Advanced Algorithms And Parallel Programming

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 analysis

1.1 Introduction

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

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*: 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*: 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*: 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 average bound 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	$T(n) = \Theta(n^2)$	Input in reverse order
Average	$T(n) = \Theta(n^2)$	All permutations equally likely
Best	$T(n) = \Theta(n)$	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:

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

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.

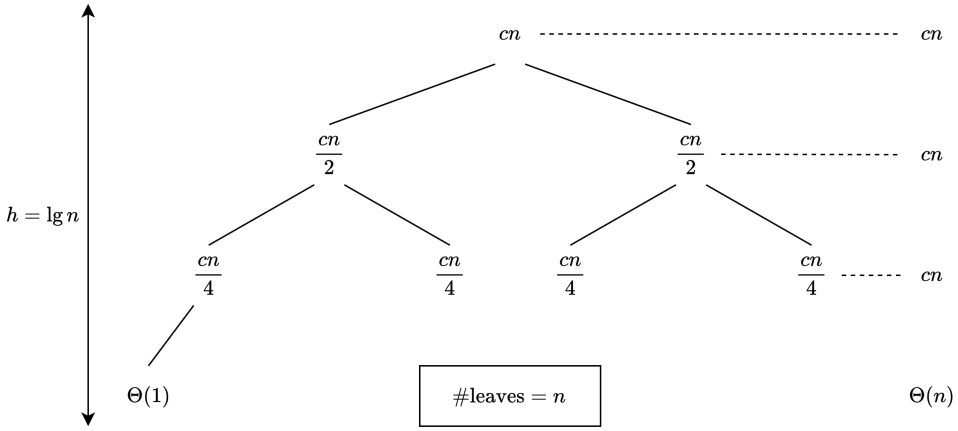


Figure 1.1: Partial recursion tree for merge sort algorithm

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

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

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 $0 < c < 1$, 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

Advanced algorithms

2.1 Divide and conquer algorithms

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:

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

2.1.1 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*: select half of the array to search for the element.
2. *Conquer*: check the middle element of the sub-array.
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:

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

2.1.2 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:

$$\Theta(\log n)$$

2.1.3 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 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.

2.1.4 VLSI layout

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

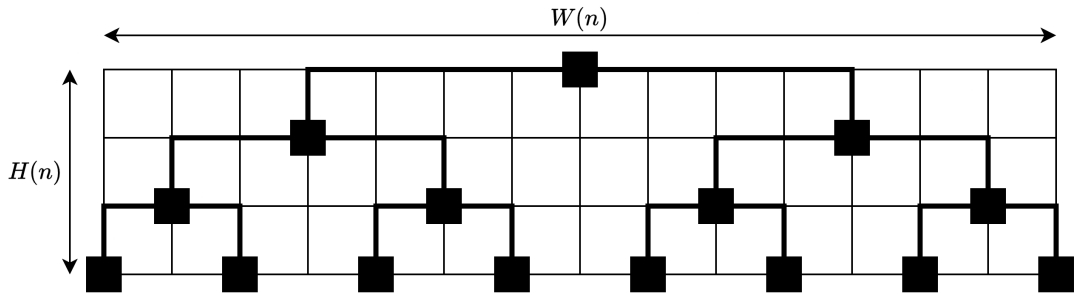


Figure 2.1: VLSI layout problem

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.

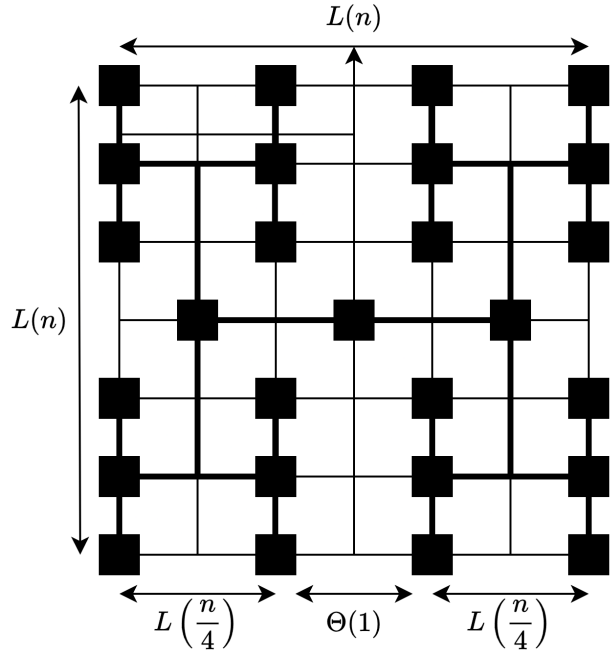


Figure 2.2: VLSI layout problem

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:

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

2.2 Dynamic Programming algorithms

Dynamic Programming is a problem-solving technique first introduced in the 1940s by Richard Bellman. It focuses on finding optimal solutions by breaking problems down into overlapping subproblems, solving each once, and storing the results to avoid redundant calculations. The term dynamic highlights the approach's adaptability to the time-varying nature of certain problems, while programming originally referred to developing an optimized sequence of decisions, akin to a military plan or schedule for logistics or training.

2.2.1 Longest Common Subsequence

Given two sequences $x[1 \dots m]$ and $y[1 \dots n]$, the objective of the Longest Common Subsequence (LCS) problem is to find the longest sequence of elements that appears in both x and y in the same relative order. A subsequence is derived by deleting some (or none) of the elements from a sequence without changing the order of the remaining elements. Thus, the LCS is the longest such sequence common to both x and y.

2.2.1.1 Naive algorithm

A naive approach to solving the LCS problem involves generating all possible subsequences of x and checking each one to see if it is also a subsequence of y. The steps for this brute-force algorithm are as follows:

1. *Generate all subsequences of* x: for a sequence of length m, there are 2^m possible subsequences. Each subsequence corresponds to a unique bit vector of length $m$where each bit indicates whether the corresponding element in x is included in that subsequence.

2. *Check each subsequence in* y: for each subsequence of x, verify if it also appears as a subsequence in y. This can be done by iterating over y and confirming that the elements of the subsequence appear in the same order within y.

3. *Track the LCS*: while iterating through all possible subsequences, keep a record of the longest one that is a subsequence of both x and y. Once all subsequences have been checked, the LCS found is the solution.

The naive approach is highly inefficient due to its exponential time complexity. With m elements in x, there are 2^m possible subsequences, as each element has the option to be included or excluded. Checking if a subsequence of x is also a subsequence of y takes $\mathcal{O}(n)$ time. Therefore, the total time complexity for the naive algorithm is:

$$T(n) = \mathcal{O}(n2^m)$$

2.2.1.2 Recursive algorithm

To solve the LCS problem using a recursive approach, we break it down into two main steps:

1. *Compute the length of the LCS*: define a recursive function that calculates the length of the LCS between prefixes of two sequences, x and y.

2. *Extend to contruct the LCS itself*: moodify the function to also record subsequence characters, allowing reconstruction of the LCS.

We define a two-dimensional array $c[i,j] = |\text{LCS}(x[1 \dots i], y[1 \dots j])|$, where each cell $c[i,j]$ represents the length of the LCS between the prefixes $x[1 \dots i]$ and $y[1 \dots j]$. The value at $c[m,n]$ then gives the length of the LCS for the entire sequences x and y.

Theorem 2.2.1. *The recursive relation for $c[i,j]$ is given by:*

$$\begin{cases} c[i-1, j-1] + 1 & \text{if } x[i] = y[j] \\ \max\{c[i-1,j], c[i,j-1]\} & \text{otherwise} \end{cases}$$

This recursive approach leverages the optimal substructure property of the LCS, meaning that the optimal solution to the LCS problem for sequences x and y depends on optimal solutions to subproblems involving prefixes of these sequences.

Definition (*Optimal substructure*)**.** An optimal solution to a problem instance includes optimal solutions to its subproblems.

Algorithm 4 Recursive LCS

```
function LCS(x, y, i, j)
    if x[i] = y[j] then
        c[i, j] = LCS(x, y, i − 1, j − 1) + 1
    else
        c[i, j] = max{LCS(x, y, i − 1, j), LCS(x, y, i, j − 1)}
    end if
    return c[i, j]
end function
```

In the worst-case scenario, when $x[i] \neq y[j]$, the algorithm recursively evaluates two subproblems for each pair (i,j), leading to an exponential time complexity of $\mathcal{O}(2^{m+n})$. Since many subproblems are recalculated multiple times, this recursive approach can be highly inefficient without further optimization.

Memoization To eliminate redundant calculations, we use memoization, storing results of previously solved subproblems in a table. When the algorithm encounters a subproblem it has solved before, it retrieves the stored result instead of recalculating it.

Definition (*Memoization*)**.** Memoization is a technique where, after computing the solution to a subproblem, we store it in a table so that future calls can retrieve the result directly, avoiding redundant work.

Algorithm 5 Memoized recursive LCS

```
procedure LCS(x, y, i, j)
    if c[i, j] = null then
        if x[i] = y[j] then
            c[i, j] = LCS(x, y, i − 1, j − 1) + 1
        else
            c[i, j] = max[LCS(x, y, i − 1, j),LCS(x, y, i, j − 1) ]
        end if
    end if
end procedure
```

By storing each subproblem's solution only once, the memoization approach reduces the time complexity to $\Theta(mn)$, where m and n are the lengths of sequences x and y. This is because we compute each of the mn subproblems only once. he space complexity is also $\Theta(mn)$, as we store each subproblem result in a table.

2.2.1.3 Dynamic Programming

The Dynamic Programming approach to solving the LCS problem avoids recursion by systematically building the solution from the smallest subproblems up to the full problem. This bottom-up approach is efficient in both time and space, eliminating the overhead of recursive calls.

The steps involved in the DP solution are as follows:

1. *Construct the table*: create a table c where $c[i, j]$ holds the length of the LCS for prefixes $x[1 \dots i]$ and $y[1 \dots j]$. Initialize $c[0, 0]$ and fill the table up to $c[m, n]$ based on the recurrence relation used in the recursive solution.

2. *Reconstruct the LCS*: after computing $c[m, n]$, use the table to backtrack from $c[m, n]$ to $c[0, 0]$, tracing the characters that contribute to the LCS.

Algorithm 6 Dynamic Programming LCS

```
procedure LCS(x, y)
    Initialize c[0 . . . m, 0 . . . n] to 0
    for i = 1 to m do
        for j = 1 to n do
            if x[i] = y[j] then
                c[i, j] = c[i − 1, j − 1] + 1
            else
                c[i, j] = max(c[i − 1, j], c[i, j − 1])
            end if
        end for
    end for
    return c[m, n]
end procedure
```

This approach has a time complexity of $\mathcal{O}(mn)$ as each entry of the $m \times n$ table is filled once. The space complexity is also $\mathcal{O}(mn)$, since we store each subproblem result in the table. This

Dynamic Programming method is both efficient and avoids redundant calculations, making it well-suited for large inputs.

2.2.2 Binary Decision Diagram

Binary Decision Diagrams (BDDs) are a compact data structure for representing Boolean functions. They improve on traditional approaches by storing evaluated sub-cases in memory, allowing for efficient retrieval and manipulation of Boolean expressions. BDDs are particularly valuable because they can be made canonical, which provides a unique representation for each Boolean function with a fixed variable ordering, enabling easier comparison and optimization.

One of the key benefits of BDDs is their efficiency in performing Boolean operations, such as conjunction, disjunction, and negation. However, the size of a BDD heavily depends on the chosen variable ordering; optimal ordering can significantly reduce the size and complexity of the BDD.

In a BDD, a Boolean function is represented as a Directed Acyclic Graph (DAG) with: single root node representing the function's entry point and two terminal nodes labeled 0 and 1, representing the function's output values. Each internal node is associated with a variable and has exactly two children, representing the function's behavior when that variable is assigned 1 or 0.

Ordered BDD An Ordered BDD (OBDD) applies only the ordering constraint, requiring that variables appear in a specific sequence along any path but without enforcing reduction.

Reduced Ordered BDD An ROBDD is a special type of BDD that is both reduced and ordered, resulting in a compact, canonical representation for Boolean functions. An ROBDD is derived from a Shannon co-factoring tree with two main modifications:

- *Reduction*: this process eliminates redundancy by:
  - Removing nodes with identical children.
  - Merging nodes with identical subgraphs, so that each unique subgraph appears only once in the diagram.
- *Ordering*: variables are processed in a fixed, consistent order along every path from the root to a terminal node. This ordering ensures that each path encounters variables in a predefined sequence, contributing to the canonical nature of ROBDDs.

The canonical structure of an ROBDD allows straightforward identification of the function's onset (the set of variable assignments that make the function evaluate to 1). This can be achieved by tracing all paths from the root to the 1-terminal node. Each such path represents a set of variable assignments that satisfy the Boolean function.

2.2.2.1 Implementation

The efficient implementation of BDDs relies on specific data structures that prevent redundancy, streamline Boolean operations, and optimize memory usage. The key components are:

- *Unique table*: ensures that each node in the BDD is unique, preventing duplication. This is typically implemented as a hash table, where the properties of each node serve as a key mapping to a unique existing or newly created node.

- *Computed table*: stores the results of previously computed operations to speed up repeated calculations. This table, which can be implemented as a hash table, indexes operations in the form (f, g, h), representing arguments passed to the ITE operator. If a result for (f, g, h) exists in the table, it can be retrieved without recomputation.

ITE operator The ITE operator is central to BDD operations, as it can implement any binary Boolean function. Mathematically, the ITE operator is defined as:

$$\text{ite}(f, g, h) = fg + f\bar{h}$$

Before adding a new node (v, g, h) is added to the BDD, the unique table is checked to avoid duplication. If a matching node exists, its pointer is reused; otherwise, a new node is created and added to the unique table.

Algorithm 7 Recursive ITE

```
function ITE(f, g, h)
    if f = 1 then
        return g
    end if
    if f = 0 then
        return h
    end if
    if g = h then
        return g
    end if
    if p = HashLookupComputedTable(f, g, h) then
        return p
    end if
    v = TopVariable(f, g, h)
    f_n = ITE(f_v, g_v, h_v)
    g_n = ITE(f_v̄, g_v̄, h_v̄)
    if f_n = g_n then
        return g_n
    end if
    if !(p = HashLookupComputedTable(v, f_n, g_n)) then
        p = CreateNode(v, f_n, g_n)
        Insert p into the unique table
    end if
    key = HashKey(f, g, h)
    InsertComputedTable(p, key)
    return p
end function
```

Complemented edges Complemented edges optimize memory and operation efficiency by allowing the BDD to represent negated functions without additional nodes. This technique reduces storage needs and accelerates NOT and ITE operations. To maintain the canonical form of the BDD, certain edge equivalences are managed, ensuring consistent representation across nodes.

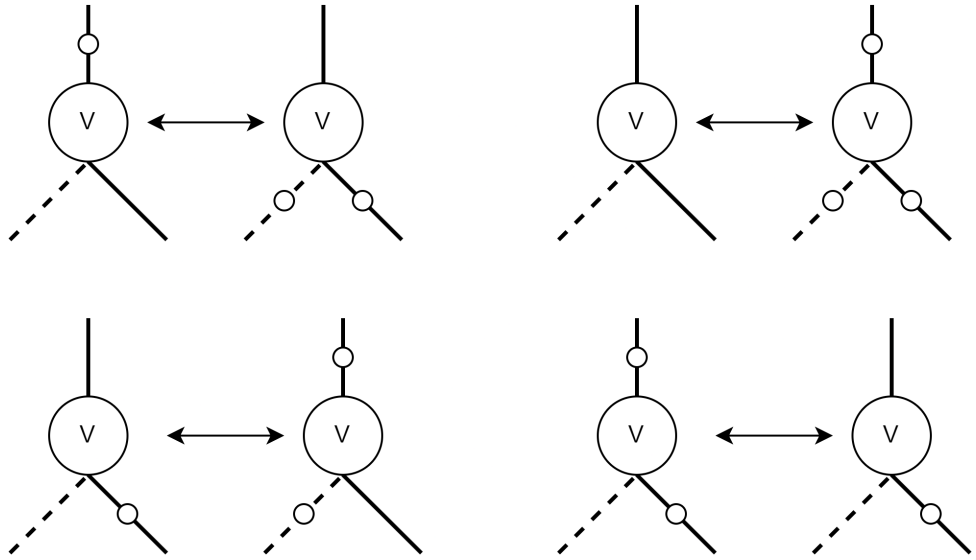


Figure 2.3: Edge equivalence

Optimization To maximize matches in the computed table, conventions are applied, such as selecting the argument with the smallest top variable (or smallest pointer in case of a tie) as the first argument. Caching mechanisms—including local caching, operation (specific caching, and shared caching) also contribute to efficient BDD operations.

2.2.2.2 Garbage collection

Effective garbage collection is essential for managing memory usage in BDDs, as unreferenced nodes consume resources and degrade performance. Garbage collection in BDDs typically involves:

1. *Reference counting*: each node tracks the number of references to it, allowing immediate deallocation when the reference count drops to zero.

