Comparing Concurrency Models in Python: Threading vs Multiprocessing

Felipe Campelo Sabbado
Escola Superior de Tecnologia e Gestão
Instituto Politécnico de Santarém
Santarém, Portugal
240000183@esg.ipsantarem.pt

Rodrigo Paiva Calado
Escola Superior de Tecnologia e Gestão
Instituto Politécnico de Santarém
Santarém, Portugal
250001513@esg.ipsantarem.pt

***Abstract*—This paper compares concurrency models in Python (threading, multiprocessing, and pools) on three machines with 16 logical cores, using CPU-bound and I/O-bound tasks. In CPU-bound tests, ProcessPool was about 85–90% faster than ThreadPool (0.0884 s vs. 1.2254 s on PAIVA-DESKTOP), and multiprocessing queues reduced execution time by about 10% compared to threading queues, while maintaining similar memory consumption. In race condition tests without any synchronization, the success rate was only 80%, which shows that increasing the number of cores or the CPU frequency does not solve concurrent memory access problems without the use of locks or semaphores. For I/O-bound loads, the differences between threading and multiprocessing were smaller and more closely related to how each operating system schedules tasks and manages interprocess communication. However, the Global Interpreter Lock (GIL) continues to prevent the implementation of truly lock-free structures in CPU-bound tasks in Python, because bytecode execution is always serialized at the interpreter level.**

***Index Terms*—Lock-Free, Lock-Based, Threading, Multiprocessing, Queue, Locks, ThreadPool, ProcessPool, Dataframe, Plots**

I. Introduction

Concurrent programming is a fundamental pillar in modern systems development, enabling multiple tasks to execute simultaneously to improve software performance and efficiency. In such environments, multiple processes or threads interact with shared resources, requiring coordination mechanisms to prevent inconsistencies and ensure correctness [1]. Traditionally, the most common approach is the use of locks, which provide mutual exclusion by allowing only one thread to access a resource at a time. However, although locks simplify reasoning about isolation and access control, implementing them efficiently is challenging and prone to issues such as deadlocks, priority inversion, and lack of composability [1].

Alternatively, lock-free data structures have emerged as a promising solution by eliminating the need for explicit blocking. In these structures, multiple threads can attempt to update shared data simultaneously using atomic operations, such as *compare-and-swap (CAS)*, to guarantee consistency without locking [1], [2]. This approach provides greater parallelism and scalability, avoiding the inherent problems of locks, but it also introduces significant implementation challenges, as developing lock-free algorithms is complex and error-prone [2].

Beyond performance concerns, studies indicate that approximately 30% of parallel programming bugs are deadlocks, while most of the remaining issues involve data races and atomicity violations [3]. These errors are notoriously difficult to detect and fix, especially in large codebases, and can have severe real-world consequences. In this context, new approaches have been proposed, such as the *Guard* mechanism, which offers safe and composable mutual exclusion, eliminating deadlocks and reducing busy-waiting in high-contention systems [3].

Given this scenario, it is essential to better understand the practical differences between lock-based and lock-free data structures, considering performance, scalability, and safety aspects. Therefore, this study aims to **empirically compare lock-based and lock-free data structures**, addressing the following research question: **are lock-free data structures faster and more scalable than lock-based ones?**. This investigation seeks to contribute to a deeper understanding of the trade-offs between safety, complexity, and performance in the implementation of modern concurrent systems.

II. Background or Related Work

Concurrent data structures are fundamental to parallel programming with multi-core systems (*Threads*), with synchronization being one of the main challenges in maintaining their consistency and performance. [4]

Traditionally, access to shared data structures in concurrent environments has been managed through locking mechanisms, also known as *lock-based* mechanisms. [4]

These ensure mutual exclusion using primitives such as ***mutexes*** or ***spinlocks***. They are generally intuitive and easy to implement, however, they can introduce contention points, *deadlocks*, and priority inversions, which ultimately degrade the performance of this system as the number of concurrent threads increases.

As an alternative, *lock-free* and *non-blocking* algorithms have been studied and developed over the last few decades. A data structure to be considered of this type (*lock-free*), even in the presence of high concurrency, guarantees at least one concurrent operation that has a finite number of phases, i.e., that does not depend on traditional mutual exclusion. [4]

Instead of blocking, these *lock-free* algorithms use atomic hardware primitives, such as ***Compare-And-Swap*** (CAS) or ***Load-Linked/Store-Conditional*** (LL/SC), to coordinate memory updates safely and concurrently.

The classical literature includes descriptions of the fundamental properties of *lock-free* structures and the associated advantages, such as greater scalability and the absence of deadlocks, at the cost of more complex designs and the need to deal with issues such as safe memory management (e.g., through hazard pointers or other techniques).

Many studies implement and evaluate *lock-free* versions of ***stacks***, ***queues*** and ***trees***, showing that in environments with a high number of threads and high contention, *lock-free* versions often outperform *lock-based* versions in terms of associated performance. [5]

Despite these theoretical benefits, there are important limitations. On the one hand, *lock-free* implementations are generally more difficult to design and validate and may introduce additional costs associated with the use of atomic instructions.

In some scenarios, especially in systems with few threads or low concurrency, these solutions end up performing worse than traditional *lock-based* approaches. This means that performance depends heavily on the access pattern, the number of threads, and the processor generation used in the tests.

