

chapter one

Sequential, Concurrent, Parallel

The words *sequential*, *concurrent*, and *parallel* answer different questions. Sequential work has no overlap. Concurrent work has multiple tasks in progress during an overlapping period. Parallel work executes more than one operation at the same instant. A program can be concurrent without being parallel, and it can use both.

We will develop these distinctions with a batch downloader. Given several URLs, the program fetches each response and performs a small amount of processing. Later versions will use threads, async tasks, queues, and processes, but the first version is an ordinary loop.

The task and a simulated network

To keep the first example repeatable and offline, `fetch` uses `time.sleep` in place of network I/O:

```
import time

def fetch(url):
    # Simulate time spent waiting for a reply.
    time.sleep(0.2)
    return f"<data from {url}>"
```

`sleep` suspends the calling thread for at least approximately the requested interval; it does not keep a CPU busy executing Python code. Real network requests diff-

sleep makes the calling thread ineligible to run until the interval has passed.

er in latency, can fail, and need timeouts. For now, the sleep isolates the scheduling idea we want to study.

The sequential baseline fetches one URL at a time:

```
def download_all(urls):
    results = []
    for url in urls:
        results.append(fetch(url))
    return results
```

We can measure its elapsed time with a monotonic performance counter:

```
urls = [f"https://example.com/{i}"
        for i in range(10)]

start = time.perf_counter()
download_all(urls)
elapsed = time.perf_counter() - start
print(f"{elapsed:.2f} s")
```

The arithmetic gives a target near 2.0 s; the clock reports what this run actually took.

Ten sleeps of roughly 0.2 seconds occur one after another, so the result should be near two seconds. Scheduler delays and measurement overhead mean that **2.00 s** is an expectation, not a guarantee.

Sequential: no overlap

In the sequential version, one call returns before the next begins:

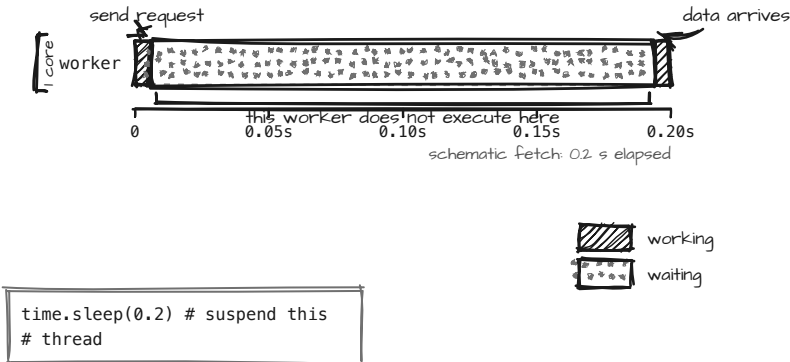
```
worker: [A=====] [B=====] [C=====]
```

“Worker” is a role in a diagram. A later chapter may implement that role with a thread, process, or event-loop task.

Here *worker* means the execution resource currently advancing the calls. In this program it is the main thread. The word will become more specific when we introduce pools and event loops.

Most of a simulated fetch is a wait rather than computation. A lane chart shows the small running intervals and

the longer interval during which the thread is asleep.



A schematic view of one fetch. Short running intervals surround the simulated wait; the 0.2-second ruler measures elapsed time, not CPU time.

The sequential loop cannot begin B during A's sleep because control has not returned from `fetch(A)`. That unused interval is an opportunity for another ready task, provided we restructure the program to allow overlap.

Concurrency: several tasks in progress

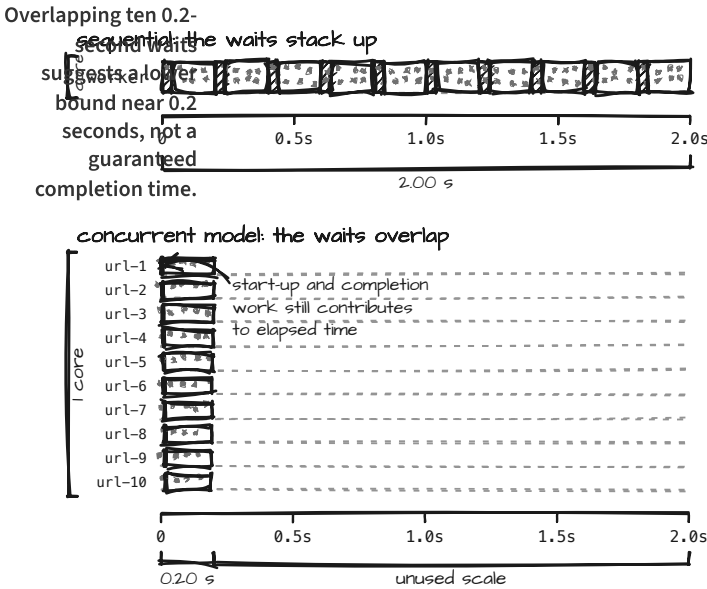
A concurrent design permits more than one task to be in progress during the same span of time. On a single execution resource, the tasks take turns: one runs, yields or blocks, and another runs. No two instructions need to execute simultaneously for this arrangement to be useful.

```
A: [run][----- wait -----][run]
B:   [run][----- wait -----][run]
C:       [run][----- wait -----][run]
```

An event loop can arrange this cooperative switching on one thread. A threaded program uses multiple threads, although on one core the operating system still schedules their execution in turns. In both cases, another task can advance while an earlier request waits.