2. *Mark-and-sweep*: this process involves marking nodes with references during a traversal and then sweeping through to delete unmarked nodes in a second pass.

Garbage collection timing is important to optimize resource usage without excessive overhead. Nodes can be deallocated on demand, at specific intervals, or after predefined operations. Since computed tables do not track references, they are cleared separately during the garbage collection process to maintain efficiency.

2.3 Randomized algorithms

Probabilistic analysis in algorithms assumes that the algorithm itself is deterministic; for a given fixed input, it will produce the same output and follow the same sequence of operations every time it runs. This analysis model considers a probability distribution over the possible inputs, evaluating the algorithm's performance based on this distribution. While this can provide useful insights into average-case behavior, it has limitations. or instance, certain specific inputs may lead to particularly poor performance, and if the assumed distribution does not accurately represent real-world inputs, the analysis may yield a misleading or overly optimistic view of the algorithm's expected efficiency.

In contrast, randomized algorithms incorporate randomness into their execution process, introducing variability in their behavior even when given a fixed input. Due to this inherent randomness, a randomized algorithm may produce different results or follow different execution paths on the same input in separate runs. Generally, randomized algorithms are designed to perform well with high probability across any input, though there remains a small probability of failure on any given run.

2.3.1 Taxonomy

	Las Vegas	**Monte Carlo**
Randomness effect	Running time	Running time Solution correctness
Efficiency (polynomial bound)	Expected running time	Worst-case running time

Monte Carlo algorithms are classified further based on their error probabilities. A Monte Carlo algorithm with two-sided error has a nonzero probability of error for both possible outputs. In contrast, a one-sided error algorithm guarantees correctness for at least one of the outputs, meaning it has zero error probability for that output.

2.4 Minimum cut problem

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 $\delta(S) = \{(u, v) \in E \mid u \in S, v \in S'\}$ defines a cut, separating the vertices in S from those in $S' = V \setminus S$. The minimum cut problem seeks to identify a cut with the fewest edges connecting S and S', effectively partitioning the graph with minimal separation.

2.4.1 Naive algorithm

A traditional approach to solving the minimum cut problem is to compute $n - 1$ minimum source-target cuts, one for each possible pair of vertices. A source-target cut partitions the graph into two disjoint sets such that one subset contains a designated source vertex s and the other contains a designated target vertex t.

The size of the minimum source-target cut is equivalent to the maximum flow between s and t. The most efficient algorithm known for the maximum flow problem has a time complexity of:

$$T(n) = \mathcal{O}\left(nm \log\left(\frac{n^2}{m}\right)\right)$$

Here, n is the number of vertices and m is the number of edges.

2.4.2 Karger's algorithm

Karger introduced a randomized algorithm that avoids explicit maximum flow calculations by using edge contraction to iteratively simplify the graph. This contraction process preserves the minimum cut with high probability, resulting in a more efficient approach.

Edge contraction involves merging two vertices connected by an edge, $e = (u, v)$, into a single new vertex w. The contraction replaces all edges incident to u or v with edges incident to w, while removing any self-loops created by this merging process.

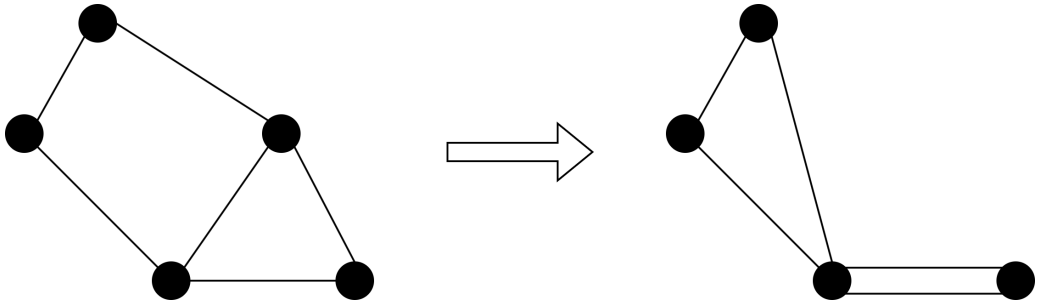


Figure 2.4: Edge contraction

Definition (*Edge contraction*)**.** For a multigraph $G = (V, E)$ without self-loops, contracting an edge $e = \{u, v\} \in E$, denoted $G \setminus e$, results in:

1. Replacing vertices u and v with a new vertex w.
2. Redirecting all edges incident to u and v to w.
3. Removing any self-loops involving w.

After contraction, the graph $G \setminus e$ remains a multigraph. Importantly, contracting (u, v) does not affect cuts where u and v are both in the same set.

Algorithm Karger's algorithm for the minimum cut works as follows:

1. Select an edge uniformly at random and contract its endpoints.
2. Repeat the contraction process until only two vertices remain.

The two remaining vertices form a partition (S, S') of the original graph, where the edges connecting S and S' defines the cut $\delta(S)$ in G.

Lemma 2.4.1. *For a minimum cut $\delta(S)$ in the graph $G = (V, E)$, Karger's algorithm produces this minimum cut with probability at least:*

$$\Pr(\text{minimum cut}) \geq \frac{1}{\binom{n}{2}}$$

To increase the success probability, we repeat Karger's algorithm $l \cdot \binom{n}{2}$ times. The probability that at least one run will successfully produce the minimum cut is:

$$\Pr(\text{one success}) \geq 1 - e^{l}$$

Setting $l = c \log n$ reduces the error probability to less than:

$$\Pr(\text{error}) \leq \frac{1}{n^c}$$

Complexity One run of Karger's algorithm takes $\mathcal{O}(n^2)$ time. By repeating the algorithm $\mathcal{O}(n^2 \log n)$ times, we obtain a randomized algorithm with total time complexity:

$$T(n) = \mathcal{O}(n^4 \log n)$$

2.4.3 Karger and Stein algorithm

Karger and Stein refined Karger's original minimum cut algorithm to improve efficiency by enhancing the edge contraction process. The core insight lies in understanding the telescoping product that emerges when calculating the probability of preserving edges in the minimum cut set $\delta(S)$ during contractions.

In the early stages, it's unlikely that an edge from the minimum cut set is contracted. However, as the graph reduces in size, the probability of contracting such an edge increases. By focusing on the probability that a fixed minimum cut $\delta(S)$ survives contraction to a subgraph with l vertices, we find that:

$$\Pr(\text{cut survives}) = \frac{\binom{l}{2}}{\binom{n}{2}}$$

Setting $l = \frac{n}{\sqrt{2}}$ ensures a survival probability of at least $\frac{1}{2}$. This suggests that, on average, running two trials of the algorithm should be sufficient to find the minimum cut with high probability.

Algorithm The Karger-Stein algorithm proceeds as follows for a multigraph G with at least six vertices:

1. Run the edge contraction algorithm on $\frac{n}{\sqrt{2}} + 1$ vertices.
2. Recur on the resulting contracted graph.
3. Repeat these steps twice, then return the smaller of the two cuts found.

Notably, setting the recursion threshold to six vertices affects only the constant factor of the runtime, without impacting the asymptotic complexity.

Algorithm 8 Karger and Stein

```
1: function CONTRACT(G = (V, E), t)
2:     while |V| > t do
3:         Choose e ∉ E uniformly at random
4:         G = G \ e
5:     end while
6:     return G
7: end function

8: function FASTMINCUT(G = (V, E))
9:     if |V| < 6 then
10:        return mincut(V)
11:    else
12:        t = ⌈1 + |V|/√2⌉
13:        G_1 = CONTRACT(G, t)
14:        G_2 = CONTRACT(G, t)
15:        return min{FASTMINCUT(G_1), FASTMINCUT(G_2)}
16:    end if
17: end function
```

Complexity The recurrence relation for the running time of the Karger-Stein algorithm is:

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

Which solves to a complexity of $\mathcal{O}(n^2 \log n)$.

The algorithm's success probability at each recursive step is at least $\geq \frac{1}{2}$. To increase the probability of finding the minimum cut, we repeat the algorithm multiple times. The probability of success is:

$$\Pr(\text{success}) = \Omega\left(\frac{1}{\log n}\right)$$

Thus, to ensure the algorithm succeeds with high probability we need to run the algorithm $\mathcal{O}(\log^2 n)$ times. Thus, the total time complexity of the Karger-Stein algorithm is:

$$T(n) = \mathcal{O}(n^2 \log^3 n)$$

Corollary 2.4.1.1. *Any graph has at most $\mathcal{O}(n^2)$ distinct minimum cuts.*

2.5 Sorting problem

The sorting problem is a fundamental computational task in which a collection of elements is arranged in a specific order, typically ascending or descending. Given an unsorted list or array, the goal is to rearrange the elements to follow a predefined sequence based on a chosen criterion, such as numerical or lexicographical order.

2.5.1 Quicksort

Quicksort, introduced by Hoare in 1962, is a highly efficient, in-place, divide-and-conquer sorting algorithm known for its practical performance across a variety of applications. By partitioning data around a pivot element, Quicksort can achieve efficient sorting with minimal extra storage, making it one of the most widely used sorting algorithms.

The Quicksort algorithm works as follows:

1. *Divide*: select a pivot element from the array, then partition the array into two subarrays. Elements less than or equal to the pivot form the left subarray. Elements greater than or equal to the pivot form the right subarray.
2. *Conquer*: recursively apply Quicksort to each of the two subarrays.
3. *Combine*: since the subarrays are sorted in place, no additional merging is needed.

The efficiency of Quicksort depends on the partitioning step, which operates in $\mathcal{O}(n)$ time.

Algorithm 9 Quicksort

```
1: function PARTITION(A, p, q)
2:     x = A[p]
3:     i = p
4:     for j = p + 1 to q do
5:         if A[j] ≤ x then
6:             i = i + 1
7:             exchange A[i] and A[j]
8:         end if
9:     end for
10:    exchange A[p] and A[i]
11:    return i
12: end function

13: procedure QUICKSORT(A, p, r)
14:    if p < r then
15:        q = PARTITION(A, p, r)
16:        QUICKSORT(A, p, q − 1)
17:        QUICKSORT(A, q + 1, r)
18:    end if
19: end procedure
```

The performance of Quicksort varies based on the choice of pivot and the input data. Here are the primary cases:

- *Worst-case*: the pivot always ends up at one of the ends of the array, resulting in highly unbalanced partitions. This scenario, often due to already sorted or reverse-sorted data when a poor pivot is chosen, yields a time complexity of $\Theta(n^2)$.
- *Average case*: the pivot splits the array into reasonably balanced parts. This is achieved with a random or median pivot selection, leading to a time complexity of $\Theta(n \log n)$, which is efficient for large datasets.

- *Best case*: the pivot consistently splits the array into two equal halves, minimizing the depth of recursive calls. This optimal scenario also results in a time complexity of $\Theta(n \log n)$.

2.5.2 Randomized Quicksort

Randomized Quicksort is an improved variant of the classic Quicksort algorithm that selects a pivot randomly from the array, significantly reducing the chance of worst-case performance. By ensuring the pivot choice is independent of the input structure, Randomized Quicksort achieves efficient performance on average for various inputs.

The randomized selection of the pivot minimizes the probability of consistently poor partitions, where the pivot might otherwise split the array in highly unbalanced ways. With a randomized pivot, the expected time complexity becomes $\Theta(n \log n)$, independent of any initial ordering of the data.

Analysis Let X denote the running time of Randomized Quicksort on an input of size n, assuming that each pivot selection is independent and uniformly random. Define an indicator variable X_k for the event that a partition results in a split of k elements on one side and $n-k-1$ elements on the other:

$$X_k = \begin{cases} 1 & \text{if PARTITION generates a } k \mid (n-k-1) \text{ split} \\ 0 & \text{otherwise} \end{cases}$$

Since any element can be chosen as the pivot with equal probability, the expected value of X_k is:

$$\mathbb{E}[X_k] = \Pr(X_k = 1) = \frac{1}{n}$$

Thus, the expected running time $\mathbb{E}[T(n)]$ can be written in terms of the recursive costs of partitioning:

$$\mathbb{E}[T(n)] = \mathbb{E}\left[\sum_{k=0}^{n-1} X_k \left(T(k) + T(n-k-1) + \Theta(n)\right)\right]$$

This simplifies to:

$$\mathbb{E}[T(n)] = \frac{2}{n} \sum_{k=1}^{n} \mathbb{E}[T(k)] + \Theta(n)$$

For sufficiently large $n \geq 2$, this recursive relation leads to:

$$\mathbb{E}[X] \leq \frac{2}{n} \sum_{k=1}^{n} ak \log k + \Theta(n) \leq an \log n$$

Thus, for a sufficiently large constant a, the $\Theta(n)$ term is dominated, leading to an overall time complexity of:

$$T(n) = \mathcal{O}(n \log n)$$

Practical performance In practice, Randomized Quicksort often outperforms Merge Sort, typically running at least twice as fast due to its efficient in-place operations and reduced memory usage. With careful code tuning and optimized implementations, Quicksort's performance can be further enhanced. Its contiguous memory access patterns make it highly cache-friendly, and it handles virtual memory effectively.

2.5.3 Comparison sort analysis

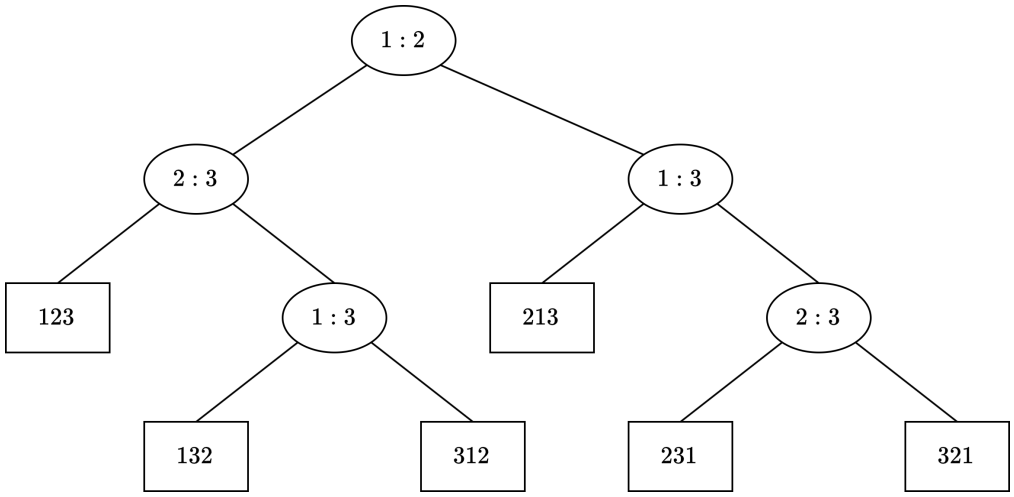
All the sorting algorithms discussed so far are comparison sorts, where element ordering is determined by comparing pairs of elements. The best worst-case time complexity for these algorithms is $\mathcal{O}(n \log n)$.

To understand this lower bound, consider sorting an array $\langle a_1, a_2, \ldots, a_n \rangle$. We can represent each sequence of comparisons made by a sorting algorithm as a path in a decision tree:

- Each internal node in this tree represents a comparison between two elements a_i and a_j.
- The left child of a node represents the branch taken if $a_i \leq a_j$, while the right child represents the branch if $a_i > a_j$.

Each leaf node in the decision tree corresponds to a unique, final sorted order (or permutation) of the array. Therefore, this tree models all possible sequences of comparisons that could occur for different input configurations.

Example:
Consider sorting the array $\langle 9, 4, 6 \rangle$. A decision tree for this array might look as follows, showing different comparison paths:



The height of the decision tree represents the worst-case number of comparisons needed to sort the array, which corresponds to the algorithm's worst-case running time. Since there are $n!$ possible ways to order n distinct elements, the decision tree must have at least $n!$ leaves to account for every possible permutation.

For a binary tree of height h, the maximum number of leaves is 2^h, so we must have:

$$2^h \geq n!$$

Taking the logarithm of both sides and applying Stirling's approximation, $n! \approx \left(\frac{n}{e}\right)^n$, we get:

$$\log n! \geq n \log n - n \log e$$

Thus, the minimum height $H(n)$ of the decision tree is:

$$H(n) = \Omega(n \log n)$$

This proves that any comparison-based sorting algorithm has a worst-case time complexity of $\Omega(n \log n)$.

Theorem 2.5.1. *Any decision tree that can sort n elements must have height $\Omega(n \log n)$.*

Corollary 2.5.1.1. *Heapsort and Merge Sort achieve this asymptotic lower bound, making them optimal comparison-based sorting algorithms.*

2.5.4 Counting sort

Counting sort is a non-comparison-based sorting algorithm that efficiently organizes elements by leveraging their value range rather than making direct comparisons between them. This makes it particularly useful for sorting arrays with integer elements drawn from a small range of possible values.

