# **Deep Research on Concurrency and Parallelism in Python using ThreadPoolExecutor and ProcessPoolExecutor**

## **1. Introduction to Concurrency and Parallelism in Python**

Understanding the fundamental concepts of concurrency and parallelism is essential for optimizing Python applications. While often used interchangeably, these terms describe distinct approaches to managing and executing multiple tasks, each with specific implications for performance and resource utilization. Python's unique architecture, particularly the Global Interpreter Lock (GIL), further shapes how these paradigms are applied.

### **1.1 Defining Concurrency vs. Parallelism**

Concurrency refers to the ability to manage multiple tasks that appear to be executing simultaneously. This is achieved through rapid context switching, where a single CPU core quickly switches between different tasks, giving the illusion of multitasking. Threads, which are lightweight execution units capable of sharing memory within a single process, are a primary mechanism for enabling this efficient context switching in Python.1 For instance, a program might manage multiple web requests concurrently; while one request waits for a response from a server, the CPU switches to process another request, making progress on several tasks over overlapping time periods.1

In contrast, parallelism involves the true simultaneous execution of multiple tasks across multiple CPU cores or processors. This is about literally doing multiple instruction sequences at the same time, leveraging the hardware capabilities of modern multi-core systems.1 For example, if a machine has four CPU cores, a parallel program could execute four computationally intensive tasks at the exact same instant, one on each core.

Python's approach to these concepts is distinct. Concurrency is primarily achieved through threading (as seen with ThreadPoolExecutor) and asynchronous programming (using libraries like asyncio). True parallelism, particularly for tasks that heavily utilize the CPU, is typically realized through multiprocessing (as provided by ProcessPoolExecutor).2 The distinction between the

*impression* of simultaneous execution (concurrency) and *actual* simultaneous execution (parallelism) is a critical consideration for Python developers. A misunderstanding here can lead to incorrect assumptions about performance gains, especially when dealing with CPU-bound workloads. Expecting ThreadPoolExecutor to linearly speed up CPU-intensive tasks with more cores, for example, would be a misapplication of the tool, resulting in inefficient design choices and performance disappointment. The core principle is that for tasks involving waiting (e.g., I/O operations), the "illusion" of concurrency provided by threads is highly beneficial, as the CPU can switch to other tasks during waiting periods. However, for tasks that demand constant CPU computation, this "illusion" becomes a significant performance bottleneck.

### **1.2 The Python Global Interpreter Lock (GIL): A Fundamental Concept**

At the heart of Python's concurrency model, particularly for the standard CPython implementation, lies the Global Interpreter Lock (GIL). The GIL is a mutex that protects access to Python objects, preventing multiple threads from executing Python bytecodes simultaneously.4 Its primary purpose is to simplify CPython's memory management and ensure thread safety by avoiding race conditions and other complex concurrency issues that would arise if multiple threads could freely modify Python objects at the same time.4

The presence of the GIL means that even in a multi-threaded Python program running on a multi-core processor, only one thread can execute Python bytecode at any given moment. This serialization of Python code execution can lead to significant performance bottlenecks in applications that are "CPU-bound"—meaning their performance is limited by the speed of the CPU rather than by waiting on external resources. In such scenarios, the interpreter may spend considerable time context switching between threads, introducing overhead without achieving true parallel execution.4

However, the GIL is not always a limitation. It is released during certain operations, most notably during I/O operations (such as network requests, file reading/writing, or database queries) or when Python calls into C extensions (like those found in popular libraries such as NumPy, SciPy, and Pandas) that are designed to explicitly release the GIL.3 This mechanism allows other Python threads to run while one thread is waiting for an external operation to complete, making

ThreadPoolExecutor effective for I/O-bound tasks.

The GIL's existence is a deliberate design choice, prioritizing ease of implementation and stability in CPython's memory management over raw multi-threaded CPU parallelism.4 This architectural decision means that Python inherently sacrifices true multi-threaded parallelism for CPU-bound tasks to gain simpler memory management and robust thread safety. This philosophical underpinning directly dictates the optimal use cases for

ThreadPoolExecutor (which benefits from GIL release during I/O) and ProcessPoolExecutor (which bypasses the GIL entirely by using separate processes). Understanding this foundational design decision is more beneficial than merely viewing the GIL as a "limitation"; it provides a framework for making informed architectural decisions in Python development. It also clarifies why alternative Python implementations, such as Jython and IronPython, do not feature a GIL, as they employ different memory management strategies.4

### **1.3 Overview of the concurrent.futures Module**

The concurrent.futures module, introduced in Python 3.2, offers a high-level interface for asynchronously executing callable tasks. It abstracts away the complexities of directly managing threads or processes, providing a streamlined way to launch and monitor concurrent operations.10 This module introduces the concept of

Executor objects, which serve as a common interface for both thread-based and process-based concurrency.

