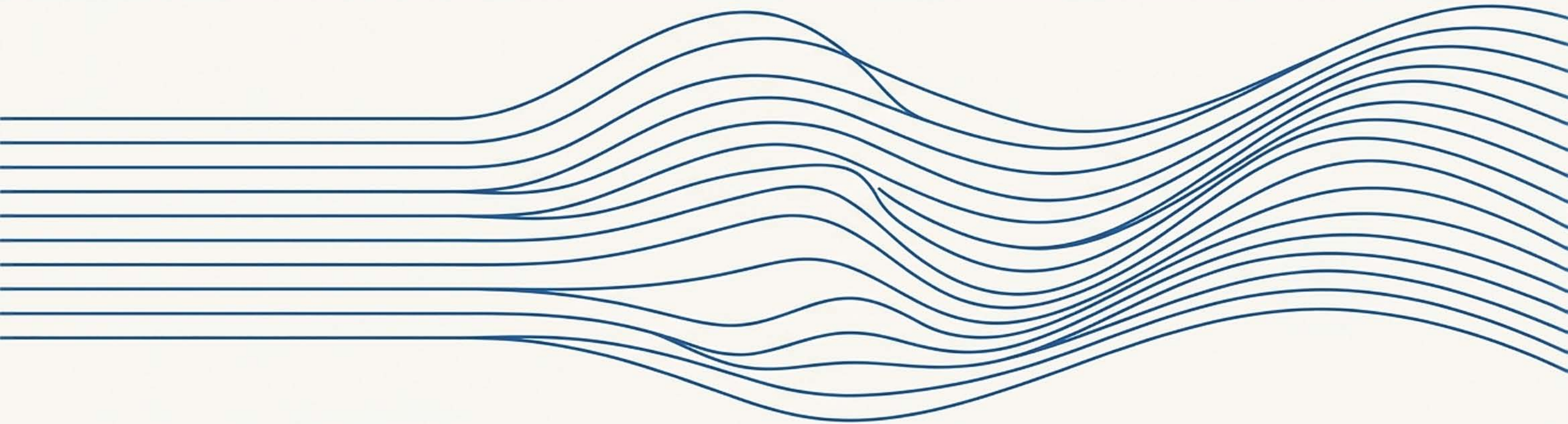
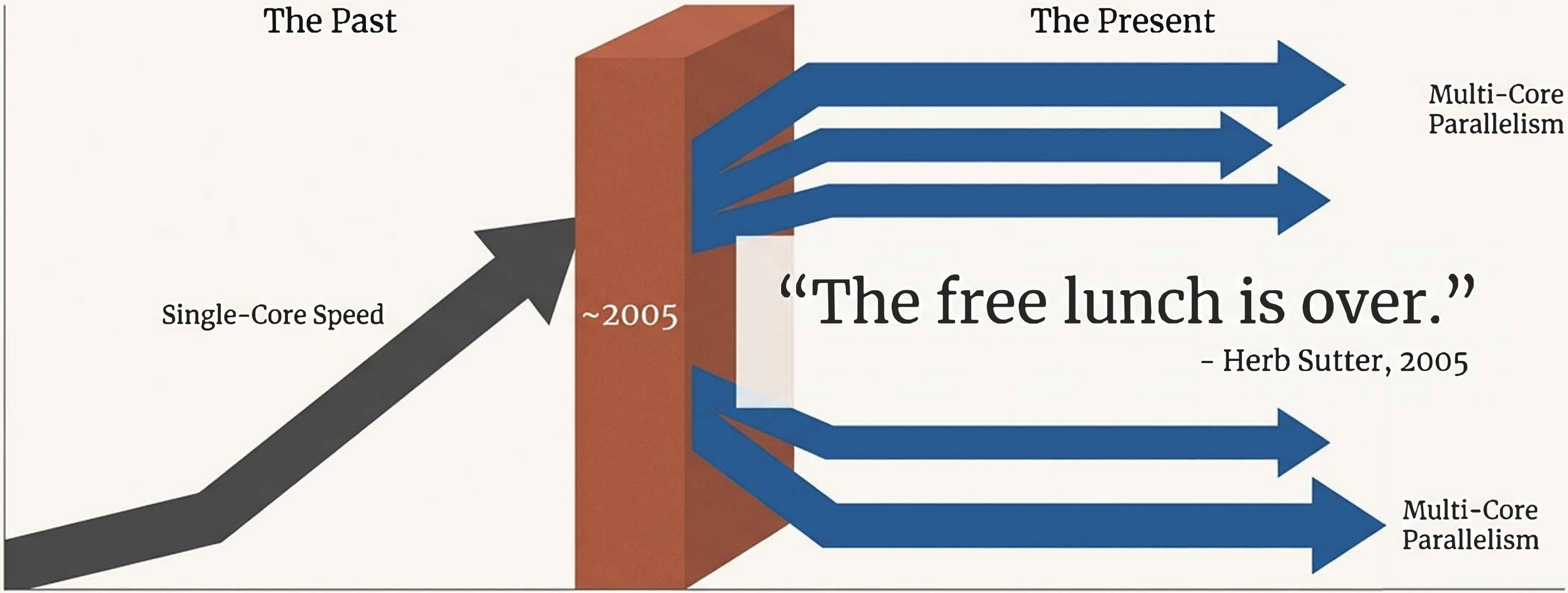
Concurrency: A Strategic Imperative for Modern Software

