Concurrency vs Parallelism

The Definitive Distinction

Concurrency: Dealing with multiple things at once (composition/structure)
Parallelism: Doing multiple things at once (simultaneous execution)

This is not semantic hair-splitting. Understanding this difference is critical for:

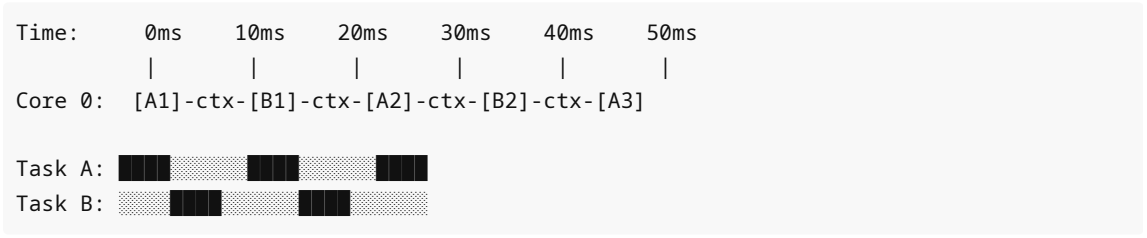
- Choosing the right approach
- Understanding performance characteristics
- Avoiding false assumptions about speed

Rob Pike's Definition

> *Concurrency is about dealing with lots of things at once.*
> *Parallelism is about doing lots of things at once.*
> *Concurrency is about structure, parallelism is about execution.*
> *Concurrency provides a way to structure a solution to solve a problem that may (but not necessarily) be parallelizable.*

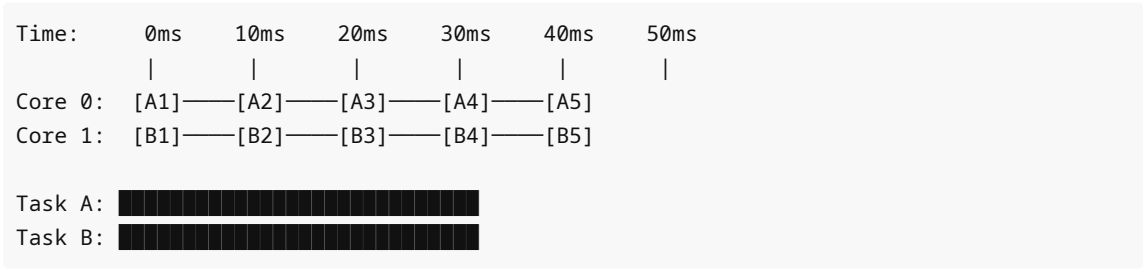
Visual Model

Concurrency (Single Core)



Tasks A and B are **concurrent**—they're structured to make progress independently, but only one executes at any instant. The CPU context-switches between them.

Parallelism (Multi-Core)



Tasks A and B execute **in parallel**—they're literally running at the same instant on different cores.

Concurrent AND Parallel

```
Core 0:  [A1]———[A2]—ctx—[C1]———[C2]———
Core 1:  [B1]———[B2]—ctx—[D1]———[D2]———

Tasks A, B, C, D are concurrent
Tasks A & B are parallel in early intervals
Tasks C & D are parallel in later intervals
```

Four concurrent tasks, but only two execute in parallel at any moment (limited by 2 cores).

Concrete Example: Coffee Shop

Concurrency (One Barista, Many Orders)

```
Barista starts order 1 (espresso)
  ↓ espresso machine running...
  ← Barista takes order 2 (drip coffee)
  ↓ coffee brewing...
  ← Barista takes order 3 (pastry)
  ↓ pastry warming...
  ← Espresso machine beeps
Barista finishes order 1
  ← Coffee pot beeps
Barista finishes order 2
  ← Microwave beeps
Barista finishes order 3
```

One worker, multiple concurrent tasks. The barista doesn't sit idle while machines run—they structure work to make progress on multiple orders.

Parallelism (Multiple Baristas)

```
Barista A: Order 1 (espresso)  ——————→ Done
Barista B: Order 2 (drip)      ——————→ Done
Barista C: Order 3 (pastry)    ——————→ Done
```

Multiple workers, literally working simultaneously. True parallel execution.

