Concurrency vs Parallelism

Ashish Pratap Singh

7 min read

Concurrency and **parallelism** are two of the most misunderstood concepts in **system design**.

While they might sound similar, they refer to fundamentally different approaches to handling tasks.

Simply put, one is about **managing** multiple tasks simultaneously, while the other is about **executing** multiple tasks at the same time.

In this chapter, we'll break down the differences between these two concepts, explore how they work, and illustrate their real-world applications with examples and code.

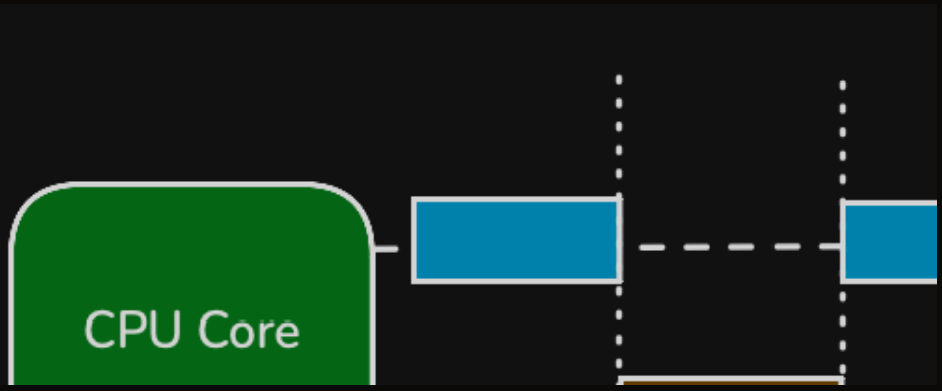
1. What is Concurrency?

> *Concurrency* means an application is making progress on more than one task at the same time.

n a computer, the tasks are executed using **Central Processing Unit (CPU).**

While a single **CPU** can work on only one task at a time, it achieves concurrency by rapidly switching between tasks.

For example, consider playing music while writing code. The CPU alternates between these tasks so quickly that, to the user, it feels like both are happening at the same time.



This seamless switching—enabled by modern CPU designs—creates the **illusion of multitasking** and gives the appearance of tasks running in parallel.

However, it's important to note **this is not parallel**. **This is concurrent**.

Concurrency is primarily achieved using **threads**, which are the smallest units of execution within a process. The CPU switches between threads to handle multiple tasks concurrently, ensuring the system remains responsive.

The primary objective of concurrency is to **maximize CPU utilization** by minimizing idle time.

For example:

- When one thread or process is waiting for I/O operations, database transactions, or external program launches, the CPU can allocate resources to another thread.

This ensures the CPU remains productive, even when individual tasks are stalled.

How Does Concurrency Works?

Concurrency in a CPU is achieved through **context switching**.

Here's how it works:

1. **Context Saving**: When the CPU switches from one task to another, it saves the current task's state (e.g., program counter, registers) in memory.
2. **Context Loading**: The CPU then loads the context of the next task and continues executing it.

3. **Rapid Switching**: The CPU repeats this process, switching between tasks so quickly that it seems like they are running simultaneously.

The Cost of Context Switching

While context switching enables concurrency, it also introduces **overhead**:

- Every switch requires saving and restoring task states, which consumes both time and resources.
- Excessive context switching can degrade performance by increasing CPU overhead.

Real-World Examples of Concurrency

1. Web Browsers

Modern web browsers perform multiple tasks concurrently:

- Rendering web pages (HTML/CSS).
- Fetching external resources like images and scripts.
- Responding to user actions such as clicks and scrolling.

Each of these tasks is managed by separate threads, ensuring the browser remains responsive while loading and displaying content.

2. Web Servers

Web servers like Apache or Nginx handle multiple client requests concurrently:

- Each request is processed independently using threads or asynchronous I/O.
- For example, a server can handle multiple users loading different pages simultaneously without blocking.

3. Chat Applications

Chat applications perform several operations concurrently:

- Processing incoming messages.
- Updating the user interface with new messages.
- Sending outgoing messages.

This ensures smooth real-time communication without delays or freezes.

4. Video Games

Video games rely heavily on concurrency to deliver an immersive experience:

- Rendering graphics.
- Processing user input (e.g., character movement).
- Simulating physics.
- Playing background audio.

For example, while a player moves a character, the game simultaneously updates the environment and plays music, ensuring smooth gameplay.

Code Example

Most popular programming languages come with inbuilt support for creating and managing threads.

Here's an example of a concurrent program in Java:

Java

```java
class Task implements Runnable {
    private String taskName;

    public Task(String taskName) {
        this.taskName = taskName;
    }

    @Override
    public void run() {
        for (int i = 1; i <= 5; i++) {
            System.out.println(taskName + " - Step "
            try {
                Thread.sleep(500); // Simulate I/O c
            } catch (InterruptedException e) {
                System.out.println(taskName + " inte
            }
        }
```

```
    }
}

public class Main {
    public static void main(String[] args) {
        Thread task1 = new Thread(new Task("Task A"))
        Thread task2 = new Thread(new Task("Task B"))
        Thread task3 = new Thread(new Task("Task C"))

        task1.start();
        task2.start();
        task3.start();
    }
}
```

Output (Interleaved Execution):

```
Task A - Step 1
Task B - Step 1
Task C - Step 1
Task B - Step 2
Task A - Step 2
Task C - Step 2
Task B - Step 3
Task C - Step 3
Task A - Step 3
Task A - Step 4
Task B - Step 4
Task C - Step 4
Task A - Step 5
Task C - Step 5
```

```
Task B - Step 5
```

---

2. What is Parallelism?

> *Parallelism* means multiple tasks are executed simultaneously.

