Electrical & Computer
Engineering Department
**UNIVERSITY OF
PELOPONNESE**

Parallel Processing in Python 3.13

Parallel Systems and Programming ECE_INF920

Students

Mantvydas Deltuva ece24601

Justinas Teselis ece24600

Professor

Παναγιώτης Αλεφραγκής

Patras, 2025

Contents

Illustrations

1. Introduction

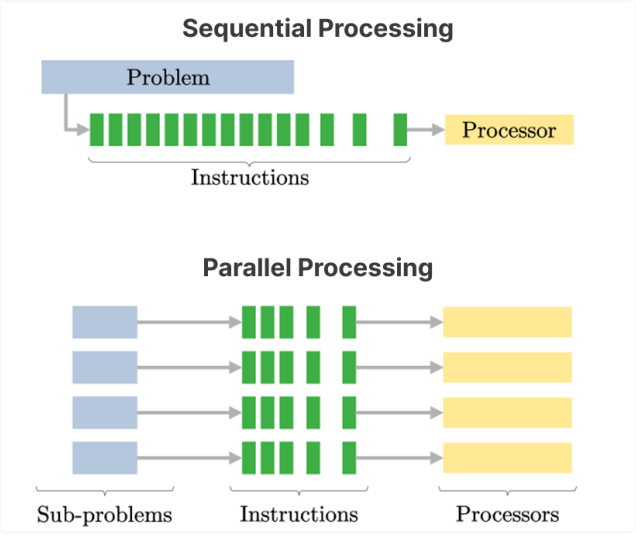
In today's era where computational demands continue to rise, parallel processing has become a cornerstone of modern computing. By leveraging multi-core processors and distributed systems, parallel processing allows for the efficient execution of tasks, enabling faster computations and improved scalability. This document explores the principles and applications of parallel processing, particularly in Python, and highlights the advancements introduced in Python 3.13.

1.1. Definition of Parallel Processing

Parallel processing provides the ability to execute multiple (usually repetitive) tasks simultaneously by distributing them across multiple threads, CPU cores or systems. It leverages today's standard multi-core processor architectures by dividing large computational workloads into smaller tasks. By doing so, it reduces overall execution time and enhances the performance of computations, particularly those involving intensive calculations or data processing.

Key aspects of parallel processing include:

- **Concurrency** - the ability to manage multiple tasks at the same time.
- **Scalability** - computing resources are efficiently utilized by managing increasing workloads.
- **Synchronization** - ensuring tasks interact seamlessly without conflicts, memory corruption or errors.



1 pic. Comparison of Sequential and Parallel Computing Approaches

1.2. Importance in Modern Computing

Parallel processing is foundational to the efficiency and scalability of modern computing systems. Its presence in various fields such as machine learning, scientific simulations, data analysis and web services is crucial for growing demands of computational resources. The primary benefits include:

- **Enhanced scalability** - modern systems often manage massive datasets and complex computations. Parallel processing allows applications to scale efficiently, accommodating these demands without bottlenecks.
- **Reduced computation time** - tasks that would take tremendous amounts of time to complete can now be completed in significantly less time, enabling quicker insights and faster decision-making.
- **Improved computational efficiency** - by dividing tasks among multiple processors or cores, parallel processing ensures optimal utilization of hardware resources, minimizing idle time and maximizing throughput.

Examples of areas that benefit from parallel processing include:

- **Real-time applications** - enhancing the responsiveness of systems such as autonomous vehicles and online gaming.
- **Scientific research** - simulating physical systems or modeling complex phenomena such as galaxies or other bodies.

1.3. Purpose of the Document

This document serves as a guide for understanding and implementing Python's parallel processing capabilities, with a focus on newly added features which were introduced in Python 3.13. Specifically, it aims to:

- **Highlight Python's progress in parallelism** – showcase Python's tools and libraries for parallel computing, including the introduction of the GIL-free mode in Python 3.13.
- **Demonstrate real-world applications** – define practical use cases where parallel processing can drive significant performance improvements and differences in implementations.

With the **experimental removal** of the **Global Interpreter Lock (GIL) in Python 3.13**, this guide is mostly focused on this feature for multi-threaded programming and the opportunities it unlocks for Python developers in achieving true parallelism.

2. Background

The efficiency of Python's concurrency model is a widely discussed topic, with much of the conversation revolving around its **Global Interpreter Lock (GIL)**. Python's threading model, often misunderstood due to the constraints of the GIL, has both strengths and weaknesses. Understanding the differences between threads and processes in Python is key to designing efficient, scalable applications, especially in multi-core or distributed computing environments.

2.1. The Global Interpreter Lock (GIL)

The most widely used Python implementation uses **Global Interpreter Lock (GIL)** which is a crucial feature that has its own advantages and disadvantages. It is a mutex (mutual exclusion lock) that enforces rules for native threads for executing Python code simultaneously. It disallows concurrent execution. This way it simplifies memory management, garbage collection and ensures thread safety. But the main drawback is that it significantly impacts the performance of multi-threaded programs.

The GIL was introduced in the early days of Python to simplify the implementation of it by:

- **Reducing complexity** - the lock eliminates the need for complex synchronization mechanisms across threads and thus makes it easier for development. Internal state remains consistent by preventing multiple threads from executing at the same time.
- **Managing memory** - the GIL ensures that memory allocation reference counts are updated safely (one by other) without race conditions.
- **Supporting extensions** - many extensions for Python depend on the GIL to avoid implementing their own thread-safety mechanisms. For extensions developers this reduces the complexity of thread-safe code.

Despite its advantages, the GIL provides certain drawbacks:

- **Latency in I/O-bound scenarios** - context switching among threads introduces latency, which can degrade performance for applications with many concurrent and repeating I/O tasks.
- **Underutilization of multi-core systems** – today’s hardware with multiple cores is not fully utilized due to the single-threaded code execution bottleneck by the GIL.

- **”True” multi-threading** – as GIL only allows one thread to execute Python code at a time, the performance of multi-threaded applications designed for CPU-bound tasks has internal limitations by the language.
- **Workarounds** - developers often use multiprocessing or rewriting critical sections in other languages to overcome GIL-related performance issues. Development overhead and complexity increases by the size of project.