Code Example: Summing 1 Million Numbers

Sequential (No Concurrency, No Parallelism)

```go
func sum(numbers []int) int {
    total := 0
    for _, n := range numbers {
        total += n
    }
    return total
}

// Time: 100ms (baseline)
```

Concurrent (Structured, But Might Not Be Parallel)

```
func sumConcurrent(numbers []int) int {
    mid := len(numbers) / 2

    ch := make(chan int)

    go func() {
        sum := 0
        for _, n := range numbers[:mid] {
            sum += n
        }
        ch <- sum
    }()

    go func() {
        sum := 0
        for _, n := range numbers[mid:] {
            sum += n
        }
        ch <- sum
    }()

    sum1, sum2 := <-ch, <-ch
    return sum1 + sum2
}

// On 1 core: Time might be 105ms (worse due to overhead)
// On 2+ cores: Time might be 55ms (better due to parallel execution)
```

Key insight: The code is **concurrent** regardless of cores. Whether it's **parallel** depends on `GOMAXPROCS` and available CPU cores.

Forcing Sequential Execution with Concurrent Structure

```
func sumFakeConcurrent(numbers []int) int {
    runtime.GOMAXPROCS(1) // Force single-threaded execution

    // Same concurrent code as above...
    // Still concurrent (structure)
    // Not parallel (execution)
}

// Time: 105ms (concurrent overhead, no parallel benefit)
```

The Critical Distinction in Go

GOMAXPROCS: The Knob Between Concurrency and Parallelism

```
runtime.GOMAXPROCS(1)  // Concurrent but not parallel
runtime.GOMAXPROCS(4)  // Concurrent AND parallel (up to 4 cores)
```

- **Concurrency** is a property of your code structure (goroutines, channels)
- **Parallelism** is a property of your runtime environment (cores, `GOMAXPROCS`)

You write concurrent code. The runtime decides if it's parallel.

When Concurrency Doesn't Imply Speed

I/O-Bound: Concurrency Helps (Even on 1 Core)

```
func fetchAll() {
    // Sequential: 3 seconds
    user := fetchUser()      // 1s network wait
    posts := fetchPosts()    // 1s network wait
    comments := fetchComments() // 1s network wait

    // Concurrent on 1 core: Still ~1 second!
    // Why? While waiting for network, other goroutines run
    go func() { user = fetchUser() }()
    go func() { posts = fetchPosts() }()
    go func() { comments = fetchComments() }()
}
```

Concurrency helps because you're not doing work—you're waiting. Other tasks can progress during waits.

CPU-Bound: Concurrency Doesn't Help (on 1 Core)

```
func calculateAll() {
    // Sequential: 3 seconds
    result1 := fibonacci(45) // 1s CPU work
    result2 := fibonacci(45) // 1s CPU work
    result3 := fibonacci(45) // 1s CPU work

    // Concurrent on 1 core: Still 3+ seconds
    // Why? You need the CPU continuously; no other core to use
    go func() { result1 = fibonacci(45) }()
    go func() { result2 = fibonacci(45) }()
    go func() { result3 = fibonacci(45) }()
}
```

Concurrency doesn't help because the CPU is the bottleneck. You need **parallelism** (multiple cores) to speed this up.

Real-World Failure: Misunderstanding Concurrency

Company: A fintech startup (2019)

What happened: Engineers built a payment processor with goroutines for each transaction. They assumed concurrency would make it faster. In production, throughput was **worse** than the sequential version.

Root cause: The workload was CPU-bound (cryptographic signing) running on 2 cores. They spawned 1000 goroutines, causing:

- Excessive context switching overhead
- Cache thrashing
- Scheduler contention

Fix: Bound concurrency to `runtime.NumCPU()` (2 goroutines). Throughput increased 3x.

Lesson:

> *More goroutines ≠ faster code. Match concurrency level to problem type and core count.*

Decision Matrix: When Does Concurrency Help?

