Advanced Algorithms and Parallel Programming

Niccoló Didoni

September 2022

Contents

Part I

Introduction

Chapter 1

Algorithms

1.1 Definition

Algorithms are the main topic of this course, hence it seems appropriate to start by giving a definition of them.

> **Definition 1.1** (Algorithm)**.** *An algorithm is a set of instructions that can be used to convert input data in output.*

Algorithms are usually confronted with respect to their performance in a rather abstract way. However, comparing algorithms with respect to their performance is complex since it depends on the machine and the input data the algorithm is fed with. Moreover, in complex scenarios, we also have to consider that

- One might use more than one executor of the algorithm.
- The executor might have to execute multiple algorithms. This means that we have to consider the case in which the algorithm has to be executed multiple times.

When approaching a problem, we also have to understand if there exist an algorithm able to solve such problem. Moreover, one might want to understand if, given an algorithm that solves a specific problem, the same algorithm can be applied to other problems. Namely, we want to understand if we can find a set of tools (i.e., algorithms) that can be used in different context and to solve different types of problems.

With respect to parallelisation, we also have to consider how much does spreading an algorithm on multiple processors help and how easy it is to parallelise. In same cases, parallelisation only requires us to run the same algorithm on different executors, using different data, while in others we need it isn't enough.

1.2 Analysis

When we want to analyse the complexity of an algorithm, we usually evaluate the number of instructions it executes. Many times, the number of instructions executed isn't fixed but depends on

the input. Consider for instance a sorting algorithm. Sorting an array of 3 elements is much faster (i.e., requires a smaller number of instructions) than sorting one of 1000 elements.

The performance of an algorithm might depend also on the input itself. For instance, if we consider a sorting algorithm on an integer vector of fixed length, we can get different performances depending on the order of the elements. If the elements are already in order, the algorithm might be much faster than the case in which the elements are randomly distributed. Namely, during the analysis of an algorithm, we have to consider the best, worst and average case. In some applications, considering the average case in enough (e.g., sorting), however in other scenarios we have to consider also the worst and best case. To sum things up:

- The **worst case** performance is the maximum execution time of an algorithm on any input of size n. Worst case analysis is used in critical systems in which we must assure a certain maximum execution time.
- The **average case** performance is the expected execution time of an algorithm over all inputs of size n. Computing the average execution time requires some assumptions on the statistical distribution of the inputs, since we have to compute an expected value.
- The **best case** performance is the minimum execution time of an algorithm over an input of size n.

1.2.1 Asymptotic analysis

Measuring the performance of an algorithm using the running time (i.e., the time needed to output a result) requires to execute the algorithm and may depend on the machine on which the algorithm is run. In some cases, we don't want to actually execute the algorithm and just get an hint on its complexity. Asymptotic analysis is used just for this and gives us an idea of the complexity of an algorithm, independently from the machine on which it's run. The basic idea of asymptotic analysis is to find out what is the growth of the execution time $T(n)$ when the size of the input goes to infinity $n \to \infty$. To analyse the asymptotic complexity of an algorithm we will use the notation

- $\Theta(g(n))$ to indicate the set of functions $f(n)$ such that there exists positive constants c_1, c_2 and n_0 such that
$$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \quad \forall n \geq n_0$$
Θ defines a strict bound for $g(n)$.
- $\mathcal{O}(g(n))$ to indicate the set of functions $f(n)$ such that there exist constants $c > 0$ and $n_0 > 0$ such that
$$0 \leq f(n) \leq cg(n) \quad \forall n \geq n_0$$
$\mathcal{O}()$ defines an upper bound for $g(n)$.
- $\Sigma(g(n))$ to indicate the set of functions $f(n)$ such that there exist constants $c > 0$ and $n_0 > 0$ such that
$$0 \leq cg(n) \leq f(n) \quad \forall n \geq n_0$$
Σ defines a lower bound for $g(n)$.

Doing asymptotic analysis might be straightforward in some cases, like in case of sequential algorithms. In such cases, we simply have to compute the number of instructions executed by the program, given an input size of n. Things get more complex when we talk about recursive functions. In this case, computing the asymptotic complexity isn't immediate and we have to use other methods like the master theorem.

Master theorem

The master theorem can be used to compute the asymptotic complexity $T(n)$ of an algorithm. In particular, if the execution time can be written as

$$T(n) = aT\left(\frac{n}{b}\right) + f(n) \tag{1.1}$$

with

- $a \geq 1$
- $b < 1$
- f asymptotically positive.

then we can use the Master theorem. In particular, we can recognise three different cases.

Case 1 When

- $f(n)$ is in the order of $n^{log_b a - \varepsilon}$ for some constant $\varepsilon > 0$.

$$f(n) = \mathcal{O}(n^{\log_b a - \varepsilon})$$

  Namely, $f(n)$ grows polynomially slower than $n^{log_b a}$ by an n^{ε} factor.

then the solution is

$$T(n) = \Theta(n^{\log_b a}) \tag{1.2}$$

Case 2 When

- $f(n)$ is in the order of $n^{log_b a} \log^k n$ for some constant $k \geq 0$.

$$f(n) = \Theta(n^{log_b a} \log^k n) \tag{1.3}$$

  Namely, $f(n)$ and $n^{log_b a}$ grow at similar rates.

then the solution is

$$T(n) = \Theta(n^{\log_b a} \log^{k+1} n) \tag{1.4}$$

Case 3 When

- $f(n)$ is in the order of $n^{log_b a + \varepsilon}$ for some constant $\varepsilon > 0$.

$$f(n) = \Omega(n^{log_b a + \varepsilon}) \tag{1.5}$$

  Namely, $f(n)$ grows polynomially faster than $n^{log_b a}$ by an n^{ε} factor.

- $f(n)$ satisfies the regularity condition

$$af\left(\frac{n}{b}\right) \leq cf(n)$$

  for some constant $c < 1$

then the solution is

$$T(n) = \Theta(f(n)) \tag{1.6}$$

The master theorem is very useful when we have to analyse an algorithm that uses the divide and conquer approach.

1.3 Divide and conquer

Divide and conquer algorithms are widely used in many fields, hence it's useful to give the tools to analyse the complexity of such algorithms. Divide and conquer algorithms follow three different rules:

1. **Divide** a problem in many sub-instances (sub-problems).
2. **Solve** (conquer) each sub-problem recursively, i.e., calling the same function again (hence dividing the sub-problem in sub-sub-problems).
3. **Combine** the solutions of the sub-problems.

1.3.1 Merge sort

The merge sort algorithm uses a divide-and-conquer approach in fact:

- In the divide part we split the input vector in two.
- In the conquer part we recursively call the merge sort on the two sub-vectors.
- In the combine part we merge the sub-vectors returned by the recursive calls.

The complexity of the divide and conquer algorithm can be written as

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

Since we recursively apply the algorithm on two sub-vector of half the size of the input vector and then we linearly merge the results. This complexity falls in case 2 of the Master theorem, with $a = 2$, $b = 2$ and $k = 0$, hence the complexity of merge sort is

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

1.3.2 Binary search

Another example of divide-and-conquer algorithm is binary search. Binary search allows to search an value inside a sorted vector. In particular,

- In the divide phase, the algorithm takes the cell in the middle of the array and checks if the value is the one searched.
- In the conquer phase, if the value is not in the middle cell, the value is recursively searched in the right sub-vector, if the searched value is bigger than the one in the middle, otherwise it's searched to the left.
- In the combine phase, the value is returned.

Formally, the algorithm is shown in Algorithm 1.

The complexity of this algorithm is

$$T(n) = T(n/2) + \Theta(1)$$

which falls in case 2 of the Master theorem since $\log_2 1 = 0$ and we can choose $k = 0$ to obtain

$$T(n) = \Theta(n^{\log_b a} \log^{k+1} n) = \Theta(n^{\log_2 1} \log^1 n) = \Theta(\log n)$$

Algorithm 1 The binary search algorithm

```
procedure BINARY-SEARCH(a, v, b, e)
    if b = e ∧ a[b] ≠ v then
        return −1
    end if
    m ← b + ((b − e)/2)                          ▷ The middle position
    if a[m] = v then
        return m
    end if
    if v > a[m] then
        return BINARY-SEARCH(a, v, m + 1, e)
    end if
    return BINARY-SEARCH(a, v, m, m − 1)
end procedure
```

1.3.3 Power

We can compute the power of a number by repeated multiplication, which requires linear time. Luckily there exist an algorithm that can compute the power in logarithmic time by using a divide-and-conquer approach. Formally the algorithm computes the power of a number by applying the following formula:

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

The pseudocode for the algorithm is shown in Algorithm 2.

Algorithm 2 The divide-and-conquer algorithm for computing the power of a number.

```
procedure POW(a, n)
    if n = 1 then
        return a
    end if
    if n = 0 then
        return 1
    end if
    if n mod 2 = 0 then
        p ← POW(a, n/2)
    else
        p ← POW(a, (n−1)/2)
    end if
    return p · p
end procedure
```

The complexity of this algorithm can be written as

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

which falls, as for binary search, in case 2 of the Master theorem, hence we have a complexity of

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

1.3.4 Matrix multiplication

The basic implementation of matrix multiplication requires to compute n^3 operations (for each of the n^2 output pixels we have to compute the product between $2n$ values, hence n products).

Basic divide-and-conquer algorithm

The divide-and-conquer algorithm considers the fact that the product between two 2×2 matrices can be computed using 8 products between the elements of the matrices. In particular, given

$$C = A \cdot B$$
$$\begin{bmatrix} c_{00} & c_{01} \\ c_{10} & c_{11} \end{bmatrix} = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} \cdot \begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{bmatrix}$$

we can compute the values in C as

- $c_{00} = a_{00} \cdot b_{00} + a_{01} \cdot b_{10}$
- $c_{01} = a_{00} \cdot b_{01} + a_{01} \cdot b_{11}$
- $c_{10} = a_{10} \cdot b_{00} + a_{11} \cdot b_{10}$
- $c_{11} = a_{10} \cdot b_{01} + a_{11} \cdot b_{11}$

This means that we can take two arbitrarily large square matrices of even dimensions (the same for both matrices), divide the matrices in 2×2 matrices of $\frac{n}{2} \times \frac{n}{2}$ sub-matrices and apply the four formulas above to compute the product. In particular:

- In the divide part, we split the input matrices in four sub-matrices of dimension $\frac{n}{2} \times \frac{n}{2}$. We obtain 8 $\frac{n}{2} \times \frac{n}{2}$ sub-matrices.
- In the conquer part, we recursively compute the products required to compute c_{00}, c_{01}, c_{10} and c_{11}.
- In the combine part, the results obtained in the conquer part are summed up to compute c_{00}, c_{01}, c_{10} and c_{11}.

The algorithm is also shown in Algorithm 3. Since we divide the two input matrices in 4 sub-matrices, then we do 8 recursive calls, each using a matrix with half the dimensions, hence we get

$$T(n) = 8T\left(\frac{n}{2}\right) + \Theta(n^2)$$

which falls into case 1 of the Master theorem. The complexity of the divide-and-conquer algorithm is therefore

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

Algorithm 3 The divide-and-conquer matrix multiplication algorithm. Note that the algorithm considers only square matrices with even dimensions.

procedure MATRIX-MULTIPLY(A, B)
  $s \leftarrow \text{SIZE}(A)$
  if $s = 1$ **then**
    return $A[0,0] \cdot B[0,0]$
  end if
  $A00 \leftarrow A[0 : \frac{s}{2}, 0 : \frac{s}{2}]$
  $B00 \leftarrow B[0 : \frac{s}{2}, 0 : \frac{s}{2}]$
  $A01 \leftarrow A[0 : \frac{s}{2}, \frac{s}{2} : s]$
  $B01 \leftarrow B[0 : \frac{s}{2}, \frac{s}{2} : s]$
  $A10 \leftarrow A[\frac{s}{2} : s, 0 : \frac{s}{2}]$
  $B10 \leftarrow B[\frac{s}{2} : s, 0 : \frac{s}{2}]$
  $A01 \leftarrow A[\frac{s}{2} : s, \frac{s}{2} : s]$
  $B01 \leftarrow B[\frac{s}{2} : s, \frac{s}{2} : s]$
  $m_1 \leftarrow \text{MATRIX-MULTIPLY}(A00, B00)$
  $m_2 \leftarrow \text{MATRIX-MULTIPLY}(A01, B10)$
  $m_3 \leftarrow \text{MATRIX-MULTIPLY}(A00, B01)$
  $m_4 \leftarrow \text{MATRIX-MULTIPLY}(A01, B11)$
  $m_5 \leftarrow \text{MATRIX-MULTIPLY}(A10, B00)$
  $m_6 \leftarrow \text{MATRIX-MULTIPLY}(A11, B10)$
  $m_7 \leftarrow \text{MATRIX-MULTIPLY}(A10, B01)$
  $m_8 \leftarrow \text{MATRIX-MULTIPLY}(A11, B11)$
  return $\begin{bmatrix} m_1 + m_2 & m_3 + m_4 \\ m_5 + m_6 & m_7 + m_8 \end{bmatrix}$
end procedure

Strassen algorithm

The Strassen algorithm uses the same structure of the basic divide-and-conquer algorithm, however it requires to compute only 7 products instead of 8, therefore the complexity becomes:

$$T(n) = 7T\left(\frac{n}{2}\right) + \Theta(n^2)$$

Again, we fall into case 1 of the Master theorem but this time we obtain a complexity of

$$T(n) = \Theta(n^{\log_2 7}) \approx \Theta(n^{2.81})$$

Chapter 2

PRAM

2.1 Model

Since we will talk about parallel computing, we have to introduce a model to define the behaviour of a machine with parallelisation capabilities. In particular, we will use a machine model, which is a simple way to describe a machine and can be used to describe the complexity of an algorithm.

To introduce the model used for a parallel machine, we have to start from a basic, non-parallel model: the RAM machine. The RAM machine is an abstract model that describes a system with

- An infinite number of memory cells.
- Shared memory.
- Cells that store any type of data, even a number with an infinite number of digits. This means that the space complexity of an algorithm is the number of memory cells used.
- A basic instruction set.
- Each instruction requiring one unit of time (e.g., a clock cycle). This means that the time complexity can be seen as the number of instructions executed.

From the RAM model, we can move to the PRAM model which has the following features:

- It's a collection of RAM machines.
- Communication happens through shared memory, hence the main memory has to be shared among all RAM machines.
- It has an infinite number of inputs, an infinite number of outputs and an infinite number of cells.
- It's an homogeneous system, namely all machines (i.e., their processors) are identical.
- Each processor has an infinite number of registers.
- Each processor requires the same time (i.e., one unit of time) to access main memory.

A computation on a PRAM machine is done in five steps:

1. All processors read a value from the input cell.
2. All processors read data from the shared memory cells.
3. All processors perform some computation on the data.
4. All processors write, each on its output cell, the result of the computation (if needed).
5. All processors write data in the shared memory, if needed.

As we can see, this model defines a machine in which every RAM executes the same computation (i.e., the same algorithm) on different data. Basically, all RAM machines execute the same program. In some cases, each processor might perform different computations, if it knows who it is.

Note that the computation on a PRAM machine is synchronous, namely, each RAM machine is in the same phase of the 5 we have defined above.

2.1.1 Shared memory conflicts

Each RAM machine in a PRAM has access to the same shared memory. This means that we have to handle conflict between processors that simultaneously access the memory. Having two processes that read the shared memory at the same time doesn't cause any conflict, since they don't modify the memory and read the same value. On the other hand, we have to carefully handle the cases in which a write is involved. In particular, we can classify PRAM machines, depending on the conflicts they can handle:

- **Exclusive reads**. In exclusive read PRAM machines, all processors can simultaneously read from distinct memory locations. Basically, each processor reads a different memory cell. These systems are the easiest to implement.
- **Exclusive writes**. In exclusive write PRAM machines, all processors can simultaneously write to distinct memory locations. In other words, each processor writes exclusively to a specific cell of memory.
- **Concurrent reads**. In concurrent read PRAM machines, all processors can simultaneously read from any memory location (hence, even from the same). Since we are talking about reads, having simultaneous reads doesn't create problems, because we don't write to memory.
- **Concurrent writes**. In concurrent write PRAM machines, all processors can write to any memory location (hence, even to the same). This is the hardest case to handle because we need to understand how the conflict has to be managed.

We can also have a combination of the aforementioned conflicts.

Since the most critical type of conflict is the one involving concurrent writes, let us analyse how to solve it. There exists three techniques to handle concurrent writes:

- **Priority concurrent writes**. In this case, processors have a priority based on which value is decided, and the processor with highest priority is the one that writes the data.
- **Common concurrent writes**. In this case, all processors are allowed to complete a write if and only if the values that have to be written are the same.
- **Arbitrary or random concurrent writes**. One randomly chosen processor is allowed to complete the write. This solution could be acceptable or not, depending on the algorithm.

2.1.2 Advantages

The PRAM model is very simple, however it can precisely describe a parallel machine, even using some approximation (e.g., we neglect the time needed by processors to communicate, or the implementation of the communication between processors and memory). The main advantages of this model are:

- It's **natural** for describing a parallel system (even if we neglect some detail). Namely, the idea of dividing a machine in many sub-machines (i.e., the RAM machines), each of which executes a program on different data, is very close to the way we think about parallelism.
- It's **formal**. Namely, once we have defined the number of processors, that upper bound will hold even when we scale the model to a real architecture.
- It's **simple**. The synchronisation mechanism is very simple since every processor does the same thing in the same stage, hence we don't have to care about synchronising the processors.
- It' **easy to write an algorithm**. We have to write just one algorithm and we only have to decide if a processor has to execute a set of instructions or not.
- It can be used to **benchmark** algorithms and give an abstract evaluation of the complexity of an algorithm. In particular, if there is no efficient solution for a PRAM model, then we can say that there is no efficient solution for any parallel system.
- It can be **mapped to other models**.

2.1.3 Computational power

If we use different solutions for handling conflict, we obtain models with different computational power. In order, from the most to the last powerful, we get:

1. Priority concurrent write machines.
2. Arbitrary concurrent write machines.
3. Common concurrent write machines.
4. Concurrent read exclusive write machines.
5. Exclusive read exclusive write machines.

That being said, exclusive read exclusive write machines are the most realistic ones, while priority concurrent write machines are the least realistic (building a priority machine is not easy).

2.1.4 Definitions

After introducing the PRAM model, let us clear up some concepts that will be useful when working with PRAMs.

Execution time

The time taken by a machine with one processor to solve a problem with input size n, using the best sequential algorithm, is indicated with

$$T^*(n)$$

This value is going to be used when evaluating the performance of a parallel algorithm, in fact we want to check if, using a parallel machine, we get a better result with respect to a single processor. The time taken to solve a problem by a parallel machine with p processors using a parallel algorithm is indicated as

$$T_p(n)$$

Note that in this case, the execution time depends on the input size n but also on the number of processors p. Note that the time $T_1(n)$ is different from $T^*(n)$ because, even if in both cases we have one processor only, the former computes the execution time of a parallel algorithm while the former of a sequential one. We also have to consider the fact that on a parallel machine, some synchronisation is required and it might not be possible to eliminate it when we have only one processor. Finally, we indicate with

$$T_\infty(n)$$

the shortest run time on any number of processors p.

Now that we have a reference $T^*(n)$ and the time $T_p(n)$ required by a parallel machine, we want to compute how much faster the parallel machine is. To do so we have to introduce the speed-up $SU_p(n)$ which is the ratio between the time required by a sequential, single processor machine and the one required by a parallel machine.

$$SU_p(n) = \frac{T^*(n)}{T_p(n)}$$

In both cases, the algorithms have to work on the same amount of data. Note that

- The speed-up is greater than 1 if the parallel algorithm performs better than the sequential one.
- The speed-up is smaller than 1 if the parallel algorithm performs worst than the sequential one.

Also note that the speed up is always bounded by the number of processor, however it might also be smaller.

$$SU_p \leq p$$

Another important parameter to consider is efficiency $E_p(n)$. Efficiency considers how the parallel program is performing with respect to the number of processors and it's computed as

$$E_p(n) = \frac{T_1(n)}{pT_p(n)}$$

We have an efficiency of 1 (which is the maximum value) if the number of instructions of the algorithm is exactly p times the time taken by the parallel program. Efficiency is related to how well we exploit the processors that we have.

Cost is related to the cost of using the processors of a parallel application and to the energy we are going to use. The cost is computed as the product of the number of processors $P(n)$ required for an input of size n and the time $T(n)$ required to execute the algorithm.

$$C(n) = P(n) \cdot T(n)$$

Finally, we have to define the concept of work

$$W(n)$$

The work is the total number of instruction that the parallel application is going to use. Work is important since it's usually compared with the number of instructions that the best sequential algorithm is going to use. Note that the work of a parallel program might be bigger than the one of a sequential program, since the former might have to compute something multiple times. However, what's important is the total time elapsed and not the actual number of instructions of the program. Namely, if the parallel algorithm is faster than the sequential one, it doesn't mean that the total number of instructions executed by the parallel machine (i.e., the sum of the instruction) has to be smaller than the number of instructions executed by the sequential machine. If we consider the same elapsed time, an algorithm that does less work is better since it consumes less energy.

2.2 Examples

Now that we have a clear idea of what a PRAM machine is, let us consider some examples of programs running on such machine.

2.2.1 Matrix-vector multiplication

Given a $N \times N$ matrix A and a $N \times 1$ vector x, we want to compute the product y (which is a column vector) between A and x.

$$y = Ax$$

The multiplication between A and x is performed as

$$y_i = \sum_{n=1}^{N} a_{in} x_n \quad \forall i = 1, \ldots, N$$

It's easy to notice that the value computed on a row i is independent from the value of a row j.

To parallelise the matrix-vector multiplication we can divide the matrix A in p groups, each

having N columns and N/p rows and assign each $\frac{N}{p} \times N$ matrix to a RAM. Long story short we get

$$A = \begin{bmatrix} A_1 \\ A_2 \\ \vdots \\ A_p \end{bmatrix} = \begin{bmatrix} \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1N} \\ a_{21} & a_{22} & \dots & a_{2N} \\ \vdots & & & \\ a_{(N/p)1} & a_{(N/p)2} & \dots & a_{(N/p)N} \end{bmatrix} \\ \begin{bmatrix} a_{((N/p)+1)1} & a_{((N/p)+1)2} & \dots & a_{((N/p)+1)N} \\ a_{((N/p)+1)1} & a_{((N/p)+1)2} & \dots & a_{((N/p)+1)N} \\ \vdots & & & \\ a_{(2N/p)1} & a_{(2N/p)2} & \dots & a_{(2N/p)N} \end{bmatrix} \\ \vdots \\ \begin{bmatrix} a_{(k+1)1} & a_{(k+1)2} & \dots & a_{(k+1)N} \\ a_{(k+2)1} & a_{(k+2)2} & \dots & a_{(k+2)N} \\ \vdots & & & \\ a_{N1} & a_{N2} & \dots & a_{NN} \end{bmatrix} \end{bmatrix} \tag{2.1}$$

Note that if we have as many processors as number of rows, each processor is given only one row of A.

For instance, if $N = 256$ and we have 32 processors, then each processor has to handle a sub-matrix A_i with 256 columns and $\frac{256}{32} = 8$ rows. Vector x is going to be shared by all processors. This problem requires each processor to write a different value of y, hence each processor accesses a different area of memory. This means that, we need an exclusive write PRAM to solve the matrix-vector multiplication problem. With respect to reads, we also have concurrent reads (all processors have to read x, which is shared), however reads are never a problem, even when concurrent.

Since the processors do not interfere one with the other, we say that the algorithm used for matrix-vector multiplication is embarrassingly parallel.

The parallel algorithm used for computing matrix-vector multiplication is shown in Algorithm 4. Let's explain the algorithm. Each processor i:

1. Reads (concurrently) the whole vector x and puts it in one of the infinite registers of processor i. This action requires to load N elements from memory.
2. Reads (exclusively) a sub-matrix A_i and puts it in one of the infinite registers of processor i. This action requires to read $\frac{N^2}{p}$ elements from memory.
3. Computes the product between the sub-matrix and the vector. This action requires, for each row (i.e., for $\frac{N}{p}$ times) to compute the product between an element of the row and an element of x. The sum is executed N times since each row contains N values, hence we need N operations. In total, if we repeat N operations for $\frac{N}{p}$ times, we get a complexity of $\frac{N^2}{p}$.
4. Writes (exclusively) the sub-vector computed at the previous step in y_i. This operation requires to write $\frac{N}{p}$ values.

Note that, index i is fundamental to tell the compiler how to divide the tasks between processors and to tell each processors which part of A to use and where to write the result.

If we were to analyse the matrix-multiplication algorithm, we would say that

- The complexity with a single processor is N^2 since we have to compute N^2 multiplications.

$$T_1(n) = \mathcal{O}(N^2)$$

Algorithm 4 The parallel matrix-vector multiplication algorithm.

$z \leftarrow x$
$B \leftarrow A_i$
$w \leftarrow Bz$
$y_i \leftarrow w$

- The complexity with p processors is lower since we can split the computation among the processors.

$$T_p(n) = \mathcal{O}\left(\frac{N^2}{p}\right)$$

This means that the speed up is

$$SU_p(n) = \frac{N^2}{\frac{N^2}{p}} = p$$

- The cost is

$$C(n) = \mathcal{O}\left(p \cdot \frac{N^2}{p}\right) = \mathcal{O}(N^2)$$

- The work is exactly the same we do for the sequential algorithm $W = C$, hence the efficiency is 1

$$E_p = \frac{T_1}{pT_p} = \frac{N^2}{p \cdot \frac{N^2}{p}} = 1$$

2.2.2 Vector unitary module

Given a vector A of N elements, we want to compute the sum of all elements of the vector. Sequentially, the best algorithm takes N time instants. To parallelise the program, we have to assume that $N = 2^k$ and split the data (i.e., the elements of the vector) among all processors so that each processor computes the partial sum between two elements. What we have obtained is a vector half the size of the previous vector. At the next iteration we split again the new vector among the processors so that, as before, each processor computes the partial sum between two elements. In this case, we aren't using all processors since the length of the vector has been halved. This process continues until we have a vector with only two elements, which are summed by one processor and saved into memory. If it wasn't clear before, the idea behind this algorithm is, at each iteration, to generate a new vector, of half the dimension, in which each element is the sum of two elements of the previous vector. Formally, the algorithm is described in Algorithm 5. As we can see, this is an example of algorithm in which each processor might or might not do something depending on its index i.

Note that this algorithm requires at least N processors since we have to copy the whole array from the input to main memory, and each processor copies one cell of the array.

To compute the complexity of Algorithm 5 we have to notice that

- Copying one cell of the array from the input cell to the main memory requires constant time.
- Each iteration of the `for` loop takes constant time (since it's a simple sum) and we do K iterations.

Algorithm 5 The parallel algorithm for computing the unitary module of a vector.

```
A ← A(I)
A → B(I)
for h = 1, ..., K do
    if i ≤ N/2^h then
        x ← B(2I − 1)
        y ← B(2I)
        z ← x + y
        z → B(I)
    end if
end for
if I = 1 then z → S
end if
```

This means that the complexity of computing the sum of all elements of a vector is

$$\mathcal{O}(k) = \mathcal{O}(\log_2(N))$$

Note that this type of algorithm can be applied to whatever associative operation (e.g., also the product). For this reason, this algorithm is recognised as a parallel pattern and it's called **reduction**. Note that a reduction algorithm can be seen as a tree in which the last level is the first computation and the root level is the last computation where we sum up the last two elements and compute the final result. This also explains the why the complexity is logarithmic.

With respect to performance we can say that:

- The time complexity with a single processor is N since in this case the processor has to compute all the sums. The complexity is also the same of the best sequential algorithm.

$$T^*(N) = T_1(N) = N$$

- The time complexity with N processors, since we said we need at least N of them is logarithmic (for the reasons we've analysed before).

$$T_{P=N}(N) = 2 + \log_2(N)$$

  The number 2 is given by the fact that each processor has to load the input value in memory.

- The speed-up can be computed as

$$SU_P(N) = \frac{N}{2 + \log_2(N)}$$

- The cost is

$$C(N) = P \times (2 + \log_2 N) \approx N \log_2 N$$

  As before, we have taken $P = N$ since N is the minimum value necessary for the algorithm to work.

- The efficiency is

$$E_p = \frac{T_1(N)}{pT_p(N)} = \frac{N}{N \log_2 N} = \frac{1}{\log_2 N}$$

  The efficiency is not 1 since some processors are not used.

To increase the efficiency of the algorithm, we can consider a number of processors much smaller than the length of the vector and:

- Each processor initially loads $\frac{N}{p}$ elements in main memory.
- The number of iterations is always k, however during each iteration a process might have to do more sums. Moreover, since $N \gg p$, processes are more active.

This means that the complexity of the parallel algorithm is

$$T_p(N) = \frac{N}{p} + \log_2 p$$

Accordingly, the other performance indices can be rewritten as follows:

- The speed-up can is
$$SU_P(N) = \frac{N}{\frac{N}{p} + \log_2 p} \approx p$$
- The cost is
$$C(N) = P \times (\frac{N}{p} + \log_2 p) \approx N$$
since $p \log_p$ is dominated by $N \gg p$.
- The work is
$$W(n) = N + P \approx N$$
- The efficiency is
$$E_p = \frac{T_1(N)}{pT_p(N)} = \frac{N}{P\left(\frac{N}{p} + \log_2 p\right)} \approx 1$$

2.2.3 Matrix multiplication

Given two $N \times N$ matrices A and B, we want to compute the row-column multiplication $C = AB$, where

$$C_{ij} = \sum_{n=1}^{N} A_{in} B_{nj}$$

To parallelise this algorithm, assuming $N = 2^k$, we have to combine the two we've seen before. More precisely:

1. Processor $p_{i,j,l}$ computes the value $A_{il}B_{lj}$ using the same pattern used for the vector-matrix multiplication. This operation is done in parallel by the N^3 processors.
2. The N processors $P_{ij1}, \ldots, P_{ijN}$ compute the sum $\sum_{l=1}^{N} A_{il}B_{lj}$. This operation is done in parallel by N^2 groups of N processors each (each group reduces the values $T_{ij1}, \ldots, T_{ijN}$).

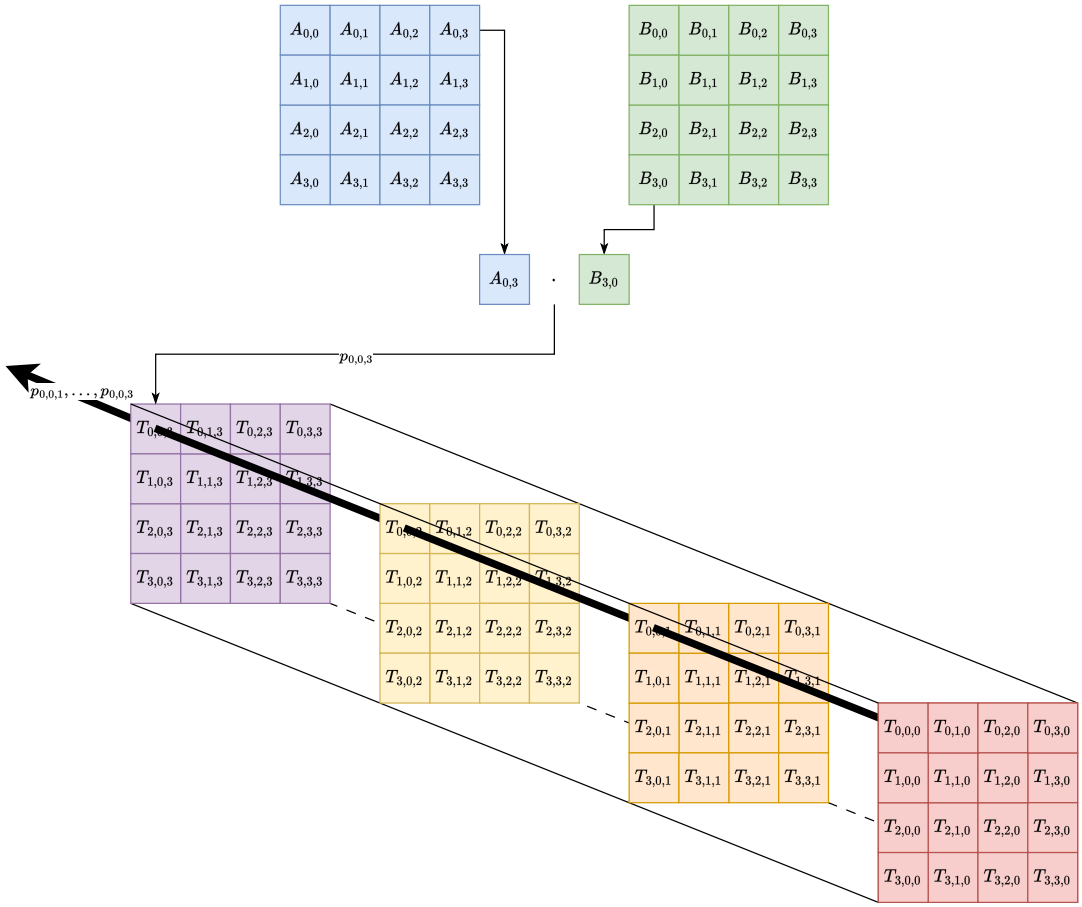


Figure 2.1: The matrix multiplication algorithm on a PRAM. Note that differently from the algorithm and the pseudo code we have used 0-indexed matrices.

Algorithm 6 The parallel matrix multiplication algorithm.

$T_{ijl} \leftarrow A_{il}B_{lj}$
for $h \leftarrow 1, \ldots, k$ **do**
  if $l \leq \frac{N}{2^h}$ **then**
    $T_{ijl} \leftarrow T_{ij(2l-1)} + T_{ij(2l)}$
  end if
end for
if $l = 1$ **then**
  $C_{ij} \leftarrow T_{ij1}$
end if

Formally, the algorithm is described in Algorithm 6 and a visual representation of the algorithm is shown in Figure 2.1.

The complexity of this algorithm is dominated by the `for` loop, which is executed k times, hence the complexity of this algorithm is

$$T_p(N) = \mathcal{O}(k) = \mathcal{O}(\log_2 N)$$

Note that, if we want to compute all the values $A_{il}B_{lj}$, we need to have $p = N^3$ processors. The reason is that we have to compute each value T_{ijl} and all indexes go from 1 to N, hence all possible combinations are N^3.

Knowing the complexity of the parallel algorithm we can compute the other performance indices:

- The complexity using one processor only is N^3.
- The speed-up is
$$SU_{P=N^3}(N) = \frac{N^3}{\log_2 N}$$
- The cost is
$$C(N) = P \cdot \log_2 N = N^3 \log_2 N$$
- The efficiency is
$$E_p = \frac{T_1(N)}{pT_p(N)} = \frac{N^3}{N^3 \log_2 N} = \frac{1}{\log_2 N}$$
Note that the efficiency is the same as for the sum of vector's elements.

2.2.4 Prefix sum

This example shows how a parallel algorithm can take advantage of idle processors. The prefix sum is a computation that takes as input a vector and computes j sums

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

A prefix sum S_i can also be seen as the sum between the element a_i and the prefix S_{i-1}. If we notice this characteristic, we immediately understand that the complexity of the best sequential algorithm is

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

We can use the reduction pattern to parallelise the prefix sum algorithm. Moreover, we can exploit the processors that are not used since, after each iteration, half of the previous active processors is not used anymore. To demonstrate how does the algorithm work, let us consider a simple setting with 8 elements $a_1, \ldots, a_8$ and 4 processors. Having 8 elements, the complexity of the algorithm should be 3 (remember we are using a reduction pattern, which has a logarithmic complexity). Let's follow the computations done by the algorithm:

1. At the first iteration:
   - p_1 computes $a_1 + a_2$, which is the sum S_2.
   - p_2 computes $a_3 + a_4$, which isn't an actual sum, but we will still save it in R_4.