Dr. Talgat Manglayev

Lecture-1-2



The Landscape Has Fundamentally Shifted



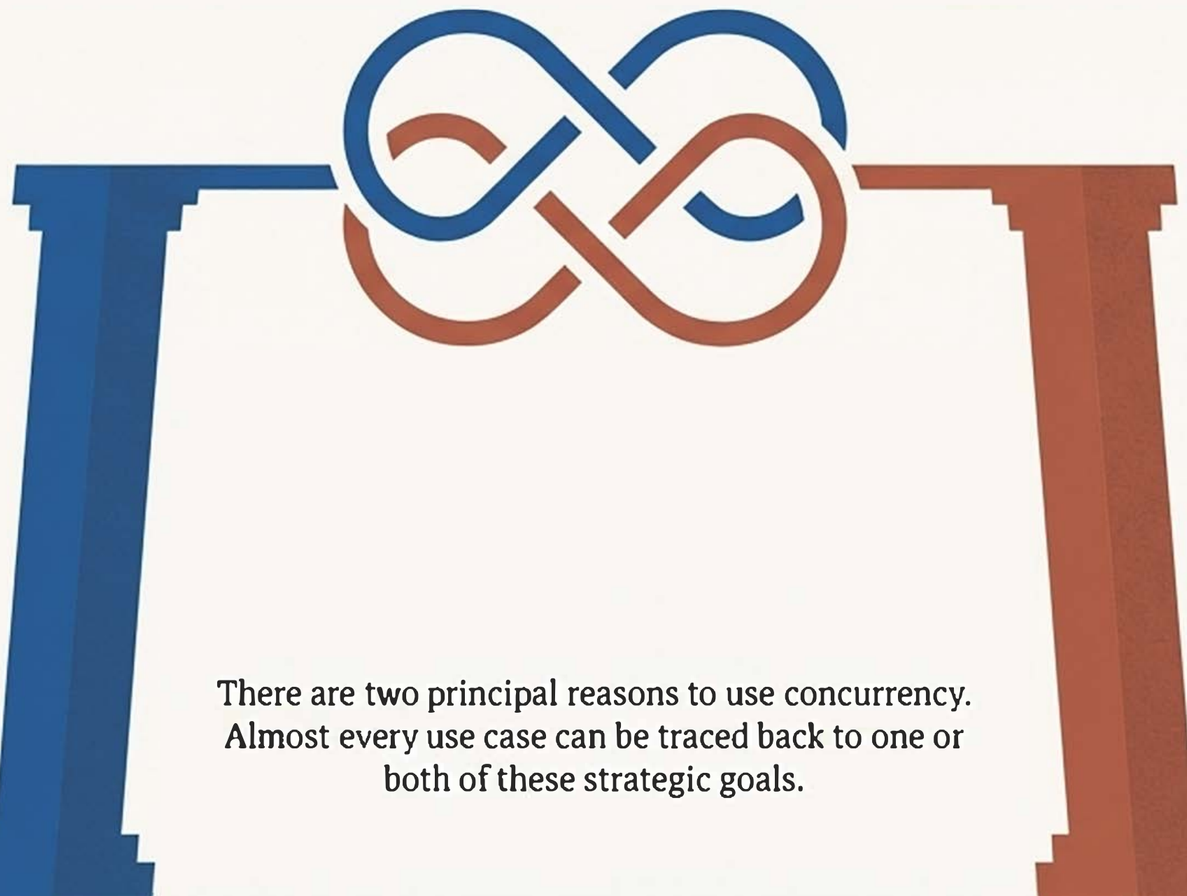
For decades, programmers could rely on single-core processor speeds to increase application performance automatically. This era has ended. Chip manufacturers now favor multi-core designs, increasing computing power by running multiple tasks in parallel, not by making a single core faster.

Concurrency is the Necessary Response

To harness the power of multi-core hardware and build robust modern applications, software must be designed to run multiple tasks concurrently.

1. Separation of Concerns

For clarity, responsiveness, and simpler logic.



