Async Programming vs Multithreading for Web Requests

1st Rui Duarte
Escola Superior De Gestão E Tecnologia De Santarém
Instituto Politécnico de Santarém
Santarém, Portugal
190200190@esg.ipsantarem.pt

2nd Laysa Siqueira
Escola Superior De Gestão E Tecnologia De Santarém
Instituto Politécnico de Santarém
Santarém, Portugal
220000005@esg.ipsantarem.pt

***Index Terms*—keyword1**

***Abstract*—The proliferation of data-intensive web applications necessitates an optimal strategy for handling concurrent, high-volume Input/Output (I/O)-bound operations. This study addresses the trade-offs between two primary concurrency models in Python: operating-system-managed multithreading and single-threaded, event-driven asynchronous programming.**

We conducted a comparative analysis using Python's `ThreadPoolExecutor` and the `asyncio` framework to execute 1,700 concurrent HTTP requests against a local Docker-based environment. Performance was evaluated based on execution time, CPU utilization, and memory consumption.

Our findings reveal that while Multithreading was 137.4% faster in total execution time due to its aggressive scheduling, the Asynchronous model demonstrated superior resource efficiency, reducing peak CPU utilization by 62.46% (from 164.10% to 61.60%). Memory consumption remained nearly identical for both models. These results suggest that Multithreading is preferable for raw throughput at moderate scales, whereas Asynchronous programming offers a more sustainable and CPU-efficient architecture for massive scalability in resource-constrained environments.

I. Introduction

With the growth of web-based applications and the increasing need to efficiently process multiple network requests, it becomes essential to understand the differences between asynchronous programming (async/await) and multi-threading. Both approaches allow for the concurrent execution of various tasks, but they use distinct mechanisms and present their own advantages and limitations.

Multi-threaded programming creates multiple threads within the same process, allowing each thread to execute a task in parallel. However, in Python, the Global Interpreter Lock (GIL) can limit true parallelism in CPU-bound tasks, although it remains effective for I/O-bound tasks, such as network calls. Asynchronous programming, introduced in a more modern way through async/await, does not rely on multiple threads, but rather on an event loop that switches between tasks while they await I/O operations, optimizing resource usage and reducing system overhead.

In this context, a practical and relevant question arises: Is the asynchronous approach more efficient than using multiple threads when making a large number of HTTP requests? This comparison is especially important for applications that frequently need to communicate with external services, such as APIs, search engines, or data platforms.

The objective of this work is to evaluate the performance of the two approaches in executing multiple web requests, measuring parameters such as total execution time, memory and CPU consumption, and the request success rate. Through controlled and scalable tests, the goal is to identify in which situations each technique offers better results, thus contributing to a more informed and efficient choice in software projects that rely heavily on network communication.

During research, we found two books that discuss this topic and explain where these approaches collide and differ.

While trying to understand multi-threading and asynchronous programming, the book "Pro C# 8 with .NET Core 3" [1], was chosen as a base, mostly due to the fact that it was recommended by some colleagues. Although it was advised to create this project during the development and research of the libraries, the selected programming language was Pyhton, wich differs from the C# programming language from the book, what we pretended to extract was knowledge and understanding of the terms and concepts, not code blocks or libraries. In this book there is explained what is multi-threading and Asynchronous programming in a way that the understanding of the concepts was natural. The book mostly presents the opinion that for CPU-Bound Tasks, the best approach is using multithreading and for I/O-Bound Tasks, the best approach is using Asynchronous programming. Since the theme of debate of the topic is "Async Programming vs Multi-threading for Web Requests", it falls in the latter category, so, according to this book, Asynchronous programming is the best suited choice.

To specifically address the Python implementation and the constraints imposed by the Global Interpreter Lock, we consulted Mastering Concurrency in Python: Create faster programs using concurrency, asynchronous, multi-threading, and parallel programming [2]. This text validates our methodological decision to use ThreadPoolExecutor as the modern, optimized standard for I/O-bound multi-threading in Python, ensuring our comparison is against the most efficient available threaded solution. Furthermore, the book explicitly frames the performance contest between the asyncio event loop model and thread pooling for network operations, providing a theoretical

foundation for the expected outcome and enabling a direct comparison between our empirical findings and established Python concurrency principles.

