Python Concurrency Research for Async, GIL, Multiprocessing & Threading

by Sertaç Ataç & Ramazan Bağış

25.06.2025

Introduction

The execution of multiple tasks within a software system can be approached through two fundamental concepts: concurrency and parallelism. While they share similarities, these terms describe distinct operational models.

Concurrency refers to the ability of a system to handle **multiple tasks at the same time**. This is achieved by interleaving the execution of tasks **on a single processing unit**. In concurrency, multiple tasks are managed over a period, allowing progress on several fronts without necessarily executing them simultaneously.

Parallelism denotes the actual **simultaneous execution of multiple tasks**. This typically **requires multiple processing units**, such as CPU cores, where different parts of a program can run at the exact same moment.

The relationship between these two concepts is hierarchical: every parallel program inherently exhibits concurrency, as it manages multiple tasks, but not every concurrent program achieves true parallel execution. A program can be concurrent without being parallel, such as when tasks are rapidly switched on a single core, giving the illusion of simultaneous progress.

Python's journey in addressing concurrency and parallelism is shaped by a unique design choice: the Global Interpreter Lock (GIL). This mechanism introduces specific challenges and necessitates particular solutions for achieving efficient concurrent and parallel execution within the standard CPython interpreter.

Table of Contents:

---

1. Global Interpreter Lock (GIL)

The Global Interpreter Lock is a mutex in CPython that **allows only one thread to execute Python bytecode at a time**. Internally, each OS-level thread must acquire the GIL before running any Python code; if another thread holds it, the new thread must wait. The GIL was introduced to simplify memory management and make threading easier to implement, but it means Python threads cannot truly run Python code in parallel in one process.

To prevent one thread from starving others, CPython periodically **releases the GIL**. In Python 3, by default a thread will hold the GIL for up to about **5 milliseconds** before releasing it if it hasn't already done so via blocking I/O.

Because of the GIL, **CPU-bound Python code cannot run in parallel** on multiple cores. Even if multiple threads start doing heavy computation, only one will execute bytecode at any moment; others just wait their turn. In fact, multi-threading can *worsen* CPU-bound throughput due to GIL contention and extra overhead. New developments aim to mitigate this, but **legacy CPython** still enforces a single active bytecode thread.

2. Threading

Threading is a **concurrency model** that enables the **execution of multiple threads** (lightweight processes) **within a single process**. Each thread runs in the same memory space and shares the same data, which makes communication between threads easier and more efficient in terms of memory usage.

Python's threads are constrained by the Global Interpreter Lock (GIL). Due to the GIL, **only one thread can execute Python bytecode at a time in a single process**. This means that Python threads are not suitable for CPU-bound tasks, as the GIL effectively serializes execution even across multiple threads.

However, for I/O-bound tasks (such as network communication or file access), Python threading can offer significant performance benefits. During **blocking I/O operations ,the GIL is released, allowing other threads to run.** This overlapping of I/O wait times enables Python to make progress on multiple tasks concurrently, even with the GIL in place.

Threading is implemented in Python via the `threading` module. It provides tools to create, manage, and synchronize threads, including locks, events, and queues. Careful synchronization is essential to avoid race conditions and deadlocks, especially when threads share mutable data.

Typical use cases for threading include network clients, web scraping, GUI applications, and any situation where tasks spend significant time waiting for external resources.

3. Multiprocessing

Multiprocessing is a **parallelism model** that bypasses the limitations of the GIL by using multiple processes instead of threads. **Each process runs in its own Python interpreter and has its own memory space**, which allows true concurrent execution of Python code across multiple CPU cores. This model is well-suited for CPU-bound tasks, where parallel execution can significantly reduce processing time. By dividing work among several processes, Python programs can leverage all available cores in a machine

Since processes do not share memory by default, **inter-process communication (IPC) is handled through queues, pipes, and shared memory constructs**. These mechanisms involve serialization (pickling) of objects, which introduces overhead. As a result, multiprocessing is generally more resource-intensive than threading, both in terms of memory and startup time.

The `multiprocessing` module provides an API similar to `threading`, with classes like `Process`, `Pool`, and `Queue`. The `Pool` class is especially useful for parallelizing map-reduce style workloads. Multiprocessing is ideal for tasks such as data processing, numerical simulations, image or video processing, and other CPU-heavy operations.

4. Asynchronous Programming

Asynchronous programming in Python is designed for **managing I/O-bound operations in a single-threaded, single-process environment** by the `asyncio` library. Instead of creating threads or processes, asyncio runs an **event loop that schedules and executes tasks cooperatively.**

The core components of asyncio are coroutines, defined with `async def`, and the `await` keyword, which allows the suspension of a coroutine **until a non-blocking operation completes**. When a coroutine awaits, it yields control back to the event loop, which can then run other ready tasks.