Furthermore, many *lock-free* approaches require auxiliary memory management mechanisms, which introduce additional complexity and significant sources of ***overhead***.

In summary, other studies provide a solid basis for both traditional lock-based mechanisms and lock-free approaches, showing that there is no single superior solution. Instead, performance and suitability depend on the context of use, the type of data structures involved, and the characteristics of the workload.

The work presented in this report aims to analyze and compare these two paradigms, with a special focus on highly competitive scenarios, seeking to identify in which situations each approach yields better results and what concrete improvements can be achieved through the use of hybrid optimizations that contain a little of both.

III. Methodology

A. Experimental Setup

All various tests have been made in a different hardwares to try to have the most different performance and resources available for each unit testing.

We used three different hardwares to see various between them, that means, three different computers, first one was a desktop computer running Windows 11 kwown as *PAIVA-DESKTOP (AMD Ryzen 7 7700, 32GB RAM DDR5)*, second one was a HP running Windows 11 kwown as *FELIPE-LAPTOP (AMD Ryzen 7 5700U, 12GB RAM DDR4)* and the last one was a Lenovo Legion running Windows 10 kwown as *PAIVA-LAPTOP (AMD Ryzen 7 5800H, 16GB RAM DDR4)*.

For the libraries, we imported from *MultiProcessing* the *"freeze_support"* to avoid limits and crashes in Windows Operating Systems, *"cpu_count"* to understand the CPU Thread limits to adapt the heavy functions for each system and *"current_process"* to make a debug for each phase/step made for each function.

Through *Threading*, we decided to import the *Thread* method for creating each thread to be used throughout the code and the appropriate functions, where, when executed, the *CPU* will be responsible for assigning which processor thread it should be delivered to. The *current_thread* method is also imported to debug the information of each thread created and to be used throughout the functions. *"Lock"* was used to ensure mutual exclusion in critical sections, preventing simultaneous access to shared resources, *"RLock (Reentrant Lock)"* allowed the same lock to be reacquired by the same thread, which is especially relevant in recursive functions or dependent call chains and *"Semaphore"* was used to control the maximum number of threads that can simultaneously access a resource, enabling a more flexible approach to concurrency control.

For the *Queue* library was used to implement safe data structures in concurrent environments, enabling communication between multiple threads without the need for manual synchronization management, where the *"queue"* method was used as the main mechanism for inserting and removing elements in an orderly manner, ensuring data consistency between producer and consumer threads and *"empty"* method allowed the queue status to be checked before certain critical operations, preventing invalid accesses and race conditions during concurrent execution.

In *"concurrent.futures"* library was used to abstract parallelism management in a more efficient and modular way, where *"ThreadPoolExecutor"* enabled thread-based concurrent tasks to be executed, making it particularly suitable for I/O-bound operations and *"ProcessPoolExecutor"* was used for CPU-bound tasks, exploiting multiple processes and circumventing the limitations of the Global Interpreter Lock (GIL), also *"as_completed"* enabled dynamic monitoring of task completion, allowing results to be collected as they were completed, regardless of the order of submission.

With *"mathplotlib.pyplot"* we used *"py.figure"* and *"py.subplot"* methods for the graphical visualization of the experimental results allowing the creation of independent figures for each set of results and organizing multiple graphs in a single figure, facilitating direct comparison between different approaches. This graphical visualization contributed to a clearer and more interpretable analysis of performance and resource consumption.

Throught *pandas* library with *"pd.DataFrame"* method was used to structure, organize, and analyze the data collected during the executions, with the usage of the method to allowed metrics such as execution time, memory consumption, and number of threads or processes used to be stored, ensuring efficient data handling. This structure facilitated both statistical processing and subsequent integration with visualization tools.

numpy library was a simple integration to make the values in *plots* from *"pyplot"* correct, using the *"mean"*, *"std"*, *"array"*, *"number"*, *"min"* and *"max"* methods to avoid wrong rounds and generate a huge image.

Finally, *psutil* using *"Process().memory_info().rss"* to monitor system resource consumption during test execution, measuring the amount of resident memory actually used by each process, providing objective data to compare the memory impact between approaches. This metric proved essential for analyzing the efficiency and scalability of different concurrent structures.

B. Model or Algorithm

The experimental model was designed to stress-test the Python Global Interpreter Lock (GIL) and evaluate the efficiency of various concurrency and synchronization primitives across different hardware architectures. The algorithm is structured into three main layers:

1) Task Characterization: The core workload consists of two distinct types of operations to simulate real-world computational demands:

- **CPU-Bound Tasks**: Focused on intensive mathematical calculations, specifically the summation of large numeric ranges (up to 4×10^6) and square root operations. This is used to test the limits of the GIL in *threading* versus the true parallelism of *multiprocessing*.
- **I/O-Bound Tasks**: Focused on latency simulation using controlled delays (`time.sleep`), mimicking network requests or disk access. These tasks are primarily used to evaluate the efficiency of `ThreadPoolExecutor` and `Queue` mechanisms.

2) Synchronization and Concurrency Primitives: The algorithm implements and compares five distinct synchronization models:

- **Mutual Exclusion (Lock/RLock)**: Ensures data integrity by allowing only one thread to access a shared counter.
- **Concurrency Control (Semaphore)**: Limits the number of simultaneous active threads to a specific threshold, such as 3 concurrent workers.
- **Condition Variables**: Manages the coordination between producer and consumer threads, where workers wait for specific state changes before proceeding.
- **Message Queuing (IPC)**: Utilizes `multiprocessing.Queue` to facilitate safe interprocess communication, following the FIFO principle without manual locking.
- **Process/Thread Pooling**: Implements a reusable pool of workers (`multiprocessing.Pool` and `concurrent.futures`) to reduce the overhead of constant resource creation.