- p_3 computes $a_5 + a_6$, which isn't an actual sum, but we will still save it in R_6.
- p_4 computes $a_7 + a_8$, which isn't an actual sum, but we will still save it in R_8.

2. At the second iteration:
   - p_1 computes $S_2 + a_3$, which is the sum S_3.
   - p_2 computes $S_2 + R_4 = S_2 + a_3 + a_4$, which is the sum S_4.
   - p_3 computes $R_6 + a_7 = a_5 + a_6 + a_7$, which isn't an actual sum, but we'll still save it in R_7.
   - p_4 computes $R_6 + R_8 = a_5 + a_6 + a_7 + a_8$, which isn't an actual sum, but we'll save it in R_9.
3. At the third iteration:
   - p_1 computes $S_4 + a_5$, which is the sum S_5.
   - p_2 computes $S_4 + R_6 = S_4 + a_5 + a_6$, which is the sum S_6.
   - p_3 computes $S_4 + R_7$, which is the sum S_7.
   - p_4 computes $S_4 + R_9$, which is the sum S_8.

Since we are using all the processors, the efficiency of the algorithm is 1, however, the work is higher since we are doing as many additions as the size of the input data.

2.3 Variants of the PRAM model

The PRAM model is rather abstract, hence one might want to build a more real model. In particular we can:

- Fix a bounded number of shared memory cells. This kind of PRAM machines are called **small memory PRAMs**. If the input data set exceeds the capacity of the shared memory, i/o values can be distributed evenly among the processors.
- Fix a bounded number of processors in a small PRAM. If the number of threads of execution is higher, processors may interleave several threads.
- Fix a bounded size of a machine word.
- Handle access conflicts putting constraints on simultaneous access to shared memory cells.

The following lemma tells us how does the execution time change when we decide to use a number of processors p' smaller than the optimal number of processors p.

Theorem 2.1 (Processor performance downscaling)**.** *Assume $p' < p$. Any problem that can be solved by a p processor PRAM in t steps can be solved by a p' processor PRAM in*

$$t' = \mathcal{O}\left(\frac{tp}{p'}\right) \tag{2.2}$$

steps (assuming same size of shared memory).

To prove this lemma we can partition the computation done by p processors in p' groups of size $\frac{p}{p'}$ each. Then we can associate each group to one of the p' processors. Each of the p' processors simulates one step of its group by

1. Executing all the reads and local computation.
2. Executing the writes.

A similar lemma can be stated for memory.

Theorem 2.2 (Memory performance downscaling)**.** *Assume $m' < m$. Any problem that can be solved for a p processor, m-cell PRAM in t steps can be solved in a* $\max(p, m')$ *processor, m'-cell PRAM in*

$$t' = \mathcal{O}\left(\frac{tm}{m'}\right) \tag{2.3}$$

steps.

The demonstration is similar to the lemma related to processors.

2.3.1 Implementation

The PRAM model is used to give an abstract description of the a parallel algorithm's complexity, however one might wander if it would be possible to implement such a machine.

The PRAM model used a single-program-multiple-data approach in which each processor is executing the same instruction but on different data. This is how parallel algorithm have initially been toughs. This is also the easiest way to describe a parallel algorithm.

A problem one might encounter in the implementation of a PRAM model is the handling of concurrent writes.

2.4 Amdahl's law and Gustafson's law

2.4.1 Amdahl's law

Amdahl's law is used to understand if, the number of processors p we have is suitable to solve a certain problem. Amdahl's law is a model that divides a program in two parts:

- A parallelised part.
- A non-parallelised part. In this part we can use only one processor.

These parts are interleaved, hence a non-parallelisable part is executed first and then the parallelisable part is executed exploiting parallelism. This sequence is repeated until the program terminates. Amdahl also assumes that the fraction of parallelisable code is the same, whatever the input. Once we know the fraction of the program which can be parallelised, the law tries to compute how fast we can go, provided we have enough processors. If we call f the fraction of program which can be parallelised, then the execution time T can be written as

$$T = fT + (1 - f)T$$

If we can use p processors to parallelise the parallelisable part the execution time becomes

$$T_p = \frac{fT}{p} + (1-f)T$$

We can use this result to compute the speed up, which, assuming $T_1 = T^*$, is

$$SU(p,f) = \frac{T_1}{T_p} = \frac{T_1}{T_1 \cdot (1-f) + \frac{T_1 \cdot f}{p}} = \frac{1}{(1-f) + \frac{f}{p}}$$

Amdal's law fixes the fraction of parallelisable code f and the input size, hence the speed up time depends only on the number of processors p. This means that, once we have f and we fix a certain speed-up, we can compute the number of processors we need.

If we consider an infinite number of processors, that is, if we have enough processors, the speed up is

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

This result is very powerful since it can be used to compare the speedup obtained by a PRAM with p processors to understand what is the best number of processors p.

2.4.2 Gustafson's law

People building parallel architectures started realising that Amdahl's law wasn't the most precise. One of them, Gustafson, proposed an alternative law to derive the number of processors required for a certain algorithm. Gustafson changed the assumption that the fraction f of parallelised program is fixed. Better said, he realised that when we have more data, the fraction of parallelisable data increases. This means that for big-data applications, it's important to have multi-core systems. For this reason, Gustafson shifts the focus from the fraction of time used by parallel code to the time used by parallel code.

If we call s the fixed serial time spent by the application executing serial instructions and $1-s$ the fixed parallel time spent by the application executing parallel instructions then a serial machine requires

$$T_1 = s + p(1-s)$$

If we can use p processors, then we can reduce the parallelisable part by p and the execution time is

$$T_p = s + (1-s)$$

This means that the speedup, computed with Gustafson's law, is

$$SU(p) = \frac{T_1}{T_p} = \frac{s + p \cdot (1-s)}{s + (1-s)} = s + p \cdot (1-s)$$

Basically, Gustafson's law doesn't go against Amdahl's, it's just based on another assumption. In fact, Gustafson's law doesn't consider a fraction of the code to be parallelisable but only the time used by parallel code.

In particular, Amdahl's law presupposes that the computing requirements will stay the same, given increased processing power. in other words, an analysis of the same data will take less time given more computing power. Gustafson, on the other hand, argues that more computing power will cause the data to be more carefully and fully analysed.

Part II

Advanced algorithms

Chapter 3

Randomised algorithms

3.1 Introduction

Algorithms are deterministic, which means that given an input of size n, the number of instruction executed is fixed (and depends on n). In some cases, an algorithm might include some random element (e.g., if a random number is greater than 1 do this, otherwise do that). In these cases, we can't apply the same techniques we saw for deterministic algorithms. Randomised algorithms (i.e., algorithms that contain random events) are very useful since

- Sometimes they can achieve better performance than their deterministic counterpart.
- They work well with high probability on every input. In particular, the probability that the average case happens is so high that the worst case is extremely rare. We can also use randomisation to shuffle the input data so that we don't fall in the worst case scenarios.
- They fail on every input with low probability.
- They don't require very complex data structures.

The first idea one might have is to analyse the problem statistically. Namely, we have to assume a distribution of the inputs and then we apply statistical analysis. This is however not easy to do since we might miss some worst cases.

3.1.1 Random variables

In the analysis of randomised algorithms we will use random variables. Say we have a random variable X, which is a composite of many random events that happen in the program. A random variable is a collection of indicator variables X_i, each of which focuses on a specific event. Typically we have that

$$X = \sum X_i$$

Talking about random variables, let us refresh the linearity of expectation that states that, given three random variables X, Y and Z such that $X = Y + Z$, then the expected value of X is the sum

of the expected values of Y and Z.

$$\begin{aligned} E[X] &= E[Y + Z] \\ &= E[Y] + E[Z] \end{aligned}$$

3.2 Las Vegas and Monte Carlo algorithms

Randomised algorithms can be divided in two classes:

- **Las Vegas algorithms**. Las Vegas algorithms depend on the realisations of random variables, however the final result computed by the algorithm is always correct.
- **Monte Carlo algorithms**. Monte Carlo algorithms depend on the realisations of random variables, however they might get to an incorrect solution. For instance, if we have to compute the minimum of a function, we might get close to the minimum, without reaching it. Namely, the algorithm might do some error with respect to the actual solution. Luckily, if we give more time to the algorithm, we might be able to converge to the desired solution.

One might think that the behaviour of a Monte Carlo algorithm is not desired, however in some contexts it's very useful to be able to stop the execution of an algorithm after some time, even if it hasn't ended. In general, we can't say which of the two classes is better, since it depends from application to application.

Note that Las Vegas algorithms can be seen as a special class of Monte Carlo algorithms in which the error's probability is 0.

3.2.1 Monte Carlo algorithms

First of all, let us consider Monte Carlo algorithms for decision problems, i.e., for problems that require a yes/no output. For decision problems, we can recognise Monte Carlo algorithms with:

- **One-sided error**. Algorithms with one-sided error can do a mistake only for one type of output. For instance, if the output for a given input is "yes", then the algorithm will correctly output "yes". If the output is "no", the algorithm might output "yes" or "no".
- **Two-sided error**. Algorithms with two-sided error can do mistakes for both type of answers.

3.2.2 Efficiency

We can define the concept of efficiency to compare the two classes of algorithms. Let's start by Las Vegas algorithms.

> **Definition 3.1** (Efficiency (Las Vegas algorithm))**.** *A Las Vegas algorithm is said to be efficient if its expected running time (i.e., the average execution time) is bounded by a polynomial function of the input size, whatever the input might be.*

Since a Monte Carlo algorithm doesn't always provide the right answer, the definition of efficiency is slightly different.

Definition 3.2 (Efficiency (Monte Carlo algorithm))**.** *A Monte Carlo algorithm is said to be efficient if the worst-case running time is bounded by a polynomial function of the input size, whatever the input might be.*

Let us consider the following example to understand the differences between these two algorithm. Say we want to find one character 'a' in an array of n elements. Let us also consider an input array, whose first half is filled with 'a's, and the second half is filler with 'b's. Let's start by the simplest between the two classes and write a Las Vegas algorithm. The algorithm, shown in Algorithm 7, simply picks a value at random in the input array, until such value is an 'a'. This algorithm, given that the searched element is in the array, succeeds with probability 1. If we consider the input array with half 'a's and half 'b's, we can be confident to find, on average, an 'a' in constant time.

Algorithm 7 A Las Vegas algorithm to find a value in an array.

```
Require: array A
Require: n
  c ← 'b'
  while c ≠ 'a' do
      c ← Randomly select one element of A
  end while
```

Now it's time to write the Monte Carlo algorithm to solve the search problem. The approach is basically the same, in fact we have a loop in which, at each iteration, we pick an element at random and we check if it's what we were looking for. The main difference is that we stop the algorithm either when we find the element or after k iterations. The algorithm is summed up in Algorithm 8. Note that in this case, we don't have guarantees on the fact that the algorithm will return the right

Algorithm 8 A Monte Carlo algorithm to find a value in an array.

```
Require: array A
Require: n
Require: k
  i ← 0
  c ← 'b'
  while c ≠ 'a' and i < k do
      c ← Randomly select one element of A
      i ← i + 1
  end while
```

result since after k iterations, we might have visited only a subset of the array. After k iterations, the probability of finding an 'a' is

$$\begin{aligned} Pr(\text{find 'a'}) &= 1 - Pr(\text{not find 'a'}) \\ &= 1 - \frac{1}{2^k} \end{aligned}$$

Note that the probability of not finding an 'a' in k iterations can be computed as the product, k times, of $\frac{1}{2}$ since half of the array is filled with 'a's, hence we have a probability of $\frac{1}{2}$ of not finding an 'a' in the array. This probability gets closer to 1 if we increase the value of k (which makes sense

because if we increase the number of iterations, we visit more elements). On the other hand we are guaranteed to end the execution in k iteration, hence the worst-case runtime is

$$T_{MC}(n) = \mathcal{O}(k)$$

Note that we have considered the worst case because we are analysing a Monte Carlo algorithm. Moreover, we can see that k can be used as a parameter to trade-off time complexity for success probability (the bigger k, the more probable it is to get a correct result, but the bigger is the time complexity).

3.3 Karger's min-cut algorithm

Let's start by stating the problem. Let $G = (V, E)$ be a undirected connected graph with $n = |V|$ vertices and $m = |E|$ edges. Given a subset $S \subset V$ of vertices, we can define a subset of edges

$$\delta(S) = \{(u, v) \in E : u \in S, v \in S', S \cap S' = \varnothing\}$$

called **cut**, since, if we remove the arcs in $\delta(S)$ from G, we get two graphs which are not connected anymore (i.e., from a point in S we can't reach a point in S'). We can usually define many cuts in a graph, however we would like to find the one with the minimum number of edges.

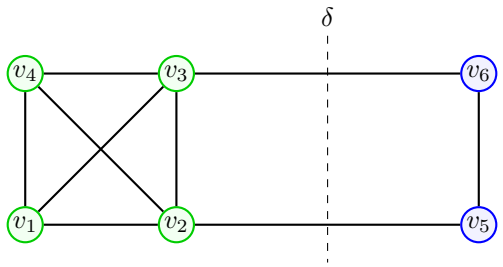


Figure 3.1: A cut in a graph.

3.3.1 Classic approach

To solve the min-cut problem we can solve another problem, called min-s,t-cut problem, which is the min-cut problem when we select two vertices $s \in S$ and $t \in S'$. We can therefore fix a vertex t and solve the min-s,t-cut problem for every other $s \in V$ and then keep the cut with minimum number of arcs. This means that we have reduced the min-cut problem to the min-s,t-cut problem, which can be solved as a max-s,t-flow problem. The best (i.e., fastest) algorithm that solves the max-s,t-flow problem has a time complexity of

$$T_{\text{max-st-flow}}(n, m) = \mathcal{O}\left(nm \log\left(\frac{n^2}{m}\right)\right)$$

Since we have to solve the max-s,t-flow problem $n-1$ times (fix t, try the other $n-1$ vertices s) and every problem is independent from the other, we can also paralleise the procedure.

3.3.2 Karger's approach

Karger used a different approach with respect to the classical one. In particular, instead of using the max-flow problem, he used randomisation.

The first thing Karger's algorithm does is to build a multiedged graph without loops from the input graph. Note that the multiedged graph, as the one shown in Figure 3.2, might have multiple undirected edges connecting any two vertices.

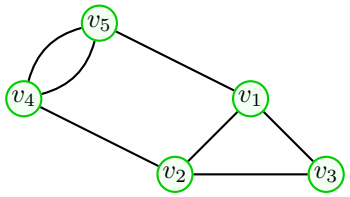


Figure 3.2: A multiedged graph.

In a graph one can contract edges, that is, join the points connected by an edge and create a new vertex. Formally, given a multigraph (i.e., a multiedged graph) $G = (V, E)$ without self loops and and edge $e = (u, v) \in E$, the contraction with respect to e, denoted G/e, is formed by:

1. Replacing vertices u and v with a new vertex w.
2. Replacing edges (u, x) and (v, x) with new edges (w, x).
3. Removing any self loop.

Say for instance we want to contract the edge (v_1, v_2) in Figure 3.2. We have to create a new vertex v_{12} and connect every vertex previously connected to v_1 or v_2 to the new vertex v_{12}. The resulting graph is shown in Figure 3.3.

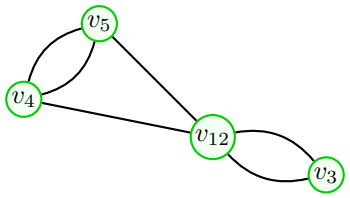


Figure 3.3: A multiedged graph with a contracted edge.

Note that a contracted arc has the same cuts as the non-contracted one, except for the cuts that contain the contracted edge. This comes from the fact that we are considering a multiedged graph. Thanks to this property the minimum cut doesn't change, if we contract an edge not in the minimum cut. The idea is therefore to repeat the contraction process until we only have two vertices. At this point, the cut is simply the set of edges left in the graph. The cut might not be minimum, hence we have to repeat the process until we find a cut with the minimum number of edges. More precisely, the algorithm:

1. Randomly picks an edge e.
2. Contracts e.

3. Repeat from point 1, until we are left with only two vertices (i.e., $n-2$ times).

This algorithm takes $\mathcal{O}(n^2)$ since, for $n-2$ times, we have to rewrite the edges involving n vertices. Note that the cut obtained by this algorithm isn't necessarily the minimum one, however we have a probability

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

that the cut is actually a minimum one. This means that we can repeat the algorithm many times until until, we have tried every possible iteration (Las Vegas algorithm) or we stop early (Monte Carlo algorithm).

Before going on, let us analyse why we get the probability shown above. First, let us call k the size of one of the minimum cuts and

$$\{e_1, e_2, \ldots, e_{n-2}\}$$

the edges we have contracted, in the order in which they have been merged. The probability that we have found a minimum cut is the probability that each e_i isn't in the min-cut $\delta(S)$. To upper bound this probability, we can consider the first iteration and upper bound the probability that the algorithm chooses an edge $e_1 \notin \delta(S)$. To do so, we need to lower bound the number of edges in the input graph in terms of k. If we assume that $|\delta(S)| = k$, then the minimum number of edges exiting from a node is k (otherwise, if e_a had $k-1$ incident edges, we could create a partition $S = e_a$ and the cut would have $k-1$ edges). If we assume that each vertex has k incident edges and we have n vertices, then the total number of edges is at least $\frac{nk}{2}$ (we take half of nk because we consider only one edge between any two nodes). If we contract the second edge e_2, and we assume it isn't in the cut, then the cut still has k edges, and the minimum number of edges in the graph is at least $\frac{(n-1)k}{2}$ since, after merging two vertices in the first iteration, we have $n-1$ vertices. If we iterate this process, assuming that each e_j isn't in the cut, we know that, after j contractions, the graph has at least

$$|E^j| \geq \frac{(n-j)k}{2}$$

edges.

The probability of finding a minimum cut is the probability that, at each choice, the algorithm

picks an edge not in the minimum cut, namely

$$\begin{aligned}
Pr(\text{cut is minimum}) &= Pr(e_1 \notin \delta(S) \wedge \cdots \wedge e_{n-2} \notin \delta(S)) \\
&\geq \prod_{j=0}^{n-3} 1 - \frac{k}{\frac{(n-j)k}{2}} \\
&\geq \prod_{j=0}^{n-3} 1 - \frac{2}{n-j} \\
&\geq \prod_{j=0}^{n-3} \frac{n-j-2}{n-j} = \frac{n-2}{n} \cdot \frac{n-3}{n-1} \cdots \frac{2}{4} \cdot \frac{1}{3} \\
&\geq \frac{(n-2)!}{\frac{n!}{2!}} \\
&\geq \frac{\frac{n!}{n(n-1)}}{\frac{n!}{2}} \\
&\geq \frac{2}{n(n-1)} = \frac{1}{\frac{n(n-1)}{2!}} = \frac{1}{\frac{n!}{2!(n-2)!}} = \frac{1}{\binom{n}{2}}
\end{aligned}$$

If we repeat the algorithm a number of times equal to $l\binom{n}{2}$, then the probability that at least one run succeeds is

$$\begin{aligned}
Pr(\text{at least 1 out of } l\binom{n}{2} \text{succeeds}) &= \left(1 - \frac{1}{\binom{n}{2}}\right)^{l\binom{n}{2}} \\
&>= 1 - e^{-l}
\end{aligned}$$

If we now consider $l = c \log n$, then we get an error probability of

$$Pr(\text{not succeeding in } c \log n \binom{n}{2} \text{ runs}) \leq \frac{1}{n^c}$$

where c is a constant value which allows us to control the probability of having an error (we are considering the Monte Carlo algorithm). Moreover, since a run is repeated $c \log n \binom{n}{2} = c \log n \frac{n!}{2!(n-2)!}$ times, the algorithm complexity is

$$T_{\text{min-cut}}(n) = \mathcal{O}\left(n^2 \cdot c \log n \cdot \frac{n(n-1)}{2}\right) = \mathcal{O}(n^4 \log n)$$

Note that this complexity isn't that far from the one obtained with the classical approach.

3.3.3 Karger and Stein's algorithm

Karger and Stein proposed a faster version of Karger's algorithm, obtained by modifying the contraction method. They realised that the probability of picking the wrong edge (i.e., one in the cut) was quite high after some iterations, hence they defined a certain threshold τ. The algorithm is the same for the first τ iteration while the iterations after the threshold are modified. In particular, they discovered that the probability that a cut survives after l iterations is at least $\frac{\binom{l}{2}}{\binom{n}{2}}$, hence if we

consider $l = \frac{n}{\sqrt{2}}$ we get a probability bigger than $\frac{1}{2}$ of succeeding. For this reason, when we reach $\frac{n}{\sqrt{2}}$ iterations, we can recursively execute the algorithm twice (the probability of getting the right cut is 0.5), until we reach a point in which we can compute the min-cut in constant time. The complete algorithm is shown in Algorithm 9.

Algorithm 9 Karger and Stein's algorithm for the min-cut problem.

```
procedure CONTRACT(G = (V, E), t)
    while |V| > t do
        e ← choose edge in E at random
        G ← G/e
    end while
end procedure
procedure FASTMINCUT(G = (V, E))
    if |V| < 6 then
        return MINCUT(V)
    else
        t ← ⌈1 + |V|/√2⌉
        G_1 ← CONTRACT(G, t)
        G_2 ← CONTRACT(G, t)
        return min{FASTMINCUT(G_1), FASTMINCUT(G_2)}
    end if
end procedure
```

Running time

Since Karger and Stein's algorithm uses recursion, we can use the Master theorem to compute its execution time. In particular, the execution time can be written as

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

where

- $a = 2$
- $b = \sqrt{2}$
- $f(n) = 2n^2$

Since $\log_b a = \log_{\sqrt{2}} 2 = 2$, then we are in case 2 of the Master theorem with $k = 0$. We can therefore compute the complexity of the Karger and Stein's algorithm as

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

Success probability

The success probability can be computed with the following recurrence

$$Pr(n) \geq 1 - \left(1 - \frac{1}{2} Pr\left(\frac{n}{\sqrt{2}} + 1\right)\right)^2$$

with solution

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

Finding all cuts

Note that, since every graph has at most n^2 minimum cuts, we can use Karger's algorithm to compute all the cuts in a graph.

3.4 Quicksort

Quicksort is a divide-and-conquer algorithm used for sorting an array in-place. In particular, quicksort:

1. Splits the array A in two parts A_1 and A_2. In particular:
   (a) It picks an element x called *pivot*.
   (b) It puts all elements smaller than x before x (i.e., in A_1) and the elements greater than x after x (i.e., in A_2).
2. Recursively sorts the left and right sub-arrays A_1 and A_2.
3. Places A_1 before x and A_2 after x.

The key part of the algorithm is the divide step, in fact the algorithm is efficient only if we can partition the array in linear time. An example of partition algorithm is shown in Algorithm 10. The full algorithm is shown in Algorithm 11

Algorithm 10 Lomuto partition algorithm.

```
procedure PARTITION(A, p, q)
    x ← A[p]                                   ▷ x is the pivot
    i ← p
    for j ← p + 1, ..., q do
        if A[j] ≤ x then
            i ← i + 1
            SWAP(A[i], A[j])
        end if
    end for
    SWAP(A[p], A[i])
    return i
end procedure
```

3.4.1 Complexity of the classical algorithm

The best case complexity is $\mathcal{O}(n \log n)$ and it's the case in which we can always split the array in half. However, if we consider the worst case, we get a quadratic complexity.

$$T_{\text{quicksort, worst-case}} = \mathcal{O}(n^2)$$

Algorithm 11 The quicksort algorithm.

```
procedure QUICKSORT(A, p, r)
    if p < r then
        q ← PARTITION(A, p, r)
        QUICKSORT(A, p, q − 1)
        QUICKSORT(A, q + 1, r)
    end if
end procedure
```

In particular, the worst case is when the array is already sorted (or reverse sorted) because in this case the partitioning splits the array A in an empty array $A_1 = \{\}$ on the left and the whole array $A_2 = A$ on the right. This means that we have to execute n partitions (and not $\log n$), hence the complexity is n^2. Formally, we can write

$$\begin{aligned} T(n) &= T(0) + T(n-1) + n \\ &= \mathcal{O}(1) + T(n-1) + n \\ &= T(n-1) + n \end{aligned}$$

Let us now consider a solution in between the best and worst case. Say for instance we can always split the array so that A_1 contains $\frac{1}{10}$ of A and A_2 contains the remaining $\frac{9}{10}$. In this case, the execution time can be expressed as

$$T(n) = T\left(\frac{1}{10}n\right) + T\left(\frac{9}{10}n\right) + n$$

The longest part to compute is the one of A_2, since it has to work on a longer array. If we keep this in mind and we write the recursion tree, we can see that the depth of the tree is $\log_{\frac{10}{9}} n$, hence we get a total complexity of

$$T(n) = \mathcal{O}\left(n \log_{\frac{10}{9}} n\right)$$

This isn't as good as the best case ($T_{\text{best}} = \mathcal{O}(n \log_2 n)$), however if we remember the formula for the logarithm base change $\log_c a = \frac{log_b a}{\log_b c}$, we understand that we can go from $\log_{\frac{10}{9}} n$ to $\log_2 n$ multiplying by dividing the former for a constant, hence we have the same asymptotic complexity in both cases. This means that even when the partition algorithm creates heavily unbalanced partitions, we still get the best case complexity. We get the same complexity even if we consider the situation in which alternatively the array is split in two halves and then in the worst-case partition.

3.4.2 Randomised quicksort

This means that there are many cases in which we get a complexity of $n \log n$. The randomised quicksort algorithm tries to partition the array so that the complexity is always $n \log n$. To achieve this goal, the algorithm uses a slightly different partition function that picks a random pivot element instead of using the first element of the array. Picking the item at random has two important effects:

- No assumption has to be made about the input distribution.
- No specific input leads to the worst case scenario.

Complexity

To analyse the complexity of the randomised quicksort we can use a random variable $T(n)$ which indicates the running time of the randomised quicksort on an input size n, assuming independent identically distributed random numbers. Let us also consider the indicator random variables

$$X_k = \begin{cases} 1 & \text{if PARTITION generates a } k : n-k-1 \text{ split} \\ 0 & \textit{otherwise} \end{cases} \qquad \forall k = 0, \ldots, n-1$$

Since we have n possible partitions, then the expected value of X_k is

$$E[X_k] = Pr(X_k = 1) \cdot 1 + Pr(X_k = 0) \cdot 0 = \frac{1}{n} \cdot 1 + \frac{1}{n} \cdot 0 = \frac{1}{n}$$

since the probability that $A[k]$ is the pivot is $\frac{1}{n}$. Since we can have multiple partitions, we have to list all the possible partitions in the recursive execution time, which can therefore be written as

$$T(n) = \begin{cases} T(0) + T(n-1) + n & \text{if } 0 : n-1 \text{ split} \\ T(1) + T(n-2) + n & \text{if } 0 : n-2 \text{ split} \\ \vdots \\ T(n-1) + T(0) + n & \text{if } n-1 : 0 \text{ split} \end{cases}$$

At every iteration, only one of the options is chosen (depending on the random choice), hence we can compute the total time variable $T(n)$ as the sum of each execution time, multiplied by the probability X_k of happening.

$$T(n) = \sum_{k=0}^{n-1} X_k \cdot \big(T(k) + T(n-k-1) + n\big)$$

Now that we have defined this variable, we can compute the expected value of the execution time $T(n)$.

$$\begin{aligned} E[T(n)] &= E\Big[\sum_{k=0}^{n-1} X_k \cdot \big(T(k) + T(n-k-1) + n\big)\Big] \\ &= \sum_{k=0}^{n-1} E\big[X_k \cdot \big(T(k) + T(n-k-1) + n\big)\big] \\ &= \sum_{k=0}^{n-1} E\big[X_k\big] \cdot E\big[T(k) + T(n-k-1) + n\big] \\ &= \sum_{k=0}^{n-1} \frac{1}{n} E\big[T(k) + T(n-k-1) + n\big] \\ &= \frac{1}{n} \sum_{k=0}^{n-1} E\big[T(k) + T(n-k-1) + n\big] \\ &= \frac{1}{n} \sum_{k=0}^{n-1} E\big[T(k)\big] + \frac{1}{n} \sum_{k=0}^{n-1} E\big[T(n-k-1)\big] + \frac{1}{n} \sum_{k=0}^{n-1} E\big[n\big] \end{aligned}$$

If we notice that the partition $k : n-k-1$ has the same execution time as the partition $n-k-1 : k$, we can write $E\big[T(n-k-1)\big]$ as $E\big[T(k')\big]$ for $k' = n-k-1$, hence we can count $E\big[T(k')\big]$ twice. What we obtain is:

$$\begin{aligned} E[T(n)] &= \frac{2}{n}\sum_{k=1}^{n-1} E\big[T(k)\big] + E\big[n\big] \\ &= \frac{2}{n}\sum_{k=1}^{n-1} E\big[T(k)\big] + n \end{aligned}$$

Thanks to some statistics magic we get to

$$\begin{aligned} E[T(n)] &= \frac{2}{n}\sum_{k=1}^{n-1} E\big[T(k)\big] + E\big[n\big] \\ &\leq a \cdot n \cdot \log n \end{aligned}$$

This means that the time complexity of the randomised quicksort is the same as the one of the classical quicksort, in the best case.

3.4.3 Quicksort in practice

Quicksort is widely used in practice since:

- It's typically twice as fast as merge sort.
- It can benefit from code tuning.
- It can exploit caching.
- It behaves well even with caching and virtual memory.

3.4.4 Sample sort

Quicksort is very fast on average, however, if we want to parallelise it, it has a big bottleneck at the beginning of the recursion when we partition the vector. Sample sort solves this problem by proposing a partition algorithm which can be parallelised, too (the remaining of the algorithm is the same). The idea is to split the input vector into many partitions and then sort each partition in parallel using some serial algorithm for each partition. More precisely, the algorithm can be divided in three phases:

1. **Bin**. In this phase we choose a number $m-1$ of pivots and we build a $m \times m$ matrix. In particular, we split the input into m chunks and we put each chunk on a row. Now we can split each row in m bins, each containing a subset of the elements of the chunk. At this point we can assign each row to a thread that sorts the elements in the tree bins according to the pivots. This means that (for each row) the first bin will contain the values smaller than the first pivot, the second those bigger than the first pivot but smaller than the second and so on.

2. **Repacking**. In this phase we transpose the matrix by putting on the column the values on the rows. This can be done in parallel using the pack pattern.

3. **Subsort**. In this phase we can sort each row recursively (or using a serial algorithm if the vector is small enough). As last step, we only have to concatenate the vectors computed by each thread since the first row contains all elements smaller than the first pivot, the second those bigger than the first pivot and smaller than the second and so on.

This way of splitting the input vector is more expensive, in fact its cost is

$$T_{split}(n, m) = \mathcal{O}(n \log m)$$

however we can split the work among m threads and parallelise it. What we obtain is a complexity of

$$T_{split,parallel}(n, m) = \mathcal{O}(n \log \frac{n}{m})$$

3.5 Linear time sorting

3.5.1 Bound to the time complexity of sorting algorithms

We have shown that even a randomised algorithm can't do better than the best classical algorithm. One might think that

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

is the best execution time we can obtain for a sorting algorithm. To demonstrate this hypothesis we can consider a decision tree. Say we have an array

$$A = [a_1, a_2, \ldots, a_n]$$

we want to sort. We can build a tree where each node represents the comparison between two elements of the array. Given a comparison $a_i : a_j$, the sub-tree on the left considers the case in which $a_i < a_j$ while the sub-tree on the right considers the case in which $a_i \geq a_j$. After having considered all possible comparisons, we obtain a tree where each leaf is a possible order of the elements. If we consider for instance an array $A = [a_1, a_2, a_3]$ with three elements we obtain the three in Figure 3.4.

Given an instance of an array, we can order it, starting from the root of the tree, with the following procedure:

1. Consider the node $a_i : a_j$.
2. Compare a_i and a_j and
   - If $a_i < a_j$, go to the node on the left.
   - If $a_i \geq a_j$, go to the node on the right.
3. Repeat from point 1, until a leaf isn't reached.

If we want to order the array $A = [9, 4, 6]$ we:

1. Start from node $a_i : a_2$. Compare 9 and 4 and since $9 \geq 4$ go to node $a_1 : a_3$, on the right.
2. Consider node $a_1 : a_3$. Compare 9 and 6 and since $9 \geq 6$ go to node $a_2 : a_3$, on the right.

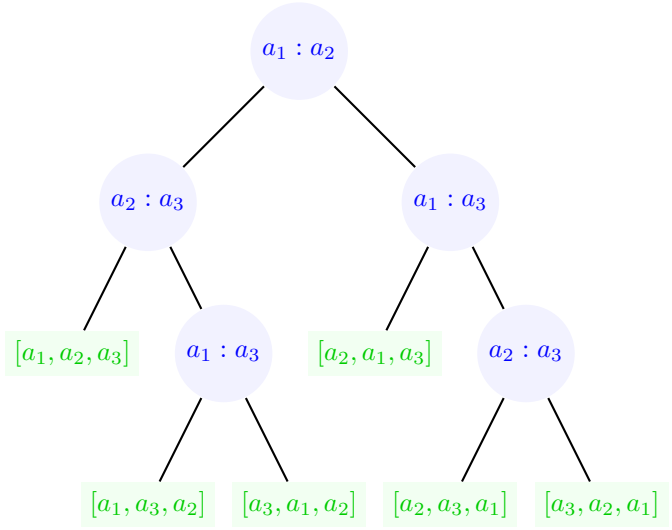


Figure 3.4: The decision tree for an array of three elements.

3. Consider node $a_2 : a_3$. Compare 4 and 6 and since $4 < 6$ go to node $[a_2, a_3, a_1] = [4, 6, 9]$, on the left.

Since the leafs of the decision tree represent every possible combination of n elements, we have at least $n!$ leafs. Moreover, since the tree is binary, it can't have more than 2^h leafs, where h is the height of the tree. This means that we can write

$$\begin{aligned} n! \leq 2^h \iff h &\geq \log(n!) \\ &\geq \log\left(\left(\frac{n}{e}\right)^n\right) \\ &= n\log\left(\frac{n}{e}\right) \\ &= n\log n - n\log e \\ &= \Theta(n\log n) \end{aligned}$$

where, since the logarithm is monotonically increasing, we have used Stirling's formula

$$n! \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$

This shows that a sorting algorithm can't be faster than $\Theta(n\log n)$.

3.5.2 Counting sort

At this point one might think that it's impossible to sort an algorithm in linear time. However, we have demonstrated that an algorithm that compares values can't sort an array in less than $\Theta(n\log n)$. This means that, if we find an algorithm capable of sorting an array without doing any comparison, we might obtain, with some restrictions, a linear complexity.

The first algorithm that can achieve this goal is called counting sort. In this case, the input array can't be any array, but its values must range from 1 to k. The algorithm is made of different stages:

1. In the first stage we initialise an auxiliary array C of k elements with all 0s.
2. In the second stage we pass the input array and for each element $A[i]$, we increase by 1 the value of C in position $A[i]$. Since A contains values from 1 to k and C contains has length k (with indices starting from 1), we never go out of bounds. Basically, $C[i]$ is counting how many elements i are there in A.
3. In the third stage we update C. In particular, we put in $C[i]$ the last position in which we should put element i. This works because $C[1]$ contains for sure the last position where element 1 should go. If $C[1] = 3$, the last element 1 goes in position 3.
4. In the fourth stage we put the elements in the output array B. In particular, for all elements $A[i]$ in A (starting from the end), we put $A[i]$ in the position specified by the array C in $C[A[i]]$. The position in C is then decreased.