This model **eliminates the need for context switching** at the operating system level, resulting in lower overhead compared to threads or processes. However, it **requires that all I/O operations be non-blocking and awaitable**. Traditional blocking libraries (e.g., `requests`) must be replaced with their asynchronous counterparts (e.g., `aiohttp`).

Asynchronous programming is **best suited for applications that handle many concurrent I/O operations, such as web servers, network clients, chat applications, and APIs**. While powerful, it demands a different programming paradigm and careful structuring of code to avoid blocking the event loop.

5. Comparison of Models

Below is a high-level comparison of **asyncio**, **threading**, and **multiprocessing** on key dimensions made by AI tools:

Feature / Dimension	Threading	Multiprocessing	Async (asyncio)
Concurrency model	OS threads (preemptive), shares memory	OS processes (parallel), separate memory	Single-threaded event loop (cooperative)
GIL effect	One thread holds GIL at a time	Bypasses GIL (each process has its own)	Uses one thread anyway (not affected by GIL issues for I/O)
Suitable for	I/O-bound tasks (network, file I/O)	CPU-bound tasks (compute-heavy)	I/O-bound tasks, high-concurrency services
CPU-bound tasks	No real speedup (GIL limits parallelism)	True parallel speedup across cores	Inefficient (runs on one core)
I/O-bound tasks	Good (GIL released during I/O)	Good (each I/O can be done by separate process, but overhead)	Excellent (built for async I/O, non-blocking)
Overhead	Low per-thread overhead; context-switch cost of GIL lock	High overhead (process creation, memory, IPC)	Low overhead (function call switches, no OS thread)
Memory usage	Threads share memory (low overhead)	Separate memory per process (higher)	Very low per task (just coroutine state)
Data sharing	Shared memory (easy access, but need locks)	No shared memory by default (need IPC or shared memory constructs)	Shared state only within the loop (no data sharing between loops)

Library compatibility	Works with most libraries; if library releases GIL, threads can compute in parallel	Works with most libraries; data must be picklable for IPC	Requires async-compatible libraries (e.g. aiohttp instead of requests)
Scalability (cores)	Limited by 1 core for Python code (threads serialized by GIL)	Scales to all cores (parallel processes)	Single core (unless combined with processes)
Scalability (machines)	Limited to one machine (within process); can use network+threads separately	Limited to one machine; inter-machine requires extra frameworks	Limited to one process (one machine); can spawn processes for multi-machine
Ease of implementation	Easy to use (threading.Thread), similar to sequential code with locking	More complex (multiprocessing API)	Harder style (async/await, must avoid blocking calls)
Debugging & maintainability	Risk of race conditions/deadlocks; stack traces show thread names but concurrency bugs are subtle	Easier to isolate (each process), but debugging multiple processes can be cumbersome; serialization errors	Async stack traces can be less straightforward (async frames), but logic is sequential; require careful handling of awaitables
Best practice scenarios	Use when tasks spend time waiting (I/O); e.g. network clients, GUI apps, background jobs	Use when tasks are CPU-bound or truly parallel; e.g. heavy computation, numeric data processing	Use when handling many concurrent I/O operations; e.g. web servers, async clients, IO-bound pipelines

6. References & Used Tools

AI Tools: DeepSeek AI, Gemini AI, ChatGPT AI

References:

docs.python.org -> https://docs.python.org/uk/3.13/library/threading.html , https://docs.python.org/3/library/multiprocessing.html

stackoverflow.com -> https://stackoverflow.com/questions/54256847/why-does-python-switch-threads , https://stackoverflow.com/questions/27435284/multiprocessing-vs-multithreading-vs-asyncio

analyticsvidhya.com -> https://www.analyticsvidhya.com/blog/2024/02/python-global-interpreter-lock/#:~:text=How%20Does%20the%20GIL%20Work%3F

medium.com -> https://medium.com/dev-jam/unlocking-pythons-multi-core-potential-5c30ed651f58 ,

https://medium.com/dev-jam/unlocking-pythons-multi-core-potential-5c30ed651f58

kdnuggets.com -> https://www.kdnuggets.com/introduction-to-multithreading-and-multiprocessing-in-python#:~:text=Multiprocessing%20is%20more%20computationally%20expensive,overcomes%20Global%20Interpreter%20Lock's%20limitations.

statusneo.com -> https://statusneo.com/concurrency-in-python-threading-processes-and-asyncio/

ionos.com -> https://www.ionos.com/digitalguide/websites/web-development/python-multiprocessing/

codefinity.com -> https://codefinity.com/blog/Asynchronous-Programming-in-Python

pythonspeed.com -> https://pythonspeed.com/articles/faster-multiprocessing-pickle/

news.ycombinator.com **->** https://news.ycombinator.com/item?id=33547323

wikipedia.org -> https://en.wikipedia.org/wiki/Global_interpreter_lock

lambdatest.com -> https://www.lambdatest.com/blog/python-asyncio/

realpython.com **->** https://realpython.com/async-io-python/