Both ThreadPoolExecutor and ProcessPoolExecutor are subclasses of the abstract Executor class. This shared interface is a significant advantage, as it allows developers to switch between thread-based and process-based execution with minimal code changes, facilitating experimentation and optimization based on workload characteristics.11

A common and highly recommended practice when using Executor objects is to employ them within a with statement. This context manager automatically handles the proper shutdown and cleanup of resources associated with the executor. When the with block exits, the shutdown() method is implicitly called, ensuring that all pending futures are completed and resources are released gracefully, preventing potential resource leaks or hangs.11 This automatic management simplifies resource hygiene and makes concurrent code more robust.

## **2. ThreadPoolExecutor: Concurrency for I/O-Bound Tasks**

ThreadPoolExecutor is a powerful tool within Python's concurrent.futures module designed to manage and execute tasks concurrently using a pool of threads. It is particularly well-suited for workloads where tasks spend a significant amount of time waiting for external operations to complete, rather than performing intensive computations.

### **2.1 Architecture and Operational Mechanics**

ThreadPoolExecutor operates by maintaining a pool of worker threads. When a task is submitted, it is placed in a queue, and an available thread from the pool picks it up for execution. A key advantage of using a thread pool is the reduction in overhead associated with creating and destroying threads for each individual task. Instead, threads are created once and then reused for subsequent tasks, leading to more efficient resource utilization.11

Tasks can be submitted to the pool using methods such as submit() for individual tasks or map() for applying a function to an iterable of arguments.11 The

submit() method returns a Future object immediately, which represents the eventual result of the task. The map() method, similar to Python's built-in map(), applies a function to each item in an iterable and returns an iterator that yields results in the order of the inputs, regardless of their completion order.11

The default number of workers (threads) in a ThreadPoolExecutor has evolved across Python versions, reflecting a refined understanding of optimal thread pool sizing. In Python 3.5, the default for max\_workers was set to os.cpu\_count() \* 5, based on the assumption that ThreadPoolExecutor is frequently used to overlap I/O operations rather than CPU work.11 This higher number of threads compared to CPU cores aimed to keep the CPU busy by switching to another thread whenever one was blocked on I/O. As of Python 3.13, the default

max\_workers has been adjusted to min(32, (os.process\_cpu\_count() or 1) + 4).11 This change indicates a more nuanced understanding: while I/O-bound tasks benefit from more threads than CPU cores, there is a practical upper limit beyond which the overhead of context switching or increased memory consumption might outweigh the benefits. This refined default suggests that for typical I/O workloads, 32 concurrent operations or a slight multiple of CPU cores is often sufficient, and exceeding this can introduce diminishing returns.

The lifecycle of threads within the pool is managed by the executor. Threads are conceptually created as soon as a Future object is associated with them (i.e., when a task is submitted) and begin execution once they are scheduled by the executor. Upon completion of their assigned task, threads are not immediately destroyed but are returned to the pool to await new tasks, thus prolonging their lifecycle and minimizing creation overhead.19

### **2.2 Key Parameters and Configuration**

Effective use of ThreadPoolExecutor involves understanding and configuring its key parameters:

* **max\_workers**: This argument controls the maximum number of threads that the pool will use simultaneously.20 Carefully setting this value is crucial to prevent system overload. While I/O-bound tasks can benefit from a higher number of threads than available CPU cores (e.g., a common guideline for I/O-intensive tasks is  
  2N where N is the number of CPU cores), excessively high thread counts can introduce overhead from context switching and increased memory usage, diminishing performance gains.24
* **thread\_name\_prefix**: (Introduced in Python 3.6) This parameter allows developers to specify a prefix for the names of worker threads created by the pool. This feature is particularly useful for debugging, as it makes it easier to identify and track threads belonging to a specific ThreadPoolExecutor instance in logs or profiling tools.11
* **initializer and initargs**: (Introduced in Python 3.7) These optional arguments allow a callable (initializer) to be executed at the start of each worker thread, with initargs providing a tuple of arguments for this callable. This is beneficial for setting up thread-local resources, such as database connections or session objects, ensuring that each thread has its own isolated setup without needing to pass these resources with every task.11