2.2. Threads vs. Processes

Threads are lightweight units of execution which live in a single specific process. This way it is using already allocated memory and resources of parent process. However, they also share the constraints of the Global Interpreter Lock in Python, which limits their effectiveness for certain tasks.

Key features of threads:

- **Low overhead** – threads create minimal memory overhead compared to processes because they operate within a single, already initiated with resources and memory, process.
- **Shared memory space** – does not require inter-process communication (IPC) because it operates within the same process. This way it can access and modify the same data structure without IPC.
- **I/O-bound tasks** - threads excel at tasks, such as reading and writing to files, network communication, or database interactions.

Limitations of threads:

- **Thread safety** – synchronization locks or semaphores are essential to ensure that race condition does not occur in shared memory space.
- **GIL dependency** – threads are ineffective for CPU-bound tasks, as GIL prevents them from executing Python code concurrently. Only one thread can do that at a time.

Processes are independent execution units, each with its own memory space and system resources. They bypass the GIL entirely, making them suitable for tasks requiring true concurrent calculation.

Key features of processes:

- **True parallelism** – since each process operates independently, they can fully utilize multi-core CPUs.

- **Independent memory space** – race condition is not present because processes do not share memory by default. This isolation enhances stability but also requires IPC mechanisms for communication between processes.
- **CPU-bound tasks** – processes are ideal for computationally intensive tasks, such as data processing, machine learning or numerical simulations.

Limitations of processes:

- **Overhead** – each process has its own CPU resources and memory allocation thus creating overhead for managing processes.
- **Complex communication** - sharing data between processes requires explicit mechanisms like queues, pipes, or shared memory.

Feature	***Threads***	***Processes***
Memory Sharing	Shared memory space	Independent memory space
Parallelism	Limited by the GIL in Python	True parallelism across cores
Overhead	Minimal	High
Best Use Case	I/O-bound tasks	CPU-bound tasks
Communication	Easier (shared memory)	Requires IPC mechanisms
Safety	Risk of race conditions	No shared memory race conditions

2 pic. Threads and Processes Feature Comparisons

2.3. Evolution of Parallel Processing in Python

Now, Python has multiple modules that leap towards efficient parallel processing, each with its own advantages and disadvantages. These tools have laid the groundwork for parallel computing in Python.

threading:

- Introduced lightweight threading for I/O-bound tasks.
- Threads share memory within a single process, enabling fast communication but making them dependent on the GIL.

multiprocessing:

- Enabled true parallelism by creating separate processes instead of threads.
- Each process has its own memory space, allowing them to operate independently and bypass the GIL.
- Suited for CPU-bound tasks but has higher overhead due to the need for inter-process communication (IPC) and separate memory allocation.

concurrent.futures:

- Provided a high-level interface for managing pools of threads and processes. Simplified the implementation of concurrent tasks.
- Designed for developers seeking to implement parallelism with minimal boilerplate, while offering flexibility in choosing between threads, processes, or both.

3. Parallel Processing Basics in Python

Python has several modules to implement concurrency and parallelism, each suited for specific use cases. Whether dealing with I/O-bound or CPU-bound tasks, these modules provide the tools to manage threads, processes, and tasks effectively. This section introduces the **threading**, **multiprocessing**, and **concurrent.futures** modules, explaining their key features, limitations, and ideal use cases, along with practical examples.

3.1. Module threading

Python threads are most effective for I/O-bound tasks therefore threading module can achieve concurrency in Python for tasks such as reading files, handling network requests or other tasks, where program must wait for external resources. Memory space is shared across threads, making data sharing simple but prone to race conditions. To ensure thread-safe execution, module provides synchronization primitives such as locks, semaphores. This module is constrained by the GIL for CPU-bound tasks which limits true parallelism and its effectiveness. For more information regarding this module, please refer to the official documentation on threading [1].

Below is an example of how to download and save multiple images concurrently using threading module. Code explanation:

- Threads are created to manage multiple file downloads simultaneously with universal image download function.
- thread.start() begins the execution of each thread.
- thread.join() ensures the main program waits for all threads to finish before proceeding.

```python
import threading
import requests
import time
import os

# Create a start event
start_event = threading.Event()

# Universal function for downloading an image in .jpg format from specified URL
def download_image(id, url, output_folder):
    try:
        # Contruct image path based on provided output folder and ID
        image_path = os.path.join(output_folder, f"image_{id}.jpg")

        # Provide initialization feedback
        print(f"Thread for image [{id}] is initialized, waiting for start signal...")

        # Wait for start signal
        start_event.wait()

        # Provide start feedback
        print(f"Starting download for image [{id}] from [{url}]")

        # Send a GET request to the URL and stream the response
        response = requests.get(url, stream = True)
        response.raise_for_status()

        # Download and save the image in 8KB chunks
        with open(image_path, "wb") as image:
            for chunk in response.iter_content(chunk_size = 8192):
                image.write(chunk)

        # Provide completion feedback
        print(f"Completed download for image [{id}], saved to [{image_path}]")
    except Exception as e:
        print(f"Failed to download image [{id}]: {e}")
```

3 pic. Module threading Example Code Snippet (part 1)

```python
# Create threads for downloading images
threads = [
    threading.Thread(
        target = download_image,
        args = (i, url, output)
    )
    for i, url in enumerate(urls)
]

# Initialize all threads
for thread in threads:
    thread.start()

# Signal all threads to start
time.sleep(3)
print("Releasing threads to start downloads...")
start_event.set()

# Wait for all threads to complete
for thread in threads:
    thread.join()

# Provide finish feedback
print("All downloads completed.")
```

4 pic. Module threading Example Code Snippet (part 2)

3.2. Module multiprocessing

Python processes are most effective for CPU-bound tasks therefore multiprocessing module can achieve parallelism in Python for tasks such as numerical computations, data processing or other tasks, where program can benefit from multi-core CPU architectures by creating separate processes, each with its own memory space and CPU resources. To have clear communication with processes there are supported tools for it like pools, queues, or pipes. This module overcomes the GIL limitation. For more information regarding this module, please refer to the official documentation on multiprocessing [2].