2. Performance

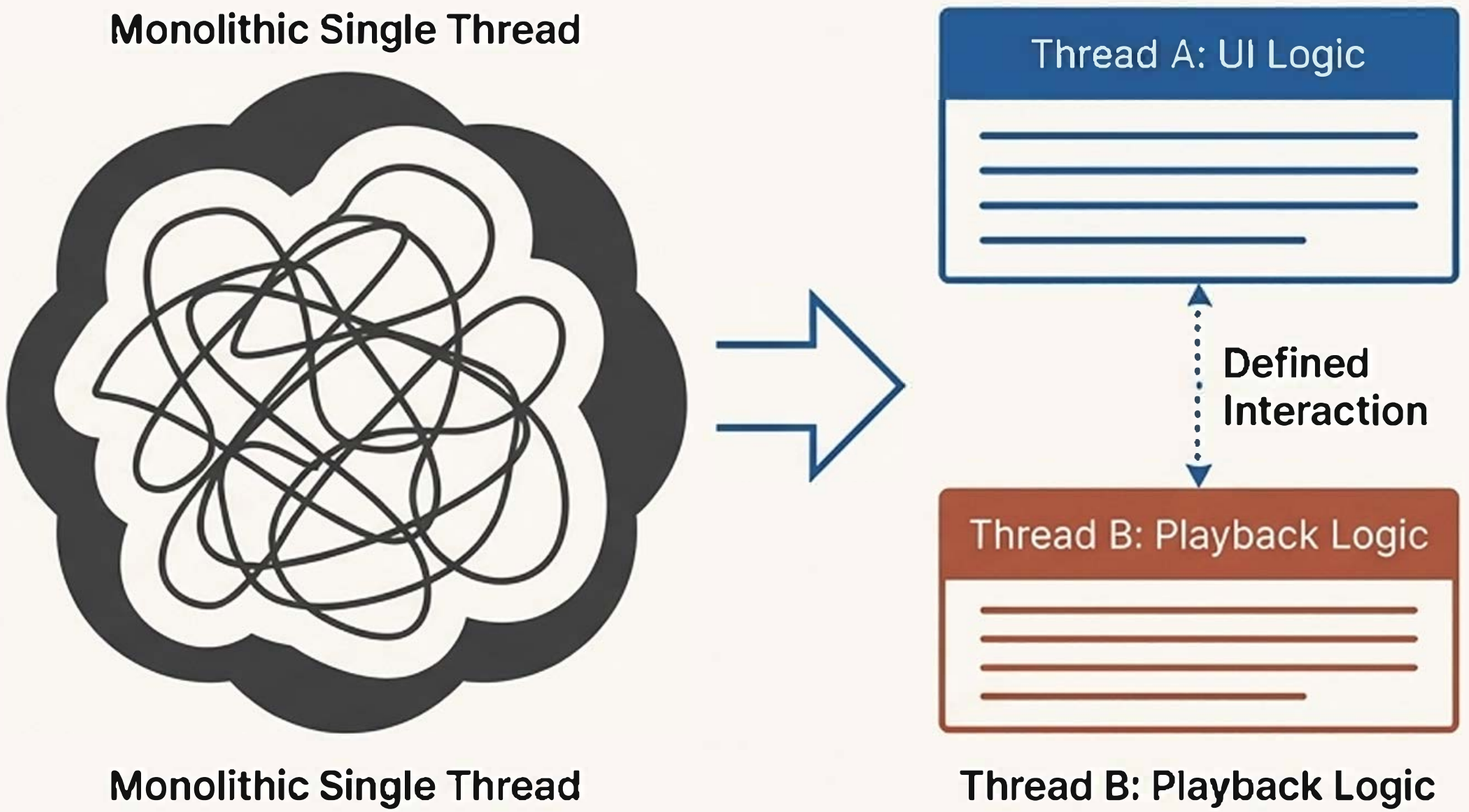
For raw speed and increased throughput.

There are two principal reasons to use concurrency. Almost every use case can be traced back to one or both of these strategic goals.

Pillar 1: Using Concurrency for Separation of Concerns

Good software design involves grouping related code and separating unrelated code.

Concurrency allows you to separate distinct areas of functionality, even when they need to operate at the same time.
This makes programs easier to understand, test, and maintain, reducing the likelihood of bugs.



Key Insight: With this approach, the number of threads is based on the *conceptual design* of the application, not the number of available CPU cores.

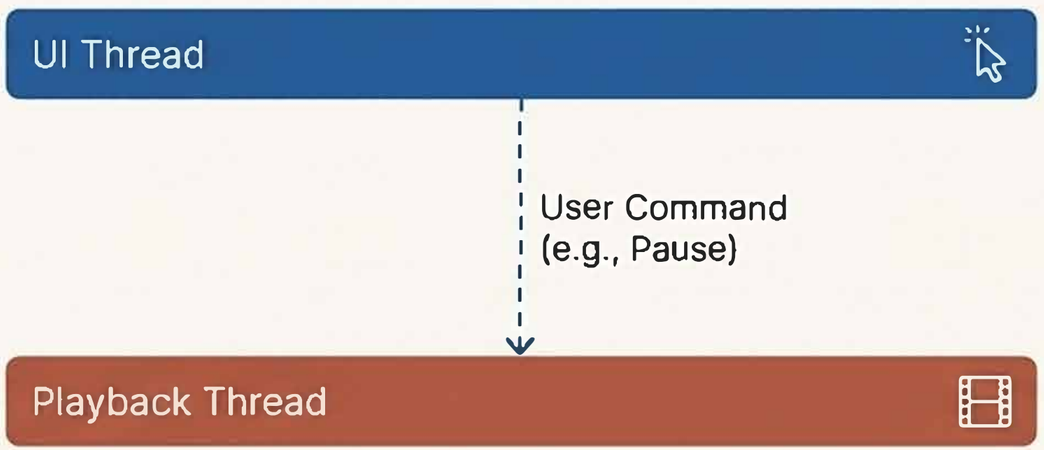
Case Study: A DVD Player Application

The Challenge (Single-Threaded)

A single thread must handle two distinct responsibilities:

1. **Playback:** Reading from disk, decoding video/audio, sending to hardware.
2. **User Interface:** Responding to user input like “Pause,” “Quit,” or “Return to Menu.”

This forces the playback code to be constantly interrupted to check for user input, conflating the two areas of logic.



The Solution (Multi-Threaded)

- **Thread 1 (UI Thread):** Manages all user interaction. Is always ready to respond.
- **Thread 2 (Playback Thread):** Manages all data processing and playback.