Algorithm Counting sort takes as input an array $A[n]$, where each element $A[j] \in \{1, \ldots, k\}$, and returns a sorted array $B[n]$. The algorithm also requires an auxiliary array $C[k]$ for counting the frequency of each element in $A[n]$.

The Counting Sort algorithm works in the following steps:

1. *Initialize the counting array*: initialize an array C of size k, where each element is initially set to zero. This array will store the frequency of each element in A.
2. *Count the occurrences*: traverse the input array A, and for each element A[j], increment the corresponding position in array C.
3. *Compute the prefix sum*: update array C by converting it to a prefix sum array. This allows the algorithm to determine the final position of each element in the sorted output.
4. *Build the sorted array*: iterate through the input array A in reverse order, and place each element at its correct position in the output array B, using the values in C for positioning. As elements are placed into B, decrement their corresponding counts in C.

Algorithm 10 Counting Sort

```
1: for i = 1 to k do                          ▷ Set all elements of array C to zero.
2:     C[i] = 0
3: end for                                    ▷ Complexity Θ(k)
4: for j = 1 to n do   ▷ Count how many times each element appears in the input array A
5:     C[A[j]] = C[A[j]] + 1
6: end for                                    ▷ Complexity Θ(n)
7: for i = 2 to k do                          ▷ Compute the prefix sum for each element in C
8:     C[i] = C[i] + C[i − 1]
9: end for                                    ▷ Complexity Θ(k)
10: for j = n to 1 do          ▷ Place elements into output array B and reduce counters in C
11:     B[C[A[j]]] = A[j]
12:     C[A[j]] = C[A[j]] − 1
13: end for                                   ▷ Complexity Θ(n)
```

Example:
Let's consider an example where we sort the array $A = \langle 4, 1, 3, 4, 3 \rangle$, with $k = 4$.

1. *Initial state*: initialize the array C to all zeros: $C = \langle 0, 0, 0, 0 \rangle$.

2. *Count elements*: count the occurrences of each element in A, resulting in: $C = \langle 1, 0, 2, 2 \rangle$.

3. *Compute the prefix sum*: compute the prefix sum over C: $C = \langle 1, 1, 3, 5 \rangle$.

4. *Place elements into output array*: starting from the last element of A, place elements into their correct position in B using the cumulative counts in C, and decrement the counts as we go. The final result will be: $B = \langle 1, 3, 3, 4, 4 \rangle$ and $C = \langle 0, 1, 1, 3 \rangle$.

The time complexity of Counting Sort is the sum of the complexities of each of the four loops

$$T(n) = \mathcal{O}(n) + \mathcal{O}(k) + \mathcal{O}(n) + \mathcal{O}(k) = \mathcal{O}(n + k)$$

If $k = \mathcal{O}(n)$, then counting sort runs in linear time:

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

Definition (*Stable sorting algorithm*)**.** A sorting algorithm is called stable if it preserves the relative order of equal elements from the input array.

Property 2.5.1. Counting Sort is a stable sort.

2.5.5 Radix sort

Radix sort is a non-comparative sorting algorithm that sorts numbers digit by digit, starting from the least significant digit to the most significant digit. It can be highly efficient for large datasets when certain conditions are met.

The idea behind radix sort is to process each digit of the numbers, from the least significant to the most significant, sorting them progressively by each digit's place value. This approach avoids direct comparisons between elements, making it suitable for sorting large datasets when combined with a stable auxiliary sorting algorithm.

Assume the numbers have already been sorted by the least significant $t - 1$ digits. Now, sort the numbers based on the t-th digit. If two numbers differ in the t-th digit, they will be correctly ordered after the pass. If they are identical in the t-th , they retain their relative order due to the stability of the auxiliary sort.

Algorithms The teps for radix sort are:

- *Sort by least significant digit*: sort the numbers based on the least significant digit, using a stable sorting algorithm like counting sort.
- *Iterate through remaining digits*: repeat the sorting process for each more significant digit, ensuring that the relative order of numbers is maintained between passes.
- *Final order*: after all digits have been processed, the numbers are fully sorted.

Analysis To analyze the efficiency of radix sort, we assume that counting sort is used as the stable sorting method for each digit. Suppose we are sorting n integers, where each integer is represented by b bits. Each integer can be thought of as having $\frac{b}{r}$ digits, where each digit is based on 2^r possible values. Each pass of counting sort processes n elements and sorts them based on a single digit, requiring $\Theta(n + 2^r)$ time. Since there are $\frac{b}{r}$ passes (each pass sorting based on one digit), the overall time complexity of radix sort is:

$$T(n) = \Theta\left(\frac{b}{r}(n + 2^r)\right)$$

Optimization To optimize radix sort, we aim to minimize the total running time. Increasing r, the number of bits used for each digit, reduces the number of passes $\frac{b}{r}$, but it also increases the cost of processing each digit, as 2^r grows exponentially.

For optimal efficiency, we want to avoid letting 2^r exceed n, since this would lead to unnecessary overhead. For efficiency, we should avoid letting $2^r > n$.

The optimal choice for r is typically $r = \log n$, as it balances the number of passes and the digit processing cost. Thus, the overall time complexity becomes:

$$T(n) = \Theta\left(\frac{bn}{\log n}\right)$$

Considerations In practice, radix sort is particularly efficient for large datasets, especially when the number of digits is relatively small compared to the number of elements. It is simple to implement and does not require comparisons between elements. However, radix sort has poorer cache locality and memory access patterns compared to algorithms like Quicksort, which can negatively affect performance for smaller datasets or systems with limited memory bandwidth.

2.6 Selection problem

The selection problem involves finding the element of a specified rank in a set of n distinct numbers. Given an integer i where $1 \leq i \leq n$, the task is to return the element that is larger than exactly $i - 1$ other elements in the set. We can have three extreme cases: minimum element ($i = 1$), maximum element ($i = n$), or median element.

2.6.1 Naive algorithm

A straightforward approach to solving the selection problem is to first sort the array and then return the i-th smallest element from the sorted array. The worst-case running time for this approach is dominated by the sorting step, which takes $\Theta(n \log n)$ time. Selecting the i-th element from the sorted array is a constant-time operation, resulting in a total complexity of:

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

While this solution is simple, it is not the most efficient, as it relies on sorting the entire array, even though only one element is ultimately needed.

However, there are more efficient algorithms that can solve the selection problem in linear time. Two popular approaches are:

- *Quickselect*: this algorithm is based on the partitioning method of Quicksort, but instead of recursively sorting both sides of the partition, it only recurses on the side that contains the desired element. Quickselect has an average-case time complexity of $\mathcal{O}(n)$.
- *Median of medians*: this more sophisticated approach uses a median of medians strategy to ensure that each partition step reduces the problem size by a constant fraction. The median of medians algorithm has a worst-case time complexity of $\mathcal{O}(n)$, making it more predictable than Quickselect in terms of performance.

2.6.2 Minmax

To determine the minimum or maximum of a set of n elements, an optimal approach requires exactly $n - 1$ comparisons.

Algorithm 11 Minimum and maximum

```
1: function MINIMUM(A)
2:     min = A[1]
3:     for i = 2 to length(A) do
4:         if min > A[i] then
5:             min = A[i]
6:         end if
7:     end for
8:     return min
9: end function

10: function MAXIMUM(A)
11:     max = A[1]
12:     for i = 2 to length(A) do
13:         if max < A[i] then
14:             max = A[i]
15:         end if
16:     end for
17:     return max
18: end function
```

This algorithm performs exactly $n-1$ comparisons, making it optimal for finding either the minimum or maximum in a set.

If both the minimum and maximum are needed, a naive approach would be to execute two passes over the array resulting in $2n-2$ comparisons. However, a more efficient approach allows both the minimum and maximum to be found in fewer than $3\left\lfloor \frac{n}{2} \right\rfloor$ comparisons.

The optimized approach works by comparing elements in pairs and adjusting the minimum and maximum accordingly. This reduces the number of total comparisons by approximately 25%.

2.6.3 Quickselect

Quickselect is an efficient, divide-and-conquer algorithm designed to find the i-th smallest element in an unsorted array with an expected time complexity of $\mathcal{O}(n)$. The algorithm builds

on the principles of randomized quicksort by using a pivot to partition the array, but it only recurses on the side that contains the desired element, reducing unnecessary work.

Algorithm 12 Quickselect

```
1: function RAND-SELECT(A, p, q, i)
2:     if p = q then
3:         return A[p]
4:     end if
5:     i = RAND-PARTITION(A, p, q)
6:     k = r − p + 1                                   ▷ k = rank(A[r])
7:     if i = k then
8:         return A[r]
9:     end if
10:    if i < k then
11:        return RAND-SELECT(A, p, r − 1, i)
12:    else
13:        return RAND-SELECT(A, r + 1, q, i − k)
14:    end if
15: end function
```

The running time of Quickselect depends on the quality of the partitioning achieved by the random pivot. This results in the following cases:

- *Best case*: if each partition splits the array evenly, the recurrence relation becomes:

$$T(n) = T\left(\frac{9}{10}n\right) + \Theta(n) = \Theta(n)$$

  Solving this recurrence yields $\Theta(n)$, meaning that Quickselect runs in linear time when the partition is balanced.

- *Worst case*: if each partition results in only one element on one side and the rest on the other, the recurrence relation is:

$$T(n) = T(n-1) + \Theta(n)$$

  This gives $\Theta(n^2)$, leading to quadratic time complexity in the worst case, although this is rare in practice due to random pivot selection.

Analysis The analysis involves defining the expected running time $\mathbb{E}[T(n)]$ and taking into account the probability distribution of possible splits. Let X_k be an indicator variable for whether the partition creates a $k \mid (n-k-1)$ split:

$$X_k = \begin{cases} 1 & \text{if PARTITION generates a } k \mid n-k-1 \text{ split} \\ 0 & \text{otherwise} \end{cases}$$

The expected running time is then:

$$T(n) = \sum_{k=0}^{n-1} X_k \left(T\left(\max\{k, n-k-1\}\right) + \Theta(n)\right)$$

Taking expectations, we get:

$$\mathbb{E}[T(n)] = \mathbb{E}\left[\sum_{k=0}^{n-1} X_k \left(T\left(\max\left\{k, n-k-1\right\}\right) + \Theta(n)\right)\right]$$

By applying bounds and the principle of linearity of expectation, it can be shown that:

$$\mathbb{E}[T(n)] \leq cn$$

For a constant $c > 0$, confirming that Quickselect achieves $\Theta(n)$ expected time complexity.

Practical performance Quickselect is highly efficient in practice, often outperforming deterministic selection algorithms due to its linear average-case time. The worst-case $\Theta(n^2)$ performance is rare, especially if a good pivot strategy or random selection is used. Its efficiency makes it a popular choice in scenarios where the expected linear time is sufficient for robust performance.

2.6.4 Median of medians

The Median of Medians algorithm is a deterministic selection algorithm that guarantees worst-case linear time complexity for selecting the i-th smallest element in an unsorted array. Unlike randomized algorithms, this method avoids the risk of quadratic behavior and provides a reliable worst-case performance.

Algorithm 13 Median of medians

```
1: function SELECT(i, n)
2:     Divide the array in 5 elements groups (each with the median)      ▷ Complexity Θ(n)
3:     Recursively select the median x of the groups medians as pivot    ▷ Complexity T(n/5)
4:     Partition around the pivot x, and let k = rank(x)                 ▷ Complexity Θ(n)
5:     if i = k then                                                     ▷ Complexity T(3n/4)
6:         return x
7:     else if i < k then
8:         SELECT(i-th smallest element in the lower part, n)
9:     else
10:        SELECT((i − k)-th smallest element in the upper part, n)
11:    end if
12: end function
```

Analysis To understand why the algorithm is efficient, note that choosing x as the median of medians ensures a reasonably balanced partition. Specifically, at least half of the group medians are guaranteed to be less than or equal to x. Since each group has five elements, at least $\left\lfloor \frac{n}{10} \right\rfloor$ elements in A are smaller than x, and at least $\left\lfloor \frac{n}{10} \right\rfloor$ elements are larger. Therefore, each partition discards at least $\left\lfloor \frac{3n}{10} \right\rfloor$ elements, ensuring significant progress with each recursive step. The recurrence relation for the algorithm's time complexity is given by:

$$T(n) = T\left(\frac{1}{5}n\right) + T\left(\frac{3}{4}n\right) + \Theta(n)$$

This relation can be solved to yield $T(n) = \Theta(n)$, proving that the algorithm runs in linear time.

Practical considerations While the median of medians algorithm offers a strong theoretical guarantee of linear time, its practical efficiency is often hindered by relatively high constant factors in its complexity. Here are a few key points about its real-world performance:

- *Work per level*: at each level of recursion, the work done is a fraction of the previous level. Although the algorithm remains linear, the constant factors are substantial due to the overhead of partitioning and the recursive calculation of medians.
- *Comparisons*: altough median of medians is optimal in the worst case, it is often outperformed in practice by the randomized Quickselect algorithm. Quickselect, with its lower constant factors, tends to be faster on average, even though its worst-case complexity is $\Theta(n^2)$.

The Median of Medians algorithm is particularly valuable in applications where a strong worst-case guarantee is essential, ensuring linear time regardless of input characteristics. However, in practical scenarios with large datasets, the randomized Quickselect algorithm is often preferred for its faster average performance, despite the possibility of quadratic worst-case behavior.

2.7 Primality problem

The primality problem involves determining whether a given integer $n \geq 2$ is a prime number.

Definition (*Prime number*)**.** An integer $p \geq 2$ is called prime if and only if it has no positive divisors other than 1 and itself.

2.7.1 Naive algorithm

The simplest way to test if a number n is prime is to check if it has any divisors other than 1 and itself. Since any factor of n greater than $\sqrt{n}$ would have a corresponding factor smaller than $\sqrt{n}$, we only need to check divisibility up to $\sqrt{n}$.

Algorithm 14 Naive primality test

```
1: if n = 2 then
2:     return true
3: end if
4: if n is even then
5:     return false
6: end if
7: for i = 1 to √(n/2) do
8:     if 2i + i divides n then
9:         return false
10:    end if
11: end for
12: return true
```

The time complexity of this naive algorithm is $\mathcal{O}(\sqrt{n})$.

2.7.2 Fermat primality test

To improve efficiency, we can use a probabilistic primality test based on Fermat's Little Theorem, which states:

Theorem 2.7.1 (Fermat)**.** *If p is a prime number and a and integer a is an integer such that $0 < a < p$, then $a^{p-1} \mod p = 1$.*

This theorem leads to a simple test for primality. If n is prime, $a^{n-1} \mod n = 1$ for some randomly chosen a. However, if n is composite, it may still satisfy this condition for certain a, in which case it is called a pseudoprime to base a.

Algorithm 15 Fermat's primality test

```
if 2^{n-1} mod n = 1 then
    n is possibly prime
else
    n is composite
end if
```

The Fermat test runs in $\mathcal{O}(\log^2 n)$ using modular exponentiation. However, it can mistakenly classify some composite numbers as prime (false positives).

2.7.3 Carmichael primality test

Definition (*Carmichael number*)**.** A composite number $n \geq 2$ is a Carmichael number if, for every integer a coprime to n, it holds that $a^{n-1} \mod n = 1$

Algorithm 16 Carmichael's primality test

```
Randomly choose a ∈ [2, n − 1]
if a^{n-1} mod n = 1 then
    n is possibly prime
else
    n is composite
end if
```

2.7.4 Miller-Rabin primality test

The Miller-Rabin test improves on Carmichael's test by checking additional properties that only hold for prime numbers. Specifically, it looks for non-trivial square roots of 1 $\mod n$.

Definition (*Non-trivial square root*)**.** An number a is a non-trivial square root of 1 $\mod n$ if:

$$a^2 \mod n = 1 \qquad a \neq 1 \qquad a \neq n - 1$$

The Miller-Rabin test randomly selects bases and tests whether they exhibit properties consistent with a prime modulus.

Algorithm 17 Miller-Rabin primality test

```
1: function POWER(a, p, n)
2:     if p = 0 then                                   ▷ compute a^p mod n
3:         return 1
4:     end if
5:     x = POWER(a, p/2, n)
6:     res = (x · x) % n
7:     if res = 1 and x ≠ 1 and x ≠ n − 1 then     ▷ check x^2 mod n = 1 and x ≠ 1, n − 1
8:         isProbablyPrime = false
9:     end if
10:    if p % 2 = 1 then
11:        res = (a · res) % n
12:    end if
13:    return res
14: end function

15: function PRIMALITYTEST(n)
16:    a = RANDOM(2, n − 1)
17:    isProbablyPrime = true
18:    result = POWER(a, n − 1, n)
19:    if res ≠ 1 or !isProbablyPrime then
20:        return false
21:    else
22:        return true
23:    end if
24: end function
```

Each iteration of the Miller-Rabin test has a low probability of incorrectly identifying a composite number as prime, and repeating the test k times reduces this probability to $\left(\frac{1}{4}\right)^k$.