Below is an example of how to compute number squares in parallel using multiprocessing module. Code explanation:

- A Pool of workers is created to divide the computation of squares across multiple processes.
- Function pool.map() distributes tasks to the worker processes and collects the results.
- Each process runs independently, leveraging multiple CPU cores for true parallel execution.

```python
from multiprocessing import Pool, current_process
import random
import time

# Universal function to compute the square of a number
def compute_square(number):
    # Provide start feedback
    process_name = current_process().name
    sleep_time = random.uniform(0.5, 2.0)
    print(f"Process {process_name} is calculating square " +
        f"for {number} with sleep time {sleep_time}")

    # Simulate a heavy computation with a random sleep time
    time.sleep(sleep_time)

    # Return the square of the number
    return number * number

if __name__ == "__main__":
    # Numbers to calculate the square of
    numbers = [1, 512, 4, 128, 16, 32, 64, 8, 256, 2]

    # Create a pool of processes to calculate the squares
    with Pool(processes = 4) as pool:
        results = pool.map(compute_square, numbers)

    # Provide finish feedback
    print(f"Squares: {results}")
```

5 pic. Module multiprocessing Example Code Snippet

3.3. Module concurrent.futures

For unified usage of threads and processes there is concurrent.futures module. It simplifies the implementation of parallelism regarding management of threads, processes, and tasks. It also includes pool executioners with helpful utility functions such as submit() or map(). Threads and processes have the same characteristics as mentioned before, where threads are dependent on GLI, and processes have overhead regarding memory. For more information regarding this module, please refer to the official documentation on concurrent.futures [3].

Below is an example of how to compute number squares in parallel using multiprocessing module. Code explanation:

- ProcessPoolExecutor is used to manage a pool of processes for computing data concurrently.
- executor.map() applies the provided function to each number in the list. This way the load is distributed evenly.
- The module simplifies thread management, reducing boilerplate code and providing more control over pools.

```python
from concurrent.futures import ProcessPoolExecutor
from multiprocessing import current_process
import random
import time
import math

# Universal function to compute factorial of a number
def compute_factorial(number):
    # Provide start feedback
    process_name = current_process().name
    sleep_time = random.uniform(0.5, 2.0)
    print(f"Process {process_name} is calculating factorial " +
        f"for {number} with sleep time {sleep_time}")

    # Simulate a heavy computation with a random sleep time
    time.sleep(sleep_time)

    # Return the factorial of the number
    return math.factorial(number)

if __name__ == "__main__":
    # Numbers to calculate the factorial of
    numbers = [1, 16, 4, 5, 8, 6, 2]

    # Create ProcessPoolExecutor for parallel execution
    with ProcessPoolExecutor(max_workers=4) as executor:
        results = list(executor.map(compute_factorial, numbers))

    # Provide finish feedback
    print(f"Results: {results}")
```

6 pic. Module concurrent.futures Example Code Snippet

4. Free-threaded mode in Python 3.13

Python 3.13 was released on October 7, 2024 and it introduced significant new features and optimizations. The most notable changes include a revamped **interactive interpreter**, an experimental **Just-In-Time (JIT) compiler**, and an experimental **free-threaded mode**, which we will explore further.

The revolutionary **free-threaded mode** significantly improves Python's ability to manage multi-threaded applications. This mode disables the **Global Interpreter Lock (GIL)**, a long-standing feature introduced in Python 1.5 that prevented multiple threads from executing Python bytecode simultaneously, making it possible that only one thread can execute Python bytecode at a time. While the GIL ensured thread safety, it severely limited Python’s performance in multi-threaded tasks.

Free-threaded mode addresses this limitation by enabling threads to execute in parallel across multiple CPU cores, fully utilizing multi-core processors. This feature is particularly advantageous for workloads that require multi-threading, such as simulations, data analysis, and other compute-intensive operations. With the enablement of true parallelism, significant performance improvements can be achieved, particularly in scenarios designed with threading in mind.

However, the feature currently remains **experimental**, and its impact may vary depending on the application:

- **Performance gains** - multi-threaded programs designed for concurrency can benefit from increased speed and efficiency on multi-core hardware.
- **Compatibility concerns** - legacy libraries or applications designed with the GIL assumption may encounter issues. Some libraries may not work or require updates to function correctly in free-threaded mode.
- **Single-threaded applications** - these applications might see performance decline due to the added complexity of thread management without the GIL.

To use free-threaded mode, developers must run their applications with a specialized executable, called *python3.13t* (or python3.13t.exe). Pre-built binaries for free-threaded mode are available in official *Windows* and

macOS installers. Alternatively, developers can also build CPython from source with the *--disable-gil* option for customized configurations [4].

At the time of writing this document, free-threaded mode is still experimental and under active development. Once major challenges are solved, it is expected to become a standard part of Python in a future release. For now, developers can try it out to see its benefits and limitations.

5. Comparing GIL and GIL-free Performance

In this section, we will explore the performance differences between traditional GIL-based execution and the new GIL-free mode by analyzing the benchmarks of their behavior across a range of scenarios, including CPU-bound tasks and I/O-bound tasks. The discussion will highlight the benefits and limitations of each approach. This analysis will provide basic understanding of how execution types influence Python application performance. **It is important to note that specific results may vary depending on each system**. Configurations, installed hardware, operating system, and Python implementation can have positive or negative effects of performance. However, the overall trends - such as the percentage of the performance gains in single-threaded, multi-threaded and multi-process of GIL-free execution versus GIL-enabled - are expected to remain consistent across most environments.