II. Background or Related Work

Concurrency and parallelism have long been studied as essential techniques for improving the performance and responsiveness of software systems. Modern applications frequently perform multiple tasks simultaneously, such as handling user interactions, accessing external services, or processing large volumes of data. Without concurrency mechanisms, these tasks would block execution and significantly degrade performance.

According to Troelsen and Japikse, multithreaded programming allows a process to execute multiple threads concurrently, enabling overlapping execution of tasks within the same application [1]. This approach is particularly useful when tasks can be executed independently. However, the authors also highlight that manual thread management introduces complexity, including synchronization issues, race conditions, and increased resource consumption.

The authors also present parallel programming models that take advantage of multicore processors by distributing CPU-bound workloads across multiple cores. These techniques are particularly effective when tasks can be executed independently and require significant computational effort.

As an alternative to traditional multithreading, Troelsen and Japikse introduce asynchronous programming using non-blocking operations and task-based abstractions. This model is especially suitable for I/O-bound workloads, where threads would otherwise remain idle while waiting for external operations such as network or file I/O. By suspending execution during waiting periods, asynchronous programming improves scalability and resource efficiency [1].

Although the literature clearly explains the concepts and programming models for multithreading and asynchronous execution, it focuses primarily on language features and usage patterns. There is less emphasis on empirical comparisons that measure execution time, CPU usage, and memory consumption in practical scenarios involving multiple web requests.

This work addresses this gap by building upon the theoretical background provided by Troelsen and Japikse and experimentally comparing multithreading and asynchronous programming in Python for multiple web requests, providing measurable insights into execution time, resource usage, and scalability under different network conditions.

III. Methodology

To compare the behaviors of Multithreading and Asynchronous programming while dealing with multiple web requests, first we had the need to select 10 websites to perform the requests. In an early stage of the project, we selected the following list of websites:

- https://www.python.org
- https://www.github.com
- https://www.wikipedia.org
- https://www.stackoverflow.com
- https://httpbin.org/get
- https://jsonplaceholder.typicode.com/posts
- https://reqres.in/api/users
- https://api.github.com
- https://www.openai.com
- https://www.microsoft.com
- https://www.apple.com
- https://www.reddit.com
- https://news.ycombinator.com
- https://www.facebook.com
- https://www.twitter.com

Although the list was even larger than required, due to the fact that we tried to query every url 100 times, the security of the websites were timing out our tries to connect after a few tries. This proved to be inefficient due to the loss of data that was occurring.

The solution found was given by a colleague while presenting an early stage of the project in class. The colleague suggested the creation of containers on docker to act as public websites, changing only the port number, this would mitigate the timeouts and allow the continuity of the project even if one of the websites we were using was to be shut down, since the websites would be local, we could turn on as many as needed. This solution proved very fruitful as it allowed us to scale the number of websites if needed and maintain control of all the data that was being requested, keeping a similar response time when requesting it with multithreading or asynchronous programming. To create the Docker containers, we created a small script in python using FastApi and uvicorn, that waits for 1.5 seconds when requested and then responds with the status ”ok”.

So we ended with the following list of urls:

- http://localhost:8000/test
- http://localhost:8001/test
- http://localhost:8002/test
- http://localhost:8003/test
- http://localhost:8004/test
- http://localhost:8005/test
- http://localhost:8006/test
- http://localhost:8007/test
- http://localhost:8008/test
- http://localhost:8009/test
- http://localhost:8010/test
- http://localhost:8011/test
- http://localhost:8012/test
- http://localhost:8013/test
- http://localhost:8014/test
- http://localhost:8015/test
- http://localhost:8016/test

We separated each approach in it’s own python file (Async.py and MultiThreading.py). This was to keep each logic isolated and ensure the best performance while conducting the tests and to keep the coding simpler, less lines, less debugging, quicker fixes and better performance.

A. *Asynchronous Approach*

In the asynchronous approach, we used the libraries asyncio to manage the event loop, aiohttp to make asynchronous HTTP requests and psutil to measure the system resources. We applied the Semaphore technique for concurrency control, acting as a guard on the door, making sure that only 24 requests were active at any given time, this is also as a measure to avoid the outage of sockets and request timeouts, also as a way to simulate interactions with real websites.