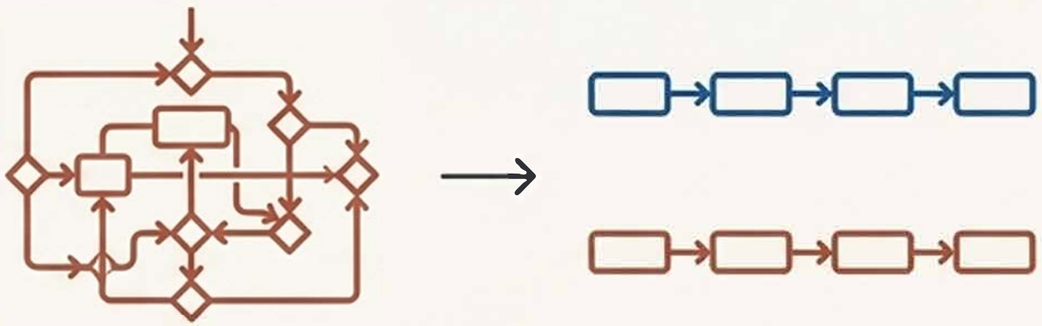
Interactions between threads are limited to clearly defined points (e.g., the UI thread sending a "pause" command to the playback thread).

The Result: A Responsive and Simpler Design

Illusion of Responsiveness

The user interface thread can respond *immediately* to a user request. Even if the main task is busy, the UI can acknowledge the request by displaying a 'Please Wait' message or a busy cursor.

Simplified Logic



The logic within each thread becomes much simpler and more focused. The code for playback doesn't need to be mixed with code for handling button clicks.

Broader Application

This pattern is also ideal for tasks that must run continuously in the background, like a desktop search application monitoring the filesystem for changes.

Pillar 2: Using Concurrency for Performance

This is the direct answer to the end of single-core performance gains. Concurrency allows us to leverage multi-core processors to execute tasks faster or to handle more work in the same amount of time.

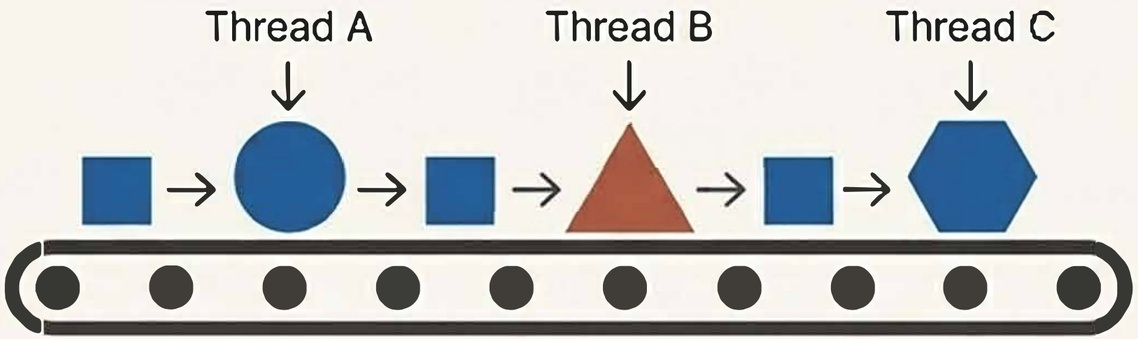
Two Primary Methods

1. **Task Parallelism:** Dividing a single, complex task into different parts and running them in parallel.
2. **Data Parallelism:** Performing the same operation on different chunks of data simultaneously.

Deconstructing Parallelism

Task Parallelism:

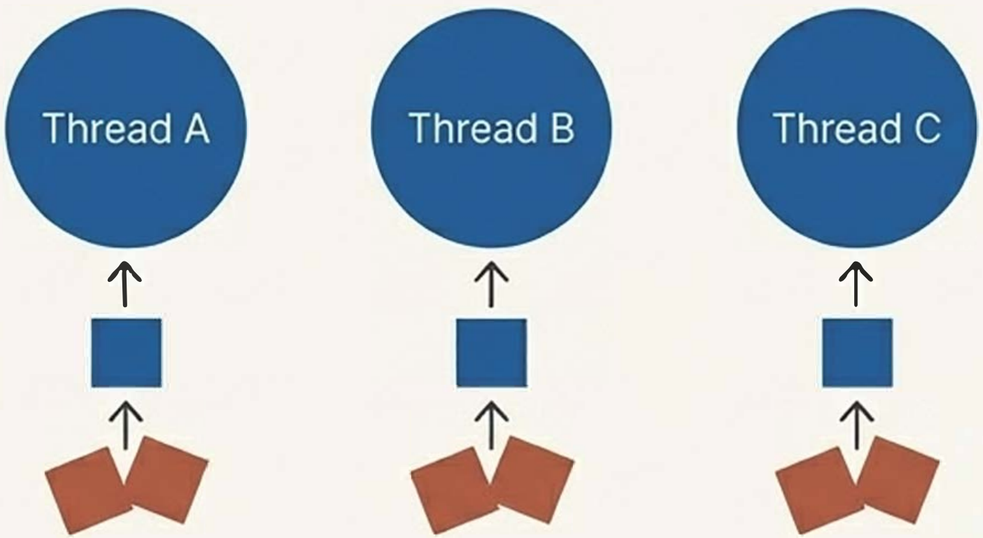
One thread performs one part of an algorithm while another thread performs a different part. Can be complex due to dependencies.



Data Parallelism:

Each thread performs the exact same operation, but on a different piece of the data. Often called “embarrassingly parallel” or “naturally parallel”, these algorithms scale well with more hardware.