The algorithm is represented in Algorithm 12. A good property of this algorithm is that the relative order of the elements is preserved.

If we assume that $n \gg k$, the execution time is dominated by the second and third set, which take both n steps, hence the total complexity of this algorithm is linear.

$$T_{\text{counting sort}} = \Theta(n)$$

Counting sort is also a stable sorting algorithm, namely elements with the same value are put in the ordered array in the same order as they were in the input array.

Algorithm 12 The counting sort algorithm.

Require: A
Ensure: $B : B[i] \leq B[j] \iff i < j \quad \forall i, j$
  for $i \leftarrow 1, ..., k$ **do** ▷ phase 1
    $A[i] \leftarrow 0$
  end for
  for $i \leftarrow 1, ..., n$ **do** ▷ phase 2
    $C[A[i]] \leftarrow C[A[i]] + 1$
  end for
  for $i \leftarrow 2, ..., k$ **do** ▷ phase 3
    $C[i] \leftarrow C[i] + C[i-1]$
  end for
  for $i \leftarrow n, ..., 1$ **do** ▷ phase 4
    $B[C[A[i]]] \leftarrow A[j]$
    $C[A[i]] \leftarrow C[A[i]] - 1$
  end for

3.5.3 Radix sort

Radix sort uses the digits of the values, instead of the values themselves, to sort the array. The original idea was to sort the items using the most significant digit, however this isn't a correct idea.

Now we know we should sort the numbers starting from the least significant digit. Moreover we should use an auxiliary sorting array to sort the values and preserving the order of the same values (i.e., performing stable sorting). This algorithm is able to sort a huge amount of data, without having restrictions on the possible values contained in the array.

Algorithm

Note that, since we want to sort only one digit at a time, we can use counting sort to sort that digit (a digit has values from 0 to 9). This means that the auxiliary structure is an array of 10 elements only. Moreover, since counting sort preserves the order of equal values, the output of radix sort, will also preserve such order. Counting sort is therefore applied to sort iteratively the array, starting from the least significant digit to the most significant one. Namely, we sort the array using as sorting criteria the last significant digit. The result is sorted using as criteria the last-but-one significant digit. This process goes on until we reach the most significant digit (which is also used as criteria for sorting).

Correctness

The correctness of this algorithm can be proved by induction. If we assume that the values are sorted by their low-order $t-1$ digits, we can sort the values using their t-th digit. Two numbers that differ in digit t are correctly sorted (799 is smaller than 800, because 7 is smaller than 8, independently from what we have after that digit).

Complexity

Let us now analyse the complexity of radix sort. Let us assume an input array of n values, each represented in binary with b bits. Note that we can also use higher basis (e.g. base 10) and obtain a bigger auxiliary array, but fewer iterations. In general, the higher the base, the bigger the auxiliary array and the lower the number of iterations. We can split the array in section of r bits, and use each section for sorting. Namely, we aren't sorting using a single bit, but using r bits (e.g., 8 bits). This means that the auxiliary array should have 2^r entries and we will execute $\frac{b}{r}$ iterations of the counting sort algorithm.

Since counting sort has a complexity of $\mathcal{O}(n+k)$, where k is the range of values in the array, and the algorithm is repeated $\frac{b}{r}$ times, we obtain a complexity of

$$T_{\text{radix sort}}(n) = \Theta\left(\frac{b}{r}(n+2^r)\right)$$

Since we can choose the value of n, we can try to optimise it. In particular, we can differentiate the

complexity (with respect to r) and put it to 0 to find out what is its minimum. In formulas

$$\begin{aligned}
\frac{dT_{\text{radix sort}}(n)}{dr} &= 0 \\
\frac{d}{dr}\frac{b}{r}(n+2^r) &= 0 \\
\frac{d}{dr}\frac{b}{r}n + \frac{b2^r}{r} &= 0 \\
-\frac{bn}{r^2} + \frac{b2^r(r\log(2)-1)}{r^2} &= 0 \\
b\big(2^r(r\log(2)-1) - n\big) &= 0 \\
2^r(r\log(2)-1) &= n \\
2^r &= n
\end{aligned}$$

where we have said that $2^r(r\log(2)-1) \sim 2^r$ since we are doing an asymptotic analysis. This means that the optimal value for r is

$$r^* = \log(n)$$

hence the best complexity we can get is

$$T_{\text{radix sort}}(n) = \Theta\left(\frac{b}{\log(n)}(n+2^{\log(n)})\right) = \Theta\left(\frac{b}{\log(n)}\cdot n\right)$$

since $2^{\log_2(n)} = n$. For numbers in the range from 0 to $n^d - 1$ we have that

$$\begin{aligned}
2^b &= n^d \\
\log_2(2^b) &= \log_2(n^d) \\
b &= d\log_2(n)
\end{aligned}$$

hence the complexity is

$$T_{\text{radix sort}}(n) = \Theta\left(\frac{d\log_2(n)}{\log_2(n)}\cdot n\right) = \Theta\Big(dn\Big)$$

In conclusion, the radix sort algorithm has a linear complexity.

Advantages and disadvantages

The main advantages of radix sort are that:

- It can sort an array with a **limited number of iterations** (fewer than quicksort, for instance).
- It has a **linear complexity**.

However, the main downsides are:

- It **doesn't exploit data locality** since values from one side of the array can be placed, especially in the last iterations, in a completely different area of the array. This means that quicksort usually works better in modern processors which extensively exploit data locality.

3.6 Order statistics

3.6.1 Randomised algorithm

Randomisation can be used also to solve the problem of finding the element with a certain rank in an array. This problem receives as input a set of n distinct numbers and an integer $1 \leq i \leq n$ and outputs the element with rank i, namely the element in position i when the element has been sorted. This problem can be solved sorting the array with a good sorting algorithm and then selecting the element in position i. The worst-case running time is therefore

$$T_{\text{order statistics sorting}}(n) = \Theta(n \log n) + \Theta(1) = \Theta(n \log n)$$

Note that the problem is much easier if we consider the cases $i = 1$ and $i = n$. In these cases we want to select the maximum and minimum elements, hence we can simply pass the whole array and find them. In these cases, the complexity is linear. Luckily, we can exploit randomisation to solve the general problem in linear time, too. The algorithm uses a divide-and-conquer approach and recursively analyses the input array (usually, each recursion step analyses a different section of the array). In particular,

- If the indices p and q that mark the start and the end of the array we are analysing have the same value, then we have found the value we where looking for and we return it.
- Otherwise,
    1. We apply the same random partition used for quicksort. This means that we can split the array in two parts, one which contains element smaller than the pivot and the other with elements that are bigger. This means that the position r of the pivot is going to be exactly the one of the ordered array.
    2. We compute the rank of the element in position r as $k = r - p + 1$.
    3. If k is the searched rank, then we return $A[r]$.
    4. Otherwise we apply recursively the algorithm on the right array if the searched rank i is bigger then r, or on the left array if $i < k$.

The algorithm is also shown in Algorithm 13. Note that, differently from quicksort, we search the element only in one of the array generated with the partition. Also note that if we look in the right partition, we have to decrease the value of the rank i, because we have less elements in the partitioned array. Consider, for instance, an array of 11 elements and $i = 7$. Say the array is partitioned in two arrays of 5 elements each and $i > k$, thus we have to visit the right half of the array. In this case we want to look for $i' = i - k = 7 - 5 = 2$ because the array only has 5 elements. We don't have the same problem for the left partition since i is in the indices of the partition (which go from 0 to 4).

To analyse the complexity of this algorithm, we can use the same procedure we used for quicksort. In particular, we can consider an unbalanced partition and assume that the searched element is always in the bigger partition. If the array is split, for instance, in an array with $\frac{9}{10}$-th of the elements and an array with $\frac{1}{10}$-th of the elements. In this scenario we can write the following recursive equation.

$$T_{\text{rand-search}}(n) = T\left(\frac{9n}{10}\right) + \Theta(n)$$

This equation satisfies the conditions of Master theorem's third case. In particular

Algorithm 13 Randomised divide-and-conquer algorithm for the order statistics problem.

```
procedure RAND-SELECT(A, p, q, i)
    if p = q then
        return A[p]
    end if
    r ← RAND-PARTITION(A, p, q)
    k ← r − p + 1
    if i = k then
        return A[r]
    end if
    if i < k then
        return RAND-SELECT(A, p, r − 1, i)
    else
        return RAND-SELECT(A, r + 1, q, i − k)
    end if
end procedure
```

- $a = 1$
- $b = \frac{10}{9}$

hence it's true that

- $\exists c < 1 : 1 \cdot \frac{9n}{10} \leq cn$
- $\Theta(n) = \Omega(n^{\log_{\frac{10}{9}} 1+1}) = \Omega(n^1)$

This means that the solution is automatically

$$T_{\text{rand-search}}(n) = \Theta(n)$$

However, if the partition is completely unbalanced (i.e., the partition on which we develop the recursion has all the elements but one), the complexity is quadratic.

$$T_{\text{rand-search, worst}}(n) = T(n-1) + \Theta(n) = \Theta(n^2)$$

This result can be obtained remembering that computing the partition requires linear time and that, having all unbalanced partitions, we repeat the process n times. Note that the worst-case complexity is worst than the complexity of the non-randomised algorithm which sorts the input array.

As for sorting, we can analyse the expected time. In particular, we can define a random variable $T(n)$ which represents the running time of the algorithm, when fed with an input of size n. Moreover, we can define k indicator random variables

$$X_k = \begin{cases} 1 & \text{if PARTITION generates a } k : n-k-1 \text{ split} \\ 0 & \text{otherwise} \end{cases} \qquad \forall k = 0, 1, \ldots, n-1$$

This means that we can write $T(n)$ as

$$T(n) = \begin{cases} T(\max\{0, n-1\}) + \Theta(n) & \text{if } 0 : n-1 \text{ split} \\ T(\max\{1, n-2\}) + \Theta(n) & \text{if } 1 : n-2 \text{ split} \\ \vdots \\ T(\max\{n-1, 0\}) + \Theta(n) & \text{if } n-1 : 0 \text{ split} \end{cases}$$

Since, for each iteration, only one of the possible equations is used, we can write $T(n)$ as

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

Now that we have an expression for $T(n)$, we can compute its expected value.

$$\begin{aligned} E\big[T(n)\big] &= E\bigg[\sum_{k=0}^{n-1} X_k \cdot \big(T(\max\{k, n-k-1\}) + \Theta(n)\big)\bigg] \\ &= \sum_{k=0}^{n-1} E\Big[X_k \cdot \big(T(\max\{k, n-k-1\}) + \Theta(n)\big)\Big] \end{aligned}$$

Since X_k are independent (identically distributed) from T, we can write

$$\begin{aligned} E\big[T(n)\big] &= E\bigg[\sum_{k=0}^{n-1} X_k \cdot \big(T(\max\{k, n-k-1\}) + \Theta(n)\big)\bigg] \\ &= \sum_{k=0}^{n-1} E\Big[X_k \cdot \big(T(\max\{k, n-k-1\}) + \Theta(n)\big)\Big] \\ &= \sum_{k=0}^{n-1} E\big[X_k\big] \cdot E\Big[T(\max\{k, n-k-1\}) + X_k \cdot \Theta(n)\big)\Big] \end{aligned}$$

As for the quicksort algorithm, we can say that $E[X_k] = \frac{1}{n}$. Moreover, applying the linearity of the expected value we get

$$\begin{aligned} E\big[T(n)\big] &= E\bigg[\sum_{k=0}^{n-1} X_k \cdot \big(T(\max\{k, n-k-1\}) + \Theta(n)\big)\bigg] \\ &= \sum_{k=0}^{n-1} E\Big[X_k \cdot \big(T(\max\{k, n-k-1\}) + \Theta(n)\big)\Big] \\ &= \sum_{k=0}^{n-1} E\big[X_k\big] \cdot E\Big[T(\max\{k, n-k-1\}) + \Theta(n)\big)\Big] \\ &= \frac{1}{n} \cdot \sum_{k=0}^{n-1} E\Big[T(\max\{k, n-k-1\})\Big] + E\big[\Theta(n)\big] \\ &= \frac{1}{n} \cdot \sum_{k=0}^{n-1} E\Big[T(\max\{k, n-k-1\})\Big] + \Theta(n) \end{aligned}$$

We can also recognise the same symmetry we noticed in quicksort, hence we obtain

$$\begin{aligned}
E\big[T(n)\big] &= E\left[\sum_{k=0}^{n-1} X_k \cdot \big(T(\max\{k, n-k-1\}) + \Theta(n)\big)\right] \\
&= \sum_{k=0}^{n-1} E\left[X_k \cdot \big(T(\max\{k, n-k-1\}) + \Theta(n)\big)\right] \\
&= \sum_{k=0}^{n-1} E\big[X_k\big] \cdot E\left[T(\max\{k, n-k-1\}) + \Theta(n)\big)\right] \\
&= \frac{1}{n} \cdot \sum_{k=0}^{n-1} E\left[T(\max\{k, n-k-1\})\right] + E\big[\Theta(n)\big] \\
&= \frac{1}{n} \cdot \sum_{k=0}^{n-1} E\left[T(\max\{k, n-k-1\})\right] + \Theta(n) \\
&<= \frac{2}{n} \cdot \sum_{k=\lfloor \frac{n}{2} \rfloor}^{n-1} E\big[T(k)\big] + \Theta(n)
\end{aligned}$$

Thanks to some magic we get to

$$E\big[T(n)\big] \leq cn - \left(\frac{cn}{4} - \Theta(n)\right)$$

If we choose c such that the term between parenthesis is always positive, then we can further upperbound $E\big[T(n)\big]$ as

$$E\big[T(n)\big] \leq cn$$

This means that the expected execution time is linear.

3.6.2 Blum, Floyd, Pratt, Rivest and Tarjan algorithm

The worst-case complexity of the randomised algorithm we just analysed is quadratic, which is very bad. Luckily, we know a randomised algorithm which is able to select the pivot always in a balanced way (i.e., almost in the centre). This means that the algorithm always has a linear complexity.

Algorithm

The Blum, Floyd, Pratt, Rivest and Tarjan algorithm is very similar to the one we previously analysed. In particular, the recursive part remains unchanged and the only difference is in the initial part. The algorithm works as follows.

1. Divide the input array of n elements into groups of 5 elements.
2. For each of the 5-element groups, compute the median.
3. Compute the median x of the medians we computed at step 2.
4. Use x as pivot and partition around x. Let $k = rank(x)$.
5. Return x if $i = k$, otherwise apply recursion.

Algorithm 14 Blum, Floyd, Pratt, Rivest and Tarjan algorithm.

```
procedure BFPRT-Partition(v, p, q)
    V ← v[p : q]
    i ← 0
    M ← [...]
    while 5 · (i + 1) < n do
        M[i] ← Median(V, 5 · i, 5 · (i + 1))
        i ← i + 1
    end while
    m ← Median(M)
    return Index(m) + p                ▷ Return the position of the median of the medians
end procedure
procedure BFPRT-Rank(A, p, q, i)
    if p = q then
        return A[p]
    end if
    r ← BFPRT-Partition(A, p, q)
    k ← r − p + 1
    if i = k then
        return A[r]
    end if
    if i < k then
        return BFPRT-Rank(A, p, r − 1, i)
    else
        return BFPRT-Rank(A, r + 1, q, i − k)
    end if
    x ← Median(v, 0, |M|)
end procedure
```

3.6.3 Complexity

To compute the complexity of the whole algorithm, we can compute the complexity of each step and then sum everything up. What we get is:

- Dividing the array in groups of 5 elements and computing their median, takes linear time.
- Selecting the median for each
- The partitioning algorithm takes linear time.

In practice, however, this algorithm isn't very fast since the constant that multiplies n is large.

3.7 Primality test

The primality testing problem consists in finding if a number n is prime.

3.7.1 Naive solution

The easiest way to compute if a number n is prime is by applying the definition. Since a number n is prime if and only if its lone divisors are 1 and n, then we check, for each number between 2 and $\sqrt{n}$, if a number divides n. As soon as we find one number that divides n, we can say that n isn't prime. On the other hand, if we can't find a divisor, we can say that n is prime. This solution has a complexity of

$$T_{\text{naive}}(n) = \Theta(\sqrt{n})$$

The complete algorithm is shown in Algorithm 15.

Algorithm 15 A naive algorithm for solving the primality test problem.

Require: $n \geq 1$
  if $n = 1$ **then**
    return false
  end if
  if $n = 2$ **then**
    return true
  end if
  for $i = 2, \ldots, \sqrt{n}$ **do**
    if $n \mod i = 0$ **then**
      return false
    end if
  end for
  return true

3.7.2 Monte Carlo algorithm

We can use a one-sided Mote Carlo algorithm to improve the complexity of the primal test's naive implementation. In particular, if the algorithm says that

- n is **not prime** (i.e., composite), then n is not prime.

- n is **prime**, then n is prime with a probability $1 - p$, with $p > 0$.

We say that this algorithm is **false-biased** since it's more prone to answer false to the question (even if the answer is true). Ideally, we want to reduce as much as possible the probability p. In particular, the algorithm will have a probability

$$Pr(\text{wrong answer}) = p^k$$

of making an error in k iterations. This means that if we repeat the algorithm k times we get a wrong answer with probability p^k.

Useful definitions and theorems

Before explaining how the randomised algorithm works, we have to do some assumptions. First, we can observe that each odd prime number p divides $2^{p-1} - 1$.

$$p \text{ prime} \Rightarrow p | 2^{p-1} - 1$$

This means that, given a number n, we can compute

$$z = 2^{n-1} - 1$$

and check if n divides z. In particular:

- If n doesn't divide z, then we are sure that n is not prime.
- If n divides z than n can be prime or non-prime.

Formally, the algorithm is described in Algorithm 16.

In other words, if n divides z we find all prime numbers and base-2 pseudoprime numbers, where the former is defined as follows.

> **Definition 3.3** (Base-2 pseudoprime number)**.** *A natural number $n \geq 2$ is a base-2 pseudoprime if n is composite (i.e., not prime) and*
>
> $$2^{n-1} \mod n = 1$$

Algorithm 16 The primality test algorithm that uses Fermat's theorem.

```
procedure FERMAT-PRIMALITY(n)
    if 2^(n-1) mod n = 1 then
        return true                          ▷ Possibly prime
    end if
    return false                             ▷ Surely not prime
end procedure
```

This result is generalised by Fermat's little theorem, an important theorem regarding prime numbers.

Theorem 3.1 (Fermat's little theorem)**.** *Let p and a be two natural numbers. If p is prime and $0 < a < p$ then $a^{p-1} \mod p = 1$.*

$$p \text{ prime} \Rightarrow a^{p-1} \mod p = 1$$

Since this theorem is a generalisation of the properties we have seen before, we can generalise the concept of base-2 pseudoprime number and define a base-a pseudoprime number.

Definition 3.4 (Base-a pseudoprime number)**.** *Let $n \geq 2$ and $0 < a < n$ be natural numbers. The number n is a base-a pseudoprime if it's composite (i.e., not prime) and*

$$a^{n-1} \mod n = 1$$

Fermat's theorem can be used to rewrite Algorithm 16 using a randomly chosen a and applying Fermat's theorem. What we get is Algorithm 17.

Algorithm 17 The primality test algorithm that uses Fermat's theorem.

```
procedure FERMAT-RANDOM-PRIMALITY(n)
    a ← RANDOM(2, n − 1)
    if a^{n−1} mod n = 1 then
        return true                                  ▷ Possibly prime
    end if
    return false                                     ▷ Surely not prime
end procedure
```

Carmicheal numbers are important to build a Monte Carlo algorithm, too.

Definition 3.5 (Carmichael number)**.** *Let n be an integer number. The number n is a Carmichael number if n is composite and, for any a with $GCD(a, n) = 1$ (i.e., n and a are coprime), we have*

$$a^{n-1} \mod n = 1$$

Another important theorem that can be used to check the primality of a number is the following.

Theorem 3.2. *Let p and a be natural numbers. If p is prime and $0 < a < p$, then the only solutions to the equation*

$$a^2 \mod p = 1$$

are $a = 1$ and $a = p - 1$.

From Theorem 3.2 we can obtain the definition of non-trivial root square.

Definition 3.6 (Non-trivial root square of 1)**.** *Let n and a be natural numbers. a is called*

non-trivial square root of $1 \mod n$ *if*

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

Fast exponentiation

Many of the checks done in primality test algorithm have to compute the power of a number. Since this operation is frequent, we should optimise it. In particular, when we compute a^p we can directly check if there exists a non-trivial square root of $1 \mod n$. We can verify the existence of non-trivial square roots of 1 by noticing that $a^p \mod n$ can be written as $(a^{\frac{p}{2}})^2 \mod n$. This means that we can compute $x = a^{\frac{p}{2}} \mod n$ (recursively) and then compute $x^2 \mod n$. If this value is 1 and x is neither 1 nor $n-1$, then we have found a non-trivial square root of 1 (and n is definitely not prime). The formal algorithm is shown in Algorithm 18.

Algorithm 18 The algorithm for computing $a^p \mod n$ and checking if there is an x different from 1 and $n-1$ such that $x^2 \mod n = 1$.

```
procedure POWER(a, p, n, prime)
    if p = 0 then
        return 1;
    end if
    x ← POWER(a, p/2, n, prime)
    r ← x^2 mod n
    if r = 1 ∧ x ≠ 1 ∧ x ≠ n − 1 then
        prime ← false
    end if
    if p mod 2 = 1 then
        r ← (a · r) mod n
    end if
    return r
end procedure
```

The complexity of this algorithm is

$$T_{\text{power}}(n) = \mathcal{O}(\log n)$$

since we do the recursion on the halved exponent.

Algorithm

The following algorithm exploits Fermat's little theorem (3.1) and non-trivial square roots of 1 mod n to check if a number n is prime. More precisely, the algorithm:

1. Picks a random value a.
2. Checks if $a^{n-1} \mod n$ is equal to 1. While doing this computation, the algorithm also checks if there exist non-trivial square roots of $1 \mod n$.
    - If not or we did find a non-trivial square root, n is composite (i.e., not prime).

- Otherwise n is probably prime and we can either return or try with another a by going back to step 1.

This algorithm can be formalised as in Algorithm 19. Its complexity is

$$T_{\text{Monte Carlo primality}}(n) = \mathcal{O}(\log^2 n \log p)$$

Algorithm 19 Monte Carlo primality test.

```
procedure PRIMALITY-TEST(n)
    a ← RANDOM(2, n − 1)          ▷ 2 and n − 1 are included in the possible values
    prime ← true
    r ← POWER(a, n − 1, n, prime)
    if r ≠ 1 ∨ prime = false then
        return false
    else
        return true
    end if
end procedure
```

Probabilities

Since we are using a Monte Carlo algorithm, we have to compute the probability of returning a wrong answer, that is of finding a pseudoprime number.

$$Pr(n \text{ not prime }, a^{n-1} \mod n = 1)$$

We can use the following theorem to compute such probability.

Theorem 3.3. *Let n be a natural number. If n is composite then there are at most*

$$n - \frac{9}{4}$$

integers $0 < a < n$ for which the Monte Carlo primality test algorithm fails.

Chapter 4

Disjoint sets

4.1 Disjoint sets

First of all, let us define what a disjoint set is.

> **Definition 4.1** (Disjoint set)**.** *A disjoint set is a dynamic collection*
>
> $$\gamma = \{S_1, S_2, \ldots, S_k\}$$
>
> *of disjoint dynamic sets S_i.*

Namely, a disjoint set is a set of sets S_i and the intersection between any two sets S_i, S_j is empty. Since there is no intersection between the sets, they can be described efficiently using one representative of the set. In other words, each set S_i is represented by an element $e_i \in S_i$, which is the representative of S_i. Note that we can choose whatever element in a set to be a representative. Moreover, it's very fast to check if an item belongs to a certain set.

4.2 Operations

4.2.1 Make set

The `MAKE-SET(x)` operation is used to create a set S_x in the disjoint set γ. Initially, the set will contain only one element x, which is also the representative of the set. This means that the result of the operation is

$$S_i = \{x\}$$

and if we add this set to a disjoint set γ we get

$$\gamma = \gamma \cup S_i$$

Note that, if we want to add S_i to γ, x can't be in any other set $S_j \in \gamma$ (otherwise the sets in γ wouldn't be disjoint).

4.2.2 Union

The `UNION(x, y)` operation is used to join two sets in a disjoint set. In particular, the union operation takes two sets $S_x \in \gamma$ and $S_y \in \gamma$, represented by x and y, and creates a new set S, which is the union of the sets represented by x and y. Note that, we have to remove the sets S_x and S_y from γ, otherwise we will have non-disjoint sets. The result of this operation can be written as follows:

$$\gamma = \{\{\gamma \setminus S_x\} \setminus S_y\} \cup \{S_x \cup S_y\}$$

4.2.3 Find set

The `FIND-SET(x)` operation allows to find the set S_i in a disjoint set γ which contains the element x. Note that this operation can be implemented very efficiently and the result is unique since the sets in γ are disjoint.

4.2.4 Evolution of the set

A disjoint set γ is very useful to describe an evolving data structure. In particular:

1. Initially, a disjoint set can be initialised with n `MAKE-SET` operations which create n disjoint sets with one element each.
2. After initialising the data structure, we can find similarities between the sets and join those which contain similar elements. In this phase we could use, for instance, a total of m `MAKE-SET`, `UNION` and `FIND-SET` operations.

4.3 Implementation

Now that we have done an overview on how disjoint sets work, we can give a deeper look on how they are implemented.

4.3.1 Linked lists

The first way to implement disjoint sets is by using linked lists. A set S is described by

- An `head` that points to the first element of the list.
- A `tail` that points to the last element of the list.
- A list of nodes, each of which contains
  - The **element** itself.
  - A **pointer** to the **next** node.
  - A **pointer** to the **first** node.

The representative of the set is the first element of the linked list, namely the one pointed by the head of the list.

Having described the internal structure of a linked list, we can analyse the implementation of the operations that can be done on a disjoint set.

Make set

The make set operation simply requires to

1. Create a node.
2. Create a head and make it point to the created node.
3. Create a tail and make it point to the created node.

These operations can be done in constant time, hence we can create a set in constant time.

$$T_{\text{make-set, linked-list}}(n) = \mathcal{O}(1)$$

Find set

The find set operation requires to

1. Find the element in the list.
2. Return the representative using the pointer to the representative.

This operation has a constant cost since we can keep an hash table that stores, for each value, the address of the node where it's saved hence we can retrieve the node in constant time and immediately follow the pointer to the head of the list where it is (i.e., to the representative node), namely

$$T_{\text{find-set, linked-list}}(n) = \mathcal{O}(1)$$

Union

If we want to compute the union between two sets S_1 and S_2 we have to:

1. Use the tail of S_1 to make the last node of S_1 point to the first of S_2.
2. Update all the pointers to the representative of the nodes in S_2.
3. Update the tail of S_1 so that it points to the last element of S_2.

The first and last operations can be done in constant time since they need only to move a pointer (in the second case we can use the tail of S_2). However, the second operation has to pass every element in S_2, hence the complexity is linear in the length of S_2. This means that

$$T_{\text{union, linked-list}}(n) = \mathcal{O}(n)$$

Sequence of operations

Since a disjoint set is a dynamic structure, it's important to analyse its behaviour when it's modified. Say we want to add n elements to the disjoint set with n `MAKE-SET` operations and then modify the data structure with $n-1$ `UNION`s. The `MAKE-SET` operations will require, in total, linear time. The `UNION` will instead work on an increasing number of elements, hence its complexity will increase from 1 to $n-1$. This means that the total cost of $n-1$ `UNION` operations is

$$T_{\text{unions}}(n) = 1 + 2 + \cdots + n - 1 = \Theta(n^2)$$

which is greater than the cost of the `MAKE-SET` operations (which was linear). This means that, to execute $2n-1$ operations (n `MAKE-SET`s and $n-1$ `UNION`s) costs $\Theta(n^2)$ and on average each operation costs $\Theta(n)$.

4.3.2 Forests

We can implement disjoint sets with forests to optimise their operations. In particular, each set S_i is represented with a tree in which:

- The **representative** is in the **root**.
- Each node in the tree points to its parent (and the root points to itself).

Make set

The make set operation simply requires to create a node, which is the root of a tree. This operation requires a constant time.

```
procedure MAKE-SET(x)
    p[x] ← x
    rank[x] ← 0
end procedure
```

Find set

The find set operation requires us to move from a node of the tree to the root. If the tree is balanced, this operation requires a time proportional to the height of the tree. Note that, if the union can't ensure to build a balanced tree, we could have a linear complexity in the number of nodes, i.e., $\mathcal{O}(n)$.

```
procedure FIND-SET(x)
    if x ≠ p[x] then
        p[x] ← FIND-SET (p[x])
    end if
    return p[x]
end procedure
```

Union

Thanks to the tree data structure, the union can be implemented in constant time, in fact we just have to add the root of one set as the child of another set. If we want to join S_1 and S_2 we can, for instance, add S_1 as a child of a node in S_2. Note that this operation takes constant time since we are only moving one pointer.

Union by rank and path compression

The find set operation depends on the union. If the latter can build a balanced tree, the former can find an element more efficiently. This means that we have to find a way to join sets in constant time without building unbalanced trees. Rank and compression is what we need and it works as follows:

- Each node is associated with a rank, which is the height of the sub-tree rooted at that node (i.e., the maximum length from the node to a leaf).

- Root with smaller rank will point to the root with larger rank.

Moreover, since the find set operation depends directly on the height of the tree, we can use path compression to reduce it. More precisely, a leaf which isn't connected to the root can decide to change its parent and point to the root. Note that this is possible since we aren't considering binary trees.

---

procedure LINK(x, y)
  if $rank[x] > rank[y]$ **then** ▷ If x's higher than y, add y as x's son.
    $p[y] \leftarrow x$
  else
    $p[x] \leftarrow y$
    if $rank[x] = rank[y]$ **then**
      $rank[y] \leftarrow rank[y] + 1$
    end if
  end if
end procedure
procedure UNION(x, y)
  LINK(FIND-SET(x), FIND-SET(y))
end procedure

In the worst case, running m `MAKE-SET`, `UNION` and `FIND-SET` operations has a complexity of

$$T(n, m) = \mathcal{O}(m\alpha(n))$$

where $\alpha(n)$ is a number that grows very slowly and it's usually smaller than 4, hence the complexity is usually linear in the total number m of operations. To prove that $\alpha(n)$ grows really slowly, we can consider its inverse and prove that it grows really fast. Let us consider a recursive function

$$A_k(j) = \begin{cases} j+1 & \text{if } k = 0 \\ A_{k-1}^{(j+1)}(j) & \text{if } k \geq 1 \end{cases} \quad \forall k \geq 0, \forall j \geq 1$$

where the power operation (i.e. $A_{k-1}^{(i)}(j)$) is defined as

$$A_{k-1}^{(i)}(j) = \begin{cases} A_{k-1}\big(A_{k-1}^{(i-1)}(j)\big) & \text{if } i \geq 1 \\ j & \text{if } i = 0 \end{cases}$$

Note that k is usually called the level of the function and $i \geq 1$ is the number of iterations. From this definition, we can obtain two important results:

Proposition 4.1. *For any integer $j \geq 1$,*

$$A_1(j) = 2j + 1$$

Proof. First we can prove that

$$\begin{aligned} A_0^i(j) &= A_0\big(A_0^{(i-1)}(j)\big) \\ &= A_0\big(A_0(A_0^{(i-2)}(j))\big) \\ &= A_0\big(A_0(\cdots A_0^{(0)}(j))\big) \\ &= A_0\big(A_0(\cdots j)\big) \\ &= A_0\big(A_0(j+i-2)\big) \\ &= A_0(j+i-1) \\ &= j+i \end{aligned}$$

Basically, since the exponent of $A_0^{(i)}(j)$ is i, the recursion is repeated i times hence it's like adding 1 to j for i times. We can use this result to write the definition of $A_k(j)$ and obtain

$$\begin{aligned} A_1(j) &= A_0^{(j+1)}(j) \\ &= A_0\big(A_0^{(j+1-1)}(j)\big) \\ &= A_0^{(j+1-1)}(j)+1 \\ &= A_0^{(j)}(j)+1 \\ &= j+j+1 = 2j+1 \end{aligned}$$

□

Proposition 4.2. *For any integer $j \geq 1$,*

$$A_2(j) = 2^{j+1} \cdot (j+1) - 1$$

Proof. First, let us prove that

$$\begin{aligned} A_1^i(j) &= A_1\big(A_1^{i-1}(j)\big) \\ &= A_1\big(A_1(A_1^{i-2}(j))\big) \\ &= A_1\big(A_1(A_1(A_1^{i-3}(j)))\big) \\ &\vdots \\ &= A_1\big(A_1(A_1(A_1(\ldots A_1(A_1^0(j))))))\big) \\ &= A_1\big(A_1(A_1(A_1(\ldots A_1(j)))))\big) \\ &= A_1\big(A_1(A_1(A_1(\ldots (2j+1)))))\big) \qquad \text{Apply} A_1(j) = 2j+1 \\ &= 2^i j + 2^i - 1 \\ &= 2^i(j+1) - 1 \end{aligned}$$

Note that the -1 is justified by the fact that the inner substitution (i.e., $A_1(j) = 2j+1$) doesn't

produce $2j + 2$. We can use this result to prove

$$\begin{aligned} A_2(j) &= A_1^{j+1}(j) \\ &= 2^{j+1}(j+1) - 1 \qquad \text{Apply } A_1^i(j) = 2^i(j+1) - 1 \end{aligned}$$

□

This results can be used to shown how fast the function A_k grows. In particular we can see that

- $A_0(1) = 1 + 1 = 2$
- $A_1(1) = 2 \cdot 1 + 1 = 3$
- $A_2(1) = 2^{1+1} \cdot (1 + 1) - 1 = 7$
- $$\begin{aligned} A_3(1) &= A_2^{(2)}(1) \\ &= A_2(A_2^{(1)}(1)) \\ &= A_2(A_2(A_2^{(0)}(1))) \\ &= A_2(A_2(1)) \\ &= A_2(7) \\ &= 2^{7+1}(7+1) - 1 = 2047 \end{aligned}$$
- $A_4(1) >> 10^{80}$ (number of atoms in the universe)

The function $\alpha(n)$ used in the complexity of the union operation is the inverse of the function A_k and it's defined as

$$\alpha(n) = \min\{k : A_k(1) \geq n\}$$

Using the results we have seen before, we can say that

$$\alpha(n) = \begin{cases} 0 & \text{for } 0 \leq n \leq 2 \\ 1 & \text{for } n = 3 \\ 2 & \text{for } 4 \leq n \leq 7 \\ 3 & \text{for } 8 \leq n \leq 2047 \\ 4 & \text{for } 2048 \leq n \leq A_4(1) \end{cases}$$

hence $\alpha(n) \leq 4$ for all numbers smaller than $A_4(1)$. But since $A_4(1)$ is much bigger than the total number of atoms in the universe, we can safely assume that $\alpha(n) \leq 4$ for all practical values of n.

Chapter 5

Randomised data structures

5.1 Dictionaries

The randomised data structures we are going to analyse are related to dictionaries, hence it's better to remember what a dictionary is.

> **Definition 5.1** (Dictionary)**.** *A dictionary is a collection of elements, each of which has a unique key.*

Note that we can define a total order between the keys of a dictionary, hence we can order the elements of the dictionary.

5.1.1 Operations

The main operations we can do on a dictionary S are:

- **Search** in S an element with key x.
- **Insert** in S an element with key x.
- **Delete** from S an element with key x.