Concurrency creates an opportunity to use otherwise idle intervals; it does not promise a speedup.

If ten independent waits begin close together, much of their elapsed time can overlap. The total may approach the longest wait plus the work needed to start and finish every request. It cannot be less than that work, and real services may impose connection limits, queueing, or rate limits.



Sequential waits occupy separate intervals; concurrent waits overlap. The drawing is a scheduling model, not a benchmark result.

This is why concurrency often helps I/O-heavy programs: it improves resource use while operations are stalled. It also introduces new obligations. Results may arrive out of order, failures may be partial, and the program needs limits, cancellation, and cleanup. We will add those mechanisms as the example becomes realistic.

Parallelism: simultaneous execution

Parallel execution means that two or more operations run at the same instant. For CPU work, that requires

multiple execution resources, such as cores. A scheduler might place three runnable workers on three cores:

```
core 1: [A=====]
core 2: [B=====]
core 3: [C=====]
```

Parallelism matters when there is computation to divide. Switching one core rapidly among three CPU-bound tasks does not create more processor time; it usually adds scheduling overhead. Multiple cores may reduce elapsed time if the work can be partitioned and the coordination cost is small enough.

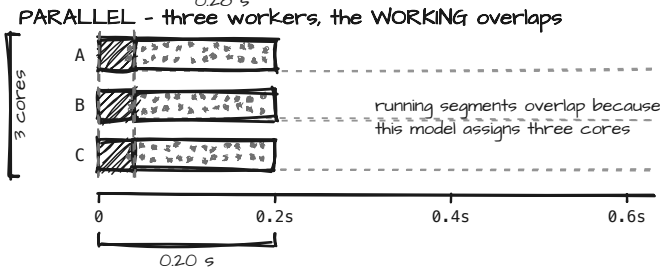
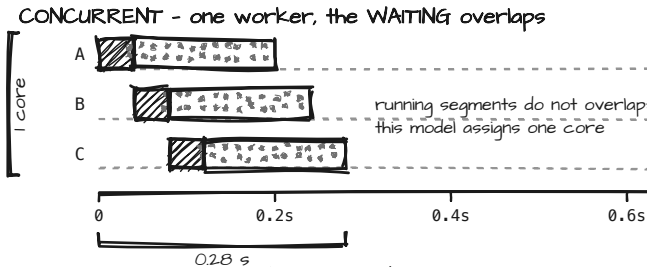
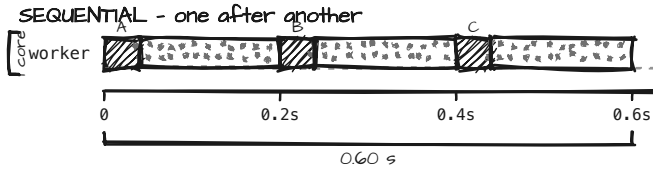
Parallel speedup requires both spare execution resources and work that can be divided.

In Python, the available routes depend on the runtime and the code. Separate processes can execute Python work on multiple cores. Native libraries often run parallel code outside the interpreter. Free-threaded CPython builds can execute Python threads in parallel, while conventional GIL-enabled CPython builds generally cannot execute Python bytecode in parallel across threads. “Use processes” is therefore a common answer, not a definition of parallelism.

Three arrangements on one ruler

The next figure keeps the tasks and time scale fixed. Only their placement changes.

Compare diagrams only after checking that their rulers and task costs match.

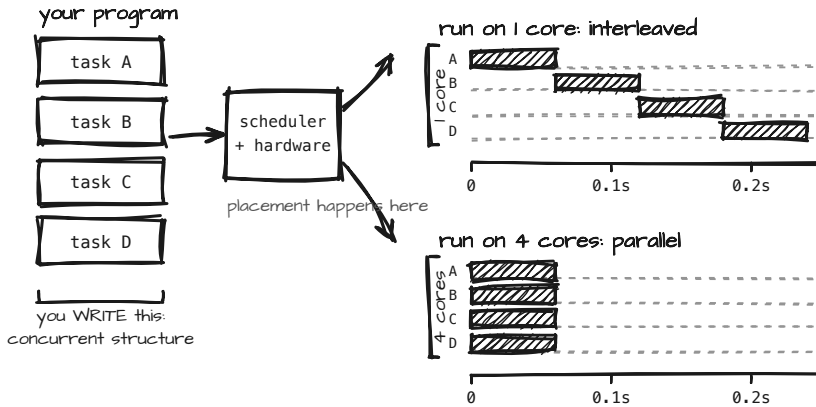


The same three schematic tasks arranged sequentially, concurrently on one core, and in parallel on three cores. Running texture overlaps only in the parallel panel.

