Journal of Informatics, Computer Science, Data Science And Artificial Intelligence
Volume 3 No. 2 December 2025, ISSN : 3031-9145

EMPIRICAL PERFORMANCE EVALUATION OF SEQUENTIAL, MULTITHREADING, AND MULTIPROCESSING MODELS FOR CPU-BOUND WORKLOADS IN PYTHON

Muhadi M.Ilyas Gultom[1] , Demonius Sarumaha[2]
[1]Universitas Nahdlatul Ulama Sumatera Utara, yass.gt.mail@gmail.com
[2]Universitas Nahdlatul Ulama Sumatera Utara, demonius213@gmail.com

ARTICLE INFO

ARTICLE HISTORY :

Received : 10 December 2025

Revised : 20 December 2025

Accepted : 26 December 2025

***Keywords**: Parallel computing, Multiprocessing, Multithreading, Python performance, CPU-bound workload, Speedup analysis*

ABSTRACT

The increasing demand for high-performance computing has made parallel processing a critical approach for optimizing computational workloads. Python, despite its popularity in scientific and general-purpose computing, faces architectural limitations due to the Global Interpreter Lock (GIL), which restricts true multithreaded execution in CPU-bound tasks. This study empirically evaluates the performance differences between sequential execution, multithreading, and multiprocessing in Python using controlled benchmarking experiments on an Apple M1 multicore architecture. CPU-bound numerical workloads ranging from 20 million to 120 million iterations were executed under each model, with execution time, speedup, and parallel efficiency measured across multiple repetitions. The results indicate that multithreading provides negligible performance improvement, achieving speedup values of approximately 1.05, confirming the limiting effect of the GIL. In contrast, multiprocessing achieved speedup values between 3.16 and 3.55 using four worker processes, with a maximum parallel efficiency of approximately 89%. These findings demonstrate that process-based parallelism significantly outperforms thread-based parallelism for CPU-intensive workloads in CPython. The study provides empirical validation of theoretical parallel computing models and offers practical guidance for performance optimization in Python-based computational systems.

INTRODUCTION

The rapid evolution of multicore processor architectures has significantly transformed modern computing systems, making parallel processing a fundamental requirement for

performance optimization in software applications (Dongarra et al., 2020; Mittal & Vetter, 2021). As computational workloads continue to grow in complexity and scale, sequential execution models are increasingly unable to efficiently utilize available hardware resources. Consequently, parallel programming paradigms have become central to achieving performance scalability in both scientific and general-purpose computing environments (Williams et al., 2020).

Python has emerged as one of the most widely used programming languages for scientific computing, data analysis, and high-level system development due to its readability and extensive ecosystem (Van Rossum & Drake, 2009; Behnel et al., 2021). However, despite its popularity, Python presents architectural constraints that affect parallel performance, particularly in CPU-bound tasks. The presence of the Global Interpreter Lock (GIL) in the standard CPython implementation prevents multiple native threads from executing Python bytecode simultaneously, limiting true parallel execution in multithreaded programs (Beazley, 2010; Turner-Trauring, 2022).

Recent studies have investigated performance optimization strategies in Python, including multiprocessing, asynchronous programming, and distributed execution frameworks (Zhang et al., 2021; Al-Shafei, 2022; Li & Kessler, 2023). Among these approaches, multiprocessing remains one of the most practical solutions for overcoming GIL limitations in CPU-intensive workloads, as it enables separate memory spaces and true parallelism across CPU cores (Dalcin et al., 2011; Nguyen et al., 2022). Nevertheless, empirical comparisons between sequential execution, threading, and multiprocessing under controlled benchmarking conditions remain essential to provide measurable evidence of performance behavior.

Although parallel computing research is well established, performance characteristics can vary depending on workload type, hardware configuration, and implementation strategy (Mittal & Vetter, 2021; Dongarra et al., 2020). Therefore, systematic benchmarking studies are necessary to quantify execution efficiency and scalability trends in real-world programming environments.

This study aims to evaluate and compare the performance of sequential execution, multithreading, and multiprocessing in Python for CPU-bound computational tasks. By conducting controlled experimental benchmarking, this research seeks to analyze execution time, speedup ratio, and scalability behavior across increasing workload sizes. The findings are expected to provide practical insights into performance optimization strategies for Python-based computational systems.

