}^d |u_i - v_i| \) is exactly one.

**Theorem 2.15** Assume that \( n_1, \ldots, n_d \) are powers of 2. If

\[ D = \sum_{i=1}^d \log_2 n_i, \]

Figure 2.6: The leftmost leaf representation of tree nodes.
then $M_d(n_1, \ldots, n_d)$ is a subgraph of $Q_D$. In particular, any mesh algorithm $\mathcal{M}$ can be simulated by a hypercube algorithm which sends no more messages than $\mathcal{M}$.

**Proof:** We first observe that the hypercube has a Hamiltonian path. (Why? Proceed inductively, run the Hamiltonian path first in the subcube $0\{0,1\}^d$, then ”move up” into the subcube $1\{0,1\}^d$ and run the path backwards.)

Let $P_j$ be a Hamiltonian path for the hypercube $Q_{\log_2 n_j}$. We map node $(i_1, \ldots, i_d)$ of $M_d(n_1, \ldots, n_d)$ to the hypercube node $v^1(i_1) \circ \cdots \circ v^d(i_d) \in Q_D$, where $v^j(i)$ is the $i$th node of $P_j$, and $\circ$ denotes the concatenation of strings.

The mapping is obviously injective. Moreover any edge of $M_d(n_1, \ldots, n_d)$ connects two nodes which differ in exactly one coordinate $j$ and their difference (in absolute value) is one. Thus, if the first endpoint is mapped to $v^1(i_1) \circ \cdots \circ v^j(i_j) \circ \cdots \circ v^d(i_d)$, then the second endpoint is mapped to (say) $v^1(i_1) \circ \cdots \circ v^j(i_j + 1) \circ \cdots \circ v^d(i_d)$ and the two hypercube nodes are connected, since $v^j(i_j)$ and $v^j(i_j + 1)$ are connected in $Q_{\log_2 n_j}$. □

### 2.3.2 Butterflies

$Q_d$ has however the severe disadvantage of a large degree. We therefore propose butterfly networks, a degree four network which is capable of simulating the hypercube quickly.

![Butterfly Network $B_3$](attachment:butternet.png)

The butterfly network $B_3$.

**Definition** 2.16 The nodes of the $d$-dimensional butterfly network $B_d$ are pairs $(u, i)$ with $u \in \{0,1\}^d$ and $0 \leq i \leq d$. Nodes $(u, i)$ and $(v, i+1)$ are joined by a “horizontal edge” iff $u = v$ or by a “crossing edge” iff $u$ and $v$ differ only in their $(i+1)$st bit. We say that $C_i := \{(w, i) \mid w \in \{0,1\}^d\}$ is the $i$th column and $R_w := \{(w, i) \mid 0 \leq i \leq d\}$ is the row of $w \in \{0,1\}^d$.
Butterfly networks are sometimes introduced differently by identifying the first and last column. This variant is called the *wrapped butterfly* $B_d^*$. Here are some important properties of $B_d^*$.

- **The recursive structure of $B_d$:** $B_d$ is composed of two disjoint copies of $B_{d-1}$ on columns $C_0, \ldots, C_{d-1}$. The first copy occupies the even rows $R_0, \ldots, R_{d-1}$ and the second copy occupies the odd rows $R_{d}$ (for $u \in \{0,1\}^{d-1}$). (Observe that edges ending in a column $C_i \ (i < d)$ connect even with even and odd with odd rows.) Two more copies can be found on columns $C_1, \ldots, C_d$ with the first copy occupying rows $R_{0u}$ and the second copy occupying rows $R_{1u}$ (for $u \in \{0,1\}^{d-1}$).

- **The butterfly network $B_d$ and the hypercube $Q_d$:** $B_d$ is tailor-made to simulate parallel computations of the hypercube $Q_d$. The nodes of a row $R_u$ play the role of node $u$ of the hypercube. Observe moreover that hypercube edges of dimension $i$ correspond to edges connecting nodes of $C_{i-1}$ to nodes of $C_i$.

**Lemma 2.17**

(a) The butterfly network $B_d$ and the wrapped butterfly have the same number $d \cdot 2^{d+1}$ of edges and both have degree 4. $B_d$ has $(d+1) \cdot 2^d$ nodes, the wrapped butterfly has $d \cdot 2^d$ nodes.

(b) The diameter of $B_d$ is $2 \cdot d$ and decreases to $\lfloor \frac{3d}{2} \rfloor$ for the wrapped butterfly.

(c) $B_d$ has bisection width $\Theta(2^d)$ and its wrapped version has bisection width $2^d$.

**Exercise 30**

Show Lemma 2.17 (a) and (b).

The advantage of the butterfly network is its low degree of 4, its disadvantage is the $O(d)$ delay in simulating arbitrary computations of $Q_d$. But wrapped butterflies are very fast when simulating normal algorithms on the hypercube.

**Theorem 2.18** If a normal algorithm on $Q_d$ starts by communicating along dimension one, then it can be simulated on $B_d^*$ with a slowdown of two. In particular, any hypercube algorithm $H$ on $Q_d$ can be simulated by a butterfly algorithm on $B_d^*$ which sends twice as many messages as $H$.

In later applications we assume a slightly different perspective and interpret a row as a single process. Horizontal edges indicate successive time steps and crossing edges stand for hypercube communications. Thus we use butterfly communications to represent normal hypercube algorithms and achieve under this perspective no slowdown whatsoever.

A further important property of $B_d$ or $B_d^*$ is its fast simulation by smaller butterflies.

**Exercise 31**

Assume that $d < e$. Then any algorithm $B_e$ on $B_e$ can be simulated by an algorithm $B_d$ on $B_d$ which sends no more messages than $B_e$ and actually may send fewer messages. The compute time of $B_d$ grows by at most the factor $2^{e-d}$ in comparison to $B_e$. 

2.3. HYPERCUBE ARCHITECTURES

2.3.3 Routing on Hypercubes

Our goal is to implement the communication primitive “permutation routing”: we are given a network $G = (V, E)$, a permutation $\pi : V \rightarrow V$ and assume that each node $v \in V$ wants to send a message packet $P_v$ along some path from $v$ to node $\pi(v)$. We require that only one packet may traverse an edge at any time and try to route all packets to their destination as quickly as possible. Hence we have to construct congestion-free routing.

We call a routing algorithm conservative, if the path of any packet only depends on the address $\pi(v)$ of the destination. This class of algorithms seems reasonable, but deterministic algorithms within this class are forced to fail!

Fact 2.1 [KKT91] Let $G$ be an undirected graph of $N$ nodes. Assume that each node in $G$ has at most $D$ neighbors. Furthermore let $A$ be a conservative, deterministic routing algorithm. Then there is a permutation $\pi$ for which $A$ requires at least $\Omega(\sqrt{N/D})$ steps.

Exercise 32
Let $G = (V, E)$ be an arbitrary connected undirected graph with $N$ nodes and degree $D$. A conservative routing algorithm is uniquely determined by the $\binom{N}{D}$ paths $P(u, v)$ from $u$ to $v$ (with $u \neq v$). We define $P(v)$ as the set of all such paths ending in $v$ and $S_k(v)$ as the set of all nodes $x \in V$ which are traversed by at least $k$ paths from $P(v)$.

(a) Show that $|P(v)| \leq |S_k(v)| + |S_k(v)| \cdot D \cdot (k - 1)$ holds and hence $|S_k(v)| \geq N/(D(k - 1) + 1)$ follows. Hint: How many paths from $P(v)$ run along an edge $\{u, w\}$ with $u \in S_k(v)$ and $w \in S_k(v)$?

(b) Show that there is a node $x \in V$ and a permutation $\pi$ of $V$ such that at least $k = \sqrt{N/D}$ paths $P(u, \pi(u))$ traverse $x$.

(c) Show that for any conservative routing algorithm $A$ there is a permutation $\pi$ of $V$ for which $A$ runs for at least $\Omega(\sqrt{N/D^2})$ steps.

Example 2.4 We consider the $d$-dimensional hypercube $Q_d$. The bit-fixing strategy checks the address bits beginning with position 1 and ending with position $d$. If position $i$ is considered and the packet $P_v$ with address $\pi(v)$ has reached node $w$, then the packet does not move, if $w$ and $\pi(v)$ agree in their $i$th bit. Should they however disagree, then $P_v$ is sent along the edge corresponding to bit $i$. (For instance, $(111, 011, 001, 000)$ is the bit-fixing path from $v = 111$ to $\pi(v) = 000$.)

Bit-fixing fails for the routing problem $(x, y) \rightarrow (y, x)$, where $x$ (resp. $y$) is the first (resp. second) half of all bits of the sender $v$. What happens? The $\sqrt{2^d}$ packets of the sending nodes $(x, 0)$ clog the bottleneck $(0, 0)$.

Exercise 33
(a) We have to implement a partial routing problem on the $d$ Butterfly network $B_d$, where some nodes in column $C_d$ wish to communicate with nodes in column $C_0$. The goal is that the $i$th node (among nodes wishing to communicate) sends its packet to the $i$th node in column $C_0$. In other words, the order among packets to be sent is preserved and gaps are closed.

Show how to implement this routing problem in time $O(d)$. Hint: bit fixing.

(b) Our goal is to implement radix sort on $B_d$ assuming that node $(u, d)$ receives the key $x_u$. Show how to implement one phase of radix sort in time $O(d)$.
If we consider the $d$-dimensional hypercube $Q_d$, then $d$ is the diameter, that is the maximal distance between any two nodes. Thus we should expect a good routing algorithm to run in time $O(d)$. However fact 2.1 forces the clearly unacceptable running time of $\Omega(\sqrt{2^d/d})$ in the worst-case. Randomized conservative algorithms however face no problems on the hypercube.

**Algorithm 2.19** Randomized routing on the hypercube

1. The permutation $\pi$ describes the routing problem on the $d$-dimensional hypercube.
2. **Phase 1**: Sending packets to a random destination.
   Each packet $P_v$ chooses a destination $z_v \in \{0, 1\}^d$ at random. Packet $P_v$ then follows the bit-fixing path from $v$ to $z_v$.
3. **Phase 2**: Sending packets to the original destination.
   Packet $P_v$ follows the bit-fixing path from $z_v$ to $\pi(v)$.

Observe that we do not require any queuing strategies whatsoever. The only requirement is that we send some packet along an edge $e$, if at least some packet is waiting in the queue of edge $e$. We begin with the analysis of phase one. The following observations are crucial.

- If two packets $P_v$ and $P_w$ meet at some time during the first phase, then they will not meet again later in phase one. (That is, if two packets $P_v$ and $P_w$ meet at some time and diverge at some later time then the two paths will be node-disjoint from that point on.)

- Assume that packet $P_v$ runs along the path $(e_1, \ldots, e_k)$. Let $\mathcal{P}$ be the set of packets different from $P_v$ which run along some edge in $\{e_1, \ldots, e_k\}$ during phase one. Observe that the waiting time of $P_v$ is bounded by $|\mathcal{P}|$ from above.

The second observation is not obvious, since a packet $P_w$ may repeatedly block a packet $P_v$. (Two packets block each other, if both belong to the same queue at some time.) Hence we establish this observation by induction on the size of $\mathcal{P}$. The basis $|\mathcal{P}| = 0$ is trivial and we assume $|\mathcal{P}| = n + 1$.

**Exercise 34**

Show that there is packet $P_w \in \mathcal{P}$, that blocks $P_v$ at most once.

The packet $P_w$ from the above exercise is only responsible for $P_v$ waiting for one time step. We remove $P_w$ from $\mathcal{P}$ and obtain the claim from the induction hypothesis. For the rest of the analysis we introduce the random variables

$$ H_{v,w} = \begin{cases} 
1 & P_v \text{ and } P_w \text{ run over at least one joint edge}, \\
0 & \text{otherwise}.
\end{cases} $$
Due to the last observation, the total waiting time of $P_v$ during phase one is upper-bounded by $\sum w H_{v,w}$. How large is the expected value of $\sum_w H_{v,w}$ and how likely are large deviations? Assume that $W_i$ is the number of bit-fixing paths which traverse the $i$th edge of $P_v$. Then

$$E \left[ \sum w H_{v,w} \right] \leq \frac{d^2}{2} \cdot E[W_1]. \quad (2.1)$$

Why? The randomized routing of phase 1 guarantees that all edges carry the same expected load. In particular, all edges carry the expected load $E[W_1]$ of the first edge of $P_v$ and the claim follows, since the path travelled by $P_v$ has expected path length $d^2/2$. (Observe that the right hand side of (2.1) counts a colliding packet multiple times, namely once for each contested edge.)

We can determine the expected load $E[W_1]$ as follows: the expected path length of a packet is $d^2/2$ and hence the $d \cdot 2^{d-1}$ edges of the hypercube all carry the same expected load

$$E[W_1] = \frac{d^2 \cdot 2^{d}}{2^d \cdot 2^{d-1}} = 1.$$  

Thus we get $E[\sum w H_{v,w}] \leq \frac{d^2}{2}$ as a consequence of (2.1). We apply the Chernoff bound (Theorem 1.8) and get

$$\text{prob} \left[ \sum w H_{v,w} \geq (1 + \beta) \cdot \frac{d^2}{2} \right] \leq e^{-\beta^2 \cdot \frac{d^2}{2^{d-1}}}.$$  

We set $\beta = 3$ and the probability of a total waiting time of at least $2 \cdot d$ is upper-bounded by $e^{-3 \cdot d/2}$. Since we are dealing with $2^d$ packets, the probability that some packet requires more than $2 \cdot d$ waiting steps is upper-bounded by $2^d \cdot e^{-3 \cdot d/2} \leq e^{-d/2}$. The analysis of the second phase is completely analogous and we can summarize.

**Theorem 2.20** The following statements hold with probability at least $1 - 2 \cdot e^{-d/2}$:

(a) each packet reaches its destination in each phase after at most $3 \cdot d$ steps,

(b) each packet has reached its final destination after at most $6 \cdot d$ steps.

**Exercise 35**

In the routing problem for a general networks we assume that packets are to be routed according to a partially defined permutation.

Assume that we can sort $n$ keys in time $t(n)$ on a given network $\mathcal{N}$. Show that any partial permutation can be routed in time $O(t(n))$ on $\mathcal{N}$.

Now assume that we are given an arbitrary undirected graph $G$ with $2^d$ nodes and degree $c$. Then it can be shown that the set of edges of $G$ is the union of at most $c$ matchings, where a matching is a set of node-disjoint edges.
Exercise 36
Show that each edge of an undirected graph of degree $c$ belongs to one of at most $c$ matchings.

We can interpret each matching as a partially defined permutation which we know how to route in time $O(d)$ with high probability and hence we can simulate any graph with $2^d$ nodes and degree $c$ in time $O(c \cdot d)$ on the $d$-dimensional hypercube.
Chapter 3

The Message Passing Interface

The message passing interface, MPI for short, is a message passing library standard which can be used in conjunction with conventional programming languages such as C, C++ or Fortran. Free versions of MPI libraries are available and MPI algorithms can be ported on a variety of different platforms, although the algorithmic performance may vary from platform to platform. Nowadays MPI is the de-facto standard when writing parallel scientific applications for high performance clusters.

MPI supports the point-to-point Send and Receive operations between a specified sending process and a specified receiving process through its communication primitives, the `MPI_Send` and `MPI_Recv` commands. `MPI_Send` basically takes as arguments the location of the data to be sent as well as the description of the receiving process. The arguments of its counterpart `MPI_Recv` describe the sending process and the memory location destined for the message.

There are several versions of send, dependent on whether message buffering is allowed, a matching “ready” message from the receiver is required or the entire communication has to terminate. The sending process starts communication after receiving a request from the recipient or after receiving a ready message from the recipient as reply to its own request to send. As long as the recipient has not replied, the sender either has to leave its send buffer unchanged or it may copy the send buffer into a system buffer. In both cases the particular version of send either allows communication to continue as a background process or it may force the sender to wait in order to synchronize communication. If the recipient has replied, the message is copied from its respective buffer into the communication buffer (such as the TCP/IP buffer) and the “bits flow into the cable”.

A communication routine is blocking, if the completion of the call is dependent on certain events such as successful message delivery or message buffering. Network interfaces transport messages from the communication buffer to their target location without processor intervention and hence non-blocking communication helps to mask the communication overhead. Non-blocking operations are more efficient than blocking operations, but now unsafe memory access has to be avoided and additional code has to check the status of the
communications process running in the background. We mention the following variants of
the send operation with MPI_Isend as the only non-blocking and MPI_Bsend, MPI_Ssend
as at least “partially non-blocking” operations:

- **MPI_Send**: MPI copies short messages into a system buffer, where the message
  length depends on the system in use. For a long message the sender has to wait until
  the message is successfully delivered. (Blocking, send buffer can be reused.)

- **MPI_Ssend** (synchronous send): terminates only if all bits have left the send buffer
  and the receiver has begun reception. At this time sender and receiver have synchro-
nized. (Blocking, send buffer can be reused.)

- **MPI_Bsend** (buffered send): If no matching receive operation is posted, then MPI
  copies the message into a user controlled buffer. The sender can resume work without
  waiting for the successful delivery of the message. (Blocking, send buffer can be re-
  used.)

- **MPI_Isend** (immediate send): the request to send is posted immediately and the
  sender can resume work. (Non-blocking, send buffer should not be reused.)

Based on its communication primitives MPI supports a variety of collective communication
functions, in which a group of processes cooperates to distribute or gather a set of values.
This set of operations is based on the concept of a communicator, which defines a set of
processes and attributes of this set. One such attribute is the topology of the communicator
such as mesh topologies and general graph topologies. Processes receive coordinates and
can be addressed by these coordinates.

We list the following fundamental collective communication functions assuming that p
processes are participating.

- **MPI_Bcast** is a one-to-all broadcast, namely a specified value is to be sent to all
  processes of the communicator.

- **MPI_Scatter**, a one-to-all personalized broadcast sends messages $M_1, \ldots, M_p$ from
  a root process to all other processes of the communicator. In particular, the $i$th
  process of the communicator is to receive message $M_i$.

- **MPI_Gather** is the counterpart of MPI_Scatter. This time the $i$th process sends a
  message $M_i$ to a specified root process.

- **MPI_Allgather** is an all-to-all broadcast operation: each process $i$ specifies a mes-
  sage $M_i$ that it wants to broadcast. At the completion of the all-to-all broadcast
  each process of the communicator has to know all messages $M_1, \ldots, M_p$.

- **MPI_Alltoall** is an All-to-all personalized broadcast: each process $i$ specifies mes-
  sages $M_{ij}$ that it wants to send to process $j$. At the completion of the broadcast
  process $j$ has to know all messages $M_{ij}$, $M_{pj}$. 
Example 3.1 Assume that we want to determine the transpose of a $p \times p$ matrix $A$, where process $i$ initially stores the $i$th row and is supposed to finally store the $i$th column. All we have to do is to implement an all-to-all personalized broadcast in which process $i$ sends $M_j^i = A[i,j]$ to process $j$.

MPI also supplies collective communication functions in which data is manipulated.

- In MPI\_Reduce an associative operation $\ast$ and a root process is distinguished. The result $M_1 \ast \cdots \ast M_p$ has to be assigned to the root process. One can choose for instance from the following list of operations: maximum, minimum, sum, product, and, or, xor, bitwise and, bitwise or, bitwise xor.

- MPI\_Allreduce works as MPI\_Reduce, but the result is distributed to all processes of the communicator.

- MPI\_Scan is the prefix version of MPI\_Reduce: process $i$ has to receive the “sum” $M_1 \ast \cdots \ast M_i$.

The above MPI operations belong to MPI version 1. In Section 3.1 we analyze the communication resources required for each of the above MPI functions.

3.1 Analyzing MPI Programs: The LogGP Model

Already the analysis of a sequential algorithm is a challenge, since the speed of the processor, access times to caches, DRAM-memory and external memory differ considerably and, as a consequence, the new area of algorithm engineering is emerging. But counting the number of “dominant operations” (such as the number of additions and multiplications when computing a matrix-vector product or the number of comparisons when sorting) and taking the memory hierarchy (cache, main memory, external memory) into account one obtains in many cases reliable estimates of the actual running time. The situation for parallel algorithms on multicomputers is even more complex, since the cost of communicating overwhelms the cost of local computing or memory operations, slowing down the algorithm in some and being harmless in other cases.

Which characteristics of a parallel machine are of interest when evaluating a parallel algorithm? In the LogGP model [CKP93, AISS95] the following parameters are distinguished as crucial parameters when determining the algorithmic performance on a wide variety of parallel platforms and it is argued that these parameters suffice for an approximate estimate of the actual running time.

- $L$ is an upper bound on the latency, which is the length of the time interval beginning when the first bit “hits the cable” and ending with receiving the last bit for a (short) point-to-point message of the canonical message length $w$. 

- During the time interval of length $o$ the processor involved in the communication process cannot perform any other operation. The overhead does depend on the variant of the send operation, for instance if message buffering is required. Therefore we single out the synchronous MPI $S_{send}$. Header information has to be supplied, the message has to be copied into the communication buffer and the sender-receiver handshake has to complete (verifying that the recipient is ready to receive).

- $g$ is the gap parameter. $g$ is defined as the minimum time interval between consecutive message transmissions or consecutive message receptions at a processor. $\frac{1}{g}$ is the available per-processor communication bandwidth.

- $G$ is defined as the time per byte for long messages. $\frac{1}{G}$ is the available per-processor communication bandwidth for long messages.

- $P$ is the number of processors.

The LogGP model has several advantages compared to other analysis models. For instance, it makes sense to differentiate overhead $o$ and latency $L$ (instead of considering $o + L + o$ as new latency), since the processor is free for $L$ steps to compute locally, but occupied for $o$ steps. Moreover short and long messages are treated differently to capture performance gains if the platform provides additional support for long messages. The model seems however rather coarse, since one does not distinguish between sending and receiving overheads $o_S, o_R$ and also tacitly assumes that messages are routed without congestion. However the difference between $o_R$ and $o_S$ is not considerable and it makes sense to postulate congestion-free routing, since only “good” algorithms deserve an analysis.

The relative sizes of the various parameters vary from architecture to architecture. For instance the current Myrinet implementation of the CSC cluster has a bandwidth of 1.92 Gbit/sec and a latency of about $7 \mu s$, whereas Gigabit Ethernet has a bandwidth of 1 Gbit/sec and a latency of about $100 \mu s$. The canonical message length $w$ is 16 KB and since $\frac{128K{}_{bit}}{1G{}_{bit}} \approx 66 \cdot 10^{-6}$ (resp. $\frac{128K{}_{bit}}{1G{}_{bit}} = 128 \cdot 10^{-6}$) we obtain $g \approx 66 \mu s$ for Myrinet (resp. $g \approx 128 \mu s$ for Gigabit Ethernet). Determining exact bounds for the overhead is difficult, but experiments give $o \approx 70 \mu s$ as an approximation for MPI $S_{send}$ on the Myrinet and gap and overhead almost coincide.

Below we assume that all parameters are measured as multiples of the processor cycle. We now describe implementations of the MPI functions discussed in the previous section and analyze our implementations in the LogGP model.

**MPI $S_{send}$**

When sending a small message (of length $w$), we incur the overhead $o$ at the sending processor, the network latency $L$ and the overhead $o$ at the receiving processor. Moreover, if we send a long message of $n$ bytes, then this message is broken up into $\lceil n/w \rceil$ messages of length $w$. The sending and receiving processors incur the overhead $o$ for every partial
message, but only the larger of overhead and gap has to be counted. Moreover observe that only the latency of the last message is of relevance, since we may inject new messages after \( g \) steps. Thus the total delay for the \( n \)-byte message is

\[
T_{\text{Send}} = o + \left( \frac{n}{w} - 1 \right) \cdot \max\{o, g\} + L + o = O(n)
\]

processor cycles, if we assume a machine \textit{without} additional support for long messages. The sending processor however loses “only” time \( o + \left( \frac{n}{w} - 1 \right) \cdot o \). If however additional support for long messages is provided, then the \( k \)-byte message incurs the delay

\[
T'_{\text{Send}} = o + (n-1) \cdot G + L + o = O(n).
\]

(The first byte goes after \( o \) steps “into the wire” and subsequent bytes follow in intervals of length \( G \). The last byte exits the wire at time \( o + (n-1) \cdot G + L \). Observe that the sending and receiving processor are kept busy only at the very beginning and at the very end. At all other times the processors may go back to their local computation and thus overlap computation with communication.)

Observe that (3.1) and (3.2) coincide for \( k = 1 \). Neither in (3.1) nor in (3.2) does latency play a decisive role for long messages. Dependent on the platform the difference between the breakup of long messages into short ones (see (3.1)) and supporting long messages directly (see (3.2)) can be considerable [AISS95].

**MPI_Bcast**

To analyze the \textit{one-to-all broadcast} we have to first define the binomial trees \( B_k \) with \( 2^k \) nodes. \( B_0 \) is a singleton tree. If \( B_k \) is defined, then we obtain \( B_{k+1} \) from \( B_k \) by attaching the root \( r_2 \) of a second copy \( T_2 \) of \( B_k \) as child of the root \( r_1 \) of a first copy \( T_1 \). Observe that \( B_k \) has depth \( k \).

![Figure 3.1: The first four binomial trees.](image)

We assume that \( r \) broadcasts a message \( M \) of length \( w \). \( r \) initiates the broadcast by sending \( M \) to process \( s \). Now \( r \) continues with recursively broadcasting \( M \) in its tree \( T_1 \), whereas \( s \) broadcasts \( M \) in its tree \( T_2 \). The broadcast then proceeds recursively and is hence called...
recursive doubling. To analyze the communication time we first observe that $r$ feeds $M$ after $o$ cycles into the network. $s$ receives $M$ after $o + L + o$ steps and it can feed $M$ after a total of $(o + L + o) + o$ cycles into the network. Thus a child of $s$ receives $M$ after $2 \cdot (o + L + o)$ cycles and, with an inductive argument, we obtain

$$T_{\text{Bcast}} \leq \lceil \log_2 p \rceil \cdot (o + L + o)$$

as the total number of required cycles. The gap parameter does not play a role, if $g \leq o + L + o$: in this case $r$ sends its second message before $s$ sends its first message.

The broadcasting problem in $T_1$ however is solved faster, since $r_1$ does not have to wait until $r_2$ receives $M$. Thus we can do better, if instead of using the binomial tree we use a tree with a higher fanout for the root $r_1$. The higher fanout helps us to keep $r_1$ busy sending messages while $r_2$ is waiting for $M$. The choice of the new fanout depends on the relative sizes of $L, o$ and $g$; thus $g$ does play a role in an optimal algorithm.

**Exercise 37**

Assume that the latency $L$, the overhead $o$, the gap $g$ and the number $p$ of processors is known. Design an efficient sequential algorithm which determines an optimal broadcast tree, that is a broadcast tree which supports a broadcast within minimal time.

Hint: use dynamic programming.

**Exercise 38**

Assume that we have to broadcast a message of general length $n$.

(a) Determine the number of cycles, if the binomial tree algorithm is used. What is the asymptotic performance?

(b) MPI uses binomial trees for short messages, but applies a combination of Scatter and Allgather for long messages. Sketch the details and analyze the performance.

**MPI_Scatter and MPI_Gather**

The communication pattern for one-to-all personalized broadcast is identical to the one-to-all broadcast and uses the binomial tree approach. However the source processor $r_1$ starts by sending the concatenated message $M_{p/2+1} \circ \cdots \circ M_p$ to $r_2$. This approach is recursively repeated by subsequent processors who break up the concatenation and propagate the corresponding substring of messages. To analyze the total communication time we assume that long messages are supported and obtain $o + (\frac{p}{2} \cdot n - 1) \cdot G + L + o$ cycles in the first round, if all messages have length $n$. Hence an inductive argument shows that

$$T_{\text{Scatter}} \leq \lceil \log_2 p \rceil \cdot (o + L + o) + \sum_{k=1}^{[\log_2 p]} \left( \frac{p}{2^k} \cdot n - 1 \right) \cdot G$$

$$\leq \lceil \log_2 p \rceil \cdot (o + L + o) + p \cdot n \cdot G = O(p \cdot n)$$

cycles suffice. Not surprisingly we can improve our approach by increasing the fanout of $r_1$ again to keep it busy while $r_2$ is waiting. Observe that the one-to-all personalized broadcast
is quite a bit more expensive than the simple broadcast, since $G$ is weighted by $p$. The MPI\_Gather operation is implemented analogously and requires the same resources.

**MPI\_Allgather**

We describe two implementations of the **all-to-all broadcast**. We first use the linear array as communication pattern. In particular we pump all messages through the network once via pipelining: process $i$ sends message $M_i$ to process $i + 1$. Then it waits for message $M_{i-1}$ and forwards it immediately to process $i + 1 \ldots$ Hence the communication time is at most

$$T_{\text{Allgather},1} \leq (o + L + o) \cdot (p - 1),$$

if all messages have length $w$. Here we assume $g \leq o + L + o$.

In our second implementation we use the hypercube of dimension $\log_2 p$ as communication pattern and perform recursive doubling: processor $u = u_1u_2u'$ sends its message $M_u$ to neighbor $u_1u_2u'$, receives message $M_{u_1u_2u'}$ in return and computes the concatenation $M_{0u_2u'} \circ M_{1u_2u'}$. This procedure is repeated for neighbor $u_1u_2u'$ and afterwards $u$ stores the concatenation $M_{0u_1u'} \circ M_{10u'} \circ M_{01u'} \circ M_{11u'}$. Thus, if all messages have the same length $n$, we get

$$T_{\text{Allgather},2} \leq \sum_{k=1}^{\lfloor \log_2 p \rfloor} \left[ o + \left( \frac{p}{2^k} \cdot n - 1 \right) \cdot G + L + o \right]$$

$$\leq \left[ \log_2 p \right] \cdot (o + L + o) + p \cdot n \cdot G = O(p \cdot n),$$

provided all messages have the same length $n$. Thus MPI\_Scatter and MPI\_Allgather have identical upper bounds. Observe however that in both of our implementations for MPI\_Allgather all processes are busy most of the time.

**Exercise 39**

What is the performance of the linear array approach, if all messages have same length $n$? Which approach, linear array or hypercube, is better assuming extra support for long messages?

In both implementations of Allgather all processors have to carry the same load whereas the tree implementation of Gather and Scatter are unbalanced and allow processors to continue with their local computations until their are called upon by their parent. Moreover, a higher root fanout improves Scatter, but since all processors in Allgather start simultaneously, comparable improvements cannot be obtained for Allgather. Thus from the perspective of the average load, it may be preferable to first run Gather followed by a broadcast.

**All-To-All Personalized Broadcast**

We show next an implementation of the **all-to-all personalized broadcast** which uses the $\log_2 p$-dimensional hypercube as its communication pattern. We have $p - 1$ phases. In
CHAPTER 3. THE MESSAGE PASSING INTERFACE

phase $b$, for $b \in \{0, 1\}^{\log_2 p}$ with $b \neq 0$, processor $v$ sends its message $M^v_{v \oplus b}$ to processor $v \oplus b$.

**Exercise 40**
Show how to construct edge-disjoint paths $v \rightarrow v \oplus b$ in the $d$-dimensional hypercube for each $b \in \{0, 1\}^d$. Thus all communications of the above implementation are congestion-free when routed on the hypercube.

Hence the total communication time for MPI_Alltoall is

$$T_{\text{Alltoall}} = o + L + (\max\{o, g\} + L) \cdot (p - 2) + o = T_{\text{Allgather}, 1},$$

if all messages have length $w$.

**Exercise 41**
Show that $T_{\text{Alltoall}} = O(p \cdot n)$, if all messages have length $n$.

**Exercise 42**
We consider a second implementation for the all-to-all personalized broadcast which also uses the $\log_2 p$-dimensional hypercube as its communication pattern. Initially processor $v$ concatenates all messages $M^v_u$ with $u_1 \neq v_1$ and sends the concatenation to processor $v_1 v_2 \cdots v_{\log_2 p}$. Processes continue to use bitfixing (along dimensions $2, \ldots, \log_2 p$) to eventually move all messages to their destination. In particular, in the $j$th iteration ($1 \leq j \leq \log_2 p$) processor $v$ concatenates all currently stored messages whenever their $j$th destination bit differs from $v_j$ and sends the concatenation to processor $v_1 v_{j-1} v_j v_{j+1} \cdots v_{\log_2 p}$.

(a) Show that each processor sends exactly one half of its messages at any time. It receives the same number of messages that is sent.

(b) Determine the total communication time of the new implementation.

**MPI_Reduce, MPI_Allreduce and MPI_Scan**

We implement MPI_Reduce with binomial trees of depth $\log_2 p$ as communication pattern. Its communication cost is comparable with MPI_Broadcast for short messages. To implement MPI_Allreduce we execute MPI_Reduce, followed by MPI_Bcast. Finally, the communication resources required for MPI_Scan are comparable to the resources of MPI_Allreduce, if we utilize our algorithm of Section 2.1.

Let us summarize. The basic operations MPI_Send and MPI_Receive are of course the cheapest functions. Then come MPI_Bcast and MPI_Reduce followed by MPI_Allreduce and MPI_Scan which require twice the resources. The operations MPI_Scatter, MPI_Gather and MPI_Allgater are significantly more expensive. The leader of the pack in resource consumption is not surprisingly MPI_Alltoall.

In the following we use asymptotic estimates of the communication time. The corresponding bounds are only rough estimates, since the asymptotic notation hides large constants. Hence, if the parallel machine is known, then the machine-specific performance of the respective MPI-functions should be used instead.
3.2 Work, Speedup and Efficiency

Assume that a parallel algorithm $\mathcal{P}$ solves an algorithmic problem $\mathcal{A}$. When should we be satisfied with its performance? Assume that our algorithm uses $p$ processors and runs in time $t_\mathcal{P}(n)$ for inputs of length $n$. Furthermore let us assume that any sequential algorithm for $\mathcal{A}$ runs in time at least $\Omega(t(n))$. Certainly we can simulate $\mathcal{P}$ by a sequential algorithm, which requires time $O(p)$ to simulate one parallel step of $\mathcal{P}$ and hence runs in time $O(p \cdot t_\mathcal{P}(n))$.\(^1\) But any sequential algorithm for $\mathcal{A}$ requires time $\Omega(t(n))$ and hence

$$p \cdot t_\mathcal{P}(n) = \Omega(t(n))$$

follows. Thus

$$\text{work}_\mathcal{P}(n) = p \cdot t_\mathcal{P}(n),$$

the work of $\mathcal{P}$ on inputs of size $n$, is an important performance parameter of $\mathcal{P}$. Observe that the performance of $\mathcal{P}$ is the better, the smaller $\text{work}_\mathcal{P}(n)$ is.

**Theorem 3.1** Assume that any sequential algorithm for $\mathcal{A}$ runs in time at least $\Omega(t(n))$ and let $\mathcal{P}$ be a parallel algorithm for $\mathcal{A}$. Then $\text{work}_\mathcal{P}(n) = \Omega(t(n))$ holds.

Thus we can evaluate $\mathcal{P}$, if we compare its work with the running time of an (asymptotic) optimal sequential algorithm for $\mathcal{A}$, respectively with the asymptotic running time of the best, currently known algorithm —provided such an algorithm exists.

From now on we assume that $\mathcal{S}$ is a sequential algorithm for $\mathcal{A}$ and that $\mathcal{P}$ is a parallelization of $\mathcal{S}$: any operation performed by $\mathcal{S}$ is also performed by some processor of $\mathcal{P}$. Let $t_\mathcal{S}(n)$ be the running time of $\mathcal{S}$. We define the **speedup** $S_\mathcal{P}(n)$ of $\mathcal{P}$ by

$$S_\mathcal{P}(n) = \frac{t_\mathcal{S}(n)}{t_\mathcal{P}(n)}$$

and the **efficiency** $E_\mathcal{P}(n)$ of $\mathcal{P}$ by

$$E_\mathcal{P}(n) = \frac{S_\mathcal{P}(n)}{p} = \frac{t_\mathcal{S}(n)}{\text{work}_\mathcal{P}(n)}.$$

Observe that the speedup $S_\mathcal{P}(n)$ is asymptotically bounded by $p$ and hence the efficiency $E_\mathcal{P}(n)$ is asymptotically at most one.

**Exercise 43**

You must clean the parking space of your car from snow. The parking space already contains $n$ cubic meters of snow. One person can remove snow at the rate of $x$ cubic meters per second. Unfortunately, snow is still falling at the rate of $y$ cubic meters per second ($y < x$). But you may

---

\(^1\)This statement is false, if the sequential algorithm has insufficient main memory and hence is forced to access external memory. Observe that the size of the main memory of the parallel system exceeds the size of the sequential system by the factor $p$. 


ask some of your friends to help. Let \( T(n, 1) \) be the time taken by one person to clean the parking space, and let \( T(n, p) \) be the time taken by \( p \) persons.

**Compute** the speedup \( s(p) \) and show that superlinear speedup is achieved, i.e. \( s(p) > p \). Why is this possible? Explain.

**Exercise 44**

In the \( n \)-body problem we are given the initial positions, masses, and velocities of \( n \) particles. We have to determine their subsequent motions as determined by classical mechanics.

We assume the following simplified setup. The force \( f_i \) acting on particle \( i \) is defined by

\[
 f_i = \sum_{j=1, j\neq i}^{n} F(x_i, x_j),
\]

where \( x_i \) describes the position, mass and velocity of particle \( i \). Design a parallel algorithm with \( p \leq n \) processors which determines all \( f_i \) with efficiency \( \Theta(1) \) and speedup \( \Omega(p) \). You may assume that any sequential algorithm runs in time \( \Omega(n^2) \).

Assume that we plan on solving the problem \( A \) with a parallel algorithm \( P \) which performs some fraction of its work sequentially. What is the best achievable speedup?

**Theorem 3.2** Let \( P \) be a parallel algorithm that uses \( p \) processors.

(a) **Amdahl’s Law:** Let \( f \in [0, 1] \) be given. If \( P \) performs \( f \cdot t_S(n) \) steps of \( S \) sequentially, then

\[
 S_P(n) \leq \frac{1}{f + (1 - f)/p}.
\]

In particular, the speedup is always bounded by \( 1/f \).

(b) **Gustafson-Barsis Law:** If \( P \) computes sequentially for \( f \cdot t_P(n) \) steps, then

\[
 S_P(n) \leq p - f \cdot (p - 1).
\]

**Proof:** (a) At best \( t_P(n) = f \cdot t_S(n) + (1 - f) \cdot t_S(n)/p \) and hence

\[
 S_P(n) = \frac{t_S(n)}{t_P(n)} \leq \frac{t_S(n)}{f \cdot t_S(n) + (1 - f) \cdot t_S(n)/p} = \frac{1}{f + (1 - f)/p}
\]

follows. For (b) we observe that at worst \( S \) runs for \( f \cdot t_P(n) + p \cdot (1 - f) \cdot t_P(n) \) steps and hence

\[
 S_P(n) = \frac{t_S(n)}{t_P(n)} \leq \frac{f \cdot t_P(n) + p \cdot (1 - f) \cdot t_P(n)}{t_P(n)} = f + p \cdot (1 - f) = p - f \cdot (p - 1)
\]

follows. \( \square \)

Observe however that the estimates of both laws for the best-case speedup are overly optimistic, since we do not account for the communication overhead.
3.2. WORK, SPEEDUP AND EFFICIENCY

3.2.1 Varying the Number of Processors

Can we scale down the parallel algorithm \( P \), that is can we come up with an equivalent parallel algorithm \( Q \) which runs on \( q \) \((q < p)\) processors and still at least roughly maintains the efficiency of \( P \)? Assume \( P \) performs a total of \( \text{op}_i(n) \) operations in step \( i \), where each individual operation is weighted by its execution time. Then the simulation of the \( i \)th step requires \( \lceil \frac{\text{op}_i(n)}{q} \rceil \) steps with \( q \) processors, provided the processors can solve the scheduling problem determine which processor is to perform which operation instantaneously. Hence, if the scheduling problem is solvable instantaneously, we get

\[
\sum_{i=1}^{\tau_P(n)} \left\lfloor \frac{\text{op}_i(n)}{q} \right\rfloor \leq \sum_{i=1}^{\tau_P(n)} \left( \frac{\text{op}_i(n)}{q} + 1 \right) \leq \frac{\text{work}_P(n)}{q} + \tau_P(n)
\]

as the new running time. Thus the efficiency of \( Q \) is at least

\[
\frac{E_Q(n)}{E_P(n)} = \frac{\text{work}_P(n)}{q \cdot \left( \frac{\text{work}_P(n)}{q} + \tau_P(n) \right)} = \frac{\text{work}_P(n)}{\text{work}_P(n) + q \cdot \tau_P(n)} = \frac{1}{1 + \frac{q \cdot \tau_P(n)}{\text{work}_P(n)}} = \frac{1}{1 + q/p}
\]

and at least efficiency does not decrease considerably.

**Theorem 3.3** Let \( S \) be a sequential algorithm and let \( P \) be a parallelization of \( S \).

(a) Assume that \( q < p \). If the scheduling problem is solvable instantaneously, then there is an equivalent parallel algorithm \( Q \), which runs in time \( O\left(\frac{\text{ts}(n)}{q} + \tau_P(n)\right) \) with \( q \) processors. Moreover \( E_Q(n) \geq \frac{1}{1+q/p} \cdot E_P(n) \) and the efficiency of \( Q \) is almost as high as the efficiency of \( P \).

(b) If we increase (resp. decrease) the number of processors, but keep input size fixed, then efficiency and speedup tend to decrease (resp. increase).

**Proof:** (b) If we increase the number of processors, then we may scale down again and at worst efficiency stays about the same. \( \square \)

**Exercise 45**

Determine the maximal efficiency when computing the Boolean OR with \( p = n \) respectively with \( p \leq \frac{n}{\log n} \) processors.

We have just seen that a parallel algorithm \( P \) can be scaled down and at least (almost) maintains its efficiency. What happens if we increase input size, but keep the number of processors \( p \) fixed? If \( N > n \), then we get

\[
\frac{E_P(N)}{E_P(n)} = \frac{t_S(N)}{t_S(n)} / \frac{t_P(N)}{t_P(n)}.
\]

Therefore we should expect efficiency to increase, since the sequential running time is faster growing than the parallel running time. We summarize the previous observations:
- If we keep input size fixed, but reduce the number of processors (i.e. if we scale down),
then efficiency tends to increase. If we however increase the number of processors
(i.e. if we scale up), then efficiency tends to decrease.
- efficiency tends to increase when we increase input size, but keep the number of
processors fixed.

Our goal is to design parallel algorithms with high efficiency and as many processors as
possible, since then we have also high efficiency for smaller numbers of processors. A good
parallel algorithm should scale well, in other words it should reach high efficiency already
for moderate input sizes.

**Definition 3.4** A parallel algorithm \( P \) is **scalable** with efficiency \( E \), if for every number
\( p \) of processors there is an input size \( N_p \) such that \( E_P(n) \geq E \) whenever \( n \geq N_p \). The
**isoefficiency function** \( f_E \) with respect to \( E \) assigns the smallest input size \( f_E(p) \) to \( p \in \mathbb{N} \) such that \( E_P(n) \geq E \) whenever \( n \geq f_E(p) \).

**Example 3.2** In Section 2.1 we have introduced a parallel algorithm \( P \) for the prefix
problem. If the operation \( * \) can be evaluated in constant time, then time \( t_P(n) = O\left(\frac{n}{p} + \log_2 p\right) \) suffices for \( p \) processors. Observe that this approach achieves work \( P(n) = \Theta(p \cdot \left(\frac{n}{p} \log_2 \frac{n}{p} + n\right)) = \Theta(n + p \log_2 p) \) and hence \( E_P(n) = \Theta\left(\frac{n}{n + p \log_2 p}\right) \). Thus, for \( E = \Theta(1) \), we get the
isoefficiency function \( f_E(p) = \Omega(p \cdot \log_2 p) \).

**Example 3.3** The Odd-Even Transposition Sort sorts \( n \) keys in time \( \Theta\left(\frac{n \log_2 \frac{n}{p}}{p} + n\right) \) with
\( p \) processors and hence work \( P(n) = \Theta(p \cdot \left(\frac{n}{p} \log_2 \frac{n}{p} + n\right)) = \Theta(n \log_2 \frac{n}{p} + p \cdot n) \) follows. As a
consequence \( E_P(n) = \frac{n \log_2 \frac{n}{p}}{n \log_2 \frac{n}{p} + p \cdot n} \) and we obtain constant efficiency iff \( p = O(\log_2 n) \). Thus,
for \( E = \Theta(1) \), we get the isoefficiency function \( f_E(p) = 2^{\Theta(p)} \).

**Example 3.4** In the exercises of Section 2.2 we have described a recursive algorithm which
sorts \( n = m^2 \) keys in time \( O(\sqrt{n}) \) with \( n \) processors. If we scale down and reduce the
number of processors to \( p \), then we should assign \( \frac{n}{p} \) keys per processor. We start by
sequentially sorting the keys for every processor. In the remaining \( O(\sqrt{p}) \) steps we perform
a merging step with running time \( \Theta\left(\frac{n}{p} \log_2 \frac{n}{p}\right) \) and hence the parallel running time \( t_P(n) \) is bounded by \( t_P(n) = \Theta\left(\frac{n}{p} \log_2 \frac{n}{p} + \sqrt{p} \cdot \frac{n}{p}\right) \). Thus work \( P(n) = \Theta(p \cdot \left(\frac{n}{p} \log_2 \frac{n}{p} + \frac{n}{\sqrt{p}}\right)) = \Theta(n \log_2 \frac{n}{p} + \sqrt{p} \cdot n) \) and as a consequence we achieve constant efficiency \( E = \Theta(1) \) for
\( p = O(\log^2 \sqrt{n}) \). In summary: we get the isoefficiency function \( f_E(p) = 2^{\Theta(\sqrt{p})} \).

Of particular interest is the isoefficiency function \( f_E \) for the largest efficiency \( E = \Theta(1) \).
We should prefer algorithms whose isoefficiency function \( f_E \) grows as slow as possible since
then the required input size for optimal efficiency is small. Thus the prefix algorithm has
the most satisfying performance among the three examples.
3.3 Partitioning and Mapping

The design process of a parallel algorithm can be broken into the partitioning and mapping phase. The algorithm has to find a good compromise between computation and communication. Moreover the solution has to be scalable, that is its isoefficiency function should grow as slow as possible.

- In the partitioning phase we decompose the data (domain decomposition) and/or a specification of the computation (functional decomposition). The domain decomposition partitions the data into fragments and associates primitive tasks (i.e., parts of the computations) with the fragments. The functional decomposition breaks the computation into primitive tasks and subsequently associates data fragments with the primitive tasks.

As a rule of thumb the number of primitive tasks should be orders of magnitude larger than the targeted number of processors. If moreover primitive tasks have roughly same size, then good load balancing may be possible.