We can also define some more complex operations:

- `MAXIMUM(S)` finds the key with the highest value in dictionary S.
- `MINIMUM(S)` finds the key with the lowest value in dictionary S.
- `LIST(S)` outputs the elements of S with increasing values of the key.
- `UNION(S1, S2)` creates a new dictionary S that contains all the elements in S_1 and S_2. Note that this operation can be done only if the keys in S_1 are smaller than the keys in S_2.
- `SPLIT(S, x, S1, S2)` splits S into two sub-dictionaries S_1 and S_2. The former contains all the elements with key smaller than or equal to x while the latter contains all the elements with key bigger than x.

5.1.2 Implementation

Dictionaries can be implemented using:

- **Unordered arrays**. With unordered arrays, we can insert elements in constant time

  $$T_{\text{insert, unordered array}} = \mathcal{O}(1)$$

  however searching and removing an element has a linear cost

  $$T_{\text{search, unordered array}} = T_{\text{delete, unordered array}} = \mathcal{O}(n)$$

  because we have to pass the whole array.

- **Ordered arrays**. With ordered arrays, we can search elements in logarithmic time using binary search (being the array sorted)

  $$T_{\text{search, unordered array}} = \mathcal{O}(\log n)$$

  however insertion and deletion takes linear time because we have to keep the structure of the array intact

  $$T_{\text{insert, unordered array}} = T_{\text{delete, unordered array}} = \mathcal{O}(n)$$

- **Binary trees**.

Usually, these implementations require auxiliary data structures to work properly.

5.1.3 Random treaps

Treaps are binary search trees that

- Don't require additional data structures.
- Don't require explicit balance information (i.e., the height of the tree, previous rotation).

Random treaps have nice characteristics, in fact they:

- Are very easy to implement and maintain.
- Require a small number of rotations to keep the tree balanced. In particular, we expect two rotations, on average, for each insertion or deletion.

Random treaps have been created starting from a simple observation. Say we have a set of elements, each of which has a key, and we build a binary search tree inserting the elements in a random way. The expected depth of the binary tree is

$$d = 1.39 \log n$$

The problem is that we can't always build a tree in a random order since usually the user decides what is the order in which elements are added. To solve this problem we need to add some randomisation in the insertion process. In particular, random treaps use priorities. In other words, thanks to priorities we can obtain a tree which has the same properties of a tree built in random order, even if the elements have been added in a specific order.

Let us now understand how priorities work. A priority is a number chosen uniformly at random in the real domain. We assign a priority to each element, hence an element has

- A **key** that uniquely identifies it.
- A **priority**, chosen at random in $\mathbb{R}$.

This means that each node of a tree contains:

- An element x.
- Its key $\text{key}(x)$.
- Its priority $\text{prio}(x)$.

The tree is built adhering to the following rules:

1. **Binary search property**. Given an element x, all the elements y in x's left subtree have a key smaller than x's.
$$\text{key}(y) < \text{key}(x) \quad \forall y \text{ in left subtree of } x$$
2. **Binary search property**. Given an element x, all the elements y in x's right subtree have a key bigger than x's.
$$\text{key}(y) > \text{key}(x) \quad \forall y \text{ in right subtree of } x$$
3. **Heap property**. For all elements x, y in the tree, if y is a child of x, then the priority of y is greater than the priority of x.
$$y \text{ child of } x \Rightarrow \text{prio}(y) > \text{prio}(x)$$

Initially we will assume that each node has a different priority. This doesn't happen in practice, however it's useful to analyse the data structure's properties.

Now that we have clear what the structure of a random treap is, let us state an important property of this data structure.

Theorem 5.1 (Random treap uniqueness)**.** *For elements $x_1, \ldots, x_n$ with key $\text{key}(x_i)$ and priority $\text{prio}(x_i)$, there exists a unique treap.*

Proof. This property can be proved by induction. In particular, it's trivially true when we have only one element x_1. If we consider $n > 1$, we have that all treaps with a number of elements smaller than n are going to be unique with respect to the combination of priorities and keys. A treap with n elements can be seen as a treap with a root and two subtrees, each having $n - 1$ elements. By induction hypothesis both subtrees are unique, hence, attaching them to the root we obtain a unique treap. □

Thanks to Theorem 5.1 we know that a treap with a given set of priorities and keys is unique. Moreover, we know that its structure is defined by the priorities. This result is summed up by the following theorem.

Theorem 5.2 (Random treap structure)**.** *The search tree has the structure that would result if elements were inserted in the order of their priorities.*

Proof. Once again, we can prove this theorem by induction. For $n = 1$ the theorem is trivially true. For $n > 1$ we assume that the theorem is true for $n - 1$, hence given a treap of $n - 1$ elements, its structure is the one obtained inserting elements in order of priority. Given the treap with n nodes, for the heap property, every child of the root must have priority greater than the one of the root, hence the root must have priority 1 and all the other nodes have a greater priority. This means that the root must be the first to be inserted and the nodes in the subtrees must have been inserted after the root, in the order given by the priority (because of the recursion hypothesis). □

This is an important result because it means that the shape of the tree is independent from the order in which elements are inserted but it depends only on the priority assigned to each element. Given that the priority is randomly generated, we have obtained a data structure that behaves like a tree in which we insert elements in a random way, even if we insert them in a specific order. This ensures that the expected depth of the tree is

$$d = 1.39 \log n$$

as we have already mentioned.

Search

Searching an element in a random treap is like searching in a binary tree. More precisely, given a key and the root of the tree, we

- Return the current node (initially the root) if the node's key is equal to the searched key.
- Search the element in the left subtree if the searched key is smaller than the node's tree.
- Search the element in the right subtree if the searched key is bigger than the node's tree.

The algorithm is also shown in Algorithm 20.

Algorithm 20 Search in a random treap.

```
procedure SEARCH(k, r)
    v ← r
    while v ≠ nil do
        if KEY(v) = k then
            return v
        end if
        if KEY(v) < k then
            v ← LEFT-CHILD(v)
        end if
        if KEY(v) > k then
            v ← RIGHT-CHILD(v)
        end if
    end while
    return nil
end procedure
```

Note that the time required to find an element depends on the longest path of the tree, namely, on its depth. Because of the depth property of a random treap (the depth is $1.39 \log n$), the time

complexity of the search is logarithmic.

$$T_{\text{search, treap}} = \mathcal{O}(\log n)$$

Let's prove this statement. Let us consider a set of nodes $x_1, \dots, x_n$ ordered by keys, that is to say

$$\text{key}(x_1) < \text{key}(x_2) < \cdots < \text{key}(x_n)$$

If M a subset of these elements, we write

$$P_{min}(M)$$

the element in M with lowest priority. Thanks to these information we can state the following proposition.

Proposition 5.1. *Let $m > i$. A node x_i is an ancestor of x_m if and only if the minimum priority of the subset $\{x_i, \dots, x_m\}$ is x_i.*

$$x_i \text{ ancestor of } x_m \iff P_{min}(\{x_i, \dots, x_m\}) = x_i$$

Let $m < i$. A node x_i is an ancestor of x_m if and only if the minimum priority of the subset $\{x_i, \dots, x_m\}$ is x_i.

$$x_i \text{ ancestor of } x_m \iff P_{min}(\{x_m, \dots, x_i\}) = x_i$$

Proof. Let's start by proving the inverse implication. We know that the minimum priority of the subset $\{x_i, \dots, x_m\}$ is x_i. This means that x_i is inserted before any other element of the subset $\{x_{i+1}, \dots, x_m\}$. Since elements are sorted in increasing order of keys, every element x_j in $\{x_{i+1}, \dots, x_{m-1}\}$ has a key bigger than $\text{key}(x_i)$ and smaller than $\text{key}(x_m)$. Since x_i has the lower priority, the first element inserted must be x_i. Since we have inserted x_i first, every other element is connected to x_i, in fact we can build the tree using the rules for inserting elements until we finally insert x_m.

Let's prove now the direct implication. We assume that x_i is an ancestor of x_m and we want to prove that $P_{min}(\{x_i, \dots, x_m\}) = x_i$. Say we take an element x_j in $\{x_i, \dots, x_m\}$ such that

$$x_j = P_{min}(\{x_i, \dots, x_m\})$$

and let us suppose that $x_j \neq x_i$. Now we want to prove, by contradiction, that if $x_j \neq x_i$ the hypothesis is always false (i.e., that x_i is not an ancestor of x_m) so that it must be that $x_j = x_i$, hence x_i has minimum priority. For hypothesis, $x_i \neq x_j$, x_j has minimum priority and the key of x_j is bigger than the one of x_i because we have taken x_j in $\{x_i, \dots, x_m\}$. This means that, x_i, when inserted in the tree, must be appended on the left of x_j. Now we have to consider two cases:

- If $x_j = x_m$, then x_i is appended on the left of x_m, hence it's not an ancestor of x_m. It's actually the opposite, in fact x_m is an ancestor of x_i.
- If $x_i \neq x_m$, then x_i is appended on the left of x_j but x_m is appended on the right, since x_m has the highest key (hence $\text{key}(x_i)$ is surely smaller than $\text{key}(x_m)$). Long story short, x_i isn't an ancestor of x_m since they are on different branches nested in x_j.

Since in both cases we reach an absurd condition, the hypothesis must have been false and $x_i = x_j = P_{min}(\{x_i, \dots, x_m\})$. □

The expected number of nodes traversed can be found precisely using the following proposition.

Proposition 5.2. *Let H_n be the n-th Harmonic number defined as*

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

Let T be a treap with elements $x_i, \ldots, x_n$, ordered in increasing key values. Then

- *In case of successful search, the expected number of nodes on the path to x_m is*

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

- *In case of unsuccessful search, the expected number of nodes one the path to x_m is*

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

where m is the number of keys smaller than the key we are looking for.

Proof. Let $X_{m,i}$ be a random variable defined as

$$X_{m,i} = \begin{cases} 1 & \text{if } x_i \text{ is an ancestor of } x_m \\ 0 & \text{otherwise} \end{cases}$$

The length of the path is the number of nodes X_m on the path from x_i to x_m (x_m included) which is the sum x_m's ancestors.

$$X_m = 1 + \sum_i X_{m,i} = 1 + \sum_{i<m} X_{m,i} + \sum_{i>m} X_{m,i}$$

If we apply the expected value to X_m we get

$$\begin{aligned} E[X_m] &= E\big[1 + \sum_{i<m} X_{m,i} + \sum_{i>m} X_{m,i}\big] \\ &= 1 + \sum_{i<m} E\big[X_{m,i}\big] + \sum_{i>m} E\big[X_{m,i}\big] \end{aligned}$$

Let us now consider the term $\sum_{i<m} E\big[X_{m,i}\big]$ which counts the expected number of ancestors with a key smaller than x_m's. By Proposition 5.1, x_i is an ancestor if it has the smallest priority in $\{x_i, \ldots, x_m\}$. Since all elements of $\{x_i, \ldots, x_m\}$ have the same probability of being the one with the smallest priority in a set of $m - i + 1$ elements, then the probability that x_i has has the smallest one is

$$E\big[X_{m,i}\big] = P[P_{min}(\{x_i, \ldots, x_m\}) = x_i] = \frac{1}{m - i + 1}$$

Let us finally consider the term $\sum_{i>m} E\big[X_{m,i}\big]$ which counts the expected number of ancestors with a key greater than x_m's. As before, x_i is an ancestor if it has the smallest priority in $\{x_m, \ldots, x_i\}$, which happens with probability

$$E\big[X_{m,i}\big] = P[P_{min}(\{x_m, \ldots, x_i\}) = x_i] = \frac{1}{i - m + 1}$$

Summing all up we get

$$\begin{aligned}
E[X_m] &= 1 + \sum_{i<m} E\big[X_{m,i}\big] + \sum_{i>m} E\big[X_{m,i}\big] \\
&= 1 + \sum_{i<m} \frac{1}{m-i+1} + \sum_{i>m} \frac{1}{i-m+1} \\
&= 1 + \frac{1}{m-1+1} + \frac{1}{m-2+1} + \cdots + \frac{1}{m-(m-1)+1} \\
&+ \frac{1}{m+1-m+1} + \frac{1}{m+2-m+1} + \cdots + \frac{1}{n-m+1} \\
&= 1 + \frac{1}{m} + \frac{1}{m-1} + \cdots + \frac{1}{2} + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n-m+1} \\
&= 1 + \frac{1}{m} + \frac{1}{m-1} + \cdots + \frac{1}{2} - 1 + 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n-m+1} \\
&= H_m - 1 + H_{n-m+1} \\
&= H_m + H_{n-m+1} - 1
\end{aligned}$$

The second point can be proved in the same way. □

Insertion

When we want to insert an element x in a treap we have to:

1. Pick a priority for x at random.
2. Search the position of x in the tree (which is given by its priority and its key). The search won't find the element (since we are inserting it, it isn't in the tree) but will put us on a leaf.
3. We add a child to the leaf x_l we have landed on (on the right if $\text{key}(x) > \textbf{key}(x_l)$, on the left otherwise).
4. Restore the heap property of the tree. The algorithm for restoring the priority property is shown in Algorithm 21.

Algorithm 21 Algorithm for restoring the heap property of a random treap.

```
while PRIO(parent(x)) > PRIO(x) do
    if x is left child then
        ROTATE-RIGHT(parent(x))
    else
        ROTATE-LEFT(parent(x))
    end if
end while
```

Since the insert operation is reduced to a search and a number of rotations, we have to compute the expected number of rotations to define the complexity of inserting an element. Let us analyse another important operation on a treap before focusing on the number of rotations and their complexity.

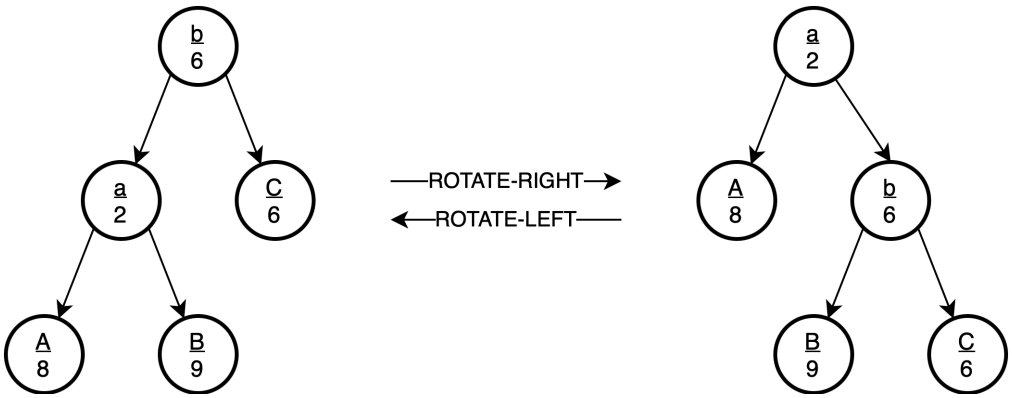


Figure 5.1: Left and right rotation of a tree.

Deletion

Another important operation we have to analyse is deletion. As the insertion, this operation is based on the search of the element. More precisely, we have to:

1. Find x in the tree.
2. Rotate the tree so that the heap property is true. The algorithm executed at this step is shown in Algorithm 22. As we can see, the main goal of the algorithm is to put the element x to delete as a leaf.
3. Delete x.

Algorithm 22 Algorithm for restoring the heap property of a random treap when deleting an element.

```
while x not a leaf do
    u ← child of x with smallest priority
    if u is left child then
        ROTATE-RIGHT(x)
    else
        ROTATE-LEFT(x)
    end if
end while
```

As for insertion, the complexity of deleting an element is reduced to the complexity of searching it and the complexity of doing some rotations.

Rotations

As we have seen rotations are fundamentals for deleting and inserting an element. An example of rotation is shown in Figure 5.1.

We have to understand what is the complexity of a rotation and, on average, what is the number of rotations required for each insertion or deletion. The number of rotations required is stated by the following proposition.

Proposition 5.3. *The expected running time of insert and delete operations is $\mathcal{O}(\log n)$. The expected number of rotations, for each operation, is 2.*

Proof. Let us consider the insert operation (the same reasoning can be applied to the deleting, since it's the opposite operation). The number of rotations is given by

$$N_{rot} = d_{x,leaf} - d_{x,rot}$$

where

- $d_{x,leaf}$ is the depth of x after being inserted as a leaf.
- $d_{x,rot}$ is the depth of x after the rotations required to ensure the heap property.

$d_{x,leaf}$ is the expected depth of the tree, without x (since x hasn't been inserted). If we consider $x = x_m$ inserted in $\{x_i, \dots, x_{m-1}\} \subseteq \{x_1, \dots, x_n\}$ then, for Proposition 5.2 we have

$$d_{x,leaf} = H_{m-1} + H_{n-m} + 1$$

where the 1 is added because we want the depth of x and not of its parent (the proposition gives us the depth of the leaf to which x should be attached). Using the same proposition we can compute $d_{x,rot}$ as

$$d_{x,rot} = H_m + H_{n-m+1} - 1$$

since, once $x = x_m$ has been placed in its spot, it will require $H_m + H_{n-m+1} - 1$ steps to find it. If we sum it all up we get

$$\begin{aligned} N_{rot} &= d_{x,leaf} - d_{x,rot} \\ &= H_{m-1} + H_{n-m} + 1 - (H_m + H_{n-m+1} - 1) \\ &= H_{m-1} + H_{n-m} + 1 - H_m - H_{n-m+1} + 1 \end{aligned}$$

If we compute this value we get that it's always smaller than 2, hence

$$N_{rot} = H_{m-1} + H_{n-m} - H_m - H_{n-m+1} + 2 < 2$$

□

This result is very important since it tells us that the number of rotation required to keep the tree balanced isn't relevant and doesn't affect the complexity of the operations.

Minimum and maximum

Given a treap, we can compute the minimum by following the left children, until we reach a leaf. Similarly, we can find the maximum by following the right children until we find a leaf. Both operations have a complexity that depends on the height of the tree. Since the tree has a logarithmic height (with respect to the number of nodes), the complexity of finding the maximum or the minimum is

$$T_{max,min}(n) = \mathcal{O}(\log n)$$

Listing

Another useful operation we can do is list all the nodes in the tree. Since we have to visit the whole tree, the complexity of this operation is the number of nodes in the tree.

$$T_{list}(n) = \mathcal{O}(n)$$

Split

Given a treap T and a key k, one might want to obtain two treaps T_1 and T_2 that contains the elements with keys smaller and bigger than k, respectively. In other words:

$$\forall x_i \in T_1 : \text{key}(x_i) \leq k$$

and

$$\forall x_i \in T_2 : \text{key}(x_i) > k$$

Let us assume, without loss of generality, that k is not in T. In fact, if $k \in T$, we can remove it before splitting T. The algorithm for splitting T works as follows:

1. Create a new element x such that $\text{key}(x) = k$ and $\text{prio}(x) = -\infty$.
2. Insert x into T.
3. Delete the root obtained after the insertion, which is x (since it has a priority smaller than any other element in T). The left subtree is T_1, the right subtree is T_2.

The worst case is when k is in T. In this case we have to:

1. Remove x.
2. Insert x in T.
3. Insert x in T_1.

All operations have a logarithmic complexity, hence the complexity of the split is logarithmic, too.

$$T_{split}(n) = \mathcal{O}(\log n)$$

Union

The union operation is the opposite of the split. In this case we have T_1 and T_2 and we want to build a new treap T which has all the elements of T_1 and T_2. Note that the union works only if

$$\forall x_1 \in T_1, x_2 \in T_2 : \text{key}(x_1) < \text{key}(x_2)$$

The union operation works as follows:

1. Find a key k (not in T) such that k is bigger than any key in T_1 and smaller than any key in T_2.

   $$\text{key}(x_1) < k < \text{key}(x_2) \forall x_1 \in T_1, x_2 \in T_2$$

   Finding the bigger and smaller keys in T_1 and T_2 takes logarithmic time (we have to use the `max` and `min` operations), hence this operation has a logarithmic complexity.

2. Generate a new element x such that $\text{key}(x) = k$ and $\text{prio}(x) = -\infty$. This operation has a constant cost since we simply have to assign a couple of values.
3. Generate a treap with root x, left subtree T_1 and right subtree T_2. This operation has a constant cost since we have to nest two subtrees on a root.
4. Remove x from T. This operation has a logarithmic cost, since it is a simple remove operation.

All in all, the union operation takes a logarithmic time, since all its operation takes a logarithmic (or constant) time.

$$T_{union}(n) = \mathcal{O}(\log n)$$

Priorities

Let us assume priorities in the interval $[0, 1)$ (which is isomorphic $\mathbb{R}$). Initially we have assumed that all priorities were distinct. We can actually loosen this hypothesis and require that the priority of a node is different from the one of its parent since we use the priority just to understand if a rotation should be made. In case of equalities, we can extend both priorities by some bits chosen uniformly at random until two corresponding bits differ. If we have, for instance

$$p_1 = p_2 = 0.010111001$$

we can extend these priorities until we get to

$$p_1 = 0.010111001011$$

and

$$p_2 = 0.010111001010$$

5.2 Skip lists

Skip lists are simple data structures, based on randomisation, uses to manage a dynamic set of n elements. Each operation on the skip list is executed, in expectation, in

$$T_{operation,skip-list}(n) = \mathcal{O}(\log n)$$

with high probability.

5.2.1 Two layers skip-list

Structure

Skip lists are based on a doubly-linked list, which requires linear time to search the list for an element. We can speed up this operation by:

- Imposing that the elements in the linked list are sorted.
- Adding another doubly-linked, also sorted, list with a subset of the first list's elements.

The elements of the second linked list are connected to their respective in the first list. An example of this structure is shown in Figure 5.2.

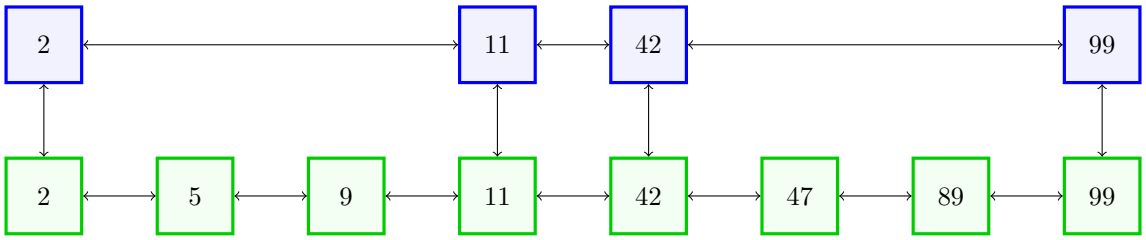


Figure 5.2: A skip-list with two layers.

Search

Since search is the operation we are trying to speed up, let us focus on it. To search for a value v in the skip-list we have to:

1. Go through the top list (i.e., the list with less nodes) until we find that the current value v_i is bigger than the value v we are looking for.
2. Go back one node (so that $v_{i-1} < v$).
3. Move to the list beneath (i.e. to the list with more nodes).
4. Search the list for the element.

Let's analyse the complexity of this algorithm. First, let us call L_2 the list on top and L_1 the list below containing all the nodes. The complexity of the search operation depends on the number of nodes in L_2, i.e. $|L_2|$. If we assume that nodes in L_2 are evenly spaced (i.e., we have the same number of L_1 nodes between adjacent L_2 nodes), the search cost is

$$T_{search,2-layers} = |L_2| + \frac{|L_1|}{|L_2|}$$

where

- $|L_2|$ is the cost of traversing every node in L_2, which happens in the worst case (e.g., if we are searching $v = 10$ and the last value of the list is smaller).
- $\frac{|L_1|}{|L_2|}$ is the number of nodes between two adjacent nodes in L_2 (assuming evenly spaced nodes in L_2).

Note that L_1 is equal to n since L_1 contains every node in the skip-list. We can compute the minimum of this cost computing the derivative over $|L_2|$, since $|L_1| = n$ is constant, and putting it

to 0. If we call $|L_2| = x$ we get

$$\begin{aligned}
\frac{d}{dx}T(x) &= 0 \\
\frac{d}{dx}\left(x + \frac{n}{x}\right) &= 0 \\
1 - \frac{n}{x^2} &= 0 \\
\frac{n}{x^2} &= 1 \\
n &= x^2 \\
x &= \sqrt{n} \\
|L_2| &= \sqrt{n}
\end{aligned}$$

This means that, in the optimal case, search has a cost of

$$\begin{aligned}
T_{search,2-layers} &= |L_2| + \frac{|L_1|}{|L_2|} \\
&= \sqrt{n} + \frac{n}{\sqrt{n}} \\
&= \sqrt{n} + \frac{n\sqrt{n}}{n} \\
&= \sqrt{n} + \sqrt{n} \\
&= 2\sqrt{n}
\end{aligned}$$

5.2.2 k-layers skip-lists

Let us now extend the structure to a generic skip-list with k layers $L_1, \ldots, L_k$ where L_1 is the one with all the nodes. With k layers, the cost of search is

$$T_{search,k-layers} = k \cdot \sqrt[k]{n}$$

This result is pretty interesting because if we take $k = \log n$, then the number of operations required to search the skip list are

$$\begin{aligned}
T_{search,k-layers} &= \log n \cdot \sqrt[\log n]{n} \\
&= \log n \cdot n^{\frac{1}{\log n}} \\
&= \log n \cdot 2^{\log n \cdot \frac{1}{\log n}} \\
&= 2 \log n
\end{aligned}$$

This means that, with $\log n$ layers and evenly separated nodes in each layer, we can achieve a logarithmic complexity for the search (on average). Note that this result isn't much of a surprise since the structure of a skip-list is very similar to the one of a tree, and search is handled like binary search. In fact, layer L_i adds nodes in the middle of the nodes of L_{i+1}.

Since this structure is fundamental for the skip-list to be efficient, we have to check how the insertion and the deletion can maintain this structure.

Insertion and deletion

We can introduce randomisation to keep the structure of a skip-list balanced. In particular, we let a random choice decide in which list, a part from the bottom one since an element is always added there, an element should be added. In particular, starting from the bottom, we decide, at random, if the node we added should be promoted to the upper layer. If the probability of promoting an element is 0.5, then half of the elements are promoted and we keep a structure with $\log n$ lists. The promotion process is repeated until the random choice decides the node shouldn't be promoted. Note that, the higher we move in the layers, the lower is the promotion probability. More precisely, the probability of being promoted to level i is

$$Pr(\text{promotion to level } i) = \left(\frac{1}{2}\right)^i$$

The deletion process is much simpler, in fact we simply have to remove the value from every list (i.e., at each layer) in which it appears.

If we add a special value $-\infty$ at the beginning of every list, we can build a well-balanced skip-list using insertion and deletion starting from an empty skip-list that contains only the value $-\infty$. Note that, adding a $-\infty$ node doesn't influence the search operation since it is smaller than any other value.

Skip-list performance

The following theorem holds an important result regarding the performance of a skip-list.

Theorem 5.3 (With high probability)**.** *With high probability, every search in an n-element skip-list costs $\mathcal{O}(\log n)$.*

Since the cost of the search operation is strictly related to the number of levels of the skip list, we can also state the following result.

Proposition 5.4 (With high probability)**.** *With high probability, an n-element skip list has $\mathcal{O}(\log n)$ levels.*

Before going on with the proof, let us state a theorem we need for demonstrating this theorem.

Theorem 5.4 (Boole's inequality)**.** *Let $E_1, \ldots, E_k$ be a set of random events. Then it holds*

$$Pr\{E_1 \cup \cdots \cup E_k\} \leq Pr\{E_1\} + \cdots + Pr\{E_k\}$$

This can be proved by induction, starting from $Pr\{E\} \leq Pr\{E\}$ and proving it for $n+1$ events.

Proof. Let us start by proving the proposition. In particular, let us write the probability of having $\mathcal{O}(\log n)$ levels as 1 minus the probability of having less than $\log n$ levels.

$$Pr\{\text{more than } c \log n \text{ levels}\} = 1 - Pr\{\text{at most } c \log n \text{ levels}\}$$

Let us write down the probability of having at most $c \log n$ levels (where c is a constant).

$$Pr\{\text{at most } c\log n \text{ levels}\} = \bigcup_{i=1}^{n} Pr\{\text{element } x \text{ promoted at least } c\log n \text{ times}\} \tag{5.1}$$

$$\leq n \cdot Pr\{\text{element } x \text{ promoted at least } c\log n \text{ times}\} \tag{5.2}$$

$$= n \cdot \frac{1}{2^{c\log n}} \tag{5.3}$$

$$= n \cdot \left(\frac{1}{2^{\log n}}\right)^{c} \tag{5.4}$$

$$= n \cdot \left(\frac{1}{n}\right)^{c} \tag{5.5}$$

$$= n \cdot \frac{1}{n^{c}} \tag{5.6}$$

$$= \frac{1}{n^{c-1}} \tag{5.7}$$

Where in 5.2 we have used Boole's inequality and in 5.3 we have used the fact that the probability of being promoted decreases with the number of levels. In particular, the probability of being promoted to level i is $(\frac{1}{2})^{i}$, hence the joint probability of being promoted at least $c \log n$ times is

$$Pr\{\text{element } x \text{ promoted at least } c\log n \text{ times}\} = (\frac{1}{2})^{1} \cdot (\frac{1}{2})^{2} \cdot \cdots \cdot (\frac{1}{2})^{c\log n}$$

It's clear to us that this isn't the result we expected but it's the only explanation we can find. Nevertheless, we can say that the probability of having at most $c \log n$ levels is smaller than $\frac{1}{n^{c-1}}$. This probability is polynomially small, i.e., at most n^{α} for $\alpha = c - 1$. This means that we can make α arbitrarily large by choosing the constant c in the $\mathcal{O}(\log n)$ bound accordingly. If the probability of having at most $c \log n$ levels is small, then the probability of having more than $c \log n$ levels is large. □

Before proving Theorem 5.3, we have to prove another important result that is used to prove Theorem 5.3.

Proposition 5.5. *The number of coin flips until obtaining $c \log n$ heads is $\Theta(\log n)$ with high probability.*

Proof. The number of coin flips required must be at least $c \log n$, hence it must be $\Omega(\log n)$ (i.e, $\log n$ is a lower bound for the number of flips). We only have to show that the number of coin flips required is $\mathcal{O}(\log n)$. Let's start by saying we do $10c \log n$ flips. The probability of having exactly $c \log n$ heads in $10c \log n$ attempts is

$$Pr(c\log n \text{ heads}) = \binom{10c\log n}{c\log n} \cdot \left(\frac{1}{2}\right)^{c\log n} \cdot \left(\frac{1}{2}\right)^{9c\log n}$$

which is the probability of having 1 head and 9 tails, considering every possible combinations of heads and tails (that's why we use the binomial). If we want at most $c \log n$ heads, the probability becomes

$$Pr(\text{at most } c\log n \text{ heads}) \leq \binom{10c\log n}{c\log n} \cdot \left(\frac{1}{2}\right)^{9c\log n}$$

which is only an overestimate. Thanks to the following bounds

$$\left(\frac{y}{x}\right)^x \leq \binom{y}{x} \leq \left(e\frac{y}{x}\right)^x$$

we can write the probability of having at most $c \log n$ heads as

$$\begin{aligned}
Pr(\text{at most } c\log n \text{ heads}) &\leq \binom{10c\log n}{c\log n} \cdot \left(\frac{1}{2}\right)^{9c\log n} \\
&\leq \left(e\frac{10c\log n}{c\log n}\right)^{c\log n} \cdot \left(\frac{1}{2}\right)^{9c\log n} \\
&\leq \left(e\frac{10c\log n}{c\log n}\right)^{c\log n} \cdot \left(\frac{1}{2}\right)^{9c\log n} \\
&= \big(10e\big)^{c\log n} \cdot \left(\frac{1}{2}\right)^{9c\log n} \\
&= \big(10e\big)^{c\log n} \cdot \big(2\big)^{-9c\log n} \\
&= \big(2^{\log(10e)}\big)^{c\log n} \cdot \big(2\big)^{-9c\log n} \\
&= 2^{\log(10e)\cdot c\log n} \cdot \big(2\big)^{-9c\log n} \\
&= 2^{\log(10e)\cdot c\log n - 9c\log n} \\
&= 2^{[\log(10e)-9]\cdot c\log n}
\end{aligned}$$

If we call $\alpha = -[\log(10e) - 9] \cdot c \log n$ we obtain

$$Pr(\text{at most } c\log n \text{ heads}) \leq \frac{1}{n^\alpha}$$

If we use consider a generic multiplier m instead of 10, the probability becomes

$$Pr(\text{at most } c\log n \text{ heads}) \leq \frac{1}{n^\alpha} = \frac{1}{2^{[9-\log(me)]\cdot c\log n}}$$

and the m goes to ∞, also α goes to ∞ and the probability is small. This means that we can choose m large enough to have a small probability. □

Proof. Let us now prove the first theorem. The idea is to analyse a search backwards (i.e., from a leaf to the root) and

1. If the node we are in (initially the node we want to find) wasn't promoted, we move left.
2. If the node we are in was promoted, we move up.

This is process is repeated starting from the node we are searching until we reach a node with value $-\infty$, which is the node at the beginning of the list. The number of times we move upwards is the number of layers of the list, since we analysed the search backwards and it started from the topmost layer with value $-\infty$. Thanks to Proposition 5.4 we know that the number of layers is, with high probability, smaller than $c \log n$. If we use a coin flip for choosing whether a node should be promoted (if head, the node is promoted), then the cost of the search is the number of coin flips required to get $c \log n$ heads (every head is a step up the layers, and we have done $c \log n$ steps up). This also has a $\mathcal{O}(\log n)$ cost, hence the whole path has a logarithmic complexity. □

Chapter 6

Dynamic programming

6.1 Introduction

Dynamic programming is a design technique like divide-and-conquer in which one needs to find the best decisions one after another.

6.2 Longest common subsequence problem

The longest common subsequence problem can be solved efficiently using dynamic programming. First, let us clarify what problem are we tackling. Say we have two sequences y and x of n and m elements each. We want to find one of the longest sequence in common to both x and y. Note that this problem is different from the longest common substring problem. In the latter case, the elements of the common sequence have to be consecutive in both sequences whilst in the former case not. Consider for instance the sequences

$$x = (A, B, C, B, D, A, B)$$

and

$$y = (B, D, C, A, B, A)$$

The longest subsequence is $BCBA$. Before going on, let us clarify a bit the notation we will use:

- The notation
$$x = (A, B, C, D)$$
indicates a sequence x containing values A, B, C and D.
- The notation
$$x[1, \ldots, m]$$
indicates a sequence x with indices going from 1 to m (both included).

Moreover, given a sequence s, we will indicate with $|s|$ the length of such sequence.

6.2.1 Brute force approach

The easiest, yet not efficient, way of solving this problem si by using a brute force approach. In particular, for every possible subsequence s of x, we want to check if s is also a subsequence of y. Since the set of possible subsequences of x is the powerset of x, there exist 2^m possible subsequences. If this isn't clear enough, imagine to represent a sequence of m elements as a bit-string (i.e., only 1s or 0s). Each string of m bits represents a possible subsequence; in particular, where we find a 1 we say that the corresponding element in the sequence is considered in the subsequence, otherwise the element isn't considered. If we take the x in the example we did before, the bit-string 0110011 represents the subsequence $BCAB$ (i.e., the sequence x where we didn't consider the first, third and fourth bit). Since we have 2^m possible bit-strings of m bits, we have the same number of subsequences. Every subsequence of x has to be matched against every character of y since every element of y could be the start of a subsequence. For this reason, the total complexity of this algorithm is

$$T_{lcs,bruteforce} = \mathcal{O}(n \cdot 2^m)$$

6.2.2 Dynamic programming approach

Instead of solving the longest common subsequence problem, we could try and solve an easier problem, given the solution of which it's easy to solve the former problem. In particular, we can

1. Find the **length** of the longest common subsequence.
2. Find the actual common longest subsequence, knowing its length.