5.1. Count

We will begin our experimentation with a simple counting algorithm. The goal of this algorithm is to count to a specific number as fast as possible. Below is the code snippet for the single threaded version of this algorithm.

```python
# A CPU-bound task: counting to a number
# Single Threaded (consecutive)
def count(n: int, start: int = 0) -> int:
    result = 0
    for _ in range(start, n, 1):
        result += 1
    return result
```

7 pic. Count Single-Threaded Code Snippet

Next, we extend this task to evaluate concurrent executions, utilizing Python's *threading* and *multiprocessing* modules. These approaches will allow us to assess the impact of threading and multiprocessing on performance.

```python
# Multi Threaded (concurrent)
def threaded_count(n: int, num_threads: int) -> int:
    threads = []
    chunk_size = n // num_threads
    results = [0] * num_threads

    def __worker(index: int, start: int, end: int) -> None:
        results[index] = count(end, start)

    for i in range(num_threads):
        start = i * chunk_size
        end = (i + 1) * chunk_size if i < num_threads - 1 else n

        thread = threading.Thread(target=__worker, args=(i, start, end))
        threads.append(thread)

    for thread in threads:
        thread.start()

    for thread in threads:
        thread.join()

    return sum(results)
```

8 pic. Count Multi-Threaded Code Snippet

```python
# Multi Process (concurrent)
def process_count(n: int, num_processes: int) -> int:
    chunk_size = n // num_processes

    tasks = [
        (
            # end index
            ((i + 1) * chunk_size if i < num_processes - 1 else n),
            # start index
            i * chunk_size,
        )
        for i in range(num_processes)
    ]

    with multiprocessing.Pool(processes=num_processes) as pool:
        results = pool.starmap(count, tasks)

    return sum(results)
```

9 pic. Count Multi-Process Code Snippet

Let us try evaluating a single target value for the counting algorithm. This initial test will provide a baseline comparison of single threaded, multi-threaded and multi-process executions with and without Global Interpreter Lock. The test was done with four threads, four processes and counting to one hundred million. The results highlight significant differences in

performance between GIL-enabled and GIL-free execution models across single-threaded, multi-threaded, and multi-process scenarios.

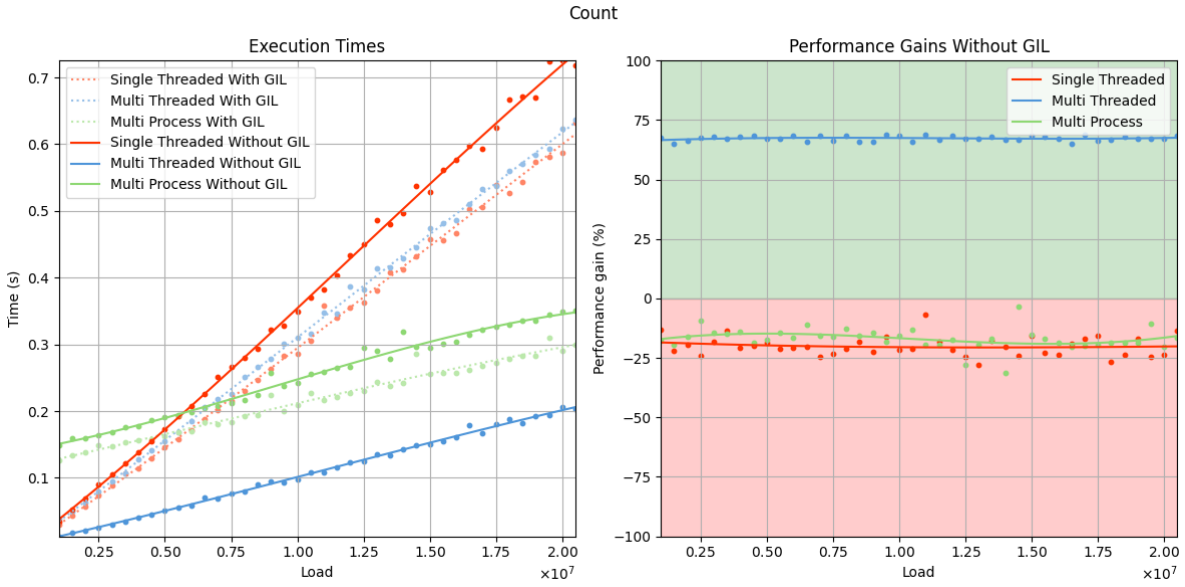
With GIL:

- **Single-threaded (2.89 seconds)** - execution performs as expected, providing a baseline for comparison. Since no concurrency is involved, the GIL does not impact this method directly.
- **Multi-threaded (3.06 seconds)** - execution is slightly slower than single-threaded due to GIL. Threads must wait for exclusive access to the GIL, which negates the potential performance gains of parallelism in CPU-bound tasks.
- **Multi-process (0.97 seconds)** - execution performs significantly faster because each process operates independently, bypassing the GIL. This demonstrates the advantage of multiprocessing for CPU-bound tasks in a GIL-enabled environment.

Without GIL:

- **Single-threaded (3.33 seconds)** - execution is slightly slower than its GIL-enabled counterpart. This may be due to implementation overheads introduced by removing the GIL.
- **Multi-threaded (0.96 seconds)** - execution sees a dramatic improvement without the GIL, completing faster than any other method. Threads can now execute concurrently and fully utilizing available CPU cores.
- **Multi-process (1.13 seconds)** - the performance of execution is slightly slower without the GIL compared to the GIL-enabled case. This may be attributed to the increased overhead of inter-process communication or other system-level variations when the GIL is not managing resource allocation.

Now, let us extend the analysis to an array of target values. For each value in the range, we will measure execution times for all methods under both GIL-enabled and GIL-free scenarios. These results will allow us to evaluate how the presence or absence of GIL influences performance as the computational workload increases. To present our findings, we will use graphical visualization to illustrate execution times across the target values. The graphs will highlight overall trends of different executions. Below is the visualization of gathered results from the target value array [1000000:20500000]. The load represents the number to which counting is happening.



10 pic. Count Performance Graphs

After evaluating the program, we can see clear differences between having the GIL enabled and disabled. With the GIL enabled, the multi-process algorithm performs the best, while the multi-threaded algorithm performs the worst, and the single-threaded algorithm performs slightly better than the multi-threaded algorithm. With the GIL disabled, the results shift entirely. The multi-threaded algorithm performs marginally better than the others, while the multi-process algorithm and single-threaded algorithm perform significantly worse, with the single-threaded being the least efficient.

From this point onward, testing of other algorithms will be performed in an equivalent manner.