- Mapping. We have to combine primitive tasks into $p$ larger tasks. The combination or mapping should preserve locality: tasks which communicate which each other, respectively tasks that depend on each other are good candidates for merging.

- Computation and Communication. Thus the partitioning phase has to define primitive tasks such that their communication pattern supports locality preserving mapping. In particular the resources required to communicate with other combined tasks should be dominated by the computing resources required for the combined task and moreover communication should be overlapped with computation whenever possible. If support for long messages is available, then replace many short messages by few (and correspondingly longer) messages and thereby save on the startup cost of communication.

- Scalability. The algorithm should be portable to systems with various numbers of processors. Its isoefficiency function should grow as slow as possible.

As an example we consider the prefix problem for operation $\ast$ and $n$ elements $x_1, \ldots, x_n$. We decide in domain decomposition to associate primitive tasks with each element $x_i$. To minimize communication it is advisable to map disjoint intervals of $n/p$ elements to the $p$ processors. Thus we are led to the algorithm of Section 2.1. Observe that the computation resources per processor far outweighs the communication resources per processor.
Chapter 4

Parallel Linear Algebra

4.1 Matrix-Vector Multiplication

We are given a matrix $A$ with $m$ rows and $n$ columns as well as a vector $x$ with $n$ components. Our goal is to compare the matrix-vector product $y = A \cdot x$ with

$$y_i = \sum_{j=1}^{n} A[i, j] \cdot x_j.$$  

(We assume that addition and multiplication is defined by some field $F$ such as the field of rationals, reals or the field of complex numbers.) The straightforward sequential algorithm computes $y_1, \ldots, y_m$ in consecutive steps. To compute $y_i$ we have to perform $n$ multiplications and $n-1$ additions. Thus overall $m \cdot n$ multiplications and $m \cdot (n-1)$ additions suffice.

We assume for the following that $p$ divides $m$ as well as $n$. To arrive at a good parallel algorithm with $p$ processes we begin with data decomposition and mapping. We consider the following options.

- Rowwise decomposition: each process is responsible for $m/p$ contiguous rows.
- Columnwise decomposition: each process is responsible for $n/p$ contiguous columns.
- Checkerboard decomposition: Assume that $k$ divides $m$ and that $l$ divides $n$. Imagine that the processes form a $k \times l$ mesh. Process $(i, j)$ obtains the submatrix of $A$ consisting of the $i$th row interval of length $m/k$ and the $j$th column interval of length $n/l$. We demand that all submatrices have size $\Theta(m \cdot n/p)$ and hence $k \cdot l = p$.

We assume the following input/output conventions: each process knows the submatrix of $A$ that it is responsible for; in particular we do not charge for redistributing $A$.\footnote{If we redistribute the matrix sequentially, then we may need time $\Omega(m \cdot n)$ and we could as well try to compute the matrix-vector product sequentially.} Process
whenever the matrix is bounded by $m/k$ length time $O$.

Thus the total communication time is $\text{block } A = O$ length!

In many applications it is important to quickly compute the matrix-vector product all three decompositions, if $O$ time $O$ the blocks of $x$ to the processors. Since this involves $p - 1$ messages of length $n/p$, time $O(n)$ is sufficient. Then process $i$ computes the matrix-vector product $y^i = A^i \cdot x^i$ for its block $A^i$ of columns and process 1 applies a Reduce operation to sum up $y^1, y^2, \ldots, y^p$. Thus the total communication time is $O(n + m \cdot \log_2 p)$ and the total running time is $O(m \cdot n/p + n + m \cdot \log_2 p)$. Hence it suffices to demand $m = \Omega(p)$ and $n = \Omega(p \cdot \log_2 p)$ to obtain constant efficiency.

For the checkerboard decomposition process 1 applies a Scatter operation to distribute the blocks of $x$ to “their” processes. Since this involves $p - 1$ messages of length $n/p$, time $O(n)$ is sufficient. Then process 1 gathers the remaining $k - 1$ vectors of length $m/k$ in time $O(m)$. If $\log_2 k \leq l$ and $\log_2 l \leq k$, then the total communication time is bounded by $O(n + m)$ and we obtain constant efficiency, if $n = \Omega(p)$ and $m = \Omega(p)$.

**Theorem 4.1** The algorithms based on the rowwise, columnwise and checkerboard decomposition all determine the matrix-vector product $A \cdot x$ for an $m \times n$ matrix $A$ in computing time $O(mn/p)$ with $p$ processors. Their communication times are bounded by $O(n \cdot \log_2 p + m)$, $O(n + m \cdot \log_2 p)$ and $O(n + m)$ — if $\log_2 k \leq l$ and $\log_2 l \leq k$— respectively.

Observe that the checkerboard decomposition has the smallest communication time among all three decompositions, if $m \approx n$. What is the reason? Its messages have the smallest length!

**Exercise 46**

Determine the isoefficiency function for all three decompositions, provided $m = \alpha \cdot n$ with $0 < \alpha$. As usual we require efficiency $E = \Theta(1)$.

In many applications it is important to quickly compute the matrix-vector product $A \cdot x$ whenever the matrix $A$ is sparse, i.e., whenever $A$ has only few nonzero entries.

**Exercise 47**

Let $A$ be an $m \times n$ matrix with $N$ nonzero coefficients. For which input representation can we obtain a parallel algorithm that computes $A \cdot x$ in time $O(N/p)$ with $p$ processors?
4.2 Matrix Multiplication

We are given the \( n \times n \) matrices \( A \) and \( B \) and have to compute the \( n \times n \) product matrix 
\[ C = A \cdot B. \]
\( C[i, j] \) is the inner product of row \( i \) of \( A \) and column \( j \) of \( B \) and hence 
\[ C[i, j] = \sum_{k=1}^{n} A[i, k] \cdot B[k, j]. \]
A sequential algorithm may compute \( C[i, j] \) with \( n \) multiplications and \( n - 1 \) additions and hence \( C \) can be computed sequentially with \( n^3 \) multiplications and \( n^2 \cdot (n - 1) \) additions. Altogether running time \( \Theta(n^3) \) is required.

We describe three parallel algorithms. Our first algorithm uses the rowwise decomposition: all rows of \( A \) and \( B \) with indices belonging to the interval \( \{(i-1) \cdot n/p + 1, \ldots, i \cdot n/p\} \) are assigned to process \( i \). Thus process \( i \) receives the submatrices \( A^i \) of \( A \) and \( B^i \) of \( B \). We further subdivide \( A^i \) and \( B^i \) into the square submatrices \( A^{i,1}, \ldots, A^{i,p} \) and \( B^{i,1}, \ldots, B^{i,p} \), where \( A^{i,j} \) and \( B^{i,j} \) consist of all columns of \( A^i \) and \( B^i \) whose indices belong to the interval \( \{(j-1) \cdot n/p + 1, \ldots, j \cdot n/p\} \). We define \( C^{i,j} \) analogously and observe that

\[ C^{i,j} = \sum_{k=1}^{p} A^{i,k} \cdot B^{k,j} \]

holds. We compute \( C^{i,j} \) in \( p \) phases. In phase 1 process \( i \) computes the products \( A^{i,i} \cdot B^{i,j} \) for \( j = 1, \ldots, p \). Time \( O(p \cdot \frac{n}{p} \cdot \frac{n}{p} \cdot \frac{n}{p}) = \frac{n^3}{p} \) suffices, since process \( i \) multiplies the \( \frac{n}{p} \times \frac{n}{p} \) matrix \( A^{i,i} \) with the \( \frac{p}{n} \times \frac{n}{p} \) matrices \( B^{i,j} \). Afterwards we use a cyclic linear array as communication pattern. In particular process \( i \) sends \( B^i \) to process \( j = (i+1) \mod p \) and hence receives \( B^i \). (We work with residues \( 1, \ldots, p \).) Observe that the communication time is proportional to \( \frac{n}{p} \cdot n = \frac{n^2}{p} \), the size of a matrix \( B^i \). We repeat these pairs of compute and communications steps: in the \( k \)th phase process \( i \) computes the products \( A^{i,i-k+1} \mod p \cdot B^{i-k+1} \mod p, j \) for \( j = 1, \ldots, p \), sends \( B^{i-k+1} \mod p, j \) and receives \( B^{i-k} \mod p \).

We have \( p \) phases with compute time \( \frac{n^3}{p} \) and communication time \( \frac{n^2}{p} \) per phase. Thus the running time is at most \( O(\frac{n^3}{p} + \frac{n^2}{p}) \) and we achieve constant efficiency iff \( n = \Omega(p) \). We can further obtain a speedup by overlapping the communication of the submatrix \( B^i \) with computation by initiating a non-blocking send/receive operation sufficiently early. Nevertheless we achieve a compute/communicate ratio of only \( O(\frac{n^3}{p^2} / \frac{n^2}{p}) = O(\frac{n}{p}) \) per phase.

The next two approaches use the checkerboard decomposition. Let \( A^{i,j}, B^{i,j} \) and \( C^{i,j} \) be the submatrix of \( A, B \) and \( C \) respectively, where we choose all rows with indices from the interval \( \{(i-1) \cdot n/\sqrt{p} + 1, \ldots, i \cdot n/\sqrt{p}\} \) and all columns with indices from the interval \( \{(j-1) \cdot n/\sqrt{p} + 1, \ldots, j \cdot n/\sqrt{p}\} \). We assume a \( \sqrt{p} \times \sqrt{p} \) mesh of processes in which process \((i, j)\) initially holds \( A^{i,j} \) and \( B^{i,j} \) and is required to finally hold \( C^{i,j} \). Observe that

\[ C^{i,j} = \sum_{k=1}^{\sqrt{p}} A^{i,k} \cdot B^{k,j} \]

holds. Fox's algorithm is composed of \( \sqrt{p} \) phases. In the first phase process \((i, i)\) broadcasts its submatrix \( A^{i,i} \) to all processes of row \( i \) and, subsequently, any process \((i, j)\) of
row \(i\) computes the matrix product \(A^{i,i} \cdot B^{i,j}\). Our goal for process \((i, j)\) in phase two is to compute the product \(A^{i,i+1 \mod \sqrt{p}} \cdot B^{i+1 \mod \sqrt{p},j}\) and hence the first phase should conclude with process \((i, j)\) sending \(B^{i,j}\) to its neighbor \((i - 1 \mod \sqrt{p}, j)\) on top and receiving \(B^{i+1 \mod \sqrt{p},j}\) from its neighbor \((i + 1 \mod \sqrt{p}, j)\) below. The general phase \(k\) has then the following form:

- Process \((i, i+k-1 \mod \sqrt{p})\) broadcasts its submatrix \(A^{i,i+k-1 \mod \sqrt{p}}\) to all processors of row \(i\).
- Any process \((i, j)\) computes the product \(A^{i,i+k-1 \mod \sqrt{p}} \cdot B^{i+k-1 \mod \sqrt{p},j}\) and sends \(B^{i+k-1 \mod \sqrt{p},j}\) to its neighbor on top and receives \(B^{i+k \mod \sqrt{p},j}\) from its neighbor below.

Our third approach, Cannon’s algorithm, follows the approach of Fox’s algorithm, but it replaces each broadcast by a simple send and hence is more efficient. In particular, the first step of Cannon’s algorithm redistributes the submatrices such that afterwards

process \((i, j)\) holds \(A^{i,i+j \mod \sqrt{p}}\) and \(B^{i+j \mod \sqrt{p},j}\). In its first phase

- process \((i, j)\) first multiplies \(A^{i,i+j \mod \sqrt{p}}\) and \(B^{i+j \mod \sqrt{p},j}\),
- passes \(A^{i,i+j \mod \sqrt{p}}\) to its left neighbor and \(B^{i+j \mod \sqrt{p},j}\) to its neighbor above and
- receives \(A^{i,i+j+1 \mod \sqrt{p}}\) from its right neighbor and \(B^{i+1+j \mod \sqrt{p},j}\) from its neighbor below.

In subsequent phases processes continue to compute the respective product, send their \(A\)-matrix to the left neighbor and their \(B\)-matrix to the neighbor above. After \(\sqrt{p}\) phases each process \((i, j)\) has computed its matrix \(C^{i,j}\) and, per phase, computing time \(O\left(\frac{n^3}{\sqrt{p}}\right)\) and communication time \(O\left(\frac{n^2}{p}\right)\) suffice. Hence the total running time is at most \(O\left(\frac{n^3}{\sqrt{p}} + \frac{n^2}{\sqrt{p}} + \sqrt{p}\right)\) and we achieve constant efficiency iff \(\sqrt{p} \cdot n^2 + p^{3/2} = O(n^3)\), respectively iff \(n = \Omega(\sqrt{p})\).

Moreover we can proceed as for the rowwise decomposition and overlap communication with computation. This time however we achieve the far better compute/communicate ratio of \(\Theta\left(\frac{n^3}{\sqrt{p}^3} / \frac{n^2}{p}\right) = \Theta\left(\frac{n^{3/2}}{\sqrt{p}}\right)\) and Cannon’s algorithm is the better the larger the number \(p\) is.

Finally we describe the algorithm of Dekel, Nassimi and Sahni, the **DNS algorithm**, which uses a **three dimensional decomposition**. We describe the DNS algorithm first for \(n^3\) processes and then show how to implement it with a given number \(p\) of processes. We imagine the \(n^3\) processes to be arranged in a three-dimensional mesh of processes. Initially process \((i, j, 1)\) stores \(A[i, j]\) and \(B[i, j]\) and is responsible to output \(C[i, j]\).
4.2. MATRIX MULTIPLICATION

In its first phase the DNS algorithm replicates data such that process \((i, j, k)\) holds \(A[i, k]\) and \(B[k, j]\). If this is achieved, then \((i, j, k)\) computes the product \(A[i, k] \cdot B[k, j]\). Finally process \((i, j, 1)\) applies a Reduce operation to compute \(\sum_{k=1}^{n} A[i, k] \cdot B[k, j]\). All steps with the exception of the replication steps can be readily implemented. To implement the replication step process \((i, j, 1)\) sends \(A[i, j]\) to process \((i, j, j)\) and \(B[i, j]\) to process \((i, j, i)\). Finally \((i, j, j)\) broadcasts \(A[i, j]\) to all processes \((i, *, j)\) and \((i, j, i)\) broadcasts \(B[i, j]\) to all processes \((*, j, i)\).

Thus the broadcast dominates during replication and hence replication takes time \(O(\log_2 n)\). The multiplication step runs in constant time and the Reduce operation runs in logarithmic time. Thus the DNS algorithm has running time \(O(\log_2 n)\) with \(n^3\) processes and its efficiency \(\Theta(1/\log_2 n)\) is rather small. However we can increase efficiency, if we scale down the number of processes to \(p\).

Set \(q = p^{1/3}\). We arrange the \(p\) processes in a three-dimensional mesh. Process \((i, j, 1)\) receives the submatrices \(A^{i,j}, B^{i,j}\) of \(A\) and \(B\) respectively, where \(A^{i,j}\) as well as \(B^{i,j}\) consists of all rows with indices in the interval \(\{(i-1) \cdot n/q + 1, \ldots, i \cdot n/q\}\) and of all columns with indices in the interval \(\{(j-1) \cdot n/q + 1, \ldots, j \cdot n/q\}\). We now mimic the approach with \(n^3\) processes by replacing entries \(A[i, j]\) and \(B[i, j]\) by submatrices \(A^{i,j}\) and \(B^{i,j}\). In the replication step the \(n/q \times n/q\) submatrices \(A^{i,j}\) and \(B^{i,j}\) have to be broadcast to all processes \((i, *, j)\) and \((*, j, i)\) respectively and hence the communication cost is bounded by \(O\left(\frac{n^2}{q} \log_2 q\right)\). The multiplication step is expensive and requires time \(O\left(\frac{n^3}{q}\right)\). Finally the Reduce operation involves \(\frac{n}{q} \times \frac{n}{q}\) submatrices and hence takes time \(O\left(\frac{n^2}{q^2} \cdot \log_2 q\right)\). Thus the running time of the DNS algorithm is bounded by \(O\left(\frac{n^3}{q^3} + \frac{n^2}{q^2} \log_2 q\right) = O\left(\frac{n^3}{p} + \frac{n^2}{p^{3/2}} \cdot \log_2 p\right)\). We achieve constant efficiency iff \(n^3 = \Omega\left(\frac{n^2}{p^{3/2}} \cdot \log_2 p\right)\) or equivalently iff \(n^3 = \Omega\left(p \cdot \log_2 p\right)\).

Finally the compute/communicate ratio is \(\Theta\left(\frac{n^3}{p} / \frac{n^2 \cdot \log_2 p}{p^{3/2}}\right) = \Theta\left(\frac{n}{p^{3/2} \cdot \log_2 p}\right)\) and the DNS algorithm is superior, provided the number of processors is sufficiently large.

**Theorem 4.2** The algorithm based on the rowwise decomposition, Cannon’s algorithm as well as the DNS algorithm all multiply two \(n \times n\) matrices in computing time \(O\left(\frac{n^3}{p} + n^2\right)\), \(O\left(\frac{n^3}{p} + \sqrt{p}\right)\) and \(O\left(\frac{n^3}{p} + \frac{n^2}{p^{3/2}} \cdot \log_2 p\right)\) respectively with \(p\) processors. Their communication times are \(O(n^2)\), \(O\left(\frac{n^2}{\sqrt{p}} + \sqrt{p}\right)\) and \(O\left(\frac{n^2}{p^{3/2}} \cdot \log_2 p\right)\) respectively.

**Remark 4.1** A main ingredient of all three algorithms is a sequential algorithm to multiply matrices. At first the multiplication problem seems trivial. However we have to exert some care. Assume that we compute \(A \cdot B\) sequentially with the three nested for-loops:

```plaintext
for i=1 to n do
  for j=1 to n do
    C[i,j] = 0;
  for k=1 to n do
```


Consider the outer \( i \)-loop and observe that performance is considerably improved, if we can keep the \( i \)th row of \( A \) and all of \( B \) within the cache. If \( B \) is too large, then we should instead use a checkerboard approach also sequentially, where block sizes should “just” fit into the cache.

Exercise 48
Assume that matrix multiplication of two \( n \times n \) matrices runs in time \( t_p(n) \) on \( p \) processors.

(a) Show how to compute the power \( A^k \) of an \( n \times n \) matrix \( A \) in time \( O(t_p(n) \cdot \log_2 k) \) with \( p \) processors.

(b) The sequence \( (x_n \mid n \in \mathbb{N}) \) is described by the recurrence
\[
x_m = a_1 \cdot x_{m-1} + a_2 \cdot x_{m-2} + \cdots + a_r \cdot x_{m-r}.
\]
Show how to determine \( x_n \) in time \( O(t_p(r) \cdot \log_2 n + r/p + \log_2 p) \) with \( p \) processors. Which efficiency is obtainable, if we assume that the sequential complexity of determining \( x_n \) is \( \Theta(n) \)?

Exercise 49
In matrix multiplication entries are computed according to the formula \( c_{i,j} = \sum_k a_{i,k} \cdot b_{k,j} \). In some applications similar expressions appear, but with addition and multiplication replaced by other operations.

(a) The transitive closure of a directed graph \( G = (V, E) \) is the directed graph \( G^\prime = (V, E^\prime) \), where we insert an edge \((i, j)\) into \( E^\prime \) whenever there is a path in \( G \) from \( i \) to \( j \). Show how to determine the transitive closure with the help of matrix multiplication.

Hint: Assume that \( G \) has \( n \) vertices and let \( A \) be the adjacency matrix of \( G \). Interpret the power \( A^{n-1} \) after replacing addition by \( \lor \) and multiplication by \( \land \).

(b) In the shortest path problem we are given a directed graph and non-negative weights on its edges. We have to determine the length of a shortest path for any pair of vertices. Reduce the shortest path problem to matrix multiplication.

(c) We are given \( p \) processors. Implement your solutions for (a) and (b), assuming that \( p \) processors are available. Determine running time and efficiency, if the sequential time complexity is \( \Theta(n^3) \) for graphs with \( n \) vertices.

Exercise 50
A \( n \times n \) matrix \( X \) is called banded with half-bandwidth \( b \), provided \( X[i,j] = 0 \) whenever \( |i-j| > b \).
Assume that the \( n \times n \) banded matrices \( A \) and \( B \) have half-bandwidth \( b_A \) and \( b_B \) respectively.

(a) Show that \( C = A \cdot B \) has half-bandwidth \( b_A + b_B \) and that \( C \) can be computed sequentially in time \( O(n \cdot (b_A + b_B)) \).

(b) Assume that the \( n \times n \) banded matrix \( X \) has half-bandwidth \( b_X \). We represent \( X \) by the matrix \( X^\prime \) with \( X'[i,j] = X[i, i - b_X + j] \), if \( 0 < i - b_X + j \leq n \), and \( X'[i,j] = 0 \) otherwise. Assume that \( A \) and \( B \) are presented by \( A' \) and \( B' \). Design a parallel algorithm to determine \( C' \) where \( C = A \cdot B \). We work with \( p \) processors. The matrices \( A' \) and \( B' \) are partitioned according to the checkerboard decomposition.
4.3 Solving Linear Systems

We are given a matrix $A$ and a right-hand side $b$ and would like to solve the linear system $A \cdot x = b$. We begin by investigating lower triangular matrices $A$ and describe back substitution in Section 4.3.1. When dealing with arbitrary matrices we have to apply Gaussian elimination and we describe efficient parallelizations in Section 4.3.2. We consider iterative methods (Jacobi relaxation, Gauss-Seidel algorithm, the conjugate gradient approach and the Newton method) in Section 4.3.3. These methods work very fast for important classes of matrices and are in particular capable of exploiting sparsity. As an application we show how to approximately solve linear partial differential equations.

4.3.1 Back Substitution

We assume that $A$ is a $n \times n$ invertible lower triangular matrix and we wish to solve the system $A \cdot x = b$ of linear equations. Since $A$ is a lower triangular matrix

$$A[i, 1] \cdot x_1 + \cdots + A[i, i] \cdot x_i = b_i$$

is the $i$th linear equation. A sequential solution is easy to obtain, if we first determine $x_1$ from the first equation. Assuming that we already know $x_1, \ldots, x_{i-1}$, we utilize the $i$th equation to determine $x_i$. Since an evaluation of the $i$th equation requires time $O(i)$, the sequential solution requires overall time $O(n^2)$.

Our goal is to solve $A \cdot x = b$ with $p \leq n$ processors in time $O(\frac{n^2}{p})$. Our first solution assumes the off-diagonal decomposition of matrix $A$: process 1 knows the main diagonal and process $i$ (for $i \geq 2$) knows the $i-1$st offdiagonal $A[i, 1], A[i+1, 2], \ldots, A[n, n-i+1]$. We use the linear array as communication pattern and start with the case of $p = n$ processes.

Process 1 successively determines $x_1, \ldots, x_n$ and passes $x_i$, immediately after computing it, to process 2. From that point on $x_i$ is sent thru the linear array. How can we solve the $i$th equation $A[i, 1] \cdot x_1 + \cdots + A[i, i] \cdot x_i = b_i$ as fast as possible? After receiving $x_1$, process $i$ computes $A[i, 1] \cdot x_1$, passes $A[i, 1] \cdot x_1$ to process $i-1$ and $x_1$ to process $i+1$. In the next step process $i-1$ receives $x_2$ from process $i-2$, computes the product $A[i, 2] \cdot x_2$, sends the sum $A[i, 1] \cdot x_1 + A[i, 2] \cdot x_2$ to process $i-2$ and $x_2$ to process $i$. Thus, following the principle of “just in time production”, process 1 obtains $A[i, 1] \cdot x_1 + A[i, 2] \cdot x_2 + \cdots + A[i, i-1] \cdot x_{i-1}$, determines $x_i$ and passes it on to process 2. The communication pattern when determining $x_1, x_2, x_3, x_4$ is shown in Figure 4.3. Now let $p$ be arbitrary. In the off-diagonal decomposition we assign the off-diagonals ($A[j, 1], \ldots, A[n, n-j+1]$) for $j \in \{(i-1) \cdot n/p + 1, \ldots, i \cdot n/p\}$ to process $i$. The computing time per unknown increases from $O(1)$ to $O(\frac{n^2}{p})$, whereas the communication time per unknown remains constant. We have $n$ unknowns and our running time is therefore bounded by $O(\frac{n^2}{p} + n)$. We achieve constant efficiency whenever $n = \Omega(p)$.

Now let us consider the rowwise decomposition. When determining $x_i$ from the $i$th equation we face the problem that processes 1, \ldots, $i-1$ do not know the $i$th row of $A$. But
process \( i \) can supply this information by sending \( A[i, j] \) to process \( i - j + 1 \) just in time. However computations do not dominate communications anymore and we therefore change our approach. Once \( x_j \) is determined, process \( j \) broadcasts \( x_j \) to processes \( j + 1, \ldots, n \). When working with an arbitrary number \( p \) of processes, we have one broadcast per unknown as well as computing time \( O(\frac{n^2}{p}) \). Now the running time is bounded by \( O(\frac{n^2}{p} + n \cdot \log_2 p) \) and we achieve constant efficiency whenever \( n = \Omega(p \cdot \log_2 p) \).

**Theorem 4.3** Let \( A \) be a \( n \times n \) invertible lower triangular matrix. Then the linear system \( A \cdot x = b \) can be solved in computing time \( O(\frac{n^2}{p}) \) for the off-diagonal as well as the rowwise decomposition. The off-diagonal decomposition has communication time \( O(n) \), whereas communication time \( O(n \cdot \log_2 p) \) suffices for the rowwise decomposition. \( A \) can be inverted in computing time \( O(\frac{n^3}{p}) \) and communication time \( O(n^2) \).

**Exercise 51**
Show how to invert a nonsingular lower triangular matrix \( A \) within the resources stated in Theorem 4.3.

**Exercise 52**
In the interleaved row decomposition of \( A \) process \( i \) receives all rows with indices in the set \( \{j \cdot p + \)}
4.3. SOLVING LINEAR SYSTEMS

Why is the interleaved row decomposition a better choice than the row decomposition?

4.3.2 Gaussian Elimination

Again let $A$ be an $n \times n$ matrix and let $b$ be vector with $n$ coordinates. Our goal is to solve the system $A \cdot x = b$ with Gaussian elimination. What is the sequential performance of Gaussian elimination? If we start with row 1 as pivot row, then we can eliminate the nonzeroes below $A[1, 1]$ in column 1 by setting

$$row_j = row_j - \frac{A[j, 1]}{A[1, 1]} \cdot row_1$$

for $j > 1$. In general, if we have already eliminated nonzeroes below the diagonals in columns 1, \ldots, $i-1$, then we eliminate nonzeroes below $A[i, i]$ in column $i$ by setting

$$row_j = row_j - \frac{A[j, i]}{A[i, i]} \cdot row_i$$

for $j > i$. The elimination step for row $i$ involves $n - i$ other rows $j$. Since any such row $j$ requires $2 \cdot (n - i + 1)$ arithmetic operations, the elimination step for row $i$ involves at most $2 \cdot (n - i + 1)^2$ arithmetic operations. Thus the total number of arithmetic operations is bounded by

$$\sum_{i=1}^{n-1} 2 \cdot (n - i + 1)^2 = O(n^3).$$

However our procedure is not numerically stable, if the pivot $A[i, i]$ is very small: $\frac{A[j, i]}{A[i, i]}$ can grow correspondingly large and significant roundoff errors result. Hence we apply Gaussian elimination with partial pivoting and use as pivot the largest entry in column $i$ among rows $i, i+1, \ldots, n$. If the largest entry sits in row $j$, then we swap rows $i$ and $j$ and compute as before.

Finally all elimination steps have to be simultaneously performed for the right-hand side $b$. We can take care of $b$ by including $b$ as the $n+1$st column of $A$. Thus we replace $A$ by the augmented matrix $A'$.

We describe a parallelization of Gaussian elimination with partial pivoting and assume a rowwise decomposition of the augmented matrix $A'$. As usual we work with $p$ processes, where process $i$ knows all rows with indices in the interval $\{(i-1) \cdot n/p + 1, \ldots, i \cdot n/p\}$. Our approach is to keep the sequential structure of pivoting steps and to parallelize each pivoting step instead. Assume that we have reached row $i$.

How can we determine the largest entry in column $i$ among rows $i, \ldots, n$? Each process $i$ begins by determining the largest entry $M_i$ among the rows it knows. As a first option we could apply MPI_Allreduce with Max as operation and values $M_1, \ldots, M_p$. Afterwards each
process knows the value $M$ of the largest entry. If $M_i = M$, then process $i$ submits $i$ and otherwise submits zero to a second MPI_Allreduce with Max as operation. Thus, after two Allreduce operations the pivot is determined. Fortunately MPI supplies the by far more appropriate operation MPI_Maxloc which determines for a set of pairs $(M_1, 1), \ldots, (M_p, p)$ a largest $M_i$ and returns $(M_i, i)$. Determining all local winners $M_1, \ldots, M_p$ takes computing time $O(n/p)$. The call of MPI_Maxloc requires communication time $O(\log_2 p)$.

Now that the pivot is known, the winning process broadcasts the relevant portion of the winning row to the remaining processes and does so in time $O(n \cdot \log_2 p)$. All processes can now simultaneously perform their respective elimination steps in time at most $O(n^2)$. Over all $n$ pivoting steps computing time $O(n^3)$ and communication time $O(n^2 \log_2 p)$ suffice. Thus we reach constant efficiency, whenever $n = \Omega(p \cdot \log_2 p)$ holds.

The weak spot of our approach is the broadcast which dominates the communication costs with communication time $O(n \log_2 p)$. Moreover it brings the entire computation to a screeching halt and does not allow to overlap communication with computation. Thus we change our pivoting strategy, select the largest entry not in column $i$, but in row $i$ instead. If process $j$ knows row $i$, it computes the largest entry $A[i, k]$ of row $i$ and all non-zeroes in row $i$ are to be eliminated next. We use column $k$ as pivot column and hence process $j$ determines the vector $m_i$ of coefficients required to eliminate all non-zeroes in row $i$.

However, instead of broadcasting $m_i$ and pivot position $k$, process $j$ only sends $m_i$ and $k$ to its neighbor process $j + 1$. Whereas process $j$ is now free to compute, process $j + 1$ has to immediately propagate the pivoting information to its neighbor process $j + 2$, but can directly afterwards begin its part of the elimination work. Thus we replace the broadcast of $(m_i, k)$ by pipelining and “cover” communication with computation.

But now distant processes are delayed and pipelining seems to slow down the computation. Observe however that the pipelined communication can start right after $m_i$ is recomputed. Thus, when performing the elimination step for pivot row $i$, the process holding row $i + 1$ recomputes row $i + 1$ as its first task, determines the maximal entry and the corresponding vector $r_{i+1}$ of coefficients and sends the new pivot information before returning to its elimination work for the current pivot. No delay is incurred when working with pivot row $i + 1$, provided the compute time $\Theta(n^2)$ per pivoting steps dominates the maximal delay $p \cdot n$. Thus, if $n = \Omega(p^2)$ and hence if $n$ is sufficiently large, pipelining is delay-free except for the first row.

Theorem 4.4 The pipelined algorithm works with the row decomposition of the augmented matrix $A'$. If $n = \Omega(p^2)$, then it performs Gaussian elimination with partial pivoting in computing time $O(n^3)$ using $p$ processors.
4.3.3 Iterative Methods

In an iterative method an approximate solution of a linear system $A \cdot x = b$ is successively improved. One starts with an initial “guess” $x(0)$ and then replaces $x(t)$ by a presumably better solution $x(t+1)$. Iterative methods are particularly well-suited for parallel algorithms whenever they are based on matrix-vector products. As a consequence this type of iterative methods can exploit sparse linear systems. We describe the Jacobi relaxation and its variants as well as the conjugate gradient method for solving linear systems. Finally we present the Newton method to approximately compute the inverse of a matrix.

**Jacobi Relaxation.**

We assume again that a linear system $A \cdot x = b$ with nonzero diagonal is to be solved. A solution $x$ satisfies

$$x_i = \frac{1}{A[i,i]} \cdot \left( b_i - \sum_{j \neq i} A[i,j] \cdot x_j \right).$$

Hence, if we assume that we already know an approximate solution $x_i(t)$, then the Jacobi iteration

$$(4.1) \quad x_i(t+1) = \frac{1}{A[i,i]} \cdot \left( b_i - \sum_{j \neq i} A[i,j] \cdot x_j(t) \right)$$

“could be” a better approximation and this is indeed the case, provided the matrices $M_t$, with the iteration matrix $M = D^{-1} \cdot (D - A)$, converge to zero with $t$ approaching infinity. ($D$ is the diagonal matrix with $D[i,i] = A[i,i].$)

**Exercise 53**

Assume that the matrix $A$ is invertible and let $x$ be the unique solution of the linear system $A \cdot x = b$. Show that $x(t+1) - x = M \cdot (x(t) - x)$ holds and consequently $x(t) - x = M^t \cdot (x(0) - x)$ follows for all $t$.

Each Jacobi iteration (4.1) corresponds to a matrix-vector product which we already know how to parallelize in time $O(n^2/p)$. Thus we obtain an extremely fast approximation, whenever few iterations suffice. Few iterations suffice for instance for the important class of diagonally dominant matrices:

**Exercise 54**

(a) Assume that $A$ is invertible and that the vectors $x(t)$ converge to a vector $x^*$. Show that $A \cdot x^* = b$ holds.

(b) Assume that the matrix $A$ is row diagonally dominant, i.e., $|A[i,i]| > \sum_{j \neq i} |A[i,j]|$ holds for all $i$. Show that the Jacobi Relaxation converges. What can you say about the rate of convergence?
In many practical applications the **Jacobi overrelaxation** converges faster: for a suitable choice of the coefficient $\gamma$ we apply the iteration

$$x_i(t + 1) = (1 - \gamma) \cdot x_i(t) + \frac{\gamma}{A[i,i]} \cdot \left( b_i - \sum_{j \neq i} A[i,j] \cdot x_j(t) \right).$$

Observe that we obtain the Jacobi relaxation (4.1) as a special case by setting $\gamma = 1$.

Another extension of the Jacobi relaxation is the **Gauss-Seidel algorithm** which incorporates already recomputed values of $x_j$ (i.e., it replaces $x_j(t)$ by $x_j(t + 1)$). An example is the rule

$$x_i(t + 1) = \frac{1}{A[i,i]} \cdot \left( b_i - \sum_{j < i} A[i,j] \cdot x_j(t + 1) - \sum_{j > i} A[i,j] \cdot x_j(t) \right).$$

(4.2)

Observe however that the Gauss-Seidel algorithm looks inherently sequential. Finally we can also apply overrelaxation to the Gauss-Seidel algorithm and obtain one of the fastest iterative methods.

**The Conjugate Gradient Method.**

We have to require that $A$ is symmetric and positive definite\(^2\). We describe the conjugate gradient method which turns out to be faster than the Jacobi Relaxation in many cases.

**Exercise 55**

Assume that $A = B^T \cdot B$ for some invertible matrix $B$.

(a) Show that $x^T Ax \geq 0$ for all $x$ and $x^T Ax = 0 \iff x = 0$.

(b) Let $z$ be an arbitrary vector. Show that $q_z(x) = \frac{1}{2} (x - z)^T A (x - z)$ is a convex function. Hint: show first that $\frac{1}{2} x^T Ax$ is convex. Then utilize that $r \cdot f(x + z)$ is a convex function of $x$, whenever $r \geq 0$ and $z$ is arbitrary.

(c) Show that $\nabla q_z(x) = Ax - Az$, where $\nabla q_z$ is the gradient of $q_z$.

(d) Set $z = x^*$ where $A \cdot x^* = b$. Then $\nabla q_z(x) = Ax - b$ and $q_z$ has a unique minimum for $x = x^*$.

Thus it suffices to minimize $q_z$. An iteration of the conjugate gradient method has the form

$$x(t + 1) = x(t) + \gamma(t) \cdot \text{direction}(t),$$

where the vector direction$(t)$ is the direction of update and the scalar step size $\gamma(t)$ is determined by

$$q_z (x(t) + \gamma(t) \cdot \text{direction}(t)) = \min_{\gamma} q_z (x(t) + \gamma \cdot \text{direction}(t)).$$

---

\(^2\)A is symmetric and positive definite iff $A = B^T \cdot B$ for some invertible matrix $B$. 

4.3. SOLVING LINEAR SYSTEMS

The choice of direction is the novel feature of this method, since the vectors \( \text{direction}(t) \) are chosen to be “\( A \)-conjugate”, namely

\[
\text{direction}(r)^T \cdot A \cdot \text{direction}(t) = 0
\]

holds, whenever \( r \neq t \).

Exercise 56
Assume that \( \text{direction}(0), \ldots, \text{direction}(n-1) \) are indeed \( A \)-conjugates.

(a) Show that the vectors \( \text{direction}(0), \ldots, \text{direction}(n-1) \) are linearly independent.

(b) Use (4.3) to show that \( \nabla q_z(x(k+1))^T \cdot \text{direction}(k) = 0 \) holds. Then infer \( \nabla q_z(x(k+1))^T \cdot \text{direction}(i) = 0 \) for all \( i \) \((0 \leq i \leq k)\).

(c) \( x(k+1) \) minimizes \( q_z \) when we restrict \( q \) to \( x(0) + V_k \), where \( V_k \) is the vector space spanned by \( \text{direction}(0), \ldots, \text{direction}(k) \).

As a consequence \( x(n) \) minimizes \( q_z \) and hence all we have to do is to guarantee that the vectors \( \text{direction}(t) \) are \( A \)-conjugates. We choose \( x(0) \) arbitrarily and set \( \text{direction}(0) = -(A \cdot x_0 - b) \). Since \( \nabla q_z(x_0) = A \cdot x_0 - b \), we initially search in the direction of the negative gradient. For \( t > 0 \) we set

\[
\begin{align*}
\text{direction}'(t) &= A \cdot x(t-1) - b \quad \text{and} \\
\text{direction}(t) &= -\text{direction}'(t) + \frac{||\text{direction}'(t)||^2}{||\text{direction}'(t-1)||^2} \cdot \text{direction}(t-1).
\end{align*}
\]

Finally we choose

\[
\gamma(t) = \frac{\text{direction}(t)^T \cdot \text{direction}'(t)}{\text{direction}(t)^T \cdot A \cdot \text{direction}(t)}
\]

as step size.

Exercise 57
Show that the vectors \( \text{direction}(t) \) are \( A \)-conjugates.

Our parallel algorithm computes the sequence \( x(t) \) and utilizes that the matrix-vector products \( A \cdot \text{direction}(t) \) can be computed in time \( O(n^2 p) \) with \( p \) processes.

Inversion by Newton Iteration.

We assume that \( A \) is an invertible matrix and that \( X_t \) is an approximate inverse of \( A \). The Newton iteration

\[
X_{t+1} = 2 \cdot X_t - X_t \cdot A \cdot X_t
\]

determines for many important classes of matrices a better approximation of \( A^{-1} \). To see why consider the residual matrix

\[
R_t = I - A \cdot X_t
\]
which measures the “distance” between \( A^{-1} \) and \( X_t \). We observe

\[
R_{t+1} = I - A \cdot X_{t+1} \\
= I - A \cdot (2 \cdot X_t - X_t \cdot A \cdot X_t) \\
= (I - A \cdot X_t)^2 = R_t^2
\]

and \( R_t \) converges rapidly towards the 0-matrix, whenever \( X_0 \) is a good approximation of \( A^{-1} \). For well conditioned matrices such a good initial approximation is obtained with\(^3\)

\[
X_0 = \frac{A^T}{\text{trace}(A^T \cdot A)}.
\]

Since the Newton iteration is based on matrix–matrix products, each iteration is easily parallelized. In [BT89] it is shown that \( O(\log_2 n) \) iterations suffice for a large class of matrices.

### 4.3.4 Finite Difference Methods

We describe the finite difference method as a method to approximately solve partial differential equations (PDEs).\(^4\) We restrict ourselves to the Poisson equation \( u_{xx} + u_{yy} = H \), but our approach can also be applied to the large class of linear PDEs. (A linear second-order PDE has the form

\[
A \cdot u_{xx} + 2B \cdot u_{xy} + C \cdot u_{yy} + E \cdot u_x + F \cdot u_y + G \cdot u = H,
\]

with functions \( A, \ldots, H \) depending on \( x \) and \( y \) only. Observe that neither \( u \) nor any of its partial derivatives is allowed to appear as a coefficient. Linear PDEs of higher degree are defined analogously.) We will see that the finite difference method leads to a sparse linear system, which is solvable with the help of the iterative methods of Section 4.3.3.

Our goal is to find a function \( u : [0,1]^2 \to \mathbb{R} \) which satisfies the Poisson equation and which has prescribed values on the boundary of the unit square. If the solution \( u \) is sufficiently smooth and if \( \Delta \) is sufficiently small, then we may approximate \( u_x(x, y) \) by

\[
u_x(x, y) \approx \frac{u(x + \Delta, y) - u(x, y)}{\Delta}
\]

and \( u_{xx}(x, y) \) by

\[
u_{xx}(x, y) \approx \frac{u(x+\Delta, y) - u(x, y)}{\Delta} \cdot \frac{u(x,y) - u(x-\Delta, y)}{\Delta} = \frac{u(x + \Delta, y) - 2u(x, y) + u(x - \Delta, y)}{\Delta^2}.
\]

\(^3\)The trace of a square \( n \times n \) matrix \( M \) is defined by \( \text{trace}(M) = \sum_{i=1}^{n} M[i, i] \).

\(^4\)We refer to the finite element method as another successful technique to solve PDEs numerically.
4.3. SOLVING LINEAR SYSTEMS

We approximate \( u_{yy} \) analogously and hence we get

\[
\frac{u(x + \Delta, y) + u(x - \Delta, y) + u(x, y + \Delta) + u(x, y - \Delta) - 4u(x, y)}{\Delta^2} \approx H(x, y)
\]

as a consequence of the Poisson equation. To apply the finite difference method we choose a sufficiently large integer \( N \) with \( \Delta = \frac{1}{N} \) and set

\[
u_{i,j} = u\left(\frac{i}{N}, \frac{j}{N}\right) \quad \text{as well as} \quad H_{i,j} = H\left(\frac{i}{N}, \frac{j}{N}\right).
\]

If we replace approximate equality in (4.4) by exact equality and choose \((x, y)\) be one of the grid points \( \{(\frac{i}{N}, \frac{j}{N}) \mid 0 < i, j < N\} \), then we obtain the linear system

\[-4u_{i,j} + u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1} = \frac{H_{i,j}}{N^2} \]

with \((N - 1)^2\) equations in \((N - 1)^2\) unknowns. (Remember that the values of \( u \) at the boundary are prescribed.) The system is extremely large, since it has \((N - 1)^4\) entries. However it is sparse, since any equation has at most five nonzero coefficients. We utilize sparsity by processing the system beginning with the lower boundary and working upwards. In other words we utilize the presentation

\[
u_{i+1,j} = 4u_{i,j} - (u_{i-1,j} + u_{i,j+1} + u_{i,j-1}) + \frac{H_{i,j}}{N^2}.
\]

After starting from an initial solution \( u_{i,j}(0) \), we can solve this system with the Jacobi relaxation

\[
u_{i+1,j}(t + 1) = 4u_{i,j}(t) - (u_{i-1,j}(t) + u_{i,j+1}(t) + u_{i,j-1}(t)) + \frac{H_{i,j}}{N^2},
\]

the Gauss-Seidel algorithm or apply an overrelaxation. We utilize the sparsity of the matrix \( A \) of the system and observe that one iteration runs in sequential time \( O(N^2) \) for either method. (Remember that \( A \) is a \((N - 1)^2 \times (N - 1)^2\) matrix and arbitrary matrices of this dimension require up to \( \Theta(N^4) \) operations!)

How can we obtain an efficient parallelization of one iteration? We assume a checkerboard decomposition of \( A \) in the following sense. We partition the \( N - 1 \) rows of the grid \( \{\frac{1}{N}, \ldots, \frac{N-1}{N}\}^2 \) into \( \sqrt{p} \) intervals \( I_1, I_2, \ldots \) as well as the \( N - 1 \) columns into \( \sqrt{p} \) intervals \( J_1, J_2, \ldots \). We imagine the \( p \) processes to be arranged in a \( \sqrt{p} \times \sqrt{p} \) mesh and assign the vector \( (u_{i,j} \mid (i, j) \in I_r \times J_s) \) to process \((r, s)\). The process responsible for determining \( u_{i+1,j}(t + 1) \) has to know \( u_{i-1,j}(t), u_{i,j+1}(t), u_{i,j-1}(t) \) and \( u_{i,j}(t) \). It can perform the update without communication unless one or more of the required values belongs to one of its three neighbors –the neighbor to the left, right or the neighbor on top–. In either case the missing values belong to the neighbor’s boundary with the exception of \( u_{i-1,j}(t) \) which may have distance one from the boundary. Thus after each process sends the \( O\left(\frac{N}{\sqrt{p}}\right) \) values
to its left, right and upper neighbor and receives values in return, it can recompute its values without communication. The computation time is bounded by $O\left(\frac{N^2}{p}\right)$, since each recomputation only involves $O(1)$ values and $\frac{N^2}{p}$ values have to be recomputed, and the communication time is bounded by $O\left(\frac{N}{\sqrt{p}}\right)$. Therefore running time $O\left(\frac{N^2}{p} + \frac{N}{\sqrt{p}}\right)$ suffices and we achieve constant efficiency whenever $N = \Omega\left(\sqrt{p}\right)$.

**Theorem 4.5** Assume that $p$ processors are available to approximate a solution of the Poisson equation. For one iteration of the Jacobi with or without overrelaxation computing time $O\left(\frac{N^2}{p}\right)$ and communication time $O\left(\frac{N}{\sqrt{p}}\right)$ suffices.

**Exercise 58** Assume a column decomposition instead of a checkerboard decomposition. Analyze the computation and communication time.

**Remark 4.2** It can be shown that the matrix of the linear system (4.5) is symmetric and positive definite [BT89]. As a consequence one can show that the Jacoby algorithm, the Gauss-Seidel algorithm as well as the conjugate gradient method converge. Moreover the overrelaxation variants converge as well whenever $\gamma \in [0, 2]$.

**Remark 4.3** Trading communications against computations.
Assume that communication costs dominate computation costs and assume that slightly longer messages result in almost unchanged communication costs. To utilize these assumptions we let neighbor processes exchange more values, namely all values of grid points within distance at most $k$ from the respective boundary. The receiving process duplicates the computation of its neighbor on the received values and thus, at the expense of redundant computations, the next $k - 1$ iterations can run without communication.

