Advanced Algorithms And Parallel Programming

Theory

Christian Rossi

Academic Year 2024-2025

Abstract

This course begins with an exploration of randomized algorithms, specifically Las Vegas and Monte Carlo algorithms, and the methods used to analyze them. We will tackle the hiring problem and the generation of random permutations to build a strong foundation. The course will then cover randomized quicksort, examining both worst-case and average-case analyses to provide a comprehensive understanding. Karger-s Min-Cut Algorithm will be studied, along with its faster version developed by Karger and Stein. We will delve into randomized data structures, focusing on skip lists and treaps, to understand their construction and application. Dynamic programming will be a key area, where we will learn about memoization and examine examples such as string matching and Binary Decision Diagrams (BDDs). The course will also introduce amortized analysis, covering dynamic tables, the aggregate method, the accounting method, and the potential method to equip students with robust analytical tools. Additionally, we will touch on approximate programming, providing an overview of this important concept. Finally, the competitive analysis will be explored through self-organizing lists and the move-to-front heuristic.

The second part of the course shifts to the design of parallel algorithms and parallel programming. We will study various parallel patterns, including Map, Reduce, Scan, MapReduce, and Kernel Fusion, to understand their implementation and application. Tools and languages essential for parallel programming, such as Posix Threads, OpenMP, and Message Passing Interface, will be covered, alongside a comparison of these parallel programming technologies. The course will also focus on optimizing and analyzing parallel performance, providing students with the skills needed to enhance and evaluate parallel computing systems. Practical examples of parallel algorithms will be reviewed to solidify understanding and demonstrate real-world applications.

Contents

CHAPTER 1

Algorithms complexity

1.1 Introduction

Definition (*Algorithm*)**.** An algorithm is a clearly defined computational procedure that accepts one or more input values and produces one or more output values.

An algorithm can be seen as a tool for solving a clearly defined computational problem. The problem statement outlines the desired relationship between input and output in broad terms, while the algorithm provides a detailed procedure to achieve that relationship.

It is essential that an algorithm terminates after a finite number of steps.

1.2 Complexity analysis

The running time of an algorithm varies with the input. Therefore, we often parameterize running time by the input size.

Running time analysis can be categorized into three main types:

- *Worst-case* (most common): here, $T(n)$ represents the maximum time an algorithm takes on any input of size n. This is particularly relevant when time is a critical factor.
- *Average-case* (occasionally used): in this case $T(n)$ reflects the expected time of the algorithm across all inputs of size n. It requires assumptions about the statistical distribution of inputs.
- *Best-case* (often misleading): this scenario highlights a slow algorithm that performs well on specific inputs.

To establish a general measure of complexity, we focus on a machine-independent evaluation. This framework is called asymptotic analysis.

As the input length n increases, algorithms with lower complexity will outperform those with higher complexities. However, asymptotically slower algorithms should not be dismissed, as real-world design often requires a careful balance of various engineering objectives.

In mathematical terms, we define the complexitybound as:

$$\Theta\left(g(n)\right) = f(n)$$

Here, $f(n)$ satisfies the existence of positive constants c_1, c_2, and n_0 such that:

$$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \qquad \forall n \geq n_0$$

In engineering practice, we typically ignore lower-order terms and constants.

Example:
Consider the following expression:

$$3n^3 + 90n^2 - 5n + 6046$$

The corresponding theta notation is:

$$\Theta(n^3)$$

Given $c > 0$ and $n_0 > 0$, we can define other bounds notations:

Bound type	Notation	Condition
Upper bound	$\mathcal{O}(g(n)) = f(n)$	$0 \leq f(n) \leq cg(n) \quad \forall n \geq n_0$
Lower bound	$\Omega(g(n)) = f(n)$	$0 \leq cg(n) \leq f(n) \quad \forall n \geq n_0$
Strict upper bound	$o(g(n)) = f(n)$	$0 \leq f(n) < cg(n) \quad \forall n \geq n_0$
Strict lower bound	$\omega(g(n)) = f(n)$	$0 \leq cg(n) < f(n) \quad \forall n \geq n_0$

Example:
For the expression $2n^2$:

$$2n^2 \in O(n^3)$$

For the expression $\sqrt{n}$:

$$\sqrt{n} \in \Omega(\ln(n))$$

From this, we can redefine the theta notation as:

$$\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$$

1.2.1 Sorting problem

The sorting problem involves taking an array of numbers $\langle a_1, a_2, \ldots, a_n \rangle$ and returning the permutation of the input $\langle a'_1, a'_2, \ldots, a'_n \rangle$ such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.

Example:
Given an array:

$$\langle 8, 2, 4, 9, 3, 6 \rangle$$

The sorted version will be:

$$\langle 2, 3, 4, 6, 8, 9 \rangle$$

Algorithm 1 Insertion sort

```
for j = 2 to n do
    key = A[j]
    i = j − 1
    while i > 0 and A[i] > key do
        A[i + 1] = A[i]
        i = i − 1
    end while
    A[i + 1] = key
end for
```

The complexities for the insertion sort are:

Case	Complexity	Notes
Worst	$T(n) = \Theta\left(\sum_{j=2}^{n} j\right) = \Theta(n^2)$	Input in reverse order
Average	$T(n) = \Theta\left(\sum_{j=2}^{n} \frac{j}{2}\right) = \Theta(n^2)$	All permutations equally likely
Best	$T(n) = \Theta\left(\sum_{j=2}^{n} 1\right) = \Theta(n)$	Already sorted

In conclusion, while this algorithm performs well for small n, it becomes inefficient for larger input sizes.

A recursive solution for the sorting problem could be implemented with the merge sort.