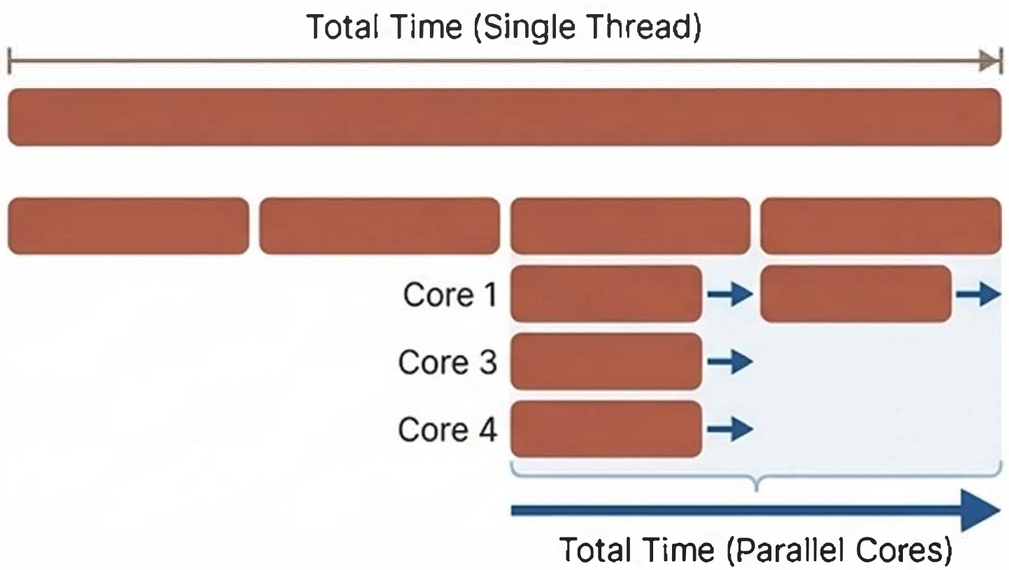
Adage: “Many hands make light work.”



Two Strategic Outcomes of Performance

Reduce Latency (Process Faster)

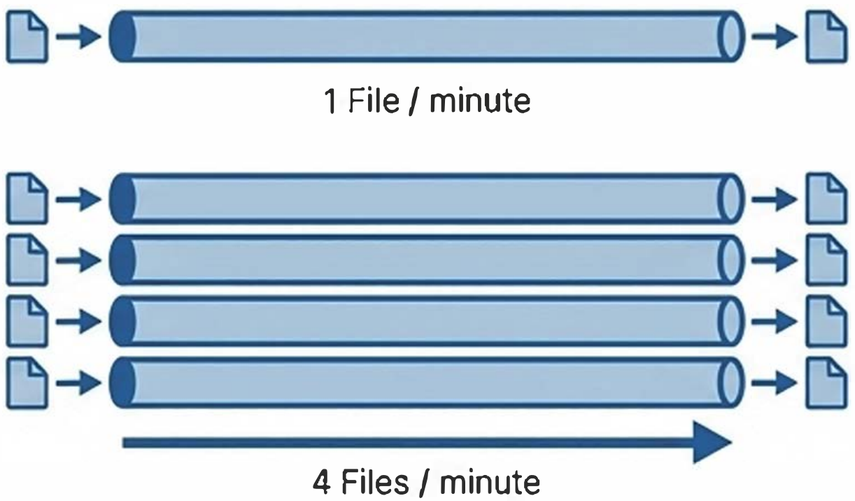
Goal: To divide a single task into parts to reduce its total runtime.



Example: Running a complex calculation on 8 cores to get the answer in 1/8th the time (in an ideal scenario).

Increase Throughput (Process More)

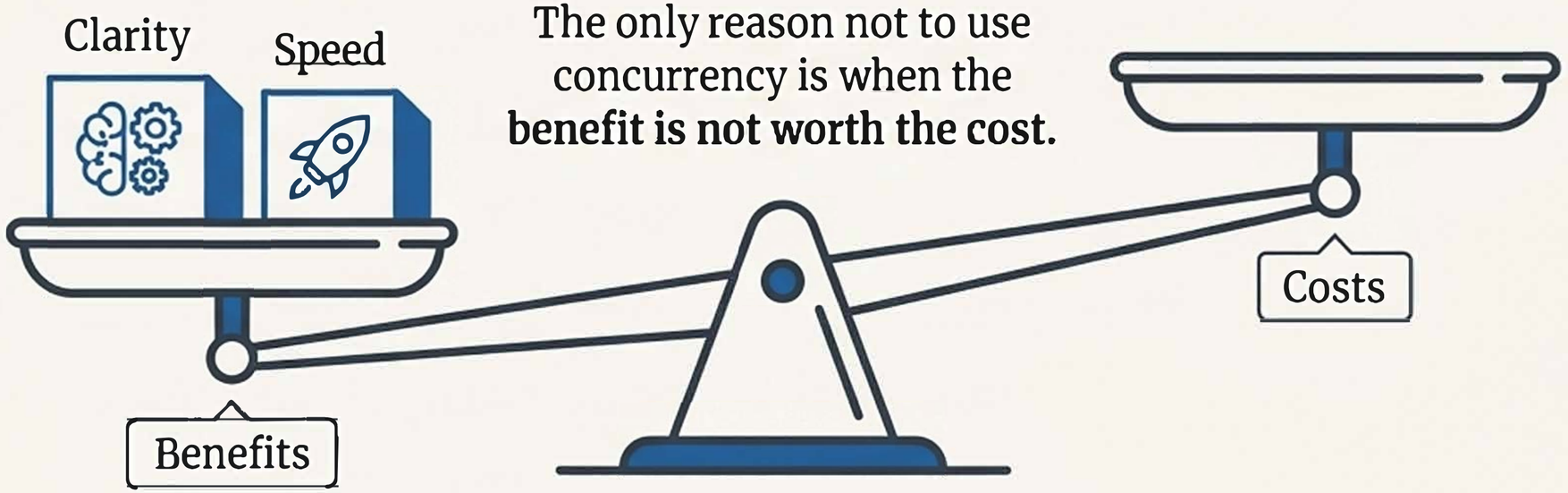
Goal: To use available parallelism to solve bigger problems.