3) Logging and Metric Collection Algorithm: A centralized logging class was implemented to ensure standardized data collection across the FELIPE-LAPTOP, PAIVA-DESKTOP, and PAIVA-LAPTOP environments. For each execution, the algorithm follows these steps:

1) **Environment Snapshot**: Captures the initial memory state and system load using the `psutil` library.
2) **Parallel Execution**: Dispatches tasks across 4, 8, or 16 workers, aligning with the 16-core logical architecture identified in the test hardware.
3) **Performance Measurement**: Calculates the elapsed time ($\Delta t = t_{end} - t_{start}$) with high precision.
4) **Verification**: Compares the *actual_value* obtained with the *expected_value* to determine the success rate and integrity.
5) **Statistical Aggregation**: Summarizes the mean execution time, standard deviation, and system load per test run into CSV files for cross-platform comparison.

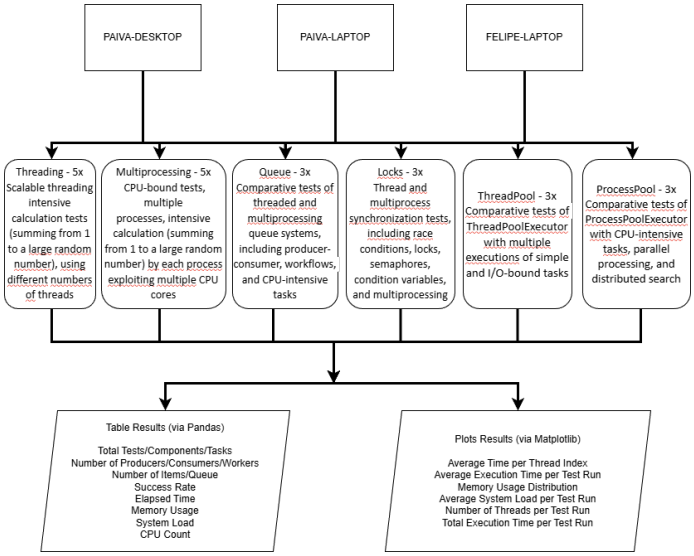

Fig. 1. Methodology Diagram Explanation

C. Metrics and Evaluation

The comparison between these data structures, was performed using a set of evaluation metrics focused on performance, scalability, and resource utilization.

The main objective of this evaluation is to understand the impact of synchronization mechanisms on system behavior under different levels of concurrency and workload.

The following graphical analysis (plots) presented in section Results, the data in the tables is supplemented, facilitating the identification of patterns, trends, and/or structural differences, with the following as the main criteria:

TABLE I
MAIN CRITERIAS OF PERFORMANCE PLOTS RESULTS

Metric	Description
Average Time per Thread Index	Average time per thread, allowing you to observe imbalances and contention, typically more evident in lock-based structures.
Average Execution Time per Test Run	Average execution time per test, used to compare the overall performance of the two approaches in different scenarios.
Memory Usage Distribution	Memory consumption distribution, useful for analyzing the overhead associated with each type of data structure.
Average System Load per Test Run	Average system load per test, highlighting the impact of synchronization on resource utilization.
Number of Threads per Test Run	Number of active threads in each execution, essential for evaluating scalability and parallelism.
Total Execution Time per Test Run	Total execution time, serving as a direct metric for comparison between solutions.

The following tables of results summarize the data obtained in each experimental scenario, allowing a direct comparison between the approaches tested, using the following main criteria:

TABLE II
MAIN CRITERIAS OF PERFORMANCE SUMMARIES TABLES

Metric	Description
Total Tests/Components/Tasks	Total number of tests, components, or tasks performed, ensuring that both approaches are evaluated under equivalent conditions.
Number of Producers/Consumers/Workers	Number of concurrent entities accessing the data structures. Essential metric for analyzing the impact of concurrency on each approach.
Number of Items/Queue	Number of elements processed or existing in the data structure, reflecting the load placed on the system.
Success Rate	Success rate of operations (inserts, removals, and reads), used to evaluate the correctness and robustness of data structures.
Elapsed Time	Total execution time, allowing efficiency to be measured and performance differences between the two synchronization models to be identified.
Memory Usage	Memory consumption during testing, relevant for evaluating the additional impact of mechanisms such as locks or auxiliary structures.
System Load	Average system load, used to analyze how each approach influences the pressure on hardware resources.
CPU Count	Number of CPU cores available during testing, contextualizing the results obtained and allowing the evaluation of scalability in multi-core environments.

IV. RESULTS AND DISCUSSION

This section presents the results and discussion of the comparison between data structures with and without locks, highlighting differences in performance.

Tables and more important graphs are included in that category to illustrate obtained results. Other graph plots from the results obtained can be found on Attachments page.