We already mentioned that the Gauss-Seidel algorithm converges, but the update rule (4.2) is in general inherently sequential. In our application however we obtain an efficient parallelization if we proceed as follows. Label the grid point $(i, j)$ white iff $i + j$ is even and black otherwise. Instead of rule (4.6) we use the update

$$u_{i,j}(t+1) = \frac{u_{i+1,j}(t) + u_{i-1,j}(t) + u_{i,j+1}(t) + u_{i,j-1}(t)}{4} - \frac{H_{i,j}}{4N^2}$$

and in this perspective white grid points are updated with the help of black grid points only. Thus we may execute one iteration of the Gauss-Seidel algorithm by first updating black grid points conventionally with the Jacobi relaxation (4.7). Then we apply the Gauss-Seidel algorithm to update white grid points for iteration $t+1$ by using the already updated black grid points.

Observe that we may need black grid points of iteration $t+1$ which belong to neighbors. If neighbors have communicated the values of grid points within distance at least two, then the process responsible for $(i, j)$ can perform the update of white grid points without communication. Thus redundant computations help in this particular case to reduce communications as well as to implement the Gauss-Seidel algorithm.
Exercise 59
We consider the linear second-order PDE $A \cdot u_{xx} + 2B \cdot u_{xy} + C \cdot u_{yy} + E \cdot u_x + F \cdot u_y + G \cdot u = H$ which is to be solved on the square $[0,1]^2$. We demand the boundary conditions $u(0,y) = K_1(y)$, $u(1,y) = K_2(y)$, $u(x,0) = K_3(x)$ and $u_y(x,0) = K_4(x)$. Compare the finite difference method for this class of PDEs to the finite difference method for the Poisson equation. Which input decomposition do you choose?

4.4 The Discrete Fourier Transform

Let $F$ be a field. We say that $\omega_n \in F$ is an \textit{n}th root of unity iff $\omega_n^n = 1$. If moreover $|\{\omega_n, \omega_n^2, \ldots, \omega_n^{n-1}, \omega_n^n\}| = n$ and $\omega_n^n = 1$, then we say that $\omega_n$ is a \textit{primitive} \textit{n}th root of unity.

Example 4.1 We consider the field $F = \mathbb{Z}_5$ of all residues modulo 5. Then 2 is a primitive fourth root of unity, since $2^2 \equiv 4 \mod 5$, $2^3 \equiv 3 \mod 5$ and $2^4 \equiv 1 \mod 5$.

Example 4.2 The field of real numbers has only $\pm 1$ as roots of unity, since they are the only real roots of the polynomial equation $x^n = 1$. Why? If $x^n = 1$, then the absolute value of $x$ must be one! But $x^n = 1$ has $n$ roots over the complex numbers. We represent a complex number $z$ by its polar coordinates $|z|$ and angle $\phi$ with $z = |z| \cdot (\cos 2\pi \phi + i \cdot \sin 2\pi \phi) = |z| \cdot e^{2\pi i \cdot \phi}$.

Hence, when multiplying complex numbers $z_1$ and $z_2$, we have to multiply lengths and add angles:

$$z_1 \cdot z_2 = |z_1| \cdot |z_2| \cdot e^{2\pi i \cdot (\phi_1 + \phi_2)}.$$

Define

\begin{equation}
\omega_n = e^{2\pi i / n}
\end{equation}

and we obtain $\omega_n^k = e^{2\pi i \cdot k / n}$: powers of $\omega_n$ are numbers of length 1 and their angles increase by multiples of $\frac{2\pi}{n}$. Since $\omega_n^n = 1$ and since all powers $\omega_n^k$ ($1 \leq k \leq n$) have distinct angles, we have shown that $\omega_n$ is an \textit{n}th root of unity.

Exercise 60
Let $w$ be an \textit{n}th root of unity.

(a) Show that $\sum_{k=0}^{n-1} w^k = 0$ holds.

(b) Assume that $n$ is even. Show that $w^{n/2} = -1$ holds, provided $w$ is a primitive root of unity.
Figure 4.2: Primitive 8-th root of unity $\omega_8 = e^{\frac{2\pi i}{8}}$ and its powers.

Let $\omega_n$ be a primitive $n$th root of unity over some fixed field $F$. We observe that each power $\omega_n^i$ is a root of $x^n = 1$ (and hence deserves the name “root of unity”), since $(\omega_n^i)^n = (\omega_n^n)^i = 1$. Thus, since polynomials of degree $n$ have at most $n$ different roots, the powers $\omega_n, \omega_n^2, \ldots, \omega_n^n$ are all the $n$ different roots of $x^n = 1$.

**Definition 4.6 The Discrete Fourier Transform**

Let $F$ be a field. The discrete Fourier transform of a vector $\vec{x} \in F^n$ is the matrix-vector product

$$\text{DFT}_n \cdot \vec{x} = \left( \sum_{k=0}^{n-1} \omega_n^{jk} \cdot x_k \mid 0 \leq j < n \right) = \left( \sum_{k=0}^{n-1} x_k (\omega_n^j)^k \mid 0 \leq j < n \right).$$

$\text{DFT}_n = (\omega_n^{jk})_{0 \leq j, k < n}$ is the matrix of the discrete Fourier transform.

Thus the discrete Fourier Transform evaluates the degree $n - 1$ polynomial with coefficients $(x_0, \ldots, x_{n-1})$ at all roots of unity.

Why would we want to compute the DFT matrix-vector product? The continuous Fourier transform decomposes a signal $x(t)$ into its spectrum of frequency components

$$X(w) = \frac{1}{\sqrt{2\pi}} \cdot \int_{-\infty}^{\infty} e^{-iwt} \cdot x(t) dt.$$ 

The discrete Fourier transform for the complex root of unity $\omega_n^{-1} = e^{-2\pi i/n}$ is applied to a discrete sample of the continuous signal and it approximates the continuous transform if $n$ is sufficiently large, i.e., if the time intervals used for sampling are sufficiently small. Thus for instance in lossy data compression, a discrete signal is decomposed into its frequency
components and high frequency components are dropped. In decompression the inverse transform is applied to the modified frequency components.

Besides the many applications in signal processing, there are further applications in solving differential equations, multiplying integers and polynomials to name only a few. All these applications are based on the Fast Fourier Transform, namely fast algorithms computing the discrete Fourier Transform.

We begin by showing how to multiply polynomials with the help of the discrete Fourier transform. The crucial observation is that the inverse transform is the original transform in disguise, and hence can be computed fast. Moreover the inverse transform can be interpreted as interpolating a polynomial at the roots of unity.

**Lemma 4.7** The inverse transform

(a) $DFT_n^{-1} = \left( \frac{\omega_n^{-j,k}}{n} \right)_{0 \leq j,k < n}$.

(b) $\vec{y} = DFT_n \cdot \vec{x}$ is the result after evaluating the polynomial with coefficients $x$ at all $n$th roots of unity.

(c) $\vec{x} = DFT_n^{-1} \cdot \vec{y}$ is the result of interpolating given the values at the $n$th roots of unity.

**Proof (a):** We multiply the $j$th row of $DFT_n$ with the $k$th column of $DFT_n^{-1}$ and obtain the inner product

$$\sum_{l=0}^{n-1} \omega_n^{j,l} \cdot \frac{\omega_n^{-l,k}}{n} = \frac{1}{n} \sum_{l=0}^{n-1} \omega_n^{(j-k)l}.$$  

If $j = k$, then the inner product is the same as adding $\frac{1}{n} \cdot n$ times and we obtain the value one as required. If $j \neq k$, then we observe the identity $1 + u + \ldots + u^{n-1} = (u^n - 1)/(u - 1)$ and can hence conclude

$$\frac{1}{n} \sum_{l=0}^{n-1} \omega_n^{(j-k)l} = \frac{1}{n} \cdot \frac{\omega_n^{n(j-k)} - 1}{\omega_n^{j-k} - 1} = \frac{1}{n} \cdot \frac{(\omega_n^{n})^{(j-k)} - 1}{\omega_n^{j-k} - 1} = 0.$$

**(b)** We already observed that $\vec{y} = DFT_n \cdot \vec{x}$ is the result after evaluating the polynomial with coefficients $x$ at all roots of unity. To show (c) we utilize that $DFT_n^{-1}$ is the inverse of $DFT_n$. Hence

$$DFT_n^{-1} \cdot \vec{y} = DFT_n^{-1} \cdot DFT_n \cdot \vec{x} = \vec{x}.$$  

Thus $\vec{x} = DFT_n^{-1} \cdot \vec{y}$ is the result of interpolating given the values at the roots of unity.

The following algorithm determines the vector of coefficients of the product polynomial $r = p \cdot q$, where $p$ and $q$ are polynomials of degree at most $n - 1$.

**Algorithm 4.8 Determining the coefficients for the product polynomial $r = p \cdot q$.**
/* p and q are polynomials of degree at most n − 1. \( x_p \) is the vector of coefficients of p and \( x_q \) is the vector of coefficients of q. */

(1) Compute \( \vec{y}_p = \text{DFT}_{2n} \cdot (0, \vec{x}_p) \) and \( \vec{y}_q = \text{DFT}_{2n} \cdot (0, \vec{x}_q) \);

/* Hence the degree \( n − 1 \) polynomials p and q are evaluated at the \( 2n \)th roots of unity. To achieve this, we have augmented the vectors of coefficients by \( n \) zeroes. */

(2) for \( j = 0 \) to \( 2n − 1 \) do

\[
    z_j = (y_p)_j \cdot (y_q)_j;
\]

/* Observe that \( z \) is the result of evaluating the product polynomial \( r \) at the \( 2n \)th roots of unity. */

(3) \( \vec{x} = \text{DFT}_{2n}^{-1}(\vec{z}) \) is the vector of coefficients of the product polynomial \( r = p \cdot q \).

Lemma 4.9 Algorithm 4.8 is correct. Its running time for degree \( n \) polynomials is proportional to the time required to perform the \( 2n \)th discrete Fourier transform.

**Proof:** In step (1) we evaluate the degree \( n − 1 \) polynomials at the \( 2n \)th roots of unity and in step (2) evaluate the product polynomial \( r \) at the \( 2n \)th roots of unity. Since the degree of \( r \) is \( 2n − 2 \), we determine the vector of coefficients of \( r \) when applying the inverse transform in step (3).

Now we come to the crucial question: how fast can \( \text{DFT}_n \vec{x} \) and \( \text{DFT}_n^{-1} \vec{y} \) be computed?

**Algorithm 4.10 The Fast Fourier Transform**

/* We assume that \( n \) is a power of two. */

(1) Let \( x_{\text{even}} = (x_0, x_2, \ldots, x_{n-2}) \) be the vector of even coordinates and \( x_{\text{odd}} = (x_1, x_3, \ldots, x_{n-1}) \) be the vector of odd coordinates of \( x \).

(2) pardo recursively determine \( u = (\text{DFT}_{n/2}) \cdot x_{\text{even}} \) and \( v = (\text{DFT}_{n/2}) \cdot x_{\text{odd}} \)

with \( \omega_n^2 \) as primitive \( \frac{n}{2} \)th root of unity;

(3) for \( j = 0 \) to \( n − 1 \) pardo

\[
    y_j = \begin{cases} 
        u_j + \omega_n^j \cdot v_j & 0 \leq j < \frac{n}{2} \\
        u_{j-n/2} + \omega_n^j \cdot v_{j-n/2} & \frac{n}{2} \leq j < n.
    \end{cases}
\]
4.4. THE DISCRETE FOURIER TRANSFORM

Let \( T(n) \) be the running time of this “idealized” parallel program for sequences \( x \) of length \( n \). We observe that step (3) “should” run in constant time with \( n \) processors and hence we obtain the recurrence

\[
T(n) = T\left(\frac{n}{2}\right) + O(1)
\]

with solution \( T(n) = O(\log_2 n) \). (If we implement algorithm 4.10 as a sequential algorithm, then we obtain the recurrence \( T(n) = 2 \cdot T\left(\frac{n}{2}\right) + O(n) \) with solution \( T(n) = O(n \cdot \log_2 n) \).

In particular, we can determine the vector of coefficients of a product polynomial in time \( O(n \cdot \log_2 n) \) and thus considerably faster than time \( O(n^2) \) with the school method.)

Why does Algorithm 4.10 work correctly? Is \( \omega_n^2 \) indeed a primitive \( \frac{n}{2} \)th root of unity?

Exercise 61
Show that \( \omega_n^2 \) is a primitive \( \frac{n}{2} \)th root of unity.

Did we choose the right values for \( y_j \)? We start with \( 0 \leq j < \frac{n}{2} \).

\[
y_j = \sum_{k=0}^{n-1} \omega_n^{j-k} \cdot x_k = \sum_{k=0, k \text{ even}}^{n-1} \omega_n^{j-k} \cdot x_k + \sum_{k=0, k \text{ odd}}^{n-1} \omega_n^{j-k} \cdot x_k
\]

\[
= \sum_{k=0}^{n/2-1} \omega_n^{j(2k)} \cdot x_{2k} + \sum_{k=0}^{n/2-1} \omega_n^{j(2k+1)} \cdot x_{2k+1}
\]

\[
= \sum_{k=0}^{n/2-1} (\omega_n^2)^j \cdot x_{2k} + \omega_n^j \cdot \sum_{k=0}^{n/2-1} (\omega_n^2)^{j-k} \cdot x_{2k+1}
\]

\[
= (\text{DFT}_{n/2} \cdot x_{\text{even}})_j + \omega_n^j \cdot (\text{DFT}_{n/2} \cdot x_{\text{odd}})_j
\]

and this was to be shown.

Exercise 62
Show that \( (\text{DFT}_n \cdot x)_i = (\text{DFT}_{n/2} \cdot x_{\text{even}})_{i-n/2} + \omega_n^i \cdot (\text{DFT}_{n/2} \cdot x_{\text{odd}})_{i-n/2} \) holds for \( \frac{n}{2} \leq i < n \).

Exercise 63
A Hadamard matrix is a square matrix \( H \) with \( H[i,j] \in \{-1, 1\} \). Moreover any two rows of \( H \) have be orthogonal. The following recursive definition describes the specific Hadamard matrix \( H_n \) with \( n \) rows and columns:

\[
H_2 = \begin{pmatrix} 1 & 1 \\
1 & -1 \end{pmatrix}, \quad H_n = \begin{pmatrix} H_{n/2} & H_{n/2} \\
H_{n/2} & -H_{n/2} \end{pmatrix}.
\]

Show how to compute the matrix-vector product \( y = H_n \cdot x \) with \( n \) processes in time \( O(\log_2 n) \).

Next we implement Algorithm 4.10 on the butterfly network \( B_d \) where \( n = 2^d \). The input \( x \) is split into the subsequences \( x_{\text{even}} \) and \( x_{\text{odd}} \) and we interpret this step as reorganizing \( x \) to obtain the new sequence \( (x_{\text{even}}, x_{\text{odd}}) \). In other words, \( x_{b_1 \cdots b_k} \) now ends up in position \( b_k b_1 \cdots b_{k-1} \). The next recursion step repeats the partition into even and components, but
restricted to $x_{\text{even}}$ and $x_{\text{odd}}$: hence $x_{b_k \cdots b_1}$ is in position $b_kb_{k-1}b_1 \cdots b_{k-2}$ after two recursion steps and in position $\text{rev}(b_1 \cdots b_k) = b_kb_{k-1} \cdots b_2b_1$ at the end of the recursion. Thus we should input $x$ in “bit reversal order” in column $d$ of $B_d$. Observe that $u_i$ and $v_i$ determine $y_i$ as well as $y_{i+n/2}$ and these computations can be carried out by communicating between columns 0 and 1. Finally, $u_i$ and $v_i$ are the result of recursive calls and their computation succeeds by communicating between columns 1 and 2 and so forth.

Thus the Fast Fourier Transform, when applied to sequences of length $n = 2^d$, uses the butterfly network $B_d$ as communication pattern. We work with $n$ processes with one process per row of $B_d$. Activity begins in column $d$ and then moves left with only one column being active at a time.

Thus our algorithm runs in time $O(d)$ with $n = 2^d$ processors and we have achieved an optimal parallelization, since the Fast Fourier Transform runs in time $O(n \cdot \log_2 n)$ and our implementation produces work $O(d \cdot 2^d) = O(n \cdot \log_2 n)$. We now have to discuss implementations for an arbitrary number of $p$ processors.

### 4.4.1 The Binary Exchange Algorithm

In the binary exchange algorithm we assign disjoint intervals of $2^{d-k}$ $x$-inputs to the $p = 2^k$ processes. (The intervals are chosen relative to the bit reversal order. In particular, for $k = 2$ and $d = 3$ assign $x_0$ and $x_4$ to process 1, $x_2$ and $x_6$ to process 2, $x_1$ and $x_5$ to process
3 and \( x_3 \) and \( x_7 \) to process 4.)

**Exercise 64**
Assume that an algorithm \( \mathcal{A} \), running on \( B_d \), uses only columns \( C_k, \ldots, C_d \) in some order.

Show how to simulate \( \mathcal{A} \) with \( p = 2^k \) processes such that no communication is required. (The \( i \)th process receives the \( i \)th interval of \( 2^{d-k} \) numbers.)

There is no need to communicate for the first \( d-k \) iterations, but in each of the remaining \( k \) iterations each process executes a point-to-point send/receive operation at a communication cost of \( O\left( \frac{n}{p} \right) \). Moreover the computing time during the first \( d-k = \log_2(p) \) iterations is \( O\left( \frac{n}{p} \log_2 \frac{n}{p} \right) \). In the remaining \( k = \log_2 p \) iterations the computing time is \( O\left( \frac{n}{p} \right) \) per iteration and we obtain the total computing time

\[
O\left( \frac{n}{p} \log_2 \frac{n}{p} + \frac{n}{p} \cdot \log_2 n \right) = O\left( \frac{n}{p} \cdot \log_2 n \right).
\]

The total communication time is bounded by \( O\left( \frac{n}{p} \cdot \log_2 p \right) \) and hence is only marginally smaller than the computing time. Thus we obtain constant efficiency, provided \( \log_2 n = \Omega(\log_2 p) \). Since the asymptotic analysis of the communication time hides large constants, we consider also other solutions.

### 4.4.2 The Transpose Algorithms

We try to reduce the communication cost by applying the following observation.

**Exercise 65**
Let \( \mathcal{A} \) be an algorithm on \( Q_d \) with \( 2^d \) processes where process \( u \in \{0, 1\}^d \) receives the input \( x_u \). Firstly \( \mathcal{A} \) uses only edges of dimensions \( \frac{d}{2} + 1, \ldots, d \). From time \( t \) onwards only edges of dimensions \( 1, \ldots, \frac{d}{2} \) are used.

We would like to simulate \( \mathcal{A} \) by an algorithm \( \mathcal{B} \) with \( 2^{d/2} \) processes, where process \( u \in \{0, 1\}^{d/2} \) receives the set of all inputs \( x_{uv} \) with \( v \in \{0, 1\}^{d/2} \). Show that there is an algorithm \( \mathcal{B} \) which simulates \( \mathcal{A} \) with one all-to-all personalized broadcast as the only communication step. Moreover \( \mathcal{B} \) is slowed down by at most the factor \( O\left( 2^{d/2} \right) \).

Imagine that the input sequence \( x \) is arranged into a \( \sqrt{n} \times \sqrt{n} \) mesh such that all communications of the first \( \log_2 \sqrt{n} \) steps of the FFT butterfly algorithm are between sequence elements of a column and hence all communications of the last \( \log_2 \sqrt{n} \) steps are between sequence elements of a row. We assign \( \sqrt{n}/p \) columns to each process and then work in three phases: at first each process implements the fast Fourier transform within its columns, then an all-to-all personalized broadcast is executed and finally, as the last and third phase, each process implements the fast Fourier transform again within its columns. This perspective is the reason why this approach is called the two-dimensional transpose algorithm.
Algorithm 4.11  The two-dimensional transpose algorithm.

/* p = 2^k processes (with p ≤ 2^{d/2}) are available to determine the discrete Fourier transform
for a sequence of length 2^d = n. Process u^1 = u_1 \cdots u_k \in \{0,1\}^k receives all numbers
x_{u_1 \cdots u_k}. */

1. Perform the first $d/2$ iterations of the FFT butterfly algorithm without any com-
   munication.

2. Determine the transpose by an all-to-all personalized broadcast, i.e., process
   $u^1 = u_1 \cdots u_k$ sends the data of butterfly node $(u^1 u^2 v^2 v^1, \frac{d}{2})$ (for $v^1 \in \{0,1\}^k$ and
   $u^2, v^2 \in \{0,1\}^{d/2-k}$) to process $v^1$.

3. Perform the last $d/2$ iterations of the FFT butterfly algorithm without any com-
   munication: process $v^1$ simulates all butterfly nodes $\ast \ast v^1 \cdots v_k$.

In the general case of $p = 2^k \leq 2^{d/2}$ processes, Algorithm 4.11 assigns $2^{d-k} = \frac{n}{p}$ numbers
to any process. The all-to-all personalized broadcast sends $p$ messages of length $\frac{1}{p} \cdot \frac{n}{p} = \frac{n}{p^2}$
each from one process to the remaining $p$ processes. Thus the communication cost is
proportional to $O(\frac{n}{p})$ whereas the computing time of $O(\frac{n \log_2 n}{p})$ is unchanged. We reach
constant efficiency assuming $n = \Omega(p^2)$. If we compare the binary exchange and the transpose
algorithm, then the transpose algorithm reduces the combined message length
by the factor $\log_2 p$, however the cheap send/receive is replaced by the expensive all-to-all
personalized broadcast and we have to expect larger constant factors.

The three-dimensional transpose algorithm is a compromise between the binary ex-
change and the two-dimensional transpose algorithm. Instead of three phases we now
work with five phases. In phase one we perform the first $\frac{1}{3} \cdot \log_2 n$ steps of the fast Fourier
transform without communication. A “restricted” all-to-all broadcast (within the sub-
butterfly on the last $\frac{2 \log_2 n}{3}$ dimensions) guarantees that the next $\frac{1}{3} \cdot \log_2 n$ steps of the
fast Fourier transform run again without communication. After a final all-to-all broadcast
the final $\frac{1}{3} \cdot \log_2 n$ steps run again without communication. Observe that each all-to-all
broadcast exchanges messages of length $\frac{n}{p}$ within groups of $p^{2/3}$ processes. In comparison
with the two-dimensional transpose algorithm, the combined message length doubles, but
the all-to-all broadcasts apply to smaller groups.

Theorem 4.12  Let $n$ be a power of two.

(a)  The binary exchange algorithm runs in time $O(\frac{n \log_2 n}{p})$ and spends $O(\frac{n \log_2 p}{p})$ steps
    on communication.

(b)  The two- and three-dimensional transpose algorithms run in time $O(\frac{n \log_2 n}{p})$ and spend
    $O(\frac{n}{p})$ steps on communication.
4.5 Conclusion

We have parallelized the matrix-vector and the matrix-matrix product for various input partitions and obtained optimal parallel algorithms.

Then we have turned our attention to solving linear systems of equations. We have developed back substitution to solve lower triangular systems and then parallelized Gaussian elimination with partial pivoting. Iterative methods are particularly well suited for sparse linear systems. We have discussed the Jacobi relaxation and overrelaxation, the Gauss-Seidel algorithm and the conjugate gradient method. We then have applied iterative methods to solve linear partial differential equations.

With the binary exchange algorithm and the transpose algorithm we have presented optimal parallelizations of the discrete Fourier transform.
Chapter 5

Algorithms for Hard Problems

We survey Monte Carlo and Markov chain Monte Carlo methods, backtracking, branch & bound and alpha-beta pruning as general techniques to solve computationally hard search and optimization problems. All these methods immediately profit from parallelization. Monte Carlo methods give better results when more more trials are performed and Backtracking, branch & bound and alpha-beta pruning require only very little interaction when partitioning their search efforts among several processes.

We do not attempt to give an in depth discussion, but rather restrict ourselves to a survey emphasizing aspects of parallelization. Most of these approaches are trivially parallelizable, however issues such as dynamic load balancing and termination detection have to be dealt with.

5.1 Monte Carlo Methods

The Monte Carlo method encompasses any technique of statistical sampling employed to approximate solutions to quantitative computational problems.\(^1\) An important application area is to simulate the behavior of various physical and mathematical systems.

**Example 5.1 Approximating \(\pi\)**

We show how to approximate \(\pi\). Observe that the area of the circle \(C\) with radius one equals \(\pi\), whereas the area of the square \(S = [-1, +1]^2\) equals four. Hence if we randomly draw \(p\) points from \(S\) and if \(p(C)\) is the number of points which already belong to the circle \(C\), then the ratio \(\frac{p(C)}{p}\) converges to \(\frac{\pi}{4}\).

\(^1\)With computer science terminology one would speak of randomized algorithms for computational problems.
CHAPTER 5. ALGORITHMS FOR HARD PROBLEMS

Exercise 66
Assume that we draw $n$ points from the square $S$ at random. Use the Chebycheff inequality to bound the error $\frac{|\text{estimate} - \pi/4|}{\pi/4}$.

Example 5.2 Evaluating Multi-Dimensional Integrals
Many deterministic methods of numerical integration such as the Newton-Coles formulas (rectangle and trapezoidal rule, Simpson’s rule) and Gaussian quadrature operate by taking a number of samples from a function. In general, this works very well for functions of few variables. However, for high dimensional functions deterministic integration methods tend to be inefficient. For instance, if we distribute points in the interval $[0, 1]$ with distance at most $\frac{1}{N}$ between neighboring points, then $N$ points are required. To obtain a similar spacing for the cube $[0, 1]^k$, $N^k$ grid points are required.

To numerically integrate a function of a single variable, $N$ equally spaced points for a sufficiently small step size $1/N$ are required. If we integrate a function depending on a $k$-dimensional vector, then for large $k$ the same spacing on a $k$-dimensional grid requires the inordinately large number of $N^k$ grid points.

The Monte Carlo method interprets an integral $\int_{x \in \Omega} f(x)dx$ as the product $V(\Omega) \cdot E_{\Omega}(f)$, where $V(\Omega)$ is the volume of the integration region $\Omega$ and $E_{\Omega}(f)$ is the expected value of $f$ restricted to $\Omega \subseteq \mathbb{R}^k$. As long as the function $f$ in question is reasonably well-behaved, its integral $\int_{x \in \Omega} f(x)dx$ is estimated by randomly selecting points $x_1, \ldots, x_M \in \Omega$ and returning the estimate $\frac{V(\Omega)}{M} \cdot \sum_{i=1}^{M} f(x_i)$. To bound the error of this estimate we use the Chebycheff inequality

$$\text{prob} \left[ |Y - E[Y]| > \frac{t}{t^2} \right] \leq \frac{\text{Var}[Y]}{t^2}$$

and apply it to the random variable $Y = \frac{1}{M} \cdot \sum_{i=1}^{M} X_i$, where $X_i = f(x_i)$ is the result of the $i$th random experiment. We interpret $X_i$ as the $i$th repetition of the experiment $X$ and observe that $E[Y] = E\left[\frac{1}{M} \cdot \sum_{i=1}^{M} X_i\right] = \frac{M}{M}E[X] = E[X]$. But the random experiments $X_1, \ldots, X_M$ are independent and in particular

$$\text{Var}[Y] = \frac{1}{M^2} \cdot \sum_{i=1}^{M} \text{Var}[X_i] = \frac{1}{M^2} \cdot \sum_{i=1}^{M} \text{Var}[X] = \frac{1}{M} \cdot \text{Var}[X]$$

holds. We have reduced variance by the factor $M$ and hence the Chebycheff inequality
implies the inequality
\[
\begin{align*}
\left| \frac{1}{M} \sum_{i=1}^{M} f(x_i) - \frac{1}{V(\Omega)} \int_{x \in \Omega} f(x) dx \right| & > t \\
\Pr \left[ \left| \frac{1}{M} \sum_{i=1}^{M} f(x_i) - \frac{1}{V(\Omega)} \int_{x \in \Omega} f(x) dx \right| > t \right] & = \Pr \left[ |Y - \mathbb{E}[Y]| > t \right] \\
& \leq \frac{\text{Var}[Y]}{t^2} = \frac{\text{Var}[X]}{M \cdot t^2}.
\end{align*}
\]

Thus as a consequence,
\[
\left| \frac{V(\Omega)}{M} \cdot \sum_{i=1}^{M} f(x_i) - \int_{x \in \Omega} f(x) dx \right| > t \cdot V(\Omega)
\]
holds with probability at most \( \frac{\text{Var}[X]}{M \cdot t^2} \). We set \( t = c \cdot \sqrt{\frac{\text{Var}[X]}{V(\Omega)}} \) and the approximation error is larger than \( t \cdot V(\Omega) \) with probability at most \( \frac{1}{c^2} \). Observe that an \( s^2 \)-fold increase in the number of sample points is required, if we want to reduce the error by a factor of \( s \).

This simple approach however converges only very slowly if the function is very complex and for instance has small islands of high activity. A refinement of this method is to somehow make the points random, but more likely to come from regions of high contribution to the integral than from regions of low contribution. In other words, the points should be drawn from a distribution similar in form to the integrand. Understandably, doing this precisely is just as difficult as solving the integral in the first place, but there are approximate methods available such as Markov Chain Monte Carlo methods (see Section 5.1.1).

In the above two examples we get more accurate answers the more trials we perform. Here we have the opportunity to profit from parallel computation by sampling with \( p \) processes instead of using just one process. Observe that this “embarrassingly parallel” approach requires no communication whatsoever with the exceptions of the last step of averaging individual results.

Good pseudo random numbers suffice for Monte Carlo methods and we discuss some important generators in Section 5.1.2. Finally in Section 5.1.1 we describe Markov chain Monte Carlo methods and in particular the Metropolis and Simulated Annealing algorithms as two prominent Monte Carlo optimization methods.

### 5.1.1 Markov Chain Monte Carlo Methods

We begin with a few basic facts on finite Markov chains.

**Definition 5.1** A finite Markov chain \( \mathcal{M} = (\Omega, P) \) is described by a finite set \( \Omega \) of states and by a matrix \( P \) of transition probabilities between states in \( \Omega \). In particular, \( P[u, v] \) is the probability to visit \( v \in \Omega \), given that \( u \in \Omega \) is currently visited. We demand that
\[
\sum_{v \in \Omega} P[u, v] = 1
\]
holds for all states \( u \).
**Exercise 67**

Let $\mathcal{M} = (\Omega, P)$ be a Markov chain. Show that $P^t[u, v]$ is the probability that the random walk reaches state $v$ after $t$ steps provided the walk begins in state $u$.

If there is a path with positive probability between any two states in $\Omega$ and if $P[x, x] > 0$ holds for all states $x$, then the following fundamental properties hold:

(1) $\mathcal{M}$ has a unique **stationary distribution** $\pi$, i.e., $\pi^T \cdot P = \pi^T$ holds. (Assume that $\pi$ is a stationary distribution. If $\mathcal{M}$ is in state $u$ with probability $\pi(u)$, then after one step $\mathcal{M}$ is in state $v$ with probability

$$
\sum_{u \in \Omega} \pi(u)P[u, v] = (\pi^T \cdot P)(v) = \pi(v)
$$

and $\mathcal{M}$ stays in the stationary distribution $\pi$.)

(b) $\lim_{t \to \infty} P^t[u, v] = \pi(v)$ holds and the frequency with which $v$ is visited does not depend on the starting state $u$.

**Exercise 68**

Show property (b).

Markov chain Monte Carlo (MCMC) methods (also called random walk Monte Carlo methods) construct a Markov chain that has a target distribution as its stationary distribution. The state of the chain after a large number of steps is then used as a sample from the target distribution. The quality of the sample improves with time, however the number of steps required to converge to the stationary distribution within an acceptable error, may be quite large. We say that a chain **rapidly mixes** if it reaches the stationary distribution quickly from any other distribution.

**Exercise 69**

(a) We consider a Markov chain with state set $\{0, 1, \ldots, n\}$ and transition probabilities $P[i, i + 1] = P[i + 1, i] = 1/2$ for $1 \leq i \leq n - 1$ and $P[0, 0] = P[n, n - 1] = 1$. Determine the expected time $T(i)$ for each state $i$ in which state 0 is reached.

(b) We have to determine whether a formula $\alpha$ in conjunctive normal form with at most two literals per clause is satisfiable. Our algorithm first chooses an arbitrary assignment $x$. Then it repeats the following steps **sufficiently often**: if the current assignment is satisfying, the algorithm halts. Otherwise determine an arbitrary clause which is not satisfied and flip the value of a randomly selected variable appearing in the clause. If no satisfying assignment is found, declare $\alpha$ to be not satisfiable.

Use part (a) to determine the number of iterations sufficient to find a satisfying assignment with probability at least $1/2$, provided $\alpha$ is satisfiable.

**Example 5.3 Multi-Dimensional Integrals**

A first important application of MCMC methods is the approximation of a multi-dimensional integral $\int_{x \in \Omega} f(x) d^k x$. In these methods, an ensemble of “walkers” moves around
the integration region randomly in the search of “high-activity” areas. A walker checks its current area to determine a point with a considerable contribution towards the integral respectively to determine the next area to walk into. In particular, a Markov chain is constructed for which the integrand corresponds to its stationary distribution.

**Example 5.4 MCMC optimization methods**

A second important application concerns the approximation of optimization problems. Assume that we have to minimize a multi-dimensional function \( f \) over some finite domain \( \Omega \). For any point \( x \in \Omega \) let \( N(x) \subseteq \Omega \) be the neighborhood of \( x \). We interpret the points in \( \Omega \) as states of a Markov chain.

The **Metropolis algorithm** starts at some initial point \( x \in \Omega \). If the algorithm is currently visiting point \( x \), then it randomly chooses a neighbor \( y \in N(x) \) and continues with \( y \) as long as \( y \) is at least as good as \( x \), i.e., \( f(y) \leq f(x) \). The case \( f(y) > f(x) \) is crucial. Since we would like to minimize \( f \) we should be reluctant to continue with \( y \), however if we do not tolerate a worse solution, then we get stuck in local minima.

![Diagram](image)

The Metropolis algorithm proposes a compromise and accepts an uphill move with probability \( e^{-\frac{f(y)-f(x)}{T}} \): the larger the “temperature” \( T \) the higher the probability that a bad neighbor is accepted. Assume that the algorithm continues its walk for a fixed, but sufficiently long time and reaches a solution \( x \in \Omega \). Since the walk has been long and since we only reluctantly accept an uphill move, the probability of a good solution \( x \) (i.e., \( f(x) \) is relatively small) should be relatively high. Indeed it turns out that the probability of \( x \) tends to the probability

\[
q_T(x) = \frac{1}{N_T} \cdot e^{-\frac{f(x)}{T}} \quad \text{with} \quad N_T = \sum_{x \in \Omega} e^{-\frac{f(x)}{T}}.
\]

**Exercise 70**

Assume that the neighborhoods are symmetric (i.e., \( x \in U(x) \) for all \( x \in \Omega \), \( y \in N(x) \) whenever \( x \in N(y) \) and \( |U(x)| = |U(y)| \) for all \( x, y \in \Omega \)). Moreover assume that there is a path from \( x \) to \( y \) for any pair \( (x, y) \in \Omega^2 \). We fix the temperature \( T \). Then the Markov chain \( (\Omega, P_T) \) has a unique stationary distribution \( \pi \).

(a) Show that \( q_T \) is reversible, i.e., \( q_T(x) \cdot P_T[x, y] = q_T(y) \cdot P_T[y, x] \) holds.

(b) Show that any reversible distribution \( \rho \) coincides with the stationary distribution. Thus \( q_T \) coincides with the stationary distribution \( \pi \).
The next example shows that we should incorporate a mechanism to reduce the temperature $T$.

Example 5.5 In the Vertex Cover problem we are given an undirected graph $G = (V, E)$ and we are looking for a cover, i.e., a subset $C \subseteq V$ of minimal size such that each edge has at least one endpoint in $C$. We say that subsets $U_1, U_2 \subseteq V$ are neighbors iff $U_2$ results from $U_1$ after inserting a node or removing a node from $U_1$.

We apply the Metropolis algorithm to the empty graph, i.e., $E = \emptyset$. Obviously the empty set is a minimal cover. Assume we start with the cover $x = V$. Initially the Metropolis algorithm removes elements from its current solution. But if its current solution $x$ has only few elements, then there are far more larger than smaller neighbors: the Metropolis algorithm begins to add nodes!

Observe that the smaller we choose $T$ the higher will be the probability of global minima and it is to be expected that the time to converge increases sharply with decreasing temperature. Simulated Annealing\footnote{Simulated Annealing has received its name in analogy with physical annealing in which a material is first heated and atoms can rearrange freely. When slowly cooling down, the movement of atoms is more and more restricted until the material reaches a minimum energy state: hopefully a perfect crystal with regular structure has formed.} (SA), an extension of the Metropolis algorithm, tries to reduce the convergence time by proposing a cooling schedule. SA starts a random walk with a fixed temperature $T = T_0$ where $T_0$ has to be sufficiently large. After “sufficiently many” steps the temperature $T$ will be cooled down geometrically by setting $T = \alpha \cdot T$ with $0 < \alpha < 1$ and a new random walk is started.

Algorithm 5.2 Metropolis and Simulated Annealing

1. Set $x = x_0$ for some initial solution $x_0 \in \Omega$. Choose a temperature $T$. /* The larger $T$ the more likely are uphill moves. */

2. Repeat sufficiently often:
   (a) Choose a neighbor $y$ of $x$ with probability $P[x, y]$.
   (b) IF $f(y) \leq f(x)$ THEN set $x = y$ ELSE set $x = y$ with probability $e^{-f(y) - f(x)}$.

3. Whereas the Metropolis algorithm now terminates, Simulated Annealing may choose a lower temperature $T$ and return to step (2).

We choose the problem of graph partition as an application of SA. We are given an undirected graph $G = (V, E)$ and look for a subset $W \subseteq V$ with $|W| = \frac{1}{2} \cdot |V|$ such that the number of crossing edges, i.e., $|\{e \in E : |e \cap W| = 1\}|$ is minimal. The minimization
5.1. MONTE CARLO METHODS

Problem is \( \mathcal{NP} \)-complete and no efficient solutions are to be expected. As solutions we accept arbitrary partitions \((W, V \setminus W)\) and choose

\[
 f(W) := | \{ e \in E : | e \cap W | = 1 \} | + \alpha \cdot (| W | - | V \setminus W |)^2
\]

as objective function, where the penalty term tries to enforce a balanced partition. The neighborhood \( N(W) \) consists of all subsets \( W' \subseteq V \) with \( | W \oplus W' | \leq 1 \). SA starts with a randomly chosen partition \((W, V \setminus W)\) with \( | W | = \frac{1}{2} \cdot | V | \) and then tries to find better partitions by successively inserting, resp. removing nodes.

[JAGS89] proposes to determine the initial temperature \( T_0 \) such that 40\% of all neighbors are accepted. Furthermore they keep the temperature constant over a period of \( 16 \cdot | V | \) steps and then cool it down by the factor \( \alpha = 0.95 \). If \( G \) is randomly generated, then SA is more successful than tailor-made heuristics and this phenomenon occurs even if we allot the same time that SA consumes. Further experiments are run on structured graphs by choosing between 500 and 1000 points from the square \([0, 1]^2\) and connecting two points whenever they are sufficiently close. For this class of “geometric random” graphs the performance of SA collapses and tailor-made heuristics turn out to be far superior even if SA is given significantly more time.

In other words one should not expects miracles from either Metropolis or Simulated Annealing. However if problem-specific algorithms are not available, then both algorithms are excellent starting points. Performing \( p \) random walks in parallel is easily achieved without any communication.

So far we have considered Markov chains whose set of states is so huge that a direct computation of the stationary distribution is completely out of question. The next example shows that parallel algorithms are of central importance when the set of states is of barely manageable size.

**Example 5.6 Google**
The search engine Google assigns a page rank \( pr(w) \) to a website \( w \), where \( pr(w) \) is evaluated by a peer review: \( pr(w) \) “should” be the higher the more websites with high page rank point to \( w \). To define the page rank Google sets up a Markov chain \((W, P)\) where \( W \) is the set of all websites. Moreover, if fanout\((w)\) is the number of pages that \( w \) points to, then Google defines

\[
 P[w_1, w_2] = \begin{cases} 
 0 & \text{if } w_1 \text{ does not point to } w_2, \\
 \frac{1}{\text{fanout}(w_1)} & \text{otherwise.}
\end{cases}
\]

Imagine that a “random surfer” begins at some website \( v \) and jumps at random from one page to the next. Then we would like to define \( pr(w) \) as the probability that \( w \) is reached after a fixed, but sufficiently large number of steps. We have to require that \( pr(w) \) does not depend on the starting part, however we face a problem for instance if we start from a
dead end. Therefore Google inserts new low-probability links and connects each page \( w_1 \) with any other page \( w_2 \). The probability to reach a given website \( w \) is now independent of the starting point and coincides with \( \pi(w) \), where \( \pi \) is the stationary distribution.

To compute page ranks one has to determine the stationary distribution, a daunting task involving a transition matrix with several billions of rows and columns. Google chooses a parallel solution employing several thousand PC’s. We begin with the uniform distribution \( \pi_0 \) and, if \( \pi_i \) is determined, set \( \pi_{i+1}^T = \pi_i^T \cdot P \). Two facts help. First of all the transition matrix \( P \) is sparse and hence the total work per iteration is “only” bounded by a few ten billions. Secondly the web chain is rapidly mixing and we obtain a good approximation of the stationary distribution after relatively few iterations. Thus ultimately the success of Google is based on the fact that the matrix-vector product is highly parallelizable.

### 5.1.2 Pseudo Random Number Generators

A generator \( G \) is a deterministic algorithm which produces a string \( y = G(x) \in \{0,1\}^\infty \) given a seed \( x \in \{0,1\}^n \). Moreover we require \( p(n) > n \) and hence \( G \) has to stretch its seed \( x \) into a longer string \( y \). A statistical test \( T \) is a randomized algorithm which either outputs zero or one and runs on inputs of length \( n \) in time polynomial in \( n \). One says that a generator \( G \) passes the test \( T \) if the acceptance probability \( r_n \) of \( T \), given a truly random string of length \( p(n) \), is not observably different from the acceptance probability \( g_n \) of \( T \), given a string \( G(x) \). \(^3\) (The acceptance probability is determined by running \( T \) on all strings of length \( p(n) \), respectively on all strings \( G(x) \) for \( x \in \{01\}^n \).) Finally we say that \( G \) is a cryptographically secure pseudo random generator, provided \( G \) passes all statistical tests.

#### Example 5.7 The Blum-Blum-Shub (BBS) Generator

For a seed \( s_0 \) determine the sequence \( s_{i+1} = s_i^2 \mod N \), where \( N = p \cdot q \) with primes \( p \equiv q \equiv 3 \mod 4 \). The BBS generator produces the pseudo-random string \( G(s_0) = (s_1 \mod 2, \ldots, s_m \mod 2) \) with, say, \( m = \lceil \log_2 s_0 \rceil^k \) for a constant \( k \). One can show that the BBS generator is cryptographically secure, provided factoring of most numbers \( N = p \cdot q \) —with primes \( p \equiv q \equiv 3 \mod 4 \) — is computationally hard. Observe however that the BBS generator is quite expensive since we have to square in order to get one random bit.

The remaining prominent generators trade cryptographical security against speed of evaluation.

#### Example 5.8 The Linear Congruential Generator

The linear congruential generator (LC) is defined by its modulus \( m \), its coefficient \( a \) and its offset \( b \). We generate a sequence \( x_i \) by starting with a seed \( x_0 \) and setting

\[
x_{i+1} = a \cdot x_i + b \mod m.
\]

\(^3\)\( r_n \) is not observably different from \( g_n \) iff for all \( k \in \mathbb{N} \) there is a bound \( N_k \) such that \( |g_n - r_n| \leq n^{-k} \) for all \( n \geq N_k \).
One can show that the LC generator is cryptographically insecure, however it is fast and has large periods, if $m$ is a sufficiently large prime number.

**Exercise 71**
We consider the linear congruential generator

$$x_{i+1} = ax_i + b \mod m.$$  

(a) Show that

$$x_i = a^i x_0 + b \cdot (1 + a + \cdots + a^{i-1}) \mod m$$  

holds.

(b) Suppose that $m$ is a prime. Show: if $x_{i+j} = x_i$, then $x_0 = x_j$.

(c) Suppose that $m$ is a prime and that $|\{a^i \mod m \mid i\}| = m - 1$ holds. How many initial values $x_0$ have period length at most $K$? ($x_0$ has period length at least $K$ iff $x_0, \ldots, x_{K-1}$ are pairwise distinct.)

**Example 5.9 The Mersenne Twister**
The Mersenne twister MT 19937 is a very fast generator with the gigantic period length $2^{19937} - 1$. Its period length is a Mersenne prime, explaining its name. The Mersenne twister is based on the technique of generalized linear feedback shift registers. Its recurrence is of the form

$$x_n = x_{n-227} \oplus (x_{n-624}^U \circ x_{n-623}^L) \cdot A$$

All sequence numbers $x_n$ are 32-bit words. $x_m^U$ is the leading bit of $x_m$ and $x_m^L$ is the string consisting of the trailing 31 bits of $x_m$. $A$ is a $32 \times 32$ bit matrix with a $31 \times 31$ identity matrix in the upper left. The first column of $A$ is zero with the exception of the entry in row 32. Its last row is 9908B0DF in hexadecimal. MT 19937 requires a seed of 19937 = $32 \cdot 623 + 1$ bits, namely 623 strings of length 32 plus the leading bit of $x_0$. The recurrence expands the seed each time by 32 bits.

The notion for cryptographical security requires an infinite family of generators and hence cannot be applied to the Mersenne twister. The Mersenne twister does pass for instance the Diehard tests\footnote{See http://random.com.hr/products/random/manual/html/Diehard.html}, but there is no strong mathematical theory in support. The twister has become increasingly popular for Monte Carlo methods due to its speed and its extremely large period length. It is now part of the GNU scientific library.

**Example 5.10 The Lagged Fibonacci Generator**
The lagged Fibonacci generator uses the recurrence

$$S_n = S_{n-j} \ast S_{n-k} \mod M$$

for $0 < j < k$. (This generalizes the recurrence $S_n = S_{n-1} + S_{n-2}$ defining the Fibonacci sequence.) The operation $\ast$ may either be addition, subtraction, multiplication or bitwise xor. Its advantages are speed and long period lengths, however the generators are mathematically only incompletely understood and their speed and period length are inferior in comparison to the Mersenne twister.
When running a randomized parallel algorithm, a master process should assign different seeds to the processes, since we would like the processes to compute independently. This can be done by creating a pseudo random sequence sequentially and by assigning subsequences to the processes.

5.2 Backtracking

In a decision problem we are given a set $U$ of potential solutions and we have to determine whether $U$ contains a true solution. For instance let $\alpha$ be a conjunction of disjunctions. $\alpha$ describes the set $U$ of all truth assignments to the variables of $\alpha$ and we have to determine whether $U$ contains a true solution, namely an assignment satisfying $\alpha$.