Example: Instead of processing one file at a time, process 20 files simultaneously. This can enable new capabilities, like processing higher-resolution video by working on different areas of the picture in parallel.

When Not to Use Concurrency

It is just as important to know when *not* to use concurrency as it is to know when to use it.



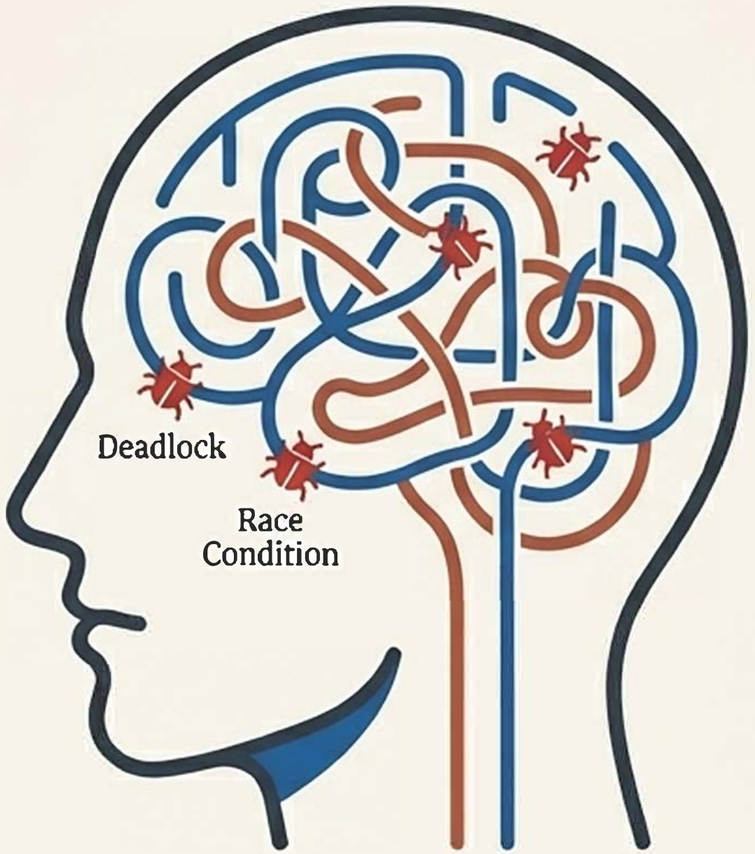
We will examine three categories of cost associated with concurrency:

1. Intellectual Cost & Complexity
2. Performance Overhead
3. Resource Limits

The Cost of Complexity

Intellectual Cost: Code using concurrency is often harder to understand, reason about, and debug. This is a direct intellectual cost to the engineering team.

Increased Bugs: The additional complexity can lead to new classes of bugs that don't exist in single-thraded code (e.g., race conditions, deadlocks).



Unless the **potential performance gain is large enough**, or the **separation of concerns is clear** enough to justify the...

...additional development time to get it right,

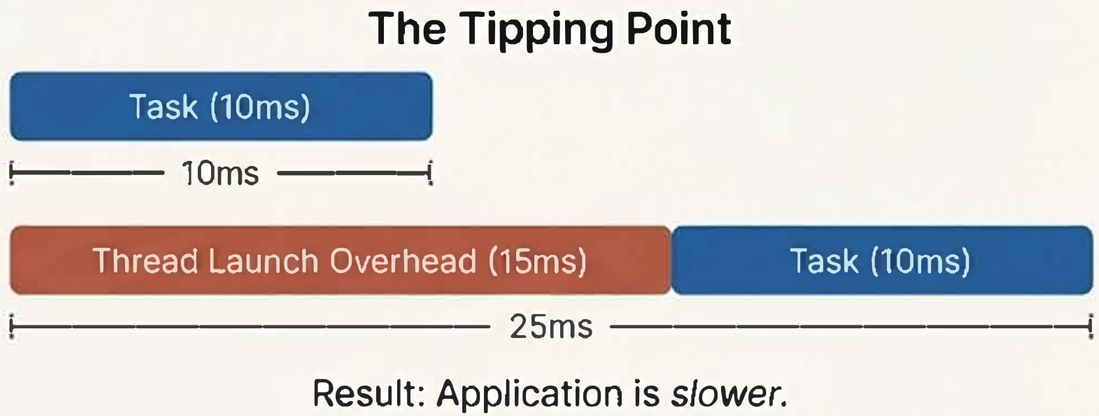
...and the additional maintenance costs over the lifetime of the code,

...do not use concurrency.