5.2. Download Images

Next, we will be assessing a program used for downloading images from the internet. The goal of this program is to download images from the provided list of URLs as fast as possible. Below is the code snippet for the single threaded version of this algorithm.

```python
# An I/O-bound task: downloading image
def __download_image(url: str, output_path: str) -> int:
    size = 0
    try:
        response = requests.get(url, timeout=2, stream=True)
        response.raise_for_status()
        with open(output_path, "wb") as f:
            for chunk in response.iter_content(chunk_size=8192):
                f.write(chunk)
                size += len(chunk)
    except Exception as e:
        logger.error(f"Failed to download {url}: {e}")
    return size
```

11 pic. Download Images Task Code Snippet

```python
# Single Threaded (consecutive)
def download_images(urls: list[str], output_folder_path: str) -> int:
    size = 0
    output_folder_path = os.path.join(
        output_folder_path, DOWNLOAD_SINGLE_THREADED_RESULT_FOLDER_NAME
    )
    os.makedirs(output_folder_path, exist_ok=True)
    for url in urls:
        output_path = os.path.join(output_folder_path, url.split("/")[-1])
        size += __download_image(url, output_path)
    return size
```

12 pic. Download Images Single-Threaded Code Snippet

Next, we extend this task to evaluate concurrent executions, utilizing Python's *threading* and *multiprocessing* modules.

```python
# Multi Threaded (concurrent)
def threaded_download_images(
    urls: list[str], output_folder_path: str, num_threads: int
) -> None:
    threads = []
    chunk_size = len(urls) // num_threads

    output_folder_path = os.path.join(
        output_folder_path, DOWNLOAD_MULTI_THREADED_RESULT_FOLDER_NAME
    )
    os.makedirs(output_folder_path, exist_ok=True)

    def __worker(start: int, end: int):
        for url in urls[start:end]:
            output_path = os.path.join(output_folder_path, url.split("/")[-1])
            __download_image(url, output_path)

    for i in range(num_threads):
        start = i * chunk_size
        end = (i + 1) * chunk_size if i < num_threads - 1 else len(urls)

        thread = threading.Thread(target=__worker, args=(start, end))
        threads.append(thread)

    for thread in threads:
        thread.start()

    for thread in threads:
        thread.join()
```

13 pic. Download Images Multi-Threaded Code Snippet

```python
# Multi Process (concurrent)
def process_download_images(
    urls: list[str], output_folder_path: str, num_processes: int
) -> None:
    output_folder_path = os.path.join(
        output_folder_path, DOWNLOAD_MULTI_PROCESS_RESULT_FOLDER_NAME
    )
    os.makedirs(output_folder_path, exist_ok=True)

    tasks = [
        (
            # url string
            url,
            # output path string
            os.path.join(
                output_folder_path,
                url.split("/")[-1],
            ),
        )
        for url in urls
    ]

    with multiprocessing.Pool(processes=num_processes) as pool:
        pool.starmap(__download_image, tasks)
```

14 pic. Download Images Multi-Process Code Snippet

The single target value test was done with four threads, four processes and two hundred URLs. The results highlight minor differences in performance between GIL-enabled and GIL-free execution models across single-threaded, multi-threaded, and multi-process scenarios.

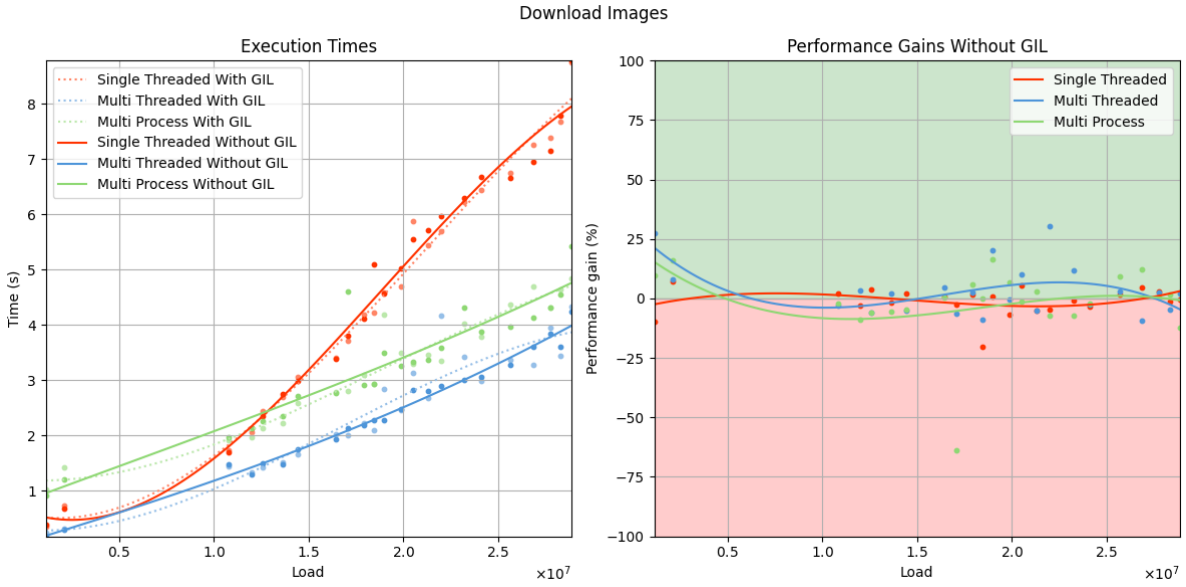
With GIL:

- **Single-threaded (20.74 seconds)** - this approach serves as the baseline for this comparison. It performs sequential processing of the URLs and is significantly slower due to the lack of concurrency.
- **Multi-threaded (9.68 seconds)** - execution shows a substantial improvement compared to single-threaded execution. In this I/O-bound scenario, the GIL does not heavily restrict threading performance because threads release the GIL during I/O operations, allowing other threads to proceed.
- **Multi-process (11.84 seconds)** - execution performs better than single-threaded but is slightly slower than multi-threaded. This is due to the higher overhead of creating and managing processes compared to threads, particularly for I/O-bound workloads.