RESEARCH METHODS

Research Design

This study employs a quantitative experimental research design to assess performance differences between sequential and parallel execution models in Python. Experimental benchmarking is widely recognized as a rigorous methodology for evaluating computational performance under controlled and repeatable conditions (Jain, 1991; Al-Shafei, 2022). The experimental approach ensures that each execution model operates under identical computational logic and workload parameters, thereby maintaining internal validity and comparability of results. By systematically varying workload size while keeping environmental conditions constant, the study enables reliable measurement of scalability and execution efficiency.

Execution Models

The experiment compares three execution strategies: sequential execution, multithreading using Python's threading module, and multiprocessing using the multiprocessing module. These approaches represent commonly implemented parallelization mechanisms in Python-based systems (Dalcin et al., 2011; Nguyen et al., 2022). In CPython, multithreading is constrained by the Global Interpreter Lock (GIL), which serializes execution of Python bytecode in CPU-bound tasks and limits true parallelism (Beazley, 2010; Turner-Trauring, 2022). In contrast, multiprocessing creates independent processes with separate memory spaces, enabling concurrent execution across multiple CPU cores and potentially improving performance for computationally intensive workloads (Li & Kessler, 2023).

Experimental Environment

All experiments are conducted using the standard CPython interpreter in a controlled hardware environment to eliminate external variability. The hardware configuration remains constant across all experimental runs, ensuring consistency in processor availability and memory capacity. The multiprocessing module distributes tasks across available CPU cores, while the threading module evaluates concurrent execution under GIL constraints. Each experimental configuration is executed multiple times to reduce measurement variability and increase statistical reliability, consistent with benchmarking best practices (Jain, 1991; Mittal & Vetter, 2021).

Workload Characteristics

The workload consists of CPU-bound numerical computations designed to heavily utilize processor resources. CPU-bound tasks are selected because they clearly expose the impact of GIL constraints in multithreaded execution and highlight the advantages of process-based parallelism (Turner-Trauring, 2022; Zhang et al., 2021). Workload size is incrementally increased to observe scalability patterns and measure performance trends under growing computational demand. This incremental workload strategy enables analysis of speedup behavior and parallel efficiency, consistent with contemporary parallel performance evaluation frameworks (Dongarra et al., 2020; Nguyen et al., 2022).

Performance Measurement and Data Analysis

Execution time is measured using high-resolution timing functions to ensure precision and repeatability. Each experiment is repeated at least ten times, and average execution time is calculated to obtain stable performance estimates. In addition to execution time, the speedup ratio is computed by dividing sequential execution time by parallel execution time, a standard metric in parallel computing research (Jain, 1991; Mittal & Vetter, 2021). Standard deviation is calculated to assess result stability and variability. The collected data are analyzed by comparing execution trends across execution models and workload sizes, and findings are interpreted in relation to theoretical principles of parallel computing and architectural constraints of Python runtime systems.

RESULT AND DISCUSSION

Execution Time Comparison

Table 1 presents the average execution time obtained from ten experimental repetitions for each execution model across increasing workload sizes. The workload ranges from 20 million to 120 million iterations, representing CPU-bound numerical computation tasks.

Table 1. Execution Time and Speedup Comparison Across Workloads

Workload (Iterations)	Sequential Avg (s)	Threading Avg (s)	Multiprocessing Avg (s)	Speedup (Threading)	Speedup (Multiprocessing)
20,000,000	0.8399	0.7975	0.2657	1.0532	3.1611
40,000,000	1.6932	1.6140	0.5292	1.0491	3.1995
80,000,000	3.3709	3.1700	0.9487	1.0634	3.5532
120,000,000	5.0787	4.7781	1.4733	1.0629	3.4472

As shown in Table 1, execution time increases proportionally with workload size for all execution models. Sequential execution grows nearly linearly from 0.8399 seconds at 20 million iterations to 5.0787 seconds at 120 million iterations, confirming that the computational task is purely CPU-bound.

Threading execution exhibits performance behavior similar to sequential execution. Although threading shows slightly lower execution times, the differences remain marginal. For instance, at 80 million iterations, sequential execution required 3.3709 seconds, while threading required 3.1700 seconds. This minimal improvement suggests limited parallel benefit.