The Overhead Cost: Performance Can Suffer

Overhead of Launching a Thread

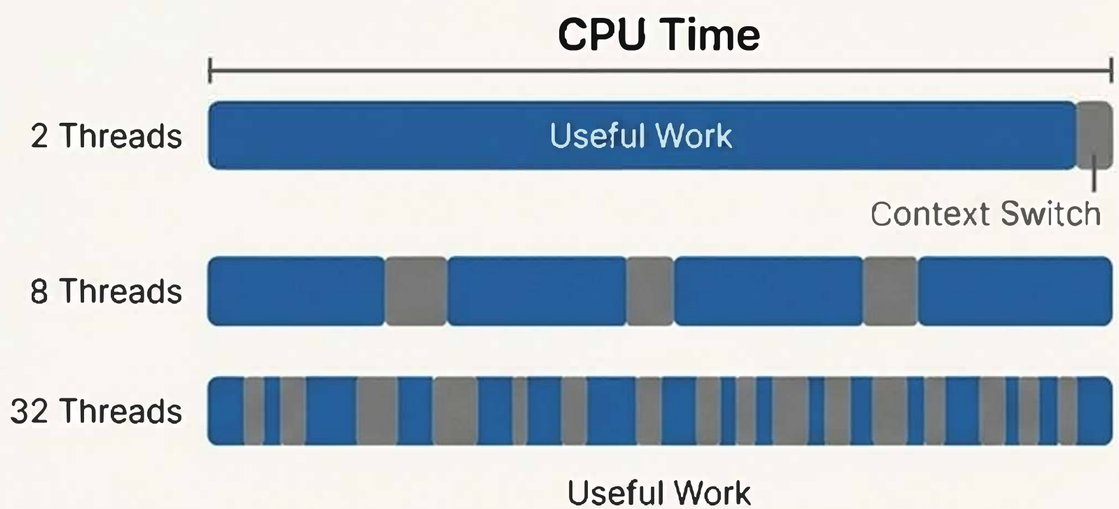
There's an inherent, non-zero cost to starting a thread. The OS must allocate kernel resources, stack space, and add the new thread to the scheduler.



Overhead of Context Switching

The more threads you run, the more the OS has to do context switching. Each switch takes time that could be spent doing useful work.

At some point, adding another thread will *reduce* overall performance.

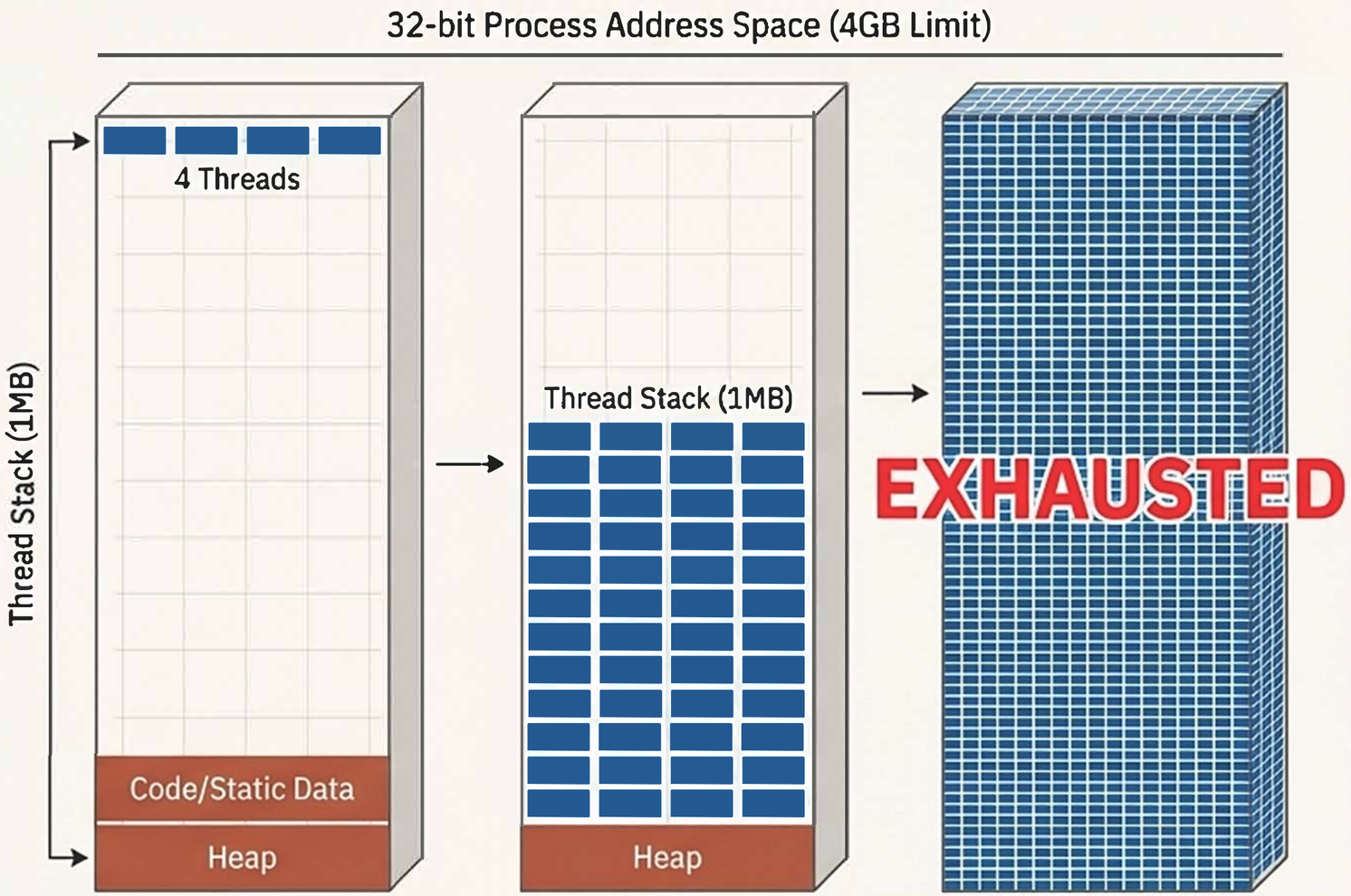


The Resource Cost: Threads Are a Finite Resource

System Resource Consumption: Running too many threads consumes OS resources and can slow down the entire system.