Without GIL:

- **Single-threaded (19.94 seconds)** - execution is marginally faster without the GIL, due to minor optimizations in the underlying implementation.
- **Multi-threaded (8.19 seconds)** - execution benefits are minor from the absence of the GIL. The lack of GIL contention allows smoother thread execution even during brief periods of CPU-bound processing.
- **Multi-process (8.97 seconds)** - execution also improves without the GIL, reducing its execution time. While still slightly slower than multi-threaded execution, the gap is narrower compared to the GIL-enabled scenario.

Below is the visualization of gathered results from the target value array [1:80]. The load represents overall downloaded bytes.



15 pic. Download Images Performance Graphs

In this scenario, we observe no significant differences between running the program with the GIL enabled or disabled, although the application appears to run slightly more stably with the GIL disabled. These results could depend on numerous factors – for instance, the server might have limited the image downloads, creating a performance bottleneck. It would be better to evaluate this application in a more controlled environment.

5.3. Merge Sort

We continue our testing now with Merge Sort algorithm. Merge Sort is a divide-and-conquer sorting algorithm that divides an array into two halves, recursively sorts each half, and then merges the sorted halves back together. The process continues until the entire array is divided into individual elements, which are then combined in sorted order. Below is the code snippet for the single threaded version of this algorithm.

```python
# A CPU-bound task: merging two sorted lists
def merge(left: list[int], right: list[int]) -> list[int]:
    result = []
    i = j = 0

    while i < len(left) and j < len(right):
        if left[i] <= right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1

    result.extend(left[i:])
    result.extend(right[j:])

    return result
```

16 pic. Merge Sort Task Code Snippet

```python
# Single Threaded (consecutive)
def merge_sort(arr: list[int]) -> list[int]:
    if len(arr) <= 1:
        return arr

    # Recursively split the array into halves
    mid = len(arr) // 2
    left = arr[:mid]
    right = arr[mid:]

    left = merge_sort(left)
    right = merge_sort(right)

    # Merge the sorted halves
    return merge(left, right)
```

17 pic. Merge Sort Single-Threaded Code Snippet

Next, we extend this task to evaluate concurrent executions, utilizing Python's *threading* and *multiprocessing* modules.

```python
# Multi Threaded (concurrent)
def threaded_merge_sort(arr: list[int], num_threads: int) -> list[int]:
    if len(arr) <= 1:
        return arr

    # Worker function
    def worker(i: int, start: int, end: int) -> None:
        results[i] = merge_sort(arr[start:end])

    # Step 1: Divide the array into chunks
    chunk_size = len(arr) // num_threads
    threads = []
    results = [None] * num_threads

    # Step 2: Sort chunks in parallel
    for i in range(num_threads):
        start = i * chunk_size
        end = (i + 1) * chunk_size if i != num_threads - 1 else len(arr)
        thread = threading.Thread(target=worker, args=(i, start, end))
        threads.append(thread)
        thread.start()

    for thread in threads:
        thread.join()

    # Step 3: Merge sorted chunks sequentially
    while len(results) > 1:
        next_round = []
        for i in range(0, len(results), 2):
            if i + 1 < len(results):
                next_round.append(merge(results[i], results[i + 1]))
            else:
                next_round.append(results[i])
        results = next_round

    return results[0]
```

18 pic. Merge Sort Multi-Threaded Code Snippet

```python
# Multi Process (concurrent)
def process_merge_sort(arr: list[int], num_processes: int) -> list[int]:
    if len(arr) <= 1:
        return arr

    # Step 1: Divide the array into chunks
    chunk_size = len(arr) // num_processes
    chunks = [arr[i * chunk_size:(i + 1) * chunk_size] for i in range(num_processes - 1)]
    chunks.append(arr[(num_processes - 1) * chunk_size:])  # Add remaining elements

    # Step 2: Sort chunks in parallel
    with multiprocessing.Pool(processes=num_processes) as pool:
        results = pool.map(merge_sort, chunks)

    # Step 3: Merge sorted chunks sequentially
    while len(results) > 1:
        next_round = []
        for i in range(0, len(results), 2):
            if i + 1 < len(results):
                next_round.append(merge(results[i], results[i + 1]))
            else:
                next_round.append(results[i])
        results = next_round

    return results[0]
```

19 pic. Merge Sort Multi-Process Code Snippet

The single target value test was done with four threads, four processes and one million numbers in array. The results highlight minor change or performance decrease between GIL-enabled and GIL-free execution models across single-threaded, multi-threaded, and multi-process scenarios.

With GIL:

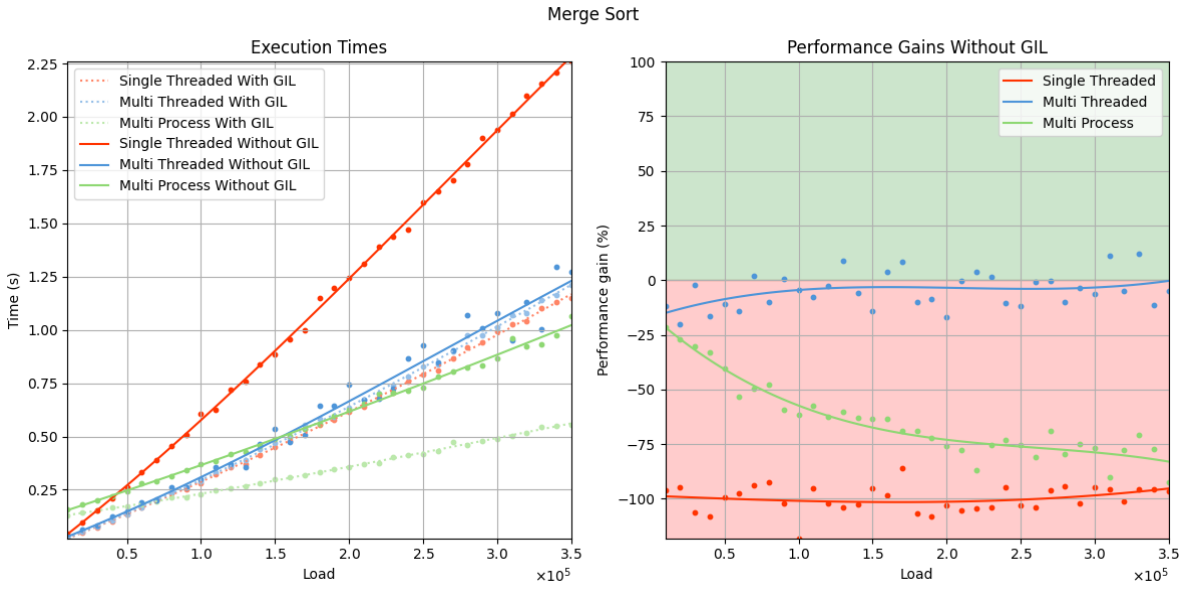
- **Single-threaded (3.83 seconds)** - execution performs well in a GIL-enabled environment as there is no concurrency. All computation occurs sequentially, avoiding the GIL overhead associated with thread context switching.
- **Multi-threaded (3.92 seconds)** - execution shows negligible improvement over single-threaded execution. In fact, it is slightly slower due to GIL contention, which serializes CPU-bound threads and introduces additional overhead.
- **Multi-process (1.64 seconds)** - multiprocessing performs significantly better, as each process runs in its own Python interpreter and bypasses the GIL entirely. This allows true parallel execution across CPU cores.