Backtracking searches for a solution by trying to construct a true solution step by step from partial solutions. Backtracking begins with the partial solution $r = U$, which corresponds to the set of all potential solutions. It then applies a branching operator $B$ to $r$ which returns a partition $r_1 \cup \cdots \cup r_k$ of $r$. The branching operator $B$ defines the backtracking tree $T$: initially $T$ consists only of the root $r$. Then we attach children $r_1, \ldots, r_k$ to $r$ to mimic the partition $r = r_1 \cup \cdots \cup r_k$. In general, if $B$ has a node $v$ which is not a singleton set, then we apply $B$ to $v$ to obtain a partition $v = v_1 \cup \cdots \cup v_l$ and correspondingly we make $v_1, \ldots, v_l$ children of $v$.

Backtracking tries to generate only a very small portion $B^*$ of $B$. Namely, whenever it finds that a node $v$ does not have a true solution, it disqualifies $v$ and $v$ will not be expanded any further. Often backtracking generates $B^*$ in a depth first search manner:

- if it currently inspects $v$ and if $v$ can be disqualified, then it “backs up” and continues with the parent of $v$. Otherwise it continues recursively with a not yet inspected child of $v$.

Example 5.11 Assume that $\alpha(x_1, \ldots, x_n)$ is a conjunction of disjunctions. We would like to determine whether $\alpha$ is satisfiable, i.e., whether $\alpha$ has a satisfying assignment (of truth values to propositional variables).

In our approach partial solutions correspond to partial assignments. To be specific, assume that we already have assigned truth value to all variables $x_j$ for $j \in J$. We determine a disjunction $d$ of minimal size and choose an arbitrary variable $x_i$ appearing in $d$. The branching operator $B$ then produces two partial solutions by additionally setting $x_i = 0$ respectively $x_i = 1$.

Why do we define a truth value for a variable appearing in a disjunction of minimal size? Our hope is that this allows us to faster falsify partial assignments. In particular, we run the following test after fixing the value of $x_i$: we look for any disjunction with exactly one unspecified variable, fix the variable appropriately and continue looking for disjunctions with exactly one unspecified variable. If some disjunction is falsified during this process, then the partial assignment is doomed and we disqualify it.
5.3. BRANCH & BOUND

Exercise 72
Can we refine the satisfiability test for partial assignments by inspecting any disjunction with at most two unspecified variables, setting one of the variables arbitrarily and checking the consequences?
Of course, the satisfiability test should be efficient.

Backtracking uses depth-first search to minimize the required memory resources. When implementing parallel Backtracking with \( p \) processes it is advisable that first each process generates the same top portion of the search tree such that the number of nodes is at least proportional to \( p \). If that is done, each process starts to search within the subtrees of “its” nodes. Observe that so far no communication is required. However this changes when a process runs out of work and hence we encounter a problem of dynamic load balancing. We discuss load balancing schemes in Section 6.1.

After deciding which load balancing scheme to use we still have to take care of an apparently innocent problem: how to decide when to terminate? Observe that a process cannot assume that it is done just because it currently does not have work to do. Even asking the remaining processes is no good idea: if a process replies that it is idle, then this process may receive a work assignment soon afterwards. We describe Dijkstra’s token termination detection algorithm for the general termination problem in chapter 7.

5.3 Branch & Bound

Our goal is to minimize a function \( f \) over a finite domain \( \Omega \). Branch & Bound again utilizes the branching operator \( B \), but does not differentiate between potential and true solutions. As for backtracking the branching operator defines a tree \( T \), which this time we call the branch & bound tree. The crucial requirement of Branch & Bound is the existence of a lower bound \( \alpha \) with

\[
\alpha(v) \leq \min \{ f(x) \mid x \in \Omega \text{ is a leaf in the subtree of } T \text{ with root } v \}.
\]

Example 5.12 In the traveling salesman problem (TSP) we are given a set of nodes in the plane and are asked to compute a tour of shortest length traversing all nodes. (Observe that we do not require the tour to return to its origin.) TSP leads to an \( \mathcal{NP} \)-complete problem and is therefore in all likelihood a computationally hard problem. A somewhat related, but computationally far easier problem is the minimum spanning tree problem (MST) in which we look for a spanning tree\(^5\) with minimal weight, i.e., minimal sum of edge lengths. What is the relation between TSP and MST? If a path \( P \) of length \( L \) traverses all nodes, then we have found a spanning tree of weight \( L \), namely the path \( P \). In other words, we have found a lower bound for TSP which is computable within reasonable resources.

Branch & Bound begins by constructing a “good” initial solution \( x \) with the help of a heuristic and sets \( \beta = f(x) \). Initially only the root of \( T \) is unexplored. In its general step

\(^5\)A spanning tree connects all nodes.
Branch & Bound has computed a set $U$ of unexplored nodes and it chooses an unexplored node $v \in U$ to explore. If $\alpha(v) \geq \beta$, then the best solution in the subtree of $v$ is not any better than the best solution found so far and Branch & Bound considers the entire subtree with root $v$ to be explored. If $v$ is a leaf corresponding to a solution $x$, then $\beta$ is updated by $\beta = \min\{\beta, f(x)\}$. Otherwise Branch & Bound generates all children of $v$ and add them to its set $U$ of unexplored nodes. It terminates with the optimal value $\beta$ if all nodes are explored.

There are various heuristics to select an unexplored node from the set $U$, among them versions and combinations of depth-first and best-first search. In best-first search an unexplored node $v$ with smallest lower bound $\alpha(v)$ is selected. If several choices remain, then the deepest node and hence the most constrained node is selected.

**A Parallel Branch & Bound Implementation**

We let all processes simultaneously determine the top portion of the Branch & Bound tree. Afterwards processes determine, according to some predefined procedure, “their” subproblems. Each process $i$ works on its subproblem by representing its set $U_i$ of unexplored nodes by its own private priority queue. Up to this point no communication is required and communication becomes necessary only

- if a process runs out of work and has to request work using some dynamic load balancing scheme or
- if a process has found a better upper bound.

Branch & Bound is the more successful the better the lower and upper bounds are. Therefore, to obtain good upper bounds as fast as possible, some parallel implementations let processes also exchange promising unexplored nodes.

**5.4 Alpha-Beta Pruning**

Assume that we have to find a winning move in a two-player game with players Alice and Bob. Alice begins and the two players alternate. We require the game to be finite and that it ends with a payment to Alice: If the payment is $-1, 0$ or $1$, then Alice wins, if she receives a payment of $1$ and Bob wins, if she receives a payment of $-1$. Our goal is to determine a winning strategy for Alice, that is a strategy that guarantees the highest possible payment to Alice.

Any such game has a game tree $B$. The root $r$ corresponds to the initial configuration and it is marked with Alice, who starts the game. If a node $v$ of $B$ corresponds to the configuration $c_v$, then we generate a child $w$ of $v$ for any possible move: the child $w$ is marked with the opposing player and it corresponds to the configuration $c_w$ as altered by
the current move. If however the game is decided in \( v \), then \( v \) becomes a leaf and we mark \( v \) with the payment \( A(v) \) to Alice.

We can determine the highest possible payment to Alice with the \textbf{minimax} evaluation of \( B \), an application of depth-first search. We say that a node \( v \) of \( B \) is a max-node, if \( v \) is marked with Alice and otherwise that \( v \) is min-node.

\textbf{Algorithm 5.3} The function \textit{Minimax}(\( v \))

1. If \( v \) is a leaf, then return \( A(v) \).
2. If \( v \) is a max-node, then \hspace{1cm} // Alice makes a move.
   - Max = \(-\infty\),
   - traverse all children \( w \) of \( v \) and set Max = max\{Max, Minimax(\( w \))\}.
     \hspace{1cm} // Alice makes her best move.
   - Return Max.
3. If \( v \) is a min-node, then \hspace{1cm} // Bob makes a move.
   - Min = \( \infty \),
   - traverse all children \( w \) of \( v \) and set Min = min\{Min, Minimax(\( w \))\}.
     \hspace{1cm} // Bob makes his best move.
   - Return Min.

Indeed the minimax evaluation can be significantly accelerated. Namely, let us have a look at the minimax evaluation of a max-node \( v \) and let us assume that the min-node \( u \) is an ancestor of \( v \). Moreover assume that the Min-variable of \( u \), when visiting \( v \), equals \( \beta \). To evaluate the max-node \( v \) we have to consider all children \( w \) of \( v \), that is we have to consider all possible moves of Alice. If Alice can enforce, for some child \( w \), a payment of at least \( \beta \), then Bob can prevent Alice from reaching \( v \) without increasing her payment: for instance he can make a move in \( u \) to exclude \( v \), but to still limit Alice to payments of at most \( \beta \). The evaluation of \( v \) can be stopped!

Of course a similar argument applies to the evaluation of a min-node \( v \): Let the max-node \( u \) be an ancestor of \( v \) and assume that the Max-variable of \( u \) currently equals \( \alpha \). If Bob has a particularly clever move \((v, w)\), which bounds the payment of Alice to at most \( \alpha \), then Alice can move in \( u \) as to make \( v \) unreachable and still receive payment \( \alpha \). How should we define \( \alpha \) and \( \beta \) when evaluating a min-node, resp. a max-node \( v \)?

\( \alpha \) should be the maximal value of a Max-Variable of a “max-ancestor” of \( v \) and \( \beta \) should be the minimal value of a Min-variable of a “min-ancestor” of \( v \).
If we implement these ideas, then we are led to the alpha-beta algorithm, which again searches the game tree via depth-first search, but accelerates the search by using parameters $\alpha$ and $\beta$.

**Algorithm 5.4** The function alpha-beta $(v, \alpha, \beta)$.

/* The first call uses parameters $\alpha = -\infty$, $\beta = +\infty$ and $v = r$. */

1. If $v$ is a max-leaf, then return $\alpha = \max\{\alpha, A(v)\}$.
   If $v$ is a min-leaf, then return $\beta = \min\{\beta, A(v)\}$.
   /* We make sure that the invariant holds for $v$. */

2. Otherwise work recursively:
   - If $v$ is a max-node, then // Alice makes a move.
     * Max = $\alpha$,
     * traverse all children $w$ of $v$: if alpha-beta($w, \alpha, \beta$) $\geq \beta$, then stop the traversal and return $\alpha$.
     // Bob can prevent Alice from reaching $v$.
     Otherwise Max = $\max\{\text{Max}, \text{alpha-beta}(w, \alpha, \beta)\}$.
     // Alice makes her best move.
     * Return $\alpha = \text{Max}$.
   - If $v$ is a min-node, then // Bob makes a move.
     * Min = $\beta$,
     * traverse all children $w$ of $v$: if $\alpha \geq \text{alpha-beta}(w, \alpha, \beta)$, then stop the traversal and return $\beta$.
     // Alice can prevent Bob from reaching $v$.
     Otherwise Min = $\min\{\text{Min}, \text{alpha-beta}(w, \alpha, \beta)\}$.
     // Bob makes his best move.
     * Return $\beta = \text{Min}$.

What is the result of Alpha-Beta($v, \alpha, \beta$)?

**Lemma 5.5** For the subtree with root $v$, let $A$ be the largest payment reachable by Alice.

(a) If $v$ is max-node, then $\max\{\alpha, A\}$ is returned, provided $A \leq \beta$.

(b) If $v$ is a min-node, then $\min\{\beta, A\}$ is returned, provided $\alpha \leq A$.

**Exercise 73**
Show Lemma 5.5.
How should we choose the parameters of the first call? Certainly we should start the evaluation of $B$ at the root $r$. If we set $\alpha = -\infty$ and $\beta = \infty$, then Lemma 5.5 guarantees, that the maximally reachable payment to Alice is returned. The first call has therefore the form

$$\text{Alpha-Beta}(r, -\infty, \infty).$$

**Exercise 74**

(a) What is the final value of $\beta$, when Alpha-Beta $(r, -\infty, \infty)$ terminates?

(b) How can one determine an optimal first move of Alice?

The next problem shows that the alpha-beta algorithm visits only $\Theta(\sqrt{N})$ vertices in the best case, if $N$ is the number of vertices of $B$.

**Exercise 75**

Let $B$ be a complete $b$-ary tree of depth $d$, whose leaves are labeled with arbitrary real-valued scores.

(a) Show: $B$ has a path, called a principal variation, which corresponds to a game played optimally by both players.

(b) Show: there is an evaluation of $B$ by alpha-beta in which at most $\text{opt} = b^{\lceil d/2 \rceil} + b^{\lfloor d/2 \rfloor} - 1$ nodes have to be traversed. Moreover any evaluation of $B$ by alpha-beta pruning has to traverse at least $\text{opt}$ nodes.

Hence, in the best case, alpha-beta reduces the search effort from $\Theta(b^d)$ to $\Theta(\sqrt{b^d})$. Hence, in comparison with the minimax evaluation the number of simulated moves can be doubled.

The experimental experience for instance for chess programs shows that the best case is indeed almost reached. To see why we consider the typical structure of a chess program, which consists of an *evaluation procedure*, which assigns a score to a configuration, and a *search procedure* which evaluates a move with as high a lookahead as possible. The search procedure is mostly based on variants of the alpha-beta algorithm, such that a heuristic powers the depth-first search component: depth-first search continues its search with the highest rated successor configuration. Thus, if the evaluation procedure is good, then it is not that surprising that the best case is reached.

However also alpha-beta search finds optimal moves only for trivial games. For instance one assumes that there are about $38^{84}$ different configurations in chess and even a reduction to $38^{42}$ is not going to help much. Therefore chess programs try to achieve as high a lookahead as possible to judge a move; the evaluation procedure is used to obtain a score for the obtained final configurations. The true power of alpha-beta search, in comparison with the minimax evaluation is the doubled lookahead.

There are many heuristic modifications on top of the alpha-beta pruning paradigm. We mention the two most important ones.

- **Aspiration search**: we start conventional alpha-beta pruning with the “search window” $[-\infty, \infty]$, i.e., we set $\alpha = -\infty$ and $\beta = +\infty$. If we have evidence that the
minimax value lies in the search window \([\alpha, \beta]\) with \(-\infty < \alpha \leq \beta < \infty\), then it is advisable to start with the smaller search window, since then the number of explored nodes might decrease considerably.

- **Iterative deepening:** assume that heuristic evaluations are available for all nodes of the game tree and assume that we have already explored all nodes up to depth \(d\). As a consequence we already know a principal variation \(PV_d\) for depth \(d\). When trying to deepen our search iteratively to say depth \(d + 1\), then it is reasonable to conjecture that a principle variation \(PV_{d+1}\) for depth \(d + 1\) is “close” to \(PV_d\) and hence it makes sense to begin alpha-beta pruning with the nodes of \(PV_d\). If our conjecture is correct, then correspondingly many nodes can be pruned.

We may also combine aspiration search with iterative deepening. If \(m_d\) is the minimax value for depth \(d\), then \(m_{d+1}\) ”should” not differ that much from \(m_d\) and a small search window around \(m_d\) ”should” suffice.

**Exercise 76**

We consider a complete binary tree \(T\) of depth \(2d\) whose leaves are labelled with 0 (corresponding to a lost game for Alice) or 1 (corresponding to a won game for Alice). We have to determine whether Alice has a winning strategy.

(a) Show that any deterministic algorithm has to inspect all \(4^d\) leaves in the worst case.

(b) Now consider the following randomized algorithm. To evaluate an inner node \(v\) of \(T\) randomly select a child and evaluate the child recursively. If the evaluation of the child already determines \(v\), then the evaluation of \(v\) terminates. Otherwise the remaining child is evaluated as well.

Show: if the algorithm is started at the root, then it inspects on the average at most \(O(3^d)\) leaves.

**Alpha-Beta Pruning, a Parallel Implementation**

Certainly we could proceed as for Backtracking and Branch & Bound and, for instance, if the number of first moves of Alice exceeds the number of processes, partition the children of the root among the processes. But this strategy will blow up the search overhead, namely the increase in the number of traversed nodes in comparison with a sequential implementation. Observe that we have to battle search overhead with communication overhead and finding the right compromise is the main challenge in the design of parallel implementations of alpha-beta pruning.

In particular let us assume that the game tree \(G\) is the complete \(b\)-ary tree of depth \(d\). Then we know that \(\text{opt}_d = b^{\lfloor d/2 \rfloor} + b^{\lceil d/2 \rceil} - 1\) nodes of \(G\) are inspected in the best case. If a parallel alpha-beta pruning implementation with \(p = b\) decides to evaluate all children of the root in parallel, then each process has to inspect \(\text{opt}_{d-1} = b^{\lfloor (d-1)/2 \rfloor} + b^{\lceil (d-1)/2 \rceil} - 1\) nodes of its subtree. If \(d\) is even, then \(\text{opt}_{d-1} = b^{\lfloor d/2 \rfloor} + b^{\lceil d/2 \rceil} / b - 1 \geq b^{\lfloor d/2 \rfloor} \geq \text{opt}_d / 2\) and the best achievable speedup is two! Observe that in this particular case the search overhead is \(p \cdot \text{opt}_{d-1} - \text{opt}_d \geq p \cdot \text{opt}_d / 2 - \text{opt}_d = (p/2 - 1) \cdot \text{opt}_d\).
More or less all parallel implementations of alpha-beta pruning try to mimic a depth-first traversal to various degrees. In particular, the **young brothers wait concept (YBWC)** is emphasized: the leftmost child (i.e., the eldest brother) has to be evaluated before processes work on the remaining siblings (i.e., the younger brothers). Even if the leftmost child is not optimal, its minimax value may help to narrow the search windows for its siblings. For instance, assume that alpha-beta pruning has determined the minimax value \( m_w \) of the leftmost child \( w \) of the MAX node \( v \). If we now explore the remaining children \( w' \) of \( v \) in parallel, then it suffices to find a move of \( B \) in \( w' \) with final minimax score at most \( m_w \) to prune the subtree of \( w' \). Thus in a setting where good moves are explored first, it pays to throw all computing power at the subtree of the leftmost child and then to process siblings in parallel.

We describe two approaches based on YBWC. The first approach uses “synchronization nodes” whereas the second approach uses asynchronous parallelism.

- **Partial synchronization**: A leftmost child \( v \) is a **synchronization nodes**, whenever a parallel implementation enforces YBWC by exploring \( v \) before its siblings. In many implementations all nodes of the leftmost path \( p \) of \( G \) are synchronization nodes. Consequences are at least initially dramatic. For instance only one process is at work when the deepest node of \( p \) is evaluated and more processes enter only after higher nodes of \( p \) are reached. Thus a strict enforcement of YBWC keeps the search overhead low at the expense of unbalanced work loads.

If the computation progresses there is sufficient work and load balancing becomes an important issue. Idle processes send work requests via random polling. If a process \( q \) receives such a request from process \( p \), it checks its current depth-first path \( p_d \) and chooses a sibling \( s \) of a node of \( p_d \). It sends \( s \) to \( p \) and enters a master-slave relationship with slave \( p \). The slave \( p \) may become a master after receiving a work request. When finishing its task and when all of its slaves have terminated as well, the slave \( p \) becomes idle. In this case \( p \) either asks its master \( q \) for more work or it launches a work request and terminates the old master-slave relationship with \( q \).

The chess program Zugzwang of the university of Paderborn applies partial synchronization. It became vice champion in the 1992 Word Computer Chess Championships. However partial synchronization seems not adequate in workstation clusters where typically processors are powerful and links are slow.

- **Asynchronous parallel hierarchical iterative deepening (APHID)** uses fixed master-slave relationships [B98]. If the first \( d \) levels of the game tree \( G \) have to be explored, then the master first evaluates the top \( d' < d \) levels. Whenever the master encounters a node of depth \( d' \) for the first time it assigns the evaluation task to a slave while trying to balance the work load.

---

\(^6\)Observe that this concept does *not* enforce a depth-first traversal, since the number of siblings is large. We may evaluate siblings in parallel and hence by a breadth-first search.
The master continuously repeats his evaluations of the top $d'$ levels, accepting updates from the slaves, performing heuristic evaluations of uncertain leaves, informing slaves to terminate a task, performing load balancing by reallocating tasks from overworked to moderately busy processes, respectively informing a slave about the (changed) relevance of its leaves. The relevance of a leaf is determined for instance by YBWC and the search depth achieved so far for the leaf: the smaller the search depth the higher the priority, thus allowing the leaf to catch up.

Slaves use iterative deepening to work on their assigned tasks completely independent of other slaves and hence work asynchronously. They also work with aspiration search, centered around the hypothetical minimax value of the root. The master may update the hypothetical value after computing better heuristic estimates due to preliminary results obtained from the slaves.

It may seem that the master is a communication bottleneck. This can certainly be the case, but a fixed master-slave hierarchy alleviates this problem.

The approach of APHID seems to be more adequate for workstation clusters. However, since slaves are working asynchronously, APHID has a considerable search overhead which it tries to reduce by firstly controlling the top $d'$ levels through a single process and secondly by assigning higher relevance for leftmost leaves. In [B98] at least competitive performance in comparison with partially synchronous alpha-beta search is claimed.

5.5 Conclusion

Monte Carlo methods are prime examples of embarrassingly parallel algorithms and are responsible for consuming a substantial percentage of supercomputer cycles. We have considered applications such as approximating multi-dimensional integrals and optimization by Markov chain Monte Carlo methods (i.e., the Metropolis and simulated annealing algorithm). Finally we have discussed pseudo-random generators to supply randomness deterministically.

Backtracking (for solving search problems) and alpha-beta pruning (for solving optimization problems) require the parallelization of sequential tree traversals. Important side issues were load balancing, a subject that we consider in the next chapter in more detail, and termination detection.

---

7A leaf is uncertain, if the responsible slave did not yet report its final result. The slave however may report preliminary results which are then part of a new heuristic estimate of the corresponding leaf.
Chapter 6

Load Balancing

We have encountered load balancing problems when performing backtracking, branch & bound and alpha-beta pruning. But load balancing is a problem occurring frequently whenever processors have different amounts of work or have different speed. As an example we consider the embarrassingly parallel problem of computing an approximation of the Mandelbrot set.

Example 6.1 Approximating the Mandelbrot Set.
Let \( c \) be a complex number and consider the iteration

\[
z_0(c) = 0, \quad z_{k+1}(c) = z_k^2(c) + c.
\]

Then the Mandelbrot set \( \mathcal{M} \) is defined as the set of all complex numbers \( c \) such that \( z_k(c) \) remains bounded, i.e., \( |z_k(c)| \leq M(c) \) for a suitable constant \( M(c) \) and all \( k \).

Exercise 77
Show that \( c \) belongs to \( \mathcal{M} \) iff \( |z_k(c)| < 2 \) for all \( k \).

To display the Mandelbrot set we use a regular grid of pixels and color a pixel \( c \not\in \mathcal{M} \) with a color which depends on the smallest \( k \) with \( |z_k(c)| \geq 2 \). There are faster methods, but for the purpose of this example we restrict ourselves to using the recurrence when determining membership in \( \mathcal{M} \). Certainly we may bound the number of iterations (and possibly incorrectly claim that a pixel belongs to \( \mathcal{M} \), if we did not reach 2 within the bound), however still different pixels present radically different work loads, if the bound is sufficiently large.

There are different load balancing techniques such as static load balancing, where pixels are assigned to processes ahead of time, or dynamic load balancing, where idle processes receive pixel groups during run time. A good static load balancing scheme is to randomly assign pixels to processes and this approach is in our application superior to dynamic load balancing, since the load will be partitioned into approximately equal-sized shares and no more communication is required.
However if we utilize that $\mathcal{M}$ and $\overline{\mathcal{M}}$ are connected and if we only want to display $\mathcal{M}$ without coloring its complement, then a dynamic load balancing scheme wins: we work with a master-slave architecture and begin with a single slave. Initially the slave receives a rectangle within the complex plane on which $\mathcal{M}$ is to be displayed. If the slave finds that all boundary pixels belong to $\mathcal{M}$, then it “claims” that the rectangle is a subset of $\mathcal{M}$. Otherwise the rectangle is returned to the master who partitions it into two rectangles and assigns one slave for each new rectangle. This procedure continues until all slaves are busy.

The static load balancing problem in the Mandelbrot example was easy, since the tasks—determining membership for pixels—are completely independent. However, in general, tasks depend on each other and we have to work with the task graph $\mathcal{T} = (T, E)$. The nodes of $\mathcal{T}$ correspond to the tasks and we have a directed edge $(s, t)$ from task $s$ to task $t$ whenever task $s$ has to complete before task $t$ can be dealt with. We assume an ideal situation in which we know the duration $w_t$ for each task $t$. Then we should try to partition $\mathcal{T}$ into $p$ disjoints subsets $\mathcal{T}_1, \ldots, \mathcal{T}_p$ such that processes

- carry essentially the same load, i.e., $\sum_{t \in \mathcal{T}_i} w_t \approx (\sum_{t \in \mathcal{T}} w_t)/p$, and
- communicate as little as possible, i.e., the number of edges connecting two tasks in different classes of the partition is as small as possible.

For $p = 2$ we have already encountered this problem as the problem of graph partition (GP) in Section 5.1.1. GP is $\mathcal{NP}$-complete and hence we cannot expect to find efficient algorithms which determine perfect assignments. However there are heuristics such as simulated annealing or the Kernighan-Lin heuristic which often give decent approximative results.

But often, as in our discussion of search problems, we do not even know the tasks, let alone their duration, and we have to come up with solutions on the fly. We distinguish the following scenarios for load balancing.

- **static** or **semi-static load balancing**: the task graph and the duration of tasks is known beforehand or at well-defined time steps. In all algorithms of Chapter 4 the duration of tasks was known beforehand and we have applied static load balancing by using a rowwise decomposition or checkerboard decomposition.

  We have encountered a simple version of static load balancing when approximating the Mandelbrot set. Here we have utilized that the tasks are independent. Difficult instances are solved by applying heuristics for GP.

- **dynamic load balancing**: the available information on the task graph is insufficient and task assignments have to be determined during execution. There are two major approaches.
6.1. WORK STEALING

- In **centralized load balancing** there is a centralized priority queue of tasks, which is administered by one or more masters assigning tasks to slaves (cp. the chess program APHID in Section 5.4). This approach normally assumes a small number of processes. Rules of thumb: use larger tasks at the beginning to avoid excessive overhead and smaller tasks near the end to even out the finish times; take different processor speeds into account.

- In **distributed dynamic load balancing** one distinguishes methods based on work stealing or task pulling (i.e., idle processes request work) and work sharing or task pushing (i.e., overworked processes assign work).

We concentrate on distributed dynamic load balancing, the most important scheduling problem in parallel computation, and describe work stealing as well as work sharing in some detail.

### 6.1 Work Stealing

We discuss the following three methods.

- **Random Polling**: If a process runs out of work, it requests work from a randomly chosen process.

- **Global Round Robin**: Whenever a process requests work, it accesses a *global* target variable and requests work from the specified process.

- **Asynchronous Round Robin**: Whenever a process requests work, it accesses its *local* target variable, requests work from the specified process and then increments its target variable by one modulo \( p \), where \( p \) is the number of processes.

To compare the three methods we make the following simplifying assumptions. We assume that the total work \( W \) is initially assigned to process 1. Whenever process \( i \) requests work from process \( j \), then process \( j \) donates half of its current load and keeps the remaining half. Finally we assume that \( p \) processes participate.

Our goal is to determine the number of rounds, when trying to achieve perfect parallelization, i.e., work \( O(W/p) \) for all processes.

**Random Polling**: Assume that \( j \) processes currently have work. We define \( V_j(p) \) to be the expected number of work requests such that each of the \( j \) processes with work have received at least one work request. Since the load of a process is halved after a request is served, after \( V_j(p) \) requests the peak load is halved. We show that \( V_j(p) = O(p \cdot \ln j) \). Thus, for any \( j \), the peak load is halved after \( O(p \cdot \ln p) \) requests are issued: the peak load is reduced to \( W/p \) after \( O(p \cdot \ln^2 p) \) work requests have been issued and the communication overhead is bounded by \( O(p \cdot \ln^2 p) \).
CHAPTER 6. LOAD BALANCING

Assume that currently $i$ of the $j$ processes with work have received work requests. Let $f_j(i, p)\lt/\gt$ be the expected number of work requests such that also the remaining $j-i$ processes receive requests. Then our goal is to determine $f_j(0, p)$ and we first claim

$$f_j(i, p) = \left(1 - \frac{j-i}{p}\right) \cdot (1 + f_j(i, p)) + \frac{j-i}{p} \cdot (1 + f(i+1, p)).$$

This relation is indeed true, since with probability $\left(1 - \frac{j-i}{p}\right)$ a request is not directed to one of the $j-i$ processes without a work request. Moreover, with probability $\frac{j-i}{p}$ we are successful and now $i+1$ processes have received requests. Thus we get

$$\frac{j-i}{p} \cdot f_j(i, p) = 1 + \frac{j-i}{p} \cdot f_j(i+1, p) \quad \text{and hence} \quad f_j(i, p) = \frac{p}{j-i} + f_j(i+1, p).$$

In particular we have $f(0, p) = \frac{p}{j-1} + \cdots + \frac{p}{j-i} + f_j(i+1, p)$ and hence

$$f(0, p) = p \cdot \sum_{i=0}^{j-1} \frac{1}{j-i} = p \cdot \sum_{i=1}^{j} \frac{1}{i} = \Theta(p \cdot \ln(j))$$

follows. When does random polling achieve constant efficiency?

**Exercise 78**

Assume that the expected communication time of some load balancing scheme is $C_p(W)$. Then show that the expected running time is bounded by $O\left(\frac{W}{p} + C_p(W)\right)$.

Random polling reduces the peak load below $O(W/p)$ after at most $O(p \cdot \ln^2 p) = O(p \cdot \log^2_p W)$ work requests and hence after $O(\log^2_p W)$ rounds\(^1\). We obtain constant efficiency, if the computing time $O(W/p)$ dominates and this is the case if $W/p = \Omega(\log^2_p W)$, resp. if $W = \Omega(p \cdot \log^2_p W)$.

**Global Round Robin.** To obtain a computing time of at most $O(W/p)$, the global target variable has to be accessed at least $\Omega(p)$ times. But then, to obtain constant efficiency, we have to demand that $\frac{W}{p} = \Omega(p)$ and equivalently that $W = \Omega(p^2)$ holds.

**Asynchronous Round Robin.** In the best case only $\log_p W$ rounds suffice and the requirement $\frac{W}{p} = \Omega(\log^2_p W)$ or $W = \Omega(p \cdot \log^2_p W)$ guarantees constant efficiency.

**Exercise 79**

Show that $\Theta(p)$ rounds may be required in the worst case until $\Omega(p)$ processes have work. Thus constant efficiency only holds for $W = \Omega(p^2)$.

The performance of asynchronous Round Robin is in general better than global Round Robin, since it avoids the bottleneck of a global target variable. However, to avoid its worst case, randomization and therefore random polling is preferable. We now show that random polling gives a provably good speedup for backtracking. In particular we describe which tasks a donating process should hand over to the requesting process.

\(^1\)Our analysis can be improved to show that a peak load of $O(W/p)$ can be reached after an expected number $O(\log_2 p)$ of rounds
The Speedup of Random Polling for Backtracking

We analyze random polling for backtracking under the following assumptions. An instance of backtracking generates a tree $T$ of height $h$ with $N$ nodes and degree $d$. We assume that the entire tree is to be searched with $p$ processes.

- In the beginning only process 1 is active and it inserts the root of $T$ into its initially empty stack.
- If at any time an active process takes the topmost node $v$ off its stack, then it expands $v$ and pushes all children of $v$ onto the stack. Thus the stack of a process is composed of generations with the current generation on top of the stack and the oldest generation at the bottom.
- An idle process $p$ uses random polling to request work from a randomly chosen process $q$: if $q$ itself is idle, then the request fails, otherwise $q$ serves an arbitrarily chosen request and sends one half of its oldest generation.

The following observation shows that this work transfer pays off, since nodes are only moved rarely.

**Remark 6.1** Whenever a node $v$ is donated, then it migrates together with one half of its current siblings. Hence the generation of $v$ is halved in each donation step involving $v$. Since $v$ is immediately expanded if it is the only received node, $v$ participates in at most $\lceil \log_2 d \rceil$ donation steps, where $d$ is the degree of $T$.

**Exercise 80** Show that any parallel algorithm that traverses $T$ by a depth-first traversal has to spend at least $\max\{N/p, h\}$ steps, where $N$ is the number of nodes of $T$ and $h$ the height of $T$.

We show that our implementation of random polling almost, namely up to the factor $\log_2 d$, reaches the lower time bound. Since nodes are moved only rarely, we only have to show that most of the time sufficiently many processes are busy or, equivalently, that work requests are answered with sufficient high probability.

**Theorem 6.1** With probability at least $1 - N \cdot d^{-\Omega(h+N/p)}$, random polling runs for at most

$$O(\log_2 d \cdot \max\{N/p, h\})$$

steps and hence its speedup is at least $\Omega\left(\frac{p}{\log_2 d}\right)$, whenever $h \leq N/p$.

**Proof.** If less than $p/2$ processes are idle in some given step, then we say that the step succeeds and otherwise that it fails. If a process is busy, then it participates in expanding a node or fulfilling a work request. However there are at most $O(\log_2 d \cdot N)$ such operations
and hence there are at most $O(\log_2 d \cdot N/p)$ successful steps. Thus it suffices to show that there are at most $O(\log_2 d \cdot h)$ failing steps.

We fix an arbitrary node $v$ of $T$. In any step there is a unique process which stores $v$ or the lowest, already expanded ancestor of $v$ in its stack. We say that $v$ receives a request, if “its” process receives a request for work. After at most $\lceil \log_2 d \rceil \cdot h$ requests for work, $v$ belongs to the oldest generation of its process and $v$, from this point on, continues to belong to the oldest generation until it is expanded. Hence $v$ is expanded after at most $\lceil \log_2 d \rceil$ further requests. After how many failing steps does $v$ receive $\lceil \log_2 d \rceil \cdot (h + 1)$ requests?

We determine the probability $q$ that $v$ receives a request in a failing step. In a failing step there are exactly $k$ idle processes with $k \geq p/2$. The probability that none of them requests node $v$ is therefore

$$(1 - \frac{1}{p})^k \leq (1 - \frac{1}{p})^{p/2} \leq e^{-1/2},$$

if we apply Lemma 1.2. But $q \geq 1 - e^{-1/2} \geq 1/3$. Therefore random polling performs in each failing step a random trial with success probability at least $1/3$ and the expected number of successes in $t$ trials is at least $t/3$. We apply the Chernoff bound of Theorem 1.8 and obtain

$$\Pr \left[ \sum_{i=1}^{t} X_i < (1 - \beta) \cdot t/3 \right] \leq e^{-(t/3) \cdot \beta^2/2}.$$ 

In particular, for $\beta = 1/2$, random polling has less than half the number of expected successes, namely less than $t/6$ successes in $t$ trials with probability at most $e^{-t/24}$. Set $t = 6[\log_2 d] \cdot (h + 1 + N/p)$ and there are less than $[\log_2 d] \cdot (h + 1)$ requests for $v$ with probability at most $e^{-\Omega(\log_2 d \cdot (h+1+N/p))} = d^{-\Omega(h+N/p)}$. We arrive at the claimed failure probability, since we have considered only a fixed node $v$, but have to consider all $N$ nodes.

\[\square\]

Remark 6.2 In [BL94] random polling is shown to work almost optimally for a larger class of computing problems.

6.2 Work Sharing

We compare work stealing and work sharing assuming that work assignments are determined by random strategies in both cases, i.e., that random polling is employed for work stealing and that in work sharing a busy process assigns work to a randomly chosen process.

- The more processes are busy the higher the probability that a busy process receives more work in work sharing. Even if a busy process may deny the work assignment its load increases due to the communication task. In work stealing however a busy process profits, since it gets rid of part of its work with high probability.
6.2. WORK SHARING

- The less processes are busy the higher the probability that a busy process assigns work to an idle process. With only few processes busy work sharing is more efficient in reducing peak loads.

Since normally only a minority of processes is idle, work stealing is preferable. However if urgent work is to be done and no volunteers are available, then work sharing is required. “Extreme work sharing”, i.e., assigning any subsequent task to other processors increases the communication overhead, but evens out the work load as the following example shows.

**Remark 6.3** [CRY94] consider tree-structured computations. A tree-structured computation begins with a root task. In general, the time to expand a node requires an unknown amount of time. If a node is expanded, a possibly empty set of children is generated. Load balancing is achieved by work sharing in the following way:

any process has its own pool of tasks. Whenever a process needs more work it picks a task from its pool. Any children of the currently expanded task are assigned to pools of randomly chosen processes.

Thus [CRY94] follow a “work rejection” strategy in which a process only performs the task of expanding a node, but assigns any subsequent task, i.e., children tasks to other processes. This strategy increases the communication overhead, but it guarantees with high probability that processes receive a uniform load even in the presence of tasks of different duration.

In particular, [CRY94] show that with high probability the run time with \( p \) processors is proportional to \( \frac{W}{p} + h \cdot T \), where \( W \) is the total work (i.e., the time of a sequential algorithm for the tree-structured computation), \( h \) is the depth of the tree and \( T = \max_v \frac{D(v)}{\min_v D(v)} \) is the ratio of longest and shortest duration of a node. Observe that this bound is better than the bound of Theorem 6.1 for random polling!

We begin by studying the following model scenario. We observe \( p \) clients, where the \( i \)th client assigns its task at time \( i \) to one of \( p \) servers. We assume that all tasks require the same run time, but to complicate matters we assume that the clients do not know of each others actions. We are looking for an assignment strategy that keeps the maximum load of a server as low as possible.

A first attempt, in analogy to random polling, is to randomly choose a server for each task. What is the probability \( q_k \) that some server receives \( k \) tasks? If \( k \) is not too large, then

\[
\binom{p}{k} \cdot \left(\frac{1}{p}\right)^k \cdot (1 - \frac{1}{p})^{p-k} \approx \binom{p}{k} \cdot \left(\frac{1}{p}\right)^k \approx \left(\frac{p}{k} \cdot \frac{1}{p}\right)^{\Theta(k)} = \left(\frac{1}{k}\right)^{\Theta(k)}
\]

is the probability that a fixed server receives exactly \( k \) tasks. Thus \( q_k \leq p \cdot k^{-\Theta(k)} \) and our analysis shows that \( q_k \) is small whenever \( k = \alpha \cdot \frac{\log_2 p}{\log_2 \log_2 p} \) for a sufficiently large constant.
The surprising fact is that our estimate is asymptotically tight and the server with the heaviest burden will have \( \Theta\left(\frac{\log_2 p}{\log_2 \log_2 p}\right) \) tasks with high probability and our load balancing attempt was not very successful.

Let \( d \) be a natural number. Surprisingly we can drastically reduce the maximum load with the following uniform allocation scheme:

Whenever a task is to be assigned, then choose \( d \) servers at random, enquire their respective load and assign the task to the server with smallest load. If several servers have the same minimal load, then choose a server at random.

**Theorem 6.2** There is \( \alpha > 0 \) such that, with probability at least \( 1 - p^{-\alpha} \), the maximum load of the uniform allocation scheme is bounded by

\[
\frac{\log_2 \log_2(p)}{\log_2(d)} \pm \Theta(1).
\]

As a consequence, whenever we ask two servers at random \((d = 2)\), then we reduce the maximum load from \( \Theta\left(\frac{\log_2(p)}{\log_2 \log_2(p)}\right) \) to now \( \Theta(\log_2 \log_2(p)) \). This case is also called the two-choice paradigm.

**Remark 6.4** [ACMR98] investigate when many clients submit their tasks simultaneously. Assume that each client selects two servers at random and that a server replies with the number of requests it has served so far. A client then chooses a server with the least number of requests. It is shown that the maximum load is bounded by \( O\left(\sqrt{\frac{\log_2 p}{\log_2 \log_2 p}}\right) \) with high probability. Thus the two choice paradigm still reduces the maximum load substantially, however not as strongly as in the sequential setup.

What is “behind” Theorem 6.2 for \( d = 2 \)? We define \( \beta_i \) as the number of servers with at least \( i \) tasks. Then it turns out that the \( \beta_i \) decrease quadratically, that is, \( \beta_{i+1}^2 \approx \beta_i \) holds and hence \( \beta_{\log_2 \log_2(n)} \approx 1 \) follows. The two-choice paradigm is successful, since there are only very few servers with high load and the choice of two servers with high load is therefore correspondingly small.

Can we further decrease the load with other allocation schemes? In non-uniform allocation we partition the servers into \( d \) groups of same size and assign tasks according to the “always-go-left” principle:

Choose one server at random from each group and assign a task to the server with minimal load. If several servers have the same minimal load, then choose the leftmost one.
To analyze non-uniform allocation we introduce the sequence $F_d(k)$ recursively. We set $F_d(k) = 0$ for $k \leq 0$, $F_d(1) = 1$, $F_d(k) = \sum_{i=1}^{d} F_d(k - i)$ for $k > 0$ and set

$$\phi_d = \lim_{k \to \infty} (F_d(k))^{\frac{1}{k}}.$$

**Exercise 81**

(a) Show $\phi_1 = 1$, $\phi_2 \approx 1.61$, $\phi_3 \approx 1.83$ and $\phi_4 \approx 1.92$.

(b) Show that $2^{(d-1)/d} < \phi_d < 2$ holds.

**Theorem 6.3** If we assign $p$ tasks to $p$ servers according to non-uniform allocation, then, with probability at least $1 - p^{-\alpha}$, the maximum load is bounded by

$$\frac{\log_2 \log_2(p)}{d \cdot \log_2(\phi_d)} \pm \Theta(1).$$

$\alpha$ is a positive constant.

Non-uniform allocation seems nonsensical, since servers in left groups are handicapped, but we obtain nonetheless a significant load reduction even for $d = 2$. What is the reason? If we randomly choose a server from among several servers of same minimal load and omit the always-go-left rule, then the load increases to the load of Theorem 6.2. If we select $d$ servers at random, and hence omit the group structure, than random tie breaking turns out to be optimal and again the load increases to the load of Theorem 6.2. Thus the combination of the group concept with the always-go-left rule does the trick.

Image the assignment of tasks over time. At any moment in time one has to expect that servers in left groups have to carry a larger load than servers in right groups. But then, in subsequent attempts, servers in right groups will win the new tasks and their load follows the load of servers in right groups. The combination of the group approach with always-go-left enforces therefore on one hand a larger load of left servers with the consequence that right servers have to follow suit. The preferential treatment of right groups enforces a more uniform load distribution.

**Proof of Theorem 6.2** Assume that a server carries a too high load. In the first part of our analysis we try to figure out the reason for this high load and introduce a witness tree. In particular, the probability that some server is overloaded will coincide with the probability that a witness tree of high depth exists. We then show in the second part of our analysis how to bound the probability that a witness tree of high depth exists.

**Part I: the Witness Tree.** A witness tree $W$ of depth $L$ has to show why a server carries at least $L + 4$ tasks. Suppose that server $s$ has at least $L + 4$ tasks. We then construct a witness tree $W$ recursively as follows. The root of $W$ represents the last task $t$ assigned to $x$. Any of the $d$ servers $s_1, \ldots, s_d$, chosen as potential servers for $t$ has already
received at least $L + 3$ tasks, since server $s \in \{s_1, \ldots, s_d\}$ carries $L + 4$ tasks including task $t$. We generate $d$ children of the root and let the $i$th child represent the last task assigned to $s_i$. We continue this construction recursively until all nodes correspond to leaves. Since we only expand down to depth $L$ each leaf has at least has at least four tasks.

The witness tree $W$ is completely defined once we label each node with the task it represents. We have obtained the complete $d$-ary tree of depth $L$, whose nodes are labeled with tasks. We say that the above task assignment activates $W$.

Part II: the Probability of Activating a Witness Tree. We first determine the number of different witness trees and then bound the probability that a particular tree of large depth has been activated. Let $v$ be an arbitrary node of the complete $d$-ary tree of depth $L$. We have to assign $p$ tasks and hence there are most $p$ choices for the task represented by $v$. Thus there are at most $p^m$ different witness trees, where $m = \sum_{i=0}^{L} d^i$ is the number of nodes.

Let $W$ be a fixed witness tree. We have to bound the probability that $W$ is activated. If $u$ is a child of $v$, then the server receiving the task of $u$ also competes for the task of $v$. But a fixed server competes for a particular task with probability $\left( \frac{p}{d} \right)^{-1} / \binom{p}{d} = \frac{d}{p}$. The “edge experiments” are independent and hence

\[
\left( \frac{d}{p} \right)^{m-1}
\]

is the probability that all $m - 1$ edges are “activated”.

When is leaf $b$ activated? Each of the $d$ servers competing for the task represented by $b$ has to have at least three tasks. But at any time there are at most $p/3$ servers with at least three tasks and hence $b$ is activated with probability at most $3^{-d}$. The probability of leaf activation does not increase, if other leaves have already been activated, and hence the probability that all leaves are activated is bounded by

\[
3^{-dL}
\]

from above. But then the probability that some witness tree of depth $L$ is activated is at most

\[
p^m \cdot \left( \frac{d}{p} \right)^{m-1} \cdot 3^{-dL} \leq p \cdot d^L \cdot 3^{-dL} \leq p \cdot 2^{-dL}.
\]

In the first inequality we bound the number $m - 1$ of edges by twice the number $dL$ of leaves. The second inequality follows, since $2 \cdot d^2 \leq 3^d$ holds. For

\[
L \geq \log_d \log_2 p + \log_d (1 + \alpha) = \frac{\log_2 \log_2 p}{\log_2 d} + \log_d (1 + \alpha)
\]
we get
\[ d^L \geq (1 + \alpha) \cdot \log_2 p \quad \text{und} \quad 2^{-d^L} \leq p^{-(1+\alpha)}. \]

But then the probability that some witness tree of depth $L$ has been activated is at most $p^{-\alpha}$ and hence the probability that some server receives at least
\[ L + 4 \geq \frac{\log_2 \log_2 p}{\log_2 d} \pm \Theta(1) \]
tasks is at most $p^{-\alpha}$. \qed

6.3 Conclusion

Load balancing, the task to evenly distribute work among all processes, is a fundamental problem. In static load balancing the task assignment is determined before the computation starts. Even if the duration of tasks is known beforehand, dependencies among tasks make the load balancing non-trivial and in general the load balancing problem is computationally as hard as the $\mathcal{NP}$-complete problem of graph partition.

For dynamic load balancing, tasks are assigned during run time. Random polling, i.e., requesting work from a randomly chosen process is the most successful work stealing method. Dynamic load balancing via work sharing proceeds by pushing tasks, i.e., processes with many tasks send some of their tasks to other processes. In uniform allocation a task is pushed to a process of minimum load chosen from a random sample of $d$ processes, ties are broken arbitrarily. The two-choice paradigm, i.e., setting $d = 2$, leads to a significant reduction of the maximum load and in particular the maximum load is reduced exponentially when distributing $p$ tasks among $p$ processes. A further reduction can be achieved when ties are broken non-uniformly.