### **2.3 Practical Use Cases and Examples (I/O-Bound)**

ThreadPoolExecutor is ideally suited for tasks that are I/O-bound, meaning they spend most of their execution time waiting for input/output operations (e.g., network responses, disk reads/writes) rather than performing intensive computations. During these waiting periods, the GIL is released, allowing other threads to execute Python bytecode, thereby maximizing CPU utilization and overall throughput.

* **Web Scraping and API Calls:** This is a quintessential application for ThreadPoolExecutor. When fetching data from multiple URLs or making numerous API requests, threads can initiate requests concurrently. While one thread waits for a server response, another can send out a new request, significantly reducing the total time required. Real-world examples demonstrate substantial speedups, such as an 88% improvement in downloading Hacker News stories with 30 threads, or an 800% performance boost for processing Airbnb listings by using 20 concurrent threads.1  
  **Example: Concurrent Web Page Fetching**  
  Python  
  import concurrent.futures  
  import urllib.request  
  import time  
    
  URLS =  
    
  def load\_url(url, timeout):  
   """Loads a URL and returns its content or raises an exception."""  
   try:  
   with urllib.request.urlopen(url, timeout=timeout) as conn:  
   return conn.read()  
   except Exception as e:  
   raise Exception(f"Error fetching {url}: {e}")  
    
  if \_\_name\_\_ == "\_\_main\_\_":  
   start\_time = time.perf\_counter()  
   print("Starting concurrent web page fetching...")  
   with concurrent.futures.ThreadPoolExecutor(max\_workers=5) as executor:  
   # Map each future to its URL for easy tracking of results  
   future\_to\_url = {executor.submit(load\_url, url, 10): url for url in URLS}  
   for future in concurrent.futures.as\_completed(future\_to\_url):  
   url = future\_to\_url[future]  
   try:  
   data = future.result()  
   print(f"'{url}' page is {len(data)} bytes")  
   except Exception as exc:  
   print(f"'{url}' generated an exception: {exc}")  
   end\_time = time.perf\_counter()  
   print(f"\nTotal time taken: {end\_time - start\_time:.2f} seconds")
* **Parallel File Operations:** Reading from or writing to multiple files concurrently, especially large datasets, can also benefit from ThreadPoolExecutor. File I/O operations are inherently I/O-bound, allowing threads to release the GIL while waiting for disk access, enabling other threads to proceed.20  
  **Example: Reading Multiple CSVs**  
  Python  
  import pandas as pd  
  from concurrent.futures import ThreadPoolExecutor  
  from pathlib import Path  
  import time  
  import os  
    
  # Create dummy CSV files for demonstration  
  def create\_dummy\_csv(filename, num\_rows):  
   df = pd.DataFrame({'col1': range(num\_rows), 'col2': [f'data\_{i}' for i in range(num\_rows)]})  
   df.to\_csv(filename, index=False)  
    
  if \_\_name\_\_ == "\_\_main\_\_":  
   data\_dir = Path("temp\_csv\_data")  
   data\_dir.mkdir(exist\_ok=True)  
    
   num\_files = 10  
   rows\_per\_file = 10000  
   csv\_files =  
   for i in range(num\_files):  
   file\_path = data\_dir / f"data\_{i}.csv"  
   create\_dummy\_csv(file\_path, rows\_per\_file)  
   csv\_files.append(file\_path)  
    
   print(f"Created {num\_files} dummy CSV files in '{data\_dir}'")  
    
   # Sequential reading  
   start\_time\_seq = time.perf\_counter()  
   dfs\_seq =  
   for file in csv\_files:  
   dfs\_seq.append(pd.read\_csv(file))  
   df\_combined\_seq = pd.concat(dfs\_seq)  
   end\_time\_seq = time.perf\_counter()  
   print(f"\nSequential reading took: {end\_time\_seq - start\_time\_seq:.4f} seconds")  
    