Without GIL:

- **Single-threaded (7.20 seconds)** - execution without the GIL is unexpectedly slower than its GIL-enabled counterpart. This is due to additional implementation overheads or less optimized handling in a GIL-free environment.

- **Multi-threaded (3.84 seconds)** - execution performs similarly to the GIL-enabled case. Without the GIL, threads can execute concurrently on multiple cores, but the lack of improvement suggests that other factors, such as thread management overhead, limit performance gains.
- **Multi-process (3.07 seconds)** - execution remains faster than single-threaded and multi-threaded approaches in the GIL-free scenario. However, the removal of the GIL results in slightly slower performance compared to the GIL-enabled case, due to system-level overhead in process communication.

Below is the visualization of gathered results from the target value array [10000:350000]. The load represents the count of numbers in an array.



20 pic. Merge Sort Performance Graphs

This time, we can see that with the GIL disabled, the algorithm performs worse across all tests. This could be because currently performance with GIL disabled is significantly degraded in single-threaded algorithms. Since the merge sort algorithm relies on a single thread during its final stage to merge the sorted subarrays into the final array, this bottleneck could be heavily impacting the overall performance.

5.4. Matrix Multiplication

Finally, we evaluate the matrix multiplication algorithm. Matrix multiplication involves multiplying two matrices, where each element of the result is the dot product of a row from the first matrix and a column from the second. In this case, we first transpose matrix B to optimize the multiplication. By transposing, we align the rows of matrix A with the columns of matrix B, which can improve efficiency. Below is the code snippet for the single threaded version of this algorithm.

```python
# A CPU-bound task: matrix multiplication of row and column
def __row_col_multiplication(row: list[int], col: list[int]) -> int:
    # Inner product of two vectors
    return sum(a * b for a, b in zip(row, col))
```

21 pic. Matrix Multiplication Task Code Snippet

```python
# Single Threaded (consecutive)
def matrix_multiplication(
    matrix_a: list[list[int]], matrix_b: list[list[int]]
) -> list[list[int]]:
    if len(matrix_a[0]) != len(matrix_b):
        raise ValueError("Matrix dimensions do not match.")

    matrix_b_transposed = list(zip(*matrix_b))

    # Matrix multiplication
    return [
        [__row_col_multiplication(row, col) for col in matrix_b_transposed]
        for row in matrix_a
    ]
```

22 pic. Matrix Multiplication Single-Threaded Code Snippet

Next, we extend this task to evaluate concurrent executions, utilizing Python's *concurrent.futures* and *multiprocessing* modules.

```python
# Multi Threaded (concurrent)
def threaded_matrix_multiplication(
    matrix_a: list[list[int]], matrix_b: list[list[int]], num_threads: int
) -> list[list[int]]:
    if len(matrix_a[0]) != len(matrix_b):
        raise ValueError("Matrix dimensions do not match.")

    # Worker function
    def __worker(i: int) -> list[int]:
        intermediate_row = []
        for j in range(len(matrix_b_transposed)):
            intermediate_row.append(
                __row_col_multiplication(matrix_a[i], matrix_b_transposed[j])
            )
        return intermediate_row

    matrix_b_transposed = list(zip(*matrix_b))
    result = []

    with concurrent.futures.ThreadPoolExecutor(
        max_workers=num_threads
    ) as executor:
        for i in range(len(matrix_a)):
            result.append(executor.submit(__worker, i))

    concurrent.futures.wait(result)
    return [future.result() for future in result]
```

23 pic. Matrix Multiplication Multi-Threaded Code Snippet

```python
# Multi Process (concurrent) worker function
def __process_worker(
    i: int,
    matrix_a: list[list[int]],
    matrix_b_transposed: list[list[int]],
) -> list[int]:
    intermediate_row = []
    for j in range(len(matrix_b_transposed)):
        intermediate_row.append(
            __row_col_multiplication(matrix_a[i], matrix_b_transposed[j])
        )
    return intermediate_row


# Multi Process (concurrent)
def process_matrix_multiplication(
    matrix_a: list[list[int]], matrix_b: list[list[int]], num_processes: int
) -> list[list[int]]:
    if len(matrix_a[0]) != len(matrix_b):
        raise ValueError("Matrix dimensions do not match.")

    matrix_b_transposed = list(zip(*matrix_b))
    tasks = [
        (
            # rows index
            i,
            # first matrix
            matrix_a,
            # second matrix transposed
            matrix_b_transposed,
        )
        for i in range(len(matrix_a))
    ]

    with multiprocessing.Pool(processes=num_processes) as pool:
        result = pool.starmap(__process_worker, tasks)

    return result
```

24 pic. Matrix Multiplication Multi-Process Code Snippet

The single target value test was done with four threads, four processes, matrix a dimensions 543x543 and matrix b dimensions 543x777. The results highlight performance increase between GIL-enabled and GIL-free execution models across single-threaded, multi-threaded, and multi-process scenarios.