Work stealing is appropriate, if relatively few processes are idle, whereas work sharing has the edge, if relatively few processes are busy.
Chapter 7

Termination Detection

The Termination Detection Problem For Active And Inactive Processes

We assume the following model: a process is either active or inactive. Whereas an active process may turn inactive, an inactive process cannot send messages and stays inactive unless it receives a message. How can we determine whether all tasks have been completed?

Observe that our load balancing schemes force idle processes to ask for work and hence these schemes do not meet the above requirements. However, if we do not consider a request for work to be a message, then the specifications of the model are met, provided an inactive process requesting work turns only active when receiving a message.

We begin with the additional assumption that messages are delivered in-order: if a process $p$ first sends a message $M_1$ directly to another process $q$ and later sends a message $M_2$ to another process which triggers a sequence of messages ending in $q$, then $M_1$ arrives at $q$ before the last message of the triggered sequence arrives at $q$. Later we describe a solution without any assumptions. Other than that we assume an asynchronous system without any restrictions. In particular, if an active process does not have a task to work on, it may or may not request a task from a colleague. A busy process on the other hand may ask a colleague to accept a subtask and suddenly new tasks may pop up.

It is not sufficient if, say, process 1 asks each process whether it is inactive, since right after a positive answer the queried process might receive a message containing a new task. In Dijkstra’s solution a token is passed around the ring of processes starting with process 1 and traveling from process $i$ to process $i+1$ and from process $p$ back to process 1. The token only leaves a process if the process turns inactive and our goal is therefore to return the token to process 1 who then should determine whether all processes are inactive. However an inactive process, and hence an already visited process, may turn active whenever it receives a message. Our algorithm has to detect the (possible) reactivation, but how?

The observation that the reactivation must have been, at least indirectly, initiated by an active and hence unvisited process is crucial: Dijkstra’s algorithm equips all processes
with a color which is initially set to white. Any process \(i\) which sends a message to process \(j\) with \(j < i\) is a suspect for reactivating a process and it turns black. If a black process receives a token, it colors the token black and hence process 1 is not forced into the wrong conclusion that all of his colleagues terminated.

**Algorithm 7.1 Dijkstra’s Token Termination Detection Algorithm I**

(1) When process 1 turns inactive, it initiates an attempt to detect termination. Process 1 turns white and sends a white token to process 2.

/* The next step is the only step that colors a process black. */

(2) If process \(i\) sends a message to process \(j\) and \(i > j\), then \(i\) turns black.

/* We demand in step (3) that a black process colors the token black and a white process does not change the color of the token. Now assume that process \(k\) is a rightmost process which changes its state from inactive to active after the token left and assume that the token is white when it left \(k\). Why does process \(k\) change its state? \(k\) passes on a white token and hence all processes to the left of \(k\) did not send messages to their left before passing on the token. Since messages and token are delivered in order, process \(k\) can only change its state due to a sequence of messages originating in a process \(l\) to its right. When process \(l\) sends its (possibly indirect) wake-up call, then \(l\) has not yet forwarded the token: remember that \(k\) is the rightmost process changing its state. But then \(l\) is black before it passes the token and hence it passes on a black token. Consequently process 1 does not incorrectly decide upon termination. */

(3) Assume that process \(i > 1\) has just received the token. It keeps the token as long as it is active. If it turns inactive, it colors the token:

If \(i\) is black, then the token turns black. Otherwise, if \(i\) is white, the color of the token is unchanged.

Process \(i\) passes the token along to process \(i + 1\), respectively to process 1, if \(i = p\). Afterwards it turns white.

/* The process turns white to get ready for the next round. */

**Theorem 7.2** Assume that messages are delivered in order. If process 1 receives a white token from process \(p\), then all processes are inactive.

**Exercise 82**
Verify Theorem 7.2.
Assume that all processes are inactive before the token is inserted into the ring and no messages are pending. Then it may still be the case that a process is colored black, but all processes are colored white after one round and only an additional round determines termination.

Observe that the token might be on its way for an unpredictable length of time, however the token itself consumes at most time $O(p)$ due to the “handshakes” (send/receive operations) between adjacent processes on the ring. Moreover, process 1 does not have to wait for the token to return, but may turn active again when receiving a message. Thus the token is nothing else but an inexpensive background process which guarantees that termination is quickly recognized.

Now assume that process 1 recognizes that all processes are inactive. Does it suffice to broadcast a shutdown message and execute MPI\_Finalize? It does suffice provided all messages sent to process 1 have been delivered. If, say, process $i$ has sent a message to process 1, then $i$ turns black and will color the token black. When the token arrives at process 1 the message sent by process $i$ must have arrived as well, provided messages are delivered in order. With the same reasoning any process $i$ may execute MPI\_Finalize after receiving the shutdown broadcast from process 1, provided messages are delivered in order.

MPI guarantees that messages are non-overtaking: if a process sends two messages to another process, namely first message $M_1$ and subsequently message $M_2$, then $M_1$ will be received before $M_2$ is received. However observe that MPI does not guarantee that messages are delivered in-order. Thus Algorithm 7.1 is inadequate, since undelivered messages may describe new tasks. In our new approach we equip all processes additionally with a message count. Initially all processes are white and their message count is zero. Whenever a process receives a message it decrements its message count and increments its count if it sends a message. We utilize that the sum of message counts is zero iff all messages have been delivered and use the token to sum message counts. But we do not have access to all message counts simultaneously and we have to worry that an already processed message count might not be accurate.

**Algorithm 7.3 Dijkstra’s Token Termination Detection Algorithm II**

1. When process 1 becomes inactive and when the token has returned, the process initiates an attempt to detect termination. Process 1 turns white and sends a white token with its overall message count to process 2.
   /* Our goal is to enforce that all processes are inactive and no messages are pending provided the token is white and its message count is zero.

   The message counts of a process adds all messages sent and subtracts all messages received. */

2. If a process $i$ sends or receives a message, then $i$ turns black.
   /* If the token is white when finally reaching process 1, then the token has passed only white processes and neither process has sent or received a message between the
last two successive visits of the token. Thus message counts did not change! It may however happen that messages are still in flight, but then the token’s message count will be non-zero. */

(3) Assume that process \( i > 1 \) has just received the token. If \( i \) is black, then the token turns black. Otherwise, if \( i \) is white, the color of the token is unchanged.

Process \( i \) keeps the token as long as it is active. If it turns inactive, process \( i \) adds its message count to the message field and passes the token along to process \( i + 1 \), respectively to process 1, if \( i = p \). Afterwards process \( i \) turns white.

**Theorem 7.4** Assume that process 1 is white, when it receives a white token from process \( p \) and that the count field of the token is zero. Then all processes are inactive and there are no more messages in the system. Hence process 1 may send a shut down message to all processes.

**Exercise 83**
Verify Theorem 7.4.

**Exercise 84**
Consider Algorithm 7.3.

(a) Does it suffice in step (2) that a process \( i \) is colored black only if it sends a message to a process \( j \) with \( j < i \)?

(b) Does it suffice in step (2) that a process \( i \) is colored black only if it sends a message?

**Termination Detection In Dynamic Load Balancing**

We **first** try a simple, but incorrect approach. If a process \( A \) finds a solution during Backtracking, then it broadcasts a shutdown message to all of its colleagues and executes MPIFinalize. To halt computation as soon as possible, each process periodically checks with MPIIprobe whether it has received a shutdown message and if so, executes MPIFinalize. If process \( B \), before receiving the shutdown message from \( A \), detects a solution by itself, then it broadcasts its own stop message. This message cannot be delivered to process \( A \) resulting in a run-time error.

Our **second** successful approach utilizes that we perform dynamic load balancing by requesting work. Process 1 is distinguished as the process which initially carries the entire load. We initially assign weight 1 to process 1 and weight 0 to the remaining processes. Whenever process \( i \) requests and receives work from process \( j \), it also receives the weight \( w_j/2 \) from process \( j \), where \( w_j \) is the current weight of process \( j \). Thus, after this transfer, we set \( w_i = w_j/2 \) and \( w_j = w_j/2 \). Finally, whenever a process finishes its (current) task and the entire weight it handed out during execution of the task has been returned, it hands its weight back to the donating process. The donor then adds the received weight to its current weight. If process 1 has received weight 1 again, it knows that all tasks have
completed. To terminate the computation, process 1 sends a “a shutdown token” into the process ring and determines the difference between sent and received requests. The token is recycled for so long until difference zero is determined: at this time all processes are inactive and all messages are delivered.

Observe that only powers $2^{-k}$ change hands and if $k$ is too large, then the weight may be identified as zero. There is however no problem, if the power $k$ is exchanged instead of $2^{-k}$.

Our third approach uses Dijkstra’s algorithm. We say that a process is active if it has a task to work on and inactive otherwise. Only assignments of work are considered to be messages; in particular requests for work are not considered messages. We now may use Algorithm 7.3, since an inactive process stays inactive unless receiving a message. When process 1 receives an OK to terminate it knows that all processes are inactive and hence all work has been completed. Now the shutdown token prepares the actual shutdown.

**Exercise 85**

Here are two more procedures to detect termination. Are they correct?

(a) Process 1 inserts a token into the process ring. A process keeps the token as long as it has work to do and passes it along otherwise. Process 1 prepares the shutdown whenever receiving the token from process $p$.

(b) Process 1 inserts a white token into the process ring. A process colors the token black if it has work to do and passes it along. An idle process passes the token along without changing its color. Process 1 prepares the shutdown whenever receiving a white token from process $p$.

**Exercise 86**

(a) The weight procedure for termination detection in dynamic load balancing has to be modified to allow early termination when a process finds a solution. How should that be done?

(b) Does the adaption of Dijkstra’s algorithm has to be modified?
In an application of divide & conquer to solve a problem $P$, we have to solve a hierarchy of subproblems. This hierarchy corresponds to a tree $T_P$ of subproblems: the root represents $P$ and a node $u$ is a child of $v$ iff the subproblem of $u$ is used when solving the subproblem of $v$. Finally leaves are directly solvable subproblems.

Observe that the subproblems represented by children of a node $v$ can be solved independently. Therefore we obtain a parallelization of a sequential divide & conquer algorithm whenever the task of solving the subproblem of a node $v$, given solutions for the subproblems of its children, can be parallelized. We have already seen a first example when parallelizing the fast Fourier transform. Mergesort and quicksort turn out to be two further examples.

### 8.1 Sorting

We assume throughout that $n$ keys are equally distributed among $p$ processes. We could have required that all keys initially belong to one process who is then supposed to distribute the keys among its colleagues. However this convention does not allow to show the difference in speed between different sorting procedures, since the distribution task is inherently sequential and requires linear time $\Omega(n)$. With the same argumentation we do not require to output the sorted sequence to a distinguished process, but instead that the keys within each process are sorted and that all keys of process $i$ are not larger than any key for process $i+1$.

We have already discussed odd-even transposition sort (OETS) which sorts $n$ keys with $p$ processes in time $O(\frac{n}{p} \cdot \log \frac{n}{p} + n)$. The performance is not convincing, since the compute/communicate ratio is very poor in each phase and communication dominates each merging effort. Moreover, OETS is efficient only if we have few, i.e., $p = O(\log_2 n)$
processors. Parallelizations of quicksort turn out to be superior.

8.1.1 Parallelizing Quicksort

The sequential quicksort determines a splitter (or pivot) \( x_i \), rearranges the input sequence into a subsequence of keys \( x_j \) with \( x_j \leq x_i \) and into a subsequence of keys \( x_j \) with \( x_j > x_i \) and then applies quicksort recursively to both subsequences. Thus we have to pay particular attention to the rearrangement step, since the two recursive calls can be parallelized.

**Algorithm 8.1 A parallel quicksort.**

/* We sort the sequence \( x \) of \( n \) pairwise distinct keys with \( p \) processes. Processor \( i \) receives the \( i \)th interval of \( X \) of length \( \frac{n}{p} \). */

if \( n = 1 \) then stop; else

(1) Each process determines a local (approximate) median. Processor 1 executes a Gather operation, determines a global approximate median \( M \) and broadcasts \( M \) to its colleagues.

/* Time \( O\left(\frac{n}{p}\right) \) and communication \( O(p + \log_2 p) = O(p) \) suffice. */

(2) for \( i = 1 \) to \( p \) pardo: process \( i \) partitions its sequence according to \( M \);

/* Time \( O\left(\frac{n}{p}\right) \) without any communication is sufficient. */

(3) Apply MPI\_Scan to determine the two prefix sums \( \sum_{j=1}^{i} \text{smaller}_j \) and \( \sum_{j=1}^{i} \text{larger}_j \), where \( \text{smaller}_j \) (resp. \( \text{larger}_j \)) is the number of keys of process \( j \) which are smaller (resp. larger) than \( M \).

/* The communication cost is bounded by \( O(\log_2 p) \). */

(4) The number \( k \) of keys smaller than \( M \) is broadcasted. The “first” \( \frac{nk}{n-1} \) processes will be responsible to recursively sort the smaller keys, the remaining processes are responsible to recursively sort the larger keys.

Determine the new key positions and send the keys to their new process;

/* The local computation time is \( O\left(\frac{n}{p}\right) \). The communication is bounded by \( O\left(\frac{n}{p}\right) \), since each process has to send up to \( \frac{n}{p} \) keys. */

(5) Apply the sequential quicksort whenever only one process is assigned. Otherwise recursively sort the first \( k - 1 \) keys and the last \( n - k \) keys in parallel.

If we determine exact local medians in step (1), then the approximate median \( M \) is larger than one half of all local medians. Therefore, by definition of a local median, \( M \) is larger than one fourth of all keys and, with the same argument, smaller than one fourth of all keys. Since we stop the recursion when only one process is responsible, Algorithm 8.1 requires at most \( \log_{4/3} p \) iterations. Finally observe that the time for computation as well as communication is bounded by \( O\left(\frac{n}{p} + p\right) \).
8.1. SORTING

We maintain throughout with the help of step (4) that a process is responsible for roughly \( \frac{n}{p} \) keys, since keys are evenly distributed among available processes. All in all we have a computing time of

\[
O((\frac{n}{p} + p) \cdot \log_{4/3} p + \frac{n}{p} \cdot \log_2 \frac{n}{p}) = O(\frac{n}{p} \cdot \log_2 n + p \cdot \log_2 p),
\]

since \( O(\frac{n}{p} \cdot \log_2 \frac{n}{p}) \) is the time to sort sequentially when recursion is stopped in step (6).

The contribution of \( p \log_2 p \) is used when processes send their local medians to process 1 in step (1). If we only perform random splitter selection, then the communication cost of \( p \) per round can be replaced by the communication cost of \( \log_2 p \) per round (for broadcasting splitter \( M \) and for performing MPI Scan in step (3)).

**Theorem 8.2** Parallel quicksort sorts \( n \) keys in time

\[
O(\frac{n}{p} \cdot \log_2 n + p \cdot \log_2 p).
\]

The randomized version sorts \( n \) keys in expected time

\[
O(\frac{n}{p} \cdot \log_2 n + \log^2 p).
\]

Hence the deterministic quicksort achieves constant efficiency whenever \( p \leq \sqrt{n} \), whereas the randomized quicksort is already optimal whenever \( p \leq n / \log_2 n \). However we should not be satisfied, since the compute/communicate ratio per iteration is constant and hence too large.

**Remark 8.1 Hyperquicksort**

Here we do not attack the real problem of extensive communication, but discuss some potential computational improvements. In particular, we study the effect of the following modifications:

- Now we require that all processes initially sort their \( \frac{n}{p} \) keys. This action has two consequences. The first positive consequence is that determining the local splitter has become trivial. Moreover, and this is the second positive consequence, if we require that all processes finish an iteration by sorting their keys again, then there is no need for the sequential sort in step (5) of the parallel quicksort. Certainly the main question is the computational cost of sorting in each iteration.

- Steps (3) and (4) implement the rearranging step and simultaneously perform some load balancing. Instead we completely give up on load balancing. Imagine that \( p \) is a power of two and that the \( p \) processes form a hypercube. Then it is natural to demand that all processes 0\( w \) (resp. 1\( w \)) are responsible to work on all elements smaller (resp. larger) than the first splitter \( M \). Hence if process 0\( w \) receives the splitter \( M \), it sends its keys larger than \( M \) to process 1\( w \) and receives keys smaller than \( M \) in return. Thus rearrangement is performed by a single communication step on the hypercube, explaining the name hyperquicksort. As an additional profit each process can apply a single merging step to sort the sorted subsequence of its remaining keys and the received sorted subsequence in linear time.
In effect a process in hyperquicksort trades the time to compute a local median and to rearrange its keys against sorting its keys all the time, but gives up on any serious load balancing. Thus hyperquicksort is easier to implement than the parallel quicksort of Algorithm 8.1. However, the missing load balancing significantly degrades its performance and it does not attack the real problem of extensive communication.

Hyperquicksort has been used for the Intel iPSC and the nCUBE/ten which organize processors in a hypercube architecture.

### 8.1.2 Sample Sort

Our goal is to compress the $O(\log_2 p)$ iterations of quicksort into essentially one phase. This is accomplished by selecting a sample of $p - 1$ splitters.

**Algorithm 8.3 Sample sort.**

/* We sort the sequence $x$ of $n$ pairwise distinct keys with $p$ processes. Processor $i$ receives the $i$th interval of $X$ of length $n/p$. */

1. Each process sorts its $n/p$ keys sequentially and determines a sample of size $s$ that it sends to process 1, who then executes a Gather operation.  
   /* Time $O(n/p \log_2 n/p)$ and communication $O(p \cdot s)$ suffice. */

2. Processor 1 sorts the received $(p - 1) \cdot s$ keys and computes the final sample $S$ of size $p - 1$ with the help of its keys. Finally it broadcasts $S$.  
   /* $O(ps \log_2 ps)$ computing steps and $O(p \log_2 p)$ communication steps suffice. */

3. Each process partitions its keys according to the sample $S$. Then an all-to-all personalized broadcast is applied in which a process earmarks each of its $p - 1$ sorted subsequences for a particular colleague.  
   /* Time $O(p \cdot \log_2 \frac{n}{p})$ suffices. The communication cost is bounded by $O(\frac{n}{p})$ in the best case: if the length of subsequences is not asymptotically related, then the all-to-all broadcast may take longer. */

4. Each process merges the $p$ sorted sequences and is done.  
   /* Time $O(\frac{n}{p} \cdot \log_2 p)$ is enough. Why? */

The computing time is at most $O(\frac{n}{p} \cdot \log_2 \frac{n}{p} + p \cdot \log_2 \frac{n}{p} + \frac{n}{p} \cdot \log_2 p) = O(\frac{n}{p} \log_2 n + p \cdot \log_2 \frac{n}{p})$ steps. (Here we assume that $p \cdot s \leq n/p$ and equivalently that $s \leq n/p^2$ holds.) The time for communication is bounded by $O(p \cdot s + p \cdot \log_2 p + \frac{n}{p}) = O(p \cdot \log_2 p + \frac{n}{p})$, again since $p \cdot s \leq n/p$. 
Theorem 8.4 Sample sort runs in time \( O\left(\frac{n}{p} \cdot \log_2 n + p \cdot \log_2 \frac{n}{p}\right) \) and with communication cost at most \( O(p \cdot \log_2 p + \frac{n}{p}) \), when sorting \( n \) keys with \( p \) processors. If \( p \leq \sqrt{\frac{n}{\log_2 n}} \), then computing time is bounded by \( O\left(\frac{n}{p} \cdot \log_2 n\right) \) and communication time is bounded by \( O\left(\frac{n}{p}\right) \).

Exercise 87 We consider the following procedure to determine a set of \( p-1 \) good splitters. After sorting its \( \frac{n}{p} \) keys a process selects keys in position \( i \cdot \frac{n}{p^2} \) for \( i = 0, \ldots, p-1 \) as its sample and sends this sample to process 1, who then sorts all \( p^2 \) keys. Then process 1 computes the final sample \( S \) by selecting all keys in positions \( i \cdot p \) for \( i = 1, \ldots, p-1 \) and broadcasts \( S \). Determine the maximal number of keys received by a process in step (4) of sample sort.

Hence sample sort performs successful load balancing if we set \( s = p \). In this case one should demand \( n = \Omega(p^3) \), since otherwise sorting all samples dominates the run time.

Sample sort has the least communication if we use parallel quicksort and hyperquicksort as a comparison. If the all-to-all broadcast in step (3) runs in time \( O\left(\frac{n}{p}\right) \), then sample sort is the superior sorting procedure. In a sense, the algorithms for the fast Fourier transform respectively for parallel quicksort seems comparable with parallel quicksort playing the role of the binary exchange algorithm and sample sort playing the role of the transpose algorithm.

8.1.3 Parallelizing Mergesort

Let \( n \) be a power of two. The sequential, non-recursive mergesort works in \( \log_2 n \) phases where within phase \( i \) it merges consecutive intervals of length \( 2^i \).

Algorithm 8.5 The sequential, non-recursive mergesort.

```c
/* The sequence \( x = (x_0, \ldots, x_{n-1}) \) is to be sorted. */

for i = 0 to \( \log_2 n - 1 \) do
    for j = 0 to \( \frac{n}{2^{i+1}} - 1 \) pardo
        merge \((x_j, 2^i, \ldots, x_{(j+1) \cdot 2^i - 1})\) and \((x_{(j+1) \cdot 2^i}, \ldots, x_{(j+2) \cdot 2^i - 1})\);
```

In our parallel implementation we work with \( p \) processes who start by sorting their \( n/p \) keys. Then the final \( \log_2 p \) merging phases have to be parallelized and the crucial question is therefore the performance of merging in parallel. We first describe odd-even merging.

Exercise 88 Let \( m \) be a power of two. For a sequence \( x = (x_0, \ldots, x_{m-1}) \) let even\((x) = (x_0, x_2, x_4, \ldots, x_{m-2}) \) and odd\((x) = (x_1, x_3, x_5, \ldots, x_{m-1}) \) be the subsequences of even- and odd-numbered components of \( x \).

Assume that \( x \) and \( y \) are sorted sequences of length \( m \) each. To merge \( x \) and \( y \), odd-even Merge recursively merges
- even\((x)\) and odd\((y)\) to obtain the sorted sequence \(u = (u_0, u_1, \ldots, u_{m-1})\) and
- odd\((x)\) and even\((y)\) to obtain the sorted sequence \(v = (v_0, v_1, \ldots, v_{m-1})\).

Finally odd-even merge computes the output sequence \(w = (w_0, \ldots, w_{2m-1})\) from \(u\) and \(v\) as follows: in parallel for \(0 \leq i \leq m - 1\), if \(u_i \leq v_i\) then set \(w_{2i} = u_i, w_{2i+1} = v_i\) and otherwise \(w_{2i} = v_i, w_{2i+1} = u_i\).

(a) Show with the 0-1 principle that odd-even merge works correctly.

(b) Assume that consecutive intervals of \(x\) and \(y\) of length \(m/2q\) each– are distributed among \(q\) processes. Show how to implement odd-even merge in computing time \(O\left(\frac{n}{q} \cdot \log_2 2q\right)\) and communication time \(O\left(\frac{n}{q} \cdot \log_2 q\right)\).

Thus, after sorting \(n/p\) keys sequentially in time \(O\left(\frac{n}{p} \cdot \log_2 \frac{n}{p}\right)\) without communication, sorted sequences of length \(n/p\) are merged. This phase is followed by merging sorted sequences of length \(2n/p\) and in general by merging sorted sequences of length \(2^k \cdot n/p\) for \(k = 0, \ldots, \log_2 p - 1\). If we fix \(k\), then we have to solve \(p/2^{k+1}\) merging problems involving sorted sequences of length \(m = 2^k \cdot n/p\). In any such merging problems \(q = 2^{k+1}\) processes participate and hence computing time \(O\left(\frac{n}{q} \cdot \log_2 2q\right) = O\left(\frac{n}{p} \cdot (k + 1)\right)\) and communication time \(O\left(\frac{m}{q} \cdot \log_2 q\right) = O\left(\frac{n}{p} \cdot (k + 1)\right)\) suffice. All in all,

**Theorem 8.6** Odd-even merge sort runs in time \(O\left(\frac{n}{p} \cdot \log_2 p + \frac{n}{p} \cdot \log_2 \frac{n}{p}\right)\) and its communication cost is bounded by \(O\left(\frac{n}{p} \cdot \log_2^2 p\right)\).

Thus we cannot expect good performance, unless communication is dominated by computation, and hence \(\log_2^2 p << \log_2 \frac{n}{p}\) is required. In conclusion, if the number of processors is too large, the performance of the parallel merge sort breaks down due to extensive communication requirements. If the number of processors is sufficiently small, the parallel merge sort is doing ok, but is in now way better than a parallel quicksort with similar resources. We make a second attempt and change our merging approach. The next problem proposes the bitonic merge algorithm.

**Exercise 89**
Let \(m\) be a power of two. A sequence \((x_0, \ldots, x_{m-1})\) is called bitonic iff there is a position \(i\) such that \((x_0, \ldots, x_i)\) is monotonically increasing and \((x_{i+1}, \ldots, x_{m-1})\) is monotonically decreasing.

We describe bitonic merge to merge an increasing sequence \((x_0, \ldots, x_{m-1})\) with a decreasing sequence \((x_m, \ldots, x_{2m})\). A single comparison suffices for \(m = 1\). In the general case compare \(x_i\) with \(x_{i+m}\) in parallel for all \(1 \leq i \leq m\). After switching keys between the two halves if necessary, bitonic merge merges the first and the second half recursively.

(a) Show that bitonic merge works correctly.

(b) Show that we obtain a sorting algorithm as follows: recursively sort the first half of all keys increasingly and the second half decreasingly. Merge the two sequences with bitonic merge. Determine the computing time and the communication cost when sorting a sequence of \(m\) keys with \(q\) processors.

(c) Is there a performance difference between sorting with odd-even merging and sorting with bitonic merging?
8.2. Dynamic Programming

Dynamic programming is a generalization of divide & conquer in which the hierarchy of subproblems corresponds to a directed graph without cycles. Typically, subproblems have few different degrees of complexity and subproblems of same complexity can be solved independently. Thus, as for divide & conquer, a successful parallelization of a sequential dynamic programming algorithm normally depends only on parallelizing the task of solving the subproblem of a node from solutions to subproblems of its immediate ancestors.

Dynamic programming has gained importance due to many applications within bioinformatics.

8.2.1 Transitive Closure and Shortest Paths

In the transitive closure problem we are given a directed graph $G = (V, E)$ with node set $V = \{1, \ldots, n\}$. Our goal is to determine the transitive closure graph $\bar{G} = (V, \bar{E})$, where the edge $(i, j)$ belongs to $\bar{E}$ iff there is a path in $G$ starting in $i$ and ending in $j$. Warshall’s algorithm determines the transitive closure graph with the help of the adjacency matrix $A$, where $A$ is defined by

$$A[i, j] = \begin{cases} 1 & (i, j) \in E \\ 0 & \text{otherwise.} \end{cases}$$

**Warshall’s Algorithm**

for $k=1$ to $n$ do
  for $i=1$ to $n$ do
    for $j=1$ to $n$ do

Why is Warshall’s algorithm correct? An inductive argument shows that, when the iteration for $k$ terminates,

$$A[i, j] = 1 \iff \text{there is a path in } G \text{ from } i \text{ to } j \text{ whose intermediate nodes all belong to } \{1, \ldots, k\}.$$ 

Observe that Warshall’s algorithm consists of three nested for-loops and hence runs in time $O(n^3)$. 
We describe a parallel algorithm which maintains the sequential outer $k$-loop, but parallelizes the inner $i, j$-loops. We work with the rowwise decomposition of $A$ and assume that all rows with indices in the interval $\{(i-1) \cdot n/p + 1, \ldots, i \cdot n/p\}$ are assigned to process $i$. Assume that we have reached $k$. All updates $A[i,j] = A[i,j] \lor (A[i,k] \land A[k,j])$ have to be performed in parallel. If a process knows the $i$th row, then it is responsible for the update of $A[i,j]$. The process does know $A[i,k]$, but it may have no information on $A[k,j]$. Thus the process holding row $k$ should broadcast it. Observe that the iteration for $k$ requires computing time $O(n^2 p)$ and broadcast time $O(n \cdot \log_2 p)$. The overall running time is therefore bounded by $O(n^3 p + n^2 \log_2 p)$ and we reach constant efficiency, if $n = \Omega(p \cdot \log_2 p)$ holds.

Let us also try the checkerboard decomposition, where processes form a $\sqrt{p} \times \sqrt{p}$ mesh and each process holds a submatrix with $n/\sqrt{p}$ rows and $n/\sqrt{p}$ columns. The process responsible for $A[i,j]$ may not know $A[i,k]$ or $A[k,j]$. Thus the process holding $A[i,k]$ (resp. $A[k,j]$) has to broadcast its values in its row (resp. column) of the mesh of processes. The communication time per $k$-iteration is bounded by $O(n^{3/2} \sqrt{p})$, whereas the computation time stays at $O(n^{3/2} \sqrt{p})$. We have $n$ $k$-iterations and the performance has improved to running time $O(n^3 p + n^2 \sqrt{p} \log_2 \sqrt{p})$. Constant efficiency is reached, whenever $n = \Omega(\sqrt{p} \cdot \log_2 \sqrt{p})$.

**Theorem 8.7** Assume that $p$ processors utilize the checkerboard decomposition. Then the transitive closure of a given directed graph with $n$ nodes can be determined in computing time $O(n^3 p + n^2 \sqrt{p} \log_2 \sqrt{p})$ and communication time $O(n^3 p + n^2 \sqrt{p} \log_2 \sqrt{p})$.

In the all-pairs-shortest-path problem we are given a directed graph $G = (V, E)$ with node set $V = \{1, \ldots, n\}$. We assign weights $w(e)$ to the edges $e \in E$ and would like to determine the length of a shortest path from $i$ to $j$ for any pair $(i, j)$ of nodes. Floyd’s algorithm solves the all-pairs-shortest-path problem with the help of the weighted adjacency matrix

$$B[i,j] = \begin{cases} w(i,j) & (i,j) \in E \\ \infty & \text{otherwise.} \end{cases}$$

**Floyd’s Algorithm**

for $k=1$ to $n$
  for $i=1$ to $n$
    for $j=1$ to $n$
      $B[i,j] = \min (B[i,j], B[i,k] + B[k,j])$;

As for Warshall’s algorithm one shows with an inductive argument that, when the iteration for $k$ terminates,

$$B[i,j] = \text{the length of a shortest path in } G \text{ from } i \text{ to } j \text{ whose intermediate nodes all belong to } \{1, \ldots, k\}.$$
Since Floyd’s algorithm also consists of three nested for-loops, it runs in time $O(n^3)$ as well. Observe that we can apply the parallelization of Warshall’s algorithm with the obvious changes and obtain a parallelization of Floyd’s algorithm.

**Theorem 8.8** Assume that $p$ processors utilize the checkerboard decomposition. Then the all-pairs-shortest-path problem of a given directed graph with $n$ nodes can be solved in computing time $O\left(\frac{n^3}{p} + \frac{n^2}{\sqrt{p}} \log_2 \sqrt{p} + n\right)$ and communication time $O\left(\frac{n^2}{\sqrt{p}} \log_2 \sqrt{p}\right)$.

**Remark 8.2** Thus we have obtained efficient parallelizations for Warshall’s and Floyd’s algorithm. These solutions are satisfying for dense graphs with $\Omega(n^2)$ edges. For sparse graphs however we obtain a faster transitive closure computation, if we run a depth-first search beginning in each node (running time $O(n^2 + n \cdot |E|)$), and a faster all-pairs shortest path solution, if we run Dijkstra’s single-source-shortest-path algorithm for each node in $V$ (running time $O((n^2 + n \cdot |E|) \cdot \log_2 n)$). Efficient parallelizations are neither known for depth-first search nor for Dijkstra’s algorithm!

**Exercise 90**
Assume we are given an $n \times n$ matrix (or image) $M$ of zeroes and ones. If $(i,j), (k,l)$ are neighbors in $M_2(n)$ or if $(i,j), (k,l)$ are neighbors via diagonal edges, then we say that $(i,j), (k,l)$ are connected, whenever $M[i,j] = M[k,l] = 1$. The connected component of $(i,j)$ consists of all pairs $(r,s)$ which are connected with $(i,j)$. Observe that an image decomposes into disjoint connected components.

We describe an algorithm to determine all connected components of a given image. Only processes $(i,j)$ with $M[i,j] = 1$ are active. First any active process $(i,j)$ determines the label $n \cdot i + j$ as well as the set of neighbors that it is connected to. In subsequent steps active processes reset their label to the smallest label among active processes that they are connected to and repeat this procedure “sufficiently long”.

Show that the minimization step may have to be repeated $\Omega(n^2)$ times.

We say that an image is convex iff $M$, when restricted to rows or columns, has the form $0^* \cdot 1^* \cdot 0^*$. Show that the algorithm runs for at most $O(n)$ steps whenever the image is convex.

**Exercise 91**
We describe an algorithm for determining all connected components of a two-dimensional $n \times n$ grid of zeroes and ones. We say that two grid points $(i,j)$ and $(k,l)$ with label one are connected iff $|i-k| + |j-l| \leq 1$ or if $(i,j)$ and $(k,l)$ are neighbors via diagonal edges. We allow ones to change to zeroes and vice versa according to the following transition rules for the upper left cell:

\[
\begin{array}{c|c}
1 & 0 \\
0 & 0
\end{array} \Rightarrow \begin{array}{c|c|c}
0 & 0 & * \\
* & & *
\end{array} \quad \text{and} \quad \begin{array}{c|c|c}
0 & 1 & 0 \\
1 & & *
\end{array} \Rightarrow \begin{array}{c|c|c}
1 & * & * \\
* & & *
\end{array}
\]

(Cells beyond the boundary are set to zero by default.) Show the following properties for the transition from the image at time $t$ to the image at time $t + 1$:

(a) A connected component never decomposes into two or more connected components.
(b) Two or more components are never merged into one component.
(c) No ones exist after $2n$ steps.
(d) Just before extinction a connected component consists of exactly one cell.

Show how to determine all connected components of an image in time $O(n)$.  

8.2.2 The Global Pairwise Alignment Problem

Quite often similarities of DNA, RNA sequences or proteins imply functional similarity. However point mutations such as insertions deletions or substitution of nucleotides makes a direct comparison of such sequences an apparently complex problem.

Let’s have a more detailed look. We view a DNA sequence as a word over the alphabet $\Sigma = \{\text{adenine, cytosine, guanine, thymine}\}$. We would like to determine how many insertions, deletions or substitutions of letters are necessary to obtain sequence $v$ from sequence $u$. Instead of tackling this problem right away we choose a slightly different perspective and imagine a blank symbol “−” inserted in several positions of $u$ as well as $v$. The new, longer strings $u^*$ and $v^*$ are required to have same length and the blank symbol is not allowed to appear in the same position for $u$ as well as for $v$. Is this the case, then $u^*$ and $v^*$ are called an alignment of $u$ and $v$.

If $u^*_i = −$, then we imagine $v^*_i$ as inserted into $u$, and if $v^*_i = −$, then we imagine that $u^*_i$ has been deleted. Finally, if both $u^*_i \neq v^*_i$ are different from the blank symbol, then we say that $u^*_i$ has been substituted by $v^*_i$. Of course we are looking for an alignment which verifies a maximal similarity between $u$ and $v$.

**Definition 8.9** The similarity of an alignment $u^*, v^*$ of $u$ and $v$ is defined by

$$s(u^*, v^*) = \sum_i d(u^*_i, v^*_i),$$

where the function $d : (\Sigma \cup \{-\}) \times (\Sigma \cup \{-\}) \rightarrow \mathbb{R}$ penalizes a disagreement with a low or negative score.

The algorithm of Needleman-Wunsch determines an alignment of maximum score for two strings $u$ and $v$ of length $n$ and $m$ respectively. It does so by computing a maximum alignment for the prefixes $u_1 \cdots u_i$ of $u$ and $v_1 \cdots v_j$ of $v$. Here is the crucial observation: if $D(i, j)$ is the maximum score, then

$$D(i, j) = \max\{D(i, j - 1) + d(−, v_j), D(i - 1, j) + d(u_i, −), D(i - 1, j - 1) + d(u_i, v_j)\}.$$

Why? An optimal alignment may match match $v_j$ with a blank, $u_i$ with a blank or match $u_i$ and $v_j$. In either case the alignment of the remaining strings has to be optimal.

**Algorithm 8.10 Global Pairwise Alignment**

1. // Initialization starts.
   
   $D(0, 0) = 0$;
   
   for $(i = 1; i \leq n; i + +)$
   
   $D(i, 0) = \sum_{k=1}^i d(u_k, −)$;
   
   for $(j = 1; j \leq m; j + +)$
   
   $D(0, j) = \sum_{k=1}^j d(−, v_k)$;
(2) for \((i = 1; i <= n; i += 1)\)

\[
\text{for } (j = 1; j <= m; j += 1) \quad // \text{The subproblem to determine } D(i, j) \text{ is solved:}
\]

\[
D(i, j) = \max \{ D(i, j-1) + d(-, v_j), D(i-1, j) + d(u_i, -), D(i-1, j-1) + d(u_i, v_j) \}.
\]

The run time of algorithm 8.10 is dominated by the two nested for-loops and hence is proportional to \(O(n \cdot m)\). Observe that to determine \(D(i, j)\) we only need to know \(D(i', j')\) for \(i' + j' < i + j\). Hence we reorganize step (2) of algorithm 8.10 by working with an outer \(k\)-loop and compute \(D(i, k-i)\) with an inner \(i\)-loop. Thus our parallelization works sequentially on the outer \(k\)-loop and parallelizes the inner loop.

In a typical application for the global alignment problem both strings \(u\) and \(v\) have approximately the same length. To simplify the exposition we assume that both strings have identical length \(n\). We choose the rowwise decomposition of the score matrix \(D\). Our parallel implementation works in \(2n\) phases and computes in phase \(k\) all entries \(D(i, k-i)\).

If a process becomes active, it evaluates its portion of the score matrix \(D\). The process begins with boundary pairs \((i, k-i)\) whose solution it immediately communicates to the respective neighbor process and finishes with interior pairs. Each phase requires the communication of two boundary pairs and can be completed in time \(O(n^2)\). Hence the total run time is bounded by \(O(n^2/p + n)\).

Exercise 92
What is the run time when choosing the checkerboard decomposition?

8.2.3 RNA Secondary Structure Prediction

RNA (ribonucleic acid) is involved in the production of proteins either as information carrier (mRNA) or by translating this information into proteins (rRNA, tRNA). Unlike DNA molecules which are composed of two strands of complementary nucleotides, most RNA molecules possess only a single strand. RNA is made up of four bases, namely the complementary pairs adenine and uracil as well as cytosine and guanine; thus uracil replaces thymine in comparison to DNA molecules.

The RNA secondary structure consists of all hydrogen bonds between complementary bases within the molecule. This two-dimensional folding pattern is of particular importance for the functionality of an RNA molecule.

Definition 8.11 Let \(R \in \{A, C, G, U\}^n\) be the primary structure of an RNA molecule. Its secondary structure is described by a set \(P \subseteq \{ \{i, j\} \mid 1 \leq i \neq j \leq n \} \) of pairs satisfying the following conditions:

(a) For any pair \(\{i, j\} \in P\) the property \(|i - j| \geq 5\) holds. Thus bonds are only possible between bases in distance at least five.
(b) The pairs in $P$ form a matching: each position $i \in \{1, \ldots, n\}$ belongs to at most one pair in $P$.

(c) Pairs in $P$ correspond to complementary bases and we have either $\{R_i, R_j\} = \{A, U\}$ or $\{R_i, R_j\} = \{C, G\}$ for any pair $(i, j) \in P$.

(d) The folding may not cross itself: if $\{i, j\}$ and $\{k, l\}$ belong to $P$ and if $i < j, k < l$ hold, then $i < k < j < l$ is excluded.

To predict the secondary structure from its primary structure we assume that the folding minimizes its free energy by maximizing the number of bonds. Under this assumptions we have to solve the following problem:

A string $R \in \{A, C, G, U\}^n$ is given. Determine an RNA secondary structure which maximizes $|P|$.

Property (4) is of great help when trying to compute an optimal secondary structure $P_{opt}$, namely if $P_{opt}$ contains the pair $\{k, n\}$, then $P_{opt}$ decomposes into the optimal secondary structures for the prefix $R_1 \cdots R_{k-1}$ and for the suffix $R_{k+1} \cdots R_{n-1}$ of $R$. Therefore our dynamic programming algorithm solves the subproblems “determine $D(i, j)$”, where

$$D(i,j) = \text{The number of bonds in an optimal secondary structure for } R_i \cdots R_j.$$
To determine \( D(i, j) \) we differentiate two cases. Firstly, if \( j \) is not involved in a match, then we have \( D(i, j) = D(i, j - 1) \). Secondly, if \( P_{opt} \) contains the \textit{legal} pair \((k, j)\), then \( D(i, j) = D(i, k - 1) + D(k + 1, j - 1) + 1 \). (The pair \((k, j)\) is legal, if \( i \leq k \leq j - 5 \) and \( \{R_k, R_j\} = \{A, U\} \) or \( \{R_k, R_j\} = \{C, G\} \) holds. Thus we enforce properties (a) and (c) of Definition 8.11. By decomposing \((i, j)\) into \((i, k - 1)\) and \((k + 1, j - 1)\) we guarantee that property (b) is satisfied, since neither \( k \) nor \( j \) can appear in another pair.) Thus we obtain the \textbf{recurrence}

\[
D(i, j) = \max_{k, (k, j) \text{ is legal}} \{D(i, k - 1) + D(k + 1, j - 1) + 1, D(i, j - 1)\}
\]

which is implemented by the following program.

\textbf{Algorithm 8.12 RNA Secondary Structure Prediction}

\begin{enumerate}
  \item for \((l = 1; l <= 4; l++)\)
    \begin{enumerate}
      \item for \((i = 1; i <= n - l; i++)\)
        \begin{enumerate}
          \item Set \( D(i, i + l) = 0; \)
        \end{enumerate}
    \end{enumerate}
  \item for \((l = 5; l <= n - 1; l++)\)
    \begin{enumerate}
      \item for \((i = 1; i <= n - l; i++)\)
        \begin{enumerate}
          \item Determine \( D(i, i + l) \) with recurrence (8.1).
        \end{enumerate}
    \end{enumerate}
\end{enumerate}

We have to solve a total of \( \binom{n}{2} \) subproblems \( D(i, j) \). Since each subproblem is solved in time \( O(n) \), our program runs in cubic time \( O(n^3) \).

The parallel implementation maintains the outer \( l \)-loop and parallelizes the inner loop. When computing \( D(i, i + l) \) we only need to know \( D(i, i + l') \) and \( D(i + l' + 1, i + l - 1) \) for \( l' + 1 < l - 1 \). We choose the rowwise decomposition of the matrix \( D \) and hence we have to guarantee only that the process responsible for computing \( D(i, i + l) \) has access to all values \( D(i + l' + 1, i + l - 1) \) for \( l' + 1 < l - 1 \). Assume that a process needs access to an unknown value \( D(i + l' + 1, i + l - 1) \) for the first time. If \( D(i + l' + 1, i + l - 1) \) is required for some row \( r \), then \( r \) is its “largest row” or \((r = i + l' \text{ and } l' = 0)\). Thus any process needs at most \( O(n) \) unknown values for any iteration of the \( l \)-loop. Moreover at most \( \frac{n}{p} \) values come from a particular process.

Thus before an iteration of the \( l \)-loop we have to perform an all-to-all personalized broadcast in which each process sends personalized messages of length \( O\left(\frac{n^2}{p}\right) \). Since the compute time per iteration is bounded by \( O\left(\frac{n^2}{p}\right) \), the run time per iteration is bounded by \( O\left(\frac{n^2}{p} + n\right) \) and the total run time is \( O\left(\frac{n^4}{p} + n^2\right) \). We reach constant efficiency whenever \( p \leq n \).

\textbf{Theorem 8.13} An optimal RNA secondary structure according to Definition 8.11 can be determined sequentially in time \( O(n^3) \) for a string of length \( n \). With \( p \leq n \) processes compute time \( O\left(\frac{n^4}{p}\right) \) and communication time \( O(n^2) \) is sufficient.
8.3 Conclusion

Divide & conquer algorithms are prime candidates for parallelization, since subproblems can be solved independently. We have parallelized quicksort and mergesort, but obtained faster solutions for quicksort. However sample sort is the fastest solution, provided the all-to-all personalized broadcast is efficiently supported.

Dynamic programming algorithms have the advantage that subproblems of same complexity can be solved independently. For example we have obtained almost optimal parallel algorithms for the all-pairs-shortest path problem and the transitive closure problem. Finally we have obtained efficient parallel algorithms the pairwise alignment problem as well as for the RNA secondary structure prediction.
Part II
Parallel Random Access Machines
Chapter 9

Introduction

9.1 Shared Memory

Since parallel machines are built from sequential machines we begin with a description of random access machines (RAM), a formal model of a von Neumann machine. Its architecture consists of

- an input device,

- a CPU which stores the program as well as a label referring to the current command,

- a storage of an unbounded number of registers,

- an accumulator which may access a register directly by its address or indirectly by an address stored in a register. The accumulator is also capable of performing arithmetic or boolean operations.

- And an output device.

For simplicity we assume that input and output device correspond to a sequentially from left to right readable, resp. writable tape, which is partitioned into cells. These cells, as well as the accumulator and the registers can store an integer. Registers have natural numbers as addresses; the address 0 is reserved for the accumulator. We specify the value of register $i$ by $v(i)$ and hence $v(0)$ is the value of the accumulator. A RAM can execute the following commands:

- direct and indirect loading:

  \[
  \begin{align*}
  \text{load } i & \quad v(0) := v(i) \\
  \text{load } *i & \quad v(0) := v(v(i))
  \end{align*}
  \]
CHAPTER 9. INTRODUCTION

- direct and indirect storing:

\[
\begin{align*}
\text{store} & \quad i \\ v(i) & := v(0)
\end{align*}
\]

\[
\begin{align*}
\text{store} & \quad *i \\ v(v(i)) & := v(0)
\end{align*}
\]

- arithmetic operations:

\[
\begin{align*}
\text{add} & \quad i \\ v(0) & := v(0) + v(i)
\end{align*}
\]

\[
\begin{align*}
\text{sub} & \quad i \\ v(0) & := v(0) - v(i)
\end{align*}
\]

\[
\begin{align*}
\text{mult} & \quad i \\ v(0) & := v(0) \times v(i)
\end{align*}
\]

\[
\begin{align*}
\text{div} & \quad i \\ v(0) & := \lfloor v(0)/v(i) \rfloor
\end{align*}
\]

Moreover, we may use the indirect variants add *i, sub *i, mult *i and div *i.