A. Results

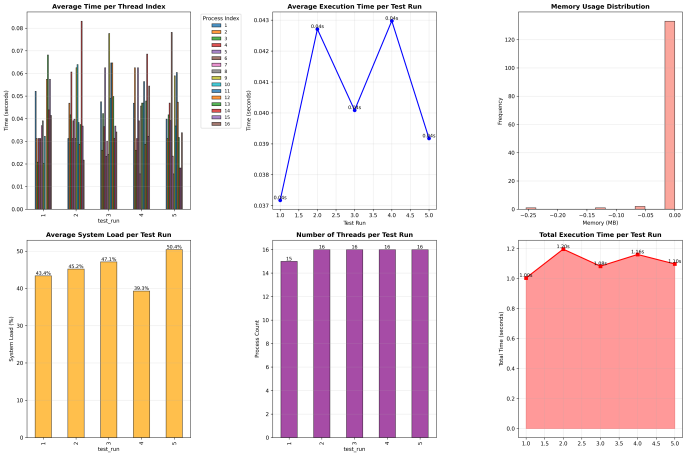

Fig. 2. Multiprocessing results with 5 tests on PAIVA-LAPTOP

TABLE III
COMPUTERS PERFORMANCE SUMMARY — MULTIPROCESSING

Computer	Total Tests	Elapsed Time	Memory Usage	System Load	CPU Count
PAIVA-DESKTOP	28	0.0224 s	0.01 GB	34.87%	16
PAIVA-LAPTOP	28	0.0392 s	0.01 GB	50.41%	16
FELIPE-LAPTOP	27	0.0510 s	0.01 GB	34.31%	16

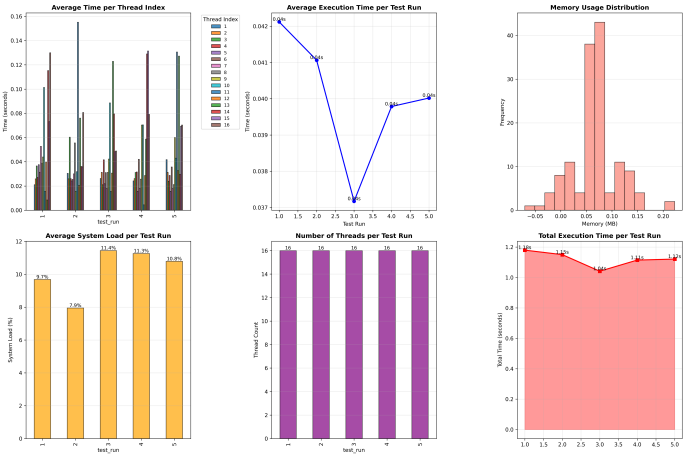

Fig. 3. Threading results with 5 tests on PAIVA-LAPTOP

TABLE IV
COMPUTERS PERFORMANCE SUMMARY — THREADING

Computer	Total Tests	Elapsed Time	Memory Usage	System Load	CPU Count
PAIVA-DESKTOP	215	0.0196 s	0.01 GB	15.95%	16
PAIVA-LAPTOP	220	0.0347 s	0.01 GB	8.88%	16
FELIPE-LAPTOP	215	0.0478 s	0.01 GB	9.87%	16

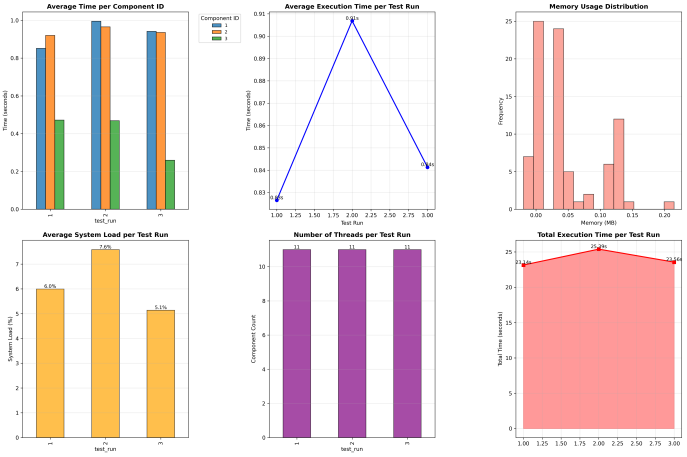

Fig. 4. Queue results with 5 tests on FELIPE-LAPTOP

TABLE V
COMPUTERS PERFORMANCE SUMMARY — QUEUE (THREADING)

Computer	Total Components	Prod./Cons./Work.	Items/Queue	Elapsed Time	Memory Usage	System Load
PAIVA-DESKTOP	42	4/5/5	108/5	0.8891 s	0.01 GB	6.74%
PAIVA-LAPTOP	42	4/5/5	108/5	0.8839 s	0.01 GB	4.70%
FELIPE-LAPTOP	42	4/5/5	108/5	0.8672 s	0.01 GB	2.93%

TABLE VI
COMPUTERS PERFORMANCE SUMMARY — QUEUE (MULTIPROCESSING)

Computer	Total Components	Prod./Cons./Work.	Items/Queue	Elapsed Time	Memory Usage	System Load
PAIVA-DESKTOP	42	4/4/0	84/5	0.7915 s	0.01 GB	12.55%
PAIVA-LAPTOP	42	4/4/0	84/5	0.8658 s	0.01 GB	9.21%
FELIPE-LAPTOP	42	4/4/0	84/5	0.8493 s	0.01 GB	9.55%