Theorem 2.7.2. *If p is prime and $0 < a < p$, the only solutions to $a^2 \mod p = 1$ are $a = 1$ and $a = p - 1$.*

Theorem 2.7.3. *If n is composite, the Miller-Rabin test incorrectly classifies n as prime with probability at most is at most $\frac{1}{4}$.*

The Miller-Rabin test runs in $\mathcal{O}(\log^2 n)$ time, making it efficient and reliable for large numbers. It is commonly used in practice for cryptographic applications where probabilistic primality testing is acceptable.

2.8 Dictionary problem

A dictionary is a collection of elements, each associated with a unique search key. The goal is to maintain the set efficiently while supporting operations such as insertions and deletions.

Operation	**Description**
$Search(x, S)$	Check if $x \in S$.
$Insert(x, S)$	Insert x into S if it is not already present
$Delete(x, S)$	Remove x from S if it exists
$Minimum(S)$	Return the smallest key in S
$Maximum(S)$	Return the largest key in S
$List(S)$	Output the elements of S in increasing order of keys
$Union(S_1, S_2)$	Merge two sets S_1 and S_2, maintaining the order such that for every $x_1 \in S_1$ and $x_2 \in S_2$, $x_1 < x_2$
$Split(S, x, S_1, S_2)$	Split S into two sets S_1 and S_2, where all elements in S_1 are $\leq x$ and all elements in S_2 are $> x$

The basic strutures complexity for the main operations are the following:

	Search	**Delete**	**Insert**
Unordered array	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$
Ordered array	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Trees	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$

2.8.1 Trees

Binary Search Tree A Binary Search Tree (BST) is a binary tree where each internal node stores an item (k, e) representing a key k and associated element e. The structure of the tree satisfies the property that:

- Keys in the left subtree of any node $v \leq k$.
- Keys in the right subtree of $v > k$.

The drawback of the standard BST is that an unbalanced sequence of insertions may degrade it into a linear structure, resulting in poor performance for searches, inserts, and deletes.

AVL An AVL tree is a self-balancing BST where the heights of the two child subtrees of any node differ by at most one. Rotations ensure that the height of the tree remains logarithmic. While AVL trees guarantee fast lookups and updates, they can be more complex to implement due to the need for maintaining balance factors.

Splay Splay trees are another type of self-adjusting BST. The key idea is the splay operation, which moves a node accessed via a search or update to the root through rotations. This ensures that frequently accessed nodes stay near the root, while infrequently accessed nodes do not contribute much to the overall cost.

2.8.2 Treaps

Treaps are a type of randomized binary search tree that provide efficient time bounds for operations while requiring minimal balance maintenance. Unlike traditional balanced search trees, treaps rely on randomized priorities to achieve a balanced structure, leading to simplicity and efficiency in implementation. The structure of a treap resembles the one obtained if elements were inserted in the order of randomly assigned priorities.

Definition (*Treap*)**.** A is a binary search tree where each node contains an element x with a unique key $\text{key}(x) \in U$ and an associated random priority $\text{prio}(x) \in \mathbb{R}$.

Property 2.8.1 (Search tree)**.** For any node x, all elements y in the left subtree satisfy $\text{key}(y) < \text{key}(x)$, and all elements y in the right subtree satisfy $\text{key}(y) > \text{key}(x)$.

Property 2.8.2 (Heap)**.** For any pair of nodes x and y, if y is a child of x, then $\text{prio}(y) > \text{prio}(x)$.

Lemma 2.8.1. *Given n elements with keys $key(x_i)$ and priorities $prio(x_i)$, there exists a unique treap that satisfies both the search tree property and the heap property.*

Thus, the structure of the treap is entirely determined by the insertion order of elements based on their priorities.

2.8.2.1 Search

Searching in a treap follows the same process as in binary search trees: starting from the root, the search path is determined by comparing the search key with the keys of nodes along the path.

Algorithm 18 Search

```
1: v = root
2: while v ≠ null do
3:     if key(v) = k then
4:         return v                                  ▷ Element found
5:     end if
6:     if key(v) < k then
7:         v = RIGHTCHILD(v)
8:     end if
9:     if key(v) > k then
10:        v = LEFTCHILD(v)
11:    end if
12: end while
13: return null                                      ▷ Element not found
```

The expected time complexity of a search depends on the depth of the path traversed. For a treap with n elements, the expected depth is $\mathcal{O}(\log n)$ due to the randomized priorities.

Definition (*Harmonic number*)**.** The n-th harmonic number is defined as:

$$H_n = \sum_{k=1}^{n} \frac{1}{k} = \ln n + \mathcal{O}(1)$$

Let T be a treap with elements $x_1, \ldots, x_n$, and let x_m be the element we are searching for.

Lemma 2.8.2 (Succesful search)**.** *The expected number of nodes on the path to x_m is given by:*

$$H_m + H_{n-m+1} - 1$$

Let m represent the number of keys smaller than the search key k.

Lemma 2.8.3 (Unsuccesful search)**.** *The expected number of nodes on the path during an unsuccessful search is:*

$$H_m + H_{n-m}$$

2.8.2.2 Insertion and deletion

Insertion and deletion operations in treaps involve rotating nodes to maintain the heap property.

Algorithm 19 Insert

```
1: Choose prio(x)
2: Search for the position of x in the tree
3: Insert x as a leaf
4: while prio(parent(x)) > prio(x) do          ▷ Restore the heap property
5:     if x is left child then
6:         RotateRight(parent(x))
7:     else
8:         RotateLeft(parent(x))
9:     end if
10: end while
```

Algorithm 20 Delete

```
1: Find x in the tree
2: while x is not a leaf do
3:     u = child with smaller priority
4:     if u is left child then
5:         RotateRight(x)
6:     else
7:         RotateLeft(x)
8:     end if
9: end while
10: Delete x
```

These operations maintain both the search tree property and heap property.

Lemma 2.8.4. *The expected running time of the insert and delete operations is $\mathcal{O}(\log n)$, with an expected 2 rotations per operation.*

2.8.2.3 Split and union

Split To split treap T by key k:

1. Insert a new element x with $\text{key}(x) = k$ and $\text{prio}(x) = -\infty$.
2. Insert x into T.
3. Delete x; the left and right subtrees of x become T_1 and T_2, respectively.

Union To merge two treaps T_1 and T_2:

1. Select a key k such that $\text{key}(x_1) < k < \text{key}(x_2)$ for all $x_1 \in T_1$ and $x_2 \in T_2$.
2. Create a new node x with $\text{key}(x) = k$ and $\text{prio}(x) = -\infty$.
3. Set T_1 and T_2 as the left and right subtrees of x, respectively.
4. Delete x from the resulting tree.

Lemma 2.8.5. *The expected time complexity of both union and split operations is* $\mathcal{O}(\log n)$.

2.8.2.4 Implementation

In treaps, priorities are random values drawn from $[0, 1)$, ensuring that tree balancing remains probabilistic rather than explicit. If two nodes have equal priorities, tie-breaking is achieved by appending uniformly random bits to the priorities until a difference is found. This preserves randomness and ensures that the heap property is maintained.

2.8.3 Skip lists

Skip lists, introduced by William Pugh in 1989, are a randomized, dynamic data structure that maintains a sorted set of elements with efficient average-case operations for search, insertion, and deletion. They offer a probabilistic time complexity of $\mathcal{O}(\log n)$ for these operations, making them simple to implement yet powerful for dynamic sets where performance is expected rather than strictly guaranteed.

Skip lists improve upon the basic sorted linked list by adding additional linked lists layered above the main list. The main list connects all elements, like a standard linked list, while each higher level connects increasingly sparse subsets of elements, allowing faster traversal by skipping over parts of the lower lists.

2.8.3.1 Search

To search for an element in a skip list:

1. Begin at the highest level list.
2. Traverse each level by moving right until the target is either found or overshot.
3. If overshot, drop down to the next level and repeat the process.
4. Continue this process until reaching the bottom level, where the target element is either located or confirmed absent.

The higher levels of the skip list serve as express lanes, allowing large jumps, while the lower levels provide finer granularity in search.

Analysis With two levels in the skip list, the search cost is approximately:

$$T(n) = |L_1| + \frac{|L_2|}{|L_1|}$$

This is minimized when:

$$T(n) = |L_1|^2 = |L_2| = n \implies |L_1| = \sqrt{n}$$

Resulting in $T(n) = 2\sqrt{n}$. Generalizing this to k levels gives a cost of $k\sqrt[k]{n}$ With $\log n$ levels, the cost becomes:

$$T(n) = \mathcal{O}(\log n)$$

This efficient layout mimics a balanced binary tree, enabling skip lists to support rapid searching in practice.

2.8.3.2 Insertion

To insert a new element x:

1. Search for x's position in the bottom list.
2. Insert x into the bottom list, which holds all elements in sorted order.
3. Randomly promote x to higher levels based on coin flips. For each level, x is promoted with a probability of $\frac{1}{2}$, ensuring that, on average, only a small fraction of elements reach the top levels. This randomized promotion keeps the structure balanced with $\log n$ expected levels.

The insertion process results in a skip list with a logarithmic number of levels, where the promotion of elements ensures balance across the structure.

2.8.3.3 Implementation

Skip lists are widely used in practice due to their efficiency, with search operations typically taking average time $\mathcal{O}(\log n)$.

Theorem 2.8.6. *With high probability, the search time for an n-element skip list is* $\mathcal{O}(\log n)$.

Here, the phrase with high probability signifies that the probability of this time complexity holding is at least $1 - \mathcal{O}\left(\frac{1}{n^\alpha}\right)$ for a chosen constant $\alpha \geq 1$. By increasing α, the likelihood of search times exceeding $\mathcal{O}(\log n)$ can be made arbitrarily low, making this bound practically reliable.

Lemma 2.8.7. *With high probability, an n-element skip list has* $\mathcal{O}(\log n)$ *levels.*

CHAPTER 3

Amortized analysis

3.1 Introduction

Amortized analysis is a technique used to assess the average cost per operation over a sequence of operations, ensuring that the overall performance remains efficient even if certain individual operations are more expensive. Unlike probabilistic analysis, which relies on random events, amortized analysis provides a guaranteed bound on the average cost per operation, even in the worst-case scenario.

There are three primary methods of amortized analysis:

- *Aggregate method*: this method calculates a simple average cost over all operations, providing an overall estimate, though it lacks the precision of more nuanced approaches.
- *Accounting method*: this approach employs a "banking" system, allocating an amortized cost to each operation to ensure that the total cost is distributed appropriately over a sequence of operations.
- *Potential method*: this method introduces a potential function to track the stored energy of a system, which helps to manage and distribute the amortized costs dynamically.

Hash table resizing A well-designed hash table strikes a balance between compactness and size to minimize overflow and maintain efficient access. However, determining an optimal size for the table upfront is often impractical, as it may not accurately reflect the growth of stored entries. To address this issue, dynamic resizing is employed. When the hash table reaches its capacity, a larger table is allocated, all existing entries are rehashed into the new table, and the memory from the old table is freed. This dynamic resizing mechanism ensures that the hash table adapts to changes in size without sacrificing efficiency, avoiding the need for a predefined size.

3.2 Aggregate method

The aggregate method of amortized analysis calculates the average cost per operation over a sequence of operations, even when some individual operations may be expensive. This technique

is especially useful for data structures that involve periodic costly operations, as it spreads the cost of these infrequent expensive operations across many cheaper ones.

Hash table resizing While a single insertion during a resizing operation might appear costly, with a time complexity of $\mathcal{O}(n)$, this does not imply that n insertions will collectively result in a cost of $\mathcal{O}(n^2)$. In practice, the total cost for n insertions remains closer to $\mathcal{O}(n)$, ensuring better overall efficiency.

To illustrate this, let the cost of the i-th insertion be denoted as c_i:

$$c_i = \begin{cases} i & \text{if } i-1 \text{ is an exact power of 2} \\ 1 & \text{otherwise} \end{cases}$$

In this scheme, when $i-1$ is an exact power of 2, the hash table doubles in size, and all existing entries must be rehashed and inserted into the new table, resulting in a higher cost. For all other insertions, the cost remains constant at 1.

Despite the occasional high cost of resizing, the total cost for n insertions over time is $\mathcal{O}(n)$. This means that, on average, the cost of each insertion is:

$$T(n) = \frac{\mathcal{O}(n)}{n} = \mathcal{O}(1)$$

3.3 Accounting method

In the accounting method of amortized analysis, each operation is assigned a fictitious amortized cost, denoted as $\hat{c}_i$, which represents an accounting balance for the operation. This balance can either be used immediately or saved to cover the cost of future operations. The amortized cost consists of two main components:

- *Immediate cost*: the actual cost incurred for performing the operation.
- *Banked cost*: any excess cost that is set aside and saved for future operations.

The core idea of the accounting method is that the total accumulated banked cost should never be negative, ensuring that there are sufficient funds to cover the costs of future operations. Mathematically, this can be expressed as:

$$\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i$$

Here, c_i represents the true cost of the i-th operation, and $\hat{c}_i$ is the amortized cost. By ensuring that the bank balance remains non-negative, the amortized cost provides an upper bound on the true total cost of all operations, thereby guaranteeing efficient performance.

Hash table resizing For a dynamic hash table that adjusts its size as needed, the accounting method can be applied to model the costs associated with insertions and table expansions. In this case, each insertion is assigned an amortized cost of $\hat{c}_i = 3$:

- *Immediate cost*: the immediate cost of inserting an element into the expanded table is one unit.

- *Banked cost*: two units are banked with each insertion, to cover the cost of future expansions.

When the table doubles in size, the banked cost ensures that the expansion remains efficient. Specifically, half of the banked units are used to insert the existing elements into the new table, while the remaining units help move the elements that were added after the last expansion. This strategy ensures that the bank balance never goes negative, allowing the amortized cost to provide an upper bound on the true cost of all operations.

By charging each insertion an amortized cost of three, and using the banked cost to effectively cover the resizing expenses, the dynamic hash table remains efficient. The amortized cost guarantees that the average cost per operation remains constant over time, even during resizing operations, thus ensuring sustained performance.

3.4 Potential method

The potential method of amortized analysis conceptualizes the "bank account" as the potential energy of a dynamic sequence of operations. The aim is to use a potential function to account for the work done by each operation and how it impacts the overall cost over time.

In this framework, we begin with an initial data structure, denoted as D_0. Each operation i transitions the data structure from D_{i-1} to D_i, incurring a cost c_i. To analyze the costs using the potential method, we define a potential function Φ that maps each data structure state D_i to a real number ($\Phi : \{D_i\} \to \mathbb{R}$). The potential function must satisfy the following properties:

$$\Phi(D_0) = 0 \qquad \Phi(D_i) \geq 0$$

The amortized cost $\hat{c}_i$ for an operation i is then defined as:

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = c_i + \Delta\Phi_i$$

Here, $\Delta\Phi_i$ is the potential difference between the states before and after the operation.

The potential difference $\Delta\Phi_i$ can be either positive or negative:

- If $\Delta\Phi_i > 0$, then $\hat{c}_i > c_i$, meaning the operation deposits work into the data structure to be used by future operations.
- If $\Delta\Phi_i < 0$, then $\hat{c}_i < c_i$, meaning the data structure delivers stored work to help cover the current operation's cost.

The total amortized cost over n operations is:

$$\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i$$

This inequality ensures that the total amortized cost provides an upper bound on the true total cost of the operations.

Hash table resizing To apply the potential method to the dynamic resizing of a hash table, we define the potential of the table after the i-th insertion as:

$$\Phi(D_i) = 2i - 2^{\lceil \log i \rceil}$$

Note that we assume $2^{\lceil \log 0 \rceil} = 0$, which accounts for the table's growth during resizing.

The amortized cost of the i-th insertion is:

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = c_i + (2i - 2^{\lceil \log i \rceil}) - (2(i-1) - 2^{\lceil \log (i-1) \rceil})$$

The true cost c_i of the i-th insertion is:

$$c_i = \begin{cases} i & \text{if } i-1 \text{ is an exact power of 2} \\ 1 & \text{otherwise} \end{cases}$$

When $i - 1$ is an exact power of 2 (i.e., the table needs to be resized), the amortized cost becomes:

$$\hat{c}_i = i + 2 - 2i + 2 + i - 1 = 3$$

In the case where $i - 1$ is not a power of 2, the amortized cost remains:

$$\hat{c}_i \approx 3$$

Thus, in both cases, the amortized cost per insertion is three. Consequently, after n insertions, the total cost is $\Theta(n)$ in the worst case, ensuring that the dynamic hash table remains efficient even with frequent resizing operations.

CHAPTER 4

Parallel programming model

4.1 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.

4.2 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 RAM processors without memory.
- 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.

4.2.1 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.

4.2.2 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.