Algorithm 2 Merge sort

```
if n = 1 then
    return A[n]
end if
Recursively sort the two half lists A[1 ... ⌈n/2⌉] and A[⌈n/2⌉ + 1 ... n]
Merge (A[1 ... ⌈n/2⌉], A[⌈n/2⌉ + 1 ... n])
```

The merge operation makes this algorithm recursive. To analyze its complexity, we consider the following components:

- When the array has only one element, the complexity is constant: $\Theta(1)$.
- The recursive sorting of the two halves contributes a total cost of $2T\left(\frac{n}{2}\right)$.
- The merging of the two sorted lists requires linear time to check all elements, yielding a complexity of $\Theta(n)$.

Thus, the overall complexity for merge sort can be expressed as:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T\left(\frac{n}{2}\right) + \Theta(n) & \text{if } n > 1 \end{cases}$$

For sufficiently small n, the base case $\Theta(1)$ can be omitted if it does not affect the asymptotic solution. The solution for the recurrence equation is:

$$T(n) = \Theta(n \log_2 n)$$

1.3 Recurrences

To determine the complexity a recurrent algorithm, we need to solve the equation:

$$T(n) = 2T\left(\frac{n}{2}\right) + c \cdot n$$

To solve this recurrence we may use three different tecniques:

1. Recursion tree.
2. Substitution method.
3. Masther method.

1.3.1 Recursion tree

In the recursion tree we expand nodes until we reach the base case.

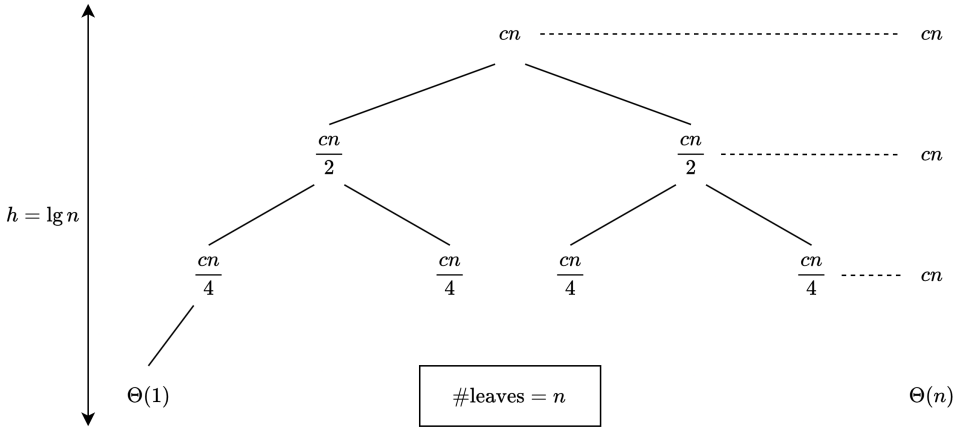

Figure 1.1: Partial recursion tree for merge sort algorithm

The depth of the tree is $h = \log_2 n$, and the total number of leaves is n. Thus, the complexity can be computed as:

$$T(n) = \Theta(n \log_2 n)$$

The merge sort outperforms insertion sort in the worst case, but in practice merge sort generally surpasses insertion sort for $n > 30$.

1.3.2 Substitution method

The substitution method is a general technique for solving recursive complexity equations. The steps are as follows:

1. Guess the form of the solution based on preliminary analysis of the algorithm.
2. Verify the guess by induction.
3. Solve for any constants involved.

Example:
Consider the expression:

$$T(n) = 4T\left(\frac{n}{2}\right) + n$$

Assuming the base case $T(1) = \Theta(1)$, we can apply the substitution method:

1. Guess a solution of $\mathcal{O}(n^3)$, so we assume $T(k) \leq ck^3$ for $k < n$.
2. Verify by induction that $T(n) \leq cn^3$.

This approach, while effective, may not always be straightforward.

1.3.3 Master method

To simplify the analysis, we can use the master method, applicable to recurrences of the form:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

Here, $a \geq 1$, $b > 1$, and $f(n)$ is asymptotically positive. While less general than the substitution method, it is more straightforward.

To apply the master method, compare $f(n)$ with $n^{\log_b a}$. There are three possible outcomes:

1. If $f(n) = O(n^{\log_b a - \varepsilon})$ for some constant $\varepsilon > 0$, then

$$T(n) = \Theta(n^{\log_b a})$$

2. If $f(n) = \Theta(n^{\log_b a} \log^k n)$ for some constant $k \geq 0$, then:

$$T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$$

3. If $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some constant $\varepsilon > 0$ and $f(n)$ satisfies the regularity condition $a \cdot f\left(\frac{n}{b}\right) \leq c \cdot f(n)$ for some constant $c < 1$, then:

$$T(n) = \Theta(f(n))$$

Example:
Let's analyze the expression:

$$T(n) = 4T\left(\frac{n}{2}\right) + n$$

In this case, we have $a = 4$ and $b = 2$, which gives us:

$$n^{\log_b a} = n^2 \qquad f(n) = n$$

Here, we find ourselves in the first case of the master theorem, where $f(n) = \mathcal{O}(n^{2-\varepsilon})$ for $\varepsilon = 1$. Thus, the solution is:

$$T(n) = \Theta(n^2)$$

Now consider the expression:

$$T(n) = 4T\left(\frac{n}{2}\right) + n^2$$

Again, we have $a = 4$ and $b = 2$, leading to:

$$n^{\log_b a} = n^2 \qquad f(n) = n^2$$

In this scenario, we are in the second case of the theorem, where $f(n) = \Theta(n^2 \log^k n)$ for $k = 0$. Therefore, the solution is:

$$T(n) = \Theta(n^2 \log n)$$

Next, consider:

$$T(n) = 4T\left(\frac{n}{2}\right) + n^3$$

With $a = 4$ and $b = 2$, we find:

$$n^{\log_b a} = n^2 \qquad f(n) = n^3$$

Here, we fall into the third case of the theorem, where $f(n) = \Omega(n^{2+\varepsilon})$ for $\varepsilon = 1$. Thus, the solution is:

$$T(n) = \Theta(n^3)$$

Finally, consider the expression:

$$T(n) = 4T\left(\frac{n}{2}\right) + \frac{n^2}{\log n}$$

Again, we have $a = 4$ and $b = 2$ yielding:

$$n^{\log_b a} = n^2 \qquad f(n) = \frac{n^2}{\log n}$$

In this case, the master method does not apply. Specifically, for any constant $\varepsilon > 0$, we have $n^\varepsilon = \omega(\log n)$, indicating that the conditions for the theorem are not satisfied.