To achieve parallelism, an application divides its tasks into smaller, independent subtasks. These subtasks are distributed across multiple CPUs, CPU cores, GPU cores, or similar processing units, allowing them to be processed in parallel.

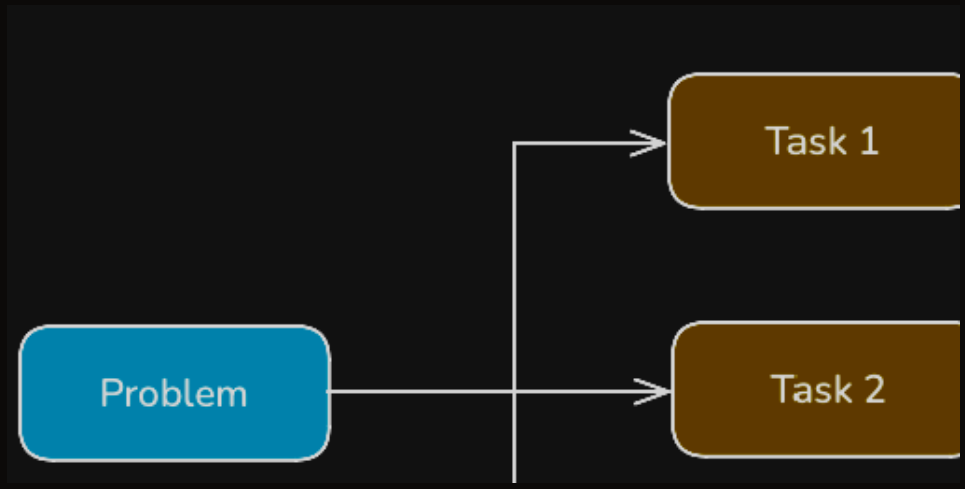
To achieve true parallelism, your application must:

1. Utilize more than one thread.

2. Ensure each thread is assigned to a separate CPU core or processing unit.

How does Parallelism Works?

Modern CPUs consist of multiple cores. Each core can independently execute a task. Parallelism divides a problem into smaller parts and assigns each part to a separate core for simultaneous processing.



- **Task Division**: The problem is broken into smaller independent sub-tasks.
- **Task Assignment**: Sub-tasks are distributed across multiple CPU cores.
- **Execution**: Each core processes its assigned task simultaneously.
- **Result Aggregation**: Results from all cores are combined

to form the final output.

Real-World Examples of Parallelism

1. Machine Learning Training

- Training deep learning models involves dividing datasets into smaller batches.
- Each batch is processed simultaneously across multiple GPUs or CPU cores, significantly speeding up the training process.

2. Video Rendering

- Video frames are rendered independently, making it possible to process multiple frames simultaneously.
- For example, rendering a 3D animation becomes much faster when using multiple cores to handle different frames in parallel.

3. Web Crawlers

- Web crawlers like Googlebot break a list of URLs into smaller chunks and process them in parallel.
- This allows the crawler to fetch data from multiple websites simultaneously, reducing the time to gather information.

4. Data Processing

- Big data frameworks like Apache Spark leverage parallelism to handle massive datasets.
- Tasks such as analyzing logs from millions of users are distributed across a cluster, enabling simultaneous processing and faster insights.

5. Scientific Simulations

- Simulations like weather modeling or molecular interactions require heavy computations.
- These computations are divided among multiple cores, allowing simultaneous execution and faster results.

Code Example

Here's a simple example of parallelism in Java using the `ForkJoinPool` framework to compute the sum of an array in parallel:

Java

```java
import java.util.concurrent.RecursiveTask;
import java.util.concurrent.ForkJoinPool;

class ParallelSum extends RecursiveTask<Integer> {
    private int[] numbers;
    private int start, end;
```

```
    private static final int THRESHOLD = 10;

    ParallelSum(int[] numbers, int start, int end) {
        this.numbers = numbers;
        this.start = start;
        this.end = end;
    }

    @Override
    protected Integer compute() {
        if (end - start <= THRESHOLD) {
            int sum = 0;
            for (int i = start; i < end; i++) sum +=
            return sum;
        } else {
            int mid = (start + end) / 2;
            ParallelSum leftTask = new ParallelSum(r
            ParallelSum rightTask = new ParallelSum(
            leftTask.fork(); // Execute left task ir
            return rightTask.compute() + leftTask.jo
        }
    }
}

public class Main {
    public static void main(String[] args) {
        int[] numbers = java.util.stream.IntStream.r
        ForkJoinPool pool = new ForkJoinPool();
        int result = pool.invoke(new ParallelSum(num
        System.out.println("Sum: " + result); // Out
    }
```

```
}
```

1. **Task Splitting**: The array is divided into smaller segments until the segment size is below the `THRESHOLD`.
2. **Parallel Execution**: Subtasks are executed in parallel using separate threads from the `ForkJoinPool`.
3. **Result Combination**: Results from all subtasks are combined to compute the final sum.