4.3 Performance

The main values used to evaluate the performance are:

Parameter	**Description**
$T^*(n)$	Time to solve a problem of input size n on one processor using best sequential algorithm
$T_1(n)$	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
$\mathrm{SU}_p = \frac{T^*(n)}{T_p(n)}$	Speedup on p processors
$E_p = \frac{T_1}{pT_p(n)}$	Efficiency
$C(n) = pT_p(n)$	Cost
$W(n)$	Work (total number of operations)

4.3.1 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 21 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
Complexity	$\mathcal{O}(n^2)$	$\mathcal{O}\left(\frac{n^2}{p}\right)$

4.3.2 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 22 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
Complexity	$\mathcal{O}(n)$	$\mathcal{O}\left(\frac{n}{p} + \log p\right)$

4.3.3 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 23 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
Complexity	$\mathcal{O}(n^3)$	$\mathcal{O}\left(\frac{n^3 \log n}{p}\right)$

4.4 Prefix sum

Given a sequence of values $\{a_1, \ldots, a_n\}$, the prefix sum S_i up to position i is defined as:

$$S_i = \sum_{j=1}^{i} a_j$$

In the case of prefix sums, the total computational work required by a parallel algorithm exceeds that of a serial algorithm.

For a serial algorithm, computing each prefix sum is straightforward: each element in the prefix sum can be computed in sequence, where S_i simply depends on S_{i-1} and a_i. This approach only requires $\mathcal{O}(n)$ operations, with each element added once.

In contrast, a parallel algorithm introduces additional overhead. To achieve parallelism, the algorithm needs to divide the work among processors, requiring intermediate calculations and combining steps. Thus, the parallel prefix sum algorithm typically involves $\mathcal{O}(n \log n)$ operations, as it requires multiple rounds to propagate intermediate results across processors.

4.5 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 4.5.1. *Assume $P' < P$, and same size of shared memory. Any problem that can be solved for a P-processor PRAM in T steps can be solved in a P' processor PRAM in $T' = O(T\frac{P}{P'})$ steps*

Lemma 4.5.2. *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 $\mathcal{O}\left(T\frac{M}{M'}\right)$ steps.*

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.

4.5.1 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.

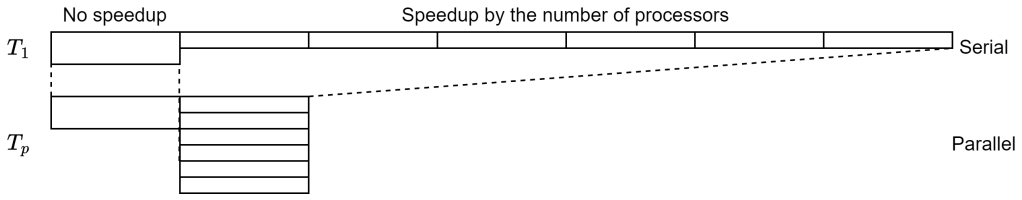


Figure 4.1: Serial and parallel models

When using more than one processor, the speedup is always less than the number of processors. In a program, the parallelizable portion is often represented by a fixed fraction f.

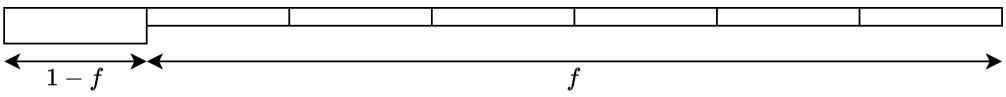


Figure 4.2: Serial model

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

$$\mathrm{SU}(p, f) = \frac{1}{(1-f) + \frac{f}{p}}$$

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.

4.5.2 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:

$$\mathrm{SU}(p) = s + p(1-s)$$

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.

CHAPTER 5

Parallel programming design

5.1 Introduction

Historically, both developers and users have approached problem-solving with a sequential mindset. This approach is reflected in the majority of existing algorithms, which are designed to execute one step at a time in a linear fashion. However, modern hardware architectures offer significant opportunities for parallelism, allowing for the simultaneous execution of multiple instructions or tasks.

Advantage	Description
Time efficiency	Parallel algorithms can complete tasks faster
Cost efficiency	Parallel architectures use multiple inexpensive components
Complex problems	Parallel algorithms solve some complex problems efficiently

Moore's law According to Moore's Law, the number of transistors on a chip doubles approximately every 24 months, but single cores can no longer fully utilize the additional transistors. Moreover, continually increasing processor frequency has become impractical due to rising power consumption and heat dissipation concerns. Consequently, parallelism is increasingly essential to leverage these advances and continue improving computational performance.

5.2 Algorithms parallelization

Automatic parallelization In automatic parallelization, developers write algorithms and implement them with sequential code, relying on automated tools to handle all parallelization. This approach allows developers to work with familiar, sequential algorithms without having to restructure them for parallel execution.

However, fully automated parallelization remains challenging and is currently not entirely feasible. The primary limitation is that these tools struggle to extract all potential parallelism

from code initially designed for sequential execution. Consequently, the performance gains from automatic parallelization are often limited, especially for complex or intricate tasks.

Manual parallelization Manual parallelization requires the programmer to play an active role in the parallelization process. Developers must design parallel algorithms and implement them using high-level parallel programming constructs. By providing targeted information to the tools, the programmer assists in optimizing the parallel execution, leaving only the compilation of code to automated tools. When using manual parallelization, there are three critical aspects to consider:

- *Parallelism type*: determine which type of parallelism is most suitable for the application.
- *Tool communication*: effectively communicate information about the parallelism to tools so that they can optimize performance during compilation.

5.2.1 Taxonomy

Parallelism comes in various forms, each offering different ways to execute instructions and process data simultaneously. The two primary types are instruction parallelism and data parallelism, which can also be combined according to Flynn's taxonomy. This classification, proposed in 1966, helps categorize computer architectures by their parallel processing capabilities.

Architecture	**Instructions**	**Data**	**Example**
SISD	Single	Single	Single-core processors
SIMD	Single	Multiple	GPU
MISD	Multiple	Single	-
MIMD	Multiple	Multiple	Multicore processors

Parallelism classification Parallelism can also be categorized by the level at which it occurs:

1. *Bit-level parallelism*: bits within data words can represent distinct data elements, allowing a single instruction to manipulate multiple data bits at once.
2. *Instruction-level parallelism*: multiple instructions are executed simultaneously on a single core. Compilers can often extract this type of parallelism automatically.
3. *Task-level parallelism*: tasks are discrete units of computational work, typically composed of program-like sets of instructions. Multiple tasks can be executed on multiple processors, supported by shared memory and cache coherence mechanisms, though this parallelism is challenging to automate fully.

   Task-level parallelism can be represented with a parallel task graph in whick tasks are represented by graph vertices, while edges indicate dependencies or data communications. Each task is ideally executed once, following the directed acyclic structure.

   Alternatively, we may use a classic pipeline, in which each processor represents a stage in the pipeline. Edges denote data passing between stages, making this model well-suited for stream processing tasks like audio and video encoding, where each stage processes data continuously.

Communication classification Effective communication between tasks is essential in parallel systems, with two primary models:

- *Shared memory*: all tasks have access to a global memory space, allowing them to read from and write to common memory locations. Any modification is visible across all processors.
- *Private memory*: each task has its own private memory, and communication is achieved through explicit message exchanges. This model offers high modularity and isolation between tasks, reducing shared memory conflicts.

5.3 Parallelism design

Designing an effective parallel algorithm involves more than simply extracting all available parallelism. Not all parallelism is usable on a given architecture, as certain types may introduce overhead or be limited by the hardware. To create a practical and efficient parallel solution, designers must consider the architecture's specific parallel capabilities and communicate this parallelism to the compilation tools effectively.

New languages developed specifically for parallel programming offer advanced features for expressing parallelism, but their adoption has been limited: since they tend to have immature compilers, and developers must invest time in learning new syntax and semantics. In contrast, extensions to established languages are easier to adopt and integrate with existing compilers but have limitations. These extensions typically only support certain forms of parallelism, making it challenging to represent more complex structures like pipeline parallelism.

To design an algorithm, we usually understand the problem to be solved, analyze the dependencies, partition the solution, and use the PCAM methodology to define other aspects.

5.3.1 PCAM mehod

The Partitioning-Communication-Agglomeration-Mapping (PCAM) methodology is a structured approach to designing efficient parallel algorithms, consisting of four interrelated phases:

1. *Partitioning*: the goal of partitioning is to reveal parallelizable components within the problem, dividing it into many small, independent tasks to create a fine-grained decomposition. Effective partitioning divides both the computational tasks and associated data without unnecessary replication.

    - *Functional partitioning*: focuses on decomposing the problem based on the actions or computations required. Tasks are defined by the different functions or operations needed to solve the problem.
    - *Domain partitioning*: divides the problem based on data, where each task processes a subset of the data. This method is often preferred for problems that naturally separate into data chunks; ideally, tasks should have equal data sizes to balance workloads.

    A combination of functional and domain partitioning may sometimes be effective to optimize parallelization.

2. *Communication*: once partitioned, tasks must interact to exchange necessary information. This phase addresses the nature and management of communication between tasks, aiming to minimize overhead and maximize efficiency. The communication can be classified as:

    - *Local or global*: limited to a small set of neighboring tasks or involve many tasks simultaneously.
    - *Structured or unstructured*: follow regular patterns or arbitrary, dynamic connections.
    - *Static or dynamic*: communication partners may be fixed in advance or determined at runtime.
    - *Synchronous or asynchronous*: tasks may synchronize their communication or communicate independently.
    - *Point-to-point or collective*: direct data transfer between two tasks or between multiple tasks within a group.

3. *Agglomeration*: this phase transitions from theoretical task partitioning to a more practical implementation by grouping tasks into larger units to improve efficiency on the target parallel architecture:

    - *Task consolidation*: combines smaller tasks into larger ones to increase computational granularity and reduce communication needs.
    - *Data and computation replication*: may replicate data or computation selectively to reduce dependency on frequent communication.
    - *Load balancing and overlapping*: ensures an even distribution of workload, overlapping communication with computation when possible.

    Agglomeration often changes key performance ratios, increasing task size to reduce communication demands, thereby enhancing efficiency but potentially decreasing parallelism. This phase must balance the reduction of communication overhead with maintaining adequate parallelism.

4. *Mapping*: mapping assigns tasks to specific processors, with the objective of minimizing total execution time. Although straightforward in shared-memory systems, mapping in distributed systems requires careful consideration of locality (place frequently communicating tasks on the same or neighboring processors to minimize communication time) and concurrency (assign concurrent tasks to separate processors to maximize parallel execution). Mapping is an $\mathcal{NP}$-complete problem, so heuristic algorithms are used to approximate optimal solutions. Mapping can be complex due to resource limitations and the potential for conflicting goals between maximizing concurrency and minimizing communication costs.

5.4 Parallelization evaluation

Evaluating parallel algorithms requires an analysis of both time complexity, which quantifies the duration needed to complete a solution, and resource complexity, which measures the computational resources, such as processors and memory, required to achieve that time. These metrics provide essential insights into the scalability and efficiency of a parallel algorithm.

To analyze a parallel algorithm, we often represent its structure as a Directed Acyclic Graph. In this framework, nodes represent tasks, while edges indicate dependencies among tasks. Tasks that can be executed independently are concurrent, while parallel tasks are tasks that are executed simultaneously due to the availability of multiple processing resources.

5.4.1 Algorithms evaluation

Work (W) is a critical metric, representing the total number of operations performed by the algorithm. Unlike a sequential algorithm, a parallel algorithm's work may be higher due to communication overhead and other parallelization costs. The span (S), on the other hand, represents the longest dependency path, or the critical path, which defines the minimum time to complete the algorithm even if unlimited processors were available. Together, these metrics allow us to derive parallelism (P), calculated as $P = \frac{W}{S}$, which measures the algorithm's efficiency in utilizing resources.

Operation	**Work**	**Span**
Single operation	$W(\text{op}) = 1$	$S(\text{op}) = 1$
Sequential tasks (e_1, e_2)	$W(e_1, e_2) = W(e_1) + W(e_2)$	$S(e_1, e_2) = S(e_1) + S(e_2)$
Parallel tasks $(e_1 \parallel e_2)$	$W(e_1 \parallel e_2) = W(e_1) + W(e_2)$	$S(e_1 \parallel e_2) = \max(S(e_1), S(e_2))$

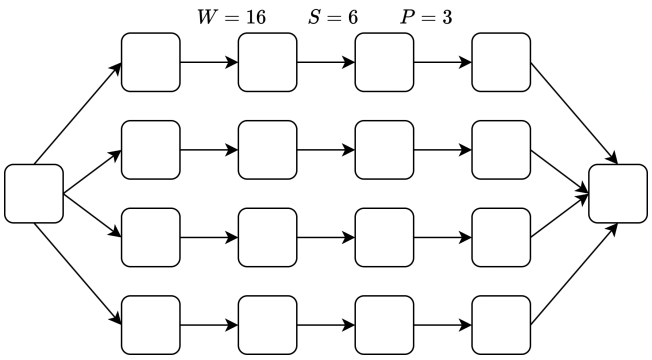


Figure 5.1: Parallel algorithm evaluation

Effective parallel algorithm design balances the goal of minimizing work to reduce resource usage with the need to maximize parallelism by reducing the span. However, optimizing for parallelism may introduce additional communication and synchronization overhead, making it necessary to find a balance that minimizes both work and span as much as possible. This trade-off between work and parallelism requires careful consideration, as adding more parallelism can increase the complexity and overhead.

CHAPTER 6

Parallel programming frameworks

6.1 POSIX Threads

Threads A thread is an independent unit of execution within a process, capable of running concurrently with other threads. Each thread has its own local data, but can access the resources shared by the parent process. Unlike a full process, which includes information about resources and execution state, a thread is more lightweight. A single process can spawn multiple threads, each of which operates as an independent stream of instructions.

Threads are managed by the operating system, which schedules them for execution. They can run in parallel, allowing for more efficient use of resources.

In this sense, threads provide a form of implicit communication, as they can read and write shared variables. However, because multiple threads can access the same memory location, this requires explicit synchronization to prevent conflicts. Without proper synchronization, concurrent threads may cause unpredictable behavior, particularly when they attempt to modify the same data simultaneously.

Threads can be created dynamically during execution, and are generally more efficient than creating multiple processes since they share resources, avoiding the overhead associated with full process management. Since threads operate independently but share the same memory space, synchronization mechanisms such as locks or semaphores must be used to prevent conflicting operations.

Race condition A race condition occurs when two or more threads access the same variable concurrently, and at least one of them performs a write operation. Since these accesses are not synchronized, there is a risk that the threads may interfere with each other, leading to inconsistent or incorrect results. To prevent race conditions, synchronization mechanisms are required, which ensure that only one thread can modify the shared resource at a time. This can be achieved through various methods, such as using mutexes, locks, or atomic operations. The programmer is responsible for managing the synchronization of threads, often using libraries, compiler directives, or other tools designed to handle parallelism and ensure the integrity of shared data.

PThreads POSIX Threads provide a standardized API for managing threads in multi-threaded programming, enabling developers to efficiently execute tasks concurrently.

Threads are peers, meaning that once created, they operate independently without any inherent hierarchy. Threads can also create additional threads without any dependencies or constraints. The maximum number of threads is determined by the system's implementation.

Creation Threads are created using the `pthread_create` function:

```
int pthread_create()
```

This function initializes a new thread that begins execution with the specified start routine.

Termination A thread terminates naturally when its start routine returns or the process in which it resides exits. Threads can also be explicitly terminated using the following functions:

```
// Exit the calling thread
int pthread_exit()
// Exit a specific thread
int pthread_cancel()
```

Joining To wait for a thread to finish execution, the `pthread_join` function is used:

```
int pthread_join()
```

When a thread is joined, the calling thread is blocked until the specified thread terminates. Threads can be created as joinable (default) or detached. Joinable threads must be joined to release their resources, while detached threads release their resources automatically upon termination.

6.1.1 Synchronization mechanisms

Synchronization is essential in multi-threaded programming to coordinate the execution of threads and protect shared resources.

Barriers Barriers synchronize multiple threads at a specific point in their execution. All threads in a group must reach the barrier before any can proceed:

```
// Initialize a barrier
int pthread_barrier_init()
// Wait at a barrier
int pthread_barrier_wait()
```

Mutex Mutexes (mutual exclusion locks) ensure that only one thread accesses a critical section at a time.

```
pthread_mutex_lock(&my_lock);
/* critical section */
pthread_mutex_unlock(&my_lock);
```

If a thread attempts to lock a mutex that is already locked, it will block until the mutex becomes available. Alternatively, `pthread_mutex_trylock` can be used to attempt locking without blocking.

Condition variables Condition variables allow threads to wait for specific conditions to be met. They are typically used in conjunction with a mutex to signal state changes between threads:

```
pthread_mutex_lock(&my_mutex);
while (!condition_met) pthread_cond_wait(&my_cond, &my_mutex);
/* condition is now met */
pthread_mutex_unlock(&my_mutex);

// Signal or broadcast to waiting threads
pthread_cond_signal(&my_cond);
pthread_cond_broadcast(&my_cond);
```

Condition variables provide a powerful mechanism for coordinating thread activities based on dynamic conditions.