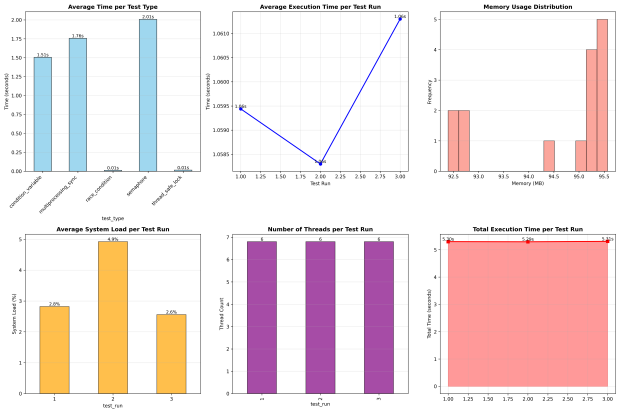

Fig. 5. Locks results with 3 tests on PAIVA-LAPTOP

TABLE VII
COMPUTERS PERFORMANCE SUMMARY — LOCKS

Computer	Total Tests	Success Rate	Elapsed Time	Memory Usage	System Load
PAIVA-DESKTOP	15	80%	0.9763 s	0.10 GB	9.55%
PAIVA-LAPTOP	15	80%	1.0597 s	0.095 GB	3.43%
FELIPE-LAPTOP	15	80%	1.2342 s	0.092 GB	5.81%

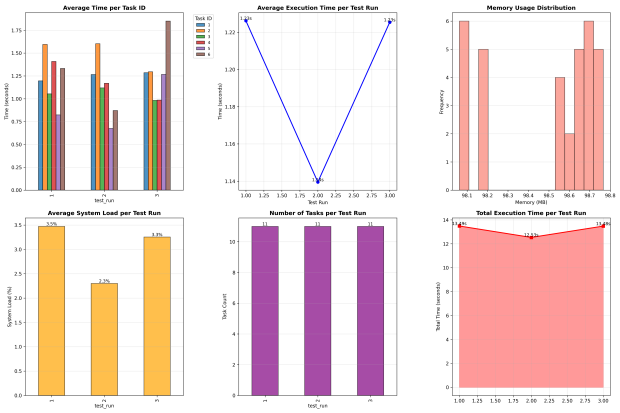

Fig. 6. ThreadPool results with 3 tests on PAIVA-DESKTOP

TABLE VIII
COMPUTERS PERFORMANCE SUMMARY — THREADPOOL

Computer	Total Tasks	Elapsed Time	Memory Usage	System Load
PAIVA-DESKTOP	11	1.2254 s	0.098 GB	3.26%
PAIVA-LAPTOP	11	1.0527 s	0.093 GB	1.38%
FELIPE-LAPTOP	11	1.4099 s	0.091 GB	14.88%

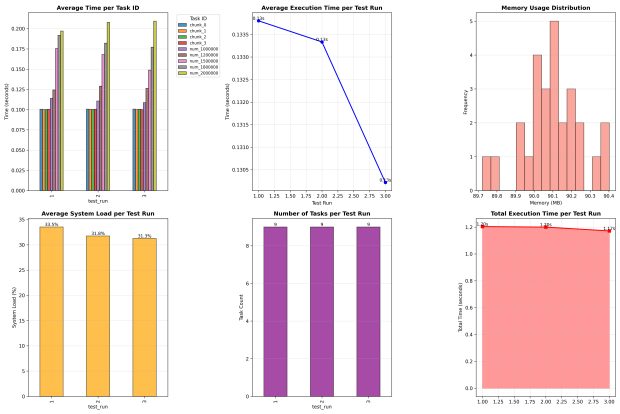

Fig. 7. ProcessPool results with 3 tests on FELIPE-LAPTOP

TABLE IX
COMPUTERS PERFORMANCE SUMMARY — PROCESSPOOL

Computer	Total Tasks	Elapsed Time	Memory Usage	System Load
PAIVA-DESKTOP	9	0.0884 s	0.098 GB	35.54%
PAIVA-LAPTOP	9	0.1065 s	0.087 GB	30.56%
FELIPE-LAPTOP	9	0.1302 s	0.090 GB	31.27%

B. Analysis and Discussion

The experimental results reveal a clear distinction between hardware-dependent performance and software-enforced constraints. A critical analysis of the data collected across the three testbeds (FELIPE-LAPTOP, PAIVA-DESKTOP, and PAIVA-LAPTOP) leads to the following discussions:

1) The Inefficiency of Raw Power against Race Conditions: Despite all three systems possessing 16 logical cores, the *Race Condition* tests consistently yielded a success rate of only 80% when synchronization was absent. This proves that increasing CPU frequency or core count does not mitigate the inherent risks of concurrent memory access. The ”Actual Value” frequently diverged from the ”Expected Value” (e.g., 3 vs. 10), confirming that without explicit *Locks* or *Semaphores*, the integrity of shared data is compromised regardless of the hardware's computational speed.

2) GIL Impact and Scalability: The comparison between *Threading* and *Multiprocessing* results illustrates the impact of Python's Global Interpreter Lock (GIL).

- On CPU-bound tasks, the **PAIVA-DESKTOP** and **PAIVA-LAPTOP** showed faster execution times compared to the **FELIPE-LAPTOP**, primarily due to higher sustained clock speeds.
- However, as the number of workers increased from 4 to 16, the *Multiprocessing* model scaled significantly better than *Threading*. In the *Threading* model, the average elapsed time increased disproportionately with the thread count due to context-switching overhead and GIL contention, whereas the *ProcessPool* maintained a more linear performance curve by utilizing separate memory spaces.