   # Parallel reading with ThreadPoolExecutor  
   start\_time\_thread = time.perf\_counter()  
   with ThreadPoolExecutor(max\_workers=os.cpu\_count() \* 2) as executor: # Use a multiple of CPU count for I/O  
   # map returns results in the order of inputs  
   dfs\_thread = list(executor.map(pd.read\_csv, csv\_files))  
   df\_combined\_thread = pd.concat(dfs\_thread)  
   end\_time\_thread = time.perf\_counter()  
   print(f"ThreadPoolExecutor reading took: {end\_time\_thread - start\_time\_thread:.4f} seconds")  
    
   # Clean up dummy files  
   for file in csv\_files:  
   file.unlink()  
   data\_dir.rmdir()

### **2.4 Impact of the GIL on ThreadPoolExecutor Performance**

The Global Interpreter Lock (GIL) fundamentally shapes the performance characteristics of ThreadPoolExecutor.

For **CPU-bound tasks**, the GIL acts as a significant bottleneck. Since only one Python thread can execute bytecode at a time, ThreadPoolExecutor cannot achieve true parallelism for such tasks, even on multi-core systems. Instead, threads are effectively serialized, and any performance gains are minimal or non-existent. In some cases, the overhead of context switching between threads can even lead to a slight performance degradation compared to sequential execution.4

Conversely, for **I/O-bound tasks**, ThreadPoolExecutor is highly effective. The GIL is released during I/O operations (e.g., waiting for network responses or disk reads). This allows other threads to acquire the GIL and execute their Python bytecode while the first thread is blocked. This mechanism enables efficient concurrent execution, maximizing CPU utilization during waiting periods and leading to substantial performance improvements.6

A subtle but important point arises when considering very short-lived CPU-bound tasks. While ThreadPoolExecutor is generally ill-suited for CPU-bound workloads, there are scenarios where, counter-intuitively, it might offer a slight advantage over ProcessPoolExecutor. This occurs when the computational work per task is extremely minimal. Process creation and the overhead of inter-process communication (IPC) are significantly higher than thread creation and context switching.14 If a CPU-bound task is so brief that the overhead of spawning a new process for it (or managing a process pool) exceeds the time saved by true parallelism, then the lower overhead of threads and their shared memory might make

ThreadPoolExecutor a marginally better choice, even with the GIL. This suggests that the general guideline of "CPU-bound tasks require ProcessPoolExecutor" is not an absolute rule; task granularity and the associated overheads must also be carefully considered.

### **2.5 Best Practices and Common Pitfalls**

To leverage ThreadPoolExecutor effectively and avoid common issues, several best practices should be followed:

* **Limit Thread Count:** It is crucial to set max\_workers carefully. While I/O-bound tasks can benefit from having more threads than CPU cores (e.g., a common heuristic is 2 \* N where N is the number of CPU cores 26), there is a point of diminishing returns. If the number of threads becomes excessively high, the system may spend more time on context switching between threads and managing increased memory usage than on productive work, leading to performance degradation.24 The "2N" rule is a general guideline, and empirical tuning based on the specific workload is often necessary to find the optimal balance.
* **Handle Exceptions Gracefully:** Tasks submitted to the executor might raise exceptions. It is essential to catch these exceptions within the submitted functions and retrieve them from the Future objects (e.g., using future.result() within a try-except block) to prevent silent failures or issues that could affect the entire pool.13
* **Manage Shared Resources:** Threads within a ThreadPoolExecutor share the same memory space. This shared memory facilitates easy data access but also introduces the risk of race conditions if multiple threads attempt to modify shared mutable data concurrently. Proper synchronization primitives, such as threading.Lock, threading.RLock, or thread-safe data structures like queue.Queue, must be used to protect shared resources and ensure data integrity.6
* **Avoid Long-Running Tasks within Threads:** ThreadPoolExecutor is generally not recommended for tasks that are extremely long-running or block indefinitely. Such tasks can delay the interpreter's exit and complicate graceful shutdown procedures, as all enqueued threads must be joined before the interpreter can exit.11
* **Beware of Deadlocks:** Deadlocks can occur if tasks submitted to the pool wait on the results of other tasks that are also within the *same* ThreadPoolExecutor and cannot proceed. This creates a circular dependency where no task can complete, causing the pool to freeze indefinitely.11 Careful design is needed to avoid such inter-task dependencies within a single pool.