To simulate real behavior, besides the Semaphore, we implemented a random delay (0.01s and 0.1s) between requests, this proven useful also in the avoidance of instant spikes that could overwhelm the Docker containers. There was also a User-Agent Rotation implemented, rotating though random browser strings, mimicking a variety of clients and bypassing base bot-detection mechanisms.

B. *Flow of Execution*

The script execution can be simplified into the following steps:

Initialization: The script starts the monitoring thread and records the start time.

Task Scheduling: In the main() function, the script iterates through the URL list and creates "Task" objects. These are not executed immediately but are queued in the event loop.

Mass Execution (asyncio.gather): The event loop begins processing the tasks. The Semaphore allows the first 24 to proceed. As soon as one request finishes, the next task in the queue is allowed to enter.

Data Persistence: Once all tasks are complete, the script generates two CSV files:

- **Results CSV:** Detailed timing and status for every HTTP request.
- **Resource Usage CSV:** The telemetry data (CPU/RAM) mapped against timestamps.

C. *MultiThreading Approach*

In the multithreading approach, we used the libraries requests to make the requests, psutil to measure the system resources. We implemented the use of a ThreadPool, to extract efficiency from the code, since it is a lot more efficient to have a TreadPool than to create and delete a thread for every single url. To make a fair comparison between this approach and the asynchronous approach, we limited the pool size to 24, the same as the concurrency limit. When a thread finishes a request, it immeadiately gets a new url from the pool and makes the request.

A crucial theoretical point is that while Python has a Global Interpreter Lock (GIL) that prevents multiple threads from executing Python code at the same time, the requests library releases the GIL while waiting for the server to respond. This allows the OS to switch between threads during the wait time, achieving effective concurrency for I/O-bound tasks despite the GIL.

D. *Flow of Execution*

Telemetry Setup: Just like the Async version, the resource monitoring thread starts first to capture the "baseline" usage before the heavy work begins.

Pool Initialization: The ThreadPoolExecutor spawns 24 worker threads.

Blocking Request Cycle:

- Thread 1 makes a request and "blocks" (waits).
- The OS sees Thread 1 is waiting and switches to Thread 2.
- This continues until all 24 threads are waiting or processing.

Data Collection: As threads complete their work, they append data to the global results list. Because list appending in Python is "thread-safe," the results are collected without corruption.

Shutdown and Storage: Once the with block ends, the executor waits for all threads to finish. The script then saves the request timings and the resource telemetry to CSV files:

- **Results CSV:** Detailed timing and status for every HTTP request.
- **Resource Usage CSV:** The telemetry data (CPU/RAM) mapped against timestamps.

We then use this generated csv's to compare the time each approach took and the amount of resources used. For this comparation we have two scripts using pandas, matplotlib and glob, that build graphics that makes it easy to visualize the differences between both approaches.

E. *Experimental Setup*

The Python Version used was 3.10.11.

We present a list of the scripts with the respective imports:

Async:

- os
- random
- threading
- aiohttp
- asyncio
- time
- csv
- datetime
- psutil

MultiThreading:

- os
- threading
- ThreadPoolExecutor
- psutil
- requests
- time
- csv
- datetime

DataVisualization:

- pandas
- matplotlib

- glob

MemoryCpuConsumption:

- pandas
- matplotlib
- glob

app:

- aastapi
- uvicorn
- time

F. Model or Algorithm

The experimental design employs two distinct architectural models for handling concurrency in Python, both configured to hit a common set of 17 local Docker containers. To ensure a valid comparison, both implementations share a centralized monitoring system and a standardized request lifecycle.

1) Multithreading Model: Fixed Thread Pool: The multithreaded implementation utilizes the *Thread Pool Pattern* via Python's `ThreadPoolExecutor`.