Note that, solving the second problem given the solution to the first is very easy (we'll see it in practice).

Longest common subsequence length

Let us now focus on the first problem, that is, finding the length of the longest common subsequence. The idea for solving this problem is to consider the prefixes of x and y, which are subsequences of x and y, respectively. In other words, we can represents the prefixes of x as

$$x[1, \dots, i] \quad \forall i \in 1, \dots, m$$

and the prefixes of y as

$$y[1, \dots, i] \quad \forall i \in 1, \dots, n$$

Given two prefixes $x[1, \dots, i]$ and $y[1, \dots, j]$, we call $c[i, j]$ the length of the longest common subsequence of those prefixes,

$$c[i, j] = |\text{LCS}(x[1, \dots, i], y[1, \dots, j])|$$

The following theorem allows to define the number $c[i, j]$ recursively.

Theorem 6.1 (Longest common subsequence length)**.** *Given two sequences $x[1, \dots, m]$ and $y[1, \dots, n]$, the length of the longest common subsequence of any prefix $x[1, \dots, i]$ and $y[1, \dots, j]$ is*

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

Proof. Let us consider the case $x[i] = y[j]$, first. Let $z[1, \dots, k] = LCS(x[1, \dots, i], y[1, \dots, j])$ where $c[i, j] = k$. Then $z[k] = x[i]$, otherwise z could be extended. This is because, $x[i] = y[j]$, hence the longest subsequence must end with $x[i]$ (or $y[j]$, equivalently) otherwise we should find a $z[\hat{k}]$ with $\hat{k} > k$ for which $z[\hat{k}] = x[i]$.

Following $z[k] = x[i] = y[j]$, we can say that $z[1, \dots, k-1]$ is a common subsequence of $x[1, \dots, i-1]$ and $y[1, \dots, j-1]$. This is because, suppose that w is a longer common subsequence of $x[1, \dots, i-1]$ and $y[1, \dots, j-1]$, i.e., $|w| > k-1$. Then, $w.z[k]$ (where the dot represents the concatenation), is a common subsequence of $x[1, \dots, i]$ and $y[1, \dots, j]$ with $|w.z[k]| > k$ (since $|w| > k-1$). This contradicts the fact that $z[1, \dots, k] = LCS(x[1, \dots, i], y[1, \dots, j])$, hence it must be that $c[i-1, j-1] = k-1$ and $c[i, j] = c[i-1, j-1] + 1$. □

Let us now state another important result for dynamic programming, which can be used to solve the LCS length.

Theorem 6.2 (Optimal substructure)**.** *An optimal solution to a problem contains optimal solutions to sub-problems.*

Translated to our problem, this result tells us that, if $z = \text{LCS}(x, y)$, then any prefix of z is a longest common subsequence of a prefix of x and a prefix of y.

Given these results, we can define the algorithm for finding the length of the longest common subsequence of two sequences as in Algorithm 23.

Algorithm 23 The recursive algorithm for finding the length of the longest common subsequence of two sequences x and y.

```
procedure LCS(x, y, i, j)
    if i = 0 ∨ j = 0 then
        return 0
    end if
    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 procedure
```

If we were to execute this algorithm we would notice that many computations happens repeatedly. For instance, $\text{LCS}(x, y, 6, 5)$ can be called both when computing $\text{LCS}(x, y, 6, 6)$ and $\text{LCS}(x, y, 7, 5)$ (for computing the maximum). This is a problem since, better stated by the following theorem.

Theorem 6.3 (Overlapping sub-problems)**.** *A recursive dynamic programming solution contains a small number of distinct sub-problems repeated many times.*

If we consider our case, the number of different sub-problems is mn where $m = |x|$ and $n = |y|$. To solve this issue, we can memorise the results already computed and, before resolving a problem, we check if its solution has already been computed. This process is called memorisation.

In our case, since we have a couple of indices (i, j) we can save the results $c[i, j]$ in a $m \times n$ table so that we can swiftly obtain them. Algorithm 23 can be therefore rewritten, using this approach, as in Algorithm 24.

Algorithm 24 The recursive algorithm for finding the length of the longest common subsequence of two sequences x and y, using memorisation.

```
procedure LCS(x, y, i, j)
    if i = 0 ∨ j = 0 then
        return 0
    end if
    if c[i, j] = NIL 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
    return c[i, j]
end procedure
```

Since, in the worst case, we fill the entire table and computing the value for a cell has a constant cost, the total complexity is the total number of cells in the table,

$$T_{lcs-mem} = \Theta(mn)$$

We can say the same for space complexity, hence

$$S_{lcs-mem} = \Theta(mn)$$

Longest common subsequence

Now that we know how to compute the length of the longest common subsequence, it's easy to compute one of the longest common subsequences. In particular, we will exploit the matrix c containing the length values for each prefix. The algorithm, formalised in Algorithm 25, works as follows:

1. Start from the cell with the highest value. Initialise the longest common subsequence with the value of that cell.
2. If
   (a) If $x[i] = y[j]$ (where i and j are the indices of the current cell), then:
      i. Add an element in the head of the longest common sequence.
      ii. Follow the diagonal (i.e. move one step up and on the left).
   (b) otherwise, move left or up towards the cell with the higher value.
3. Go to point 3, unless we've reached a cell with value 0.

Note that if the number of rows is bigger than the number of columns, we move always to the left, otherwise we move always up. Since this algorithm traverses the whole table, it has a time complexity of $\Theta(nm)$. Adding this complexity to the one of finding the length of the longest common subsequence, we have that the total complexity of the algorithm is

$$T_{lcs} = \Theta(nm + nm) = \Theta(nm)$$

Algorithm 25 The algorithm for finding the longest common subsequence.

```
procedure LCS(x, y, i, j)
    if i = 0 ∨ j = 0 then
        return ""
    end if
    if x[i] = y[j] then
        return LCS(x, y, i − 1, j − 1).x[i]
    end if
    if c[i − 1, j] > c[i, j − 1] then
        return LCS(x, y, i − 1, j)
    end if
    return LCS(x, y, i, j − 1)
end procedure
```

Chapter 7

Algorithm analysis

7.1 Amortised analysis

Amortised analysis tries to compute the execution time of a program by putting more focus on its real world performance. This is obtained by analysing data structures instead of functions. More formally:

> **Definition 7.1** (Amortised analysis)**.** *An amortised analysis is any strategy for analysing a sequence of operations to show that the average cost per operation is small, even though a single operation within the sequence might be expensive.*

Note that even if we are considering the average case, no probability is involved when doing amortised analysis. In fact we are considering the average cost over a sequence of operations. Moreover, amortised analysis guarantees the average performance of each operation in the worst case.

The rational behind amortised analysis is that some operations on a data structures are very fast but sometimes require that an expensive operation is executed. If the expensive operation is executed very few times and the most frequent operations are very fast then we can still obtain a good average performance. Amortised analysis can be done in many ways. Some methods are:

- The **aggregate method**.
- The **accounting method**.
- The **potential method**.

7.1.1 Aggregate method

The aggregate methods finds an upper bound on the time complexity of a set of operations and then we can compute the worst case sequence of operations and find an upper bound for that. In other words, this method shows that, for all sequences of n operations, there is an aggregate cost $T(n)$ covering the worst-case cost for the sequence. The amortised cost of each operation is therefore

$$\frac{T(n)}{n}$$

The idea is then to spread the overall time over n operations.

Stack operations

Let us consider a stack with the usual operations of POP and $PUSH$. These operations take a constant time since they only need to put or remove a value on top of the stack. Let us also introduce the $MULTIPOP$ operation which pops a number of elements from the stack. Basically, the $MULTIPOP$ operation is implemented as in Algorithm 26. From this implementation we see

Algorithm 26 Multipop operation on a stack.

```
procedure MULTIPOP(S, k)
    while ¬STACK-EMPTY(S) ∧ k > 0 do
        POP(S)
        k ← k − 1
    end while
end procedure
```

that the complexity of a $MULTIPOP$ on stack S with parameter k is $\min\{|S|, k\}$ since we remove k elements or all the elements in the stack if $k > |S|$. As first approximation we can compute the worst complexity of n operations by taking the worst operation and repeating it n times. This means that the worst complexity is

$$T(n) = \mathcal{O}(n \cdot \min\{|S|, k\})$$

This is however a very loose bound since if we make n $MULTIPOP$ in sequence on an initially empty stack, each $MULTIPOP$ has constant complexity (checks that the stack is empty and returns immediately). If we want to do an amortised analysis we have to consider the real world worst case scenario which is a sequence of n $PUSH$es followed by a $MULTIPOP$ of size n. In this case the cost of the sequence is

$$T(n) = \sum_{i=1}^{n} 1 + \min\{|S|, n\} = 2n = \mathcal{O}(n)$$

This means that the amortised cost is

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

Counter increment

Let us consider a counter with k bits and the operation $INCREMENT$ that increments the counter as in Algorithm 27. This algorithm scans the vector and as soon as it encounters 1s, it flips them to 0s and stops when it finds a 0 which is flipped to 1 (without going out of bounds). In this case we have a single operation whose cost changes depending on the state of the counter. The worst case scenario is when the vector is filled with 1s hence we have to scan the whole vector which takes k steps. This means that the cost of doing n operations is

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

As before, this is a very loose bound since we don't always execute the increment on a vector filled with 1s, hence we have to find a real-world case. In particular, we have to compute how many times

Algorithm 27 The counter increment algorithm.

```
procedure INCREMENT(A, k)
    i ← 0
    while i < k ∧ A[i] = 1 do
        A[i] ← 0
        i ← i + 1
    end while
    if i < k then
        A[i] ← 1
    end if
end procedure
```

a bit is flipped. If we do n operations, the rightmost bit is flipped $\frac{n}{2}$ times, the one after it $\frac{n}{4}$ times until we reach the last which is flipped only once. This means that we obtain

$$T(n) = \sum_{i=1}^{k} \left\lfloor \frac{n}{2^i} \right\rfloor$$

We can now remove the floor and obtain an upper bound for the increment operation. If we take the number of operations outside the sum we obtain

$$T(n) = n \sum_{i=1}^{k} \frac{1}{2^i}$$

We can now recognise the geometric series which converges to $\frac{1}{1-\frac{1}{2}} = 2$, hence the complexity is

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

Now we can divide this complexity by n to obtain the amortised cost which is

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

7.1.2 Accounting method

The accounting method defines for each operation i an amortised cost $\hat{c}_i$ which is the amortised cost we want to pay for performing such operation. If we sum all amortised costs we obtain a budget, which we want to be smaller than the actual cost. Namely, we want

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

The difference between the amortised cost $\hat{c}_i$ and the real cost c_i can be saved and used later on to pay the cost of a very expensive operation (i.e., with a big cost c_i).

Stack operations

Let us consider the same example we did before with a stack and the $MULTIPOP$ operation. As we have seen before, the $PUSH$ and POP operations have a constant cost whilst the $MULTIPOP$

has a cost of $\min\{|S|, k\}$. Now we also have to define the amortised cost for each operation. Let's start with the $PUSH$. When we push an element on the stack we pay for the $PUSH$ but also for the POP that will eventually be done in the future, hence we pay a constant cost of 2, i.e., $\hat{c}_{push} = 2$. This means that the cost required for a POP is 0 since it has already been considered by the $PUSH$, namely we have $\hat{c}_{pop} = 0$. The same is true for the $MULTIPOP$ since every pop done by the function has already been paid, hence we have $\hat{c}_{multipop} = 0$.

Counter increment

The same analysis can be done with the counter increment example. In this case the idea is to look for the least significant 0 and:

- Pay 1 for each cell that we check.
- Pay 1 for the cell with the least significant 0.

7.1.3 Potential method

The idea behind the potential method is to assign to a data structure a function Φ that assigns a potential energy to any given state of the structure. The structure starts in a state D_0 and operation i moves the structure from state D_{i-1} to state D_i with cost c_i. The potential function

$$\Phi : \mathbf{D} \to \mathbb{R}$$

assigns a real number to each $D_i \in \mathbf{D}$ such that

- $\Phi(D_0) = 0$.
- $\Phi(D_i) \geq 0 \quad \forall i$.

The amortised cost $\hat{c}_i$ for operation i is defined as

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

Since we are dealing with potential energy, we can define a potential difference

$$\Delta\Phi_i = \Phi(D_i) - \Phi(D_{i-1})$$

and

- If $\Delta\Phi_i > 0$ then the amortised cost $\hat{c}_i$ is bigger than the actual cost c_i and we are saving some energy for a later operation.
- If $\Delta\Phi_i < 0$ then the amortised cost $\hat{c}_i$ is smaller than the actual cost c_i and we are using some of the store energy.

Moreover, if we sum the costs we obtain

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} \big(c_i + \Phi(D_i) - \Phi(D_{i-1})\big)$$

Now we can see that every potential apart the first and the final one cancels itself out, hence we are left with

$$\begin{aligned}\sum_{i=1}^{n} \hat{c}_i &= \sum_{i=1}^{n} \big(c_i + \Phi(D_i) - \Phi(D_{i-1})\big) \\ &= \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0) \\ &= \sum_{i=1}^{n} c_i + \Phi(D_n) && \Phi(D_0) = 0\end{aligned}$$

This means that we can define an upperbound for the sum of amortised cost, in fact, since $\Phi(D_i) \geq 0$ for all i, we have

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

Stack operations

In the stack example we can define the potential energy proportional to the size of the stack, namely we can define

$$\Phi(D_i) = |S|$$

Now that we have a potential function we can compute the amortised costs. The amortised cost for the $PUSH$ is

$$\begin{aligned}\hat{c}_{push} &= c_{push} + \Phi(D_i) - \Phi(D_{i-1}) \\ &= 1 + |S| - (|S| - 1) \\ &= 1 + |S| - |S| + 1 \\ &= 2\end{aligned}$$

The amortised cost for the POP is

$$\begin{aligned}\hat{c}_{pop} &= c_{pop} + \Phi(D_i) - \Phi(D_{i-1}) \\ &= 1 + (|S| - 1) - |S| \\ &= 1 + |S| - 1 - |S| \\ &= 0\end{aligned}$$

The amortised cost for the $MULTIPOP$ is

$$\begin{aligned}\hat{c}_{pop} &= c_{multipop} + \Phi(D_i) - \Phi(D_{i-1}) \\ &= \min\{|S|, k\} + (|S| - \min\{|S|, k\}) - |S| \\ &= \min\{|S|, k\} + |S| - \min\{|S|, k\} - |S| \\ &= 0\end{aligned}$$

As we can see the amortised costs correspond to the ones found with the accounting method.

Counter increment

In the counter example we can define the potential energy as the energy required to flip 1s to 0s, namely

$$\Phi(D_i) = \text{number of bits equal to 1}$$

Let us also call $p(i)$ the number of bits that are put from 1 to 0 with increment i.

7.1.4 Hash table example

An hash table is a structure that deals with an arbitrary large amount of data, hence it must be able to grow when new data is added. Usually, an hash table of a fixed size is created and then, whenever it's filled up, a new bigger hash table (usually double the size of the original one) is allocated and the data in the former is moved to the latter.

Accounting method

Let's try to compute the amortised cost using the accounting method. Whenever the hash table is full, we have to allocate a new hash table and copy the data. We can cover this costs by paying a cost of 3 every time we insert an element. This cost is the sum of:

- The cost for inserting the element.
- The cost for moving the element inserted and an element already present in the table.

In this analysis we aren't considering the cost of allocating the actual memory. The actual cost for inserting an element is

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

Thanks to this cost we can compute the total cost as

$$\begin{aligned} T(n) &= \sum_{i=1}^{n} c_i \\ &\leq n + \sum_{j=0}^{\lfloor \log(n-1) \rfloor} 2^j \\ &\leq 3n = \mathcal{O}(n) \end{aligned}$$

7.2 Competitive analysis

Competitive analysis assess the complexity of operations on a data structure. Competitive analyses uses an adversary that, at each step of a sequence of operations, selects the operation that maximises the total cost.

7.2.1 Self-organising list

Let's start by analysing a self-organising list, namely a list of elements, implemented as a linked list, and every time we have an access to the list, we reorganise in order to minimise future accesses to the list. We can define a policy to organise the list which, given two items in the list, tells if the items should be swapped. Let us also assume that a swap between any two elements has a constant cost. The final goal is to find a policy at runtime that optimises the total cost of a sequence of operations. Policy optimisation has to be done **online**, hence we don't know the list of elements that will be added to the list but we can only see one element at a time. The policy can then be compared against an **offline** policy, i.e., a policy defined looking at the whole sequence of instructions.

Competitive analysis computes the total cost by using an adversary that, at each step, selects an element that maximises the total cost. In our example, the adversary selects always the last element of the list, which is the one for which we need more time.

Move to front heuristic

The first technique we can use to optimise the total cost of using a self-organising list is the move-to-front heuristic that moves to the head of the list the last accessed element and shifts the others towards the tail. This solution exploits time locality but requires us to pay always twice the rank of the accessed element (i.e., the distance to the beginning of the vector).

We can use the α-competitiveness to understand how good this policy is. This technique states that

> **Definition 7.2** (Alpha-competitiveness)**.** *An algorithm A is α-competitive if there exist a constant k such that, for any sequence S of operations, the cost $C_A(S)$ of accessing the elements in the sequence is bounded by some constant α multiplied by the overall cost $C_{opt}(S)$ required by the optimal offline algorithm plus some constant k, namely if*
>
> $$C_A(S) < \alpha \cdot C_{opt}(S) + k$$

We can show that the the move-to-front heuristic is 4-competitive for self organising list. Let's start by stating the cost of moving an element from its position to the front (and shift the others towards the tail). In general, this operation requires to swap the searched element x with its predecessor until we reach the beginning of the list. Since element x is in position $\text{rank}_L(x)$, we have a cost of

$$T_{move}(n) = \text{rank}_L(x)$$

If we also consider access to the list, we have to add $\text{rank}_L(x)$ since we have to scan the whole list until we reach x. The full cost of accessing x and moving it to the front is therefore

$$T_{accessandmove}(n) = \text{rank}_L(x) + \text{rank}_L(x) = 2\text{rank}_L(x)$$

Let us now call L_i the list after the i-th access (updated using the move-to-front policy) and L_i^* the list after the i-th access (updated using the optimal offline policy). Let c_i be the cost of the i-th access on the list handled using MTF and c_i^* the cost of the i-th access on the list handled using the optimal policy. As we have seen, the first cost is

$$c_i = 2 \cdot \text{rank}_{L_{i-1}}(x)$$

whereas the optimal cost is

$$c_i = \text{rank}_{L_{i-1}^*}(x) + t_i$$

where t_i is the number of swaps required after executing operation i, according to the optimal offline policy. Now we can define a potential function, as we did for amortised analysis as

$$\Phi(L_i) = 2 \cdot |\{(x, y) : x \prec_{L_i} y \wedge y \prec_{L_i^*} x\}|$$

which is twice the number of elements that are in the wrong order in list L_i with respect to L_i^*. This is a potential since:

- Starting from the same list, the potential is 0, since L_0 and L_0^* have the same elements in the same positions.
- The potential is the cardinality of a set, hence it's always a non-negative number.

Let us now suppose that operation i accesses element x. We can now split a list in four sections:

- $A = \{y \in L_{i-1} : y \prec_{L_{i-1}} x \wedge y \prec_{L_{i-1}^*} x\}$
- $B = \{y \in L_{i-1} : y \prec_{L_{i-1}} x \wedge y \succ_{L_{i-1}^*} x\}$
- $C = \{y \in L_{i-1} : y \succ_{L_{i-1}} x \wedge y \prec_{L_{i-1}^*} x\}$
- $D = \{y \in L_{i-1} : y \succ_{L_{i-1}} x \wedge y \succ_{L_{i-1}^*} x\}$

This means that we can represent L_{i-1} as $(A \cup B).x.(C \cup D)$ and L_{i-1}^* as $(A \cup C).x.(B \cup D)$. In this scenario, the rank of x is

$$r = |A| + |B| + 1$$

in L_{i-1} and

$$r^* = |A| + |C| + 1$$

in L_{i-1}^*. When we access element x using the MTF policy, we create $|A|$ inversions (since we move values from before to after x) and we delete $|B|$ inversions (since we move values from before x to after x, where they belong). Each transpose by the optimal policy creates at most 1 inversion, hence t_i inversions in total. This means that we can write

$$\Phi(L_i) - \Phi(L_{i-1}) \leq 2(|A| - |B| + t_i)$$

Thanks to this inequality we can write the amortised cost for operation i as

$$\begin{aligned}
\hat{c}_i &= c_i + \Phi(L_{i-1}) - \Phi(L_i) \\
&\leq 2r + 2(|A| - |B| + t_i) \\
&= 2r + 2(|A| - |B| + t_i) \\
&= 2r + 2(|A| - (r - 1 - |A|) + t_i) && r = |A| + |B| + 1 \\
&= 2r + 2(2|A| - r + 1 + t_i) \\
&= 2r + 4|A| - 2r + 2 + 2t_i \\
&= 4|A| + 2 + 2t_i \\
&\leq 4(r^* + t_i) && r^* = |A| + |C| + 1 \geq |A| + 1 \\
&= 4c_i^*
\end{aligned}$$

Thanks to the amortised cost we can compute the total cost of a sequence S of $|S|$ operations as

$$\begin{aligned}
c_{MTF}(S) &= \sum_{i=1}^{|S|} c_i \\
&= \sum_{i=1}^{|S|} \big(\hat{c}_i + \Phi(L_{i-1}) - \Phi(L_i)\big) \\
&\leq \sum_{i=1}^{|S|} 4c_i^* + \Phi(L_0) - \Phi(L_{|}S|)\big) \\
&\leq 4\sum_{i=1}^{|S|} 4c_i^* && \Phi(L_0) = 0, \Phi(L_i) \geq 0 \\
&\leq 4c_{OPT}(S)
\end{aligned}$$

This means that the move-to-front heuristic is 4-competitive with respect to the optimal offline policy.

Part III

Parallel algorithms and parallel programming

Chapter 8

Introduction

Parallelisation is the idea of dividing a complex task into smaller and simpler tasks that can independently be executed on multiple machines (or cores) in parallel.

8.1 Motivations

Building parallel programs is hard since we have to shift our way of thinking about a program (we usually think of a program as a sequence of instructions, i.e., we think in a sequential way), however, it has several advantages:

- It's **time saving**. Modern computers have multiple processing elements which we can exploit to run tasks in parallel and reduce the time required to run an algorithm.
- It's **money saving**. An architecture made of many small processors can be cheaper than one made of one single powerful CPU.
- It allows to **solve Grand Challenge problems**. These are problems that can be solved in a reasonable amount of time only by exploiting parallelism.

Moreover, we always have to remember Moore's law. This law states that the number of transistors in a processor doubles approximately every 24 months. Whilst this is true, putting more transistors in a CPU means using more energy, hence this growth rate has reduced. To keep increasing the number of transistors, we have to take simpler cores and join them to build multi-core processors. This allows the number of transistors to grow (following Moore's law) but without increasing the power consumption and the frequency of a single core.

8.2 Automatic and manual parallelisation

Parallelisation can be split in:

- **Automatic parallelisation**. In this case, a machine can automatically take a task and make it parallel without the intervention of a human.
- **Manual parallelisation** (or by hand). In this case, a human has to take a task and build a parallel program out of it.

The former choice is much more difficult to use since we lack the instruments to automatically parallelise programs. In particular, there are some problems that compilers can't figure out how to solve, hence human intervention is required.

8.2.1 Automatic parallelisation

When using automatic parallelisation we have to:

1. Write a sequential algorithm.
2. Translate the sequential algorithm into a sequential high-level code.
3. Give the sequential code to a tool that automatically generates the parallel assembly implementation of the sequential algorithm we wrote in point 1.

Automatic translation done by compilers isn't however easy since there are problems it can't solve. Consider for instance the piece of code in Listing 8.1.

```
1 void and_or(int SIZE, int* and, int* or, int* b, int* c) {
2     int i = 0;
3     for (i=0; i<SIZE; i++)
4         and[i] = b[i] & c[i];
5     for (i=0; i<SIZE; i++)
6         or[i] = b[i] | c[i];
7 }
```

Listing 8.1: Code that has to be parallelised.

One issue could be that the four arrays passed to the function overlap. Consider for instance that we use the function as follows:

```
int array_a[10], array_b[10];

and_or(9, &array_a[1], &array_a[1], array_b, array_a);
```

If we don't tell the compiler that some inputs overlap, it might just execute the two loops in parallel and compute a wrong result (i.e., a result different from the one obtained by executing the code in a sequential way). In C and C++ the `restrict` keyword has been introduced to tell the compiler that the two pointers are not overlapped, however, this doesn't solve the problem of automatic parallelisation, since what we've done is just an example of how things can go wrong. The main issue in C and C++ is that the compiler has to deal with pointers and it can't extract from the code the key information required to correctly parallelise the program. In recent years, new parallelisation-friendly languages have been introduced to allow compilers to automatically parallelise a program, however they lack the background and testing of older and more solid languages, hence they struggle to gain popularity.

8.2.2 Parallelisation by hand

Since a compiler can't infer key information from the code, we have to give them to it. When building a parallel program we have to take care of three details:

- Which type of parallelism we want to use.
- How we want to design the algorithm.
- How we should provide the compiler information to deal with parallelisation.

In particular, there exist two ways of implementing parallel code:

- Starting from sequential code we can analyse dependencies between instructions and recurrent patterns (for which we already know the parallel implementation) and then write the parallel code.
- Rewrite completely the algorithm starting directly from the parallel code.

8.3 Types of parallelism

8.3.1 Data and instruction parallelism

When dealing with parallelism, we have to consider:

- **Data parallelism**.
- **Instruction parallelism**.

This means that we can parallelise one (i.e., an instruction executed in parallel on multiple data), both (multiple instructions in parallel on multiple data) or none (i.e., sequentially). In particular, we have four different combinations:

- **Single Instruction Single Data** (SISD) parallelism. In this type of parallelism, an instruction is applied to a single data.
- **Single Instruction Multiple Data** (SIMD) parallelism. In this type of parallelism, which is the most common, a single instruction is applied to different data. This is what happens in GPUs. In this class, execution is synchronised and deterministic, namely, the same instruction is executed at the same time on different data.
- **Multiple Instructions Single Data** (MISD) parallelism. In this type of parallelism, we apply multiple different instructions on a single data. This is much harder to achieve than SIMD. An example of MISD is multimedia elaboration where a single stream of data passes through multiple steps of elaboration.
- **Multiple Instruction Multiple Data** (MIMD) parallelism. In this type of parallelism, we can apply different instructions to different data in parallel. All multi-core systems fall in this class.

8.3.2 Level of parallelism

When dealing with parallelism we also have to consider the size of the objects that we want to parallelise. In particular, we have:

- **Bit level parallelism**. This type of parallelism encloses all hardware implementations of instructions that act on multiple bits in parallel (e.g. a bit-wise or on a 32-bit word).
- **Instruction level parallelism**. This type of parallelism encloses all instructions that are not dependent on one other hence they can be run in parallel. Compilers can easily understand if instructions have dependencies and processors can use out-of-order execution to change the order in which instructions are executed (if they don't share a dependency).

- **Task level parallelism**. In this case, we parallelise multiple instructions (which we'll call a task). This means that a sequence of instructions is executed in parallel with another sequence of instructions, without putting any constraints on the single instructions of the two tasks.

The program in Listing 8.2 contains all three classes of parallelism, in particular:

- The bit-wise operation between the values in the vector is an example of bit-level parallelism.
- Each iteration of the for loops is an example of instruction level parallelism since one instruction is independent of the other (vectors do not overlap thanks to the `restrict` keyword).
- Each loop is an example of task-level parallelism since every loop (which is a sequence of instructions, hence a task) writes different regions of memory, hence it can be run in parallel.

```
1 void and_or(int SIZE, int* restrict and, int* restrict or, int* restrict b, int*
      restrict c) {
2     int i = 0;
3     for (i=0; i<SIZE; i++)
4         and[i] = b[i] & c[i];
5     for (i=0; i<SIZE; i++)
6         or[i] = b[i] | c[i];
7 }
```

Listing 8.2: Code that has to be parallelised.

Note that, especially in cases like this, creating a task might not have many advantages since it'll execute a very simple job and it will require some code to set up the tasks. This means that, if we can't optimise the code executed by a task, the parallelised code will perform worse since the tasks combined execute at the same time as the two loops executed sequentially, but we need some more time to set up the tasks (hence we have an overhead in the parallelised code). Note that the only case in which we have no overhead is when we use vectorisation.

Parallel task graph

When dealing with task-level parallelism, we have to understand the dependencies between them. One way of representing such dependencies is by using a graph where:

- Nodes represent tasks.
- Edges represent dependencies. In particular, an arc from T_1 to T_2 can represent different things:
  - A data relation, namely T_2 uses a data generated by T_1 (in a producer-consumer fashion).
  - A precedence relation, namely T_1 has precedence (i.e., should be executed before) T_2.

Using this graph we can say that a task T is executed only if all the tasks with an arc directed to T have completed their execution. Note that in some cases even a graph with no bifurcations (i.e., a task is connected only to its successor) can be seen as a parallel executor, in fact, if we add memories before each task, we can represent a pipeline, which is able to execute different instructions in each of its stages.

Communication

When dealing with tasks, they can exchange data and communicate. Communication can be achieved in two ways:

- Using **shared memory**. Memory is shared between different tasks that run in parallel so that they can read and write from and to it in parallel.
- Using **message passing**. Some data is passed from one task to the other using messages that are delivered by a runtime. In this case, all tasks are working in their own memory space (hence no memory is shared).

8.4 Parallel and sequential programming

Programs written in a sequential way are very easy to be compiled and ported from one architecture to the other, even if these architectures are very complex and different. On the other hand, things get complicated when we deal with parallel programming. For parallel programming we can use two different approaches:

- Use new programming languages custom-made for parallel programming. This approach isn't very popular since compilers for such languages aren't that advanced and it's required to learn a new programming language.
- Extend existing languages. This solution is better since we can leverage the existing tools for the programming language and we only have to extend them (e.g., compilers are already built and well tested, we only have to extend them). Unfortunately, we can describe only some types of parallel paradigms using these languages.

The list of parallel programming languages is long, and every language has a different approach to the problem:

- **Actor model**. Some examples are: Axum, Elixir, Erlang, Janus, Red, SALSA, Scala/Akka, Smalltalk, Akka.NET.
- **Coordination languages**. Some examples are: CnC, Glenda, Linda, coordination language, Millipede.
- **Dataflow programming**. Some examples are: CAL, E, Joule, LabView, Lustre, Preesm, Signal, SISAL, BMDFM.
- **Distributed computing**. Some examples are: Bloom, Hermes, Julia, Limbo, MPD, Oz, Sequoia, SR.
- **Event-driven and hardware description**. Some examples are: Esterel, SystemC, SystemVerilog, Verilog, Verilog-AMS, VHDL.
- **Functional programming**. Some examples are: Clojure, Concurrent ML, Elixir, Erlang, Futhark, Haskell, Id, MultiLisp, SequenceL, Elm.
- **Logic programming**. Some examples are: Parlog, Prolog.
- **Monitor-based**. Some examples are: Concurrent Pascal.

- **Multi-threaded**. Some examples are: C=, Cilk, Cilk Plus, C#, Clojure, ParaSail, Rust, SequenceL.
- **Object-oriented programming**. Some examples are: μC++, Ada, C*, C#, C++ AMP, Charm++, D Programming Language, Eiffel SCOOP, Emerald, Java, Join Java, ParaSail, Smalltalk.
- **Partitioned Global Address Space (PGAS)**. Some examples are: Chapel, Coarray Fortran, Fortress, High Performance Fortran, Titanium, Unified Parallel C, X10, ZPL.
- **Message passing**. Some examples are: Ateji, Rust.
- **Communicating Sequential Processing**. Some examples are: JCSP, Alef, Ease, FortranM, Go, JoCaml, Joyce, Limbo, Newsqueak, Occam, Occam-π, PyCSP, SuperPascal, XC.
- **APIs/Frameworks**. Some examples are: Apache Hadoop, Apache Spark, Apache Flink, Apache Beam, CUDA, OpenCL, OpenHMPP, OpenMP.

Tables 8.1, 8.2, 8.3 8.4 and 8.5 show some characteristics of the most popular parallel programming languages.

Language	Bit	Instruction	Task
Verilog/VHDL	Yes	Yes	No
MPI	(Yes)	(Yes)	Yes
PThread	(Yes)	(Yes)	Yes
OpenMP	(Yes)	(Yes)	Yes
CUDA	(Yes)	No	(Yes)
OpenCL	(Yes)	No	Yes
Apache Spark	(Yes)	No	(Yes)

Table 8.1: Supported levels of parallelism.

Language	SIMD	MISD	MIMD
Verilog/VHDL	Yes	Yes	Yes
MPI	Yes	Yes	Yes
PThread	Yes	(Yes)	Yes
OpenMP	Yes	Yes	Yes
CUDA	Yes	No	(Yes)
OpenCL	Yes	(Yes)	Yes
Apache Spark	Yes	No	No

Table 8.2: Supported types parallelism.

8.4.1 Granularity

Different languages can offer different levels of granularity. Granularity is typically related to the overhead required to handle parallelism (e.g., creating a task). More precisely,

- Languages with low overhead (like Verilog, CUDA or VHDL) offer fine-grained parallelism.
- Languages with a high overhead (like Apache Spark, MPI or Pthread) offer coarser parallelism.

Language	Processors	Memory
Verilog/VHDL	ASIC or FPGA	
MPI	Multi CPUs	Distributed memory
PThread	Multi-core CPUs	Shared memory
OpenMP	Multi-core CPUs	Shared memory
CUDA	CPUs and GPUs	Distributed shared memory
OpenCL	Heterogeneous architectures	Distributed shared memory
Apache Spark	Multi CPUs	Distributed memory

Table 8.3: Target architectures of parallel programming languages. Note that distributed memory means that each CPU has its own address space.

Language	Parallelism	Communication
Verilog/VHDL	Explicit	Explicit
MPI	Implicit	Explicit
PThread	Explicit	Implicit
OpenMP	Explicit	Implicit
CUDA	Implicit (also explicit)	Implicit (also explicit)
OpenCL	Explicit and implicit	Explicit and implicit
Apache Spark	Implicit	Implicit

Table 8.4: Target architectures of parallel programming languages.

8.4.2 Overview of different parallel programming languages

Verilog/VHDL

Pros:

- Complete control on computation and memory.
- No overhead is introduced in the computation.
- Provides access to potentially large computational power.

Cons:

- Requires specific Hardware (e.g., ASIC or FPGA) to implement the functionality.

Language	Target independent code	Development platforms
Verilog/VHDL	Yes (behavioural), No (structural)	Linux
MPI	Yes	All
PThread	Yes	All
OpenMP	Yes	All
CUDA	Depends on CUDA capabilities	All
OpenCL	Yes	All
Apache Spark	Yes	Linux

Table 8.5: Target architectures of parallel programming languages.

- Difficult to learn: completely different programming language and programming paradigm.
- Depends on the chosen target architecture.

MPI

Pros:

- Works well on distributed systems.
- Can be adopted on different types of architectures.
- Scalable solutions.
- Synchronisation and data communication are explicitly managed.

Cons:

- Communication can introduce significant overhead.
- Programming paradigm is more difficult than shared memory-based ones.
- Standard does not reflect immediate advances in architectural characteristics.

PThread

Pros:

- Can be adopted on different architectures.
- Explicit parallelism and full control over applications.

Cons:

- Task management overhead can be significant.
- Not easily scalable solutions.
- Low-level API.