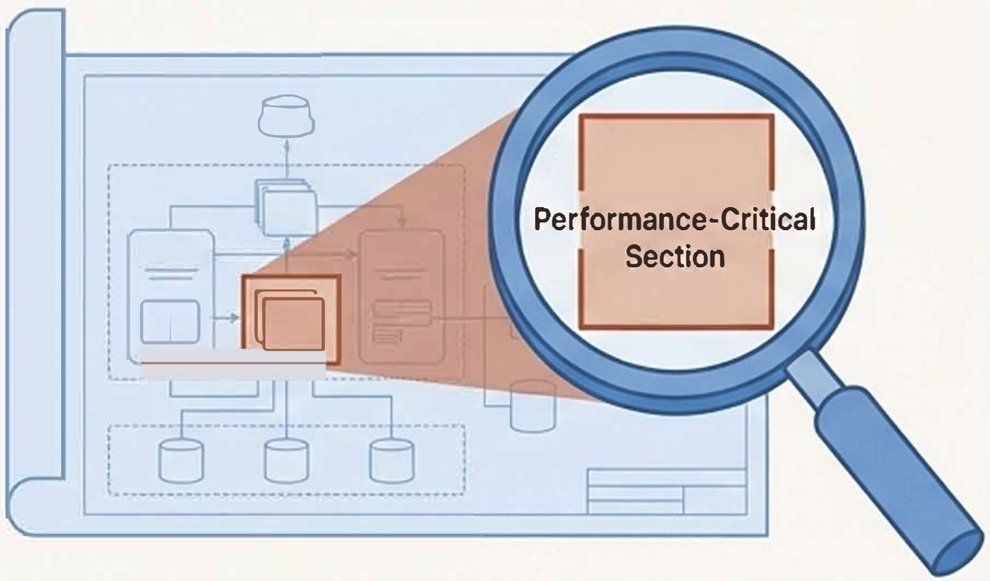
Memory and Address Space Exhaustion: Each thread requires a separate stack space. 64-bit systems have finite memory, but 32-bit systems face a hard address-space limit.

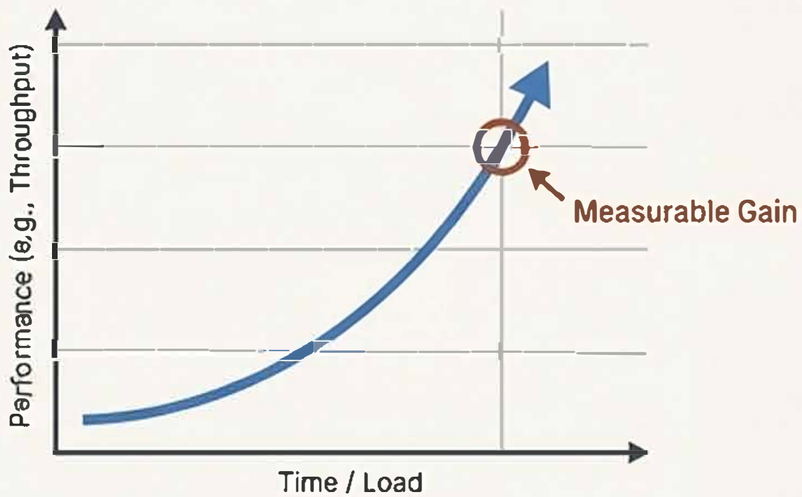
Classic Anti-Pattern: A server that launches a separate thread for each client connection. This works for a few connections, but will quickly exhaust system resources under high demand.



Treat Concurrency as a Deliberate Optimization Strategy

Concurrency has the potential to greatly improve your application, but it also complicates the code, making it harder to understand and more prone to bugs.





Apply concurrency only to the performance-critical parts of the application where there is potential for a **measurable gain.**

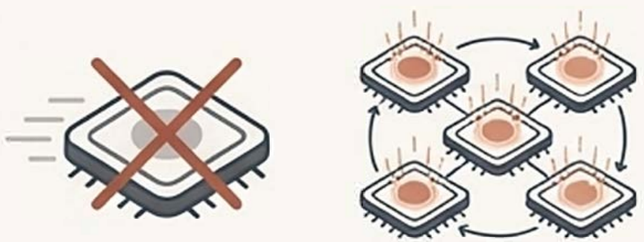
This rule can be relaxed if your primary goal is a clearer design through separation of concerns, and you are willing to accept the complexity trade-off.

To achieve optimal performance, you must tune the number of threads to the available hardware concurrency.

Mastering Concurrency: From Obligation to Strategy

The Problem

The free lunch is over. Single-core speed is no longer the path to performance; multi-core parallelism is the new reality.



The Solution

Concurrency is the essential tool, driven by two key strategic goals:

- **Separation of Concerns:** To build clearer, more responsive, and robust applications.
- **Performance:** To unlock the full power of modern hardware for speed and throughput.

The Discipline

This power comes at a cost—in complexity, overhead, and resources. Effective use demands a **deliberate analysis** of whether the benefits justify these costs.

Concurrency is a core competency **for the modern developer. The true** mark of expertise is not just knowing how to write multi-threaded code, but **knowing precisely *why* and *when*** to apply it.