6.2 Open Multi Processing

OpenMP (Open Multi-Processing) is an application programming interface (API) designed for parallel programming on shared memory systems. It allows developers to efficiently utilize multiple processors or cores for concurrent execution, making it a powerful tool for parallelizing computationally intensive tasks. OpenMP provides compiler directives, library routines, and environment variables to manage multi-threading in C, C++, and Fortran programs. The key features of OpenMP are:

- *Compiler support*: OpenMP requires support from the compiler.
- *Portability*: OpenMP is a standardized approach for parallel programming and is portable across various shared memory architectures.
- *Ease of use*: OpenMP is known for its simplicity and ease of adoption. It provides a minimal set of directives that are sufficient to implement significant parallelism.
- *Scalability*: OpenMP supports both coarse-grained and fine-grained parallelism, allowing it to scale from small systems to large multi-core processors.

Programming model OpenMP is based on the fork-join parallel model:

1. The master thread (the main program thread) begins execution.
2. It forks a specified number of slave threads to execute tasks in parallel.
3. The slave threads execute tasks concurrently, with the runtime environment managing the allocation of threads to different processors.
4. Once all tasks are completed, the threads join back to the master thread, which continues execution.

Directives The core component of OpenMP is its use of pragmas, which are preprocessor directives that provide the compiler with instructions about parallel execution. Pragmas are used to define parallel tasks, control synchronization, and manage the execution flow. The basic syntax for an OpenMP directive is:

```
#pragma omp <name> [list of clauses]
```

6.2.1 Control structures

OpenMP programs typically execute serially until they encounter a parallel directive, which instructs the program to spawn multiple threads. The thread executing the code becomes the master thread (with thread ID 0), and it creates a group of slave threads that execute the same block of code concurrently. Each thread operates on a copy of the code within the parallel block, and once the block is completed, an implicit barrier ensures that all threads synchronize before the master thread continues.

A basic parallel directive in OpenMP looks like this:

```
#pragma omp parallel
{
    /* parallel section */
}
```

Several clauses can be added to control the behavior of the parallel region:

- `if(condition)`: this clause allows for conditional parallelization. The parallel region will only be executed in parallel if the specified condition evaluates to true.
- `num_threads(int)`: this specifies the number of threads to be used in the parallel region, overriding the default behavior.
- *Data scope clauses*: these clauses control the visibility and lifetime of variables within the parallel region, and we'll cover them in more detail in later sections.

Under the hood, the compiler may replace the OpenMP parallel directive with a Pthreads-based implementation for thread management.

6.2.2 Work sharing

Work-sharing constructs in OpenMP allow the division of execution across multiple threads within a parallel region. Unlike parallel regions, work-sharing constructs do not create new threads. Instead, they divide the work already assigned to the threads in the team that is executing the parallel region. Importantly, work-sharing constructs do not imply a barrier when entering, but there is an implicit barrier when exiting.

These constructs are designed to split work across threads in different ways, depending on the nature of the task. They provide mechanisms for both data parallelism and functional parallelism, allowing efficient parallel execution.

For loop The `for` construct is used to divide the iterations of a loop among the threads in the team, which is a form of data parallelism. The loop iterations cannot be modified internally within the loop.

```
#pragma omp parallel
{
    #pragma omp for
    /* for loop */
}
```

The possible clauses are:

- `schedule`: specifies how iterations are assigned to threads.
- `nowait`: prevents synchronization at the end of the loop, allowing threads to proceed without waiting for others to complete.
- *Data-scope clauses*: manage variable scope within the parallel loop.
- `reduction`: aggregates results across iterations. A private copy of the variable is maintained by each thread, and at the end of the loop, the reduction operation is applied across all private copies.

The scheduling types are:

- `static`: divides loop iterations into equal-sized blocks (specified by chunk), which are statically assigned to threads. If no chunk is specified, iterations are evenly distributed.
- `dynamic`: divides loop iterations into chunks of size chunk, distributed at runtime. Once a thread completes a chunk, it is assigned another.
- `runtime`: the schedule is determined at runtime based on the `OMP_SCHEDULE` environment variable.
- `guided`: starts with large chunks and gradually decreases their size.

The trade-off in scheduling lies between load balancing (dynamic scheduling with small chunks) and low overhead (static scheduling with large chunks).

Sections The `sections` construct allows different sections of code to be executed in parallel, each by a different thread. This is an example of functional parallelism where the work is divided into discrete, independent sections.

```
#pragma omp parallel
{
    #pragma omp sections
    {
        #pragma omp section
        {
            /* code section 1 */
        }
        #pragma omp section
        {
            /* code section 2 */
        }
    }
}
```

Each section is executed exactly once by a thread, and the sections are executed concurrently.

The directive `single` ensures that a section of code is executed by only one thread, typically the master thread or a specific thread designated by the OpenMP runtime. The directive `master` ensures that a section of code is executed only by the master thread, typically the one with thread ID 0. This can be useful for tasks that should only be done once or for thread-specific operations.

6.2.3 Synchronization

Synchronization constructs in OpenMP are used to coordinate the execution of multiple threads, ensuring that certain sections of code are executed in a controlled manner. These constructs are critical when dealing with shared resources, preventing race conditions, and maintaining consistency in parallel programs.

Critical directive The `critical` directive ensures that a section of code is executed by only one thread at a time, preventing concurrent access to shared resources. This is essential for protecting critical sections where race conditions could occur.

```
#pragma omp critical [name]
{
    /* code section */
}
```

The name allows you to define multiple critical sections. If different critical sections share the same name, they are treated as the same critical region, ensuring that only one thread can access them at a time. If no name is provided, all unnamed critical sections are treated as a single region.

Barrier directive The `barrier` directive synchronizes all threads within the team. When a thread reaches a barrier, it will wait until all other threads have reached the same point, after which all threads can resume executing the code that follows the barrier.

```
#pragma omp barrier
```

This construct is less commonly used because many other OpenMP constructs implicitly synchronize threads, making the explicit use of barriers less frequent.

Atomic directive The `atomic` directive ensures that a specific memory location is accessed atomically. This guarantees that no other thread can modify the memory location during a read-modify-write operation, thus avoiding race conditions.

```
#pragma omp atomic
    /* statement */
```

The `atomic` directive can only be used for individual statements, not for entire code blocks. It ensures that the operation on a specific memory location is completed without interruption, supporting safe updates in multi-threaded environments. Only one atomic operation can be applied to a specific memory location at a time. Multiple reads and writes are not allowed within the atomic region.

6.2.4 Data environment

OpenMP is based on the shared-memory programming model, where most variables are shared by default among threads. However, in parallel computing, explicit control over how variables are scoped is crucial for managing data consistency and performance. The OpenMP Data Scope Attribute Clauses provide the necessary control over how variables are handled in parallel regions. These clauses include `private`, `shared`, `default`, and `reduction`, which allow developers to define the visibility and allocation of variables across threads.

Private clause The `private` clause ensures that each thread gets its own instance of a variable. A new object of the same type is created for each thread, and any reference to the original variable is replaced with a reference to the private instance.

```
#pragma omp <name> private(list)
```

Each thread gets a unique copy of the variable, initialized independently of other threads. Useful when a variable's value should not be shared between threads and should be initialized separately for each.

Shared clause The `shared` clause declares variables that are shared among all threads in a parallel region. There is only one memory location for a shared variable, and all threads can read and write to this location. The programmer must ensure that the variable is accessed safely to avoid race conditions.

```
#pragma omp <name> shared (list)
```

All threads access the same memory location for the variable. Ideal for variables that need to be updated or read by all threads, such as counters or buffers.

Default clause The `default` clause sets the default data scope for variables that are not explicitly scoped by `private` or `shared`. By default, variables are `shared`,

```
#pragma omp <name> default(shared | none)
```

Controls the default scoping of variables in the parallel region. Helps enforce stricter control over variable scoping, reducing errors in large parallel applications.

Reduction clause The `reduction` clause is used to perform a reduction operation on a variable across threads. Each thread gets a private copy of the variable, and at the end of the parallel region, these private copies are combined using a specified operator.

```
#pragma omp <name> reduction(operator: list)
```

Allows threads to operate on private copies of variables, and at the end of the region, a reduction operator is applied to aggregate the results. Useful for operations like summing up values or finding the minimum or maximum in parallel loops.

6.2.5 Memory model

In OpenMP, threads share a common memory space, but each thread also has its own private memory. The interaction between private and shared memory, and how updates to shared

variables are synchronized across threads, is governed by the memory consistency model. This model ensures that threads have a coherent view of memory, especially when accessing or modifying shared variables. The key elements of OpenMP memory model are:

- *Memory access*: each thread has access to private memory, which is used for data specific to the thread. Threads can also access shared memory, a common address space, where variables can be read or modified by multiple threads.
- *Relaxed memory consistency*: OpenMP uses a relaxed memory consistency model where threads have their own temporary view of memory between consistency points. A consistency point is a point in the program where memory views of all threads are synchronized, ensuring that all threads have the same view of shared memory at those moments. In between consistency points, each thread may have its own version of the shared memory, which can lead to data races if not carefully controlled.
- *Shared data modifications*: if shared data is modified by multiple threads, the potential for data races arises. A data race occurs when multiple threads modify the same shared variable without proper synchronization, leading to unpredictable behavior.

To manage memory consistency explicitly, OpenMP provides the `flush` directive, which enforces a global consistency of shared variables.

```
#pragma omp flush [flush-set]
```

The `flush` directive ensures that all threads have a consistent view of shared variables at a particular point in the program. Between a `flush` and the next update of shared variables, all threads are guaranteed to have the same global view of shared memory. If the `flush-set` is not specified, all shared variables are affected by the flush operation. This is generally best avoided if possible, as specifying the set of variables to flush can be more efficient. The compiler is allowed to move `flush` operations if they involve disjoint sets of variables, which can improve performance but can also lead to incorrect behavior if used incorrectly.

Certain OpenMP constructs implicitly enforce memory consistency:

- At barrier regions: All threads synchronize at the barrier, ensuring consistency.
- At entry and exit of parallel and critical regions: These constructs synchronize threads and provide consistency points.
- At exit from work-sharing constructs, unless the nowait clause is used: this ensures synchronization among threads at the end of a work-sharing region.

While certain synchronization points implicitly enforce consistency, in some cases, the programmer must explicitly use the flush directive to guarantee a consistent memory view at entry to work-sharing constructs and master regions, a flush is not implied and may need to be used manually.

6.2.6 Runtime functions

OpenMP standard provides a set of runtime functions that allow you to control and gather information about the execution of a parallel program. These functions are part of the OpenMP API and enable developers to dynamically manage the parallel execution environment:

```
// Returns the number of threads currently executing in the parallel region from
    ↪ which the function is called.
int omp_get_num_threads()

// Returns the unique identifier (thread ID) of the calling thread.
int omp_get_thread_num()

// Sets the number of threads that will be used in the next parallel region.
double omp_get_wtime()

// Provides the wall clock time in seconds since some arbitrary point in the past.
double omp_get_wtick()
```

Environment variables In addition to runtime functions, OpenMP allows you to control the execution of parallel programs using environment variables. These variables influence the behavior of the OpenMP runtime system and can be set at runtime.

6.2.7 Nested parallelism

OpenMP supports nested parallelism, where parallel regions can be nested within other parallel regions. This allows for more flexible and efficient parallel execution, particularly in complex workloads.

Nested parallelism can be enabled or disabled using the following methods:

- *Using the runtime function*: you can enable or disable nested parallelism through the `omp_set_nested` function.
- *Using the environment variable*: you can also control nested parallelism through the environment variable `OMP_NESTED`.

The default behavior for nested parallelism is implementation-dependent, and different OpenMP compilers may have varying defaults. In some systems, nested parallelism may be disabled by default. To control the number of threads in nested parallel regions, you can use the following environment variables:

- `OMP_NUM_THREADS=<list of integers>`: sets the default number of threads for different levels of nested parallelism. Each entry in the list corresponds to a level of nesting. If the nesting level exceeds the number of entries, the last value in the list is used for all subsequent levels.
- `OMP_MAX_ACTIVE_LEVELS`: specifies the maximum number of active parallel regions that can be nested.
- `OMP_THREAD_LIMIT`: sets a limit on the total number of threads that can be created, which can help prevent excessive thread creation in recursive applications.

In nested parallelism, each nested parallel region has its own set of threads, and thread IDs are reset to 0 for each new nested team. Therefore, the global thread IDs from `omp_get_thread_num` are not sufficient to uniquely identify threads across nested regions.

To manage nested parallelism more effectively, OpenMP provides several runtime functions:

```
// Returns the maximum number of threads that can be used in the current parallel
    ↪ region
int omp_get_thread_limit()

// Returns the maximum number of active parallel regions in the current nested
    ↪ parallel execution
int omp_get_max_active_levels()

// Sets the maximum number of active parallel regions that can be nested
void omp_set_max_active_levels(int max_levels)

// Returns the current level of nesting within parallel regions. It returns the
    ↪ depth of the current parallel region
int omp_get_level()

// Returns the level of the current active parallel region
int omp_get_active_level()

// Returns the thread ID of the ancestor thread at a specific level of nesting.
// The level argument specifies the ancestor level.
int omp_get_ancestor_thread_num(int level)

// Returns the size of the thread team for a specified nesting level
int omp_get_team_size(int level)
```

6.2.8 Thread cancellation

Prior to OpenMP 4.0, once a parallel region was started, it would run to completion. It was not possible to abort or cancel a parallel region during execution. However, OpenMP 4.0 introduced the ability to cancel parallel execution, providing more flexibility for certain applications like divide-and-conquer algorithms or error handling.

OpenMP's thread cancellation mechanism is designed as a best effort approach. This means that the system does not guarantee an immediate termination of threads, but it provides a way to attempt cancellation under certain conditions.

Cancellation is useful for scenarios where the execution of parallel tasks can be halted early, such as in a search algorithm where processing can stop once a result is found. Similarly, cancellation is useful for handling errors or unexpected situations where continuing execution is no longer desirable.

Cancellation directives The `cancel` directive is used to mark a region of the code where threads can potentially be canceled.

```
#pragma omp cancel <construct-type>
```

The `<construct-type>` specifies which type of construct is to be canceled

A thread can check whether cancellation has been requested by encountering a cancellation point. When a thread reaches this point, it checks for a cancellation flag set by the cancel directive. If cancellation is requested, the thread will attempt to terminate at this point.

```
#pragma omp cancellation point <construct-type>
```

At a cancellation point, the thread checks the cancellation flag. If the flag is set, the thread terminates its execution. Cancellation points can occur: at another cancel region or at a barrier (either implicit or explicit).

6.2.9 Tasks

OpenMP tasks are a powerful construct for parallelizing algorithms that have irregular or runtime-dependent execution flows, where the work cannot be easily divided into independent iterations or chunks. Unlike loops that execute a predictable number of iterations, tasks allow for more dynamic and flexible execution.

An OpenMP task is a block of code within a parallel region that can be executed concurrently with other tasks in the same region. The task is created when encountered in the code but is not necessarily executed immediately or in the order it appears in the source code. Instead, tasks are dynamically assigned to threads by the OpenMP runtime system. Tasks are particularly useful in situations like while loops or any algorithm that requires dynamic decision-making about what work needs to be done next. OpenMP manages the queuing and scheduling of tasks, allowing for better load balancing and better use of resources in cases where the execution flow is not regular.

Task synchronization Once a task is created, it is not executed immediately; instead, it is added to a task queue. Tasks will only be executed when resources are available. However, there are points at which synchronization is required to ensure that tasks complete before proceeding further. OpenMP provides constructs to handle synchronization of tasks:

1. `taskwait`: this directive forces the current thread to wait for all child tasks to complete before it resumes execution. It's particularly useful when the completion of tasks needs to be ensured before continuing.

   ```
   #pragma omp taskwait
   ```

2. `taskgroup`: the `taskgroup` directive synchronizes not only the child tasks but also their descendants. This is useful when there are multiple layers of tasks, and synchronization is required at multiple levels.

   ```
   #pragma omp taskgroup
   ```

Tasks are also guaranteed to complete at synchronization points such as a barrier (explicit or implicit) or task synchronization points.

Task dependencies OpenMP tasks support task dependencies, which help manage complex parallel workflows, particularly when certain tasks need to wait for others to finish before they can start (e.g., in pipeline parallelism). This allows computation to overlap with other activities, such as I/O, and can reduce idle time.

Depend clause The `depend` clause specifies the dependencies between tasks, indicating that one task must finish before another begins. This helps ensure that tasks are executed in the correct order.

```
#pragma omp task depend(in: data) depend(out: result)
```

Priority clause The `priority` clause gives a hint to the runtime system about the importance of a task. Tasks with higher priority are executed first, although this is just a hint and not a strict guarantee. To use the priority feature, the environment variable `OMP_MAX_TASK_PRIORITY` must be set before running the program (with a default value of 0).

```
#pragma omp task priority(5)
```

Task scheduling The scheduling of tasks can introduce overhead. OpenMP tasks provide a mechanism to reduce this overhead by allowing for the scheduling of larger units of work, but at the cost of less flexibility in handling load imbalances. This approach is efficient for cases where the tasks are large enough to offset the scheduling overhead, but it limits flexibility in handling load imbalances.