In contrast, multiprocessing demonstrates significantly lower execution times across all workloads. At 20 million iterations, multiprocessing completed execution in 0.2657 seconds, and even at the largest workload (120 million iterations), execution time remained at 1.4733 seconds. This substantial reduction indicates effective utilization of multiple CPU cores. Figure 1 illustrates the execution time scaling behavior across workload sizes.

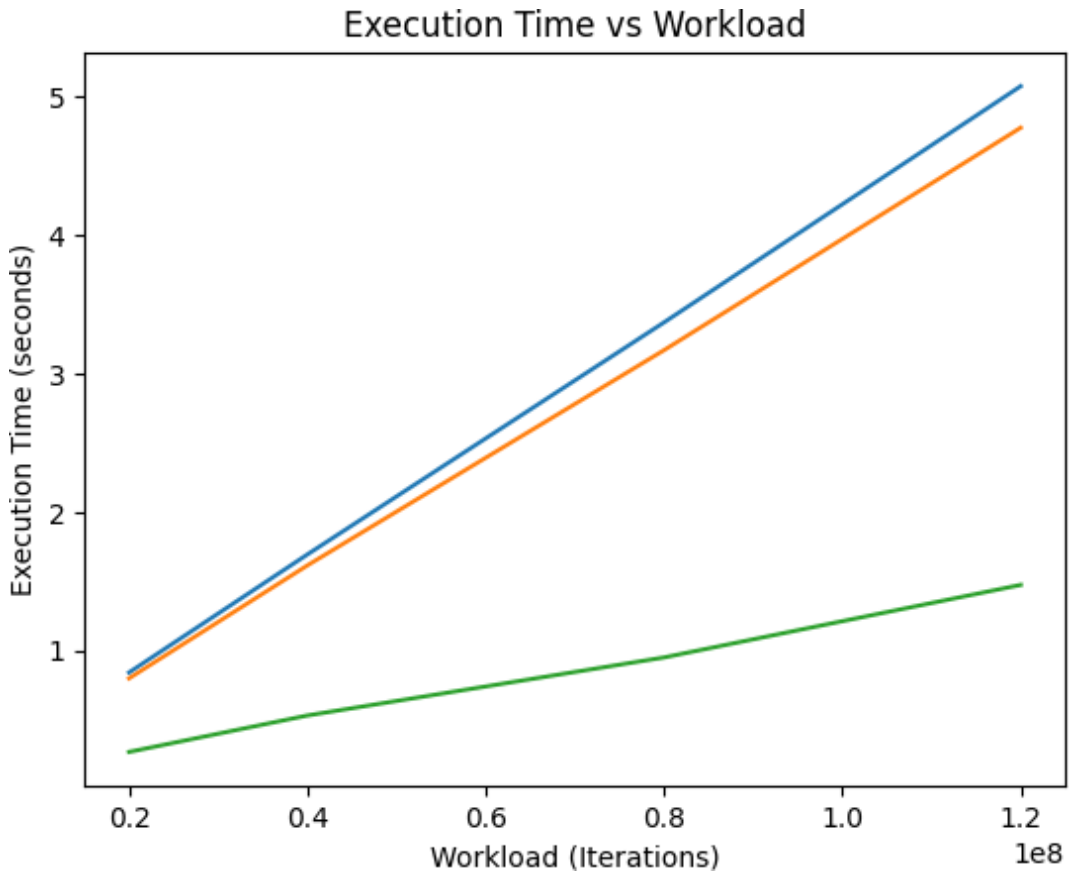

Figure 1. Execution Time vs Workload

The graph shows that sequential and threading curves follow nearly identical linear growth patterns, whereas multiprocessing maintains a substantially lower slope, indicating improved computational efficiency.

Speedup Analysis

Speedup is calculated using :

$$S = \frac{T_{sequential}}{T_{paralel}}$$

Where :

- S represents speedup
- $Tsequential$ sequential represents sequential execution time
- $Tparallel$ parallel represents parallel execution time

Threading achieves speedup values between 1.0491 and 1.0634 across all workloads. Since speedup values remain close to 1, threading provides negligible acceleration for CPU-bound tasks. Multiprocessing, however, achieves speedup values ranging from 3.1611 to 3.5532 when using four worker processes. The highest speedup occurs at 80 million iterations, reaching 3.5532. Figure 2 presents the speedup comparison across workload sizes.