CHAPTER 2

Divide and conquer algorithms

2.1 Introduction

The divide and conquer design paradigm consists of three key steps:

1. Divide the problem into smaller sub-problems.
2. Conquer the sub-problems by solving them recursively.
3. Combine the solutions of the sub-problems.

This approach enables us to tackle larger problems by breaking them down into smaller, more manageable pieces, often resulting in faster overall solutions.

The divide step is typically constant, as it involves splitting an array into two equal parts. The time required for the conquer step depends on the specific algorithm being analyzed. Similarly, the combine step can either be constant or require additional time, again depending on the algorithm.

Merge sort The merge sort algorithm, previously discussed, follows these steps:

- *Divide*: the array is split into two sub-arrays.
- *Conquer*: each of the two sub-arrays is sorted recursively.
- *Combine*: the two sorted sub-arrays are merged in linear time.

The recursive expression for the complexity of merge sort can be expressed as follows:

$$T(n) = \underbrace{2}_{\text{\#subproblems}} \underbrace{T\left(\frac{n}{2}\right)}_{\text{subproblem size}} + \underbrace{\Theta(n)}_{\text{work dividing and combining}}$$

2.2 Binary search

The binary search problem involves locating an element within a sorted array. This can be efficiently solved using the divide and conquer approach, outlined as follows:

1. *Divide*: check the middle element of the array.
2. *Conquer*: recursively search within one of the sub-arrays.
3. *Combine*: if the element is found, return its index in the array.

In this scenario, we only have one sub-problem, which is the new sub-array, and its length is half that of the original array. Both the divide and combine steps have a constant complexity.

Thus, the final expression for the complexity is:

$$T(n) = 1T\left(\frac{n}{2}\right) + \Theta(1)$$

By applying the master method, we find a final complexity of:

$$T(n) = \Theta(\log n)$$

2.3 Power of a number

The problem at hand is to compute the value of a^n, where $n \in \mathbb{N}$. The naive approach involves multiplying a by itself n times, resulting in a total complexity of $\Theta(n)$.

We can also use a divide and conquer algorithm to solve this problem by dividing the exponent by two, as follows:

$$a^n = \begin{cases} a^{\frac{n}{2}} \cdot a^{\frac{n}{2}} & \text{if } n \text{ is even} \\ a^{\frac{n-1}{2}} \cdot a^{\frac{n-1}{2}} \cdot a & \text{if } n \text{ is odd} \end{cases}$$

In this approach, both the divide and combine phases have a constant complexity, as they involve a single division and a single multiplication, respectively. Each iteration reduces the problem size by half, and we solve one sub-problem (with two equal parts).

Thus, the recurrence relation for the complexity is:

$$T(n) = 1T\left(\frac{n}{2}\right) + \Theta(1)$$

By applying the master method, we find a final complexity of:

$$\Theta(\log_2 n)$$

2.4 Matrix multiplication

Matrix multiplication involves taking two matrices A and B as input and producing a resulting matrix C, which is their product. Each element of the matrix C is computed as follows:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$$

The standard algorithm for matrix multiplication is outlined below:

Algorithm 3 Standard matrix multiplication

```
1: for i = 1 to n do
2:     for j = 1 to n do
3:         c_ij = 0
4:         for k = 1 to n do
5:             c_ij = c_ij + a_ik b_kj
6:         end for
7:     end for
8: end for
```

The complexity of this algorithm, due to the three nested loops, is $\Theta(n^3)$.

Divide and conquer For the divide and conquer approach, we divide the original $n \times n$ matrix into four $\frac{n}{2} \times \frac{n}{2}$ sumatrices:

$$\begin{bmatrix} r & s \\ t & u \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \cdot \begin{bmatrix} e & f \\ g & h \end{bmatrix}$$

This requires solving the following system:

$$\begin{cases} r = ae + bg \\ s = af + bh \\ t = ce + dg \\ u = cf + dh \end{cases}$$

This results in a total of eight multiplications and four additions of the submatrices. The recursive part of the algorithm involves the matrix multiplications. The time complexity can be expressed as $T(n) = 8T\left(\frac{n}{2}\right) + \Theta(n^2)$. Using the master method, we find that the total complexity remains

$$\Theta(n^3)$$

Strassen To improve efficiency, Strassen proposed a method that reduces the number of multiplications from eight to seven matrices. This is achieved using the following factors:

$$\begin{cases} P_1 = a \cdot (f - h) \\ P_2 = (a + b) \cdot h \\ P_3 = (c + d) \cdot e \\ P_4 = d \cdot (g - e) \\ P_5 = (a + d) \cdot (e + h) \\ P_6 = (b - d) \cdot (g + h) \\ P_7 = (a - c) \cdot (e + f) \end{cases}$$

Using these products, we can compute the elements of the resulting matrix:

$$\begin{cases} r = P_5 + P_4 - P_2 + P_6 \\ s = P_1 + P_2 \\ t = P_3 + P_4 \\ u = P_5 + P_1 - P_3 - P_7 \end{cases}$$

This approach requires seven multiplications and a total of eighteen additions and subtractions.