## **3. ProcessPoolExecutor: True Parallelism for CPU-Bound Tasks**

ProcessPoolExecutor is another key component of the concurrent.futures module, specifically designed to achieve true parallelism for CPU-bound tasks in Python. Unlike ThreadPoolExecutor, it leverages separate processes, each with its own Python interpreter and memory space, thereby circumventing the limitations imposed by the Global Interpreter Lock (GIL).

### **3.1 Architecture and Operational Mechanics**

ProcessPoolExecutor manages a pool of worker processes. When tasks are submitted, they are distributed among these processes for execution. The fundamental advantage of this architecture is that each process operates independently, possessing its own Python interpreter and its own GIL. This isolation allows multiple CPU-bound tasks to execute simultaneously on different CPU cores, achieving true parallelism that is not possible with threads in CPython.3

The ProcessPoolExecutor is built on top of Python's multiprocessing module, inheriting its capabilities and some of its characteristics.11 A critical implication of using separate processes is that data and functions passed to or returned from worker processes must be "picklable." Pickling is the process of serializing Python objects into a byte stream, which is necessary for inter-process communication (IPC). If an object or function cannot be pickled, it cannot be transferred to or from a worker process, leading to errors.11

Furthermore, for worker subprocesses to execute functions, the \_\_main\_\_ module (the script being run) must be importable by these worker processes. This means that ProcessPoolExecutor typically will not work correctly in an interactive interpreter session (like a standard Python REPL or Jupyter Notebook) unless specific workarounds are employed. Functions intended for execution in the pool should generally be defined at the top level of a module, often guarded by an if \_\_name\_\_ == '\_\_main\_\_': block, to ensure they are properly imported by the child processes.11 This requirement for picklability and module importability constitutes an implicit architectural contract. It imposes a stricter discipline on the codebase compared to

ThreadPoolExecutor, as complex objects, custom classes, or closures must be designed with serialization in mind. Functions must be accessible and correctly imported by the new process environments.

Python's multiprocessing module, and by extension ProcessPoolExecutor, supports different "start methods" for creating new processes: fork, spawn, and forkserver. The spawn method is the default on Windows and macOS and is often recommended for robustness across platforms. It starts a fresh Python interpreter process for the child, inheriting only necessary resources. The fork method (default on POSIX systems) is faster as it duplicates the parent process, but it can lead to issues if the parent process has threads holding locks or has imported modules in a way that causes a corrupted state in the child.11 The

forkserver method creates a dedicated server process that forks new workers on demand, offering a balance between fork's speed and spawn's robustness.

### **3.2 Key Parameters and Configuration**

Optimizing ProcessPoolExecutor performance involves careful configuration of its parameters:

* **max\_workers**: This parameter specifies the maximum number of worker processes in the pool. By default, it often aligns with the number of CPU cores available on the machine (os.process\_cpu\_count()) 11, or  
  min(32, (os.process\_cpu\_count() or 1) + 4) in Python 3.13.11 For CPU-bound tasks, setting  
  max\_workers close to the number of physical CPU cores is generally optimal. Exceeding this number can lead to CPU oversubscription, where processes compete for limited CPU time, resulting in frequent context switching overhead and reduced overall efficiency.34
* **chunksize**: When using the map() method with ProcessPoolExecutor, the chunksize argument can significantly improve performance for very long iterables. Instead of submitting individual tasks one by one, map() can group items from the iterable into larger "chunks" and submit each chunk as a single task to a worker process. This reduces the overhead of inter-process communication and task management, especially when the individual tasks are relatively small.11
* **mp\_context**: This parameter allows explicit control over the process start method (fork, spawn, or forkserver) used for worker processes. This can be important for ensuring compatibility and robustness across different operating systems and specific application requirements.11
* **max\_tasks\_per\_child**: (Introduced in Python 3.11) This optional argument specifies the maximum number of tasks a single worker process can execute before it is terminated and replaced with a fresh worker process. This parameter is a crucial mitigation strategy for potential memory leaks that might occur within long-running worker processes. Even though processes have isolated memory, a worker process might accumulate memory over many tasks (e.g., due to C extensions not releasing memory properly, or complex objects not being garbage collected within the worker's scope). If a worker's memory footprint grows indefinitely, it can lead to system instability or BrokenProcessPool errors.11 By periodically restarting workers,  
  max\_tasks\_per\_child helps maintain stability and prevent resource exhaustion, even if the root cause of a subtle memory leak is not fully identified or fixable in the application code.

