Sinkron : Jurnal dan Penelitian Teknik Informatika
Volume 10, Number 3, July 2026
DOI : https://doi.org/10.33395/sinkron.v10i3.16262
e-ISSN : 2541-2019
p-ISSN : 2541-044X

Comparative Scalability Analysis of Python Multiprocessing and OpenMP on Windows and WSL2

Achmad Fauzan[1)*], Agung Purwo Wicaksono[2)], Elindra Ambar Pambudi[3)]
[1,2,3)] Department of Informatics Engineering, Faculty of Engineering and Science, Universitas Muhammadiyah Purwokerto, Indonesia
[1)]achmadfauzan@ump.ac.id, [2)] wicaksono@ump.ac.id, [3)] elindraambarpambudi@ump.ac.id

Submitted :May 21, 2026 | **Accepted** : May 30, 2026 | **Published** : July 5, 2026

Abstract: The advent of multicore processors has made efficient parallel programming models increasingly important for CPU-bound workloads. Process-based and thread-based parallelism in native Windows and WSL2 has not been comprehensively compared, despite extensive research on Python multiprocessing, OpenMP, and Windows Subsystem for Linux 2 (WSL2). This paper presents a unified multicore scalability assessment of Python multiprocessing and OpenMP, based on a CPU-intensive floating-point benchmark. Experiments were performed on an Intel Core i9-9900 processor with 8 physical cores and 16 logical threads in different process and thread configurations. Performance was evaluated in terms of execution time, speedup, parallel efficiency, CPU utilization, and statistical reliability metrics from ten repeated executions. Results demonstrate OpenMP performance advantage over Python multiprocessing for all tested configurations. OpenMP with 16 execution units provided maximum speedups of 6.482 on Windows and 7.065 on WSL2, compared to 5.624 and 5.790 for Python multiprocessing. The highest CPU utilization was achieved by OpenMP on WSL2 (97.31%). The reliability analysis confirmed experimental consistency, with coefficient-of-variation values below 10% for all the considered platforms. In general, WSL2 also had slightly better scalability and processor utilization than native Windows. The results show that WSL2 is a suitable environment for multicore computing and that thread-based parallelism provides better scalability for CPU-bound workloads. This study provides a comprehensive perspective on multicore scalability across different parallel programming models and execution environments by integrating multiple performance and reliability metrics into a single benchmark framework.

Keywords: Multicore Computing; OpenMP; Parallel Efficiency; Python Multiprocessing; Scalability; Windows Subsystem for Linux 2

INTRODUCTION

The increasing demand for scientific computing, numerical simulation, and large-scale data processing has raised the need for efficient usage of modern multicore processors. Parallel computing has also become vital when executing complex machine learning tasks, such as optimizing neural network structures or selecting accurate training algorithms based on data pattern matching (Mustafidah et al., 2019, 2022). As Schryen (2024) reported, the conventional strategy of increasing the clock frequency of processors has gradually faced practical barriers due to power consumption and thermal limits. This has led processor manufacturers to adopt multicore architectures to sustain performance growth. This transition has changed the design and implementation of computationally intensive applications fundamentally (Dai et al., 2025). Parallel computing today is no longer restricted to large-scale computing facilities but has become an integral part of mainstream computing systems from scientific workstations to personal computers.

However, the availability of multicore processors does not guarantee efficient scalability. Performance gains depend on a combination of factors, including workload characteristics, synchronization overhead, memory management and operating-system scheduling policies. As pointed out by Ciccozzi et al. (2023), there is no single parallelization strategy that works for all cases, as different workloads have different computational and

*name of corresponding author

e-ISSN : 2541-2019
p-ISSN : 2541-044X

communication patterns. Similarly, Martell et al. (2022) show that task granularity and runtime overhead can have a significant effect on the multicore scalability, especially as the number of execution units increases.

Two of the most widely adopted approaches for exploiting multicore systems are Python multiprocessing and OpenMP. Multiprocessing has become a practical way to overcome the limitations of the Global Interpreter Lock and enable parallel execution using processes (Arboleda et al., 2023; Arjona et al., 2023). Python multiprocessing has become very popular in scientific and data-driven applications. In comparison, OpenMP is a lightweight shared memory programming model using multithreading that has been the de facto standard in high-performance computing environments for a long time (Dagum & Menon, 1998). The continued relevance of OpenMP-based parallelism within modern Python ecosystems, as demonstrated by Piñeiro & Pichel (2026), further highlights the convergence of productivity and performance-oriented computing frameworks.