Task loops OpenMP also provides a convenient way to handle loops using tasks. By using the `taskloop` construct, the runtime system can efficiently divide loop iterations into tasks, reducing the complexity of manually managing task creation for each iteration. When using loops with tasks, OpenMP automatically handles task dependences and scheduling. This allows for flexible parallel execution of loops with a dynamic number of iterations or varying workloads.

6.2.10 SIMD vectorization

SIMD (Single Instruction, Multiple Data) vectorization is a powerful technique used to enable parallelism within a single thread by processing multiple data elements simultaneously with vector instructions. In OpenMP, SIMD directives allow loops to be vectorized, helping to improve performance by taking advantage of CPU vector units.

The simplest way to apply SIMD vectorization is with the `#pragma omp simd` directive:

```
#pragma omp simd
    /* for loop */
```

It instructs the compiler to vectorize a loop and execute all iterations with SIMD instructions, which means multiple loop iterations are processed in parallel by a single thread using vector registers. The loop iterations are divided into chunks, and each chunk is executed by a SIMD lane. The compiler generates the SIMD instructions necessary to execute the loop efficiently, but it's up to the user to ensure that the vectorization maintains correctness.

Several options can be used to fine-tune SIMD performance and ensure optimal execution:

1. *Data scope clauses*: clauses such as `private`, `firstprivate`, and `reduction` can be used with the SIMD directive to control how variables are scoped during the vectorized loop.

2. `collapse` *clause*: used to combine two perfectly nested loops into a single larger loop, which can help improve vectorization. However, this comes with additional complexity, so it should be used carefully.

3. `simdlen(size)` *clause*: the `simdlen` clause suggests a preferred vector length. The compiler is free to ignore this value, but it can help optimize performance in some cases.

4. `safelen` *clause*: the `safelen` clause sets an upper limit on the vector length that the compiler cannot exceed. This is useful when dealing with loop-carried dependencies, as the vector length must be smaller than the smallest dependence distance to avoid incorrect behavior.

Composite contruct The `#pragma omp for simd` directive is a combination of both the SIMD vectorization and work-sharing constructs. It tells the compiler to distribute iterations among threads in a team while also vectorizing the loop. This construct distributes iterations across multiple threads, and within each thread, the loop is vectorized using SIMD instructions. It's important to note that the number of threads and scheduling policy can greatly affect performance. If the number of threads increases, the work per thread becomes smaller, so each thread should ideally work with a chunk corresponding to the vector length.

Scheduling policies To avoid performance degradation, OpenMP allows specifying SIMD scheduling policies in conjunction with the `simd` directive. The `schedule(simd)` clause defines how iterations are distributed among threads while using SIMD instructions for parallel execution.

```
#pragma omp for simd schedule(simd:static, 5)
```

6.2.11 OpenMP for heterogeneous architectures

Recent versions of OpenMP support parallel execution not only on CPUs but also on heterogeneous architectures. This allows a single OpenMP codebase to be executed both on the host CPU and on target devices like GPUs or other accelerators, without requiring changes in the code.

The `target` directive is used to offload computation to a device while keeping the same code that runs on the host. The code inside the target region will be executed on the device if available; otherwise, it will continue to execute on the host.

```
#pragma omp target
{
    // Code region to be offloaded to the target device
}
```

By default, execution is synchronous, meaning that the host thread will block until the device finishes executing the offloaded region. If you want to avoid blocking, you can use the `nowait` clause, which tells the host to not wait for the target device to finish before continuing with other work.

Map clause The `map` clause is crucial when working with offloaded code, as it controls the transfer of data between the host and the target device memory.

```
#pragma omp target map(to: x[0:n]) map(from: y[0:m])
{
    // Offloaded code
}
```

Here:

- `map(to)`: data is copied from the host to the target device before the offloaded code executes.
- `map(from)`: data is copied from the target device back to the host after the offloaded code executes.
- `map(tofrom)`: data is both sent to the device before execution and received from the device after execution.

The map-type option tells the compiler how data should be transferred between the host and device, allowing optimizations to minimize unnecessary copies. The default mapping type is tofrom, meaning the data is both sent to and retrieved from the device unless specified otherwise.

6.3 Message Passing Interface

Shared memory systems are typically easier to program but harder to build. In these systems, all threads have access to the same address space, enabling efficient communication between threads within a single node. However, they face limitations when extending across multiple nodes. The main limitations of shared memory architectures are:

- *Execution across multiple nodes*: shared memory systems struggle to efficiently scale across several nodes in a distributed environment.
- *Communication across nodes*: data transfer between nodes requires specialized communication mechanisms, such as send/receive operations and synchronization, to coordinate and manage the process.

MPI is a standardized framework designed for communication in distributed memory systems, addressing the challenges of multi-node communication in high-performance computing (HPC) environments. It defines a set of interfaces for sending and receiving data between processes on different nodes, focusing on performance optimization in parallel and distributed systems. MPI is widely used in scenarios where performance is critical, especially in parallel computing environments, assuming a predictable setup based on prior experience with hardware configurations.

It is essential to write code that is independent of a specific MPI implementation. Programs should only rely on behaviors defined in the MPI standard, ensuring portability across various systems and implementations. The essential elements are:

- `libmpo.so`: the core library that implements the MPI standard.
- *Compiler wrappers*: wrappers around Fortran, C, and C++ compilers, designed to simplify the build process for MPI programs.
- `mpiexec`: a command-line tool used to launch MPI applications. It can spawn multiple instances of the application, potentially across different nodes, and manage process execution. Additionally, it can coordinate with thread libraries like OpenMP for hybrid parallelization.

To run an MPI application we perform the following steps:

1. *Build the application*: compile the application using the appropriate machine environment and compiler.
2. *Create a bash*: a job script automates the interaction with the resource manager to request resources, sets up the environment used during compilation, and launches the application.
3. *Submit the job script*: submit the job script to the resource manager's job queue for execution on the cluster.

6.3.1 Program

In MPI-based applications, we follow a structured process to initialize, run, and finalize MPI operations:

- *Initializing MPI* (`MPI_Init_thread`): at the beginning of an MPI program, you must initialize MPI and negotiate the thread usage level. MPI was originally designed to work with processes, but modern implementations support multithreading.
- *Finalizing MPI* (`MPI_Finalize`): at the end of the program, MPI must be finalized. This ensures that all operations have been completed and resources are released. After finalizing, you cannot re-initialize MPI.

Multithreading MPI supports several levels of thread usage, allowing you to control how multiple threads interact with the MPI library. The thread level defines how many threads can concurrently make MPI calls and how they are synchronized. The available thread levels are:

- `MPI_THREAD_SINGLE`: only the main thread is active; no other threads are involved in MPI communication.
- `MPI_THREAD_FUNNELED`: the application is multithreaded, but only the main thread makes MPI calls. Other threads may perform computations but cannot call MPI functions.
- `MPI_THREAD_SERIALIZED`: the application is multithreaded, but MPI functions can be called by only one thread at a time. The calls are serialized, so one thread accesses MPI at a time, though others can run concurrently.
- `MPI_THREAD_MULTIPLE`: the application is fully multithreaded, and multiple threads can safely call MPI functions simultaneously. This level offers no restrictions on how threads interact with MPI.

Most modern MPI implementations support all the threading levels, ensuring flexibility for developers working in a multithreaded environment. If the implementation supports the desired level, all MPI functions are thread-safe and can progress independently, allowing different threads to work concurrently without interfering with each other.

6.3.2 Communicator

A communicator in MPI defines a group of processes that share a communication context. It represents the environment in which processes can communicate with each other. Each communicator is associated with a variable of type `MPI_Comm`, which identifies the communication group. A process group within a communicator may include all, some, or just one of the processes in an MPI application. Two primary communicators are automatically created during MPI initialization:

- `MPI_COMM_WORLD`: represents a communicator that includes all the processes in the MPI environment.
- `MPI_COMM_SELF`: represents a communicator that includes only the current process.

To interact with communicators, the following functions are commonly used to obtain information about the communicator:

```
// Returns the number of processes in a communicator
int MPI_Comm_size()

// Returns the rank of the calling process within the specified communicator
int MPI_Comm_rank()
```

The rank of a process in a communicator is important for several operations:

- *Data movement between processes*: the rank is often used to direct data from one process to another.
- *Partitioning computation*: you can partition tasks based on the rank, such as distributing parts of a matrix or dividing a large dataset into subsets. Each process with a different rank might handle a different part of the computation.
- *Assigning roles*: the rank is also useful for assigning different roles or responsibilities to processes.

6.3.3 Point-to-point communications

Point-to-point communication in MPI involves functions to send and receive messages between two processes. MPI provides both blocking and non-blocking communication primitives to suit different application needs.

Message A message in MPI typically consists of the following elements:

- *Sender and receiver ranks*: the unique identifiers of the source and destination processes.
- *Tag*: an integer representing the topic of the message.
- *Buffer*: the actual data being sent or received.
- *Communicator*: the group of processes involved in the communication.

MPI supports different modes for handling outgoing messages:

1. *Synchronous*: the `MPI_Send` operation blocks until a matching `MPI_Recv` is posted.
2. *Buffered*: messages are stored in a buffer, allowing the `MPI_Send` call to complete before the `MPI_Recv` starts.
3. *Standard*: MPI dynamically decides between synchronous and buffered modes based on implementation and runtime conditions.
4. *Ready*: the send operation assumes a matching receive is already posted, reducing overhead from handshaking.

Specialized functions like `MPI_Bsend`, `MPI_Ssend`, and `MPI_Rsend` allow explicit use of these modes.

Data types MPI defines a set of built-in Plain Old Data types via the `MPI_Datatype` enumeration. Applications can define custom derived types based on these.

6.3.3.1 Blocking semantic

Message sending MPI provides several functions to send messages. The following are commonly used blocking send operations:

```
int MPI_Send()
int MPI_Send_c()
```

A blocking `MPI_Send` call ensures:

- The message buffer is safe for reuse after the call returns.
- For synchronous communication, a matching `MPI_Recv` operation has been posted.

Message receiving To receive messages, MPI provides blocking functions like:

```
int MPI_Recv();
int MPI_Recv_c();
```

A blocking `MPI_Recv` call waits until:

- A matching `MPI_Send` operation is posted.
- The data transfer to the buffer and status object is complete.

If the buffer is insufficient to hold the incoming data, an overflow error occurs.

Handling message information The `MPI_Status` object contains metadata about received messages. The following functions help extract details like message size:

```
int MPI_Get_count();
int MPI_Get_count_c();
```

When dynamic allocation is required, you can "probe" the message to determine its size:

```
Int MPI_Probe()
```

Limitations Messages sent by the same source to the same destination are delivered in order. There is no global fairness or ordering between messages from different processes. Deadlocks can occur due to improper mixing of blocking and non-blocking operations.

6.3.3.2 Communication with non-blocking semantics

Non-blocking communication in MPI decouples the initiation and completion of operations, enabling concurrent computation and communication. This approach improves application performance by allowing tasks to progress while waiting for communication to complete.

Non-blocking operations in MPI are structured as follows:

- *Initiation*: the function initiating the communication returns immediately, allowing the program to proceed without waiting.

- *Completion*: a separate function ensures the operation is finalized, potentially blocking until the communication completes.

Accessory functions, such as `MPI_Test`, help monitor and contribute to the progress of these operations.

MPI provides functions to initiate non-blocking communication:

```
int MPI_Isend()
int MPI_Isend_c()
int MPI_Irecv()
int MPI_Irecv_c()
```

The `MPI_Request` output parameter represents the handler for the initiated operation. This handler is used to track or finalize the communication.

Request ending The following function blocks the execution flow until the corresponding non-blocking operation is complete:

```
int MPI_Wait()
```

Request checking To check the status of a non-blocking operation without blocking, you can use:

```
int MPI_Test()
```

This function sets the `flag` variable to a nonzero value if the operation is complete, while also contributing to the progress of the operation.

Request cancellation Non-blocking communication can be explicitly canceled:

```
int MPI_Cancel()
```

This function is agnostic to the type of request and attempts to abort the ongoing operation.

6.3.3.3 Comparison

Non-blocking communication and multithreading share the goal of overlapping computation with communication. However, there are distinct differences:

- *Code clarity*: multithreading may provide a more structured approach, but requires thread-safe programming and proper synchronization.
- *Operation cancellation*: non-blocking operations can be explicitly canceled, which is not easily achievable with threads, as terminating threads is typically discouraged.
- *Legacy code*: high-performance computing (HPC) applications often use non-blocking semantics for backward compatibility and finer control.

Despite similarities, non-blocking operations are preferred in many HPC scenarios due to their simplicity, explicit control, and compatibility with MPI's architecture.

6.4 Collective communications

Collective communication in MPI involves operations that require the participation of all processes within a communicator. These operations enable efficient data exchange and synchronization across processes.

6.4.1 Communicator handling

Collective communication operations are blocking and follow these rules:

1. All processes in the communicator must call the matching collective function.
2. Execution is blocked until the process's contribution is complete, and the associated buffer (if any) is safe for reuse.

Completion for one process does not imply completion for others unless explicitly guaranteed by the operation.

Communicators define groups of processes that can participate in collective communication. Each communicator represents an independent universe (`MPI_COMM_WORLD`), which enables:

- Avoidance of unintended side effects, useful for library development.
- Reduction in synchronization overhead.
- Better alignment with specific application algorithms.

Duplicating a communicator Create a copy of an existing communicator:

```
int MPI_Comm_dup()
```

Splitting a communicator Partition a communicator into multiple sub-communicators:

```
int MPI_Comm_split()
```

Release a communicator Obsolete communicators should be released using:

```
int MPI_Comm_free()
```

Advanced operations Other advanced operations are:

1. Extract the group of processes in a communicator:
2. Manipulate the group.
3. Create a new communicator from the modified group:

6.4.2 Synchronization

Synchronization in MPI ensures that processes within a communicator align their execution at specific points. The most common mechanism for this is the barrier operation.

A barrier enforces all processes in a communicator to reach the same point in their execution before any can proceed. This is achieved using the following function:

```
int MPI_Barrier()
```

Unlike other collective operations, completion of the barrier for one process does not guarantee the same for all others. The barrier completes for a process only when all processes in the communicator have reached it. It is primarily used to establish a clear synchronization point, ensuring that all processes are ready before proceeding to subsequent computations or communications.

6.4.3 Data transfer

MPI provides a collection of functions tailored to handle common data exchange patterns, combining sending and receiving operations into a single, efficient interface. These collective communication routines typically outperform point-to-point communication due to optimizations in their implementation.

Broadcasting Broadcasting allows one process (the root) to send the same data to all other processes in the communicator. The following functions implement broadcasting:

```
int MPI_Bcast()
int MPI_Bcast_c()
```

The difference between `MPI_Bcast` and `MPI_Bcast_c` lies in the size of the count parameter, allowing support for larger data sizes in the `_c` variant.

Gathering Gathering collects data from all processes and consolidates it into a single buffer on the root process. This is performed using:

```
int MPI_Gather()
int MPI_Gather_c()
```

All processes must exchange the same amount of data. For variable-sized data, use `MPI_Gatherv` or `MPI_Gatherv_c`. Only the root process ends up with the consolidated data. If all processes need the gathered data, consider using `MPI_Allgather` or `MPI_Allgather_c`.

Scattering Scattering is the inverse of gathering, where the root process distributes data to all other processes. This is achieved through:

```
int MPI_Scatter()
int MPI_Scatter_c()
```

All processes must receive the same amount of data.

6.5 Heterogeneous computing

General-purpose processors are often inefficient in terms of energy consumption. In contrast, the circuitry needed to perform specific computations can be relatively simple and optimized for efficiency. Selecting the appropriate tool for a task is crucial for achieving the best balance between performance and energy consumption:

- *CPU*: control-oriented, easy to program but less energy-efficient.
- *GPU*: throughput-oriented, up to 10x more energy-efficient than CPUs.
- *FPGA*: can be up to 100x more energy-efficient, though programming is more complex (active research aims to simplify this).
- *ASIC*: fixed-function, up to 100-1000x more efficient but not programmable and costly.

To further reduce energy consumption, consider these two strategies:

1. *Use specialized processors*: these are more energy-efficient as they perform more operations per joule.
2. *Minimize data movement*: reducing communication overhead not only boosts performance but also cuts down on energy usage.

The ideal parallel programming language should balance performance, productivity, and generality. Achieving this balance allows for effective development while maintaining high efficiency and flexibility across different computing environments.