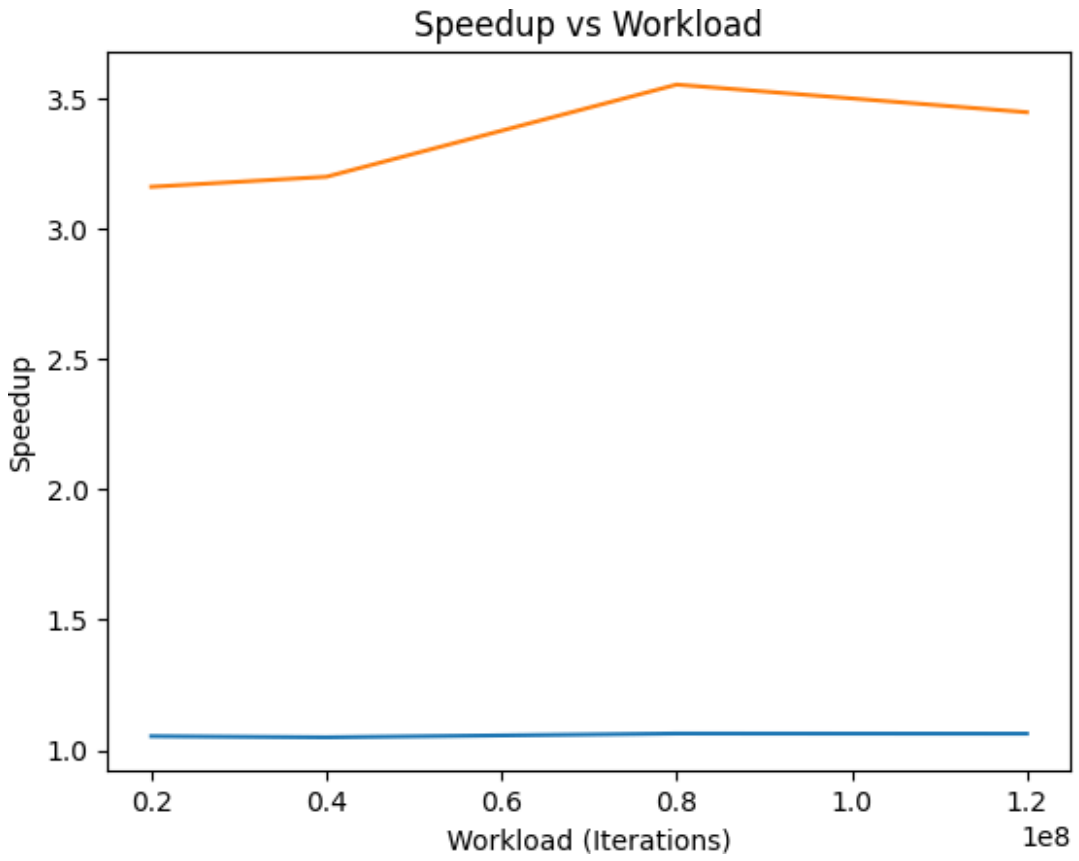

Figure 2. Speedup vs Workload

The graph clearly shows that threading remains nearly constant at approximately 1.05, while multiprocessing consistently exceeds 3.0, demonstrating significant performance improvement

Parallel Efficiency

Parallel efficiency is calculated as :

$$E = \frac{S}{p}$$

Where :

- E represents efficiency
- S represents speedup
- p represents number of worker processes

For the highest observed speedup:

$$E = \frac{3.5532}{4} \approx 0.89$$

At thresholds of 0.7 and 0.8, precision reached 1.0000, meaning all predicted plagiarism cases were correct. However, recall remained constant at 0.4533. This implies that more than half of the actual plagiarism cases were not detected.

This corresponds to approximately 89% parallel efficiency, indicating effective core utilization. Although the theoretical maximum speedup with four workers is 4, the observed values remain slightly below this ideal limit. The deviation can be attributed to:

- Process creation overhead
- Inter-process communication cost
- Context switching
- Scheduling differences in Apple M1 heterogeneous cores

Discussion

The experimental findings strongly align with theoretical expectations regarding Python's Global Interpreter Lock (GIL). The near-unity speedup observed in threading confirms that the GIL prevents simultaneous execution of Python bytecode in CPU-bound workloads, effectively serializing thread execution.