The emergence of Windows Subsystem for Linux 2 (WSL2) has further expanded the landscape of parallel computing on desktop platforms. WSL2 provides a lightweight virtualization of a Linux-like environment so developers can run Linux-based development and scientific-computing tools without the overhead of a full Linux install. Furthermore, the performance characteristics of WSL2 are often comparable to those of native systems (Sobieraj & Kotyński, 2024; Tesser & Borin, 2023). Consequently, WSL2 is becoming an increasingly attractive platform for computational workloads.

These areas are usually treated separately, but great advances have been made in multiprocessing, OpenMP optimization, virtualization platforms and multicore scalability. Most existing studies focus on a particular programming model, execution environment or performance aspect (Arjona et al., 2023; Piñeiro & Pichel, 2026; Poolla & Saxena, 2023). Hence, there is no holistic understanding of the behavior of process-based and thread-based parallelism across native Windows and WSL2 environments when scalability and resource-utilization metrics are considered together.

This study benchmarks and compares Python multiprocessing and OpenMP on native Windows 11 and Ubuntu 24.04 under WSL2 using a CPU-intensive floating-point benchmark. The evaluation takes into account execution time, speedup, parallel efficiency, CPU utilization and statistical reliability obtained from repeated measurements. The study integrates these measures into one experimental setup, providing a unified assessment of multicore scalability for process and thread parallelism across different execution environments. The results provide a broader view on scalability behaviour and resource usage and practical guidance for choosing parallelization strategies for CPU-bound workloads. Through this approach, the study contributes a unified multicore scalability analysis across different parallelization models and execution environments.

LITERATURE REVIEW

Python Multiprocessing Parallel Computing Environment

The widespread use of Python in data science and scientific computing makes the multiprocessing module a practical approach for CPU-bound parallelism, bypassing the Global Interpreter Lock through the creation of independent processes (Arboleda et al., 2023; Arjona et al., 2023). However, this approach has a significant overhead due to process creation, inter-process communication and memory duplication, which is usually greater than the overhead of thread-based models like OpenMP (Fink et al., 2021; Piñeiro & Pichel, 2026). The number of processes tends to reduce the scalability because of process-management and synchronization expenses (Ruys et al., 2025). It is not enough to evaluate only based on execution time, and metrics such as speedup, efficiency, and resource utilization are required (Ruys et al., 2025). Furthermore, the performance of multi-processing is influenced by the operating-system scheduler, hardware configuration, and workload characteristics, leading to varying benchmark results across different platforms (Blandino & Meneses, 2022; Gonnord et al., 2023).

OpenMP and Shared Memory Parallelism

OpenMP is a de facto standard for shared memory parallel programming that employs a fork-join model and compiler directives that enable serial code to be parallelized incrementally (Chapman et al., 2008; Dagum & Menon, 1998). Speedup and efficiency are common measures of its scalability: Amdahl's law (Poolla & Saxena, 2023) precludes linear speedup by the serial fraction of the work and the cost of synchronization as the number of threads grows, and efficiency deteriorates when the number of logical threads exceeds the number of physical cores. Efficient OpenMP utilization requires careful handling of thread mapping, data locality and synchronization mechanisms (Pasqualin et al., 2025; Santos-da-Silva et al., 2025), and despite its higher implementation complexity compared to Python multiprocessing, it is one of the most efficient approaches for shared-memory workloads on a single node (Hunold & Kraßnitzer, 2023). The repeated comparisons of parallel programming models indicate that no single approach is optimal for all workloads and environments (Ciccozzi et al., 2023; Martell et al., 2022).

*name of corresponding author

e-ISSN : 2541-2019
p-ISSN : 2541-044X

WSL2 and Lightweight Virtualization

WSL2 implements lightweight Hyper-V-based virtualization with a native Linux kernel, making it more compatible with Linux than its predecessor and increasingly suitable for scientific and compute-intensive workloads (Tesser & Borin, 2023; Zhou et al., 2023). Its performance is mostly comparable to native Linux, with less overhead than traditional virtual machines (Aqasizade et al., 2025). However, overhead is still visible in file-system operations and I/O-intensive workloads (Sobieraj & Kotyński, 2024). Although some of the studies have reported the overall performance features of WSL2 and lightweight virtualization environments, the information regarding multicore scalability, processor utilization, and parallel-efficiency performance under CPU-bound parallel workloads is relatively limited.