6.5.1 Domain-Specific Languages

Domain-specific languages (DSLs) aim to bridge the gap between performance and productivity by offering specialized solutions for specific domains. These languages are designed with restricted expressiveness, focusing on particular problem sets to provide more efficient and intuitive approaches. Typically, they have a high-level, often declarative syntax and are deterministic in nature. DSLs can be external or embedded within another general-purpose language, such as SQL, MATLAB, or Halide, allowing developers to work at a higher abstraction level while still benefiting from specialized optimizations.

Image processing Image processing is a prime example of a data-intensive domain where significant optimization is needed. Imaging applications are pervasive in our daily lives, and almost every imaging sensor is coupled with a powerful compute unit to process the captured data. The demand for optimization in image processing is immense, driven by the need to maximize performance while minimizing power consumption.

The key challenges in this area revolve around two core concepts: parallelism and locality. Parallelism refers to the ability to split computations into smaller tasks that can be performed simultaneously across multiple processors. Locality, on the other hand, focuses on efficient memory access patterns that reduce latency and energy consumption.

There are several approaches to improving performance in image processing. First, faster algorithms are necessary, and sometimes approximate computing methods are employed, where precision is sacrificed for faster results. Second, advancements in hardware can significantly impact performance. Faster CPUs, with higher frequencies and increased memory bandwidth,

along with SIMD vector operations, can process data more rapidly. Additionally, leveraging the parallel processing capabilities of GPUs offers substantial improvements in throughput for tasks that are well-suited to parallel execution.

However, hardware efficiency goes beyond just having faster processors. The key lies in utilizing the hardware as effectively as possible, which can be achieved through parallelism, better cache management, and optimized memory usage. All of these factors contribute to faster and more energy-efficient image processing.

To achieve these goals, it is often necessary to reorganize computation, restructuring both the program and data to exploit parallelism and locality effectively. Common approaches in existing solutions include using C++ with multithreading and SIMD operations, as well as utilizing CUDA or OpenCL for GPU acceleration. Optimized libraries such as BLAS, IPP, MKL, and OpenCV also provide specialized functions to improve performance.

6.5.1.1 Halide

Halide proposes a radically different approach: decoupling the algorithm from the schedule. This allows for flexibility in optimization decisions, creating a choice space for improving performance. The performance of an image processing pipeline requires complex trade-offs and frequent reorganization of computation. Halide simplifies this iterative process, enabling easier exploration of optimization options.

Strategy
Compute and store producer, then compute consumer
Compute producer values when needed by consumer, then throw them away
Compute producer values when needed by consumer, then keep the old values for reuse
Compute producer values when needed by consumer, then throw them away
Compute a subset of producer values all at once, then pass them to the consumer

Achieving optimal performance involves navigating complex trade-offs and frequently reorganizing computations. Halide simplifies this iterative process, providing a more efficient and flexible framework for exploring and implementing optimizations.

6.5.2 High-Level Synthesis

Traditional design flow The Register-Transfer Level (RTL) design approach involves specifying both the behavior and structure of a system, including components like multiplexers and registers, using hardware description languages like Verilog or VHDL. This method provides precise, fine-grained control over the design but is time-consuming and error-prone, requiring significant expertise and effort from hardware designers.

Behavioral design flow In contrast, the Behavioral Design Flow automates the translation of high-level descriptions into RTL code, akin to how compilers translate software code into machine instructions.

FSMD model The FSMD (Finite State Machine with Datapath) model is a key framework for designing digital systems. It consists of two main components: the Datapath and the Controller.

The datapath is responsible for handling data operations and storage. The controller governs the operation of the datapath using a finite state machine (FSM).

The process of converting high-level code to hardware circuits involves several steps:

1. *Build the FSM for the controller*: this step defines the sequence of states and actions.
2. *Build the datapath*: this includes designing the functional, memory, and interconnection resources.

In High-Level Synthesis (HLS), these steps are typically performed automatically, streamlining the design process. The key tasks involved include:

1. *Scheduling*: determining the order in which operations will be performed.
2. *Resource allocation*: allocating the necessary hardware resources for each operation.
3. *Binding*: mapping operations to specific resources in the datapath.

One potential optimization in the FSMD model is reducing the number of operations required.

6.5.2.1 Scheduling

Scheduling refers to the assignment of operations to specific time steps or control steps in the execution process, often constrained by hardware resources and timing requirements. It aims to exploit parallelism wherever possible to optimize the overall system performance. The possible solutions are:

- *Minimum latency unconstrained scheduling*: operations have bounded delays that are expressed as the number of clock cycles, but there are no area constraints. The possible solutions are:
  - ASAP scheduling: this method schedules each operation as early as possible, subject to dependency constraints.
  - ALAP scheduling: this method schedules operations as late as possible without violating dependencies.

  The mobility is the difference between the ASAP and ALAP start times. Operations with zero mobility are critical and must occur on the critical path, meaning any delay in their execution would directly increase the overall latency.
- *Constrained scheduling*: in more complex scenarios, scheduling must also consider constraints on resources or latency. The possible solutions are:
  - *Minimize latency with resource constraints* (ML-RCS): the goal is to reduce latency while respecting the hardware resource limits.
  - *Minimize resources with latency constraints* (MR-LCS): this approach aims to reduce resource usage while ensuring the latency does not exceed a predefined limit.

  Both these problems are $\mathcal{NP}$-hard.

List-based scheduling List-based scheduling is a heuristic method used for both ML-RCS and MR-LCS. It does not guarantee an optimal solution but uses a greedy strategy to provide a quick, approximate solution. The time complexity is $\mathcal{O}(n)$, where n is the number of operations.

Algorithm 24 List-based scheduling

1: Construct a priority list based on factors such as operation mobility.
2: **while** not all operations have been scheduled **do**
3: For each available resource, select an operation from the ready list, following the descending priority order.
4: Assign the selected operations to the current clock cycle.
5: Update the ready list to reflect the remaining operations.
6: Increment the clock cycle.
7: **end while**

CHAPTER 7

Parallel programming patterns

7.1 Dependencies

Parallel execution is inherently constrained by the sequence of operations required to ensure a correct result. To achieve efficient parallelism, it is crucial to address dependencies. A dependency occurs when one operation must complete and produce a result before a subsequent operation can proceed.

In addition to the traditional dependencies among operations, we extend this concept to resource dependencies, where certain operations may rely on the availability of specific resources to execute.

The fundamental assumption in concurrent execution is that processors operate independently. There are no assumptions regarding the speed of execution between processors, meaning each processor can run at its own pace.

Sequential consistency Sequential consistency ensures that the execution of statements does not interfere with one another, and the computation results remain consistent, regardless of the order in which the operations are executed. When this condition holds true, we can conclude that the two statements are independent of each other.

Definition (*True dependence*)**.** A statement S_2 has a true (flow) dependence on S_1 if and only if S_2 reads a value that was written by S_1.

Definition (*Anti-dependence*)**.** A statement S_2 has an anti-dependence on S_1 if and only if S_2 writes a value that was previously read by S_1.

Definition (*Output dependence*)**.** A statement S_2 has an output dependence on S_1 if and only if S_2 writes a value that was also written by S_1.

Condition	Dependence Type	Description
$\text{out}(S_1) \cap \text{in}(S_2)^1$	Flow Dependence	$S\delta S_2$
$\text{in}(S_1) \cap \text{out}(S_2)^1$	Anti-Dependence	$S\delta^{-1} S_2$
$\text{out}(S_1) \cap \text{out}(S_2)^1$	Output Dependence	$S\delta^0 S_2$

Both anti-dependences and output dependences are referred to as name dependencies, where the dependence arises due to the reuse of variable names, rather than the actual values. Two statements S_1 and S_2 can be executed in parallel only if there are no dependencies between them.

Dependencies relations can be analyzed by comparing the in and out sets for each statement:

- $\text{in}(S)$: the set of memory locations (variables) that may be read by S.
- $\text{out}(S)$: the set of memory locations (variables) that may be modified by S.

Loops Significant opportunities for parallelism often exist within loops. A loop-carried dependence is a dependence that occurs only when the statements are part of a loop's execution. In contrast, dependencies between two instances of a statement in the same iteration are loop-independent. Loop-carried dependencies can limit the parallelization of loop iterations, as they require that certain operations occur in sequence.

7.2 Patterns

Parallel patterns refer to recurring combinations of task distribution and data access strategies that address specific challenges in parallel algorithm design. One of the key strengths of parallel patterns is their universality.

7.2.0.1 Serial control patterns

Structured serial programming relies on four fundamental control patterns:

- *Sequence*: a sequence is an ordered list of tasks executed in a specific order.
- *Selection*: in a selection pattern, a condition c is evaluated first. Depending on whether the result of c is true or false, either task a or task b is executed. The assumptions are that neither task a nor task b can be executed before c, and only one of them is executed.
- *Iteration*: an iteration involves evaluating a condition c. If c is true, task a is executed, and the condition is evaluated again. This cycle repeats until c becomes false.
- *Recursion*: dynamic form of nesting where functions can call themselves.

7.2.0.2 Parallel control pattern

Parallel control patterns extend the serial control patterns, building on the same principles but relaxing some of the assumptions inherent in serial execution. Each parallel control pattern corresponds to at least one serial control pattern but allows for concurrent execution, enabling parallelism. The key parallel control patterns are:

- *Fork-join*: this pattern enables the control flow to fork into multiple parallel flows, which later rejoin. Functions that spawn another function call continue to execute, while the caller synchronizes with the spawned function to join both paths.
- *Map*: this pattern applies a function to every element of a collection. It replicates a serial iteration pattern where each iteration is independent, the number of iterations is known in advance, and the computation depends only on the iteration index and input data from the collection.
- *Stencil*: an elemental function accesses a set of neighbors. Stencil is a generalization of the map pattern and is commonly used in iterative solvers or to evolve a system over time.
- *Reduction*: combines every element in a collection using an associative combiner function. The associativity of the combiner function allows different orderings of the reduction, making the process parallelizable.
- *Scan*: computes all partial reductions of a collection. For each output in the collection, a reduction of the input up to that point is computed. If the function used in the scan is associative, the operation can be parallelized.
- *Recurrence*: a more complex version of the map pattern, the recurrence pattern allows loop iterations to depend on one another. For a recurrence to be computed in parallel, there must be a serial ordering of the recurrence elements, ensuring that each element is computed using previously computed outputs.

7.2.0.3 Serial data management patterns

Serial programs manage data in various ways depending on how it is allocated, shared, read, written, and copied. Data management is essential for ensuring that data is handled correctly and efficiently, especially when transitioning from serial to parallel execution. The following are key serial data management patterns:

- *Random read and write*: this pattern involves accessing memory locations indexed by addresses.
- *Stack allocation*: stack allocation is used for dynamically allocating data in a LIFO manner. It is efficient, as an arbitrary amount of data can be allocated in constant time.
- *Heap allocation*: heap allocation is useful when data cannot be allocated in a LIFO manner. While more flexible than stack allocation, heap allocation is slower and more complex due to the dynamic nature of memory management.
- *Objects*: objects are constructs in object-oriented languages that associate data with functions to manipulate and manage that data.

7.2.0.4 Parallel data management patterns

To avoid issues such as race conditions, it is crucial to understand when data is potentially shared among multiple parallel workers. Effective data management is essential for ensuring correct and efficient parallel execution. Some parallel data management patterns help improve data locality and prevent conflicts. The following are key parallel data management patterns:

- *Pack*: used to eliminate unused space in a collection. Elements marked as false are discarded, and the remaining elements are placed in a contiguous sequence, preserving their original order.
- *Pipeline*: connects tasks in a producer-consumer fashion, where one task produces data that is consumed by another.
- *Geometric decomposition*: divides data into subcollections, which can be either overlapping or non-overlapping. This pattern offers an alternative view of the data to improve parallel processing.
- *Gather*: reads a collection of data based on a set of indices. It combines elements of the map pattern with random serial reads. The output collection shares the same type as the input collection but takes the shape of the indices collection.
- *Scatter*: requires a set of input data and corresponding indices. Each element of the input data is written to the output collection at the specified index.

7.3 Map pattern

The map pattern is a widely used parallel programming technique where a function is applied independently to each element in a collection. The independence of each operation is key to the pattern's parallelizability, allowing for efficient computation, especially when combined with optimizations such as code fusion and cache management.

Independence The key advantage of the map pattern is its inherent independence. This makes it ideal for parallelization. In a map operation, there should be no shared state between iterations, meaning that each iteration operates only on its input and produces its output without interfering with other iterations. This independence allows map operations to be executed in parallel, yielding significant speedups, often in the order of $\mathcal{O}(\log n)$, particularly for large datasets.

Optimizations To optimize the mapping it is possible to use some techinques:

- *Multi-maps*: the map pattern is extended to multiple collections, where a function is applied to elements across more than one collection simultaneously.
- *Code fusion*: fuse map operations, performing them all in a single pass.

7.3.1 Implementation

The main map patterns include the following:

- *Stencil*: each instance of the map function accesses neighboring elements of its input, typically offset from its usual position.
- *Workpile*: the workpile pattern allows new work items to be dynamically added to the map during its execution.
- *Divide-and-conquer*: this pattern applies when a problem can be recursively divided into smaller subproblems, with each subproblem being solved independently.

7.4 Reduce pattern

The reduce operation is a powerful tool for aggregating a collection of elements into a single summary value. It is commonly used in parallel and distributed computing frameworks to perform summarization tasks efficiently.

The reduce operation relies on a combiner function that aggregates elements pairwise. The combiner function must be associative to ensure that the operation can be parallelized.

Tiling Tiling breaks the overall task into smaller chunks, allowing individual workers to process and reduce these chunks serially. This division ensures efficient workload distribution across multiple processing units.

7.5 Scan pattern

The scan pattern produces partial reductions of an input sequence and generates a new sequence. Unlike reduce, scan computes all intermediate results, making it trickier to parallelize effectively. The scan may be:

- *Inclusive*: an inclusive scan includes the current element in its partial reduction.
- *Exclusive*: an exclusive scan excludes the current element, producing a partial reduction of all prior elements.

One algorithm for parallelizing scan involves two main phases:

1. *Up sweep*: perform a reduction operation on the input sequence to aggregate partial results.
2. *Down sweep*: use the results of the up sweep to generate intermediate values for the scan.

Tiling Similar to reduce, tiling can be applied to divide the work into manageable chunks for individual workers. This approach involves:

1. Breaking the input sequence into tiles.
2. Performing partial scans within each tile.
3. Combining the results to produce the final output.

7.6 Pack pattern

The pack operation is used to eliminate unused elements from a collection and consolidate retained elements into contiguous memory locations. This reorganization improves performance by enhancing data locality and reducing cache misses. The pack operation follows these steps:

1. Convert the input array into Booleans, where 1 represents retained elements and 0 represents discarded ones.
2. Perform an exclusive scan on this integer array using the addition operation to compute offsets.
3. Write retained values to the output array based on the computed offsets.

7.6.1 Implementation

The main pack patterns include the following:

- *Unpack*: restores the original structure of the data by spreading retained elements back to their initial locations.
- *Split*: generalizes the pack operation by moving elements into distinct regions of the output collection based on a classification state. Unlike pack, split retains all input data and does not discard any information.
- *Unsplit*: reconstruct the original input collection from the separated data.
- *Bin*: extends the split operation to support more than two categories.
- *Expand*: elements meeting specific criteria are retained and packed together.

7.7 Gather pattern

The gather pattern is a mechanism for creating a new collection of data by extracting elements from an existing source collection based on a set of indices. This operation is particularly useful in parallel and distributed computing for data reorganization and preparation tasks.

Given a collection of ordered indices, the gather pattern reads data from the source collection at each specified index, and wrtites the retrieved data to the output collection, maintaining the order of indices. The resulting output collection has the same element type as the source collection, and the same dimensionality and shape as the index collection.

7.7.1 Implementation

The main gather patterns include the following:

- *Zip*: combines multiple collections into a single collection of tuples, preserving corresponding elements across input arrays.
- *Unzip*: separate a collection of tuples into individual sub-collections.
- *Shift*: move data within a collection by fixed offsets.

7.8 Scatter pattern

The scatter pattern involves distributing data from an input collection to specific locations in an output collection based on a set of write locations. The output collection size may need to be larger to accommodate all write locations without overlap.

Scatter-gather conversion Scatter operations are generally more expensive than gather operations due to cache-related challenges. These issues can be mitigated by converting scatter operations into gather operations when the write addresses are known in advance. This allows optimizations to be applied, particularly when the scatter pattern is repeated, amortizing the conversion cost.

7.8.1 Collisions

Writes to the same location are possible, leading to potential collisions. Race conditions occur when two or more values are written to the same location in the output collection. This results in undefined behavior unless specific rules are enforced to resolve these collisions.

Collision rules Several strategies can be applied to resolve collisions in scatter operations:

- *Atomic*: a non-deterministic approach where one and only one of the colliding values is written.
- *Permutation*: collisions are detected in advance.
- *Merge*: associative and commutative operators are used to combine colliding elements.
- *Priority*: each input element is assigned a priority based on its position.