Workload Type	Single Core	Multiple Cores	Why
I/O-bound (network, disk)	✅ Much faster	✅ Much faster	Overlap wait times
CPU-bound (computation)	❌ Same or slower	✅ Faster	Needs parallel execution
Mixed I/O + CPU	✅ Somewhat faster	✅ Much faster	Overlap I/O, parallelize CPU
Coordination-heavy	❌ Slower	❌ Slower	Synchronization overhead dominates

Concurrency Enables Parallelism

You cannot have parallelism without concurrency:

```
// Not concurrent → cannot be parallel
func notConcurrent() {
    a()
    b()
    c()
}

// Concurrent → can be parallel if cores available
func concurrent() {
    go a()
    go b()
    go c()
}
```

Concurrency is the structure that enables parallelism.

Without concurrent structure (independent tasks), the runtime has nothing to parallelize.

Interview Traps

Trap 1: "My code has goroutines so it's parallel"

Wrong. Goroutines make code concurrent. Parallelism requires multiple cores and depends on the runtime scheduler.

Correct answer:
"My code is structured concurrently using goroutines, which allows the Go runtime to execute it in parallel if multiple cores are available and `GOMAXPROCS` > 1. Concurrency is the structure; parallelism is one possible execution model."

Trap 2: "Concurrency always improves performance"

Wrong. Concurrency adds overhead (scheduling, synchronization). It improves performance only when:

- You're I/O-bound (overlap wait times), or
- You're CPU-bound AND have multiple cores (parallel execution)

Correct answer:
"Concurrency helps when tasks can make independent progress. For I/O-bound work, concurrency helps even on a single core by overlapping wait times. For CPU-bound work, you need both concurrency (structure) and parallelism (multi-core execution) to see improvements."

Trap 3: "Setting GOMAXPROCS(100) makes my program 100x faster"

Wrong. `GOMAXPROCS` limits OS threads, not goroutines. Setting it higher than core count usually hurts (context switching overhead).

Correct answer:
"`GOMAXPROCS` controls how many OS threads can execute Go code simultaneously. Setting it above the number of CPU cores typically degrades performance due to context switching. The default (runtime.NumCPU()) is usually optimal."

Trap 4: "Parallelism is always better than sequential"

Wrong. Parallel execution has overhead: synchronization, cache coherency, work distribution.

Correct answer:
"Parallel execution is beneficial when the problem decomposes independently and the work per task exceeds synchronization overhead. For small tasks or tightly coupled work, sequential execution can be faster."

Mental Model: The Restaurant Analogy

Aspect	Restaurant	Computing
Concurrency	Taking multiple orders, managing kitchen flow	Structuring tasks as goroutines
Parallelism	Multiple chefs cooking simultaneously	Tasks executing on multiple cores
Efficiency	One chef handling many dishes efficiently	I/O-bound concurrency on one core

Throughput	More chefs = more meals per hour	CPU-bound work on multiple cores
Overhead	Chefs coordinating costs time	Synchronization and context switching

Key Takeaways

1. **Concurrency = Structure** (how you organize code)
2. **Parallelism = Execution** (how runtime runs code)
3. **Concurrency without parallelism** is still useful (I/O-bound work)
4. **Parallelism requires concurrency** (you must structure independent tasks)
5. **More goroutines ≠ faster** (match to problem and cores)
6. `GOMAXPROCS` controls parallel execution capacity
7. **Profile before parallelizing** (measure, don't assume)

What You Should Be Thinking Now

- "How many cores does my machine have?"
- "Is my workload I/O-bound or CPU-bound?"
- "How does Go schedule goroutines onto threads?"
- "What's the cost of context switching?"

Next: cpu-cores-context-switching.md - We'll answer these questions by looking at how hardware and the Go scheduler work.

Exercises (Do These Before Moving On)

1. Run this on your machine:

```
fmt.Println("CPU cores:", runtime.NumCPU())
fmt.Println("GOMAXPROCS:", runtime.GOMAXPROCS(0))
```

2. Write two versions of a number-summing function: sequential and concurrent. Benchmark both with `GOMAXPROCS=1` and `GOMAXPROCS=4`. Explain the results.

3. Identify a piece of code you've written: Is it I/O-bound or CPU-bound? Would concurrency help?

Don't continue until you can explain the coffee shop analogy in your own words to someone else.