Research Gap and Positioning

Research on parallel computing has evolved along several directions, including Python multiprocessing, OpenMP-based shared-memory parallelism, multicore scalability, and the performance of virtualized execution environments such as WSL2. While these studies provide valuable insights, they typically focus on individual aspects of parallel execution rather than examining them within a single evaluation framework.

Table 1 Summary of Related Studies and Research Gaps

Study	Main Focus	Research Limitation
Arjona et al. (2023)	Python multiprocessing and parallel execution scalability	No comparison with OpenMP, Windows, or WSL2.
Piñeiro & Pichel (2026)	OpenMP integration and parallel execution in Python	No Windows–WSL2 comparison or unified scalability evaluation.
Sobieraj & Kotyński (2024)	WSL2 and virtualization performance	No analysis of parallel programming models.
Aqasizade et al. (2025)	Virtualized and containerized computing performance	No evaluation of multiprocessing or OpenMP.
Poolla & Saxena (2023)	Multicore scalability and performance modeling	No experimental comparison of parallel programming approaches.

This pattern is common in the existing literature as shown in Table 1. Previous works have studied individual aspects of parallel computing, such as multiprocessing, OpenMP optimization, virtualization platforms or scalability modeling. These limitations not only are methodological differences, but also confine the interpretation of multicore scalability behavior. Evaluations that focus solely on execution time may overlook resource-utilization characteristics, whereas studies conducted within a single operating-system environment provide limited insight into platform-dependent scheduling effects. Thus, the combined influence of programming models and execution environments remains insufficiently understood.

This study addresses these gaps through a unified benchmark-based evaluation of Python multiprocessing and OpenMP on Windows and WSL2. The study integrates execution time, speedup, parallel efficiency, CPU utilization, and repeated-measurement reliability analysis within a single benchmark framework, providing a unified multicore scalability assessment across both programming models and execution environments.

METHOD

Environment of Test

All experiments were conducted on an Intel Core i9-9900 processor with 8 physical cores, 16 logical threads (Hyper-Threading enabled), and 16 GB RAM, running Windows 11 as the native host and Ubuntu 24.04 deployed through WSL2. Python benchmarks were implemented using Python 3.14.5 and the multiprocessing module; OpenMP benchmarks were implemented in C and compiled using GCC with the -O3 and -fopenmp flags.

Benchmark Design

The benchmark represents a CPU-intensive workload based on iterative floating-point square root (sqrt) calculations, which are commonly encountered in scientific and numerical computing applications (Poolla & Saxena, 2023). A fixed workload of 160,000,000 iterations was used for all experiments.

The benchmark was intentionally designed without file I/O, network communication, or external dependencies to isolate processor scalability effects. Consequently, performance differences are primarily influenced by variations in the number of processes or threads. The evaluation focused on execution time, speedup, efficiency, and CPU utilization.

To minimize thermal-throttling effects, all experiments were executed under stable system conditions with no competing computational workloads. Although cache behavior and memory bandwidth were not directly profiled,

*name of corresponding author

e-ISSN : 2541-2019
p-ISSN : 2541-044X

the selected benchmark was intentionally designed as a CPU-bound floating-point workload to reduce the influence of disk I/O and external system activities.

Python Multiprocessing Implementation

The Python implementation employed the Pool method from the multiprocessing module to distribute the workload among multiple worker processes. As demonstrated by Arjona et al. (2023), this approach enables concurrent execution while bypassing the Global Interpreter Lock (GIL). The evaluated configurations consisted of 1, 2, 4, 8, and 16 processes.

Multiprocessing introduces overhead associated with process creation, memory management, and inter-process communication (Piñeiro & Pichel, 2026). Furthermore, multiprocessing behavior differs across operating systems. Windows uses the spawn process creation mechanism, whereas WSL2 relies on fork(), potentially affecting runtime behavior and scalability characteristics.

OpenMP Implementation

OpenMP was implemented in C using GCC and a shared-memory parallel execution model. Parallel execution was achieved through the #pragma omp parallel for directive combined with a reduction mechanism to prevent race conditions during floating-point accumulation, following the approach of Dagum & Menon (1998).