The divide and conquer steps are as follows:

1. *Divide*: partition matrices A and B into $\frac{n}{2} \times \frac{n}{2}$ submatrices and formulate terms for multiplication using addition and subtraction.
2. *Conquer*: recursively perform seven multiplications of $\frac{n}{2} \times \frac{n}{2}$ submatrices.
3. *Combine*: construct matrix C using additions and subtractions on the $\frac{n}{2} \times \frac{n}{2}$ sumbatrices.

The recurrence relation for the complexity is: $T(n) = 7T\left(\frac{n}{2}\right) + \Theta(n^2)$ By solving this recurrence with the master method, we obtain a complexity of:

$$\Theta\left(n^{\log_2 7}\right) \approx \Theta\left(n^{2.81}\right)$$

Although 2.81 may not seem significantly smaller than 3, the impact of this reduction in the exponent is substantial in terms of running time. In practice, Strassen's algorithm outperforms the standard algorithm for $n \geq 32$.

The best theoretical complexity achieved so far is $\Theta\left(n^{2.37}\right)$, although this remains of theoretical interest, as no practical algorithm currently achieves this efficiency.

2.5 VLSI layout

The problem involves embedding a complete binary tree with n leaves into a grid while minimizing the area used.

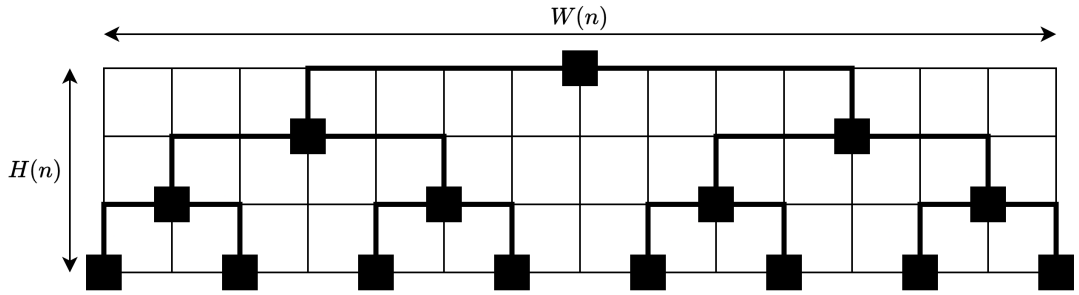

Figure 2.1: VLSI layout problem

For a complete binary tree, the height is given by:

$$H(n) = H\left(\frac{n}{2}\right) + \Theta(1) = \Theta(\log_2 n)$$

The width is expressed as:

$$W(n) = 2W\left(\frac{n}{2}\right) + \Theta(1) = \Theta(n)$$

Thus, the total area of the grid required is:

$$\text{Area} = H(n) \cdot W(n) = \Theta(n \log_2 n)$$

H-tree An alternative solution to this problem is to use an h-tree instead of a binary tree.

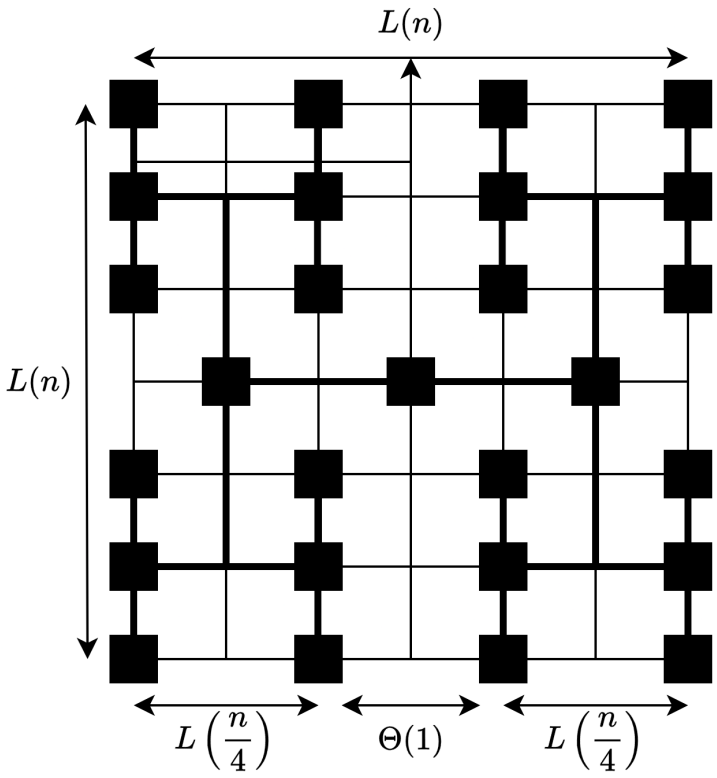

Figure 2.2: VLSI layout problem

For the h-tree, the length is given by:

$$L(n) = 2L\left(\frac{n}{4}\right) + \Theta(1) = \Theta(\sqrt{n})$$

Consequently, the total area required for the h-tree is computed as:

$$\text{Area} = L(n)^2 = \Theta(n)$$

CHAPTER 3

Parallel machine model

3.1 Random Access Machine

Definition (*Random Access Machine*)**.** A Random Access Machine (RAM) is a theoretical computational model that features the following characteristics:

- *Unbounded memory cells*: the machine has an unlimited number of local memory cells.
- *Unbounded integer capacity*: each memory cell can store an integer of arbitrary size, without any constraints.
- *Simple instruction set*: the instruction set includes basic operations such as arithmetic, data manipulation, comparisons, and conditional branching.
- *Unit-time operations*: every operation is assumed to take a constant, unit time to complete.

The time complexity of a RAM is determined by the number of instructions executed during computation, while the space complexity is measured by the number of memory cells utilized.

3.2 Parallel Random Access Machine

A Parallel Random Access Machine (PRAM) is an abstract machine designed to model algorithms for parallel computing.

Definition (*Parallel Random Access Machine*)**.** A Parallel Random Access Machine (PRAM) is defined as a system $M' = \langle M, X, Y, A \rangle$, where:

- M represent an infinite collection of identical RAMs.
- X represent the system's input.
- Y represent the system's output.
- A are shared memory cells between processors.