OpenMP

OpenMP has been introduced to describe parallelism at a very high level (hence not for programmers)
Pros:

- Easy to learn.
- Scalable solution.
- Parallel applications can also be executed sequentially.

Cons:

- Mainly focused on shared memory.
- homogeneous systems.
- Require small interaction between tasks.

CUDA

Pros:

- Provides access to the computational power of GPUs.
- Writing a CUDA kernel is quite easy.
- Already optimized libraries.

Cons:

- Targets only NVIDIA GPUs.
- Difficult to extract massive parallelism from applications.
- Difficult to optimize CUDA kernel.

OpenCL

Pros:

- Target-independent standard.
- Hides architecture details.
- Same programming infrastructure for very heterogeneous architectures.

Cons:

- Difficult programming paradigm for its heterogeneity.
- Hiding of architectural details makes it difficult to obtain the best performances.

Apache Spark

Pros:

- API for different languages.
- Explicit parallelisation and communication are not required.
- Preinstalled on cloud provider VMs.

Cons:

- Suitable only for big data applications.
- Does not (yet) fully support GPUs.

Halide

Halide is a domain-specific language. Its focus is accelerating a program on a heterogeneous system. Namely, in a system with many different computing elements, we have a language for each element that describes the program running on that element. This language is meant for a very specific application.

8.4.3 Mixing parallelism technologies

Parallel programming languages can be used jointly to exploit the different strengths of each technology.

OpenMP and CUDA

By using OpenMP and CUDA, we can handle parallelism both on the CPU and GPU side. In particular, the former can be used for parallelising CPU code whilst the latter is for GPU code.

MPI and OpenMP

OpenMP can be used on a single node and communication between different nodes can be handled by MPI. Alternatively, MPI can express coarser parallelism (multi-CPU) and OpenMP to express finer parallelism (multi-core).

OpenCL and Verilog

OpenCL can be used to describe parallelism among different processing elements while hardware accelerators implemented on FPGAs are described by Verilog or VHDL.

Chapter 9

Overview of parallel patterns

9.1 Dependencies

When parallelising a program we have to understand which parts can be executed at the same time. We can have parallelism related to:

- The **control**, namely the order in which instructions are executed.
- The **data**, namely how an instruction can be applied to different data.

If we want to parallelise an application we have to understand the dependencies between the tasks in term of flow of the data. Dependencies can be divided into:

- **True flow dependencies** δ. An instruction I_2 has a true dependency on a previous instruction I_1 if I_2 uses (reads) a value computed (written) by I_1, and we write

  $$I_1 \ \delta \ I_2$$

  For instance, we have a true dependency in the following program:

  ```
  I1: a=1
  I2: b=a
  ```

- **Output dependencies**. An instruction I_2 has an output dependency on a previous instruction I_1 if I_2 redefines (writes) a value computed by I_1, and we write

  $$I_1 \ \delta^0 \ I_2$$

  For instance, we have a true dependency in the following program:

  ```
  I1: a=c
  I2: a=b
  ```

- **Anti-dependencies** δ^{-1}. An instruction I_2 has dependency on a previous instruction I_1 if I_2 redefines (writes) a value read by I_1, and we write

  $$I_1 \ \delta^{-1} \ I_2$$

  For instance, we have a true dependency in the following program:

```
I1: a=b
I2: b=1
```

This is a dependency because if we swap I_1 and I_2 the result of the program changes.

9.1.1 Dependencies graphs

We can represent dependencies between instructions using a graph in which:

- Nodes represent instructions.
- Edges represent dependencies. In particular, an arc from I_1 to I_2 means that I_2 has a dependency on I_1.

For each node I in the dependencies graph, we can compute:

- The $IN(I)$ set, which is the set of memory locations (variables) that may be used by instruction I.
- The $OUT(I)$ set, which is the set of memory locations (variables) that may be modified (i.e., written) by instruction I.

Thanks to this sets, it's possible to compute the dependencies between instruction. In particular, given two instructions I_1 and I_2 connected by an arc, we have:

- A true flow dependence between I_1 and I_2 if

$$OUT(I_1) \cap IN(I_2) \neq \varnothing$$

- An output dependence between I_1 and I_2 if

$$OUT(I_1) \cap OUT(I_2) \neq \varnothing$$

- An anti-dependence between I_1 and I_2 if

$$IN(I_1) \cap OUT(I_2) \neq \varnothing$$

As we can see, this control is easy to automatise, hence it's easy for compilers and automatic tools to check the dependencies between instructions.

9.1.2 Loop-level parallelism

Loops can be parallelised if the instructions inside a loop depend only on the current iteration and not on the other iterations. In this case, we talk about **doall** loops or **foreach** loops. In doall loops, all statements can be executed in parallel. Some examples of doall loops are

```
for(i=0; i<100; i++) {
    a[i] = i;
}
```

and

```
for (i=0; i<100; i++) {
    a[i] = i;
    b[i] = 2 * i;
}
```

To understand how to parallelise a doall loop, we can draw the iteration space using dependency graphs. Since every iteration is independent from the other, we can unroll the loop and build a dependency graph for each iteration.

Note that, even if a loop has some dependencies, we might still be able to parallelise it. Consider for instance the piece of code

```
for (int i=0; i<=14; i++) {
    a[i] = a[i+5];
}
```

This loop has some dependecies since we write data from other iterations, however we can still parallelise it. Let us unroll the loop to understand how.

```
a[0] = a[5];
a[1] = a[6];
a[2] = a[7];
a[3] = a[8];
a[4] = a[9];
a[5] = a[10];
a[6] = a[11];
a[7] = a[12];
a[8] = a[13];
a[9] = a[14];
a[10] = a[15];
a[11] = a[16];
a[12] = a[17];
a[13] = a[18];
a[14] = a[19];
```

As we can see, the first instruction depends only on the sixth and the eleventh. This is true for any other instruction which depends only on two other instructions, hence we can split the instructions in groups such that each group contains the instructions with dependencies and then execute the groups in parallel. In code, we would write:

```
for (int i=0; i<5; i++) {
    a[i] = a[i+5];
    a[i+5] = a[i+10];
    a[i+10] = a[i+15];
}
```

Loop dependencies classification

In general, loop dependencies can be split into:

- **Loop-carried** dependencies. A loop-carried dependence is a dependence between two instructions of different loop iterations.
- **Loop-independent** dependencies. A loop-independent dependence is a dependence between two instruction that are not part of different iterations, including dependencies between two statements instances in the same loop iteration.

Moreover:

- A dependency is **lexically forward** if the source comes before the target.
- A dependency is **Lexically backward** if the target comes before the source.

9.2 Nesting pattern

The nesting pattern is used when data can be described in a hierarchical way. This means that a system can be seen as a composition of blocks and each block can be further detailed.

9.3 Serial control patterns

Serial control patterns allow to describe a structured serial program. In particular, we can use four patterns:

- The **sequence pattern**.
- The **selection pattern**.
- The **iteration pattern**.
- The **recursion pattern**.

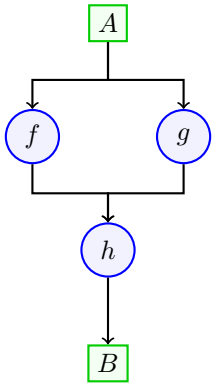
The nesting pattern can also be used to hierarchically compose these four patterns.

9.3.1 Sequence pattern

A sequence pattern describes an ordered list of tasks that are executed in a specific order. Looking at the hierarchical representation of a program we can immediately understand what instructions can be parallelised. Consider for instance the instructions

```
T = f(A);
S = g(A);
B = h(S, T)
```

that generate the following diagram



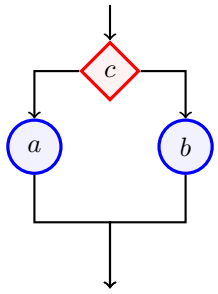
As we can see from the diagram, the functions f and g can be executed in parallel after A has been executed.

9.3.2 Selection pattern

The selection pattern is used when we want to decide what instruction block has to be executed depending on a condition c. For instance, the instructions

```
if (c) {
    a();
} else {
    b();
}
```

are represented as

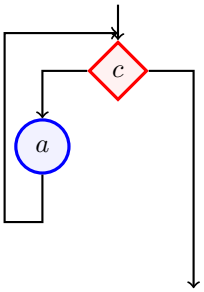


9.3.3 Iteration pattern

The iteration pattern is used when we want to express loops. For instance, the instructions

```
while(c) {
    a();
}
```

are represented as



Finding parallelism in a iteration pattern is harder than the previous cases since we have to consider dependencies inside an iteration and between different iterations.

9.3.4 Recursion pattern

The recursion pattern is used whenever a block calls itself. Sometimes recursion can be removed and transformed in a loop when the recursive function is a tail recursive function.

9.4 Parallel control patterns

Parallel control patterns extend parallel control patterns. Some extensions deal with data parallelism, others with control-flow parallelism. The main parallel control patterns are:

- **Fork-join**.
- **Map**.
- **Stencil**.
- **Reduction**.
- **Scan**.
- **Recurrence**.

9.4.1 Fork-join

The fork-join pattern extends the sequence patterns. In particular, we have a starting point and a bifurcation that allows to define two block that can run in parallel. After forking the control-flow, the tasks executing in parallel have to join. More precisely, the tasks can be joined in two different ways:

- With a **sync** that waits all tasks and allow only one to continue.
- With a **barrier** that waits all the tasks and restarts them allowing all of them to continue.

9.4.2 Map

The map pattern is a pattern in which the input is a collection of data to which an elemental function is applied. Namely, the same elemental function is applied to every element in the collection of data to obtain a collection as output. A graphical representation of the map pattern is shown in Figure 9.1.

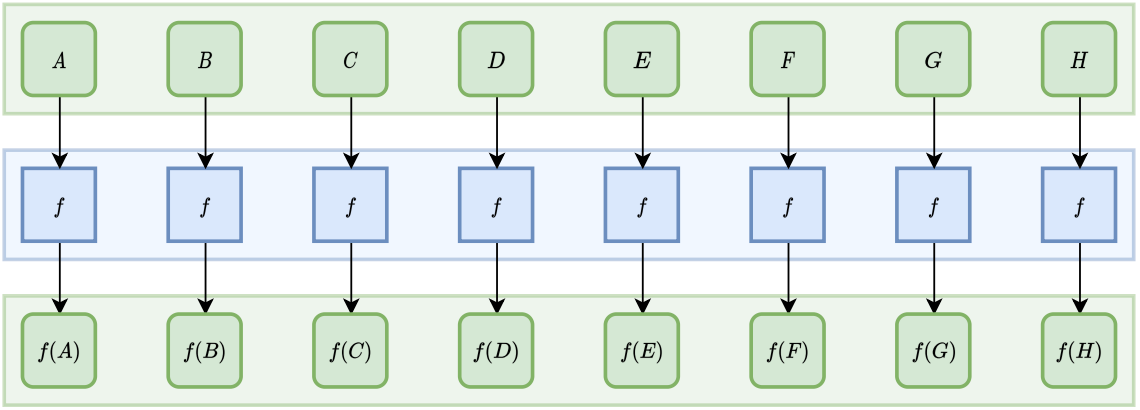


Figure 9.1: Map pattern.

The elemental function is the key block of the map pattern and it's a simple function whose output should only depend on the input and not on an internal state, otherwise it'd be impossible to parallelise the elemental function. In fact, if the output depends only on the input, we can apply the elemental function on each element of the input collection in parallel.

9.4.3 Stencil

The stencil pattern is an extension of the map pattern. In particular, the elemental function is applied to an input element and some neighbours of the input. An example is convolution. Handling parallelisation in this case is more complex since the output depends on values which are also used by other tasks executing in parallel.

9.4.4 Reduction

The reduction pattern uses a function, called **combiner function**, that computes a single value from the values of a collection. Some examples of combiner functions are the sum or product of the elements of a vector. The only constraint of the combiner function is that it must be associative, otherwise it would be impossible to compute the output exploiting parallelisation.

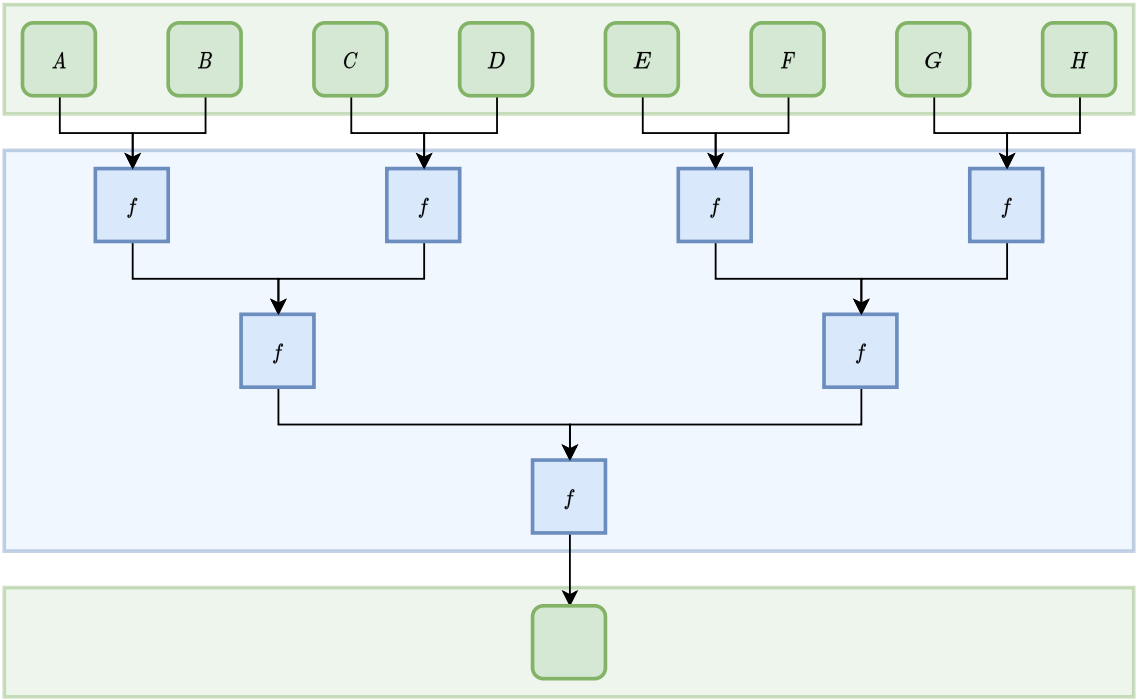


Figure 9.2: The reduction pattern.

9.4.5 Scan

The scan pattern is a specific case of the reduction pattern. For every output in a collection, a reduction of the input up to that point is computed. As for the reduction pattern, if the function

being used is associative, the scan can be parallelised. Parallelising a scan is not obvious at first, because of dependencies to previous iterations in the serial loop. A parallel scan will require more operations than a serial version.

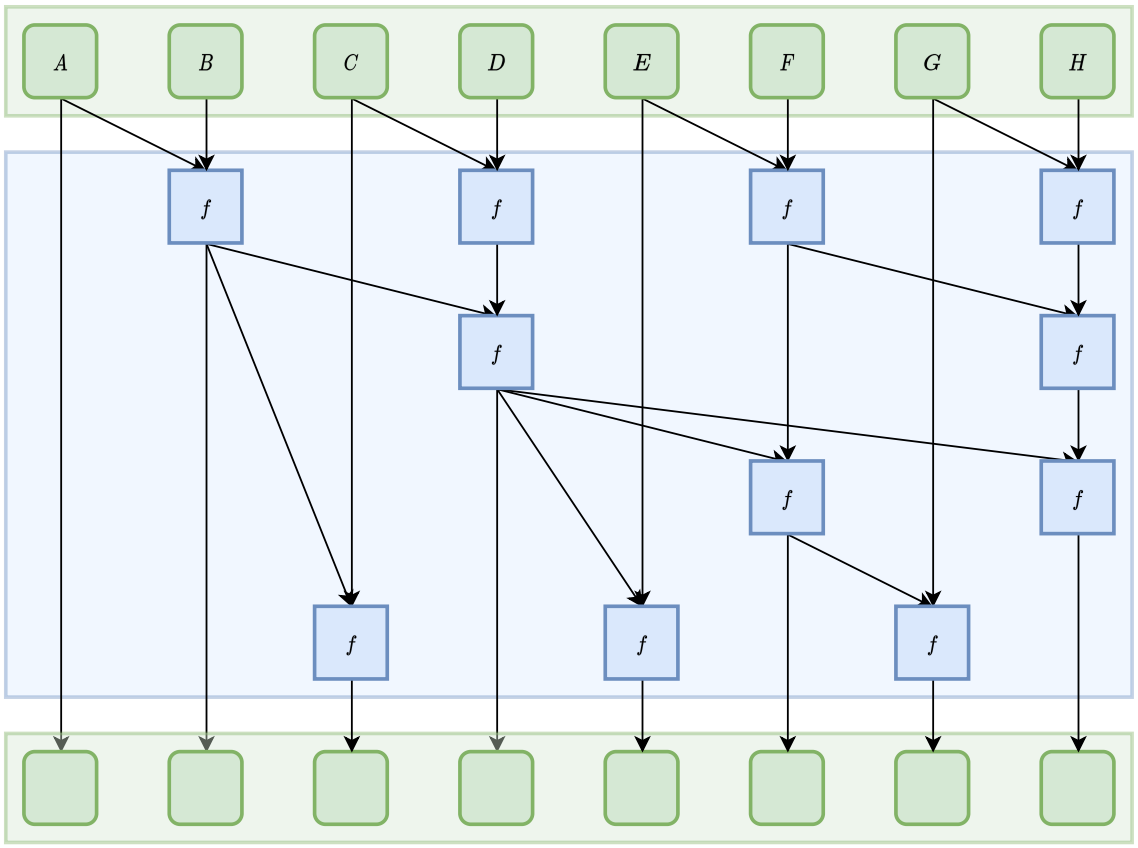


Figure 9.3: The scan pattern.

9.4.6 Recurrence

The recurrence pattern is a more complex version of the map pattern where we have a combine function instead of the loop iteration, hence each function can use the output of previous functions.

9.5 Serial data management patterns

Data management patterns deal with how data is allocated, shared, read, written, and copied. The main serial data management patterns are:

- **Random read and write**.
- **Stack allocation**.
- **Heap allocation**.

- **Objects**.

9.5.1 Random read and write

Random parts of the memory can be accessed using addresses. In programming languages we use pointers to refer to memory addresses. Pointers usually make data dependencies much more complex because we have to follow the pointer and dereference it, which makes the analysis of code hard. In particular, aliasing (uncertainty of two pointers referring to the same object) can cause problems when serial code is parallelised.

9.5.2 Stack allocation

A stack is used to allocate data in a LIFO fashion. Stacks preserve data locality but are hard to parallelise, hence it's usually better to use a different stack for each task.

9.5.3 Heap allocation

Heap allocation is used for dynamically allocating data without in a non-LIFO fashion. Heap allocation is slower and more complex.

If we have only one memory allocator, access to the memory allocated should be unsynchronised with respect to the other tasks. This makes allocation much simpler.

9.5.4 Objects

Objects are data and functions and the way they are parallelised depends on the language.

9.6 Parallel data management patterns

Parallel data management patterns focus on how data should be prepared. For instance, there exist patterns that allow to filter some data before a given function is applied to the data. The main parallel data management patterns are:

- **Pack**.
- **Pipeline**.
- **Geometric decomposition**.
- **Gather**.
- **Scatter**.
- **Superscalar sequences**. It writes a sequence of tasks, ordered only by dependencies.
- **Futures**. It's similar to fork-join, but tasks do not need to be nested hierarchically.
- **Speculative selection**. It's a general version of serial selection where the condition and both outcomes can all run in parallel.

- **Workpile**. It's a general map pattern where each instance of elemental function can generate more instances, adding to the pile of work. Work items can be added to the map while it is in progress, from inside map function instances. The work grows and is consumed by the map. The workpile pattern terminates when no more work is available.
- **Search**. It finds some data in a collection that meets some criteria.
- **Segmentation**. Operations on subdivided, non- overlapping, non-uniformly sized partitions of 1D collections.
- **Expand**. It's a combination of pack and map.
- **Category reduction**. Given a collection of elements each with a label, find all elements with same label and reduce them.

Moreover, data movement patterns allow to better place data in memory so that data locality can be exploited and data can be retrieved in block from memory in one shot. These patterns allow then to organise data to minimise data access times, costs and cycles.

9.6.1 Pack

The pack data pattern is used to eliminate some data from an input collection and reduce the space used by the input collection. In other words, we have a mask that says what input has to be kept and what has to be deleted. Parallelising this pattern isn't that easy since the position where an input should be put in output depends on the mask values of the previous inputs. This problem can be solved applying the scan pattern to compute the positions where each element should be mapped in the output collection.

The inverse of the pack pattern is the unpack pattern, which allows to place packed elements in their original position.

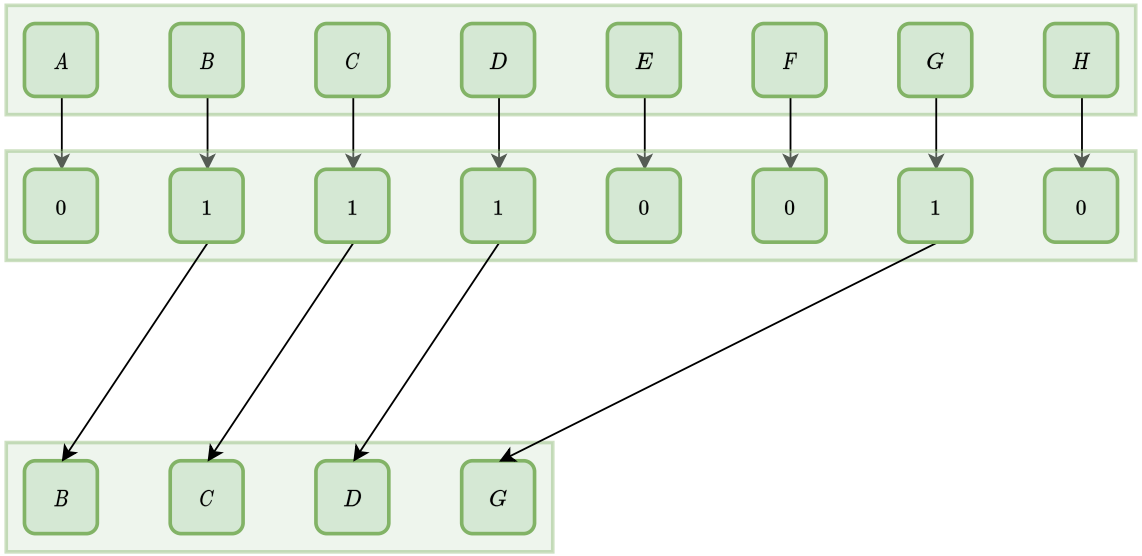


Figure 9.4: The pack pattern.

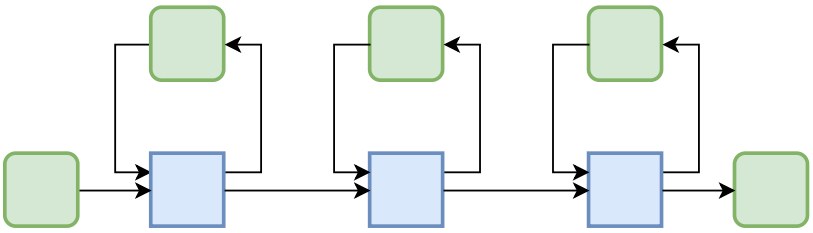


Figure 9.5: The pipeline pattern.

9.6.2 Pipeline

The pipeline pattern collects tasks in a producer-consumer manner. It allows to parallelise tasks that were supposed to run sequentially. In particular a pipeline is a sequence of blocks and each block computes a part of each task. While a task is executing in a block, another task is executing in another block and it's consuming other data.

9.6.3 Geometric decomposition

The geometric decomposition pattern allows to divide the input data collection into subsections which can be of any size and can even overlap.

9.6.4 Gather

The gather pattern starts from a collection of data and a collection of indices. The collection of indices specifies which elements should be put in the output collection. In particular, given a collection of indices $[i_1, i_2, i_3, i_4, i_5]$, the first element of the output collection is the one in position i_1 of the input, the second in position i_2 and so on.

This pattern is quite easy to parallelise since we already know where an input value should be placed.

9.6.5 Scatter

The scatter data pattern works opposite with respect to the gather pattern. In particular, given an input collection and a collection of indices, we want to put the values of the input collection in the positions specified by the collection of indices. This means that given a collection of indices $[i_1, i_2, i_3, i_4, i_5]$, the first element of the first element of the input collection is put in position i_1 of the output, the second element is put in position i_2 of the output and so on. Note that data races might occur, in fact if the collection of indices contains two equal values, then two elements of the input sequence are put in the same output position.

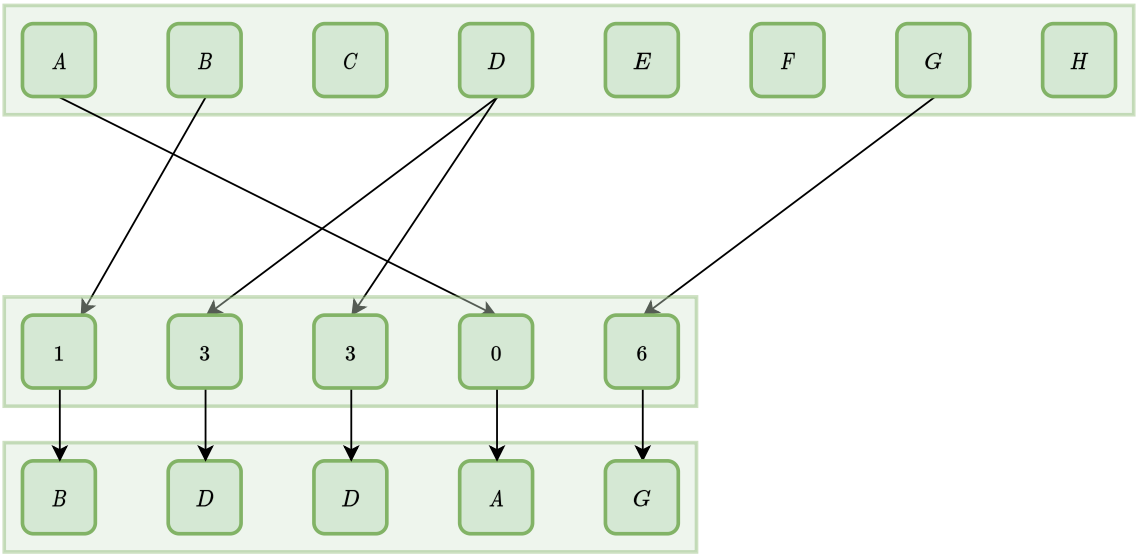


Figure 9.6: The gather pattern.

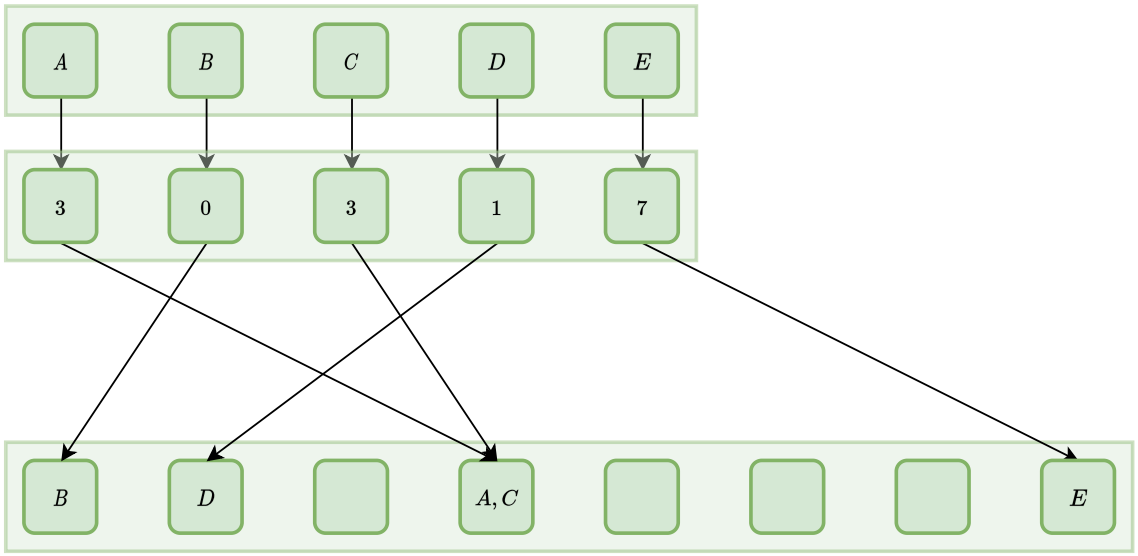


Figure 9.7: The scatter pattern.

Chapter 10

Parallel patterns

10.1 Map

The map pattern allows to apply a function to every element in the input collection. If the function depends only on one input element, then the map can be computed in parallel. In particular, since each function can be executed independently, we can run each function on a task in parallel. More formally, the functions should be pure, hence they should not produce side effects and should not modify shared states. If the function is not pure, we can have

- data races,
- non-determinism,
- undefined behaviours and
- segmentation faults.

Many languages allow to apply the map pattern using an array notation. For instance, if we have an array `A` and we want to multiply each element by 5 we can write

```
A[:] = A[:] * 5;
```

10.1.1 N-ary maps

Maps can be applied not only to a single input collection but also to N collections. For instance we can take two collections with the same number of elements and compute the sum of values in the same positions.

10.1.2 Sequences of maps and code fusion

Maps can also be applied in sequence. For instance, we can apply a map to a collection, obtain another collection to which we can apply another map and so on. Since the input collection might be very large, we might want to apply the maps in one shot. This allows to save memory since we don't have to cache partial results. This process is called **code fusion** since we fuse maps. Note that code fusion is possible if we can compute the operations all together (i.e., we have enough registers).

If it's impractical to fuse together the map operations we can instead break the work into blocks, giving each CPU one block at a time. Basically, instead of computing a sequence of maps on the whole input collection, we apply it to a sub-collection and then we merge the results.

10.1.3 Scaled vector addition

One example of application of the map pattern is scaled vector addition. Given two vectors `x` and `y` and an integer `a` we want to compute `y = a*x + y` where `a*x` is the multiplication of every element of `x` for `a`. The result of this operation is a vector of the same size of `x` and `y` and the output element in position i depends only on the inputs in position i, hence this operation can be parallelised. In particular, we can compute

```
y[i] = a * x[i] + y[i]
```

If we haven't enough cores to execute one operation for core, we can split the input in different blocks and use vectorised operations to compute the output for a certain sub-vector.

The C serial implementation of the scalar vector addition is shown in Listing 10.1. The same function implemented in OpenCL is shown in Listing 10.2 while the OpenMP implementation is shown in Listing 10.3. The difference between the parallel implementations are that the latter implementation is the same as the serial one but we have simply added a pragma that automatically handles thread creation. Note that we can add the pragma annotation because we are sure that the loop iterations are independent hence they can be parallelised.

```
1 void saxpy_serial(size_t n, float a, const float x[], const float y) {
2     for (size_t i=0; i<n; i++){
3         y[i] = a * x[i] + y[i];
4     }
5 }
```

Listing 10.1: Serial implementation of the scaled vector addition.

```
1 __kernel void saxpy_opencl(__constant float a, __global float* x, __global float* y
      ){
2     int i = get_global_id(0);
3     y[i] = a * x[i] + y[i];
4 }
```

Listing 10.2: Parallel implementation of the scaled vector addition in OpenCL.

```
1 void saxpy_openmp(int n, float a, const float x[], const float y){
2     #pragma omp parallel for
3         for (int i=0; i<n; i++){
4             y[i] = a * x[i] + y[i];
5         }
6 }
```

Listing 10.3: Parallel implementation of the scaled vector addition in OpenMP.

10.2 Reduce

The reduce pattern is a collective pattern since it's applied to a collection of data. Reduction allows to combine the values in a collection to obtain a single value. If we want to apply the reduction pattern we have to ensure that the combining operation is associative. An example is therefore

the summation of the values in the input collection. Note however that the addition of floating point data can't be considered associative because the mantissa and exponent representation might return different results depending on the order in which operations are applied. When parallelising the reduce pattern, the parallelisation power decreases with time. To understand why this is true, we have to analyse how this pattern can be implemented. Say we have an input collection `[1, 3, 2, 4, 9, 5, 8, 7]` and we want to apply the sum reduction. We can start by applying the sum in pairs, hence we compute `1+3`, `2+4`, `9+5`, `8+7`. These operations can be done in parallel hence we can exploit 4 executors. Now we have to combine the results applying once again the sum, in particular we get `4+6` and `14+15` and we can only exploit two executors.

From this example we can see that the computation can be represented as a binary tree, hence we can immediately say that the computation of the sum of n elements requires a logarithmic time, namely $\log n$, which is a good speedup with respect to the sequential case which requires linear time.

10.2.1 Real word implementations

Vectorisation

Reduction can be implemented on a real machine in different ways. One way is the one we have shown above, but some processors which can execute vector operations can also apply vectorisation and compute the sum of two vectors in one shot. Considering the example we did before, we can compute `[1, 3]+[2, 4]` in parallel thanks to vectorisation. In this case the speedup isn't logarithmic but linear. In particular, if the processor can execute vector operations on vectors of size n, then the reduction will be n times faster.

Tilting

Another way of implementing reduction is by using tilting. In this case we decompose the input collection in chunks and apply a sequential operation to them. The sequential operation can however be run in parallel on different executors hence we can have a speedup in terms of execution. Say we have for instance `[4, 8, 6, 5, 3, 2, 9, 7, 1]` and we build a task that can sequentially reduce a vector of three elements. We can split the input collection in three chunks `[4, 8, 6]`, `[5, 3, 2]` and `[9, 7, 1]` and reduce each chunk using the serial combination function which is run in parallel for the three different chunks. What we get as output is a collection of three elements, namely `[18, 10, 17]`. We can now apply the serial combinator once again to obtain the final result, which is `45`. In this case the speedup is logarithmic with base the size of the combinator's input vector.

10.2.2 Fusing

Reduction can be applied with other parallel patterns like, for instance, with a map. To avoid storing partial results we can fuse the two patterns and apply them all together in one shot.

10.2.3 Dot product

An example of reduction and map pattern is the dot product between two vectors. Given two vectors $\mathbf{v}$ and $\mathbf{w}$ of the same size n, we can compute the dot product as

$$\mathbf{v} \cdot \mathbf{w} = \sum_{i=1}^{n} v_i w_i$$

In this example we can exploit:

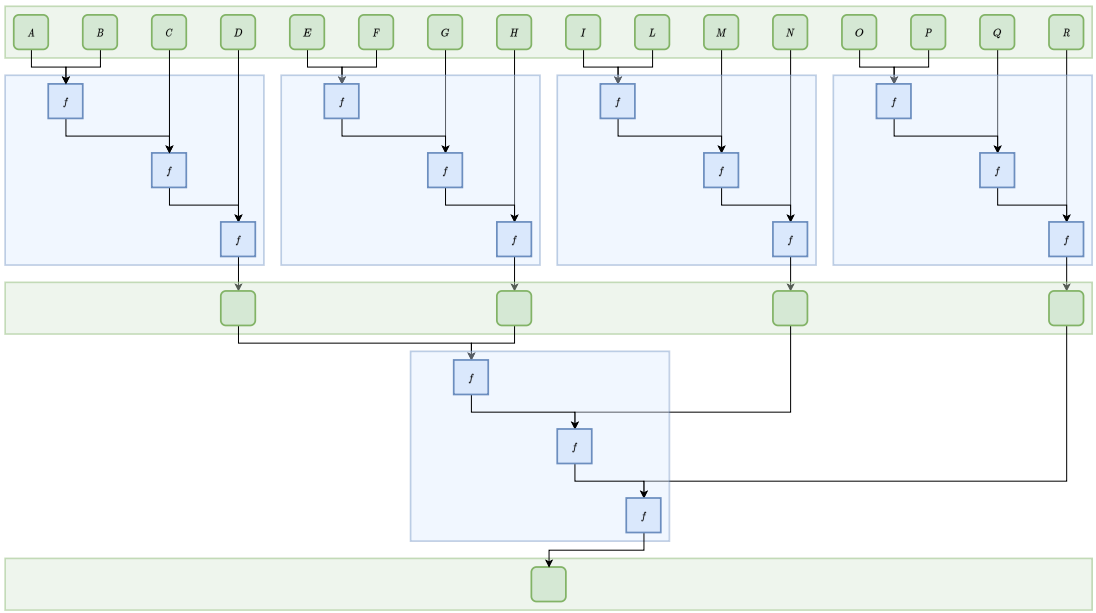


Figure 10.1: The reduce pattern implemented using tilting.