The OpenMP benchmark was compiled using GCC with the optimization flags -O3 and -fopenmp. The -O3 optimization level enables aggressive compiler optimizations for computational workloads, while -fopenmp activates OpenMP-based parallel execution support.

Thread counts of 1, 2, 4, 8, and 16 were evaluated using omp_set_num_threads(). Since the experimental platform provides 8 physical cores and 16 logical threads, the selected configurations include both physical-core utilization and Hyper-Threading scenarios, enabling the investigation of multicore scalability behavior.

Performance Metrics

In this research, performance evaluation was performed by using some common metrics of parallel computing such as execution time, speedup, efficiency and CPU utilization as discussed by Poolla & Saxena (2023). The overall combination of these metrics allows for a more holistic performance analysis in terms of not only execution time but also resource utilization efficiency.

Execution time was used to measure the total time needed by a benchmark to finish a certain workload using a specific number of processes or threads. Equation (1) is defined as:

$$T_p \quad (1)$$

where, T_p is the execution time when using p processes or threads.

The acceleration achieved in parallel execution in comparison to the serial execution was calculated by the speedup. The speedup (2) is defined as follows:

$$S_p = \frac{T_1}{T_p} \qquad (2)$$

where T_1 is the execution time in single thread or single process configuration and T_p is the execution time in parallel configuration.

Efficiency was used to measure how effectively the available processes or threads contributed to the obtained speedup. The parallel efficiency (3) is calculated as:

$$E_p = \frac{S_p}{p} \times 100\% \qquad (3)$$

An efficiency value close to 100% indicates optimal utilization of parallel resources, whereas lower values reflect parallel overhead, synchronization cost, or resource contention. Lower efficiency usually indicates parallel overhead, synchronization cost, or resource contention.

To provide a theoretical perspective on scalability, the experimental results were also interpreted using Amdahl's Law (4) (Amdahl, 1967):

$$S(N) = \frac{1}{(1-P)+\frac{P}{N}} \qquad (4)$$

where $S(N)$ represents the theoretical speedup obtained using N processing units and P denotes the parallelizable fraction of the workload. This model was employed as a conceptual framework for interpreting scalability limitations observed in the experimental results.

Execution time, speedup, and efficiency were obtained from benchmark-only executions to avoid monitoring overhead. CPU utilization was collected through a separate monitoring procedure. Python monitoring used the psutil library, whereas OpenMP monitoring relied on GetSystemTimes() on Windows and /proc/stat on WSL2/Linux.

*name of corresponding author

Sinkron : Jurnal dan Penelitian Teknik Informatika
Volume 10, Number 3, July 2026
DOI : https://doi.org/10.33395/sinkron.v10i3.16262
e-ISSN : 2541-2019
p-ISSN : 2541-044X

Repeated Measurement and Statistical Analysis

To improve experimental reliability, both benchmark and monitoring experiments were repeated ten times for each configuration. Repeated measurements are commonly recommended in performance evaluation studies to reduce the influence of transient system variations and improve result reliability (Georges et al., 2007; Jain, 1991).

The arithmetic mean (5) was calculated as:

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i \quad (5)$$

where $\bar{x}$ denotes the arithmetic mean, x_i represents the value obtained from the i-th execution, and n is the total number of repeated executions.

Measurement variability (6) was quantified using the sample standard deviation:

$$s = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2} \quad (6)$$

where s denotes the sample standard deviation and $\bar{x}$ represents the arithmetic mean of the repeated measurements.

To facilitate relative variability analysis across different benchmark configurations, the coefficient of variation (CV) (7) was calculated as:

$$CV = \frac{s}{\bar{x}} \times 100\% \quad (7)$$

where CV denotes the coefficient of variation.

To provide an additional measure of statistical reliability, the 95% confidence interval (CI) (8) was calculated as:

$$CI = \bar{x} \pm t\left(\frac{s}{\sqrt{n}}\right) \quad (8)$$

where $\bar{x}$ denotes the arithmetic mean, s is the sample standard deviation, n is the number of repeated executions, and t is the critical value of the Student's t-distribution for a 95% confidence level.