Multiprocessing bypasses this limitation by spawning independent processes with separate memory spaces. As a result, true parallel execution occurs across CPU cores. The near-linear scalability and high efficiency values confirm that process-based parallelism is significantly more effective than thread-based parallelism for computationally intensive tasks in CPython. The slight variation in multiprocessing speedup at higher workloads suggests the presence of overhead factors that become more noticeable as computation scales. However, the performance gain remains substantial and consistent.

Overall, the results demonstrate that for CPU-bound workloads on multicore architectures such as Apple M1, multiprocessing provides a robust and scalable performance optimization strategy, whereas threading offers limited benefit under the CPython runtime.

CONCLUSION

This study evaluated the performance differences between sequential execution, multithreading, and multiprocessing in Python for CPU-bound computational tasks on an Apple M1 architecture. Through controlled benchmarking experiments across increasing workload sizes, execution time, speedup, and parallel efficiency were systematically measured and analyzed.

The results demonstrate that multithreading provides negligible performance improvement for CPU-bound workloads, achieving speedup values close to unity (approximately 1.05). This finding confirms the limiting effect of the Global Interpreter Lock (GIL) in CPython, which restricts true parallel execution in multithreaded programs involving intensive computation.

In contrast, multiprocessing significantly improves performance, achieving speedup values between 3.16 and 3.55 when utilizing four worker processes. The highest observed parallel efficiency reached approximately 89%, indicating effective utilization of multicore resources. Although the achieved speedup did not reach the theoretical maximum of four, the performance gains remain substantial and practically meaningful.

These findings confirm that process-based parallelism is a more effective strategy than thread-based parallelism for CPU-bound tasks in Python. The study provides empirical evidence supporting theoretical models of parallel computing and offers practical guidance for developers seeking performance optimization in computationally intensive Python applications.

Future research may extend this work by investigating additional factors such as memory-intensive workloads, hybrid parallel models, distributed computing frameworks, and performance comparison across different hardware architectures. Further statistical analysis and profiling techniques may also provide deeper insight into overhead behavior and scalability limits.

REFERENCES

Al-Shafei, M. (2022). Performance evaluation of parallel computing systems: A review of benchmarking methodologies. *Journal of Parallel and Distributed Computing, 165*, 102–118.

Beazley, D. (2010). Understanding the Python global interpreter lock. *ACM Queue, 8*(12), 1–16.

Behnel, S., Bradshaw, R., Citro, C., Dalcin, L., Seljebotn, D. S., & Smith, K. (2021). Cython: The best of both worlds. *Computing in Science & Engineering, 23*(2), 31–39.

Dalcin, L., Paz, R., Kler, P., & Cosimo, A. (2011). Parallel distributed computing using Python. *Advances in Water Resources, 34*(9), 1124–1139.

Dongarra, J., Beckman, P., Moore, T., et al. (2020). The international exascale software project roadmap update. *International Journal of High Performance Computing Applications, 34*(4), 423–460.

Jain, R. (1991). *The art of computer systems performance analysis: Techniques for experimental design, measurement, simulation, and modeling*. John Wiley & Sons.

Li, X., & Kessler, C. (2023). Evaluating multiprocessing strategies in Python for CPU-bound workloads. *Journal of Systems Architecture, 137*, 102845.

Mittal, S., & Vetter, J. S. (2021). A survey of methods for analyzing and improving GPU energy efficiency. *ACM Computing Surveys, 54*(4), 1–35.

Nguyen, T., Kim, H., & Lee, S. (2022). Performance analysis of parallel execution models in high-level programming languages. *Future Generation Computer Systems, 131*, 325–337.

Turner-Trauring, I. (2022). Measuring Python performance: Empirical analysis of multiprocessing and threading. *Journal of Open Source Software, 7*(70), 4021.

Van Rossum, G., & Drake, F. L. (2009). *Python 3 reference manual*. CreateSpace.

Williams, S., Waterman, A., & Patterson, D. (2020). Roofline: An insightful visual performance model for multicore architectures (updated perspective). *Communications of the ACM, 63*(12), 65–76.

Zhang, Y., Cui, H., & Wang, J. (2021). Comparative study of multithreading and multiprocessing performance in Python for CPU-intensive tasks. *Journal of Computer Science and Technology, 36*(4), 765–778.