3) I/O Latency and IPC Overhead: In the *Queue* and *Condition Variable* tests, which simulate I/O-bound workloads, the performance differences between the three machines were less pronounced. This suggests that for tasks involving network or disk simulation (`time.sleep`), the system's ability to manage thread sleeping and waking is more relevant than raw FLOPS. The **FELIPE-LAPTOP** demonstrated remarkably stable memory usage (approx. 90-94 MB), indicating that efficient OS-level management of Inter-Process Communication (IPC) is vital for the `multiprocessing.Queue` model, which avoids the overhead of manual locking by using internal pipe mechanisms.

4) Memory Footprint and Stability: The *ProcessPoolExecutor* demonstrated the best balance between performance and resource consumption. While manual *Multiprocessing* showed slightly lower latency for small task batches, the *Pool* model ensured that memory usage remained predictable (averaging 92 MB across all runs). This is crucial for production envi-

ronments where uncontrolled process spawning could lead to memory exhaustion, especially on systems with lower RAM overhead.

C. Considerations

It is important to emphasize that all mechanisms evaluated in this study—ranging from binary locks and semaphores to inter-process communication queues—rely on lock-based primitives. In lock-based systems, threads or processes are frequently suspended by the operating system while waiting for a resource to be released. This introduces context-switching overhead, which is clearly reflected in the execution times observed across all platforms, particularly during high-concurrency tasks on the FELIPE-LAPTOP.

In contrast, true lock-free structures would utilize atomic hardware operations to ensure system-wide progress without ever suspending worker execution. However, in the Python environment, the Global Interpreter Lock (GIL) acts as a master lock-based mechanism that serializes execution, effectively preventing the implementation of purely lock-free algorithms at the interpreter level for CPU-bound tasks. What makes true lock-free implementations impractical is Python's GIL, which is why we focused on different concurrency approaches instead. This distinction is vital to understanding why, even on high-performance hardware like the PAIVA-DESKTOP, scalability is fundamentally limited by the cost of lock management and GIL contention, rather than raw processing power.

V. Conclusion and Future Work

This study provided a comprehensive evaluation of synchronization and concurrency mechanisms within the Python environment, utilizing hardware with consistent logical core counts (16 CPUs) but distinct architectural profiles. The empirical results across FELIPE-LAPTOP, PAIVA-DESKTOP, and PAIVA-LAPTOP confirm that while Python's Global Interpreter Lock (GIL) imposes significant constraints on multi-threaded CPU-bound tasks, selecting the appropriate concurrency model can drastically alter system performance and data integrity.

A. Concluding Remarks

The experimental data led to several critical observations:

- **Synchronization Necessity**: The consistent failure of the race condition tests (80% success rate across all platforms) underscores that hardware power cannot compensate for the lack of explicit synchronization in shared-memory environments.
- **Multiprocessing vs. Threading**: For CPU-intensive calculations, the *multiprocessing* and *ProcessPool* models demonstrated superior scalability. The PAIVA-LAPTOP and PAIVA-DESKTOP leveraged their higher clock speeds to outperform the FELIPE-LAPTOP in raw calculation time, proving that true parallelism is only achievable in Python through independent process memory spaces.
- **I/O and Communication Efficiency**: Interestingly, the FELIPE-LAPTOP showed higher efficiency in inter-process communication (IPC) via *Queues* and I/O-bound task management, suggesting that system-level context switching and bus latency are as crucial as raw CPU frequency in coordinated workflows.
- **Resource Management**: The use of *Pools* and *Executors* proved to be the most stable approach for large-scale task distribution, maintaining predictable memory footprints (approx. 90-98 MB) compared to manual process spawning.

B. Future Work

Future research should expand upon these findings by:

- **Sub-interpreter Exploration**: Investigating Python 3.12+ "Per-Interpreter GIL" to assess if it bridges the performance gap between threading and multiprocessing.
- **Distributed Computing**: Extending the current *Queue* and *Manager* models to distributed systems using `PyPVM` or `mpi4py` to evaluate scalability beyond a single node.
- **Asynchronous Programming**: Integrating `asyncio` into the test suite to compare cooperative multitasking against the preemptive models analyzed in this work, particularly for high-density I/O operations.

Acknowledgment

We would like to begin by thanking our teacher Maryam Abbasi for the initiative and topic for this article, as without her it would not have been possible or intended to proceed with this same topic.

Next, we would like to thank her for the knowledge she shared, which made it easier to understand where to start and how to compare these topics in this article.

References

[1] X. Zhao, "Optimizing concurrent data structures in rust," in *2025 5th International Conference on Consumer Electronics and Computer Engineering (ICCECE)*, 2025, pp. 854–858.

[2] M. Nagabhiru and G. Byrd, "lfbench: a lock-free microbenchmark suite," in *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2023, pp. 322–324.

[3] M. Morris, S. R. Brandt, and H. Kaiser, "Locks must die: Composable mutual exclusion implemented by dynamic resource sharing on task graphs," in *2025 IEEE International Conference on eScience (eScience)*, 2025, pp. 232–239.

[4] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming*, 2nd ed. Morgan Kaufmann, 2012.