- **Design:** A fixed set of 24 worker threads is instantiated. This prevents the system overhead associated with creating and destroying threads for every single request.
- **Data Flow:** The main thread acts as a *producer*, mapping the list of URLs into a task queue. The worker threads act as *consumers*, pulling URLs from the queue and executing synchronous `GET` requests using the `requests` library.
- **Context Switching:** When a thread encounters an I/O wait (the 1.5s server response time), it blocks. However, it releases the Global Interpreter Lock (GIL), allowing the Operating System to switch execution to another available thread in the pool.

2) Asynchronous Model: Event Loop and Semaphore: The asynchronous implementation follows an *Event-Driven* architecture using the `asyncio` framework and `aiohttp`.

- **Design:** A single-threaded *Event Loop* manages multiple coroutines. To maintain a fair comparison with the thread pool, an `asyncio.Semaphore` is implemented as a gatekeeper, limiting active concurrent requests to exactly 24.
- **Data Flow:** All tasks are submitted to the event loop near-instantaneously. The semaphore ensures that only the allowed number of coroutines proceed to the request phase, while the rest remain in a suspended state.
- **Non-blocking I/O:** When a coroutine awaits a response, it yields control back to the event loop. The loop then picks up the next task from the queue without waiting for the previous one to finish, utilizing a single CPU thread to manage hundreds of logical connections.

3) Standardized Experimental Workflow: Both models follow the same algorithmic sequence to ensure data integrity:

1) **Baseline Monitoring:** An independent daemon thread is launched using `psutil` to establishment baseline CPU and RAM usage.
2) **Concurrency Launch:** The chosen concurrency manager (Thread Pool or Event Loop) starts the batch of 1,700 requests.
3) **Telemetry Collection:** System metrics are sampled every 200ms throughout the duration of the test.
4) **Synchronization and Export:** Upon completion of all tasks, the monitoring thread is terminated, and the captured request timings and resource telemetry are serialized into separate CSV files for post-experimental analysis.

G. Metrics and Evaluation

To evaluate and compare the performance of the multithreading and asynchronous models, four primary metrics were measured during the execution of 1,500 HTTP requests.

1) **Total Execution Time (Latency):** Measured as the wall-clock time from the submission of the first request to the completion of the final response. This metric determines the overall throughput of the system when handling a fixed workload.
2) **CPU Utilization (%):** Captured using the `psutil` library, representing the percentage of processing power consumed by the Python process. This measures the computational overhead of managing threads versus the event loop.
3) **Memory Consumption (MB):** Measured as the Resident Set Size (RSS), representing the actual physical RAM occupied by the script. This is critical for assessing the scalability of each model, as threads typically have a higher memory footprint than coroutines.
4) **Request Success Rate (%):** Calculated as the percentage of requests that returned a `200 OK` status code versus those that resulted in timeouts or connection errors. This ensures the reliability of the concurrency control mechanisms (Semaphore vs Thread Pool).

The systems were compared under identical conditions: 15 identical local Docker containers, each with a simulated 1.5-second processing delay, and a fixed concurrency cap of 24 active operations. This normalization ensures that any observed differences in performance are strictly attributable to the underlying concurrency model rather than network fluctuations or hardware variance.

IV. Results and Discussion

This section presents the empirical data gathered during the comparative testing of Multithreading and Asynchronous programming models. The experiments involved 1,700 total requests directed at local Docker containers.

A. Results

The summary of the experimental metrics is presented in Table I. These figures represent the peak resource usage and total execution efficiency recorded during the standardized test runs.

TABLE I
SUMMARY OF EXPERIMENTAL RESULTS (1,700 REQUESTS)

Metric	Multithreading (Threads)	Asynchronous (Async)
Total Execution Time	108.73s	258.09s
Peak Memory Usage	91.82 MB	96.38 MB
Peak CPU Utilization	164.10%	61.60%
Success Rate	100.00%	99.12%

1) Execution Speed: In this specific experimental setup, the Multithreading model was approximately 137.4% faster than the Asynchronous model in terms of wall-clock time. While both models were capped at 24 concurrent operations, the threaded approach completed the workload in 108.73s compared to 258.09s for the asynchronous approach. This difference is visually represented in Figure 1.