The set of RAMs M contains an unbounded collection of processors P, that have unbounded registers for internal storage. The set of shared memory cells A is unbounded and can be accessed in constant time. This set is used by the processors P to communicate with each other.

3.2.1 Computation

The computation in a PRAM consists of five phases, carried out in parallel by all processors. Each processor performs the following actions:

1. Reads a value from one of the input cells X_i.
2. Reads from one of the shared memory cells A_i.
3. Performs some internal computation.
4. May write to one of the output cells Y_i.
5. May write to one of the shared memory cells A_i.

Some processors may remain idle during computation.

Conflicts Conflicts can arise in the following scenarios:

- *Read conflicts*: two or more processors may simultaneously attempt to read from the same memory cell.
- *Write conflicts*: two or more processors attempt to write simultaneously to the same memory cell.

PRAM models are classified based on their ability to handle read/write conflicts, offering both practical and realistic classifications:

PRAM model	**Operation**
Exclusive Read	Read from distinct memory locations
Exclusive Write	Write to distinct memory locations
Concurrent Read	Read from the same memory locations
Concurrent Write	Write to the same memory locations

When a write conflict occurs, the final value written depends on the conflict resolution strategy:

- *Priority CW*: processors are assigned priorities, and the value from the processor with the highest priority is written.
- *Common CW*: all processors are allowed to complete their write only if all values to be written are equal.
- *Arbitrary CW*: a randomly chosen processor is allowed to complete its write operation.

3.2.2 Conclusion

The PRAM model is both attractive and important for parallel algorithm designers for several reasons:

- *Natural*: the number of operations executed per cycle on P processors is at most P.
- *Strong*: any processor can access and read/write any shared memory cell in constant time.
- *Simple*: it abstracts away communication or synchronization overhead.
- *Benchmark*: if a problem does not have an efficient solution on a PRAM, it is unlikely to have an efficient solution on any other parallel machine.

Some possible variants of the PRAM machine model are:

- *Bounded number of shared memory cells*: when the input data set exceeds the capacity of the shared memory, values can be distributed evenly among the processors.
- *Bounded number of processors*: if the number of execution threads is higher than the number of processors, processors may interleave several threads to handle the workload.
- *Bounded size of a machine word*: limits the size of data elements that can be processed in a single operation.
- *Handling access conflicts*: constraints on simultaneous access to shared memory cells must be considered.

3.3 Performance

The main values used to evaluate the performance are:

$T_1(n)$	Time to solve a problem on one processor
$T_p(n)$	Time to solve a problem on p processors
$T_\infty(n)$	Time to solve a problem on ∞ processors
$\mathrm{SU}_p = \dfrac{T_1(n)}{T_p(n)}$	Speedup on p processors
$E_p = \dfrac{T_1}{pT_p(n)}$	Efficiency
$C(n) = P(n)T(n)$	Cost
$W(n)$	Work (total number of operations)

3.3.1 Matrix-vector multiplication

Matrix-vector multiplication involves multiplying a matrix by a vector.

To perform the multiplication, each element of the resulting vector is computed by taking the dot product of the rows of the matrix with the vector. Specifically, if you have a matrix $\mathbf{A}$ of size $n \times n$ and a vector $\mathbf{v}$ of size n, the resulting vector $\mathbf{u}$ will have size $n \times n$:

$$\mathbf{u} = \mathbf{A}\mathbf{v}$$

The entry u_i of the resulting vector is calculated as:

$$u_i = \sum_{j=1}^{n} a_{ij} v_j$$

Here, a_{ij} are the elements of the matrix $\mathbf{A}$. The algorithm that computes the vector $\mathbf{u}$ is:

Algorithm 4 Matrix-vector multiplication

1: Global read $x \leftarrow \mathbf{v}$ ▷ Broadcast vector $\mathbf{v}$ to all processors
2: Global read $y \leftarrow \mathbf{a}_i$ ▷ Read corresponding rows of matrix $\mathbf{A}$
3: Compute $w = xy$ ▷ Multiply matrix row with vector $\mathbf{v}$
4: Global write $w \rightarrow u_i$ ▷ Write result to the output vector $\mathbf{u}$

The performance measures of this algorithm in the best-case scenario are shown in the following table:

Measure	T_1	T_p	E_p	C	W
Complexity	$\mathcal{O}(n^2)$	$\mathcal{O}\left(\frac{n^2}{p}\right)$	1	$\mathcal{O}(n^2)$	p

3.3.2 Single program multiple data sum

In single program multiple data (SPMD), each processor operates independently on its subset of the data, typically using the same code but possibly with different input data. This model is commonly used in high-performance computing, scientific simulations, and data analysis tasks, enabling significant performance improvements by leveraging parallelism.

In the context of SPMD, a sum refers to the process of aggregating data from multiple processors or cores that are executing the same program on different segments of data. Here's how it typically works:

1. *Data distribution*: the data is divided into chunks, with each CPU assigned a specific subset to work on.

2. *Local computation*: each processor executes the same summation program on its assigned data.

3. *Local results*: after computing their local sums, each processor has a partial sum.

4. *Reduction*: the partial sums are then combined (reduced) to get the final sum.

5. *Final output*: the final result is the total sum of all the partial sums computed by the individual processors.