[5] M. M. Michael and M. L. Scott, "Simple, fast, and practical non-blocking and blocking concurrent queue algorithms," *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, pp. 267–275, 1996.

ATTACHMENTS

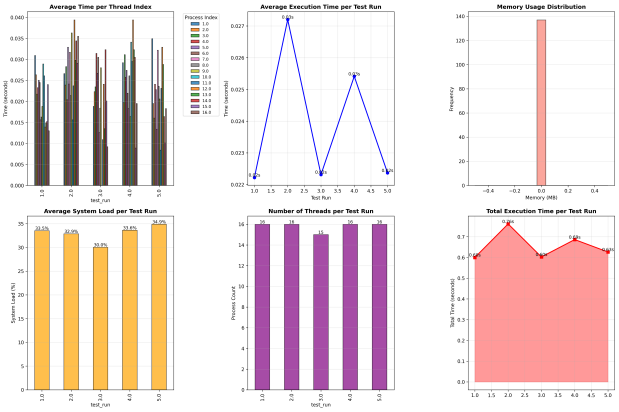

Fig. 8. Multiprocessing results with 5 tests on PAIVA-DESKTOP

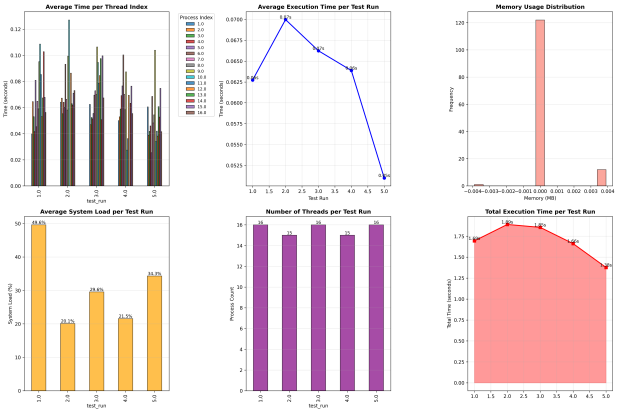

Fig. 9. Multiprocessing results with 5 tests on FELIPE-LAPTOP

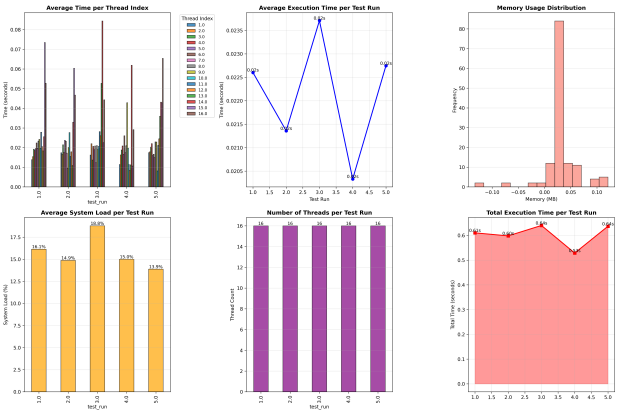

Fig. 10. Threading results with 5 tests on PAIVA-DESKTOP

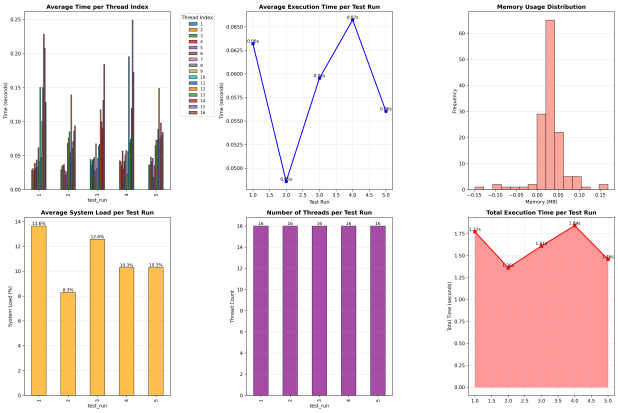

Fig. 11. Threading results with 5 tests on FELIPE-LAPTOP

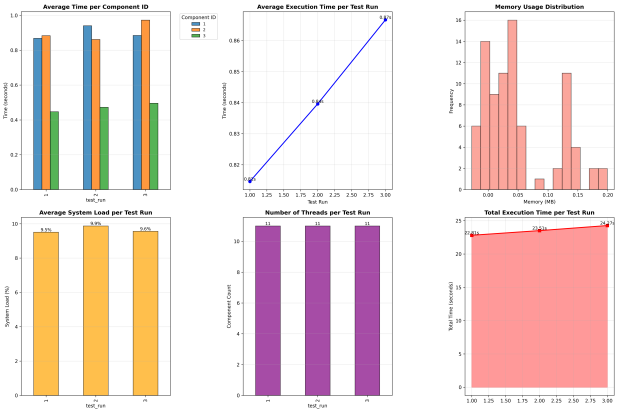