The concurrent panel overlaps waits but not running segments. The parallel panel overlaps running segments because the bracket assigns separate cores. Real schedules are less tidy, but the distinction remains: overlapping lifetimes establish concurrency; overlapping execution establishes parallelism.

Tasks expose scheduling choices; they do not choose their own core placement.

Concurrency is partly a property of program structure. Independent tasks give a scheduler choices that a chain of strict dependencies does not. Actual placement, however, depends on the runtime, operating system, hardware, and current load. Source code alone does not guarantee that two eligible tasks will run simultaneously.



Four independent tasks can be interleaved on one core or placed across several cores. The runtime and system determine the actual schedule.

The categories can combine:

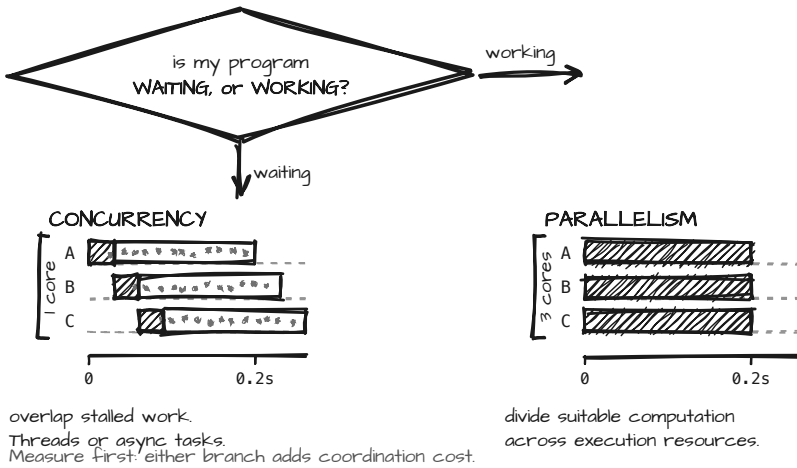
- An async server may manage thousands of concurrent connections on one event-loop thread without parallel Python execution.
- A process pool may run several independent computations concurrently and in parallel.
- A numeric routine may parallelize one operation internally even when the calling application presents it as a single task.

For that reason, “concurrent or parallel?” is not always an either/or question. One describes overlapping progress; the other describes simultaneous execution.

Start with the bottleneck

The simulated downloader spends most of its elapsed time waiting. Giving it a way to begin other requests during that wait is promising. A busy numeric loop presents the opposite case: if it has no waits to overlap, useful speedup requires parallel execution or a faster algorithm or implementation.

“Waiting or working?” is a first diagnosis, not the whole design decision.



A first branch in tool selection: overlap stalled I/O with concurrency, or divide suitable computation across execution resources. Measurement comes before the choice.

Next, measure CPU
use and waiting time
before choosing a
concurrency
mechanism.

The diagnosis does not select an API by itself. Threads and async tasks have different integration costs. Processes must exchange data. Every approach can be limited by memory, external capacity, dependencies, or serial sections. Chapter 2 shows how to determine where the time goes before changing the design.

Exercises

Exercise 1.1. Classify each situation as concurrent, parallel, or both, and state what evidence would settle the answer: (a) one cook tends five kettles as they become ready; (b) three cooks work at three stoves; (c) several `asyncio` tasks run on one event-loop thread; (d) a numeric library divides a matrix multiplication across four cores.

Exercise 1.2. Draw lane charts for three simulated fetches in these arrangements: sequential; concurrent on one execution resource, with the waits overlapping; and parallel on three resources. Give each fetch a 0.2-second

wait, use one ruler for all three drawings, and label any additional running time you assume. Which totals can you infer from the model, and which require measurement?

Exercise 1.3. A colleague sees that `download_all` is slow and proposes using all eight CPU cores. What does the current example spend most of its time doing? Explain why extra cores are not the first resource it needs, then name one limit you would add before issuing real requests concurrently.

Exercise 1.4. Replace `time.sleep(0.2)` conceptually with 0.2 seconds of uninterrupted computation. Would interleaving those calls on one core reduce the total CPU work or elapsed time? What additional condition would allow them to execute in parallel?