- Comparisons have the form:

\[
\text{if } \text{Test} \text{ then } \text{command}
\]

We may use “end-of-file” as a test as well as comparisons of the form

\[
v(0) \{=\{\geq\{\leq\} v(i).
\]

- A goto has the form “goto label”, where label is the unique identifier of a command.

- The input command “read” has as consequence that the currently read integer is written into the accumulator and the input head is advanced by one position to the right. The input command “write” has as consequence that the value of the accumulator is written onto the output tape. The output head is advanced by one position to the right.

- The computation halts, if the command “stop” is executed.

To measure the running time of a program we consider uniform running time and logarithmic running time. The uniform running time is the number of executed operations, whereas the logarithmic running time is proportional to the binary length of the operands and proportional to the binary length of memory locations involved in the computation. Logarithmic running time is more realistic, since it considers the size of the operands. We will however use uniform running time which is far easier to analyze. Moreover our algorithms will only work on relatively small operands and hence uniform and logarithmic running time are strongly related.

We define PRAMs as a powerful multiprocessor model with a shared memory and synchronous computation. Thus we demand the best of all worlds and can concentrate purely on algorithmic questions.
Definition 9.1 A PRAM with \( p \) processors consists of

- \( p \) random access machines which we also call processors. Each processor has its own name which is an integer.

- A shared memory of an unbounded number of registers. Each register has its own unique address, a natural number, and is capable of storing an integer.

A PRAM is computing in a synchronous manner, i.e., all processors are following a global clock. An input \( x_1, \ldots, x_n \) of \( n \) integers is stored in the first \( n \) registers of the shared memory and each processor receives the input length \( n \) as well as the number of processors in a special local register. All processors work according to the same RAM-program, but we also assume that a processor has its own set of local data (such as its name) stored in its local registers as a priori information. All other local and global registers are initialized to be zero. Read- and write-commands have to be adjusted as follows.

- read*: if \( i \) is the contents of the first local register of the processor, then the contents of the \( i \)th register from the shared memory is written into the accumulator

- write*: if \( i \) is the contents of the first local register of the processor, then write the contents of the accumulator into the \( i \)th register from the shared memory.

Thus each processor uses its first local register as an address register. If \( m \) outputs have to be determined, then they have to be stored in the first \( m \) global registers at the end of the computation. We say that an input is accepted (resp. rejected) iff the first register receives the value 1 (resp. 0) at the end of the computation.

We still have to clarify how to deal with read and write conflicts. We have a read-conflict (resp. write conflict) iff at some time at least two processors try to read from (resp. write into) the same register.

Definition 9.2 Let \( P \) be a PRAM program.

(a) \( P \) is a EREW-PRAM (exclusive read exclusive write PRAM) iff there are neither read nor write conflicts.

(b) \( P \) is a CREW-PRAM (concurrent read exclusive write PRAM) iff there are no write conflicts.

(c) ERCW-PRAM and CRCW-PRAM algorithms are defined analogously.
Definition 9.3  (a) The common model demands that all processors writing into the same register at the same time write the same value.

(b) The arbitrary model writes an arbitrary value (from among the values to be written) into the register. It is demanded that the PRAM program works for all choices correctly.

(c) In the priority model the processor with the smallest address wins and its value is written.

Observe that the common model provides the most restrictive conflict resolution scheme, since any PRAM algorithm for the common model also works in the arbitrary and in the priority model. Finally, the arbitrary model is more restrictive than the priority model.

When should we call a problem parallelizable? We require a super-fast parallel algorithm which runs in at most poly-logarithmic time and uses at most a polynomial number of processors.

Definition 9.4 The functions $p, t : \mathbb{N} \to \mathbb{N}$ are given. We define $\text{EREW}(p(n), t(n))$ as the class of languages $L \subseteq \{0, 1\}^*$ which are recognizable with a uniform EREW-PRAM algorithm in time $O(t(n))$ using not more than $O(p(n))$ processors. The classes $\text{ERCW}(p(n), t(n))$, $\text{CREW}(p(n), t(n))$ and $\text{CRCW}(p(n), t(n))$ are defined analogously.

We set $\mathcal{NC} = \bigcup_{k,l \in \mathbb{N}} \text{CRCW}(n^k, (\log_2 n)^l)$. Thus the complexity class $\mathcal{NC}$ consists of all languages $L \subseteq \{0, 1\}^*$ such that $L$ can be recognized by a CRCW-PRAM algorithm in poly-logarithmic time with a polynomial number of processors.

$\mathcal{NC}$ is shorthand for Nick’s Class, since Nick Pippenger has proposed the class to capture all problems in $\mathcal{P}$ which are parallelizable.\footnote{$\mathcal{P}$ is the class of all languages which are recognizable by Turing machines within polynomial time.}

**Exercise 93**
Show that $\mathcal{NC} \subseteq \mathcal{P}$.

**Open Problem 1**
Show that $\mathcal{NC}$ is a proper subset of $\mathcal{P}$.

The concurrent access to cells from shared memory is unrealistic from a hardware point of view. However we may simulate even CRCW-PRAM’s by message passing algorithms with relatively small delay. In particular, assume that we are given a CRCW-PRAM $P$ with $p(n)$ processors which runs in time $t(n)$. We simulate $P$ by a message passing algorithm with $p(n)$ processors arranged in a hypercube network.

Each processor $\rho$ of $P$ is simulated by exactly one node of the hypercube, namely node $v(\rho)$. Moreover we distribute cells of the shared memory among the nodes of the hypercube.
and achieve a “randomly behaving” distribution with the help of a good hash function. When simulating a particular processor \( \rho \) we have to differentiate two cases.

**Case 1:** Processor \( \rho \) is executing a local operation, that is, it neither executes a read nor a write statement. This case is easy, since node \( v(\rho) \) may perform the operation with the information it already knows.

**Case 2.** Processor \( \rho \) executes a read or write statement and node \( v(\rho) \) will have to fetch the required information from one of its colleagues. Thus we are faced with a routing problem, which we know how to solve on the hypercube. The distribution of the shared memory via hashing implies that, with high probability, no bottlenecks arise. Ranade [R91] provides the details of this sketch. In particular he shows

**Theorem 9.5** A CRCW-PRAM algorithm which runs in time \( t(n) \) with \( p(n) \) processors can be simulated by a message passing algorithm with \( p(n) \) processors in time \( O(t(n) \cdot \log_2 p(n)) \) with high probability.

**Exercise 94** Design a parallel algorithm in pseudo code for the the prefix problem. The algorithm should run in time \( O\left(\frac{n}{p(n)} + \log_2 n\right) \) on a EREW-PRAM with \( p(n) \) processors, provided the associative operation can be evaluated in constant time.

### 9.2 Comparing PRAMs of Different Types

Our first result shows a simple hierarchy of the different parallel computation modes.

**Lemma 9.6** Let \( \text{RAM}(t) \) be the class of all languages which can be recognized by a (sequential) random access machine in time \( t(n) \) for inputs of length \( n \). Then

\[
\text{EREW}(p(n),t(n)) \subseteq \text{CREW}(p(n),t(n)) \subseteq \text{CRCW}(p(n),t(n)) \subseteq \text{RAM}(p(n) \cdot t(n)).
\]

**Exercise 95** Verify Lemma 9.6.

What are the known limitations of the various PRAM models? We mention the following results, which are shown in [Ja92].

**Fact 9.1** (a) The input for the zero-counting problem is a sorted 0/1 sequence \( A \). We have to determine the position \( i \) with \( A[i] = 0 \) and \( A[i + 1] = 1 \). Then any EREW-PRAM algorithm for the zero-counting problem requires time \( \Omega(\log_2 \frac{n}{p}) \) with \( p \) processors.

(b) Any CREW-PRAM algorithm requires time \( \Omega(\log_2 n) \) to compute the Boolean OR of \( n \) bits, even if there is an unlimited number of processors.
(c) The problem of computing the *parity function* on $n$ bits requires time $\Omega(\log^2 \frac{n}{\log \log n})$ on a priority CRCW-PRAM with a polynomial number of processors.

We use Fact 9.1 to separate the different PRAM modes.

**Exercise 96**
A summation CRCW-PRAM solves a write conflict by summing all values requesting the same register and assigning the sum to the register.

(a) Show that the parity function of $n$ bits can be computed with $n$ processors in constant time.

(b) Show that $n$ keys can be sorted in constant time with $n^2$ processors.

---

### I: Exclusive Reads Versus Concurrent Reads

Are concurrent reads more powerful than exclusive reads? The answer is yes as shown by the following **parallel search problem**. The input consists of a sorted array $X$ of $n$ keys and an additional key $y$.

**Definition 9.7** Set $X[0] = -\infty$ and $X[n + 1] = +\infty$. We define $\text{rank}(y \mid X)$ to be the position $j$ with $X[j] < y \leq X[j + 1]$.

Our goal is to determine $\text{rank}(y \mid X)$, i.e., the number of elements of $X$ which are smaller than $y$.

**Algorithm 9.8** *Solving the parallel search problem with a CREW PRAM*

1. $p$ processors are available.
2. $m = n + 1$; $l = 0$; $X[0] = -\infty$, $X[m] = +\infty$;
   /* $y$ belongs to the interval $[X[0], X[m]]$, since $[X[0], X[m]] = [-\infty, +\infty]$. This property will be maintained as an invariant. */
3. repeat $\lceil \log_p(n + 2) \rceil$ times
   for $i = 1$ to $p$ pardo
       if $X[l + (i - 1) \cdot \frac{m}{p}] \leq y \leq X[l + i \cdot \frac{m}{p} - 1]$ then
           $l = l + (i - 1) \cdot \frac{m}{p}$, $m = \frac{m}{p}$;
           if $X[l + i \cdot \frac{m}{p} - 1] < y < X[l + i \cdot \frac{m}{p}]$ then
               $\text{rank}(y \mid X) = l + i \cdot \frac{m}{p} - 1$; stop;
               /* There are no write conflicts. However there are read conflicts, since $y$ and $l$ have to be known to all processors. */
Algorithm 9.8 performs a parallel search by partitioning the sorted sequence into $p$ intervals of same size. Within constant time the interval containing key $y$ is determined. Thus time $\lceil \log_p (n + 2) \rceil = \Theta(\log_p n) = \Theta\left(\frac{\log_2 n}{\log_2 p}\right)$ is sufficient.

**Theorem 9.9** (a) Algorithm 9.8 determines rank($y$ | $X$) for a sorted sequence $X$ of size $n$ on a CREW PRAM. Thus it solves the parallel search problem with $p$ processors in time $O\left(\log_2 \frac{n}{\log_2 p}\right)$.

(b) Each EREW-PRAM with $p$ processors requires at least $\Omega(\log_2 \frac{n}{p})$ steps to determine rank($y$ | $X$) and hence concurrent reads are provably more powerful than exclusive reads.

**Proof (b)** follows from Fact 9.1 (a).

**Exercise 97**

We would like to solve the all-to-all broadcasting problem, assuming that one processor sends a message to $n$ other processors.

(a) Show how to solve the one-to-all broadcasting problem in time $O(1)$ on a CREW PRAM with $n$ processors.

(b) Show that time $\Theta(\log_2 n)$ is sufficient and necessary, if we work with an EREW PRAM with $n$ processors.

**II: Exclusive Writes Versus Concurrent Writes**

**Example 9.1** We want to compute the Boolean OR of $n$ Bits $x_1, \ldots, x_n$ with $n$ processors. We assume that $x_i$ is initially stored in register $i$. Processor $i$ reads $x_i$ from the shared memory. If $x_i = 1$, then processor $i$ writes a 1 into register 1 of the shared memory. The computation stops immediately with the output stored in register 1.

Alternatively we could have described our algorithm as follows.

**Algorithm 9.10** An ERCW-Algorithm for the Boolean OR

1. $n$ is the number of input bits. $x$ is a shared array variable storing $n$ bits.
2. Set $N = n$;
3. for $p = 1$ to $N$ pardo
   - if $x[p]$ then $x[1] = true$;

Algorithm 9.10 is an ERCW-Algorithm, since we have write conflicts when several processors try to write into register 1. But processors always write the same value 1 and hence Algorithm 9.10 works in the common model.
Thus ERCW-PRAMs and hence also CRCW-PRAMs are provably more powerful than CREW-PRAMs. On the positive side, EREW-PRAMs and hence also CREW-PRAMs are successful, if time $O(\log_2 n)$ is allowed. To achieve this result we design an EREW-algorithm which evaluates $x_1 \lor \cdots \lor x_n$ on a binary tree with $n$ leaves.

**Algorithm 9.11** An EREW-Algorithm for the Boolean OR

(1) $n$ is the number of input bits. $x$ is a shared array variable storing $n$ bits.

(2) $N = \lfloor n/2 \rfloor$;
for $p = 1$ to $N$ pardo
\begin{align*}
r &= n; \\
\text{while } r > 1 \text{ do} & \quad m = r; \quad r = \lceil r/2 \rceil; \\
\text{if } (p \leq m) \land (p + r \leq m) \text{ then} & \quad x[p] = x[p] \lor x[p + r];
\end{align*}

Thus initially the or’s $x[p] = x[p] \lor x[p + \lfloor n/2 \rfloor]$ are computed simultaneously for $p \leq n - \lfloor n/2 \rfloor = \lfloor n/2 \rfloor$. It remains to determine $x[1] \lor \cdots \lor x[\lfloor n/2 \rfloor]$ recursively. Hence this EREW-algorithm determines the Boolean OR in time $\log_2 n$ with $\lfloor n/2 \rfloor$ processors.

**Example 9.2** We want to find the minimum of $n$ integers $x_1, \ldots, x_n \in \{1, \ldots, n\}$ using the priority model.

**Algorithm 9.12** An ERCW-Algorithm to determine the minimum

(1) $n$ numbers are stored in the shared array $x$.

(2) for $p = 1$ to $n$ pardo
\begin{align*}
x[x[p]] &= 0; \\
\text{if } x[p] = 0 \text{ then } x[1] &= p;
\end{align*}

We utilize the priority model in the last step. Register 1 contains the minimal value, since the processor with the smallest name is allowed to write. But processor $p$ is only allowed to compete if $p$ is among the $n$ input numbers. Observe that we use $n$ processors and work in two steps.

This is another example for the power of concurrent writes over exclusive writes: Assume that the input sequence is a sequence of zeroes and ones and assume that a CREW-PRAM algorithm can find a minimal element in time $o(\log_2 n)$. We design another CREW-PRAM algorithm that computes an $n$-bit OR in time $o(\log_2 n)$, contradicting Fact 9.1 (b). In its first step the CREW-PRAM changes each one into a zero and each zero into a one. Then it determines the minimal element. The minimal element is a one iff the OR of the original bits evaluates to zero.
Exercise 98
We are given a binary sequence \( x = (x_0, \ldots, x_n) \), which is stored in the first \( n \) registers of the shared memory. Show how to determine the position of the first one in time \( O(1) \) with \( n \) processors on a CRCW PRAM.

Hint: Partition the sequence into \( \sqrt{n} \) intervals of length \( \sqrt{n} \) and determine for each interval, if it contains a one. Can you determine the smallest interval with a one in time \( O(1) \)?

We have seen already that ERCW-PRAMs are more powerful than CREW-PRAMs when computing an \( n \)-bit OR. Here we show as a consequence of Theorem 9.5 that EREW-PRAMs can catch up to CRCW-PRAMs, if given a little more time.

**Theorem 9.13** \( \text{CRCW}(p, t) \subseteq \text{EREW}(p, t \cdot \log_2 p) \), if we assume the priority mode for CRCW algorithms and allow randomized EREW computations.

**Exercise 99**

III: Comparing Write Resolution Schemes

First we compare the powerful “priority” resolution scheme with the weak “common” resolution scheme.

**Theorem 9.14** Any priority-CRCW PRAM \( P \) with \( p \) processors and time \( t \) can be simulated by a common-CRCW PRAM \( Q \) with \( O(p \cdot \log_2 p) \) processors in time \( O(t) \).

**Proof:** The two models differ only in their write statement and we have to find the processor with smallest address among all processors trying to write into the same shared memory location.

We reserve for each register of \( P \) a block of \( 2 \cdot p - 1 \) registers within the shared memory of \( Q \). We demand that the blocks of different registers are disjoint and all cells are initialized to be zero.

Processor \( i_Q \) of \( Q \) will simulate processor \( i_P \) of \( P \) with the help of \( \log_2 p \) assistants. If \( i_P \) tries to write into memory location \( r_i \), then \( i_Q \) writes a one into the \( i \)th cell of the block of register \( r_i \). We can use the \( 2 \cdot p - 1 \) cells of a block to implement a binary tree of depth \( \log_2 p \) with the first \( p \) cells as its leaves. Thus at this moment the leaves that correspond to processors with a write request receive a one and the rest is labeled by a zero. Processor \( i_Q \) uses its assistants to enforce the write request of processor \( i_P \): the assistants write a one in each predecessor of its leaf for which the leaf belongs to a left subtree. Observe that all writes are according to the common model, since all processors write a one.

After writing into the nodes of the tree, each processor \( i_Q \) determines with the help of its assistants, whether there is a predecessor with a one for which its leaf belongs to the
right subtree. If this is the case, then $i_Q$ has to give up the write request. If this is not the case, then $i_Q$ can execute the desired write.

How big is the computing power of CRCW-PRAMs? The XOR of $n$ bits turns out to be a tough problem.

**Theorem 9.15** Let $P$ be a priority CRCW PRAM. Then $P$ requires at least $\Omega(\frac{\log n}{\log \log n})$ steps to compute the XOR $x_1 \oplus \cdots \oplus x_n$ with a polynomial number of processors.

**Proof:** This is a consequence of Fact 9.1 (c).

Observe that $\oplus$ is an associative operation and hence Theorem 9.15 shows that the prefix problem has no solutions with running time $o(\frac{\log n}{\log \log n})$ and a polynomial number of processors.

**9.3 Conclusion**

We have introduced PRAMs as a synchronous parallel computing model with a shared memory and have distinguished exclusive and concurrent read/write resolution schemes. We have defined the complexity class $NC$ as the class of all parallelizable problems in $\mathcal{P}$, that is as the class of problems in $\mathcal{P}$ with PRAM algorithms running in poly-logarithmic time with a polynomial number of processors.

Finally we have compared PRAM algorithms with message passing algorithms and have compared the various conflict resolution schemes.
Chapter 10

Linked Lists and Trees

We describe a collection of methods which are of great help in designing parallel algorithms for graph problems. In particular we discuss the doubling technique, Euler tours and tree contraction.

10.1 The Doubling Technique

We describe the pointer jumping technique, a method which allows a fast traversal of trees and linked lists in particular. A forest $F$ is represented by its set $V = \{1, \ldots, n\}$ of $n$ nodes and by an array parent, where $\text{parent}[i]$ is the parent of node $i$. (If $i$ is a root, then $\text{parent}[i] = i$.) Our goal is to determine the respective root for all nodes in parallel.

**Algorithm 10.1 Finding the root via pointer jumping**

for $i = 1$ to $n$ pardo
    while $\text{parent}[i] \neq \text{parent[\text{parent}[i]]}$ do
        $\text{parent}[i] = \text{parent[\text{parent}[i]]}$;
    Output ($i$, $\text{parent}[i]$);

Algorithm 10.1 works with $n$ processors, where processor $i$ is responsible for node $i$. Node $i$ advances to become a child of its current grandfather unless its parent is a root. Observe that the edges “double in length”.

155
Theorem 10.2
Algorithm 10.1 is a CREW-PRAM algorithm working with \( n \) processors. If \( F \) is a forest with \( n \) nodes and if \( d \) is the depth of \( F \), then algorithm 10.1 determines a root for all nodes of \( F \) in time \( O(\log_2 d) \).

Proof: The shared array “parent” is concurrently read, but only exclusively written and hence Algorithm 10.1 is a CREW-PRAM algorithm. We still have to determine the running time. We claim inductively that if node \( i \) becomes a child of node \( j \) in round \( t \), then \( i \) and \( j \) have distance \( 2^t \) in the original forest, provided \( j \) is not a root.

The claim is true for \( t = 0 \). If \( i \) is a child of \( k \) in round \( t \) and becomes a child of \( j \) in round \( t + 1 \), then, by induction hypothesis \( i \) and \( k \) as well as \( k \) and \( j \) have distance \( 2^t \) in \( F \) and hence \( i \) and \( j \) have distance \( 2^{t+1} \) in \( F \).

Exercise 100
Show that time \( \Omega(\log_2 n) \) is required for an EREW PRAM with \( n \) processors. This statement holds, even if we work with forests of depth two.

Exercise 101
A tree \( T \) is given by parent pointers. Moreover we assign a real number \( x_v \) to each node \( v \). Show how to determine, in parallel for all nodes \( v \), the sum of all values which are assigned to nodes on the path from \( v \) to the root. You may work with CREW-PRAM in time \( O(\log_2 n) \) with \( n \) processors, provided \( T \) has \( n \) nodes.

In our second application of pointer jumping we consider the list ranking problem:

Given is a singly linked list of \( n \) nodes. The linked list is represented by a shared array \( S \), where \( S[i] \) is the successor of \( i \) within the linked list. (If \( i \) is the end of the list, we set \( S[i] = 0 \).) Each node \( i \) has also a value \( V[i] = a_i \) and the problem is to...
determine the $n$ suffix sums

$$a_n \cdot a_{n-1} \cdot a_1 \cdot a_2 \cdot \ldots \cdot a_{n-1} \cdot a_n$$

for an associative operation $\ast$.  

If we set for instance $V[i] = 1$ for all $i$, then the output for list element $i$ is its distance (plus one) from the end of the list. There is a strong similarity to the prefix problem. Whereas we are now interested in suffix sums, we had to compute prefix sums in the prefix problem, however this difference is not substantial. The crucial difference is that we had random access to elements in the prefix problem and are now restricted to a list access only.

**Algorithm 10.3 List ranking via pointer jumping**

for $i = 1$ to $n$ pardo

$W[i] = V[i]; T[i] = S[i]$;

while ($T[i] \neq 0$) do

$W[i] = W[i] \ast W[T[i]]$;

$T[i] = T[T[i]]$;

**Exercise 102**

Verify Algorithm 10.3 and show that it runs on an EREW-PRAM.

Change of the successor array in Algorithm 10.3.
Theorem 10.4 Assume that the operation \( \ast \) can be evaluated sequentially in time \( T \). Then Algorithm 10.3 solves the list ranking problem for \( n \) elements and can be implemented on a EREW-PRAM in time \( \Theta(T \cdot \log_2 n) \) with \( n \) processors.

Algorithm 10.3 is simple and fast, but not efficient since the total work is \( \Theta(T \cdot n \cdot \log_2 n) \) whereas a sequential algorithm just requires time \( \Theta(T \cdot n) \). Efficient algorithms with \( O\left(\frac{n}{\log_2 n}\right) \) processors and running time \( O(T \cdot \log_2 n) \) are known.

10.2 Euler Tours and Tree Contraction

Traversing a graph is a fundamental method in many graph algorithms and depth-first search is a very fast sequential graph traversal.

Algorithm 10.5 Depth-first search

1. Depth-first search is a recursive procedure which is started at node \( v \). We assume that initially no node is marked as “visited”.

2. Depth-first-search(\( v \))
   - Mark \( v \) as visited;
   - Let \( u \) be the first node on the adjacency list of \( v \);
   - while \( u \neq \text{nil} \) do
     - if \( u \) is not marked visited then
       - depth-first-search(\( u \));
     - Replace \( u \) by the next node on the adjacency list of \( v \);

However an efficient parallel implementation of depth-first search is not known. In this chapter we describe a fast parallel implementations of depth-first search for trees which are known beforehand and show how to use this method to determine a preorder, postorder or breadth first search traversal for trees. (Observe that our parallel backtracking algorithm for implicitly defined trees –see Section 6.1– does not produce a depth-first search traversal.)

How does depth-first search work when applied to trees? Starting at the root it follows the tree edges to reach a leaf. After reaching a leaf, the traversed edges are traversed in backwards direction. Depth-first search stops, when all edges are traversed exactly once in either direction.

Definition 10.6 Let \( T = (V, E) \) be an undirected tree. We set \( E^* = \{ (i, j) \mid \{i, j\} \in E \} \) and define the “Euler graph”

\[
\text{Euler}(T) := (V, E^*)
\]

An Euler tour is a path in Euler(T) which traverses all edges of Euler(T) exactly once and returns to its starting point.
10.2. EULER TOURS AND TREE CONTRACTION

Exercise 103
Show that an Euler tour of a tree $T$ corresponds to a depth-first search traversal of $T$.

Thus it suffices to come up with a fast algorithm that determines an Euler tour. Assume that $T$ is given by its adjacency list representation. Hence all neighbors of a node $v \in V$ are collected in a linked list $N[v]$ and the linked list orders neighbors according to their position within the list. Which edge should follow after edge $(u, v)$ on the Euler tour?

If $N[v] = (\ldots, u, w, \ldots)$, then continue with edge $(v, w)$.

Moreover, if $u$ is the last node in $N[v]$, then we continue with edge $(v, w)$, where $w$ is the first node in $N[v]$. We call $(v, w)$ the successor of edge $(u, v)$.

Lemma 10.7 If we always select the successor of the last edge as the next edge, then we obtain an Euler tour in Euler($T$).

Proof: We give an inductive proof on the number $n$ of nodes of $T$. If $n = 1$, then $T$ and thus Euler($T$) has no edges. Now assume that $T$ is a tree with $n + 1$ nodes. If $l \in V$ is a leaf, then $l$ its adjacency list consists only of its parent $v$. We remove $l$ (from $T$ and Euler($T$)) and our “recipe” produces an Euler tour $(e_1, \ldots e_{n-1})$.

![Euler tour diagram]

Assume that leaf $l$ is the right neighbor of $u$ in list $N[v]$ and that $w$ is the right neighbor of $l$; in particular $N[v] = (\ldots, u, l, w, \ldots)$. Now consider the edge $e_j = (u, v)$. Our recipe for the original graph requires that, after traversing edge $(u, v)$, we continue with edge $(v, l)$. Parent $v$ is its own right neighbor in list $N[l]$ and we continue with edge $(l, v)$. But $w$ is the right neighbor of $l$ in list $N[v]$ and $e_{j+1} = (v, w)$ is the next edge. The recipe works, since

$$(e_1, e_2, \ldots, e_j, (v, l), (l, v), e_{j+1}, e_{j+2}, \ldots, e_{n-1}).$$

is an Euler tour. \qed

To come up with fast implementation for finding Euler tours we have to figure out how to compute the successor $e' = (v, w)$ of an edge $e = (u, v)$. We simplify this problem by demanding that $T$ is given as an adjacency list with cross references. In particular we require that
- the adjacency list $N[v]$ is given as a circular list and
- for any element $w$ in $N(v)$ there is a link to element $v$ in $N(w)$.

**Example 10.1** We consider the following tree:

![Diagram of a tree with adjacency lists](image)

The circular adjacency lists without cross references.

![Diagram of the Euler tour](image)

The Euler tour which begins in edge $(4, 6)$. 

160  

CHAPTER 10. LINKED LISTS AND TREES
If we start with the edge $(4, 6)$, then we obtain the Euler tour

$$(4, 6), (6, 5), (5, 6), (6, 4),$$

$$(4, 2), (2, 1), (1, 2), (2, 9), (9, 2), (2, 4),$$

$$(4, 7), (7, 3), (3, 7), (7, 10), (10, 7), (7, 8), (8, 7), (7, 4).$$

**Theorem 10.8** Assume that a tree $T = (V, E)$ is given as an adjacency list with cross references. Then we can find the successors of all edges in parallel in time $O(1)$ with $2(|V| - 1)$ processors on an EREW-PRAM. Hence an Euler tour, represented as a linked list, can be constructed in time $O(1)$ with $|V|$ processors on an EREW-PRAM as well.

We now bring in the harvest. Postorder$(v)$ and preorder$(v)$ are recursive procedures according to the following recipes

- Postorder$(v)$: first the descendants of a node $v$ are traversed and then $v$.
- Preorder$(v)$: first $v$ is traversed and then its descendants.

The **numbering of a traversal** assigns the time to each node at which it is traversed. The level-order numbering assigns the depth, that is the length of the path from the root, to a node.

**Theorem 10.9** Let $T = (V, E)$ be an undirected rooted tree which is presented as an adjacency list with cross references. Then the following problems are solvable on an EREW-PRAM in time $O(\log_2 |V|)$ with $\frac{|V|}{\log_2 |V|}$ processors:

(a) determine the parent parent$(v)$ for each node $v \in V$,
(b) determine a postorder numbering,
(c) determine a preorder numbering,
(d) determine a level-order numbering and
(e) determine the number of descendants for every node $v \in V$.

**Proof:** All our solutions compute an Euler tour, assign weights to the edges and compute prefix sums. (Since we apply list ranking and thus are only able to compute suffix sums, we have to reverse edge directions.)

**Algorithm 10.10 The Euler tour algorithm**

(1) A tree is given as an adjacency list with cross references.
(2) Assign weights “\( \text{weight}(e) \)” to edges \( e \) of Euler(\( T \)).

// The weight assignment has to be done in a problem dependent way.

(3) Determine an Euler tour beginning in the root.

(4) Apply list ranking, with addition as operation, to the reversed list of the Euler tour.

(5) Evaluate the sums in a problem specific way.

(a) We have to determine parents in parallel and set \( \text{weight}(e) = 1 \) for all edges \( e \). Let \( \text{value}(e) \) be the sum computed by list ranking for edge \( e \). We can now characterize the parent of a node \( v \) by

\[
\text{u is parent of v} \iff \text{value}(u, v) < \text{value}(v, u).
\]

Why? \( \text{value}(u, v) \) is the position in which \( (u, v) \) appears within the Euler tour. \( \text{value}(u, v) < \text{value}(v, u) \) implies that the Euler tour has a segment of the form \( u \rightarrow v \rightarrow \cdots \rightarrow v \rightarrow u \). But then \( v \) has to belong to the subtree of \( u \) and hence \( u \) is the parent of \( v \).

(b) To determine the postorder numbering we first determine parents and set

\[
\text{weight}(u, \text{parent}(u)) = 1, \quad \text{weight}(\text{parent}(u), u) = 0.
\]

We claim that

\[
\text{post}(v) = \begin{cases} 
  n & \text{v is the root,} \\
  \text{value}(v, \text{parent}(v)) & \text{otherwise}
\end{cases}
\]

is a postorder numbering. This is indeed true, since we assign to node \( v \) the moment in time at which \( v \) was visited the last time. (c) As for postorder we first determine parents and then set

\[
\text{weight}(u, \text{parent}(u)) = 0, \quad \text{weight}(\text{parent}(u), u) = 1.
\]

This time we claim that

\[
\text{post}(v) = \begin{cases} 
  1 & \text{v is the root,} \\
  \text{value}(\text{parent}(v), v) + 1 & \text{otherwise}
\end{cases}
\]

is a preorder numbering. Also this claim holds, since we assign the moment in time at which node \( v \) is visited for the first time.
10.2..Euler tours and tree contraction

Preorder numbering with Euler tours.

(d) The level-order is also easy to determine. We set \( \text{weight}(\text{parent}(v), v) = 1 \) to capture that depth increases by one and \( \text{weight}(v, \text{parent}(v)) = -1 \) to capture that we move back up to decrease depth by one. Thus

\[
\text{level}(v) = \begin{cases} 
0 & \text{if } v \text{ is the root,} \\
\text{value}(\text{parent}(v), v) & \text{otherwise}
\end{cases}
\]

is a level-order numbering.

(e) The task is to determine the number of descendants in parallel for all nodes \( v \). (We count a node as its own descendant.) Set

\[
\text{weight}(v, \text{parent}(v)) = 1, \quad \text{weight}(\text{parent}(v), v) = 0
\]

and

\[
\text{value}(v, \text{parent}[v]) - \text{value}(\text{parent}[v], v)
\]

is the number of descendants of \( v \), provided \( v \) is different from the root, since each descendant is counted once by counting its “back edge”. Obviously the root has \( n \) descendants.

Determining the number of descendants with the help of Euler tours.

We conclude the argument with a running time analysis. List ranking dominates the running time and we utilize that list ranking can be supported in time \( O(\log_2 n) \) with \( \frac{n}{\log_2 n} \) processors.
Exercise 104
Determine an inorder numbering of a binary tree which is given as an adjacency list with cross references. Your algorithm should run on an EREW-PRAM with $\frac{n}{\log_2 n}$ processors in time $O(\log n)$.

Hint: an interior node $v$ is traversed by inorder after crossing the edge $(u, v)$, where $u$ is the left child of $v$. Therefore it seems natural to assign weights different from zero only to these edges $(u, v)$. However how should we proceed, if $v$ is a leaf?

Exercise 105
A tree is given as an adjacency list with cross references. Determine the least common ancestor of a given pair $(u, v)$ of nodes in time $O(\log n)$ with $\frac{n}{\log_2 n}$ processors on a EREW-PRAM.

We describe another important application of Euler tours, the evaluation of arithmetic expressions. Let $T = (V, E)$ be an expression tree, i.e., a tree all of whose inner nodes have degree two and are labeled with either $+, -, \cdot$ or $/$. Leaves are labeled by real numbers and our goal is to evaluate the corresponding arithmetic expression. We again assume that the tree is represented as an adjacency list with cross references.

We apply $O(\log n)$ iterations to $T$, where each iteration contracts $T$ until at most three leaves “survive” and we can evaluate immediately. We have to combine contraction with evaluation, but restrict ourselves first to describing the contraction process. Let’s assume that our current tree has the structure of the left hand side tree in the following figure.

Locally, a contraction step consists in removing leaf $u$ as well as its parent $v$ and making the sibling $x$ of $u$ a child of the grandparent $y$. (When evaluating we have to partially evaluate $v$ excluding the contribution of $x$. The partial result is stored in the grandparent $y$.)

However, globally, many contraction steps have to be performed in parallel without collisions: If leaf $u$ is supposed to be removed, then we have to guarantee that

- neither the sibling $x$ of $u$ nor
- the grandparent $y$ of $u$ is removed by other contraction steps.

Our first approach is to remove only every second leaf, that is all leaves in an odd position relative to the preorder numbering restricted to leaves. But this is not sufficient:
10.2. EULER TOURS AND TREE CONTRACTION

Collisions when removing leaves in odd positions only.

The grandparent \( v \) of \( u_2 \) is removed when we remove the leaf \( u_1 \), but \( v \) has to survive if we also remove \( u_2 \). Therefore we impose a further restriction and remove in a first subphase only left leaves in an odd position followed by a second subphase where right leaves in odd position are removed. (Hence, in the above example, we first remove the left leaf \( u_1 \) and subsequently the right leaf \( u_2 \). How to determine leaves in odd position?

**Algorithm 10.11** **The selection process: finding leaves in odd position**

1. Determine an Euler tour of Euler(\( T \)).

2. A node is a leaf iff it has exactly one neighbor.

3. Set weight(\( e \) = \( 1 \) if \( e = (u, l) \) for a leaf \( l \), 0 otherwise;

4. Determine the prefix sums value(parent[\( l \)], \( l \)) for all leaves \( l \) via list ranking.  
   // value(parent[\( l \)], \( l \)) is the number of leaves “to the left” of \( l \).

The contraction process consists of a logarithmic number of iterations where we first eliminate left leaves in odd position followed by eliminating right leaves in odd position.

**Algorithm 10.12** **The contraction process.**

1. Perform the selection process of Algorithm 10.11;

2. while there are more than three leaves do

   - Remove all left leaves in odd position;

   - Remove all right leaves in odd position;

   /* Now only leaves in even position remain. All positions are divided by two and hence the new leaves in odd positions can be determined in time \( O(1) \). */
Theorem 10.13 Assume that a given binary tree is represented by an adjacency list with cross references. Then Algorithm 10.12 performs the contraction process in time $O(\log_2 n)$ on an EREW-PRAM with $\frac{n}{\log_2 n}$ processors, where $n$ is the number of nodes.

Proof: The selection process runs in time $O(\log_2 n)$ on an EREW-PRAM with $\frac{n}{\log_2 n}$ processors. Now observe that we half the number of leaves after one iteration and hence only a logarithmic number of iterations are performed. We are done, since each iteration runs in constant time.

We still have to combine contraction and evaluation. Consider a local contraction step

We have to remove node 2, although we do not know its value. We do know however that nodes are computing rational functions and hence we represent all values as rational functions

$$ax_v + b \over cx_v + d'$$

where $x_v$ is the (possibly unknown) value computed by the subtree with root $v$ and parameters $a, b, c$ and $d$ are to be determined during contraction. We begin by choosing $a = d = 1, b = c = 0$ for all nodes $v$.

In the above example of removing node 2 the function typ of node 3 has to be recomputed. We assume that node 3 is currently computing the function $\frac{ax_3 + b}{cx_3 + d}$ and let $e$ be the result of node 1. If node 2 is performing a division, then $\frac{e}{\frac{ax_3 + b}{cx_3 + d}} = \frac{(ec)x_3 + (de)}{ax_3 + b}$ is the new function of node 3. In case of an addition, the new value is $e + \frac{ax_3 + b}{cx_3 + d} = \frac{(a+ec)x_3 + (b+ed)}{cx_3 + d}$.

The cases of node 2 performing a subtraction or multiplication are similar. We can now conclude

Theorem 10.14 Let $T$ be an expression tree which is represented as an adjacency list with cross references. Then the arithmetic expression of $T$ can be evaluated in time $O(\log_2 n)$ with $O(\frac{n}{\log_2 n})$ by an EREW-PRAM algorithm, where $n$ is the number of nodes.

Exercise 106
We are given a rooted binary tree $T$ via parent and child pointers. Moreover each node $v$ is labelled
with the real number $x_v$. Show how to determine, in parallel for all nodes $v$, the minimal value in the subtree with root $v$. If $T$ has $n$ nodes, then your algorithm should run in time $O(\log_2 n)$ on an EREW PRAM with $3n$ processors.

Exercise 107
Show how to derive an expression tree from a fully bracketed arithmetic expression. You may assume that the input is given as an array with the following structure:

$A$: array $[1 \ldots n]$ of \{'(', ')', '+', '-', '*', '/', '0', '1', \ldots, '9'\}

10.3 Conclusion

We have solved the list ranking problem, a version of the prefix problem for singly linked lists, with pointer jumping. Pointer jumping is an instance of the doubling technique.

The method of Euler tours allows to parallelize many tree traversals such as preorder, postorder, inorder and level-order. Another important application is the evaluation of arithmetic expressions, which was achieved by also applying the method of tree contraction. In each of these applications list ranking was a central tool.

In all cases we obtained optimal parallel algorithms with running time $O(\log_2 n)$ by utilizing optimal algorithms for list ranking. Is it possible to reduce the running time even further? The prefix problem is not harder than the list ranking problem and an “easy” instance is to determine all prefix sums of $x_1 \oplus \cdots \oplus x_n$. However we have shown in Theorem 9.15 that any reduction below time $\Omega(\frac{\log_2 n}{\log_2 \log_2 n})$ will force a super-polynomial number of processors and hence $\Omega(\frac{\log_2 n}{\log_2 \log_2 n})$ is also a lower bound for the list ranking problem.

Open Problem 2
Determine the smallest running time for CRCW-PRAM algorithms, which solve the list ranking problem with a polynomial number of processors.
Chapter 11

Algorithms for Graphs

We have already solved the all-pairs-shortest path problem with a message passing algo-
rithm in computing time $O\left(\frac{n^2}{p} + \frac{n^2}{\sqrt{p}} \log_2 \sqrt{p} + n\right)$ and communication time $O\left(\frac{n^2}{\sqrt{p}} \log_2 \sqrt{p}\right)$. We can utilize its solution to determine a breadth-first search tree in parallel, but this solution will be quite expensive for sparse graphs.

Exercise 108
Determine a breadth-first search tree for a given directed graph $G = (V, E)$. What is the total work and the running time of your solution?

The situation for depth-first search trees is even more complex, since deterministic solutions with poly-logarithmic time and polynomially many processors are not known. (In [AAK90] a randomized algorithm with poly-logarithmic time and polynomially many processors is described. The solution is however not efficient.) Our goal is nevertheless to obtain almost optimal parallel algorithms for fundamental graph problems such as determining connected components, computing minimal spanning trees, coloring graphs and determining minimal independent sets. In all these cases we have to work without graph traversals.

Many important graph problems have almost optimal parallel algorithms with fast running time, with the single-source-shortest-path problem as the most prominent graph problem that is missing from this list.

11.1 Connected Components

Let $G = (V, E)$ be an undirected graph with $n$ nodes and $m$ edges. Our goal is to determine the connected components\(^1\) of $G$. Our algorithm begins by forming singleton loops, i.e., by connecting each node to itself. Singleton loops are trivial example of stars, which consist

\(^1\)A subset $C \subseteq V$ is a connected component of $G$ iff any two nodes of $C$ are connected by a path and there is no node in $V \setminus C$ which is connected to a node in $C$. 

169
of a distinguished node, the root, and other nodes; all nodes of a star, including the root, are connected with the root.

![Figure 11.1: A star.](image)

Subsequent steps of our algorithm merge stars. Our hope is that we can represent connected components as stars after relatively few merging steps. In particular, our implementation has to organize the merging step and we have to tackle the following issues.

- Each merging step (or “hooking step”) attaches some star below others. This process destroys the star-property.

- A merging schedule has to be determined such that merging loops (i.e., attach star \( s_1 \) below star \( s_2 \) and star \( s_2 \) below \( s_1 \)) are impossible.

**Algorithm 11.1 Determining connected components.**

/* The undirected graph \( G = (V, E) \) with \( |V| = n \) and \( |E| = m \) is given. We work with \( n + m \) processors, i.e., we have one processor per node and one processor per edge. */

(1) for \( i = 1 \) to \( n \) pardo parent\[i\] = \( i \);

/* The parent-array defines \( n \) stars where each star consists of the root only. An edge \( \{u, v\} \in E \) is called alive, if \( u \) and \( v \) belong to different stars. Observe that initially all edges are alive. */

(2) while (at least one edge of \( G \) is alive) do

(a) each root \( w \) (i.e., each node \( w \) with parent\( (w) = w \)) chooses a gender \( g(w) \in \{0, 1\} \) at random and assigns its gender to its children. We interpret gender 0 as male and gender 1 as female;

/* We apply the method of symmetry breaking by randomization. Symmetry breaking allows to assign different tasks to different processors. */

(b) each processor responsible for connecting a male star \( m \) with a female star \( f \) tries to write \( f \) into the register of \( m \).

(c) The arbitrary-write resolution scheme assigns a female star \( f(m) \) to any male star \( m \). The root of \( m \) is made to point to the root of \( f(m) \).
(d) for \( i = 1 \) to \( n \) pardo parent\([i]\) = parent[parent\([i]\)];
// One round of pointer jumping repairs the star property.

**Example 11.1** We show the result of the first iteration for a graph on eight nodes. Female nodes are distinguished by two circles and edges chosen by male nodes are marked.

Thus we have obtained five stars. For the second iteration we assume that the stars of nodes 1 and 8 are female and that all other stars are male.
and \(\{1, 2, 3, 4, 5\}, \{6, 7, 8\}\) are the connected components of \(G\).

We begin the analysis with the following immediate observations.

**Lemma 11.2**  
(a) One round of pointer jumping suffices to repair the star property.  
(b) At all times do the nodes of a star belong to the same connected component.  
(c) Each iteration of the while-loop runs in time \(O(1)\) on a CRCW-PRAM with \(n + m\) processors. The arbitrary mode is used.

We say that a star is *alive*, if one of its nodes is endpoint of an alive edge. We say that the root of a star *disappears* if its star is attached below another star.

**Lemma 11.3** The root of a fixed alive star disappears with probability at least \(\frac{1}{4}\).

**Proof:** Consider an alive edge \(\{u, v\}\), where \(u\) belongs to the fixed star. With probability \(\frac{1}{4}\) the star of \(u\) will be male and the star of \(v\) will be female. Hence the root of the star of \(u\) disappears with probability \(\frac{1}{4}\). \(\Box\)

**Theorem 11.4** Algorithm 11.1 runs in expected time \(O(\log_2 n)\) with \(n + m\) processors on a CRCW-PRAM in the arbitrary mode. In particular \(5 \cdot \log_2 n\) iterations of the while-loop suffice with probability at least \(1 - \frac{1}{n}\).
11.1. CONNECTED COMPONENTS

Proof: It suffices to show that $5 \cdot \log_2 n$ iterations of the while-loop suffice with probability at least $1 - \frac{1}{n}$. We fix some node $w$ and determine the probability $p_w$ that $w$ does not disappear as root after $5 \cdot \log_2 n$ iterations:

$$p_w \leq \left( 1 - \frac{1}{4} \right)^{5 \cdot \log_2 n} = \left( \frac{3}{4} \right)^{5 \cdot \log_2 n} = \left( \frac{243}{1024} \right)^{\log_2 n} \leq \left( \frac{1}{4} \right)^{\log_2 n} = 2^{-\log_2 n} = n^{-2}.$$

But at most $n$ nodes can be a root and hence the probability that at least two roots of a connected component do not disappear is at most $\frac{1}{n}$ and this was to be shown. \hfill \Box

Algorithm 11.1 is efficient, but not optimal, since a sequential algorithm can determine connected components in time $O(n + m)$. An optimal probabilistic algorithm can be found in [Ga86].

Exercise 109
Show that Algorithm 11.1 runs in expected time $O(\log_2 n)$.

Exercise 110
(a) Modify Algorithm 11.1 so that the determined spanning forest is given as an adjacency list with cross references.

(b) Show how to determine in time $O(\log_2 n)$ with $n + m$ processors, whether an undirected graph $G$ with $n$ nodes and $m$ edges is bipartite. You may use a CRCW-PRAM with the arbitrary mode.

Exercise 111
Our goal is determine all bridges of an undirected graph $G = (V, E)$. A bridge is an edge whose removal leaves its endpoints in two different connected components. Begin by computing a spanning forest and a preorder numbering.

(a) Show that only edges of the spanning forest can be bridges.

(b) Show how to determine all bridges in time $O(\log_2 |V|)$ with $|V| + |E|$ processors on a CRCW PRAM with the arbitrary mode. You may assume that minimal (resp. maximal) keys can be determined in time $O(\log_2 |V|)$ with $|V|$ processors in parallel for all subtrees of the spanning forest.

Hint: for any edge $e$ of the forest consider the preorder numbers of nodes appearing below edge $e$.

Exercise 112
We want to solve the 2-SAT problem: determine, if existing, a satisfying assignment for a formula $F$ in conjunctive normal form with at most two literals per clause. (Observe that the corresponding problems for three literals is $NP$-complete.)

Consider the directed graph $G$, which contains a node for every positive and a node for every negative occurrence of a variable. Moreover there is are two edges $\neg L_1 \rightarrow L_2$ and $\neg L_2 \rightarrow L_1$ for every clause $L_1 \lor L_2$. Thus we interpret clauses as edges.