---

3. Concurrency and Parallelism Combinations

3.1 Concurrent, Not Parallel

An application can be concurrent without being parallel. In this case:

- The application makes progress on multiple tasks at the same time **seemingly** (concurrently).
- However, it achieves this by **switching** between tasks rapidly, rather than running them simultaneously.
- **Example**: A single-core CPU alternating between tasks, giving the illusion of multitasking.

3.2 Parallel, Not Concurrent

An application can be parallel without being concurrent. Here:

- A single task is divided into subtasks, and these subtasks are executed simultaneously on separate cores.
- There is no overlap between tasks; one task (and its subtasks) completes before the next task starts.
- **Example**: Video rendering, where a single video is divided into frames, and each frame is processed in parallel.

3.3 Neither Concurrent Nor Parallel

Some applications are neither concurrent nor parallel. This means:

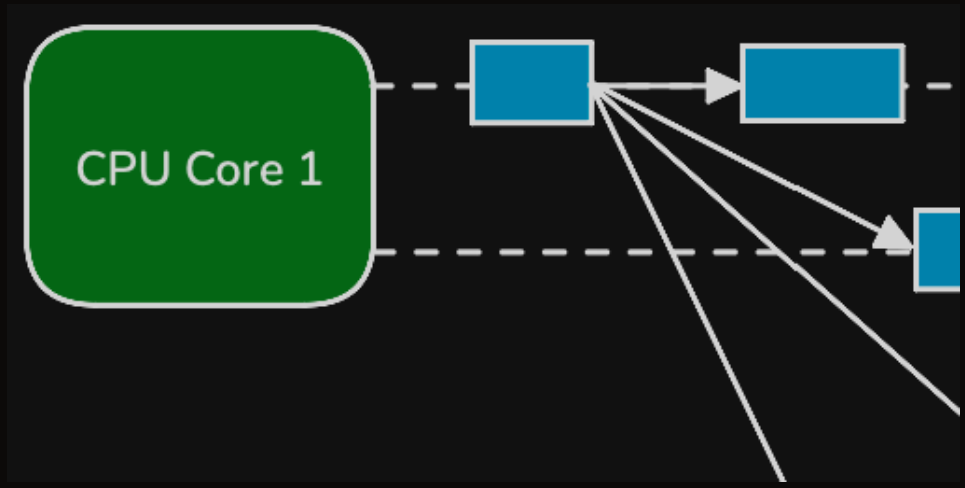
- Tasks are executed sequentially, one at a time, without any overlap or parallel execution.
- **Example**: A single-core CPU where only one task is processed, and it completes fully before the next task begins.

3.4 Concurrent and Parallel

An application can be both **concurrent and parallel**, combining the strengths of both execution models.

In this approach:

1. Multiple tasks make progress at the same time, and each task is also divided into subtasks that are executed in parallel.
2. **Example**: A Multi-core CPU where some subtasks run concurrently on the same core, while others run in parallel on separate cores.



In the above example, a single task is broken into 4 **subtasks**, which are distributed across 2 CPU cores for **parallel execution**. These subtasks are executed by multiple threads. Some threads run on the same CPU core (concurrent execution), while others run on separate CPU cores (parallel execution).

If each subtask is executed by its own thread on a **dedicated CPU** (e.g., 4 threads on 4 CPUs), the task execution be-

comes fully **parallel**, with no concurrency involved.

It’s often challenging to break a task into exactly as many subtasks as there are CPUs. Instead, tasks are typically divided into a number of subtasks that align naturally with the problem's structure and number of CPU cores available.

---

4. Summary

Aspect	Concurrency	Parallelism
Definition	Dealing with multiple tasks by interleaving their execution. Tasks may overlap but may not run simultaneously.	Executing multiple tasks at the same time on separate cores or processors.
Execution	Achieved through context switching on a single core or thread.	Requires multiple cores or processors to execute tasks simultaneously.
Focus	Managing multiple tasks and maximizing resource utilization.	Splitting a single task into smaller sub-tasks for simultaneous execution.

Use Case	Best suited for I/O-bound tasks like handling multiple network requests or file operations.	Ideal for CPU-bound tasks like data processing or machine learning training.
Resource Requirement	Can be implemented on a single core or thread.	Requires multiple cores or threads.
Outcome	Improves responsiveness by efficiently managing task switching.	Reduces overall execution time by performing tasks simultaneously.
Examples	Asynchronous APIs, chat applications, or web servers handling multiple requests.	Video rendering, machine learning training, or scientific simulations.
Analogy	A single chef multitasking — preparing multiple dishes by working on them in parts.	Multiple chefs working on different dishes at the same time.