- The **map pattern** to take two vectors and multiply their values in the same positions, hence computing $v_i w_i$ for each i in parallel.
- The **reduce pattern** to apply the sum operation to the result of the previous step (which is a collection of n elements).

10.3 Scan

The scan pattern works similarly to the reduce pattern but we don't only have the combination of all values but also the combinations of all values until a certain point. This means that the scan operation returns an output collection of the same size of the input. For instance if we have a vector `[1, 2, 3, 4]` and we want to compute the scan using the sum we obtain

```
[1, 1+2, 1+2+3, 1+2+3+4] = [1, 3, 6, 10]
```

In other words, the scan computes all partial reductions of the input collection. The scan pattern can also be divided in:

- **Exclusive scan** if the current element is not considered when applying reduction. If we consider the example we did before, we would have obtained `[0, 1, 1+2, 1+2+3]` by applying exclusive scan.
- **Inclusive scan** if the current element is considered when applying reduction. The result obtained in before's example is computed by applying inclusive scan.

10.3.1 Implementations

Parallel combination

The simplest way to implement the scan pattern is to divide the input collection in couples and combine them in parallel. The results have to be combined in parallel again until we obtain the whole output vector. This implementation doesn't waste as many resources as the reduce's since we have to compute more output elements, hence we can exploit all the processing units.

Tiling

As for the reduce pattern we can build a serial function that computes serially reduction for an input collection of a fixed size, divide the input collection in chunks of that size and then apply the function until we get the output vector.

10.3.2 Fusing

As for reduction, the scan pattern can be combined with a map, hence it's useful to merge these operations to avoid wasting memory for partial results. As always this is possible only if the processor has enough register to store partial results itself.

10.3.3 Merge sort

The reduction pattern is used, together with the map pattern, in the merge sort. In particular, given an input collection `[14,3,4,8,7,52,1]` we can:

- Use the **map pattern** to map each element into a vector containing only that element. Namely, we get `[14],[3],[4],[8],[7],[52],[1]`.
- Use the **scan pattern** to merge the vector. In particular we compute

  ```
  R = [14] <> ([3] <> ([4] <> ([8] <> ([7] <> ([52] <> [1])))))
    = [14] <> ([3] <> ([4] <> ([8] <> ([7] <> [1,52]))))
    = [14] <> ([3] <> ([4] <> ([8] <> [1,7,52])))
    = [14] <> ([3] <> ([4] <> [1,7,8,52]))
    = [14] <> ([3] <> [1,4,7,8,52])
    = [14] <> [1,3,4,7,8,52]
    = [1,3,4,7,8,14,52]
  ```

  where `<>` represents the merge operation.

This algorithm has done n merges, each having a liner cost of n, hence its complexity is $\mathcal{O}(n^2)$. If we want to have a better result we have to apply reduction as follows

```
R = (([14] <> [3]) <> ([4] <> [8])) <> (([7] <> [52]) <> [1])
  = ([3,14] <> [4,8]) <> ([7,52] <> [1])
  = [3,4,8,14] <> [1,7,52]
  = [1,3,4,7,8,14,52]
```

which has a complexity of $\mathcal{O}(n \log n)$ as expected. This means that the shape of the merge function is vital to obtain a good complexity.

10.4 Gather

The gather pattern

- Takes a collection of data and a collection of indices as input. The collection of indices has the same size of the output.
- Writes in position i of the output collection the input element in the position specified by the collection of indices in position i.

Note that when generating the output collection, other patterns can be applied, like for instance the map. A serial implementation of this pattern is shown in Listing 10.4.

```
template<typename Data, typename Idx>
void serial_gather(size_t n, size_t m, Data in[], Data out[], Idx idx[]) {
    for (size_t i=0; i<m; i++){
        size_t j = idx[i];
        assert(0 <= j && j <= n);
        out[i] = in[j];
    }
}
```

Listing 10.4: The C++ serial implementation of the gather pattern.

From the serial implementation we can notice that the output value in position `i` is written only once and depends only on the input (which is never modified) hence the loop can be fully parallelised.

10.4.1 Shift

The shift pattern is a simplified specialisation of the gather pattern. In particular, given:

- An input collection of size n.
- An element outside the input collection.
- An index collection of the shape `[1, 2, 3, 4, 5, ..., n]`

The element outside the input collection is required since we want to access the element in position `n` of the input, which doesn't exist, hence we use the element outside the input. The shift we have just described shifts values to the left, however we might want to shift values also to the right. In this case we have to use `[-1, 0, 1, 2, ..., n-1]` as index collection and use the external element when using index `-1`. If we don't want to use an external element, we can apply:

- **Rotation** and use the element in the input sequence that would otherwise go lost. In this case, for a left rotation, the index collection is `[1, 2, 3, 4, 5, ..., n-1, 0]`.
- **Duplication** and use an element twice. In this case, for a left rotation, the index collection is `[1, 2, 3, 4, 5, ..., n-1, n-1]`

10.4.2 Zip

The zip pattern is yet another specialisation of the gather pattern. This pattern takes two input collections and interleaves its values. Being a gather pattern, the zip can be easily parallelised in fact each element in the output collection depends only on one input and it's easy to understand from which input, in fact, given an output position i,

- If i is even, we have to take the value from position $\frac{i}{2}$ of the first input collection.
- If i is odd, we have to take the value from position $\frac{i-1}{2}$ of the second input collection.

Note that the zip pattern can be generalised to an arbitrary number of input collections.

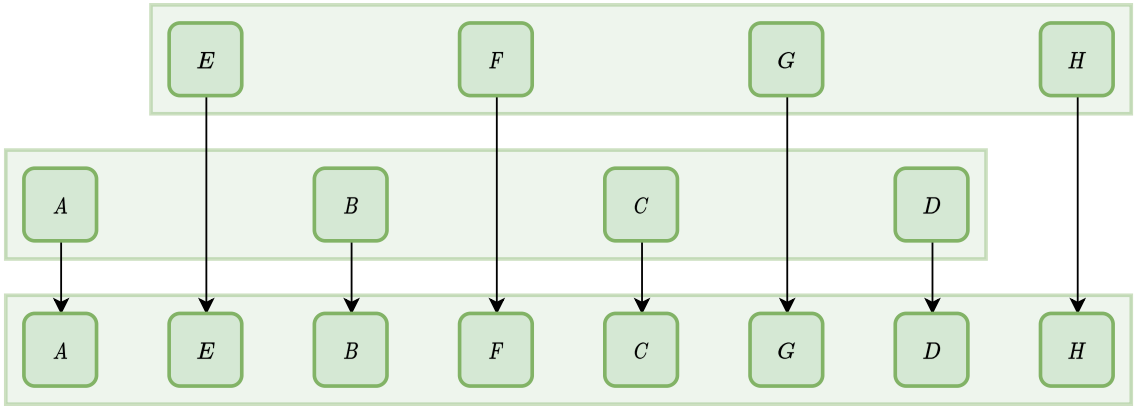


Figure 10.2: The zip pattern.

10.4.3 Unzip

The unzip pattern does the opposite of the zip pattern. In particular, given input sequence, it puts the elements in two output vectors. More precisely,

- Elements in even positions of the input are put in the first output collection, in position $\frac{i}{2}$.
- Elements in odd positions of the input are put in the second output collection, in position $\frac{i-1}{2}$.

As for the zip pattern, unzipping can be completely parallelised because each output depends only on one input and the index from where to take the value can be computed without problems.

10.5 Scatter

The scatter pattern:

- Takes an input collection and a collection of indices, of the same size.
- Writes the element in position i of the input vector at the index specified by the indices collection in position i.

A serial implementation of this pattern is shown in Listing 10.5.

```
1 template<typename Data, typename Idx>
2 void serial_gather(size_t n, size_t m, Data in[], Data out[], Idx idx[]) {
3     for (size_t i=0; i<m; i++){
4         size_t j = idx[i];
5         assert(0 <= j && j <= n);
6         out[j] = in[i];
7     }
```

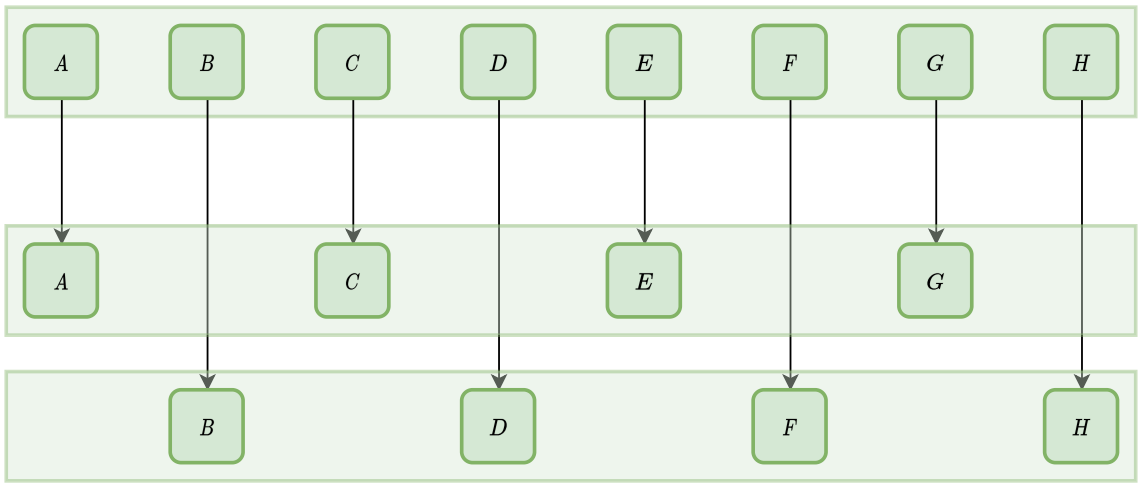


Figure 10.3: The unzip pattern.

```
8 }
```

Listing 10.5: The C++ serial implementation of the scatter pattern.

10.5.1 Race conditions

Differently from the gather pattern, parallelisation is more complex in this case. In fact, if the collection of indices contains duplicates, we will write two input values in the same position of the output, hence we generate a data race. Race conditions can be solved in different ways.

Atomic scatter

One way of solving race conditions is by using an atomic scatter, which means using atomic operations to write the output collection. This solves race conditions, however the result is still non deterministic since the output collection depends on the order in which operations are applied.

Permutation scatter

If the scatter performs only a permutation of the input collection, no data collision is generated. An example is matrix transposition. The permutation scatter imposes that the index collection has no repetitions and if that's not the case the scatter is transformed into a gather operation.

Merge scatter

If possible, we can combine the values that map to the same output position so that the result is still deterministic even in case of conflicts. Note that the function that combines the values has to be associative and commutative, otherwise the result wouldn't be deterministic.

Priority scatter

A priority scatter assigns a priority to each value in the input collection. The priority is then used to decide which value should be written first. For instance we can decide to write a value only if all values with lower priority have been written.

10.5.2 Converting a scatter to a gather

A scatter is more expensive than a gather because of its parallelisation issues. Moreover, since we might write in positions far away of the output, we can't rely too much on caching and data locality. We can avoid problems if addresses are know in advance and some optimisations to be applied. Moreover, we can convert addresses for a scatter into those for a gather. This is useful if the same pattern of scatter address will be used repeatedly so the cost is amortised.

10.6 Pack

The pack pattern is used to eliminate some elements from the input collection. Put it in another way, the pack pattern creates a bit mask as long as the input collection that defines, for each element of the input collection if it should be copied in the output collection. Note that the values in the output collection are compacted, hence we don't leave blank spaces where a value is filtered out.

Making the pack pattern parallel is not trivial since the position in which an input element has to be put depends on the number of zeros in previous positions of the input mask. We can use the scan pattern to solve this problem, in fact the exclusive scan pattern can compute, for each element in the input collection its position by summing all the previous values in the mask. The positions corresponding to a zero in the mask will be repeated but this isn't a problem since we can use the mask to understand if a value should be written in a certain position or not.

10.6.1 Unpack

The unpack pattern is the inverse of the pack pattern. In particular, given an input mask and an input collection we want to pass all the values in the mask and, when we find a 1 we copy the leftmost value of the input collection in the position where we found a 1. The serial implementation of the unpack pattern is shown in Listing 10.6.

```
template<typename Data, typename Idx>
void serial_unpack(size_t n, size_t m, Data in[], Data out[], Idx idx[]) {
    int j=0;

    for (int i=0; i<n; i++){
        if (idx[i] == 1)
            out[i] = in[j++];
    }

}
```

Listing 10.6: Serial C++ implementation of the unpack pattern

As for the pack pattern, we can exploit the scan pattern to compute the positions in which each element should be inserted.

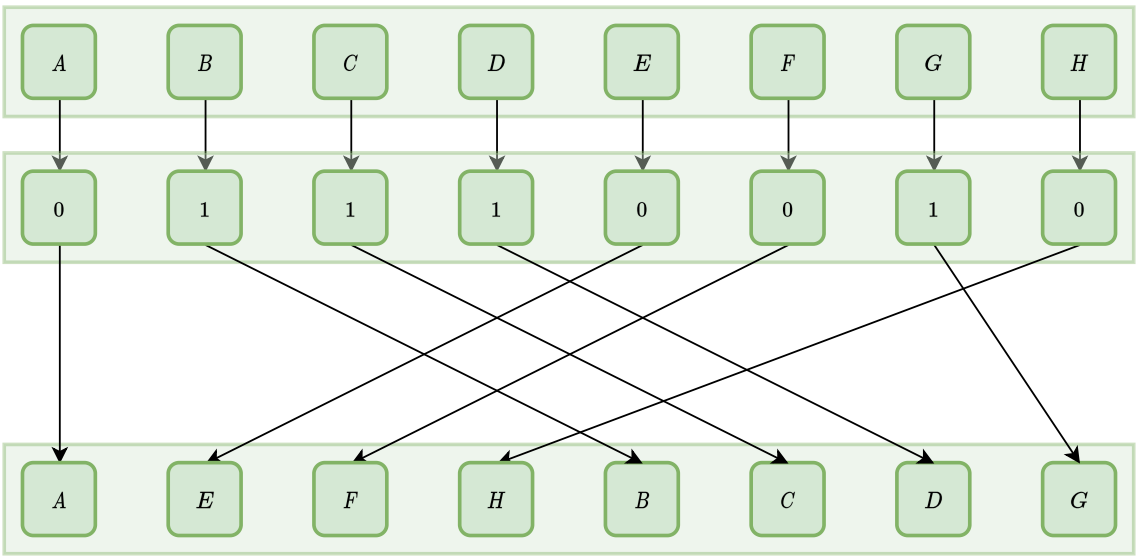


Figure 10.4: The split data pattern.

10.6.2 Split

The split data pattern is a generalisation of the pack pattern. In particular, we don't lose any data, namely every point in the input collection is copied, in a different position, in the output. More precisely, all the values with a mask value of 0 are put at the beginning of the input collection whereas the values with a mask value of 1 are put at the end.

This pattern can be easily parallelised in fact we can compute the position of each element and the total number of zeros and ones thanks to the scan pattern (hence we can compute it in parallel).

10.6.3 Unsplit

The unsplit pattern is the opposite of the split. In particular, given an input collection obtained through a split and the mask used to obtain such collection, we put the elements back in their original position, using the 1s and 0s in the mask. In practice, let us divide the input collection is two sub-collections C_0 and C_1. Given an index collection $I = \{i_0, i_1, i_2, \ldots, i_{n-1}\}$, for each element $j \in \{0, \ldots, n-1\}$,

- If $i_j = 0$, we put the first value not already used in C_0 in position j of the output.
- If $i_j = 1$, we put the first value not already used in C_1 in position j of the output.

10.6.4 Bin

The bin pattern is a generalisation of the split pattern to an arbitrary number of partitions, called bins. This means that the mask is not binary and its values can go from 0 to $N-1$, with N number of partitions. The bin pattern is shown in Figure 10.6.

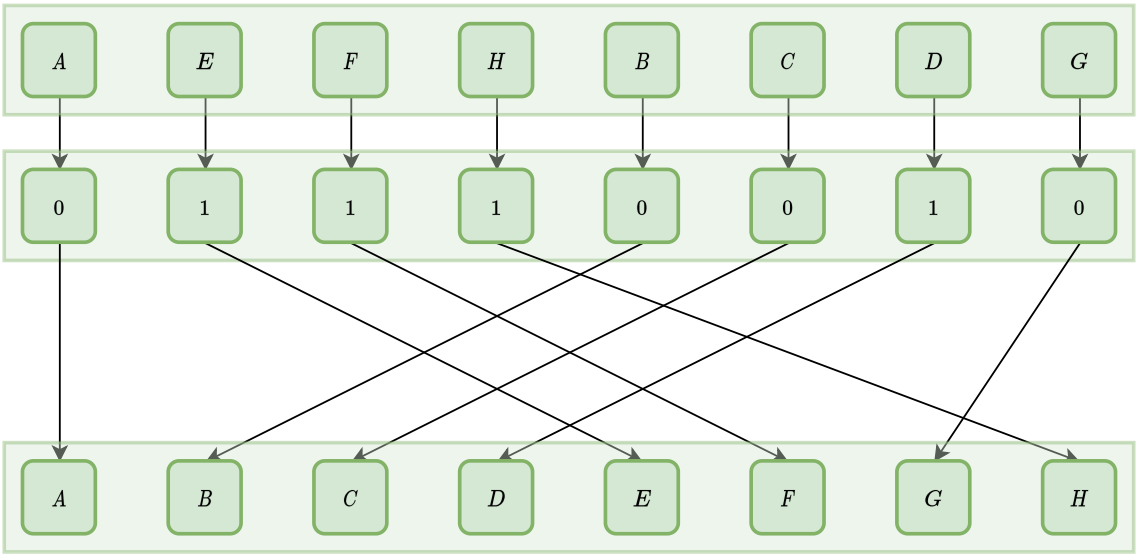


Figure 10.5: The unsplit data pattern.

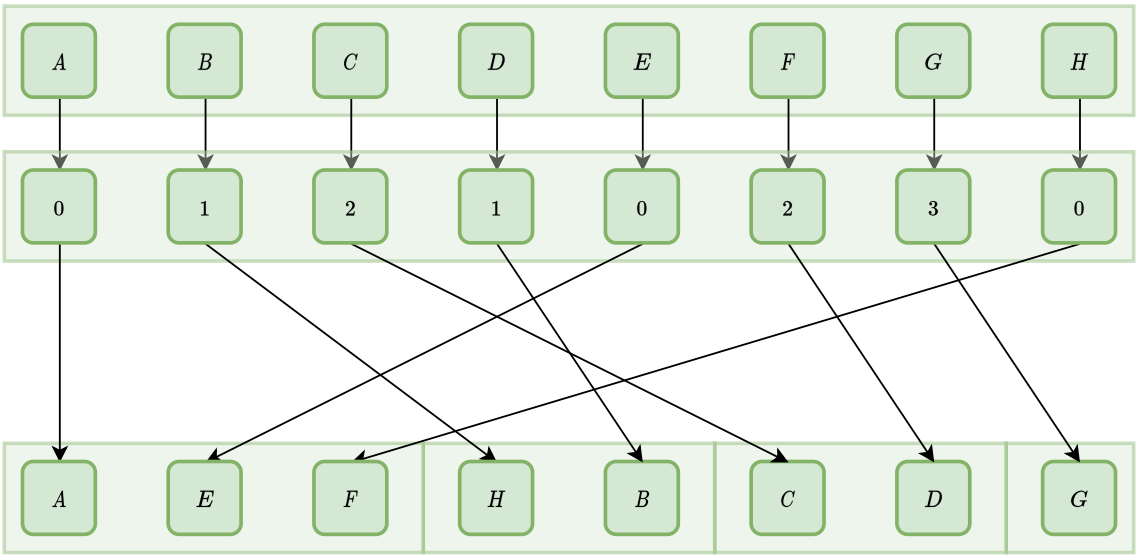


Figure 10.6: The bin pattern.

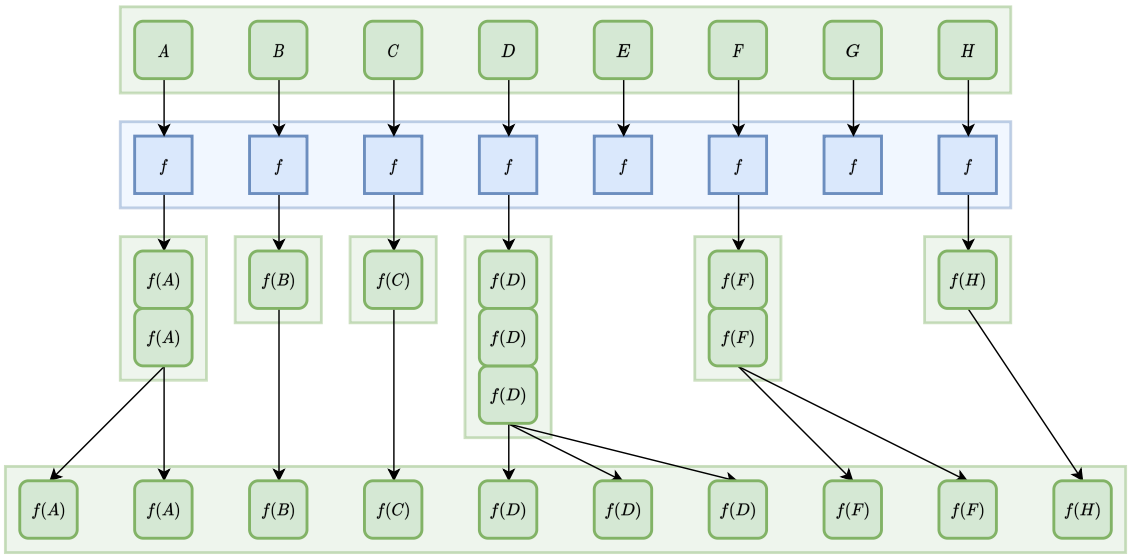


Figure 10.7: The expand data pattern.

10.6.5 Fusion

As always, we can apply the split pattern with the map. If possible, we can speed up the execution and reduce memory consumption by fusing the split and the map. In particular:

- The map checks pairs for collision.
- The pack stores only actual collisions.

Note that fusion is advantageous if most of the elements of a map are discarded.

10.6.6 Expand

The expand pattern is yet another generalisation of the pack pattern. More precisely, each element in the input collection can generate a collection of outputs, possibly empty. The collections are then merged to generate the final output.

Applying parallelisation is possible if we can compute the number of values generated by each input element since we need to compute the prefix count (using a scan) for each output element.

10.7 Data partition

Being able to partition data is fundamental for parallelising some algorithms. We can in fact split the computational domain into sections, work on each section individually and independently and then combine the results obtained. Partitioning can be done in many different ways, some of which are:

- **Divide-and-conquer**.
- **Fork-join**.

- **Geometric decomposition**.
- **Partitions**.
- **Segments**.

10.7.1 Partitioning

Partitioning allows to divide the input collection into regions which:

- do not overlap and
- have the same size.

This helps in parallelisation since we have less write conflicts and race conditions.

10.7.2 Segmentation

Segmentation allows to divide the input collection into regions which:

- do not overlap and
- can have different sizes.

10.8 Arrays of structures and structure of arrays

Let us now consider an example in which we show different ways of organising data and their advantages and disadvantages. An array of structures is a collection of structured data, hence each item of the collection is a structure. A structure of array is a structure in which each field is an array. Each array contains the values, for each object, of the field. Say for instance that we want to represent a collection of three dimensional points. We can:

- Create a structure `point` with integer fields `x`, `y` and `z` and then create an array of `points`. Each item of the array is a point.
- Create a structure `points` with vector fields `xs`, `ys` and `zs`. Each vector of the structure contains the values of the xs, ys and zs of the collection of points.

Using an array of structures is advantageous if we use the fields of an item in a sequential manner, in fact we can exploit data locality since the fields are stored in a structure. On the other hand, structures of arrays are better suited when we have to vectorise operations.

10.8.1 Memory layout

Arrays of structures and structures of arrays can be stored in memory in different ways. Say for instance we want to represent a collection of 7 3D points. Let us also assume that each coordinate of a point is one byte long and that memory is split blocks of four bytes. Regarding arrays of structures we can:

- **Pad at the end**. This means storing each structure one after the other and then leave some free cells at the end (i.e., after the 7 structures). This layout is the most efficient since we have to occupy 5 complete data blocks and 1 incomplete data block (since 7 points occupy $3 \cdot 7 = 21$ bytes and 6 blocks are $6 \cdot 4 = 24$ bytes).

- **Pad after each structure**. This means aligning each structure at the beginning of a memory block by adding padding block after each structure. This layout is the less efficient since we have to occupy 7 data blocks, however it allows to handle data access more easily and faster.

The same idea applies to structures of arrays, in fact we can:

- **Pad at the end**. This means storing the arrays in a structure's fields one after the other, without padding.
- **Pad after each field**. This means aligning the arrays in a structure's fields to the start of a memory block.

10.9 Stencil

A stencil pattern works similarly to the map pattern but an output value depends on an input value and on some other values close to it (i.e., in the neighbourhood). For instance, convolution is an example of the stencil pattern since an output pixel depends on a section of the input image. A general serial implementation of the stencil pattern is shown in Listing 10.7.

```
template<int NumOff, typename In, typename Out, typename F>
void serial_stencil(int n, const In in[], Out out[], In b, F func, const int
    offsets[]) {
    // array to store neighbours
    In neighbourhood[NumOff];

    // loop over all output locations
    for (int i=0; i<n; i++) {
        for (int j=0; j<NumOff; j++) {
            int k = i + offsets[j];
            if (0 <= k && k <= n) {
                // read input location
                neighbourhood[j] = in[k];
            } else {
                // handle boundary case
                neighbourhood[j] = b;
            }
        }
        out[i] = func(neighbourhood);
    }
}
```

Listing 10.7: The C++ of the stencil pattern

If the input and output collection are disjoint the for loop can be completely parallelised and we can use as many threads as the number of input points.

The key characteristic of the stencil pattern is the neighbourhood. There exist different types of neighbourhoods and, depending on their shape, we can have different way to parallelise and optimise the pattern.

10.9.1 Overlapping regions

We've seen that parallelising the stencil pattern is easy in case the input and output collections are disjoint. Let us now consider the case in which the two collections overlap. In particular, let us consider the case in which the input and output collection are the same. The serial implementation can be modified to obtain the function in Listing 10.8.

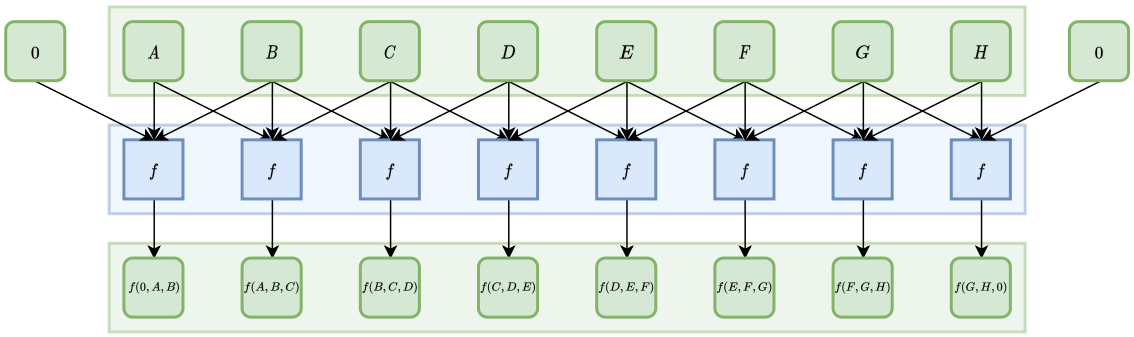


Figure 10.8: The stencil pattern

```
template<int NumOff, typename In, typename F>
void serial_stencil(int n, const In in[], In b, F func, const int offsets[]) {
    // array to store neighbours
    In neighbourhood[NumOff];

    // loop over all output locations
    for (int i=0; i<n; i++) {
        for (int j=0; j<NumOff; j++) {
            int k = i + offsets[j];
            if (0 <= k && k <= n) {
                // read input location
                neighbourhood[j] = in[k];
            } else {
                // handle boundary case
                neighbourhood[j] = b;
            }
        }
        in[i] = func(neighbourhood);
    }
}
```

Listing 10.8: The C++ of the stencil pattern

Parallelising a stencil pattern with overlapping input and output collections is not easy since writes and reads might overlap, however it's still possible to achieve some parallelism. In particular, let us consider a one-dimensional collection of 8 items $C = \{e_1, e_2, \ldots, e_7, e_8\}$ and a system with two CPUs. In this context we can parallise the stencil pattern by dividing the collection C in two sub-collections and give each collection to a processor. If we want to compute the stencil with a neighbourhood of three items (the input, one element before and one after), we have to copy the elements on the boundary. In practice, we can split C in $C_1 = \{e_1, e_2, e_3, e_4\}$ and $C_2 = \{e_5, e_6, e_7, e_8\}$. Since the second CPU needs e_4 and the first need e_5, we have to include these values in all collections, hence we obtain $C_1 = \{e_1, e_2, e_3, e_4, e_5\}$ and $C_2 = \{e_4, e_5, e_6, e_7, e_8\}$. Now, each CPU has to compute the output that uses e_4 and e_5 first and then it can freely compute the other outputs.

10.9.2 Iterative codes

Iterative codes are functions in which we have a step and each step is computed using some formulas. Then the results are updated using this formula in the input collection itself. Namely, we start from a collection of data, we apply some formula on the collection of data and then we update the

input collection with the values computed. This process is repeated until it output doesn't change anymore.

Iterative codes can be implemented using a stencil pattern which is applied until the output (or input) doesn't change.

10.9.3 Jacobi iterations

Let's assume we have a two-dimensional matrix and we want to update a cell of the matrix using the mean of the cells above, below, on the right and on the left. We can consider one pixel at a time and compute the average until convergence (i.e., nothing changes anymore).

10.9.4 Successive Over Relaxations

Successive over relaxations allow to speed up the computation not only on a single iteration but along multiple iterations. While the Jacobi iteration scheme is very simple and parallelisable, its slow convergent rate renders it impractical for any real world application. One way to speed up the convergent rate would be to over predict the new solution by linear extrapolation.

Red-black Successive Over Relaxations

Red-Black SOR is a way to enable parallel updates in place. The input collection is divided into red and black cells and, considering 4-cell stencil, the computation is divided in two steps:

1. In the first step, the values of the red cells are computed using the neighbouring black cells.
2. In the second step, the values of the black cells are computed using the neighbouring red cells.

This technique allows to divide cells and to update cells only of one type, hence one type of cell is only read, the other only written. This method allows to achieve much more parallelism and to speed up the computation since the values of cells of one colour can be computed in parallel.

10.9.5 Implementation with shifts

The stencil pattern can be implemented using shifts. Let us consider a one dimensional input collection in which the output in position i is computed using the inputs in position $i-1$, i and $i+1$. We can duplicate the input collection three times to create three identical input collections and:

- Apply a right shift to the first input collection.
- Apply the identity function to the second input collection.
- Apply the left shift to the third input collection.

Finally we can combine the results in parallel to obtain the output of the stencil operation.

This implementation is possible only when the neighbourhood is made of contiguous memory cells.

10.9.6 Cache optimisations

Let us assume a two-dimensional arrays stored row-wise in memory, namely rows are contiguous in memory. This means that when an algorithm accesses data on a row it can exploit data locality. On the other hand, if the algorithm works by columns, it will generate cache misses. In particular, we can:

- Assign **rows to cores**. This maximises data locality but creates redundant reads if we have vertical accesses since we access adjacent rows.
- Assign **columns to cores**. This might need to read data from other cores (this phenomenon is called false sharing). We can use strip mining to solve the problem related to assigning columns to cores by assigning a strip to each core. A strip has a size which is a multiple of a cache line in width and has the same height of the 2D array. In this way we don't have any shared item between two cache lines and we can easily parallelise data if we explore data from top to bottom in each core.

10.9.7 Data communication

When applying the stencil pattern, data has to be communicated from one task to the other after each iteration since every value changes. Usually, it's better to replicate neighbourhood items that are shared among different tasks in a task's local memory. When the computation of an iteration is over a task can ask the other task to communicate the new values of the shared items. If the number of threads (or tasks) increases, then the number of duplicated cells increases rapidly, hence we need a lot of memory.

Another way of approaching communication is computing the output on a set of data and then exchanging the shared neighbourhood using another set of data. Then the set of items are swapped.

In general, if we call **halo** the shared neighbourhood, also called the ghost cells, then we can have:

- A large (or deep) halo. In this case we minimise the communication cost but we require more memory.
- A small halo. In this case we increase the communications but we require less memory.

Moreover, we can use buffers to compute the interior of stencil while waiting for ghost cell updates. This is called **latency hiding**.

10.9.8 Recurrence

In some cases, the data used in one iteration isn't only related to the current input but also to other iterations of an external loop. Consider for instance the following piece of code.

```
void my_recurrence(size_t v, size_t h, const float a[v][h], float b[v][h]) {
    for (int i=1; i<v; i++){
        for (int j=1; j<h; j++) {
            b[i][j] = f(b[i-1][j], b[i][j-1], a[i][j]);
        }
    }
}
```

If we want to parallelise this loop we have to understand data dependencies between instructions. In the example above, since the recurrence depends on a constant -1 we can find a plane that cuts

through grid of intermediate results such that previously computed values are on one side of plane and values to still be computed on other side of plane. This plane is called a separating hyperplane and computation proceeds perpendicular to plane through time (this is known as a sweep).

Chapter 11

APIs

11.1 Pthread

11.1.1 Thread creation

A thread is explicitly created using the function

```
int
pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine
    )(void *), void *arg);
```

to which we have to pass

- A `pthread_t*` where the function can save the thread identifier of the new thread created. Note that each implementation has a fixed number of threads that can be created, hence we are not always guaranteed that the a thread is created.
- A `pthread_attr_t *attr` which is a pointer to a set of attributes that have to be passed to the thread. These attributes are used to specify if a thread should be
  - Joinable or detachable. In the first case a thread is able to wait another and collects the results of the thread that it's waiting. In the second case it's not possible for another thread to wait the execution of a detachable thread.
  - Scheduling.
  - Stack size.
- A pointer to the function that has to be executed by the thread.
- A pointer to the arguments that have to be passed to the subroutine.

11.1.2 Termination

A thread, being a function, can end with the `return` statement but we can also use the function

```
void
pthread_exit(void *value_ptr);
```

to exit from the thread and allow other threads to join (if the thread is joinable). This function takes as parameter that the thread should return. If all threads exit the function `exit(0)` is called to terminate the process. Note that a thread can be joined also when we do a return.