### **3.3 Practical Use Cases and Examples (CPU-Bound)**

ProcessPoolExecutor is the preferred choice for tasks that are CPU-bound, meaning they involve intensive computations and benefit directly from being executed in parallel across multiple CPU cores.

* **Intensive Mathematical Computations:** Tasks requiring heavy number crunching, such as generating prime numbers, performing complex simulations, data encryption/decryption, or scientific calculations, are ideal candidates. By distributing these computations across multiple processes, the total execution time can be significantly reduced, leveraging the full power of multi-core processors.13  
  **Example: Prime Number Check**  
  Python  
  import concurrent.futures  
  import math  
  import time  
  import os  
    
  PRIMES\_TO\_CHECK = [  
   112272535095293,  
   112582705942171,  
   112272535095293, # Duplicate for demonstration  
   115280095190773,  
   115797848077099,  
   1099726899285419,  
   # A very large number to simulate a long computation  
   1000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000

#### Works cited

1. Concurrency, Parallelism, and Asynchronous Programming in Python, accessed June 23, 2025, <https://zenduty.com/blog/concurrency-and-parallelism-in-python/>
2. Concurrency vs parallelism: the main differences - Oxylabs, accessed June 23, 2025, <https://oxylabs.io/blog/concurrency-vs-parallelism>
3. Python Parallelization - Threads vs. Processes – bytewax, accessed June 23, 2025, <https://bytewax.io/blog/threads-vs-processes>
4. Understanding Python's Global Interpreter Lock (GIL) | Codementor, accessed June 23, 2025, <https://www.codementor.io/@richardbailey85/understanding-python-s-global-interpreter-lock-gil-2qobxrmg8m>
5. www.codementor.io, accessed June 23, 2025, <https://www.codementor.io/@richardbailey85/understanding-python-s-global-interpreter-lock-gil-2qobxrmg8m#:~:text=%22In%20CPython%2C%20the%20global%20interpreter,conditions%20and%20ensures%20thread%20safety.%22>
6. Python Thread Safety: Using a Lock and Other Techniques, accessed June 23, 2025, <https://realpython.com/python-thread-lock/>
7. Which is faster, Python threads or processes? Some insightful examples, accessed June 23, 2025, <https://towardsdatascience.com/which-is-faster-python-threads-or-processes-some-insightful-examples-26074e90848f/>
8. An Intro to Threading in Python – Real Python, accessed June 23, 2025, <https://realpython.com/intro-to-python-threading/>
9. Implementing Multi-threading for Faster Web Scraping, accessed June 23, 2025, <https://web.instantapi.ai/blog/implementing-multi-threading-for-faster-web-scraping/>
10. threading — Thread-based parallelism — Python 3.13.5 documentation, accessed June 23, 2025, <https://docs.python.org/uk/3.13/library/threading.html>
11. concurrent.futures — Launching parallel tasks — Python 3.13.5 documentation, accessed June 23, 2025, <https://docs.python.org/3/library/concurrent.futures.html>
12. Multiprocessing — Python notebooks, accessed June 23, 2025, <https://pnavaro.github.io/python-notebooks/10-Multiprocessing.html>
13. How to use concurrent futures in Python - LabEx, accessed June 23, 2025, <https://labex.io/tutorials/python-how-to-use-concurrent-futures-in-python-452184>
14. Understanding the Differences: ThreadPoolExecutor vs. ProcessPoolExecutor in Python, accessed June 23, 2025, <https://en.ittrip.xyz/python/thread-vs-process-executors>
15. Python ThreadPool vs. Multiprocessing - The New Stack, accessed June 23, 2025, <https://thenewstack.io/python-threadpool-vs-multiprocessing/>
16. concurrent.futures — Manage Pools of Concurrent Tasks - PyMOTW 3, accessed June 23, 2025, <https://pymotw.com/3/concurrent.futures/>
17. Python: Wait on all of `concurrent.futures.ThreadPoolExecutor`'s futures - Stack Overflow, accessed June 23, 2025, <https://stackoverflow.com/questions/21143162/python-wait-on-all-of-concurrent-futures-threadpoolexecutors-futures>