With GIL:

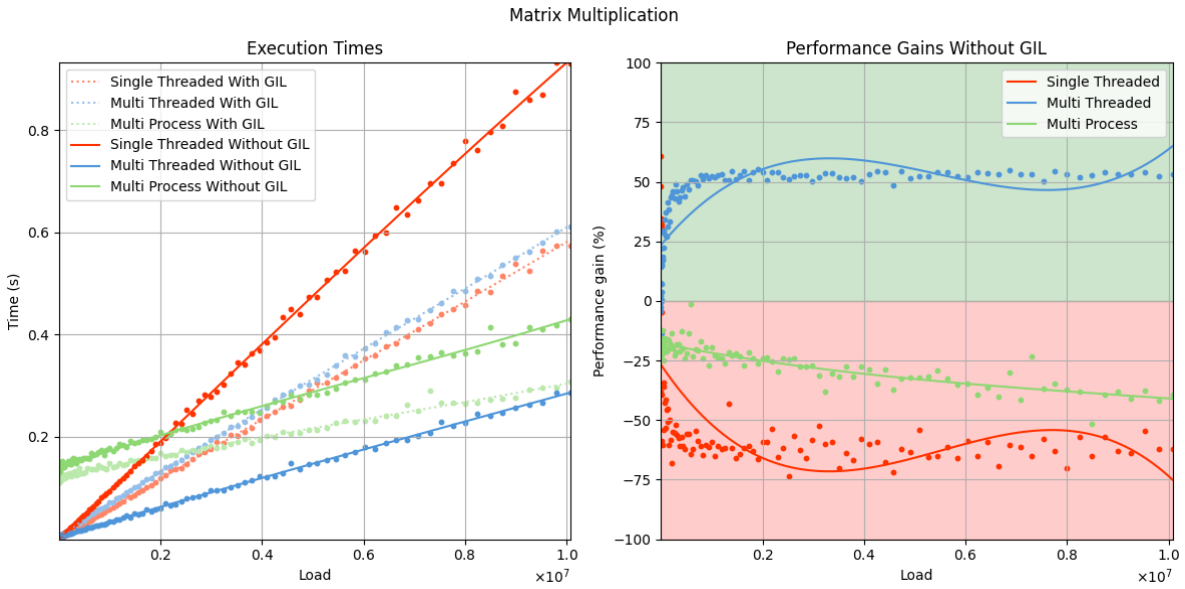
- **Single-threaded (12.98 seconds)** - execution serves as the baseline for comparison. It performs all computations sequentially, unaffected by the GIL, but cannot utilize multiple CPU cores.
- **Multi-threaded (13.52 seconds)** - execution is slightly slower than single-threaded execution. The GIL prevents true parallelism in this CPU-bound task, and thread management overhead adds to the execution time.

- **Multi-process (3.89 seconds)** - execution significantly outperforms single-threaded and multi-threaded approaches. By bypassing the GIL, it utilizes multiple CPU cores effectively for parallel computations.

Without GIL:

- **Single-threaded (19.26 seconds)** - execution without the GIL is unexpectedly slower. This performance drop may be attributed to additional implementation overheads or less efficient handling of single-threaded operations in a GIL-free environment.
- **Multi-threaded (5.71 seconds)** - execution improves dramatically without the GIL, as threads can now execute concurrently across CPU cores. This highlights the benefits of GIL removal for CPU-bound tasks.
- **Multi-process (5.87 seconds)** - execution performs similarly to multi-threaded execution in the GIL-free scenario, with a slight increase in overhead compared to the GIL-enabled case.

Below is the visualization of gathered results from the target value array. The load represents the total amount of multiplication done.



25 pic. Matrix Multiplication Performance Graphs

In the case of matrix multiplication, we see a substantial performance improvement when the GIL is disabled in multi-threaded test compared to tests with the GIL enabled. With the GIL enabled, the best algorithm was multi-process with single-threaded and multi-threaded algorithms performing similarly worse. After disabling the GIL, the situation changed, and the multi-threaded algorithm outperformed all the other tests.

6. Conclusion

The experimental removal of the Global Interpreter Lock (GIL) in Python 3.13 marks a significant step forward in addressing one of the most longstanding limitations of the language, particularly in the context of multi-threaded programming. With the increasing demand for concurrent processing, driven by the rise of artificial intelligence and machine learning applications, this move is seen as an essential shift in making Python more competitive in modern development environments. The removal of the GIL holds the potential to unlock substantial performance improvements in multi-threaded workloads, which have been limited for years due to GIL's enforcement of single-threaded bytecode execution.

However, the transition away from the GIL is not without its challenges. At the time of writing, many libraries still depend on GIL being supported and could not run or do not run properly, for example, the popular math library *NumPy*. Furthermore, our testing revealed mixed results when running multi-threaded applications in the GIL-free environment. In some cases, we saw significant performance improvements, while in others, there were no noticeable benefits or even a decline in performance compared to when the GIL was enabled.

The removal of the GIL is undoubtedly a positive and necessary direction for Python's future, but further optimization and library support are needed to fully realize the benefits. With continued development and fine-tuning, Python could finally overcome its multi-threading limitations, enabling faster and more scalable applications in a variety of use cases.

Sources

[1] "threading - Thread-based parallelism - Python 3.13.1 documentation," Python Software Foundation, 27 December 2024. [Online]. Available: https://docs.python.org/3/library/threading.html. [Accessed 28 December 2024].

[2] "multiprocessing - Process-based parallelism - Python 3.13.1 documentation," Python Software Foundation, 27 December 2024. [Online]. Available: https://docs.python.org/3/library/multiprocessing.html. [Accessed 28 December 2024].

[3] "concurrent.futures - Launching parallel tasks - Python 3.13.1 documentation," Python Software Foundation, 27 December 2024. [Online]. Available: https://docs.python.org/3/library/concurrent.futures.html. [Accessed 28 December 2024].

[4] A. Turner and T. Wouters, "What’s New In Python 3.13 - Python 3.13.1 documentation," Python Software Foundation, 27 December 2024. [Online]. Available: https://docs.python.org/3.13/whatsnew/3.13.html. [Accessed 28 December 2024].