11.1.3 Joining

A thread can synchronise with another thread and wait until it ends using the function

```
int
pthread_join(pthread_t thread, void **value_ptr);
```

which takes as parameter:

- The id of the thread that the caller wants to wait.
- A pointer where to store the return value of the thread waited by the caller.

11.1.4 Barriers

Joins aren't enough for synchronising threads, in fact we need another set of functions used to create barriers. Barriers are objects that allow to block threads at a certain execution point until a certain number of threads has reached that point. In practice, a barrier is an instance of the structure `pthread_barrier_t`. The function

```
int
pthread_barrier_init(pthread_barrier_t* barrier, pthread_barrierattr_t* attr,
    unsigned int count);
```

allows to initialise a barrier by specifying:

- The pointer to the barrier to initialise.
- A set of attributes to be used for initialisation.
- The number of threads that the barrier has to block before being opened.

A thread can wait a barrier using the function

```
int
pthread_barrier_wait(pthread_barrier_t* barrier);
```

which only requires to specify the barrier.

11.1.5 Mutexes

Mutexes are artefacts used for synchnronising threads. In particular, a mutex is a variable which can be locked by a thread. If the variable is already locked, then the thread has to wait until the holder unlocks it. In practice, a mutex is a variable of type `pthread_mutex_t`. A mutex can be locked using the function

```
int
pthread_mutex_lock(pthread_mutex_t *mutex);
```

which sleeps if the mutex is locked (i.e., it's a blocking function) or the function

```
int
pthread_mutex_trylock(pthread_mutex_t *mutex);
```

which tries to lock the mutex and returns either if it's locked or not (i.e., it's not blocking). A mutex can be unlocked using the function

```
int
pthread_mutex_unlock(pthread_mutex_t *mutex);
```

11.1.6 Condition variables

Condition variables are yet another artefact used for synchronising threads. Condition variables are particularly useful when some threads have to wait a result produced by other threads. A condition variable is implemented as a variable of type `pthread_cond_t`. The function

```
int
pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t *attr);
```

is used to initialise the condition variable. The function

```
int
pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
```

is used by a thread to wait on a certain condition variable. Note that a condition variable waiting doesn't only require the condition variable but also the mutex used to synchronise the access to the shared condition variable. Note that the mutex should be already locked when calling the wait (and will be automatically released when calling the wait). A thread that produces some data can use the function

```
int
pthread_cond_broadcast(pthread_cond_t *cond);
```

to notify and wake up all waiting threads, or the function

```
int
pthread_cond_signal(pthread_cond_t *cond);
```

to notify and wake up only one thread.

11.2 OpenMP

11.2.1 Introduction

OpenMP works on the same architecture of the `pthread` API, namely it's a shared memory parallel programming model that uses threads. OpenMP is mostly based on compiler directives (i.e., pragmas) but it also provides library routines and environment variables. When compiling, we have to add the `-popenmp` flag and we have to add the `omp.h` header in the source code if we want to use openMP routines. OpenMP also allows to compile some code both for serial and parallel execution, this is done by enclosing a piece of code between the macros `#ifdef _OPENMP` which allows to skip parallel code when compiling without `-popenmp`.

The advantage of using compiler directives instead of function calls is that the program is more scalable (since the compiler decides how many threads should be spawned depending on the host architecture) and the programmer doesn't have to deal directly with threads.

OpenMP handles threads by using the master-slave paradigm. This means that we have a master thread that can spawn and handle threads and many slave threads that can only execute and that run in parallel. Eventually, all slaves will be joined. After a join the master can do some serial work and then spawn new slaves to do some other parallel work.

OpenMP has many language features which can be divided into five categories:

- **Parallel control structures** which are used to fork the execution flow.
- **Work sharing directives** which are responsible to take a task and assign different pieces of the task to different threads.

- **Environment variables** which allow to specify the visibility of variables.
- **Synchronisation directives** which are used to handle critical sections.
- **Run-time functions and environment variables** which change the behaviour of the program at run-time

11.2.2 Syntax

OpenMP mainly uses pragmas to specify how a piece of code should be parallelised. The general syntax of an openMP pragma is

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

11.2.3 Parallel directive

The parallel directive, specified with the `parallel` pragma as follows

```
#pragma omp parallel [clauses] {
    /* parallel directives */
}
```

executes the directives specified in multiple threads. Note that all threads will execute the same instructions, namely each slave spawned runs a copy of the code specified between braces. The parallel directive implicitly defines a barrier at the end of the code such that all slaves can be synchronised when they finish executing.

Note that if a return statement is put inside the parallel directive, the behaviour is undefined. This means that return instructions should always be written after joining all threads, i.e., after the parallel directive. Also note that resources aren't managed by the operating system, hence we have to take care of how they are handled.

Clauses

The parallel directive allows to specify a number of clauses:

- The `if(condition)` clause allows to specify under which condition a thread should be created.
- The `num_threads(int)` clause allows to specify how many threads should be spawned. In general, to understand how many threads should be spawned, the complier checks, in this order, the `num_threads` clause, the `omp_set_num_threads()` function and the `OMP_NUM_THREADS` environment variable.

Scope

When talking about the scope of a parallel directive we have to differentiate between:

- **Static extent** which is the code directly enclosed in braces.
- **Dynamic extent** which is the code that is called by the code in braces.

This means that if we call a function inside a parallel directive, its code is part of the dynamic extent.

11.2.4 For

The for directive, specified with the `for` pragma as follows

```
#pragma omp for [clauses]
<for loop>
```

allocates different iterations of a for loop to different threads, hence it allows to run different iterations of a for loop in parallel. Note that it's possible to use a for pragma only if the iteration variable is not dependent on the loop body (i.e., we are not changing the number of iterations inside the loop). Moreover the code can't have data dependencies between different iterations.

Clauses

The for directive allows to specify some clauses:

- The `nowait` clause removes the implicit barrier at the end of the directive.
- The `schedule(type, [, chunk])` clause that specifies how different iterations of the loop are divided among different threads. In particular, we can specify the type of the scheduler (static, dynamic or runtime) and the chunk size. If we use a static scheduler, loop iterations are divided into blocks of size chunk and then statically assigned to threads. If chunk is not specified, the iterations are evenly divided among threads contiguously. If we use the dynamic scheduler, loop iterations are divided into blocks of size chunk, and dynamically scheduled among threads. The default chunk size is 1. If we use a runtime scheduler, the compiler will use the value specified in the `OMP_SCHEDULE` environment variable.

11.2.5 Sections

The sections directive, specified using a `sections` pragma as follows

```
#pragma omp sections [clauses]{
    #pragma omp section {
        /* code */
    }
    #pragma omp section {
        /* code */
    }
}
```

executes different pieces of code on different threads. In particular each piece of code defined inside a `section` pragma is executed in a thread in parallel with the code of other sections. Note that nested sections require nested parallel directives.

Clauses

The sections directive allows to specify different clauses:

- The `nowait` clause is used to remove the barrier implicitly added at the end of the sections directive.

11.2.6 Single and master

The single and master directives are mainly focused on synchronisation. The single directive, specified with the `single` pragma as follows

```
#pragma omp single [clauses] {
    /* code */
}
```

is used to execute a piece of code only in one thread (the first available). The master directive, specified with the `master` pragma as follows

```
#pragma omp master {
    /* code */
}
```

does the same as the single directive, but the code is executed by the master thread. This is for instance useful when we want to print something. Another difference between these directives is that the former has an implicit barrier at the end, whereas the latter doesn't.

11.2.7 Critical

The critical directive, specified with the `critical` pragma as follows

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

is used to handle critical sections. More precisely it allows to specify a piece of code that can be accessed only by a thread at a time.

This pragma allows to specify also the name of the critical section so that different threads can access different critical sections.

11.2.8 Barrier

The barrier directive, specified with the `barrier` pragma as follows

```
#pragma omp barrier
```

is used to explicitly add a barrier. Note that differently from the pthread implementation of the variable we don't have to specify the number of threads to wait since this number is defined at runtime depending on the architecture and the system.

11.2.9 Atomic

The atomic directive, specified with the `atomic` pragma as follows

```
#pragma omp atomic [clauses]
<single statement>
```

atomically performs an instruction. This means that a thread performing that instruction can't be interrupted by other threads.

Clauses

The clauses of the atomic directive allow to specify which part of the instruction should be executed atomically. In particular we can use:

- The `read` clause is used to say that a variable should be read atomically (e.g., when reading a field of a structure).
- The `write` clause is used to say that a variable should be written atomically.
- The `update` clause is used to say that an operation on a variable should be done atomically.
- The `capture` clause is used to say that a write and an update should be done atomically.

11.2.10 Variables scope

OpenMP is based on a shared memory model, hence threads can read the same values. This means that in the code

```
int main() {
    int x = 0;

    #pragma omp parallel {
        x = f();
    }
}
```

the variable `x` is shared hence every thread can modify it. This is openMP default behaviour however we can use some directives to modify the scope of a variable. The most used scope directives are:

- `private`
- `shared`
- `default`
- `reduction`

Private

The private clause is used to declare variables which can be accessed only by the current thread. The private clause is used as follows

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

where we specify the variables that have to be private in the list after the keyword private. In practice, the following piece of code

```
int main() {
    int x = 0;

    #pragma omp parallel private(x) {
        x = f();
    }
}
```

specifies that the variable `x` declared before the parallel directive should be duplicated by every thread created by the parallel directive. More precisely, when we specify a variable inside `list`:

1. A new object of the same type is declared once for each thread in the team.
2. All references to the original object are replaced with references to the new object. Assume that each variable is uninitialised for each thread.

This means that the original value of the variable (i.e., the value before the pragma) is not copied in the private variables. If we want to copy the initial value when the private variables are initialised, we have to use the keyword `firstprivate` instead of `private`. Moreover, we can port the values computed by threads outside the directive using the keyword `lastprivate` instead of `private`. In particular, the value computed by the last task that reached the barrier is assigned to the variable.

Shared

The shared clause is used to declare variables which are shared among all threads. The shared clause is used as follows

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

where we specify the variables that have to be shared in the list after the keyword shared. In practice, the following piece of code

```
int main() {
    int x = 0;

    #pragma omp parallel shared(x) {
        x = f();
    }
}
```

specifies that variable `x` is shared among all threads created by the parallel directive. Since a variable is shared, the programmer should handle concurrent accesses to the variable. Note that when not specified, all variables are considered to be shared, however it's important to specify that a variable is shared because we want to be sure that variables are shared.

Default

The default clause is used to be sure that no variable is left without scope specifier. The default clause is used as follows

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

where

- If we specify `default (shared)`, then all variables are set as shared.
- If we specify `default (none)` we get a compile time error if some variable isn't explicitly assigned a scope specifier.

Reduction

The reduction clause is used when we want to specify that a variable contains the result of a reduction operation. Namely, a reduction variable aggregates a value that

- depends on each loop iteration and
- doesn't depend on the iteration order.

The syntax of the reduction clause is

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

In practice, one reduction variable per thread is created and updated by the thread and at the end a reduction operation is computed among the private variables of every thread. When specifying a reduction variable we have to state the operation we use for reduction (one among `+`, `-`, `*`, `&`, `|`, `^`, `&&`, `||`, `min`, `max`) and the variable name. Note that if the reduction variable was declared as private the reduction wouldn't work. An example of usage of the reduction variable is

```
int result = 0;
#pragma omp parallel for reduction(+:result)
for(i=0; i<SIZE; i++)
    result = result + process(data[i]);
```

11.2.11 Task

The task directive, specified with the `task` pragma as follows

```
#pragma omp task [clauses] {
    /* structured block */
}
```

defines a structured block, typically a function, that is executed in parallel. In particular, the whenever a thread encounters a `task` directive, a new task is created which can be executed or deferred to another thread. Created tasks are put in a pool and are fetched and executed by free threads. Tasks execution can be tied (default) or untied (using the clause `untied`) to the thread that created the task using the dedicated clause. The task directive is very useful to handle loops whose length isn't defined. Say we have for instance a linked list and a for loop that cycles through the list until the pointer to the next node is `NULL`. In this case we can use a single thread, using the `single` directive, to fill the task pool. The tasks in the pool will later be executed in parallel by all threads. In practice, we obtain something like

```
#pragma omp parallel {
    #pragma omp single {
        for(node_t* n = list; n != NULL; n = n->next) {
            #pragma omp task
            process(n);
        }
    }
}
```

Task dependencies

The `depend(dependence-type: locator-list)` can be used to enforce an execution order between task by specifying task dependencies. More precisely, we can specify if a task read, writes, or read and writes variables from another task. In practice, we can specify if a task reads, writes or both using the dependence types `in` and `out` followed by the names of the variables (a list of names separated by commas) on which the task depends.

Wait and yield

The

```
#pragma omp taskwait
```

directive suspends a thread that encounters it until all tasks are completed. The

```
#pragma omp taskyield
```

directive, which is supposed to be used inside a task, interrupts the execution of the current task and makes it schedulable again. Using the yield directive makes more sense when used with untied tasks.

11.2.12 Run-time functions

OpenMP is mainly based on pragmas however it also offers some functions. Some of the most important are:

- The function

  ```
  int omp_get_num_threads()
  ```

  which returns the number of threads that are currently in the team executing the parallel region from which it is called.

- The function

  ```
  int omp_get_thread_num()
  ```

  which returns an identifier for the thread making this call. This number will be between 0 and `omp_get_num_threads - 1`. The master thread of the team is thread 0.

- The function

  ```
  omp_get_num_threads - 1
  ```

  which sets the number of threads that will be used in the next parallel region.

- The function

  ```
  double omp_get_wtime()
  ```

  which provides a wall clock timing routine. We can use this function to compute the execution time of a section of code. In particular, we can call it before executing the code and after and then compute the difference.

- The function

  ```
  double omp_get_wtick()
  ```

  which returns the precision of the timer used for `omp_get_wtime()`.

11.2.13 Environment variables

OpenMP also offers some environment variables which are set to control execution of all programs in the same environment. The most important are:

- `OMP_SCHEDULE`. This is used to set a default scheduling policy for the `for` directive.
- `OMP_NUM_THREADS`. This is used to set the default number of threads to be executed in parallel.
- `OMP_NESTED`.
- `OMP_STACKSIZE`.
- `OMP_WAIT_POLICY`.
- `OMP_PROC_BIND`.

11.3 MPI

MPI, acronym of Message Passing Interface, is a library-based specification for parallel and distributed computing. This means that MPI isn't only able to work on a single machine with shared memory but also on a complex distributed machine. MPI is just a specification and each architecture has to provide its own implementation. MPI is a message passing parallel programming model and, differently from pthread and openMP, it uses processes to achieve parallelism in distributed environments.

Using messages to communicate data can generate more overhead with respect to shared memory models. Consider for instance a large matrix. In shared memory models, the matrix can be stored in shared memory and each thread can access to it through pointers. On the other hand, in message passing models, we have to send the matrix to every process, which implies a large overhead. Luckily, in distributed systems (especially over a network) the main bottleneck is bandwidth and not overhead.

The Message Passing Interface API (which is included in the `mpi.h` header) has functions that handle:

- **Buffer management**, namely functions used to handle the buffers where messages are store.
- **Message passing**, namely functions used to send and receive messages.

11.3.1 System initialisation

In order to perform communication we have to initialise all processes such that they are ready to communicate. The system is initialised using the

```
int MPI_Init (int * argc_p, char *** argv_p);
```

which requires:

- A pointer to `main`'s `argc` parameter.
- A pointer to `main`'s `argv` parameter.

and returns an error code where `MPI_SUCCESS` is the code returned when the function is successfully executed. This function sets up the storage for the buffers and initialises the processes. If the program runs on a single node and it's a multi-threaded program we should use the function

```
int MPI_Init_thread (int * argc_p, char *** argv_p);
```

11.3.2 Finalisation

Once the program has done performing all communication it should call the function

```
int MPI_Finalize(void);
```

This is required because we want to clean up all data structures used to perform communication. Ideally, `MPI_Finalize` should be called only once in the program as soon as we don't need MPI anymore.

11.3.3 Point to point communication

MPI is based on communication and it uses **communicators** to send and receive messages. A communicator is a set of processes that can send messages one to the other, namely a communicator is like a channel for processes. By default, MPI uses the world communicator (stored in the environment variable `MPI_COMM_WORLD`) which allows every process to communicate, however it's always possible to define custom communicators. A communicator is an instance of type `MPI_Comm`

Every communicator ranks all the processes it contains using an index from 0 to the number of processes in the communicator (minus 1). A process can know its rank in its communicator by using the function

```
int MPI_Comm_rank (MPI_Comm comm, int *rank);
```

to which we pass:

- The communicator to which the calling process belongs.
- A pointer to an integer where the function will save the rank of the caller.

Note that a process might belong to different communicators and, if that's the case, it's not guaranteed that the same process has the same rank inside every communicator. Moreover, a process can also understand the number of processes in the communicator by using the function

```
int MPI_Comm_size (MPI_Comm comm, int *size);
```

to which we pass:

- The communicator to which the calling process belongs.
- A pointer to an integer where the function will save the size of the communicator `comm`.

Sending messages

A message can be sent using the function

```
int MPI_Send (const void *buf, int count, MPI_Datatype datatype, int dest, int tag,
     MPI_Comm comm);
```

to which we pass:

- The data `buf` to send.
- The number of bytes `count` to send.
- The type `datatype` of the data that's been sent.
- The rank `dest` of the process to which we want to send the message.

- A tag `tag` to distinguish messages on the same communicator. The tag is a non-negative value whose meaning depends on the application.
- The communicator `comm` on which the message has to be sent.

The `MPI_Send` is a blocking subroutine, which means that a sending process stops until the receiver has read the data from the receiving buffer. The data type passed to the `MPI_Send` function can be one of the following:

- `MPI_CHAR`
- `MPI_UNSIGNED_CHAR`
- `MPI_FLOAT`
- `MPI_SHORT`
- `MPI_UNSIGNED_SHORT`
- `MPI_DOUBLE`
- `MPI_INT`
- `MPI_UNSIGNED`
- `MPI_LONG_DOUBLE`
- `MPI_LONG`
- `MPI_UNSIGNED_LONG`
- `MPI_BYTE`

Receiving messages

A message has to be explicitly received using the function

```
int MPI_Recv (const void *buf, int count, MPI_Datatype datatype, int source, int
    tag, MPI_Comm comm, MPI_Status *status)
```

to which we pass:

- The buffer `buf` where to store the received data.
- The number of bytes `count` to receive.
- The type `datatype` of the data that we want to receive.
- The rank `source` of the process from which we want to receive data. If we want to receive messages from any source we can use the `MPI_ANY_SOURCE` rank.
- A tag `tag` to distinguish messages on the same communicator. A receiver can receive messages with any tag by using the `MPI_ANY_TAG` tag. In this case, the value of the tag can be obtained from the status.
- The communicator `comm` on which the message has to be waited.

- A `status` that contains some information about the data that's been received. If we don't need to use the status we can pass the `MPI_STATUS_IGNORE` status but, if we need it we can obtain the rank of the receiver by calling `status(MPI_SOURCE)` and the tag of the received message by calling `status(MPI_TAG)`.

As for the send function, the `MPI_Recv` is a blocking function, hence the receiver is blocked until a message arrives on the receive buffer. This means that the functions to send and receive messages mutually block themselves.

Note that, a message is successfully exchanged if every field of the sender matches the fields of the receiver. This means that we can't have cases in which a sender sends a message with size `size_send` and a receiver tries to receive that message by specifying a bigger size `size_recv`.

Deadlocks

Since the `MPI_Send` and `MPI_Recv` are blocking function we might face some deadlocks. Deadlocks verify because messages are sent and received by processes in different orders; consider for instance two processes with identifier `h` and `k` that run the code in Listing 11.2 and 11.1, respectively.

```
1 /* Process h */
2 MPI_Send(k);
3 MPI_Recv(k);
```

Listing 11.1: Code of process h.

```
1 /* Process k */
2 MPI_Send(h);
3 MPI_Recv(h);
```

Listing 11.2: Code of process k.

In this scenario say `h` sends the message. It will be forever blocked because at the same time `k` sends a message instead of receiving `h`'s message. Namely, both are stuck waiting the other process, which can't receive a message because it's waiting. Deadlocks can be solved in two ways:

- Rearranging sends and receives such that we can't have mutually blocking calls.
- Using the non-blocking version of `MPI_Send` and `MPI_Recv`, which are called `MPI_Isend` and `MPI_Irecv`. Note that these functions should be used together with the `MPI_Wait` that checks that all communications have completed (otherwise we can't finalise).

It's important to remember that even non blocking communications are done in sequence. This means that if a message m_1 is sent to a process P before another message m_2 using non-blocking sends, then m_1 will always be received by P before m_2. However, there is no restriction on the arrival of messages sent from different processes. That is to say that if Q and T both send messages to R, then even if Q sends its message before T sends its message, there is no guarantee that Q's message becomes available to R before T's message.

11.3.4 Input and output

On a distributed system we have to be able to specify what the standard input and outputs are, since we have multiple computers each with its own operating system. Typically, the process with rank 0 of the world communicator is in charge of writing to the standard output however, since the standard doesn't specify what the standard output should be, every process can have access to the

standard output. This results in all processes writing to the standard output in a random order (because of concurrency).

The same isn't true for the input. In fact, program arguments are passed to every process and are cleaned by the launcher using the `MPI_Init` function. If we want to read from the standard input, the common practice is to make the standard input perform reads by process with rank 0 and then broadcast that data to every process. It is also possible to allow all processes to read from standard input using the flag `{stdin` all when running the program, however it's not suggested.

11.3.5 Compilation

MPI code is usually compiled using a compiler wrapper. An example is `mpicxx` which can be used as follows:

```
mpicxx file1.cpp [file2.cpp ...] -o exe
```

The use of a specific compiler can be forced setting the `OMPI_CXX` environment variable. Moreover, MPI executables have to be run using a dedicated launcher, called `mpiexec`, with the following syntax:

```
mpiexec -np 4 exe
```

where `np` is the number of processes to be spawned. If compiling MPI on a cluster, we should also provide a file listing all the involved nodes with the flag `-machinefile /path/to_node_list`.

11.3.6 Collective communicators

Collective communicators involve all processes inside a communicator. This means that when using a collective communicator, a process sends a message to every other process in the collective communicator. Since messages are collective, no tag is sent with a message. Moreover, collective communications are blocking routines, hence it's possible to include deadlocks even for collective communications.

Broadcast

One way to send a message to every process of the collector is by using the function

```
int MPI_Bcast (void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm
    comm);
```

to which we pass:

- The buffer `buffer` containing the data to send or the buffer where to store the data received.
- The size `count` of the buffer in number of objects of type `datatype`.
- The type `datatype` of the objects in the buffer.
- The rank `root` of the process sending the message.
- The communicator `comm` used to broadcast the message.

This function is used both for receiving and sending messages, in fact if the `root` rank is the same as the rank of the caller, the message is sent, otherwise, the function is used to wait for a message.

Reduce

The function

```
int MPI_Reduce (const void *sendbuf, void *recvbuf, int count, MPI_Datatype
    datatype, MPI_Op op, int dest, MPI_Comm comm);
```

is used to perform a reduction over data coming from multiple processes. All processes perform the same call with the same parameters, which are:

- A buffer `sendbuf` that contains the local data of the process.
- A buffer `recvbuf` that is used only by the process on which we want to perform reduction.
- The size `count` of the buffers. If count is bigger than 1, `sendbuf` and `recvbuf` are treated as arrays of `datatype` elements and the reduction is performed element-wise on the array. This is because we do the reduction of the contents of different processes' `sendbuf`s and if each `sendbuf` is an array, then the result is also an array. In addition to the types enumerated when analysing the `MPI_Send` function, we need other types to compute the `MPI_MAXLOC` and `MPI_MINLOC` operations. MPI defines for this reason data-types representing couples that can be matched with the STL pair template.
- The operation `op` used to perform reduction. The available operations are shown in Table 11.1.
- The rank `dest` of the process that has to compute the reduce operation.
- The communicator `comm`.

Note that in this case, differently from openMP, we don't use a local variable for each process but reduction is computed by a single process on `recvbuf`.

Value	Operation
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical AND
MPI_BAND	Bit-wise AND
MPI_LOR	Logical OR
MPI_BOR	Bit-wise OR
MPI_LXOR	Logical XOR
MPI_BXOR	Bit-wise XOR
MPI_MAXLOC	Maximum value and location (rank of process with maximum)
MPI_MINLOC	Minimum value and location (rank of process with minimum)

Table 11.1: Allowed operations for the reduction routine.

The function

```
int MPI_Allreduce (const void *sendbuf, void *recvbuf, int count, MPI_Datatype
    datatype, MPI_Op op, MPI_Comm comm)
```

is an extension of the `MPI_Reduce` that allows to perform a reduce and then a broadcast from the process that was the destination of the reduce (i.e., the process computing the reduce) and the other processes in the same communicator.

When doing a reduction, we require double the memory needed to store the data since we need a send and a receive buffer. MPI allows to save space by specifying the keyword `MPI_IN_PLACE` for the `sendbuf` that allows to use a single buffer and do the reduction in place.

Scatter

The function

```
int MPI_Scatter (const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *
    recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm);
```

evenly divides data among the processes in the communicator. This function requires:

- The buffer `sendbuf` containing the data to scatter. This pointer can be `nullptr` if the process calling this function is not the one scattering the data.
- The number `sendcount` of objects of type `sendtype` to be stored inside the `recvbuf`.
- The type `sendtype` of the objects in the `sendbuf`.
- The buffer `recv` where to store the scattered data for each receiving process.
- The number `recvcount` of objects of type `recvtype` that each receiver is expecting in the `recvbuf`.
- The type `recvtype` of the objects in the `recvbuf`.
- The rank `root` of the process that scatters the data.
- The communicator `comm`.

The scatter allows to split data evenly but in some cases we have knowledge of the system and we might want to scatter data in a very specific way. The function

```
int MPI_Scatterv (const void *sendbuf, const int sendcounts[], const int displs[],
    MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int
     root, MPI_Comm comm);
```

works exactly like `MPI_Scatter` but it allows to distribute data not evenly, hence specifying a different size for each receiving process. This is achieved replacing the parameter `sendcount` with two parameters:

- An array `sendcounts` containing, for each process in the communicator, the number of objects that should be sent to the process.
- An array `displs` containing, for each process in the communicator, the offset from the beginning fo the `sendbuf` from where data should be taken for that process.

Note that this function allows to send overlapping data or to filter out some data.

Gather

The function

```
int MPI_Gather (const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *
    recvbuf, int recvcount, MPI_Datatype recvtype, int dest, MPI_Comm comm);
```

joins even chunks of data coming from the processes of a communicator in a single buffer. This function requires:

- The buffer `sendbuf` containing, for each process, the data to gather. This pointer can be `nullptr` if the process calling this function is the receiver.
- The number `sendcount` of objects of type `sendtype` that every sender sends.
- The type `sendtype` of the objects in the `sendbuf`.
- The buffer `recv` where to store the gathered data.
- The number `recvcount` of objects of type `recvtype` that the receiver expects to put in the `recvbuf`.
- The type `recvtype` of the objects in the `recvbuf`.
- The rank `root` of the process that gathers the data.
- The communicator `comm`.

The gather function comes in some variations which allow to modify the behaviour of the main function and add some functionalities. The most useful are:

- The `MPI_Allgather` that performs a gather and then broadcasts the result to every other process.
- The `MPI_Gatherv` that allows to specify the position where a received chunk should be placed.
- The `MPI_Allgatherv` that performs an `MPI_Gatherv` and then broadcasts the result.

11.3.7 Synchronisation

The functions we've described to send messages either to a process or in broadcast are blocking, however there exist non-blocking versions of some of them.

Barriers

MPI allows to use barriers using the function

```
int MPI_Barrier (MPI_Comm comm)
```

which waits until all processes in the communicator `comm` have reached the barrier.

11.3.8 Time

The function

```
double MPI_Wtime(void)
```

returns the current time of execution. Note that the time is processor dependent hence it doesn't make sense to use this routine in multiple processes and then broadcast the results. The times provided may be processor independent if `MPI_WTIME_IS_GLOBAL` environment variable is defined as true.

11.3.9 Custom communicator

The world communicator contains every process spawned however we might want to create communicators with only a subset of processes. This is done using the function

```
int MPI_Comm_split (MPI_Comm comm, int color, int key, MPI_Comm *newcomm);
```

which allows to obtain a subset of a communicator. This function has to be run by all processes in a communicator and requires:

- The communicator `comm` to split.
- An identifier `color` of the partition that we want to perform. If we don't want to assign a process to a new communicator we can use the `MPI_UNDIEFINED` colour which makes the function assign `MPI_COMM_NULL` to `newcomm`.
- A `key` which is used to define the rank of the processes inside the new communicator. Note that the ranks aren't modified in the original communicator but only in the new one.
- The communicator `newcomm` where to store the new communicator.

Note that this function is blocking. In practice, this function is used by each process to specify to which new communicator is belongs. Say for instance we have three processes P_1, P_2 and P_3 in the world communicator. The processes call

```
MPI_Comm_split (MPI_COMM_WORLD, 0, 0, &newcomm1);

MPI_Comm_split (MPI_COMM_WORLD, 0, 1, &newcomm1);
```

and

```
MPI_Comm_split (MPI_COMM_WORLD, 1, 2, &newcomm2);
```

After these calls two new communicators are created:

- One communicator contains processes P_1 and P_2 since they have called the split specifying the same colour. The former has rank 0 whereas the latter rank 1 since the former has specified a lower value with respect to the latter.
- One communicator contains only process P_3 since it has called the split function with colour 1, which is different from P_1's and P_2's colour. Process P_3 has rank 0 in the new communicator.

Note that the key isn't used to assign directly a rank but to order the ranks. In fact, the process with rank 0 is the one that has specified the lowest key, independently from the fact that such key is 0 or not. If two processes specify the same key, the order of the rank in the source communicator is used to decide which should have lower rank in the new communicator.

11.4 Halide

Halide is a tool used to speed up parallel image processing. If we want to improve the speed at which images are processed and reduce the power consumption we can:

- Use **faster algorithms** or approximate counting (since in images we might not care if the output has some noise).
- Use **faster hardware** like GPUs.
- Improve **parallelism** and **cache usage**.

If we focus on the third solution, we already have many solutions to parallelise operations (e.g., C++ multi-threading with vectorised operations, CUDA or openMV) however they all rely on a specific architecture. Halide uses a very different approach in fact it decouples the algorithm from the scheduling, namely we decouple what we want to compute from when and where we want to compute it. In particular, we can define once and for all the code we want to run and then try different scheduling to optimise the execution for the architecture on which we are running the code.

Halide allows to specify pipelines of functions that will be compiled just in time or statically and then invoked later. When compiling:

1. We provide **functions** and a **schedule** to the compiler.
2. The compiler does loop nesting and allocates the structures required to store partial results.
3. The compiler tries to vectorise operations.
4. The compiler generates the LLVM bitcode.
5. The compiler generates the executable for on of the supported architectures.

Halide is very powerful and allows to parallelise code without having to specify too many details, however it also has some downsides. Some being:

- All computations are done over regular grids. This is not a problem in image analysis since images are three-dimensional matrices.
- We can use only feed-forward pipelines.
- Recursions must have bounded depth.
- Schedules must be specified manually (with the exception of the auto-scheduler).

11.4.1 Syntax

Let us consider a two-dimensional matrix. If we want to scan the matrix by rows we can write

```
Serial x,
serial y
```

whereas if we want to scan column-wise, we can write

```
Serial y,
serial x
```

We can also consider parts of a row or column as a single block to visit. For instance, if we write

```
Serial y,
Vectorize x by 4
```

we visit the matrix by column and each column is made of 4 cells. Moreover, we can parallelise the scan and visit some cells at the same time. For instance, by writing

```
Parallel y,
Vectorize x by 4
```

we visit in parallel every block of the first column and then in parallel every block of the second column where each block is a row of 4 cells (because of `Vectorize x by 4`). Halide defines the keywords:

- `Func` to define pure functions over the integer domain. Functions can be executed using the `realise` keyword. In particular, given a function `f`, we execute it by writing

  ```
  f.realise(image_width, image_height)
  ```

- `Var` to define abstract variables for the domain of `Func`
- `Expr` to define algebraic expressions of `Func`s and `Var`s.
- `Image` to define an array used as input and output.

A basic program written in Halide is shown in Listing 11.3.

```
1 Image<float> input = load<float>("images/rgb.png");
2
3 Var x, y;
4 Func blurx, blury;
5
6 blurx(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;
7 blury(x, y) = (blurx(x, y-1) + blurx(x, y) + blurx(x, y+1))/3;
8
9 Image<float> output = blury.realize(input.width()-2, input.heigth()-2);
```

Listing 11.3: A basic program in Halide.

11.4.2 Scheduling

Halide allows to specify when a function should be called. In particular we can:

- Compute and store the producer values, then compute the consumer values. This avoids computing a result twice but achieves poor locality.
- Compute producer values when needed by the consumer, then throw them away. This allows to have a good locality and to avoid redundant work but it doesn't achieve good parallelism.
- Compute producer values when needed by the consumer, then throw them away. This allows to have a good locality and to avoid redundant work.
- Compute a subset of producer values all at once, then pass them to the consumer. The performance in this case depends on many factors.

Let us now consider a three-dimensional image to show some scheduling instructions. Lets start from the serial code that follows:

```
brighten(x, y, c) = for c:
                        for y:
                            for z:
                                brighten(...) = ...

Default:
    Serial c,
    Serial y,
    Serial x
```

This piece of code serially increases the brightness of each pixel by a fixed value.

Parallelisation

Let's say that we know that the brighten operation can be independently computed on each column, hence we want to parallelise it. We should write

```
brighten(x, y, c) = for c:
                        parallel y:
                            for z:
                                brighten(...) = ...

brighten.parallel(y)
```

to specify that the second for loop has to be parallelised, using some implementation which we can't control.

Vectorisation

We can also specify that we want to apply vectorisation over some variable by writing

```
brighten(x, y, c) = for c:
                        parallel y:
                            for z:
                                vectorized x.v in [0,7]:
                                    brighten(...) = ...

brighten.parallel(y)
        .vectorized(x, 8)
```

In this case we must also specify the size of the vector we want to use. Halide also allows to unroll loops by writing

```
brighten(x, y, c) = for c:
                        parallel y:
                            for z:
                                unrolled x.v in [0,3]:
                                    brighten(...) = ...

brighten.parallel(y)
        .unroll(x, 4)
```

Split

In Halide we can also split some loop by writing

```
brighten(x, y, c) = for c:
                        for y0:
                            for y1 in [0,63]:
                                for z:
                                    brighten(...) = ...

brighten.split(y, y0, y1, 64)
```

Reorder

The order in which loops are executed can be changed by writing

Split

In Halide we can also split some loop by writing

```
brighten(x, y, c) = for y0:
                        for c:
                            for y1 in [0,63]:
                                for z:
                                    brighten(...) = ...

brighten.split(y, y0, y1, 64)
        .reorder(c, y0)
```

If we need to do splitting and reordering over multiple variables we can directly use tiling with the following syntax

```
Var x, y;
blur_y(x, y) = (input(x, y) + input(x, y+1) + input(x, y+2))/3;
Var x0, y0, x1, y1;
blur_y.tile(x, y, x0, y0, x1, y1, 256, 32);
```

The explicit code that implements the tile is:

```
allocate out(width, height)
for y0 in (0..height/32):
    for x0 in (0..width/256):
        for y1 in (0..32):
            y = y0*32 + y1
            for x1 in (0..256):
                x = x0*256 + x1
                out[x,y] = (input(x, y) + input(x, y+1) + input(x, y+2))/3
```

Definitions

Algorithms

Theorems and principles

Propositions