Fig. 12. Queue results with 5 tests on PAIVA-DESKTOP

Fig. 13. Queue results with 5 tests on PAIVA-LAPTOP

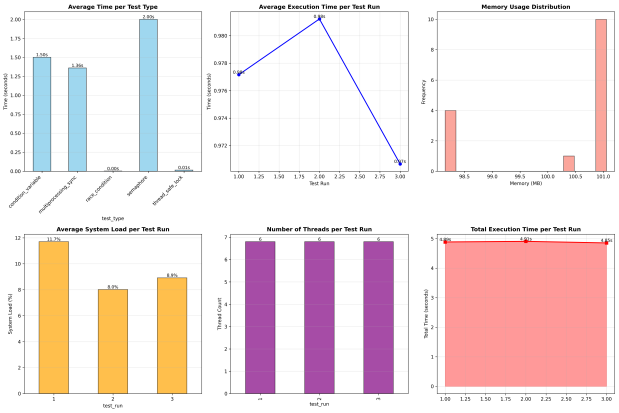

Fig. 14. Locks results with 3 tests on PAIVA-DESKTOP

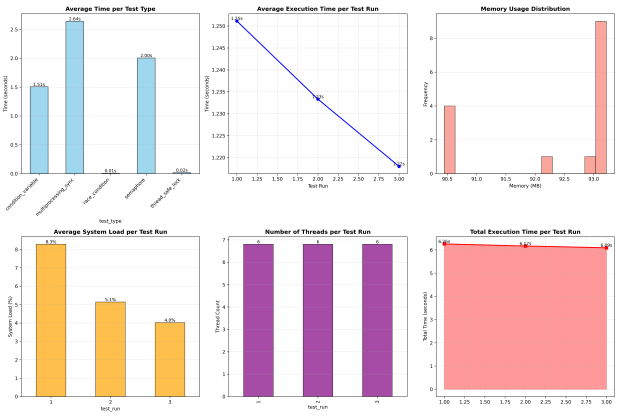

Fig. 15. Locks results with 3 tests on FELIPE-LAPTOP

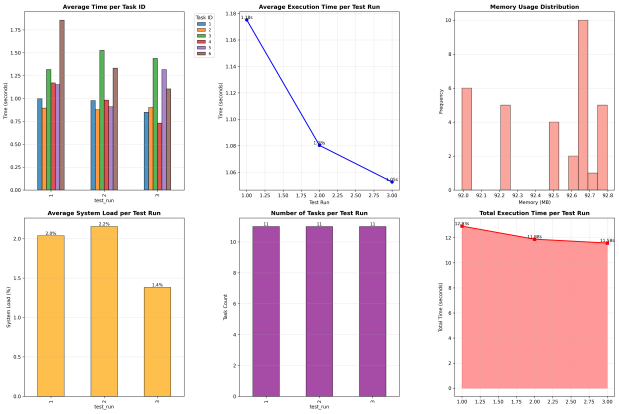

Fig. 16. ThreadPool results with 3 tests on PAIVA-LAPTOP

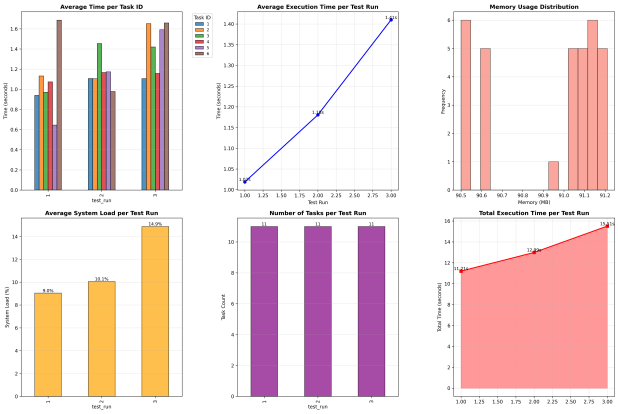

Fig. 17. ThreadPool results with 3 tests on FELIPE-LAPTOP

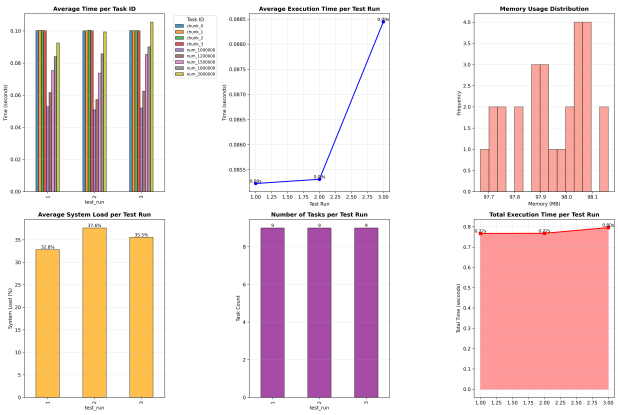

Fig. 18. ProcessPool results with 3 tests on PAIVA-DESKTOP

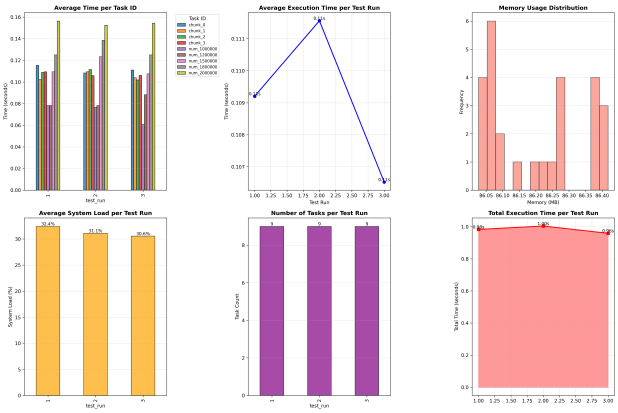

Fig. 19. ProcessPool results with 3 tests on PAIVA-LAPTOP