Algorithm 5 SPMD sum

1: Global read $x \leftarrow \mathbf{b}$	▷ Broadcast array **b** to all processors
2: Global write $y \rightarrow \mathbf{c}$	▷ Broadcast array **c** to all processors
3: Compute $z = x + y$	▷ Sum all vectors elements
4: Global write $z \rightarrow \mathbf{a}$	▷ Write result to the output array **a**

The performance measures of this algorithm are shown in the following table:

Measure	T_1	T_p	SU_p	E_p	C	W
Complexity $p = n$	$\mathcal{O}(n)$	$\mathcal{O}\left(2 + \log n\right)$	$\mathcal{O}\left(\frac{n}{2 + \log n}\right)$	$\frac{1}{\log n}$	$\mathcal{O}(n \log n)$	-
Complexity $p = n$	$\mathcal{O}(n)$	$\mathcal{O}\left(\frac{n}{p} + \log p\right)$	$\mathcal{O}(p)$	1	$\mathcal{O}(n)$	$\mathcal{O}(n)$

3.3.3 Matrix-matrix multiplication

Matrix-matrix multiplication involves multiplying a matrix by another matrix.

To perform the multiplication, each element of the resulting matrix is computed by taking the dot product of the rows of the first matrix with the columns of the second matrix. Specifically, if you have a matrix **A** of size $m \times n$ and a matrix **B** of size $n \times p$, the resulting matrix **C** will have size $m \times p$:

$$\mathbf{C} = \mathbf{AB}$$

The entry c_{ij} of the resulting matrix is calculated as:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

Here, a_{ik} are the elements of matrix **A** and b_{kj} are the elements of matrix **B**.

Algorithm 6 Matrix-matrix multiplication

1: Global read $x \leftarrow \mathbf{a}_i$	▷ Read corresponding rows of matrix **A**
2: Global read $y \leftarrow \mathbf{b}_i$	▷ Read corresponding columns of matrix **B**
3: Compute $w = xy$	▷ Multiply matrix **A** row with matrix **B** column
4: Global write $w \rightarrow \mathbf{u}_i$	▷ Write result to corresponding row of output matrix **u**

The performance measures of this algorithm are shown in the following table:

Measure	T_1	T_p	SU_p	E_p	C
Complexity	$\mathcal{O}(n^3)$	$\mathcal{O}\left(\log n\right)$	$\mathcal{O}\left(\frac{n^3}{\log n}\right)$	$\frac{1}{\log n}$	$\mathcal{O}\left(n^3 \log n\right)$

3.4 Model analysis

Definition (*Computationally Stronger*)**.** A model A is said to be computationally stronger than model B ($A \geq B$) if any algorithm written for B can run unchanged on A with the same parallel time and basic properties.

Lemma 3.4.1. *Assume $M' < M$. Any problem that can be solved for a P-processor and M-cell PRAM in T steps can be solved on a $(\max(P, M'))$-processor M'-cell PRAM in $\mathcal{O}\left(\frac{TM}{M'}\right)$ steps.*

Proof. We can partition the M simulated shared memory cells into M' continuous segments S_i of size $\frac{M}{M'}$ each. Each simulating processor P'_i $(1 \leq i \leq P)$ will simulate processor P_i of the original PRAM. Each simulating processor P'_i $(1 \leq i \leq M')$ stores the initial contents of segment S_i into its local memory and will use $M'[i]$ as an auxiliary memory cell for simulating accesses to cells of S_i.

Each P'_i $(i = 1, \ldots, \max(P, M'))$ repeats the following for $k = 1, \ldots, \frac{M}{M'}$. Write the value of the k-th cell of segment S_i into $M'[i]$ for $i = 1, \ldots, M'$. Read the value that the simulated processor P_i $(i = 1, \ldots, P)$ would read in this simulated substep, if it appeared in the shared memory. The local computation substep of P_i $(i = 1, \ldots, P)$ is simulated in one step by P'_i. The simulation of one original write operation is analogous to that of the read operation. □

The direct implementation of a PRAM on real hardware poses certain challenges due to its theoretical nature. Despite this, PRAM algorithms can be adapted for practical systems, allowing the abstract model to influence real-world designs. In some cases, PRAM can be implemented directly by translating its concepts to hardware. PRAM's CRCW model can be implemented using detect-and-merge techniques, where write conflicts are resolved by merging results. Priority CRCW, on the other hand, resolves conflicts by detecting and prioritizing certain writes over others.

PRAM is an attractive model for parallel computing due to several key factors. One major advantage is the large body of algorithms developed specifically for it, offering a rich resource for problem-solving. Its simplicity makes it easy to conceptualize, as PRAM abstracts away the complexities of synchronization and communication, allowing a pure focus on algorithm design. The synchronized shared memory model in PRAM eliminates many of the challenges related to synchronization and communication that arise in practical implementations. However, PRAM retains the flexibility to incorporate these issues when necessary, enabling exploration of more complex scenarios. A notable strength of PRAM is its adaptability. Algorithms initially designed for this model can often be converted into asynchronous versions, which are better suited to real-world architectures.

3.5 Amdahl law

In parallel computing, we consider two types of program segments: serial segments and parallelizable segments. The total execution time depends on the proportion of each.

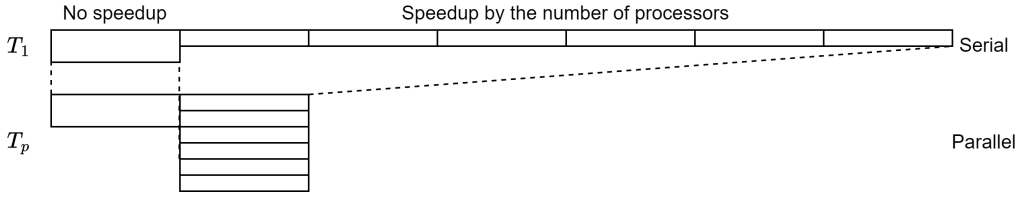

Figure 3.1: Serial and parallel models

When using more than one processor, the speedup (SU_P) is always less than the number of processors (P). The relationship can be expressed as:

$$T_P > \frac{T_1}{P} \implies \mathrm{SU}_P < P$$

Here, T_P is the time taken with P processors, and T_1 is the time for a serial execution.

In a program, the parallelizable portion is often represented by a fixed fraction f.