(a) Show that $F$ is satisfiable iff $L$ and $\neg L$ belong to different strongly connected components for all literals $L$.

(b) Determine in time $O(\log_2 n)$ on a CRCW-PRAM with polynomially many processors, if a satisfying assignment exists.

(c) All literals belonging to the same connected component are forced to obtain the same truth value in a satisfying assignment.

(d) Determine a satisfying assignment in time $O(\log_2 n)$ on a CRCW-PRAM with polynomially many processors.
11.2 Minimum Spanning Trees

We are given an undirected connected graph $G = (V, E)$ and a function $\text{weight}: E \to \mathbb{R}$, assigning real numbers to edges. We look for a spanning tree $T = (V, E_T)$ of $G$ such that

$$\text{weight}(T) = \sum_{e \in E_T} \text{weight}(e)$$

is minimal among all spanning trees for $G$. The sequential algorithms of Kruskal and Prim are not suitable for parallelization. Instead we choose the algorithm of Boruvka:

- begin with a forest of singleton trees.
- as long as the forest $F$ contains more than one tree:
  - determine for each tree $T$ of $F$ an edge $e_T$ with minimal weight connecting a node of $T$ with a node outside of $T$. Insert $e_T$.

We verify Boruvka’s algorithm by induction on the number of iterations. We assume that edges have pairwise distinct weights; we can enforce this, if we replace $\text{weight}(e)$ by $n^2 \cdot \text{weight}(e) + n \cdot u + v$, provided $e = \{u, v\}$. (We assume that $V = \{0, \ldots, n-1\}$ is the set of nodes.)

Let $(T_1, \ldots, T_k)$ be the trees in the current forest $F$. Hence, if $V_i$ is the set of nodes of $T_i$, then we obtain the partition $V = \bigcup_{i=1}^{k} V_i$. By induction hypothesis we may assume that any minimum spanning tree contains each $T_i$ as a subtree. Boruvka’s algorithm determines edges $(e_1, \ldots, e_k)$, such that $e_i$ is the edge of minimal weight connecting a node in $T_i$ with a node outside of $T_i$.

**Lemma 11.5** Any minimum spanning tree contains all edges in \{e_1, \ldots, e_k\}.

**Proof:** We assume otherwise and obtain a minimal spanning tree $T$ which does not, say, contain the edge $e_1 = \{u, v\}$ with $u \in V_1$ and $v \in V \setminus V_1$. We know that there is a path

$$w_0 = u \to w_1 \to w_2 \to \ldots \to w_r \to v = w_{r+1}$$

in $T$. Let $e = \{w_i, w_{i+1}\}$ be the first edge of this path, which connects a node in $V_1$ with a node in $V \setminus V_1$.

---

$^2T$ is a spanning tree of $G$, if $T$ is a tree containing all nodes of $G$. 
11.2. MINIMUM SPANNING TREES

We remove \( e \) and insert \( e_1 \) instead. Observe that we have again obtained a tree, but with smaller weight than \( T \), a contradiction.

\[\square\]

**Example 11.2** We run an ideal parallel version of Boruvka’s algorithm on the graph

\[
\begin{array}{c}
\text{\includegraphics[width=0.5\textwidth]{example112}}
\end{array}
\]

and obtain two super nodes after merging singleton trees in the first phase of Boruvka’s algorithm.

\[
\begin{array}{c}
\text{\includegraphics[width=0.5\textwidth]{example112a}}
\end{array}
\]

The first super node consists of the set \( S_1 = \{v_2, v_3, v_4, v_5\} \) of nodes and the second super node consists of the set \( S_2 = \{v_1, v_6\} \). In the second iteration we choose the edge with weight 4 connecting the two super nodes which we can subsequently merge to obtain the minimum spanning tree.

\[
\begin{array}{c}
\text{\includegraphics[width=0.5\textwidth]{example112b}}
\end{array}
\]
We still have to organize the merging step and do so by utilizing the ideas of Algorithm 11.1. We determine a minimal weight external edge for each tree in the current forest and assign genders to all trees. Any male tree, which is linked to a female tree by its minimal weight external edge is then attached as a subtree of the female tree. In order for any node to recognize its current tree as fast as possible we again use the star representation of forests. Finally we choose the extended priority mode of a CRCW-PRAM to determine a minimal weight external edge: the processor trying to write the smallest value wins.

Algorithm 11.6 Computing a minimum spanning tree.

/* The undirected, connected graph $G = (V, E)$ with $|E| = m$ is given. We work with $m$ processors, i.e., we have one processor per edge. */

(1) for $i = 1$ to $n$ pardo parent[$i$] = $i$;
/* We initialize the collection of singleton stars. Again we call an edge alive, if its two endpoints belong to different stars. */

(2) while (at least one edge of $G$ is alive) do

(a) for (all edges $e = \{u, v\}$) pardo

The processor assigned to $e$ checks if the endpoints of $e$ belong to different stars. If this is the case, then it writes weight($e$) into the register of its root via the priority mode.

The processor then checks the register of the root to determine if it has won, i.e., if the weight stored in the register coincides with the weight $w(e)$ of its edge. If this is the case, then it write the edge $e$ into the register.

(b) each root $w$ chooses a gender $g(w) \in \{0, 1\}$ at random. If its gender is male, then it checks whether the external endpoint of the winning edge $e$ is female and discards $e$ otherwise.
/* We again apply the method of symmetry breaking, but after determining shortest crossing edges. */

(c) for $i = 1$ to $n$ pardo

parent[$i$] = parent[parent[$i$]];
/* One round of pointer jumping repairs the star property. */

Theorem 11.7 Algorithm 11.6 determines a minimal spanning tree on a CRCW-PRAM in extended priority mode. $m$ processors suffice and its running time is bounded by $O(\log_2 n)$ with probability at least $1 - \frac{1}{n}$.

Exercise 113
A connected undirected graph $G = (V, E)$ is given as well as weights on its edges. In the all-pairs-most shallow path problem we have to determine for each pair of nodes $u, v \in V$ a path from $u$ to $v$ whose maximal weight edge is as small as possible.
11.3. MAXIMAL INDEPENDENT SETS

(a) Show that there is a tree such that the unique path between any two nodes \( u \) and \( v \) is the most shallow path between \( u \) and \( v \).

(b) Solve the all-pairs-most shallow path problem in time \( O(\log_2 |V|) \) with \( |V| + |E| \) processors on a CRCW PRAM with priority mode.

Exercise 114
We are given a complete undirected graph \( C_n = (V, E) \) with \( n \) nodes and a weight function \( \text{weight}: E \to \mathbb{R}_{\geq 0} \). We require that the triangle inequality

\[
\text{weight}(u, v) \leq \text{weight}(u, w) + \text{weight}(w, v)
\]

is satisfied for all nodes \( u, v \) and \( w \). Our goal is to determine a tour which visits all nodes exactly once and whose length is at most twice as large as the minimal tour. Your solution should run on a EREW-PRAM with \( |V|^2 \) processors in time \( O(\log^2 |V|) \).

Hint: Show that the weight of a minimum spanning tree is less than or equal to the length of a minimal tour. How do you transform a spanning tree into a not much longer tour? Note that the traveling salesman problem, namely to determine a tour of minimal length, is \( \mathcal{NP} \)-complete.

11.3 Maximal Independent Sets

We say that \( I \) is an independent set, if no two nodes of \( I \) are connected by an edge. Determining a largest independent set is a hard problem. In particular it is known that the language

\[
\{(G, k) \mid G \text{ has an independent set of size at least } k \}\]

is \( \mathcal{NP} \)-complete. We say that \( I \) is a maximal independent set, if \( I \) is not contained in a larger independent set, i.e., each node in \( V \setminus I \) is connected to a node in \( I \). Thus whereas a largest independent set corresponds to a global maximum, a maximal independent set corresponds to a local maximum.

Sequential solutions are trivial as the following program shows.

Algorithm 11.8 A simple heuristic

\[
I := \emptyset; \\
\text{for } v = 1 \text{ to } n \text{ do} \\
\quad \text{If (\( v \) is not connected with a node from \( I \)) then} \\
\quad \quad I = I \cup \{v\};
\]
Algorithm 11.8 is trivial, but seems to be inherently sequential, since we cannot predict in early stages if a node is eventually included into $I$. Indeed, we show in Section 12.3.4 that the for-loop of Algorithm 11.8 cannot be parallelized: the problem to determine whether a node $v$ of $G$ belongs to $I$ has no fast parallel algorithms even if we are allowed to compute with a polynomial number of processors. But is it at least possible to compute some maximal independent set? We define

$$N(u) := \{ v \in V \mid \{v, u\} \in E \}$$

as the set of neighbors of node $u$ and extend this notion to subsets $X \subseteq V$ by defining

$$N(X) := \bigcup_{u \in X} N(u)$$

as the set of neighbors of $X$. $d(u) = |N(u)|$ is the degree of node $u$.

**Algorithm 11.9 A parallel Independent Set Algorithm.**

1. Set $I = \emptyset$;
2. while $V \neq \emptyset$ do
   2a) for $v \in V$ pardo
      if ($v$ is isolated) then
         $I = I \cup \{v\}; V = V \setminus \{v\};$
         else mark $v$ with probability $\frac{1}{2d(v)}$;
   2b) for $\{u, v\} \in E$ pardo
      if ($u$ and $v$ are both marked) then
         unmark the node of smaller degree, resp. unmark both nodes if their degrees coincide;
   2c) /* Let $X$ be the set of nodes which are still marked. */
      Set $I = I \cup X; V = V \setminus (X \cup N(X))$;
      /* The set $I$ is independent, but possibly not maximal. We have removed $X$ and the set of its neighbors to prepare the next iteration. Observe that degrees have to be recomputed after any iteration. */

Algorithm 11.9 performs symmetry breaking by randomly selecting nodes. The selection probability of a node is inversely proportional to its degree and we hope of course that the set $X$ of step (2c) is almost independent.

**Example 11.3** Let $G$ be a cycle of length $n$. Then each node has degree 2 and we mark nodes with probability $\frac{1}{4}$. Observe that a node $i$ is inserted into $X$ iff neither $i - 1$ nor $i + 1$ is marked and this happens with probability $\left(\frac{3}{4}\right)^2$. Thus we expect $|I| = \frac{1}{4} \cdot \left(\frac{3}{4}\right)^2 \cdot n \geq \frac{n}{8}$ to hold after the first round.
Exercise 115
Determine the expected number of iterations for the cycle of length $n$.

If marked nodes are connected to each other, then we remove the node of small degree. Why not remove the node of large degree? If a node of high degree survives, then we can eliminate all of their many neighbors in step (2c) and thus considerably reduce the number of nodes we have to consider.

Example 11.4 This hope is deceptive. We consider a complete bipartite graph with $n^{1/4}$ nodes in the first layer and $n - n^{1/4}$ nodes in the second layer. It is highly probable that Algorithm 11.9 does not choose any high-degree node from the first layer and about $n^{3/4}/2$ nodes from the second layer. In this case only the $n^{1/4}$ nodes of the first layer will be removed from $X$.

This is bad news, however the bad news turn into good news if we observe that we have removed all edges by removing all layer 1 nodes.

We now show that the above examples are quite characteristic, namely that we can expect

1. about 50% of all marked nodes to survive
2. and to remove a constant percentage of all edges per iteration.

Observe that property (2) guarantees an expected number of logarithmically many iterations and Algorithm 11.9 would serve its purpose.

Lemma 11.10 Any marked node $v$ is inserted into $I$ with probability at least $1/2$.

Proof: A marked node $v$ is not inserted into $I$, if a neighbor is marked, whose degree is at least as high as $v$. Let $p$ be the probability of such an event and set

$$\text{dangerous}(v) = \{ u \in N(v) \mid d(u) \geq d(v) \}.$$ 

Then we get

$$p \leq \sum_{u \in \text{dangerous}(v)} \text{prob}[u \text{ is marked}] = \sum_{u \in \text{dangerous}(v)} \frac{1}{2 \cdot d(u)} \leq \sum_{u \in \text{dangerous}(v)} \frac{1}{2 \cdot d(v)} \leq \frac{d(v)}{2 \cdot d(v)} = \frac{1}{2},$$

and this was to be shown. $\square$

The notion of good nodes and good edges are crucial.
Definition 11.11 A node $v$ is called good iff at least one third of its neighbors have a degree of at most $d(v)$. An edge is good, if at least one endpoint is good.

Example 11.4 shows that there are graphs with only very few good nodes. However we will show later that most edges are good. Thus we have made substantial progress, if we can show that a large proportion of good nodes is removed from $V$ and this is what we show next. If $v$ is good, then $v$ has $d(v)/3$ neighbors of degree at most $d(v)$ and with probability at most

$$p = (1 - \frac{1}{2} \cdot d(v)/3)^{d(v)/3}$$

none of these neighbors is marked. We apply Lemma 1.2 and obtain

$$p \leq e^{-1/6}.$$ 

Thus, with probability at least $1 - e^{-1/6}$, at least one such neighbor $w$ is marked. We already know with Lemma 11.10 that a marked node is inserted into $I$ with probability at least $\frac{1}{2}$ and hence the good node $v$ is removed from $V$ with probability at least $\frac{1-e^{-1/6}}{2}$.

Lemma 11.12 Let $v$ be a good node with at least one neighbor. Then $v$ is removed from $V$ after one iteration with probability at least $\frac{1-e^{-1/6}}{2}$.

Thus we expect to lose a constant fraction of all good nodes. Our next goal is therefore to show that most edges are good, i.e., are incident to at least one good node.

Lemma 11.13 At least one half of all edges is good.

Proof: A bad node $v$ has by definition fewer than $d(v)/3$ neighbors of degree not exceeding its own degree and hence such a node has at least twice the number of higher-degree than lower-degree neighbors.

We direct edges from a low-degree to a high-degree endpoint. If a (now directed) edge $e = (u, v)$ is bad, then we can assign injectively two further edges $(v, u_1), (v, u_2)$ to $e$, since the bad node $v$ has double the number of higher-degree neighbors.

Thus we have constructed an injection $g$ from the set of bad edges into the set of all pairs of edges and $g$ assigns two edges to any bad edge. But this is only possible, if there are at most $\frac{|E|}{2}$ bad edges.

Let $F$ be the set of all edges before an iteration (2) in Algorithm 11.9 and let $F'$ be the set of edges which still belong to the graph after the iteration. At least one half of all edges in $F$ has a good node as endpoint. But a good node is eliminated in one iteration with probability at least $c = \frac{1-e^{-1/6}}{2}$ and hence we get $E[|F'|] \leq (1 - c) \cdot |F|$.

Exercise 116 Show that Algorithm 11.9 stops after an expected number of logarithmically many iterations.
Theorem 11.14 Algorithm 11.9 determines a maximal independent set with an expected number of $O(\log_2 n)$ iterations for graphs with $n$ nodes and $m$ edges. $n + m$ processors on a CRCW-PRAM are used.

Exercise 117
Let $G$ be an undirected graph with $n$ nodes and $m$ edges. A matching of is a subset $M \subseteq E$ of node-disjoint edges. We say that a matching $M$ is locally optimal, if no proper superset $M \cup \{e\}$ is a matching. A matching is called optimal, if $M$ is a matching with a maximal number of edges.

(a) Let $M$ be a locally optimal matching and let $M'$ be an optimal matching. Show $|M'| \leq 2 \cdot |M|$.

(b) In a group of $n$ people each person knows at least one and at most two persons. Each person decides at random to shake hands with a person she knows. However, a hand shake only counts, if both involved persons choose each other. Show that the expected number of hand shakes is at least $n/4$.

(c) We again consider a group of $n$ people, but now assume no upper bound on the number of acquaintances per person. This time a person first offers her left hand to a randomly chosen acquaintance, and in a second step, shakes a randomly chosen left hand offered to her with her right hand. In the third and last step each person which is involved in two handshakes withdraws one of her hands at random.

Now we have formed couples which remain stable. We iterate this procedure until no more new couples can be formed. Show that this procedure determines at least $\text{opt}/2$ couples, where $\text{opt}$ is the optimal number of couples that can conceivably be formed.

(d) The procedure from part (c) can be implemented on a randomized CRCW PRAM to determine a locally optimal matching with $n + m$ processors. Thus the implementation extends the current matching in each iteration by a matching $M$. Show for a good node (see Definition 11.11), that

$$\text{prob}[v \text{ is endpoint of an edge from } M] \geq 1/4$$

holds.

(e) Show that the expected time of the CRCW PRAM implementation is bounded by $O(\log_2 n)$.

11.4 Coloring

Our goal is to color nodes of an undirected graph with as few colors as possible such that no two adjacent nodes share the same color. We start with coloring very simple graphs, namely cycles which we interpret as circular lists.

11.4.1 List Coloring

In the list coloring problem, we have to color a singly linked, circular list with three colors such that neighboring list elements receive different colors. (Observe that there is no coloring with two colors, if the list has an odd number of elements.) The list coloring problem appears to be simple, but our goal is to achieve an almost constant running time.

We assume that the singly linked, circular list is represented by a successor array $S$, where $S[i]$ specifies the successor of $i$ in the list. Our first goal is to find a coloring with
few colors such that adjacent list elements receive different colors. Assume that the list consists of $n$ elements. We start with an $n$-coloring by assigning color $i$ to element $i$ and set $\text{color}[i] = i$.

**Algorithm 11.15 Reducing the number of colors**

for $i = 1$ to $n$

Determine a bit position $k$ in which $\text{color}[i]$ and $\text{color}[S[i]]$ differ. Let $b$ be the $k$th bit in the binary representation of $\text{color}[i]$, where we count bit positions starting from the least significant bit.

Set $\text{color}^*[i] = 2 \cdot (k - 1) + b$.

**Lemma 11.16** Assume that the list has $n$ elements which are legally colored with $D$ colors from the set \{0, $\ldots$, $D - 1$\}. Then Algorithm 11.15 finds a legal coloring with at most $d = 2 \cdot \lceil \log_2 D \rceil$ colors from the set \{0, $\ldots$, $d - 1$\}. It runs on a EREW-PRAM in constant time with $n$ processors. Here we assume that a differing bit position can be found in time $O(1)$.

**Proof:** We show by contradiction that the new coloring is legal. Assume that there is a list element $i$ with $\text{color}^*[i] = \text{color}^*[S[i]]$. By definition of $\text{color}^*$, there are $k$ and $l$ with

$$\text{color}^*[i] = 2(k - 1) + b \quad \text{and} \quad \text{color}^*[S[i]] = 2(l - 1) + c.$$

Since $b, c \in \{0, 1\}$, we have $b = c$ and consequently $k = l$. But $\text{color}[i]$ and $\text{color}[S[i]]$ differ in their $k$th bit. We have $k = l$ and $\text{color}[i], \text{color}[S[i]]$ agree in their $k$th bit position after all.

How many colors are used? The original colors have binary representations of length at most $\lceil \log_2 D \rceil$ and hence we use at most $2 \cdot (\lceil \log_2 D \rceil - 1) + 2 = 2 \cdot \lceil \log_2 D \rceil$ new colors. An EREW implementation is obvious, since each processor just has to read the color of his right neighbor. □
11.4. COLORING

Example for Algorithm 11.15

To come up with a 3-coloring, we repeatedly apply Algorithm 11.15. Observe however that we reduce the number of colors only if $2 \cdot \lceil \log_2 D \rceil < D$ and this is the case for $D \geq 7$.

Algorithm 11.17 Determining a 3-coloring

1. Set $\text{color}[i] = i$;
2. Apply Algorithm 11.15 as long as at least seven colors are used.
3. /* Each list element is now colored with a color from the set $\{0, \ldots, 5\}$. Next we remove the colors 3, 4 and 5. */
   
   for $f = 3$ to 5 do
     
     for $i = 1$ to $n \text{ pardo}$
       
       if $\text{color}[i] = f$ then
         
         choose a color $f^* \in \{0, 1, 2\}$ which is not used by a neighbor and
         
         set $\text{color}[i] = f^*$.

Definition 11.18 For $x > 1$ we set $\log^{(1)}(x) := \log_2 x$ and $\log^{(k)}(x) := \log_2 \left( \log^{(k-1)}(x) \right)$ for $k > 1$. Finally set $\log^*(x) := \min\{i \in \mathbb{N} \mid \log^{(i)}(x) \leq 1\}$.

$\log^*(x)$ is an extremely slow growing function: We have $\log^*(2) = 1$, $\log^*(2^2) = 2$, $\log^*(2^{2^2}) = 3$ and $\log^*(2^{2^{2^2}}) = 4 \geq \log^*(64.000)$. Hence we have $\log^*(x) < 5$, whenever $x \leq 2^{64.000}$, and the $\log^*$ function is for all practical purposes bounded by 5.
Theorem 11.19 Algorithm 11.17 determines a legal 3-coloring for a list of \( n \) elements in time \( O(\log^* n) \) with \( n \) processors. It can be implemented on an EREW-PRAM.

Proof: In step (3) we also need to know the color of our left neighbor. This is easily achieved, if each processor communicates its color to its right neighbor.

Algorithm 11.17 is obviously correct and we only have to determine the running time. The number of colors is reduced from \( D \) to \( 2 \cdot \lceil \log_2 D \rceil \leq 2 (\log_2 D + 1) = 2 \log_2(2D) \).

Applying Algorithm 11.15 twice brings a reduction to at most

\[
2 \cdot \log_2(4 \cdot \log_2(2D)) \leq 2 \cdot \log_2 \log_2(2D) + 4 \leq \log_2 D \quad \text{for} \quad D \geq 2^{12}.
\]

Hence Algorithm 11.15 is applied at most \( (2 \log^* n) \) times until \( D \leq 2^{12} \) is reached. Then a constant number of applications suffices. \( \square \)

We now show how to obtain an optimal 3-coloring algorithm, if we allow time \( O(\log_2 n) \).

We apply Algorithm 11.15 only once to obtain a legal coloring with \( O(\log_2 n) \) colors and then construct a 3-coloring by sequentially replacing all colors from 3 onwards.

Algorithm 11.20 An optimal 3-coloring

1. Set \( \text{color}[i] = i \). We work with \( p = \frac{n}{\log_2 n} \) processors.

2. Apply Algorithm 11.15 once.

3. Sort the list elements according to their color in time \( O(\log_2 n) \) with an EREW-PRAM algorithm.

4. for \( f = 3 \) to \( 2 \cdot \lceil \log_2 n \rceil \) do

   Distribute the elements with color \( f \) evenly among the \( p \) processors.

   for \( i = 1 \) to \( p \) pardo

   Processor \( i \) inspects all of its elements sequentially and chooses each time a color different from the color of the two neighbors.

We show in section 11.4.3 how to implement step (3) with \( \frac{n}{\log_2 n} \) processors. Thus we obtain

Theorem 11.21 Algorithm 11.20 determines a 3-coloring for a singly linked, circular list in time \( O(\log_2 n) \) with \( \frac{n}{\log_2 n} \) processors on an EREW-PRAM.
11.4. COLORING

11.4.2 Coloring Graphs

The coloring problem for graphs \( G = (V, E) \) is hard, since it is known that the language

\[ \{(G, k) \mid G \text{ can be colored with at most } k \text{ colors}\} \]

is \( \mathcal{NP} \)-complete. However, we are content with good and not necessarily optimal solutions and aim at coloring graphs of maximal degree \( d \) with at most \( d + 1 \) colors. It is known that such a coloring always exists and can be found by a simple greedy algorithm. (Observe that \( d + 1 \) colors are also required in some cases, since a clique\(^3\) on \( d \) nodes has degree \( d - 1 \) and requires \( d \) colors.)

Example 11.5 Assume that we have a collection of \( n \) processes which compete for resources such as printers or disk accesses. We connect two processes by an edge, if both require the same resource. Assume now that we have found a coloring with \( c \) colors. All nodes with the same color form an independent set and hence can be executed in parallel. Thus all processes can be executed within \( c \) scheduling steps.

Our algorithmic solution is based on Algorithm 11.17 which 3-colors a cycle of \( n \) nodes in time \( O(\log^* n) \) with \( n \) processors (see Theorem 11.19). In particular we proceed now as follows.

- We direct all edges in an arbitrary way and label all edges leaving a node with numbers from \( \{1, \ldots, d\} \).
- The subgraph of all edges with the same fixed label is a “pseudo-forest”, i.e., a directed graph of fan-out one. Show that any pseudo-forest can be 3-colored with a modified version of Algorithm 11.17.
- Thus the graph is now decomposed into \( d \) pseudo-forests, each of which has been 3-colored. Show how to obtain a \( d + 1 \)-coloring by suitably compressing the \( 3 \cdot d \) colors.

We assume that edges are directed and try to color a pseudo-forest with 3-colors. As for Algorithm 11.17 we start with a coloring which assigns color \( i \) to node \( i \) and then subsequently reduce the number of colors radically. First we adapt the reduction procedure used by Algorithm 11.17 as follows.

Algorithm 11.22 Coloring pseudo-forests: Reducing the number of colors.

\[
\text{for } i = 1 \text{ to } n \text{ pardo}
\]

\(^3\)A clique is an undirected graph such that any two nodes are connected by an edge.
if \( (i \) has no successor) then
\[
\text{set color}[i] = \begin{cases} 
0 & \text{color}(i) \text{ is even}, \\
1 & \text{otherwise}; 
\end{cases}
\]
else
\[
/* \text{Let } j \text{ be the unique successor of } i. \text{ Determine a bit position } k \text{ in which color}[i] \\
\text{and color}[j] \text{ differ. Let } b \text{ be the } k\text{th bit in the binary representation of color}[i], \\
\text{where we count bit positions starting from the least significant bit.} */
\]
\[
\text{set color}^*[i] = 2 \cdot (k - 1) + b;
\]

**Exercise 118**
Assume that we start with a \( k \)-coloring. Show that algorithm 11.22 determines a new coloring with \( 2 \cdot \lceil \log_2 k \rceil \) colors.

Thus we have obtained the radically reduced coloring as promised and can color pseudo-forests after \( O(\log^* n) \) rounds of reducing the number of colors. All we have to do is to run Algorithm 11.17, where we replace calls of Algorithm 11.15 by calls of Algorithm 11.22 and adjust the final compression to three colors.

**Algorithm 11.23 A 3-coloring for pseudo-forests**

1. Set color\([i] = i;\)
2. Apply Algorithm 11.22 as long as at least seven colors are used.
3. /* Each node is now colored with a color from the set \( \{0, \ldots, 5\} \). Next we remove the colors 3, 4 and 5. */
   for \( i = 1 \) to \( n \) pardo
   if \( (i \) has no successor) then color\([i] = 5 - \text{color}[i]; \)
   else color\([i] = \text{color}[S[i]]; \)
   /* \( S[i] \) is the unique successor of \( i. \) We have chosen a new legal coloring such that all immediate predecessors of a node have the same color. If \( i \) has no successor, then we have assigned a new color guaranteeing that our revised coloring stays legal. */
   for \( f = 3 \) to \( 5 \) do
   for \( i = 1 \) to \( n \) pardo
   if color\([i] = f \) then
   set color\([i] = f^* \), where color \( f^* \in \{0, 1, 2\} \) is not used by some neighbor;

We can now formulate the coloring algorithm for graphs.

**Algorithm 11.24 Coloring graphs of degree \( d \) with \( d + 1 \) colors.**

/* \( G = (V, E) \) is a graph of degree at most \( d. \) */
11.4. COLORING

(1) for all edges \{u, v\} ∈ E pardo
direct \{u, v\} arbitrarily;
for all v ∈ V pardo
label all edges leaving v injectively with numbers from \{1, ..., d\};

(2) for f = 1 to d do
color the pseudo-forest of f-edges with three colors
from \{3 \cdot (f - 1), 3 \cdot (f - 1) + 1, 3 \cdot (f - 1) + 2\}.
// We have computed a 3d-coloring, which we now compress into a d + 1 coloring.

(3) Sort nodes according to their color.

(4) for f = d + 1 to 3d do
distribute nodes with color f evenly among processors and
replace their color with a new legal color in \{0, ..., d\}.

We work with n processors, where n is the number of nodes. Step (1) runs in time O(d),
since up to d · n edges have to be processed. Step (2) runs in time O(d · log n), since d
pseudo-forests have to be 3-colored. Step (3) is cheap as we will see in section 11.4.3: time
O(\log_2 n) suffices.

Exercise 119
Show how to implement step (4) in time O(d) using the priority write resolution scheme. You may
assume that the elements of the adjacency list of a node v are stored in an interval of consecutive
registers and that the length of this interval is known.

Theorem 11.25 Algorithm 11.24 colors a graph G of n nodes and maximal degree d with
d + 1 colors. It runs in time O(\log_2 n + d) on an ERCW-PRAM with n processors.

Exercise 120
Determine a good implementation of step (4) for an EREW-PRAM.

11.4.3 A Parallel Counting Sort

The sequential algorithm Counting Sort sorts an array A of n keys from the set \{0, ..., n - 1\}
in linear time. First the number Count[i] of occurrences of key i is determined. Then we
sort A by writing Count[0] zeroes, followed by Count[1] ones etc.

```c
void counting_sort( )
{
    int Count[m]; int where = 0;
    for (int i = 0; i < m; i++) // Initializing Count.
```
Count[i] = 0;
for (int j = 1; j <= n; j++) // Computing Count.
    Count[A[j]]++;
for (i = 0; i < m; i++) //
{
    for (j = 1; j <= Count[i]; j++)
        A[where + j] = i;
    where += Count[i];
}

Thus Counting Sort runs in time $O(n + m)$. We first describe an efficient parallelization for $m = \log_2 n$ and $p = \frac{n}{\log_2 n}$ processors.

**Algorithm 11.26 Stable Sorting of $n$ numbers from \{0, \ldots, \log_2 n - 1\}.
**(1) for $i = 1$ to $p$ pardo

    processor $i$ determines the frequency $f_{i,j}$ of value $j$ in its subsequence $A[(i-1)\log_2(n)+1], \ldots, A[i\cdot\log_2 n]$. // Time $O(\log_2 n)$ and $p$ processors suffice.

(2) the processors determine all prefix sums for the following sequence in row-order

$$
\begin{align*}
    f_{1,0} & \quad f_{2,0}, & \ldots & \quad f_{p,0}, \\
    f_{1,1} & \quad f_{2,1}, & \ldots & \quad f_{p,1}, \\
    \ldots & \quad \ldots & \ldots & \quad \ldots \\
    f_{1,\log_2(n)-1} & \quad f_{2,\log_2(n)-1} & \ldots & \quad f_{p,\log_2(n)-1}
\end{align*}
$$

This is done with the parallel prefix algorithm after writing $f_{i,j}$ into register $i + j \cdot p$. // Time $O(\log_2 n)$ and $p$ processors suffice.

(3) /* register $i + j \cdot p$ stores now the number $s_{i,j}$ of elements less than $j$ plus the number of elements with value $j$ stored by processors $1, \ldots, i$. */

    processor $i$ writes $j$ into the registers with addresses in $[s_{i-1,j} + 1, s_{i-1,j} + f_{i,j}]$.

**Theorem 11.27** Algorithm 11.26 sorts $n$ numbers from the set \{0, \ldots, \log_2 n - 1\} in time $O(\log_2 n)$ with \frac{n}{\log_2 n} processors. It can be implemented as a stable\textsuperscript{4} sorting algorithm.

Hence we can sort $n$ numbers from the set \{0, \ldots, \log_2^k n - 1\} by running $k$ phases. In the first phase we sort the $n$ numbers, restricted to their least significant $\log_2 \log_2 n$ bits, with

\textsuperscript{4}A sorting algorithm is stable, if in the sorted sequence the “information” attached to element $A[i]$ appears before the “information” attached to element $A[j]$, provided $A[i] \leq A[j]$ and $i < j$. 
11.4. COLORING

We utilize its stability and run it a second time but now on the following \( \log_2 \log_2 n \) bits. We can iterate this process \( k \) times and have successfully sorted \( n \) numbers with \( \frac{n}{\log_2 n} \) processors in time \( O(k \cdot \log_2 n) \).

Can we do better for \( n \) numbers from the set \( \{0, \ldots, n - 1\} \)? We first assume an apparently unfair advantage, namely that we know upper bounds \( f_i \) on the number of times element \( i \) appears. We also demand that the upper bounds are quite good, namely we demand that

\[
\sum_{i=0}^{n-1} f_i = O(n)
\]

holds. The idea is now to reserve registers \( 1 + 2 \sum_{i=1}^{k-1} f_i, \ldots, 2 \sum_{i=1}^k f_i \) for value \( k \): any processor storing an element \( A[i] \) with \( A[i] = k \) tries to write into one of the registers reserved for \( k \) by choosing such a register at random. We say that the processor is successful, if no one else tries to write into the same register. A successful processor is happy, failing processors have to try until they are successful. There seems to be a good chance that processors with value \( k \) will all succeed rather quickly, since we reserved twice as many registers as there are requests. Once all attempts have succeeded, all we have to do is to remove gaps with the parallel prefix algorithm and we have sorted.

**Algorithm 11.28 A parallel Counting Sort with upper bounds**

/* The array \( A \) is to be sorted. We assume that frequencies \( f_0, \ldots f_{n-1} \) are known with \( \sum_{i=0}^{n-1} f_i = O(n) \). Frequency \( f_i \) is an upper bound on the number of elements with value \( i \), i.e., \( |\{i \mid x_i = j\}| \leq f_j \). We work with \( p = \frac{n}{\log_2 n} \) processors. */

1. Compute all prefix sums \( s_i = \sum_{i=0}^{n-1} f_i \).
2. for \( i = 1 \) to \( p \) pardo
   
   for \( j = 1 \) to \( \log_2 n \) do /* sequential for */

   processor \( i \) inspects \( k = A[(i - 1) \log_2(n) + j] \) and chooses a random number \( q \) with \( q \in \{2s_{k-1} + 1, 2s_{k-1} + 2, \ldots, 2s_k\} \);
   If the register stores a value different from a default value, then its attempt has failed. Otherwise it tries to write its address into register \( q \) and checks in the subsequent step whether its address is stored. If it is, it replaces its address by \( k \) and its attempt has been successful.
   
   If the attempt failed, the processor has to try again.

3. remove gaps with the parallel prefix algorithm.

How long does it take until each processor has successfully stored all of its \( \log_2 n \) numbers?
Lemma 11.29 Let $d \geq 4$ be arbitrary. Algorithm 11.28 solves the sorting problem with upper bounds in time $O(d \cdot \log_2 n)$ with $n \log_2 n$ processors. This statement holds with probability at least 

$$1 - n \cdot e^{-\frac{d}{16} \log_2 n}.$$ 

Proof: We have to worry about using too many attempts. But we know that the success probability is at least $\frac{1}{2}$, since the number of registers is at least twice as large as the number of elements to be stored.

We define the 0-1 random variables $Y_1, Y_2, \ldots, Y_{d \cdot \log_2 n}$ by setting $Y_i = 1$, if processor 1 was successful during its $i$th attempt to write one of its numbers. We define $P$ as the probability that some processor had to use more than $d \cdot \log_2 n$ attempts to successfully write all its $\log_2 n$ numbers. Define $P_1$ as the probability that processor 1 had to use more than $d \cdot \log_2 n$ attempts. Then

$$P \leq \frac{n}{\log_2 n} \cdot P_1$$

and we concentrate only on processor 1 from now on. Let’s check the expected behavior. We have

$$E\left[ \sum_{j=1}^{d \cdot \log_2 n} Y_i \right] \geq \frac{d}{2} \cdot \log_2 n$$

and can apply the Chernoff bound (Theorem 1.8), observing $\frac{2}{d} \leq \frac{1}{2}$ since $d \geq 4$,

$$P_1 \leq \operatorname{prob}\left[ \sum_{j=1}^{d \cdot \log_2 n} Y_j < \log_2 n \right]$$

$$\leq \operatorname{prob}\left[ \sum_{j=1}^{d \cdot \log_2 n} Y_j \leq \frac{2}{d} \cdot \frac{d}{2} \cdot \log_2 n \right]$$

$$\leq \operatorname{prob}\left[ \sum_{j=1}^{d \cdot \log_2 n} Y_j \leq \left(1 - \frac{1}{2}\right) \cdot \frac{d}{2} \cdot \log_2 n \right]$$

$$\leq e^{-\frac{1}{4} d \log_2 n}.$$ 

Thus we can conclude $P \leq \frac{n}{\log_2 n} \cdot e^{-\frac{d}{16} \log_2 n} \leq n \cdot e^{-\frac{d}{16} \log_2 n}$ and this was to be shown. $\square$

Our goal is to determine upper bounds for frequencies in expected time $O(\log_2^2 n)$ with $p = \frac{n}{\log_2^2 n}$ processors. We work probabilistically and draw a sample of $s = \frac{n}{n \log_2 n}$ numbers from the array $A$. We then sort the sample with the randomized quicksort in expected time $O(\frac{\log_2 s}{p} + \log_2^2 p) = O(\log_2^2 n)$. We will see that determining frequencies, restricted to the sample, is easy with a parallel prefix algorithm, since the sample is already sorted.

However it is quite likely that the sample frequencies are inaccurate and hence we take them only as a first rough guess and “round them up”. However the rounding process may
11.4. COLORING

make the sum of new frequencies too large. Therefore we restrict all values of \( A \) to belong to the set \( \{0, \ldots, m - 1\} \) and determine first for which values of \( m \) reliable upper bounds can be determined.

We fix the value of \( j \). Let \( f_j \) be the observed frequency of \( j \) in the sample and let \( h_j \) be the true frequency in \( A \). To investigate \( f_j \) we assume that the sample is determined by successively choosing \( s \) components from \( A \) with replacement. We therefore define random variables \( X_1, \ldots, X_s \) with \( X_i = 1 \), iff \( j \) is chosen as the \( i \)th element of the sample, and \( X_i = 0 \) otherwise. Obviously we have

\[
f_j = \sum_{i=1}^{s} X_i.
\]

The next step is an application of the Chernoff bound, utilizing that \( X_1, \ldots, X_s \) are independent random variables. We observe that \( E[f_j] = \frac{h_j}{n} \cdot s \) is the expected value of the sample frequency \( f_j \) and obtain

\[
\Pr\left[ \sum_{i=1}^{s} X_i \leq (1 - \beta) \cdot E\left[ \sum_{i=1}^{s} X_i \right] \right] \leq e^{-E[\sum_{i=1}^{s} X_i] \cdot \beta^2 / 2}.
\]

We set \( \beta = 1/2 \), utilize that \( f_j = \sum_{i=1}^{s} X_i \) as well as \( E[\sum_{i=1}^{s} X_i] = E[f_j] = \frac{h_j}{n} \cdot s \) hold and obtain

\[
\Pr\left[ f_j \leq \frac{1}{2} \cdot \frac{h_j}{n} \cdot s \right] \leq e^{-\frac{h_j}{n} \cdot s / 8}.
\]

The observed frequency \( f_j \) should be extrapolated as \( f'_j = \frac{n}{s} \cdot f_j \) and we get \( \Pr\left[ f'_j \leq \frac{1}{2} \cdot h_j \right] \leq e^{-\frac{h_j}{n} \cdot s / 8} = e^{-\frac{h_j}{n} / 8} \). However the estimate \( f'_j \) is insufficient for instance if \( h_j \) is small: in this case \( j \) might not even occur in the sample. Moreover we have \( m \) experiments, one for each value in \( \{0, \ldots, m - 1\} \), and errors accumulate. We therefore “round up” and choose

\[
f_j^* = 16 \cdot \max\{f'_j, \ln^2 n\}
\]

as upper bound.

- If \( h_j \leq 16 \ln^2 n \), then \( h_j \leq f_j^* \) holds with certainty.

- If \( h_j > 16 \ln^2 n \), then \( h_j > f_j^* \) holds with probability at most \( e^{-2 \ln n} = n^{-2} \). Thus \( h_j \leq f_j^* \) holds for all values \( j \) with probability at least \( 1 - \frac{1}{n} \).

- We have to bound the sum of upper bounds and get

\[
\frac{1}{16} \cdot \sum_{j=0}^{m-1} f_j^* \leq m \cdot \ln^2 n + \sum_{j=0}^{m-1} f_j = m \cdot \ln^2 n + \frac{n}{s} \cdot \sum_{j=0}^{m-1} f_j = m \cdot \ln^2 n + n.\]

We therefore choose \( m = \frac{n}{\ln^2 n} \).

**Algorithm 11.30 Upper-bounding frequencies**

/* The array \( A \) has only elements from the set \( \{0, \ldots, m - 1\} \), where \( m = \frac{n}{\ln^2 n} \). We work with \( p = \frac{n}{\log^2 n} \) processors. */
(1) draw a sample of \( s = \frac{n}{\ln n} \) elements from \( A \);

(2) sort the sample with the randomized quicksort.
   
   /* Time \( O(\log^2 n) \) with \( \frac{n}{\log^2 n} \) processors suffices. */

(3) to determine the frequency \( f^*_j \) of value \( j \) within the sample, apply the parallel prefix algorithm over the domain \( \mathbb{N} \times \{0, \ldots, \frac{n}{\log^2 n} - 1\} \) and define the associative operation \( * \) by

\[
(i, j) * (k, l) = \begin{cases} 
(i + k, j) & \text{if } j = l, \\
(k, l) & \text{otherwise.}
\end{cases}
\]

/* The pair \((i, j)\) expresses that so far \(i\) elements with value \(j\) appeared. The operation \(*\) is continuing the counting, as long as identical values meet and otherwise resets the counter. Before applying the parallel prefix algorithm, each element \(A[i]\) of the sample has to be replaced by \((1, A[i])\). */

(4) Let \( t_1, \ldots, t_s \) be the sequence of prefix sums.
   
   for \( j = 0 \) to \( m - 1 \) pardo
   
   \[
f_j := \begin{cases} 
i & \exists k \text{ with } t_k = (i, j), t_{k+1} = (i', j') \text{ and } j < j', \\
0 & \text{otherwise.}
\end{cases}
\]

(5) for \( j = 0 \) to \( m - 1 \) pardo
   
   \[
f^*_j = 16 \cdot \max\{\frac{n}{s} \cdot f_j, \ln^2 n\}.
\]

We summarize.

**Theorem 11.31** Let \( m = \frac{n}{\ln^2 n} \). Algorithm 11.30 sorts an array of length \( n \) with values in \( \{0, \ldots, m - 1\} \) with \( \frac{n}{\log^2 n} \) processors in time \( O(\log^2 n) \). This statement holds with probability at least \( 1 - \frac{1}{\ln n} \).

Finally we utilize that algorithm 11.30 is stable.

**Algorithm 11.32** A parallel Counting Sort for \( \{0, \ldots, n \cdot \log^2 n\} \)

(1) Sort the sequence, but restricted to the last \( \log_2 n - \log_2 \ln^2 n \) least significant bits. In particular

   determine upper bounds on frequencies with algorithm 11.30;
   sort with algorithm 11.28;

(2) sort according to the remaining \( O(k \cdot \log_2 \log_2 n) \) bits by applying the stable sorting algorithm 11.26 \( O(k) \) times.
Theorem 11.33 Algorithm 11.32 sorts $n$ numbers from the set $\{0, \ldots, n \cdot \log^k n\}$ with $\frac{n}{\log^2 n}$ processors. With probability at least $1 - \frac{1}{n}$ time $O(k \cdot \log^2 n)$ is sufficient.

Exercise 121
A summation CRCW-PRAM solves a write conflict by summing all values requesting the same register and assigning the sum to the register.

Show how to sort $n$ keys from the set $\{0, \ldots, n^2 - 1\}$ in time $O(\log n)$ with $\frac{n}{\log n}$ processors.

11.5 Conclusion

We have seen crucial applications of symmetry breaking in all problems we discussed, emphasizing that symmetry-breaking may be one of the most important algorithmic methods for graph problems. In all applications, with the exception of graph coloring, we have used randomization as “symmetry-breaker”.

Chapter 12

A Complexity Theory for Parallel Computation

What are “reasonable” parallel computation models? Is it possible that we overlooked reasonable parallel computation models that allow even faster parallel computations for important problems? What are limits of parallel computations: are there problems with efficient sequential computations which are hard to parallelize? Is it possible to relate parallel computations to sequential computations?

The next sections try to give some answers. In particular, we relate parallel time and sequential space and define the notion of $P$-completeness which allows to identify problems which are (apparently) hard to parallelize. We begin by introducing the space complexity of problems.

12.1 Space Complexity

We consider off-line Turing machines to define the space complexity of problems. An off-line Turing machine consists of

- an input tape which stores the input $w_1 \cdots w_n$ for letters $w_1, \ldots, w_n$ belonging to the input alphabet $\Sigma$. The dollar symbol is used as an end marker.

- a working tape which is infinite to the left and to the right. The tape alphabet is binary.

- an output tape which is infinite to the right. Cells store letters from an out alphabet $\Gamma$.

The input tape has a read-only head which can move to the left or right neighbor of a cell. At the beginning of the computation the head visits the cell storing the leftmost
letter of the input word. The working tape has a read/write head with identical movement options. At the beginning of the computation all cells store the blank symbol B. Finally the output tape has a write-only head which can move to the right only.

An off-line Turing machine computes by reading its input, performing computations on its working tape and printing onto its output tape. If the machine stops with output 1, then we say that the machine accepts the input; the machine rejects the input, if it writes a 0. For Turing machine $M$ and input alphabet $\Sigma$, let

$$L(M) = \{ w \in \Sigma^* \mid M \text{ accepts } w \}$$

be the language accepted by $M$.

Off-line Turing machines are ideally suited to define the space complexity of a sequential computation. For Turing machine $M$ let $\text{space}_M(w)$ be the number of different cells of the working tape which are visited during the computation of $M$ on input $w$. We define the space consumption of $M$ on input length $n$ and input alphabet $\Sigma$ as the worst-case over all inputs $w \in \Sigma^n$, i.e.,

$$\text{space}_M(n) := \max_{w \in \Sigma^n} \text{space}_M(w).$$

We can now introduce the space complexity classes.

**Definition 12.1**

(a) Let $s : \mathbb{N} \to \mathbb{N}$ be a function. Then

$$\text{DSPACE}(s) = \{ L \subseteq \{0,1\}^* \mid \text{L = L}(M) \text{ for a Turing machine } M \text{ with } \text{space}_M(n) = O(s(n)) \}$$

is the complexity class of all languages computable in (deterministic) space $s$.

(b) Instead of $\text{DSPACE}(\log_2 n)$ we use the abbreviation $\text{DL}$, deterministic logarithmic space.

$\text{DL}$ is the smallest non-trivial space complexity class, since we need logarithmic space to remember a single position within the input.