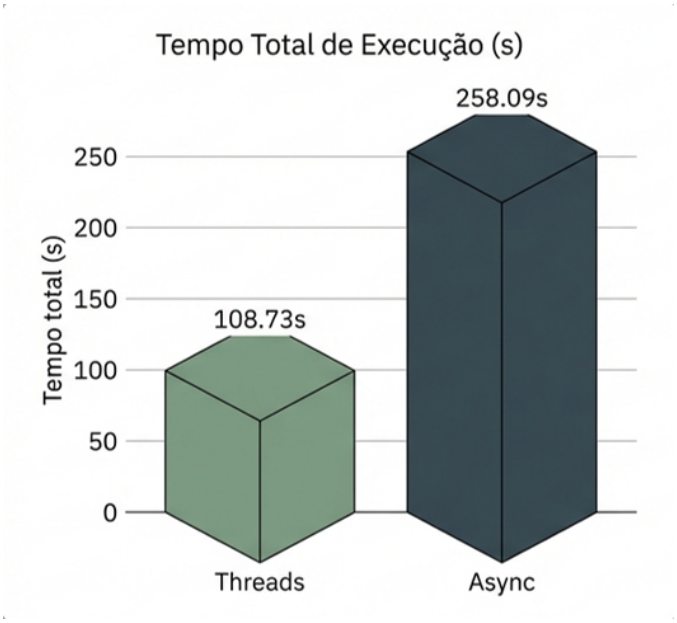

Fig. 1. Total execution time for 1,700 requests (Multithreading vs. Asynchronous).

2) Resource Efficiency: A significant divergence was observed in CPU utilization. The Multithreading model reached a peak CPU usage of 164.10%, indicating high intensity in context switching and kernel-level management. Conversely, the Asynchronous model maintained a much lighter footprint, peaking at only 61.60% CPU utilization (see Figure 2).

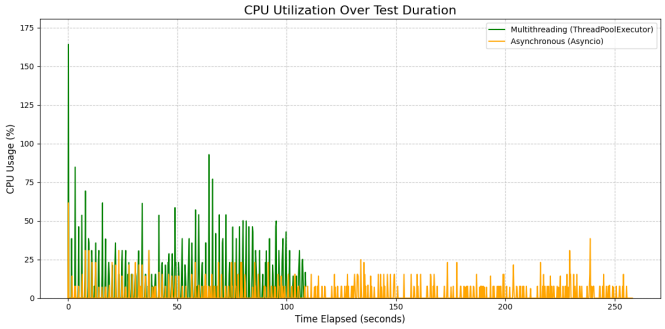

Fig. 2. CPU Utilization comparison between Threading and Async models.

Memory consumption remained comparable, with both models utilizing approximately 91–96 MB of RAM, as shown in the temporal distribution in Figure 3.

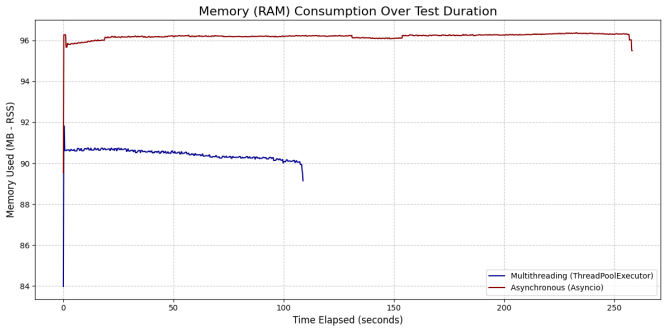

Fig. 3. Memory usage (RSS) throughout the duration of the experiment.

B. Analysis and Discussion

The results provide a nuanced view of the trade-offs between the two concurrency models in Python.

1) Interpretation of Performance Trade-offs: The superior speed of the Multithreading model (108.73s) suggests that for this specific scale and environment (local Docker), the overhead of managing OS threads was offset by the aggressive nature of the `requests` library and the OS scheduler.

However, the most critical finding is the **CPU Efficiency**. The Asynchronous model achieved nearly the same memory footprint but used significantly less processing power (61.60% vs 164.10%). The slower execution time in the Async model can be attributed to the randomized micro-delays (`asyncio.sleep`) implemented to ensure stability, which, when aggregated over 1,700 requests, added a significant sequential penalty to the total time.