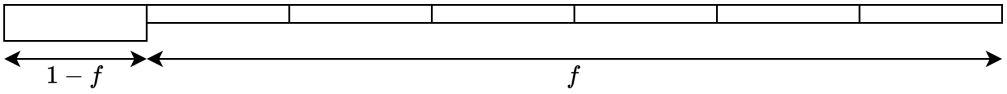

Figure 3.2: Serial model

Using the serial version of the model, the speedup function $\mathrm{SU}_P(f)$ is derived as follows:

$$\mathrm{SU}_P(f) = \frac{T_1}{T_P} = \frac{T_1}{T_1(1-f) + \frac{T_1 f}{P}} = \frac{1}{1 - f + \frac{f}{P}}$$

As the number of processors P approaches infinity, the speedup is limited by the serial portion:

$$\lim_{P \to \infty} \mathrm{SU}_P(f) = \frac{1}{1-f}$$

This shows that even with an infinite number of processors, the maximum speedup is constrained by the serial fraction of the program.

3.5.1 Gustafson law

In contrast to Amdahl's Law, John Gustafson proposed a different view in 1988, challenging the assumption that the parallelizable portion of a program remains fixed. Key differences include:

- The parallelizable portion of the program is not a fixed fraction.
- Absolute serial time is fixed, while the problem size grows to exploit more processors.
- Amdahl's law is based on a fixed-size model, while Gustafson's law operates on a fixed-time model, where the problem grows with increased processing power.

The speedup in Gustafson's model is expressed as:

$$\mathrm{SU}(P) = \frac{T_1}{T_P} = \frac{s + P(1-s)}{s + 1 - s} = s + P(1-s)$$

Here, s is the fixed serial portion of the program. As a result, this model suggests linear speedup is possible as the number of processors increases, especially for highly parallelizable tasks. Gustafson's law is empirically applicable to large-scale parallel algorithms, where increasing computational power enables solving larger and more complex problems within the same time frame.

3.5.2 Conclusion

Amdahl's Law assumes that the overall computing workload remains constant, and thus adding more processors merely reduces execution time for the same task. This makes Amdahl's model practical for scenarios where the problem size is fixed, but speedup is limited by the serial portion of the task.

On the other hand, Gustafson's Law takes a more expansive view, arguing that as computing power increases, the problem size grows accordingly. More processors allow for deeper, more detailed analysis, which wouldn't have been feasible with limited computational resources. Increasing the power of computation enables more comprehensive simulations, something impossible within fixed-size, limited-time models.

CHAPTER 4

Randomized algorithms

4.1 Introduction

In probabilistic analysis, the algorithm is deterministic, meaning that for any given fixed input, the algorithm will always produce the same result and follow the same execution path each time it runs. This analysis relies on a specific technique:

- A probability distribution is assumed for the inputs.
- The behavior of the algorithm is then analyzed over this distribution, focusing on the item of interest (such as runtime or accuracy).

However, there are important caveats to this approach:

- Certain specific inputs may result in significantly worse performance.
- If the assumed distribution of inputs is inaccurate, the analysis may present a misleading or overly optimistic view of the algorithm's behavior.

In contrast, randomized algorithms introduce randomness into their execution. For a fixed input, the outcome may vary depending on the results of internal random decisions. The key features of randomized algorithms are:

- They generally work well with high probability for any input.
- There is, however, a small chance that they may fail on any given input, though this probability is low.

4.1.1 Analysis tools

Key analysis tools for randomized algorithms are:

- *Indicator variables*: to analyze a random variable X, which represents a combination of many random events, we can break it down using indicator variables. Let X_i be an indicator variable that captures the outcome of an individual event. The overall random variable is then expressed as the sum of these indicator variables:

$$X = \sum X_i$$

- *Linearity of expectation*: one powerful tool in the analysis of randomized algorithms is the linearity of expectation. Suppose we have random variables X, Y, and Z, where X is the sum of Y and Z. The expected value of X is then the sum of the expected values of Y and Z:
$$\mathbb{E}[X] = \mathbb{E}[Y + Z] = \mathbb{E}[Y] + \mathbb{E}[Z]$$
This holds true regardless of whether the variables are independent, making it a versatile and widely applicable tool in randomized algorithm analysis.

- *Recurrence relations*: recurrence relations often arise in the analysis of algorithms, especially when dealing with recursive procedures. These relations describe the behavior of an algorithm in terms of smaller subproblems and are fundamental in understanding the overall performance of both deterministic and randomized algorithms.

4.2 Hiring problem

Imagine you need to hire a new employee, and a headhunter sends you one applicant per day for n days. If an applicant is better than the current employee, you fire the current one and hire the new applicant. Since both hiring and firing are costly, you are interested in minimizing these operations.

We may have two estreme cases:

- *Worst-case scenario*: the headhunter sends applicants in increasing order of quality, meaning each new applicant is better than the previous one. In this case, you hire and fire each applicant, resulting in n hires.

- *Best-case scenario*: the best applicant arrives on the first day, so you hire them and make no further changes. The total cost is just one hire.

In the average case, the input to the hiring problem is a random ordering of n applicants. There are $n!$ possible orderings, and we assume that each is equally likely (though other distributions could be considered). We want to compute the expected cost of our hiring algorithm, which in this case is the expected number of hires.

Let $X(s)$ be the random variable representing the number of applicants hired given the input sequence s. We are interested in $\mathbb{E}[X]$, the expected number of hires. To solve this, we can break the problem down using indicator random variables. Instead of counting the total number of hires with a single random variable, we define n indicator variables—one for each applicant.