We are interested as well in the space complexity of nondeterministic Turing machines. For a nondeterministic Turing machine $M$ we define $L(M)$ as the set of inputs for which there is an accepting computation of $M$ and set

$$\text{nspace}_M(w) = \left\{ \begin{array}{l} \text{the maximal number of cells of the working tape, which are} \\ \text{visited during an accepting computation on input } w. \end{array} \right\}$$

as the nondeterministic space complexity of $M$ on $w$. The space consumption of $M$ on input length $n$ and input alphabet $\Sigma$ is then

$$\text{nspace}_M(n) := \max_{w \in \Sigma^n \cap L(M)} \text{nspace}_M(w).$$
12.1. SPACE COMPLEXITY

Definition 12.2  (b) Let \( s : \mathbb{N} \to \mathbb{N} \) be a function. Then

\[
\text{NSPACE}(s) := \{ L \subseteq \{0, 1\}^* \mid \text{L = L(M) for a nondeterministic Turing machine with nspace}_M(n) = O(s(n)) \}.
\]

(b) \( \text{NSPACE}(\log_2 n) \) is abbreviated by \( \text{NL} \).

Is there a relationship between \( \text{DL} \), \( \text{NL} \) and Nick’s class \( \text{NC} \)? We answer this question by first defining the computation graph of an off-line Turing machine.

Definition 12.3  Let \( M \) be a nondeterministic off-line Turing machine with space complexity at most \( s \). For an input \( w \) we define the computation graph \( G_M(w) \) as follows. The configurations\(^1\) of \( M \) on input \( w \) are the nodes of \( G_M(w) \). An edge leads from configuration \( k_1 \) to configuration \( k_2 \) iff \( k_2 \) is a potential immediate successor of \( k_1 \). An additional node \( \text{yes} \) is added and each accepting configuration \( k \) sends an edge into \( \text{yes} \).

Lemma 12.4  Let \( M \) be a nondeterministic off-line Turing machine.

(a) Let \( k_0 \) be the initial configuration of \( M \). Then

\( M \) accepts \( w \) \( \iff \) there is a path from \( k_0 \) to \( \text{yes} \) in \( G_M(w) \).

(b) If \( M \) has space complexity \( s(n) = \Omega(\log_2 n) \), then \( G_M(w) \) has at most \( 2^{O(s(n))} \) nodes and edges.

Exercise 122

Prove Lemma 12.4 (a).

How large is \( G_M(w) \), if \( w \) has length \( n \) and if \( M \) has space complexity \( s \)? If \( M \) has \( Q \) as its set of states, then \( G_M(w) \) has at most

\[ |Q| \cdot (n + 2) \cdot s(n) \cdot 2^{s(n)} \]

nodes, since there are \( n + 2 \) positions of the input head, \( s(n) \) positions of the working tape head and \( 2^{s(n)} \) possible contents of the working tape. Thus, if \( s(n) = \Omega(\log_2 n) \), then \( G_M(w) \) has at most \( 2^{O(s(n))} \) nodes and edges. \( \square \)

Theorem 12.5

(a) \( \text{DL} \subseteq \text{NL} \).

---

\(^1\)A configuration consists of the contents of the working tape, the current state and the positions of the input and working tape heads.
(b) Any language in NL can be accepted in time $O(\log^2 n)$ with a polynomial number of processors.

**Proof:** The inclusion $DL \subseteq NL$ is obvious, since a nondeterministic machine, given the same resources, is at least as powerful a deterministic machine. Hence it suffices to show (b). Let $M$ be a nondeterministic off-line Turing machine working in logarithmic space. If $|w| = n$, then $G_M(w)$ has poly($n$) nodes. We construct the computation graph $G_M(w)$ in logarithmic time with a polynomial number of processors: reserve one processor for any pair of configurations and let this processor check, whether the pair corresponds to a transition.

To determine whether $M$ accepts $w$, all we have to do is to check whether there is a path from $k_0$ to yes in $G_M(w)$. Our parallelization of Warshall's algorithm (Theorem 8.7) is not fast enough. Instead we use matrix multiplication to obtain a faster solution. Moreover we even work with a conventional $\{\neg, \land, \lor\}$-circuit.

**Lemma 12.6** The transitive closure of a graph directed $G$ with $n$ nodes can be computed by a circuit in depth $O(\log^2 n)$ and size $O(n^3 \cdot \log_2 n)$.

**Proof:** Let $A$ be the adjacency matrix of $G$, i.e.,

$$A[k_1, k_2] = \begin{cases} 1 & (k_1, k_2) \text{ is an edge of } G, \\ 0 & \text{otherwise.} \end{cases}$$

We also set the diagonal to one (i.e., insert self loops $(k, k)$ for all configurations $k$) and call the new matrix $B$. Let $B^r$ denote the $r$th power of $B$, where the logical OR replaces addition and the logical AND replaces multiplication. Then

$$B^r[k_1, k_2] = 1 \iff \text{there is a path in } G \text{ from } k_1 \text{ to } k_2 \text{ of length at most } r.$$ 

This equivalence is shown inductively. First observe that it is true for $r = 1$ due to the definition of $B$. Assume the equivalence holds for $r$. Then $B^{r+1}[k_1, k_2] = \bigvee_k B^r[k_1, k] \land B[k, k_2]$ by definition of the boolean matrix product. We apply the induction hypothesis and obtain

$$B^{r+1}[k_1, k_2] = 1 \iff \text{there is } k \text{ and a path in } G \text{ from } k_1 \text{ to } k \text{ of length at most } r. \text{ Moreover } (k, k_2) \text{ is an edge of } G.$$ 

This establishes the claim. But $G$ has $n$ nodes and hence $B^{n-1}[k_1, k_2] = 1$ iff $G$ has a path from $k_1$ to $k_2$. Thus it suffices to successively compute the matrix powers $B^2, B^4, \ldots, B^{2^t}$ with $t = \lceil \log_2 n \rceil$.

Each matrix product $B^{2^t+1}$ can be computed very fast: first compute all conjunctions $B^{2^t}[k_1, k] \land B^{2^t}[k, k_2]$ in parallel and then “sum up” conjunctions with a tree of OR’s. The summation step involves $n$ operands and therefore runs in time $\log_2 n$ with $O(n^3)$ gates. Thus one matrix multiplication requires $O(\log_2 n)$ steps and all matrix products can be computed in time $O(\log^2_2 n)$. $\square$
12.2 Circuit Depth Versus Space

We work with \{∧, ∨, ¬\}-circuits of fan-in two. We are particularly interested in the complexity parameters size and depth of the circuit. The size of \(S\) is the number of nodes of \(S\) and its depth is the length of the longest path. Thus depth is the appropriate parameter to measure the parallel computing time of \(S\).

A circuit family \((S_n \mid n \in \mathbb{N})\) specifies one circuit for each input size \(n \in \mathbb{N}\). We require that the description of the family is only moderately complex. In particular, we say that the circuit family is uniform if its circuits can be described by off-line Turing machines whose space complexity is logarithmic in the size of the circuit.

**Definition 12.7** Set \(S = (S_n \mid n \in \mathbb{N})\) be a family of circuits, where \(S_n\) is a circuit with \(n\) inputs and size \(s(n)\). We say that \(S\) is a uniform circuit family iff there is a deterministic off-line Turing machine \(M\) which, on input \(1^n\), eventually prints all edges of \(S_n\) onto its output tape as well as all nodes with their assigned operation, respectively with their assigned input bit. \(M\) has to have space complexity at most \(O(\log_2(n \cdot s(n)))\).

Observe that an off-line Turing machine \(M\) with space complexity \(t(n) \geq \log_2 n\) runs for at most \(2^{O(t(n))}\) steps, since otherwise \(M\) has to enter a configuration twice and thus enters a loop. Hence \(M\) can describe circuits of size at most \(2^{O(t(n))}\) and our definition of uniformity is quite restrictive, since we only allow the minimal space complexity. Is there a connection between circuit depth and space complexity?

**Theorem 12.8** Assume that \(s(n) = \Omega(\log_2 n)\). If \(M\) is a nondeterministic off-line Turing machine with space complexity \(s(n)\), then there is a uniform circuit family \((S_n \mid n \in \mathbb{N})\) with depth \(O(s^2(n))\) recognizing \(L(M)\).

**Proof:** We know that an input \(w\) belongs to \(L(M)\) iff there is a path in \(G_M(w)\) from the initial configuration \(k_0\) to yes. We compute the transitive closure of \(G_M(w)\) with the help of Lemma 12.6 in depth \(O(\log_2^2 N)\), where \(N = 2^{O(s(n))}\) is the number of nodes of \(G_M(w)\).

We are required to build a uniform circuit family. But configurations have length \(O(\log_2 n + s(n))\) and hence space \(O(\log_2(n) + s(n))\) is sufficient to output all valid transitions between configurations on input \(1^n\). \(\square\)

But we can also simulate circuits of small depth by off-line Turing machines of small space.

**Theorem 12.9** If the uniform family \((S_n \mid n \in \mathbb{N})\) consists of circuits of depth \(s(n)\), then there is a deterministic off-line Turing machine with space complexity \(O(s^2(n))\) which accepts exactly those words which are accepted by the circuit family.
Proof: We describe an off-line Turing machine \( M \) which accepts exactly those inputs \( w \) which are accepted by \( S_{|w|} \). Set \( n = |w| \). Since \( S_n \) has depth \( s(n) \), the circuit has at most \( 2^{s(n)+1} - 1 \) nodes. But we may evaluate \( S_n \) with depth-first search using a stack of height \( O(s(n)) \). Since the stack holds nodes from the circuit, all we need is space complexity \( O(s^2(n)) \).

Exercise 123
Show that the space complexity of the simulating Turing machine of Lemma 12.9 can be improved to \( O(s(n)) \).

12.2.1 The Parallel Computation Thesis

By Lemma 12.8 and Lemma 12.9 we have a quadratic relationship between the depth of circuits and the space complexity of Turing machines. One can also show that circuit families and MPI algorithms, resp. parallel register machines are polynomially related, when comparing depth and parallel time, and the conjecture is natural that

parallel time for any reasonable parallel computing model is polynomially related to the space complexity for sequential computations.

This conjecture is exactly the contents of the parallel computation thesis which classifies any parallel computing model violating the thesis as unreasonable.

Observe however that the parallel computation thesis does not restrict the “size” of a parallel machine and the polynomial relationship between space and parallel time can in general only be achieved with parallel machines of exponential size.

12.3 \( \mathcal{P} \)-completeness

We remember that all languages in \( \mathcal{NC} \) can be accepted efficiently by a sequential computation.

Lemma 12.10 \( \mathcal{NC} \subseteq \mathcal{P} \).

Which languages in \( \mathcal{P} \) are hard to parallelize? We are not able to answer this question, however we can show that there are languages which are the hardest to parallelize: if any such language belongs to \( \mathcal{NC} \), then \( \mathcal{NC} = \mathcal{P} \). Since \( \mathcal{NC} \neq \mathcal{P} \) holds in all likelihood, we have strong evidence that a hardest language does not belong to \( \mathcal{NC} \). To compare the hardness of languages we define a reduction between languages.

Definition 12.11 We say that language \( L_1 \) is reducible to language \( L_2 \) (\( L_1 \leq_{\text{par}} L_2 \)), iff there is a “translation” \( T \) such that

\[
 w \in L_1 \iff T(w) \in L_2.
\]
Moreover we demand that $T$ is computable in poly-logarithmic time by an MPI algorithm with a polynomial number of processors.

Which languages are the hardest languages?

**Definition 12.12** A language $K$ is called $\mathcal{P}$-hard iff $L \leq_{\text{par}} K$ for all languages $L \in \mathcal{P}$. $K$ is called $\mathcal{P}$-complete iff $K$ is $\mathcal{P}$-hard and $K$ belongs to $\mathcal{P}$.

We summarize the most important properties of our reduction.

**Lemma 12.13**

(a) If $L_1 \leq_{\text{par}} L_2$ and $L_2 \leq_{\text{par}} L_3$, then $L_1 \leq_{\text{par}} L_3$.

(b) Assume that $K$ is $\mathcal{P}$-hard and that $K \leq_{\text{par}} L$ holds. Then $L$ is $\mathcal{P}$-hard.

(c) Assume that $K \leq_{\text{par}} L$ holds. Then, if $L$ belongs to $\mathcal{NC}$, so does $K$.

(d) Assume that language $L$ is $\mathcal{P}$-hard. If $L$ belongs to $\mathcal{NC}$, then $\mathcal{NC} = \mathcal{P}$.

**Proof (a):** is obvious. (b) Let $M$ be an arbitrary language in $\mathcal{P}$. We know that $M \leq_{\text{par}} K$, since $K$ is $\mathcal{P}$-hard. But $K \leq_{\text{par}} L$ by assumption and we conclude $M \leq_{\text{par}} L$ with part (a). Since $M$ is an arbitrary language in $\mathcal{P}$, this shows that $L$ is $\mathcal{P}$-hard.

(c) Since $K \leq_{\text{par}} L$ holds, there is a translation $T$ with $w \in K \iff T(w) \in L$. Moreover the translation $T$ can be implemented by an MPI algorithm in poly-logarithmic time with a polynomial number of processors. We also know that $L$ belongs to $\mathcal{NC}$ and hence $L$ can be accepted by an MPI algorithm $T_L$ in poly-logarithmic time with a polynomial number of processors. Now to decide, whether an input $w$ belongs to $K$, we first run $T$ on $w$, pipe the result into $T_L$ and output the verdict of $T_L$. Hence $K \in \mathcal{NC}$.

(d) Since $L$ is $\mathcal{P}$-hard, we have $K \leq_{\text{par}} L$ for an arbitrary language $K$ in $\mathcal{P}$. If however $L$ belongs to $\mathcal{NC}$, then $K$ also belongs to $\mathcal{NC}$ with part (b). Thus in this case $\mathcal{P} \subseteq \mathcal{NC}$ and the claim $\mathcal{P} = \mathcal{NC}$ follows with Lemma 12.10. \qed

**Exercise 124**

We say that language $K$ is reducible to language $L$ with respect to constant depth ($K \leq_C L$) iff there is a uniform family $S_n$ of unbounded fan-in circuits of constant depth and polynomial size which accepts $K$. Moreover $S_n$ may have oracle gates which accept input $z$ iff $z \in L$.

(a) Show that $\leq_C$ is transitive.

(b) If $K \leq_C L$ holds, then $L$ requires asymptotically the same depth as $K$, provided only circuits of polynomial size are considered.

**Exercise 125**

We consider the following problems.
- MULTIPLICATION: determine the product of two $n$-Bit numbers.
- SUM: determine the sum of $n$ $n$-Bit numbers.
- BINARY COUNTING: determine the binary representation of $\sum_{i=1}^{n} x_i$ when summing $n$ bits.
- UNARY COUNTING: determine the unary representation of $\sum_{i=1}^{n} x_i$ when summing $n$ bits.
- SORTING: sorting $n$ $n$-bit numbers.
- MAJORITY: determine if $x = (x_1, \ldots, x_n)$ has at least as many ones as zeroes.

Show the following reductions:

(a) MULTIPLICATION $\leq_C$ SUM.
(b) SUM $\leq_C$ BINARY COUNTING.
(c) BINARY COUNTING $\leq_C$ UNARY COUNTING.
(d) UNARY COUNTING $\leq_C$ SORTING.
(e) SORTING $\leq_C$ MAJORITY.
(f) MAJORITY $\leq_C$ MULTIPLICATION.

Thus all problems are equivalent.

**Exercise 126**

We consider the following problems:

- UR: for a given undirected graph and two nodes $u$ and $v$, decide whether $u$ and $v$ belong to the same connected component of $G$.
- UCON: decide if a given undirected graph is connected.
- DR: for a given directed graph $G$ and two nodes $u$ and $v$, decide whether there is a path in $G$ from $u$ to $v$.
- DCON: decide if a given directed graph is strongly connected.

Show the following reducibilities:

(a) UCON $\leq_C$ UR.
(b) UR $\leq_C$ DR.
(c) DCON $\leq_C$ DR and DR $\leq_C$ DCON.
(d) PARITY $\leq_C$ UCON.
(e) DR $\leq_C$ 2-SAT.

### 12.3.1 The Circuit Value Problem

The input of the Circuit Value problem (CVP) consists of a circuit $S$ and an input $w$ of $S$. The pair $(S, w)$ belongs to CVP iff the circuit $S$ accepts $w$. (Hence we assume that $S$ has a single output node.)
Observe that CVP is a very easy problem for sequential computation, since circuit evaluation is immediate with graph traversals such as depth-first search. However a parallel evaluation in poly-logarithmic time seems hard for circuits of large depth.

**Theorem 12.14** CVP is \( \mathcal{P} \)-complete.

**Proof:** We already observed that CVP belongs to \( \mathcal{P} \) and it suffices to show that CVP is \( \mathcal{P} \)-hard. Thus we pick an arbitrary language \( L \) in \( \mathcal{P} \) and have to show that \( L \leq_{\text{par}} \text{CVP} \) holds. What do we know about \( L \)? There is a Turing machine \( M \) with a single tape which computes in polynomial time and accepts exactly the words in \( L \). Hence

\[
w \in L \iff M \text{ accepts } w.
\]

Let us assume that \( M \) runs in time \( t(n) \). Our goal is to simulate \( M \) by a uniform circuit family \( (S_n \mid n \in \mathbb{N}) \) of polynomial size such that

\[
M \text{ accepts } w \iff S_{|w|} \text{ accepts } w
\]

holds. Why does this suffice? We define the translation \( T \) by \( T(w) = (S_{|w|}, w) \) and

\[
w \in L \iff S_{|w|} \text{ accepts } w
\]

follows. Can we compute \( T(w) \) in poly-logarithmic time? Our construction will produce a uniform circuit family of polynomial size and hence there is an off-line Turing machine \( M^* \) which writes the description of circuit \( S_{|w|} \) onto its output tape and uses not more than logarithmic space. Thus the question, whether a pair of nodes is an edge, can be answered in logarithmic space. But \( \text{DL} \subseteq \mathcal{NC} \) with Theorem 12.5 and hence the translation can be computed within \( \mathcal{NC} \).

Assume that input \( w \) has length \( n \). Since \( M \) runs in time \( t(n) \), it visits only the cells with addresses

\[-t(n), -t(n) + 1, \cdots, -1, 0, +1, \cdots, t(n).
\]

Here we assume that letter \( w_i \) is written on the cell with address \( i \) and that the read/write head of \( M \) initially “sits” on the cell with address 0. We simulate \( M \) on input \( w \) with a circuit \( S_n \) which is built like a two-dimensional mesh. The \( i \)th “row” of \( S_n \) is supposed to reflect the configuration of \( M \) on input \( w \) at time \( i \) and hence its nodes correspond to the tape cells, express the current state of \( M \), respectively indicate the current position of the read/write head. Indeed we have to block the nodes into small groups \( S_{i,j} \) \((-t(n) \leq j \leq t(n))\) of constant size to enable the required coding.

How does row \( i+1 \) depend on row \( i \) and in particular, how are the groups \( S_{i+1,j} \) working? If cell \( j \) is not visited at time \( i \), then its contents has to remain unchanged. Obviously this can be easily enforced, since group \( S_{i+1,j} \) just has to “ask” group \( S_{i,j} \), whether the head was visiting. Since we assume here a negative answer, group \( S_{i+1,j} \) assigns the letter remembered by group \( S_{i,j} \) to cell \( j \).
If however cell \( j \) is visited at time \( i \), then this is reported by group \( S_{i,j} \). We assume in this case that \( S_{i,j} \) also stores the current state which it transmits together with the current contents of cell \( j \) to group \( S_{i+1,j} \). Now group \( S_{i+1,j} \) can determine the new contents of cell \( j \), the new state and the neighbor which will be scanned at time \( i + 1 \).

Observe that all groups \( S_{i,j} \) are identical for \( i \geq 1 \). Only the groups \( S_{0,j} \) vary, since cell \( j \) stores \( w_j \) for \( 1 \leq j \leq n \) and the blank symbol otherwise. Finally we have to add a binary tree on top of row \( t(n) \) which checks whether the final state is accepting. Due to its easy structure we can construct \( S_n \) with an off-line Turing machine within logarithmic space and the family \( (S_n \mid n \in \mathbb{N}) \) is therefore universal.

For later application we introduce variants of CVP. In \( M\text{-CVP} \) we assume a monotone circuit \( S \) and ask whether \( S \) accepts input \( w \). (A circuit is monotone, if it does not have negations.) \( M^2\text{-CVP} \) is defined as \( M\text{-CVP} \), however the circuit is required to have fan-out at most two. Finally we investigate \( NOR\text{-CVP} \), a variant of CVP, in which we assume that the circuit is built from NOR-gates only. (Remember that \( \text{NOR}(u,v) = \neg(u \lor v) \).)

We will reduce CVP to \( M\text{-CVP} \) and need to push negations of a given circuit down to the input bits.

**Lemma 12.15** Let \( S \) be a circuit. Then there is an equivalent circuit \( S' \) which uses negations only for inputs and which can be constructed in logarithmic space from \( S \). The size of \( S' \) at most doubles in comparison to \( S \).

**Proof:** We push negations in \( S \) to the inputs using the De Morgan rules
\[
\neg(x \land y) \equiv \neg x \lor \neg y \quad \text{and} \quad \neg(x \lor y) \equiv \neg x \land \neg y.
\]
However there is a problem, since the result of a gate may be needed in its negated as well as in its pure form. The equivalent circuit \( S' \) should therefore also provide both forms. In particular, the circuit \( S' \) has two gates \( u_{\text{pos}} \) and \( u_{\text{neg}} \) for any gate \( u \) of \( S \).

- If \( u \) is an input bit \( x_i \), then \( u_{\text{pos}} = x_i \) and \( u_{\text{neg}} = \neg x_i \).

- for an interior gate \( u \) of \( S \), \( u_{\text{pos}} \) keeps the functionality of \( u \), whereas \( u_{\text{neg}} \) interchanges \( \land \) and \( \lor \).

For any edge \((u,v)\) of \( S \), the equivalent circuit \( S' \) has two edges, namely \((u_{\text{pos}}, v_{\text{pos}})\) and \((u_{\text{neg}}, v_{\text{neg}})\). On the other hand, for any path \( u \rightarrow \neg \rightarrow v \) in \( S \), we insert the edges \((u_{\text{pos}}, v_{\text{neg}})\) and \((u_{\text{neg}}, v_{\text{pos}})\) into \( S' \).
12.3. \( \mathcal{P} \)-COMPLETENESS

Obviously \( S' \) can be constructed from \( S \) in logarithmic space. Since the number of nodes doubles, the claim follows.

\[\]

**Theorem 12.16**

(a) \( M\text{-CVP} \) is \( \mathcal{P} \)-complete.

(b) \( M_2\text{-CVP} \) is \( \mathcal{P} \)-complete.

(c) \( \text{NOR-CVP} \) is \( \mathcal{P} \)-complete.

**Proof:** All three problems obviously belong to \( \mathcal{P} \) and it suffices to show \( \mathcal{P} \)-hardness in each case. (a) Assume that \((S, w)\) is an input of CVP. By Lemma 12.15 there is an equivalent circuit \( S' \) with negations only at the input bits such that the transformation \( S \mapsto S' \) is computable by an off-line Turing machine in logarithmic space.

We define an input \( w_m \) which agrees with \( w \) on all not negated sources and disagrees with \( w \) on all negated sources. Finally we remove all negations from \( S' \) and obtain the monotone circuit \( S_m \). Observe that the function computed by \( S \) on \( w \) coincides with the function computed by the new monotone circuit \( S_m \) on input \( w_m \). Hence

\[ (S, w) \in \text{CVP} \iff S \text{ accepts } w \iff S_m \text{ accepts } w_m \iff (S_m, w_m) \in \text{M-CVP}. \]

Therefore the translation \((S, w) \mapsto (S_m, w_m)\) has the required properties and we have verified the reduction \( \text{CVP} \leq_{\text{par}} \text{M-CVP} \).

(b) It suffices to show the reduction \( \text{M-CVP} \leq_{\text{par}} \text{M}_2\text{-CVP} \). Assume that \((S, w)\) is an input for \( \text{M-CVP} \). We have to transform the monotone circuit \( S \) into an equivalent monotone circuit \( S^* \) such that no node has fan-out larger than two.

**Exercise 127**

Show that such a circuit \( S^* \) can be constructed by an MPI algorithm in poly-logarithmic time with a polynomial number of processors.

The translation \((S, w) \mapsto (S^*, w)\) establishes the reduction \( \text{M-CVP} \leq_{\text{par}} \text{M}_2\text{-CVP} \).
(c) We verify the reduction CVP ≤_par NOR-CVP. We observe first that
\[
\begin{align*}
- u & \equiv \text{NOR}(u, u), \\
 u \land v & \equiv \text{NOR}(-u, -v) \quad \text{and} \\
 u \lor v & \equiv \text{NOR}(\text{NOR}(u, v), \text{NOR}(u, v))
\end{align*}
\]
holds. Thus, if \((S, w)\) is an input of CVP, then replace all gates by small NOR-circuits to obtain an equivalent NOR-circuit \(S^*\). The translation \((S, w) \mapsto (S^*, w)\) establishes the wanted reduction to NOR-CVP.

Exercise 128
In the emptiness problem for context-free grammars (ECFG) we are given a context-free grammar \(G\) and we have to check whether the generated language \(L(G)\) is non-empty.
Show that ECFG is \(P\)-complete. Hint: verify the reduction M-CVP ≤_par ECFG.

12.3.2 Linear Programming

In linear programming a linear objective function
\[
\sum_{i=1}^{n} c_i x_i
\]
is to be minimized subject to linear constraints \(\sum_{j=1}^{n} A[i, j] \cdot x_j \geq b_i\) for \(j = 1, \ldots, m\) and to non-negativity constraints \(x_1 \geq 0, \ldots, x_n \geq 0\). The vector \(x = (x_1, \ldots, x_n)\) is a vector of real numbers. Thus, if \(A\) is the \(m \times n\) matrix with \(A = (A[i, j])_{1 \leq i \leq m, 1 \leq j \leq n}\), \(c = (c_1, \ldots, c_n)\) and \(b = (b_1, \ldots, b_m)\) is the vector of right hand sides, then we have to solve the minimization problem
\[
\min c^T \cdot x \quad \text{s.t.} \quad A \cdot x \geq b \quad \text{and} \quad x \geq 0.
\]
(12.1)
Linear programs have many important applications in computer science (flow problems, matching, scheduling) and business (logistics). Moreover, efficient sequential algorithms such as interior point methods exist and, in a sense, linear programming is the most powerful class of optimization problems with efficient algorithmic solutions. Thus the main question is the existence of parallel algorithms with substantial speedups, but, as we will see shortly, substantial speedups cannot be expected in general.

We begin our analysis by defining language versions of Linear Programming.

Definition 12.17

- In the Linear Inequalities problem (LIP) we are given an \(m \times n\) matrix \(A \in \mathbb{Z}^{m \times n}\) and a vector \(b \in \mathbb{Z}^m\). The pair \((A, b)\) belongs to LIP iff the system \(A \cdot x \geq b\) of linear inequalities has a solution.
In the Linear Programming Problem LPP we receive the same input as in LIP, but obtain additionally a vector \( c \in \mathbb{Z}^n \) and a rational number \( \alpha \). The quadruple \((A, b, c, \alpha)\) belongs to LPP iff there is a vector \( x \) that solves the linear system \( A \cdot x \geq b \) and also satisfies \( c^T x \leq \alpha \). Thus LPP is the language version of Linear Programming.

**Theorem 12.18**

(a) LIP is \( \mathcal{P} \)-complete, even if all parameters are either \(-1, 0\) or \(1\).

(b) LPP is \( \mathcal{P} \)-complete, even if all parameters are either \(-1, 0\) or \(1\).

**Proof:** Both LIP and LPP belong to \( \mathcal{P} \) due to efficient algorithms by Karmarkar [Ka84] and Khachiyan [Kh79] and hence it suffices to show \( \mathcal{P} \)-hardness. (a) We verify the reduction \( \text{M-CVP} \leq_{\text{par}} \text{LIP} \). Let \((S, w)\) be an input of M-CVP. We assign linear inequalities to any node \( c \) of the monotone circuit \( S \).

**Case 1:** \( c \) is a source storing the \( i \)th input bit. Use the inequalities

\[
\begin{align*}
x_c & \geq 0, \quad -x_c \geq 0 & \text{if } w_i = 0, \\
x_c & \geq 1, \quad -x_c \geq -1 & \text{if } w_i = 1.
\end{align*}
\]

**Case 2:** \( c \equiv a \land b \) is an AND-gate. This time we use the inequalities

\[
\begin{align*}
x_a - x_c & \geq 0, \quad x_b - x_c \geq 0, \quad x_c - x_a - x_b \geq -1, \quad x_c \geq 0.
\end{align*}
\]

**Case 3:** \( c \equiv a \lor b \) is an OR-gate. Now we choose

\[
\begin{align*}
x_c - x_a & \geq 0, \quad x_c - x_b \geq 0, \quad x_a + x_b - x_c \geq 0, \quad -x_c \geq -1.
\end{align*}
\]

We show inductively that \( x_c \) coincides with the value of gate \( c \), if \( w \) is the input for \( S \). We observe that the variables \( x_c \) for sources \( c \) are forced in Case 1 to take on the value of the corresponding input bit. If \( c \equiv a \land v \), then we already know that \( x_a \) and \( x_b \) coincide with the values of gates \( a \) and \( b \) respectively. Thus \( x_a, x_b \in \{0,1\} \) and an inspection of Case 2 shows that \( x_c \) has to coincide with the value of gate \( c \). The argument for \( c \equiv a \lor v \) is similar.

Finally, if \( t \) is the output gate of the circuit, then we add the inequality \( x_t \geq 1 \) to the system. The claimed reduction of M-CVP to LIP follows, since the translation from \((S, w)\) to the system of inequalities is computable in poly-logarithmic time with a polynomial number of processors.

(b) The reduction \( \text{LIP} \leq_{\text{par}} \text{LPP} \) is immediate, if we choose the translation \((A, b) \mapsto (A, b, 0, 0)\), since the existence of a solution \( x \) with \( A \cdot x \geq b \) and \( 0^T \cdot x \geq 0 \) is equivalent to the existence of a solution of \( A \cdot x \geq b \). \( \square \)
12.3.3 Network Flow

We describe the Flow problem (FP). An input for FP consists of

- a directed graph \( G = (V, E) \),
- two distinguished nodes, the source \( s \in V \) and the sink \( t \in V \)
- and a capacity function \( c : E \to \mathbb{R} \).

We call a function \( f : E \to \mathbb{R} \) a flow, if \( 0 \leq f(e) \leq c(e) \) for all edges \( e \in E \) and if flow conservation

\[
\sum_{u \in V, (u,v) \in E} f(u,v) = \sum_{u \in V, (v,u) \in E} f(v,u)
\]

holds for any node \( v \in V \setminus \{s,t\} \): the amount of flow entering \( v \) has to coincide with the amount of flow leaving \( v \). Observe that flow conservation is not required for the source and the sink. In the optimization version of the flow problem a maximal flow \( f \) is to be determined: we say that a flow \( f \) is maximal, if \( f \) maximizes

\[
|f| = \sum_{u \in V, (s,u) \in E} f(s,u) - \sum_{u \in V, (u,s) \in E} f(u,s),
\]

i.e., if the net flow pumped out of \( s \) is maximal.

We say that input \( (G, s, t, c) \) belongs to FP iff all capacities are natural numbers and the maximal flow is odd. It turns out that the flow problem, even as an optimization problem can be solved with the help of linear programming and hence FP is certainly not harder than LIP or LPP. However FP is still \( \mathcal{P} \)-complete.

**Theorem 12.19** FP is \( \mathcal{P} \)-complete.

**Proof:** We mention without proof that FP belongs to \( \mathcal{P} \) and hence it suffices to show that FP is \( \mathcal{P} \)-hard. We claim that the reduction \( \text{M}_2\text{-CVP} \leq_{\text{par}} \text{FP} \) holds.

Let \( (S, w) \) be an input for \( \text{M}_2\text{-CVP} \) and let \( G = (V, E) \) be the directed graph of circuit \( S \). We may assume that the output gate of \( S \) is an OR-gate: if not, then replace \( S \) by two copies of \( S \) which are fed into an OR-gate. We know that the fan-out of \( G \) is at most two. Moreover we may assume that each input node of \( G \) has fan-out one: if an input node has two outgoing edges, then add a copy and distribute the two edges among the original and its copy.

To translate \( (S, w) \) into an equivalent input for FP, we define a new graph \( G^* \) by adding two new nodes \( s \) and \( t \) to \( G \). \( s \) will be the unique source of \( G^* \) and we add connections from \( s \) to all sources of \( G \). (The sources of \( G \) correspond to the inputs of \( S \).) \( t \) will be the unique sink of \( G^* \) and we add one connection from the old sink \( t_G \) of \( G \) to \( t \). (\( t_G \) computes
the output of the circuit.) Finally we add edges from OR-gates to $s$ and from AND-gates to $t$.

To define capacities, we first compute a topological sort $nr : V \rightarrow \{1, \ldots, |V|\}$ of $G$. Assume that $G$ has exactly $n$ nodes. We say that an edge of $G$ is an internal edge of $G^*$. The fanout of a node $z$ of $G$ only counts internal edges leaving $z$ except for the old sink $t_G$: we set $\text{fanout}(t_G) = 1$ to account for the edge $(t_g, s)$.

- If $z$ is an input node for variable $x_i$, then set
  \[
  c(s, z) = \begin{cases} 2^{n-nr(z)} & w_i = 1, \\ 0 & \text{otherwise}. \end{cases}
  \]

- If $z$ is not an input node, then $z$ receives edges from exactly two nodes $x$ and $y$. We define
  \[
  c(x, z) = 2^{n-nr(x)} \quad \text{and} \quad c(y, z) = 2^{n-nr(y)}.
  \]
  If $z$ is an AND-gate, then set
  \[
  c(z, t) = 2^{n-nr(x)} + 2^{n-nr(y)} - \text{fanout}(z) \cdot 2^{n-nr(z)},
  \]
  respectively if $z$ is an OR-gate, then
  \[
  c(z, s) = 2^{n-nr(x)} + 2^{n-nr(y)} - \text{fanout}(z) \cdot 2^{n-nr(z)}.
  \]

- If $z = t_G$, then $c(t_G, s) = 2^{n-nr(x)} + 2^{n-nr(y)} - 1$. We set $c(t_G, t) = 1$ and hence, if $t_G$ receives the maximal possible flow $2^{n-nr(x)} + 2^{n-nr(y)}$, the edge $(t_G, t)$ can transport the remaining flow of one to $t$. Observe that all other nodes have links to $t$ of even capacity.

\[
\begin{aligned}
&\text{fanout}(v) \cdot 2^{n-nr(v)} \\
&\quad \rightarrow \\
&2^{n-nr(u)} + 2^{n-nr(w)} \\
&\quad \rightarrow -\text{fanout}(v) \cdot 2^{n-nr(v)} \\
&\quad \rightarrow 2^{n-nr(u)} \\
&\quad \rightarrow 2^{n-nr(w)}
\end{aligned}
\]

AND-gates and capacities of incident edges.
We claim that the translation \((S, w) \mapsto (G^*, s, t, c)\) establishes a reduction from \(M_2\text{-CVP}\) to FP.

As a first step we determine a maximal flow \(f^*\). We set \(f^*(s, z) = c(s, z)\) and hence we pump a flow of \(2^{n-\text{nr}(z)}\) into \(z\) whenever the input bit of gate \(z\) is one. Observe that only one edge is leaving \(z\). This edge has capacity \(2^{n-\text{nr}(z)}\) and will be completely filled by \(f^*\). If however the input bit of \(z\) is zero, then we just move flow zero across the one edge leaving \(z\). We maintain the following policy for all other nodes \(z\) of \(G\):

1. If \(z\) evaluates to one, then fill its fan-out\((z)\) many internal edges to capacity and push the excess flow to \(s\) respectively to \(t\). If \(z = t_G\), then push flow one along edge \((t_G, t)\).
2. If \(z\) evaluates to zero, then push zero flow along its fan-out\((z)\) many internal edges and any excess flow to \(s\) respectively to \(t\).

**Exercise 129**
Show that \(f^*\) is a flow.

To show that \(f^*\) is a maximal flow, we define the node sets

\[
V_1 = \{s\} \cup \{z \in G \mid \text{the gate } z \text{ evaluates to one} \}
\]
\[
V_0 = \{t\} \cup \{z \in G \mid \text{the gate } z \text{ evaluates to zero} \}
\]

By property (1) of \(f^*\), all internal edges from a node in \(V_1\) to a node in \(V_0\) are filled to capacity. Edges from AND-gates in \(V_1\) to \(t\) are also filled to capacity, whereas edges from OR-gates in \(V_1\) stay in \(V_1\). The edge \((t_G, t)\) is filled to capacity as well, provided \(t_G\) evaluates to one. Hence all edges from a node in \(V_1\) to a node in \(V_0\) are filled to capacity.

By property (2) \(f^*\) pushes zero flow across all internal edges from a node in \(V_0\) to a node in \(V_1\). Edges from AND-gates in \(V_0\) to \(t\) stay within \(V_0\). An OR-gate in \(V_0\) receives zero flow along its two internal edges and hence an OR-gate in \(V_0\) has no flow to distribute. Since the edge \((t_G, t)\) stays inside \(V_0\) if \(t_G \in V_0\), all edges from a node in \(V_0\) to a node in \(V_1\) receive zero flow. But then

\[
|f^*| = \sum_{x \in V_1, y \in V_0, (x, y) \in E} c(x, y).
\]

**Exercise 130**
Show that \(|g| \leq \sum_{x \in V_1, y \in V_0, (x, y) \in E} c(x, y)\) holds for any flow \(g\).

Thus \(f^*\) is a maximal flow. We are done, if we can show

**Claim 12.1** \(|f^*|\) is odd iff \(S\) accepts \(w\).

Remember that the old sink \(t_G\) is the only node which may push a flow of one to \(t\), whereas all other nodes push either zero flow or even flow to \(t\). But \(t_G\) pushes flow one iff it evaluates to one, that is iff the circuit accepts the input \(w\).
Exercise 131
Why do we leak out excess flow of AND-gates to \( t \) and excess flow of OR-gates to \( s \)?

Exercise 132
Let \( G = (V_1 \cup V_2, E) \) be a bipartite graph with \( n = |V_1| = |V_2| \), i.e., all edges are incident with a node in \( V_1 \) and a node of \( V_2 \). We associate a matrix \( A = (A_{i,j}) \) of indeterminates with \( G \) by setting
\[
A_{i,j} := \begin{cases} 
X_{i,j} & (v_i, v_j) \in E, \\
0 & \text{otherwise.}
\end{cases}
\]
Show that \( G \) has a perfect matching iff the determinant of \( A \) is not the zero-polynomial. (A matching is a set of node-disjoint edges. A matching \( M \) is perfect, if all nodes are endpoint of some edge of \( M \).)

Exercise 133
Decide whether a given bipartite graph has a perfect matching. You may use a randomized MPI algorithm with poly-logarithmic run time and polynomially many processors.

Hint: the determinant can be computed in poly-logarithmic run time by an MPI algorithm with polynomially many processors. Moreover you may utilize that a polynomial \( p \not\equiv 0 \) (with \( n \) indeterminants) of degree \( d \) does not vanish on an input from \( \{-d, \ldots, d\}^n \) with probability at least \( \frac{1}{2} \).

Exercise 134
Show with the help of the constant depth reduction \( \leq_C \) that
- MATCHING: Does the bipartite graph \( G \) have a matching of size at least \( k \)?
- NETZWORK-FLOW: Does a network have a flow of at least \( k \)? (We require that all capacities are polynomially bounded (in \(|V|\)) natural numbers.)

possess also randomized \( \mathcal{NC} \) algorithms.

Hence by checking all flow-values we can determine in randomized \( \mathcal{NC} \), if the maximal flow is even or odd, provided capacities are polynomially bounded. Our negative result on network flow suggests that the polynomial bound is crucial.

12.3.4 Parallelizing Sequential Programs

When is it possible to parallelize sequential programs? We first consider the simple greedy heuristic for the independent set problem of Section 11.3 and ask whether we can find a fast parallel program with identical output.

Algorithm 12.20 A simple heuristic
\[
I := \emptyset;
\]
for \( v = 1 \) to \( n \) do
   If \( (v \text{ is not connected with a node from } I) \) then
      \[
      I = I \cup \{v\};
      \]
In the Lexicographically First Maximal Independent Set problem (LFMIS) we are given an undirected graph \( G = (\{1, \ldots, n\}, E) \) and a node \( v \). The pair \((G, v)\) belongs to LFMIS iff \( v \) belongs to the set \( I \) computed by Algorithm 12.20.

Before showing that LFMIS is \( \mathcal{P} \)-complete we determine longest paths in a directed acyclic graph \( G = (V, E) \).

**Lemma 12.21** The all-pairs-longest path problem for a directed acyclic graph \( G \) with \( n \) nodes can be solved in time \( O(\log^2 n) \) with \( n^3 \) processors.

**Proof:** We proceed as in Lemma 12.6 and reduce the all-pairs-longest-path problem to a problem in matrix multiplication. In particular, for \( n \times n \) matrices \( X \) and \( Y \) we introduce a non-standard matrix multiplication by setting

\[
X \ast Y[u, v] = \max_{w \in V} X[u, w] + Y[w, v].
\]

Let \( A \) be the adjacency matrix of \( G \) and assume that the diagonal of \( A \) is zero, i.e., \( G \) has no self-loops. Then

\[
A^k[u, v] = \max_{w_1, \ldots, w_{k-1}} A[u, w_1] + A[w_1, w_2] + \cdots + A[w_{k-1}, v]
\]

and hence \( A^k[u, v] \) is the length of a longest path \( u \rightarrow v \), among all paths \( u \rightarrow v \) of length at most \( k \). In particular \( A^{n-1}[u, v] \) is the length of a longest path from \( u \) to \( v \).

We may compute \( A^{n-1} \) by computing the matrix powers \( A^2, \ldots, A^{2t}, A^{2t+1}, \ldots, A^{2^t} \) for \( t = \lceil \log_2 n \rceil \) and hence time \( O(\log^2 n) \) with \( n^3 \) processors is sufficient. Observe that the matrix product allows to traverse nodes multiple times and hence we have to require \( G \) to be acyclic. \( \square \)

**Theorem 12.22** LFMIS is \( \mathcal{P} \)-complete.

**Proof:** It is obvious that LFMIS belongs to \( \mathcal{P} \). We show that LFMIS is \( \mathcal{P} \)-hard, by showing the reduction NOR-CVP \( \leq_{\text{par}} \) LFMIS.

The idea of the reduction is simple. Let \((S, w)\) be an input for NOR-CVP and let \( G \) be the directed graph for circuit \( S \). A NOR gate

\[
\begin{array}{c}
v \\
\text{NOR} \\
\bigcirc \\
\bigcirc \\
u \quad w
\end{array}
\]
“fires”, if both \( u \) and \( w \) evaluate to zero. Thus all nodes that evaluate to one form an independent set \( I \). However how to make sure that Algorithm 12.20 picks \( I \)? We modify \( G \) and change the names of nodes.

We first determine a topological sort of \( G \), that is a bijection \( \pi : V \rightarrow \{1, \ldots n\} \) such that \( \pi(u) < \pi(v) \) for any edge \((u, v)\). Let \( \text{length}(v) \) be the length of a longest path ending in \( v \). To determine a topological sort we replace each node \( v \) by the pair \((\text{length}(v), v)\) and sort all pairs. Since \( \text{length}(v) \) can be determined fast with Lemma 12.21, a topological sort can be computed fast.

Now replace the name of a node by its rank within the topological sort. Thus the smallest number is given to a source which may however evaluate to zero. Since the node with smallest number will belong to the independent set, we invent a new node 0 and connect 0 with all sources which evaluate to 0. We call the new (undirected) graph \( G^* \).

**Claim 12.2** Algorithm 12.20 determines for \( G^* \) the independent set

\[
I = 0 \cup \{z \mid z \text{ evaluates to one} \}.
\]

Observe that we are done, if claim 12.2 holds: we define the translation \((S, w) \mapsto (G^*, n)\) and \( S \) accepts \( w \) iff the sink \( n \) of \( G \) evaluates to one.

**Proof of Claim 12.2:** Certainly 0 is chosen, since it has the smallest number. The remaining nodes of \( G \) are processed by Algorithm 12.20 according to their topological number and we therefore argue inductively.

**Case 1:** \( z \) is an input node of \( G \). Then \( z \) is either a source, which is equivalent to \( z \) evaluating to one, or \( z \) receives an edge from 0. In the first case \( z \) will be inserted into \( I \) and in the second case 0 prevents an insertion of \( z \).

**Case 2:** \( z \) is not a source and \( z \) fires. We already observed that a NOR gate fires, if its two immediate predecessors evaluate to zero. By induction hypothesis the two predecessors have not been chosen and thus \( z \) is included into \( I \).

**Case 2:** \( z \) is not a source and \( z \) does not fire. Then at least one immediate predecessor of \( z \) fires and hence has been included into \( I \). Thus \( z \) is not chosen. \( \square \)

The depth-first search algorithm is another famous sequential algorithm which is notoriously hard to parallelize. In the Ordered Depth-First Search problem (ODFS), the input consists of a directed graph \( G = (V, E) \) and the three nodes \( s, u, v \in V \). The quadruple \((G, s, u, v)\) belongs to ODFS iff node \( u \) is visited before node \( v \) in the depth-first search started in node \( s \). (Observe that the depth-first search traversal is uniquely defined, once we specify the adjacency list representation of the graph.)

**Theorem 12.23** ODFS is \( \mathcal{P} \)-complete.

**Proof:** See [Ja92]. \( \square \)
12.4 Conclusion

We have introduced the space complexity of sequential computations with the help of off-line Turing machines and have observed that all languages recognizable by a nondeterministic off-line Turing machine within logarithmic space are parallelizable. Moreover it turns out that circuit depth and space are polynomially related, provided we consider only uniform circuits, i.e., circuit families whose circuits are easy to construct. This observation is evidence for the parallel computation thesis which calls a parallel model of computation unreasonable, if its parallel time is not polynomially related to space complexity. There is further evidence for the parallel computation thesis, since all known “reasonable” models of parallel computation such as MPI algorithms, parallel register machines or circuits satisfy the polynomial relationship.

The “parallel reducibility” $\leq_{\text{par}}$ allows us to compare two languages with respect to their complexity for parallel algorithms. We have defined $\mathcal{P}$-complete languages as those languages in $\mathcal{P}$ which are at least as hard to parallelize as any other problem in $\mathcal{P}$. We have seen that the Circuit Value problem is a generic $\mathcal{P}$-complete problem which for instance allows us to show that Linear Programming and Network Flow are $\mathcal{P}$-complete. We also considered the problem of parallelizing sequential programs and have shown that the Lexicographically First Maximal Independent Set problem and the Ordered Depth-First Search problem are in all likelihood not parallelizable. Further $\mathcal{P}$-complete problems can be found in [GHR95].
Bibliography