2) Reliability and Scalability: Both systems proved highly reliable, though Multithreading achieved a perfect 100% success rate. The slight dip in Async reliability (99.12%) suggests that at high volumes, the event loop requires very precise tuning of the `TCPConnector` and `Semaphore` to avoid occasional socket drops, even in a local environment.

3) Strengths and Limitations:

- **Multithreading:** Proved to be faster in raw execution time for this specific workload. However, the high CPU

usage suggests that scaling this model to tens of thousands of requests would likely lead to system saturation.
- **Asynchronous:** Demonstrated superior CPU efficiency. Its primary limitation in this test was the total duration, which could be improved by further reducing the client-side sleep intervals or increasing the concurrency cap.

4) Conclusion of Comparison: The data suggests that **Multithreading** is highly effective for high-speed, moderate-scale I/O tasks where CPU resources are available. However, for systems where CPU efficiency and extreme scalability are paramount, the **Asynchronous** model remains the theoretical choice, provided the developer optimizes the scheduling delays and handles the complexities of non-blocking I/O flow control.

V. Conclusion and Future Work

A. Conclusion

This research successfully benchmarked the performance of Multithreading and Asynchronous programming for high-volume HTTP operations in Python. By executing 1,700 requests against a controlled Docker environment, we demonstrated that the choice of concurrency model is a significant architectural decision that involves a trade-off between throughput and resource efficiency.

The study's primary insight is that **Multithreading** (via *ThreadPoolExecutor*) offers superior raw speed for moderate workloads, completing the task 137.4% faster than the asynchronous model. This efficiency stems from the ability of threads to utilize multiple CPU cores and the optimized handling of blocking I/O by the *requests* library, which releases the Global Interpreter Lock (GIL) during wait periods.

Conversely, the **Asynchronous** model (*asyncio*) proved to be the more sustainable choice for resource-constrained environments. Despite a slower total execution time—influenced by the implementation of scheduling delays to ensure loop stability—it consumed 62.46% less CPU power than the multithreaded approach. This validates the theory that event loops are significantly lighter on system overhead than OS-level threads, making them ideal for massive scalability where CPU cycles are a limiting factor.

B. Future Work

While this study established a baseline for I/O-bound tasks, several avenues remain for further exploration:

- **Higher Concurrency Limits:** Future research could test the models at an order of magnitude higher (e.g., 10,000+ requests) to identify the exact point where the memory overhead of OS threads causes system instability compared to the event loop.
- **Library Variation:** Comparing different asynchronous libraries, such as *httpx* or *asks*, against *aiohttp* could provide insights into library-specific optimizations.
- **Mixed Workloads:** Testing a "hybrid" workload involving both CPU-bound data processing and I/O-bound web requests would allow for a more complex analysis of the GIL's impact on overall performance.
- **Cloud Infrastructure Latency:** Expanding the test environment from a local Docker cluster to distributed cloud instances (e.g., AWS or GCP) would introduce real-world network jitter and latency, potentially altering the throughput results observed in this controlled setting.

In summary, for developers building high-performance Python applications, Multithreading remains the fastest choice for specialized I/O tasks, while Asynchronous programming is the superior architectural pattern for building lean, highly scalable network services.

Acknowledgment

First, we would like to thank each other for the commitment shown during the duration of this project, and the effort put into making it the best possible.

Secondly, we would like to thank Instituto Politécnico de Santarém, for taking us in and providing all the conditions to make our journey in our Master's degree, as smooth as possible, and providing us with projects like this, where we have free will to investigate, perform our trials and analyse our errors without boundaries, still providing insight when it was needed.

Lastly, but not less important, we would like to extend our thanks to our parents and close family for always having that blind trust in our work and decisions, and that everything will work out in the end, which keeps us motivated in moments when we start to lose focus and faith.

References

[1] A. Troelsen and P. Japikse, "Multithreaded, parallel, and async programming," in *Pro C# 8 with. NET Core 3: Foundational Principles and Practices in Programming*. Springer, 2020, pp. 525–570.

[2] Q. Nguyen, *Mastering Concurrency in Python: Create faster programs using concurrency, asynchronous, multithreading, and parallel programming*. Packt Publishing Ltd, 2018.