18. concurrent.futures — Asynchronous computation - Pythonhosted.org, accessed June 23, 2025, <https://pythonhosted.org/futures/>
19. Threads, ThreadPools and Executors - Multi Thread Processing In Java - SoftwareMill, accessed June 23, 2025, <https://softwaremill.com/threadpools-executors-and-java/>
20. Understanding Thread Pool Executor in Python - HeyCoach | Blogs, accessed June 23, 2025, <https://blog.heycoach.in/thread-pool-executor-2/>
21. How to use ThreadPoolExecutor in Python3 ? - GeeksforGeeks, accessed June 23, 2025, <https://www.geeksforgeeks.org/python/how-to-use-threadpoolexecutor-in-python3/>
22. Parallelism, Concurrency, and AsyncIO in Python - by example | TestDriven.io, accessed June 23, 2025, <https://testdriven.io/blog/python-concurrency-parallelism/>
23. Concurrency Models in Python - Jensen Low, accessed June 23, 2025, <https://www.jensenlwt.com/blog/concurrency-in-python/>
24. # Boost Your Python Tasks with `ThreadPoolExecutor` - DEV Community, accessed June 23, 2025, <https://dev.to/blamsa0mine/-boost-your-python-tasks-with-threadpoolexecutor-4dbf>
25. Thread pool of synchronous I/O vs. single process using async I/O - Reddit, accessed June 23, 2025, <https://www.reddit.com/r/ExperiencedDevs/comments/1iotsvk/thread_pool_of_synchronous_io_vs_single_process/>
26. Java Thread Pool Implementation and Best Practices in Business Applications, accessed June 23, 2025, <https://www.alibabacloud.com/blog/java-thread-pool-implementation-and-best-practices-in-business-applications_601528>
27. python: read files in parallel and merge them - Pratap Vardhan, accessed June 23, 2025, <https://pratapvardhan.com/notes/python/read-files-parallel-merge/>
28. What is the difference between ProcessPoolExecutor and ThreadPoolExecutor?, accessed June 23, 2025, <https://stackoverflow.com/questions/51828790/what-is-the-difference-between-processpoolexecutor-and-threadpoolexecutor/51829082>
29. Python Concurrency: Threads, Processes, and asyncio Explained - Newvick's blog, accessed June 23, 2025, <https://newvick.com/posts/python-concurrency/>
30. Scalable and Computationally Reproducible Approaches to Arctic Research - 7 Software Design I: Functions and Concurrency, accessed June 23, 2025, <https://learning.nceas.ucsb.edu/2022-09-arctic/sections/07-software-design-1.html>
31. The Challenges and Pitfalls of Using Executors in Java - DZone, accessed June 23, 2025, <https://dzone.com/articles/the-challenges-and-pitfalls-of-using-executors-in>
32. multiprocessing — Process-based parallelism — Python 3.13.5 documentation, accessed June 23, 2025, <https://docs.python.org/3/library/multiprocessing.html>
33. Concurrency in Python - Pool of Processes - Tutorialspoint, accessed June 23, 2025, <https://www.tutorialspoint.com/concurrency_in_python/concurrency_in_python_pool_of_processes.htm>
34. Multiprocessing best practices - PyTorch documentation, accessed June 23, 2025, <https://docs.pytorch.org/docs/stable/notes/multiprocessing.html>
35. ProcessPoolExecutor Class in Python - GeeksforGeeks, accessed June 23, 2025, <https://www.geeksforgeeks.org/python/processpoolexecutor-class-in-python/>
36. Debugging Heisenbugs: A tale of parallel processing - DeepSource, accessed June 23, 2025, <https://deepsource.com/blog/debugging-heisenbugs>
37. How to stop memory leakage? · Issue #131 · uqfoundation/pathos - GitHub, accessed June 23, 2025, <https://github.com/uqfoundation/pathos/issues/131>
38. Python's Multiprocess Pool Memory Leak | Konpat's Record of Struggles, accessed June 23, 2025, <https://blog.konpat.me/dev/2016/11/08/pythons-multiprocess-pool-memory-leak.html>
39. Python - paralellizing CPU-bound tasks with concurrent.futures - Eli Bendersky's website, accessed June 23, 2025, <https://eli.thegreenplace.net/2013/01/16/python-paralellizing-cpu-bound-tasks-with-concurrent-futures>