Let X_i be the indicator variable for whether applicant i is hired:

$$X_i = \begin{cases} 1 & \text{if applicant } i \text{ is hired} \\ 0 & \text{otherwise} \end{cases}$$

The total number of hires, X, is the sum of these indicator variables:

$$X = X_1 + X_2 + \cdots + X_n$$

Now, using the linearity of expectation, we have:

$$\mathbb{E}[X] = \mathbb{E}\left[\sum X_i\right] = \sum \mathbb{E}[X_i]$$

Next, we need to compute $\mathbb{E}[X_i]$, the probability that applicant i is hired. An applicant i is hired only if they are better than all previous applicants ($i-1$ applicants). For a uniformly random order of applicants, the probability that applicant i is better than the previous $i-1$ applicants is $\frac{1}{i}$.

Thus, the expected value for each X_i is:

$$\mathbb{E}[X_i] = \Pr(\text{applicant } i \text{ is hired}) = \frac{1}{i}$$

Finally, the expected total number of hires is:

$$\mathbb{E}[X] = \sum_{i=1}^{n} \frac{1}{i}$$

This sum is the harmonic series, which is bounded by $\ln n + 1$. Therefore, the average number of hires is approximately $\ln n$, which is significantly better than the worst-case scenario of n hires.

4.2.1 Active randomization

The analysis above assumes that the headhunter sends applicants in a random order. However, if the headhunter is biased you cannot rely on this randomness. In such cases, if you have access to the entire list of applicants in advance, you can take control by randomizing the input yourself.

By randomly permuting the list of applicants before interviewing them, you essentially convert the hiring problem into a randomized algorithm. This way, the hiring process no longer depends on the headhunter's input order, and you maintain the same expected number of hires, $O(\log n)$, regardless of the original order.

In general, randomized algorithms allow for multiple possible executions on the same input, which ensures that no single input can guarantee worst-case performance. Instead of assuming some distribution for the inputs, you actively create your own distribution, thereby moving from passive probabilistic analysis to a more robust, actively randomized approach. This is especially useful when facing potentially adversarial or biased inputs.

4.2.2 Summary

In summary, the hiring problem demonstrates the effectiveness of probabilistic analysis and randomized algorithms. By leveraging indicator variables and the linearity of expectation, we can show that, on average, the number of hires is much lower than in the worst case. Moreover, through active randomization, you can control the input order and reduce the number of hires to $O(\log n)$, making the process much more efficient in practice.

4.3 Randomized algorithms

Randomized algorithms can be broadly classified into two main types: Las Vegas algorithms and Monte Carlo algorithms. These categories are distinguished by whether the randomness affects the correctness of the solution or the efficiency of the algorithm.

4.3.1 Las Vegas algorithms

An example of a Las Vegas algorithm is a randomized sorting algorithm, where randomness influences the algorithm's performance, but it always produces the correct solution. The only aspect that varies between different runs of the algorithm is its running time.

- *Correctness*: guaranteed to produce the correct result every time.
- *Randomness*: affects only the running time, not the output.

For example, Randomized Quicksort is a Las Vegas algorithm. It randomly selects a pivot at each step to partition the array, and the sorting process continues recursively. Although the running time can vary depending on the choice of pivots, the final output will always be a correctly sorted array.

4.3.2 Monte Carlo algorithms

In contrast, the min-cut algorithm (such as Karger's Min-Cut Algorithm) is an example of a Monte Carlo algorithm, where randomness affects the correctness of the solution. The algorithm might return an incorrect answer (i.e., it may fail to find the minimum cut), but we can bound the probability of this happening. By running the algorithm multiple times, we can reduce the probability of error to a desired level.

- *Correctness*: may return an incorrect solution with a known probability.
- *Randomness*: affects both the running time and the correctness of the output.

For instance, Karger's min-cut algorithm works by randomly contracting edges in the graph until only two vertices remain. The cut between these two vertices is the candidate for the minimum cut. However, due to the randomness involved, the algorithm might not find the actual minimum cut, though it can do so with high probability if repeated enough times.

4.3.3 Comparison

Las Vegas Algorithms always give the correct solution, but running time may vary due to randomness. The focus is on ensuring that the algorithm always terminates with the correct result.

Monte Carlo Algorithms may produce an incorrect result with some probability, but this error can be controlled and reduced. These algorithms typically have a fixed running time but sacrifice certainty for speed.

Both types of algorithms highlight different ways to incorporate randomness. In Las Vegas algorithms, randomness is used for performance improvement without compromising correctness, whereas in Monte Carlo algorithms, randomness is used to trade some degree of correctness for efficiency.

4.4 Karger's min-cut algorithm

Let $G = (V, E)$ be a connected, undirected graph, where $n = |V|$ and $m = |E|$ represent the number of vertices and edges, respectively. For any subset $S \subset V$, the set $\delta(S) = \{(u, v) \in E : u \in S, v \in S'\}$ represents a cut, since removing these edges from G would disconnect the graph into two or more components. The goal of the min-cut problem is to find the cut with the smallest size, i.e., the fewest number of edges.

St-cut problem A closely related problem is the minimum st-cut problem. In this version, for specified vertices $s \in V$ and $t \in V$, we restrict attention to cuts $\delta(S)$ where $s \in S$ and $t \notin S$. The goal here is to find the cut that minimizes the number of edges crossing between S and S'.

Traditionally, the min-cut problem could be solved by computing $n-1$ minimum st-cuts, one for each pair of vertices. In the min-st-cut problem, we are given two vertices s and t, and the goal is to find the set S such that $s \in S$ and $t \notin S$, minimizing the size of the cut (S, S'), i.e., minimizing $|\delta(S)|$. By linear programming duality, the size of the minimum st-cut is equal to the value of the maximum st-flow. The fastest known algorithm for solving the max-st-flow problem runs in time $O(nm \log(\frac{n^2}{m}))$. Furthermore, all $n-1$ max-st-flow computations can be performed simultaneously within the same time bounds.

4.4.1 Karger's Solution

Karger introduced a randomized algorithm to solve the min-cut problem that avoids the need for max-flow computations. This clever approach is based on the idea of randomly contracting edges to shrink the graph while preserving the minimum cut with high probability.