The mean values were used as the primary performance indicators, while the standard deviations and coefficient of variation were used to assess experimental stability and measurement variability. This approach reduces the influence of operating-system scheduling variations, background services, and runtime noise, thereby improving the robustness of performance comparisons (Georges et al., 2007).

Experimental Procedure

Each benchmark was initially executed using a single process or thread to establish the baseline execution time. The number of processes or threads was then increased to 2, 4, 8, and 16 while maintaining a constant workload size. All experiments were conducted under identical hardware and software conditions, and repeated measurements were used to minimize the influence of transient thermal effects, operating-system activities, and other runtime variations.

Two experimental procedures were performed. The first measured execution time, speedup, and efficiency without runtime monitoring. The second collected CPU utilization through dedicated monitoring mechanisms. Each configuration was executed ten times, and the resulting mean, standard deviation, and coefficient of variation values were used for comparative analysis and reliability assessment.

Finally, the results were visualized using the matplotlib library in Python through execution time, speedup, efficiency, and CPU utilization comparisons to analyze multicore scalability and resource utilization across both platforms.

RESULT

Execution Time Results

Execution time was evaluated to measure the total duration required to complete the benchmark workload under different process and thread configurations. The arithmetic mean and standard deviation obtained from ten repeated executions are summarized in Table 2.

Table 2 Execution Time Results (Mean ± Standard Deviation)

Processes / Threads	Python Windows (s)	Python WSL2 (s)	OpenMP Windows (s)	OpenMP WSL2 (s)
1	11.798 ± 0.086	14.879 ± 0.092	0.206 ± 0.002	0.206 ± 0.002
2	6.233 ± 0.339	7.838 ± 0.432	0.106 ± 0.005	0.104 ± 0.001
4	3.302 ± 0.198	4.644 ± 0.976	0.055 ± 0.004	0.053 ± 0.001
8	2.366 ± 0.389	3.411 ± 0.336	0.042 ± 0.012	0.054 ± 0.000
16	2.098 ± 0.039	2.571 ± 0.021	0.032 ± 0.001	0.029 ± 0.001

*name of corresponding author

Sinkron : Jurnal dan Penelitian Teknik Informatika
Volume 10, Number 3, July 2026
DOI : https://doi.org/10.33395/sinkron.v10i3.16262
e-ISSN : 2541-2019
p-ISSN : 2541-044X

The execution-time measurements indicate a consistent reduction in runtime as the number of processes and threads increases. Both Python multiprocessing and OpenMP benefited from parallel execution; however, OpenMP consistently achieved substantially lower execution times than Python multiprocessing across all configurations. The largest reductions were observed when moving from single-worker execution to moderate levels of parallelism.

At the highest configuration, OpenMP achieved execution times of 0.032 s on Windows and 0.029 s on WSL2, whereas Python multiprocessing required 2.098 s and 2.571 s, respectively. These results demonstrate a substantial performance advantage of OpenMP for the evaluated CPU-intensive workload. The execution-time trends across all evaluated configurations are further visualized in Fig. 1(a).

Parallel Speedup Results

Parallel speedup was calculated relative to the baseline single-process or single-thread configuration. Increasing the number of processes and threads generally produced higher acceleration, although the magnitude of improvement varied across implementations.

Table 3 Parallel Speedup Results (Mean ± Standard Deviation)

Processes / Threads	Python Windows	Python WSL2	OpenMP Windows	OpenMP WSL2
1	1.000 ± 0.000	1.000 ± 0.000	1.000 ± 0.000	1.000 ± 0.000
2	1.898 ± 0.093	1.903 ± 0.099	1.944 ± 0.091	1.974 ± 0.024
4	3.582 ± 0.197	3.311 ± 0.580	3.766 ± 0.215	3.849 ± 0.061
8	5.103 ± 0.798	4.399 ± 0.433	5.321 ± 1.532	3.829 ± 0.032
16	5.624 ± 0.123	5.790 ± 0.049	6.482 ± 0.191	7.065 ± 0.164

The speedup values reported in Table 3 demonstrate the scalability benefits obtained from additional processes and threads. The highest acceleration was achieved by OpenMP on WSL2, reaching a speedup of 7.065 at 16 threads. Python multiprocessing achieved lower but still substantial speedups, reaching 5.624 on Windows and 5.790 on WSL2.

Overall, OpenMP maintained higher speedup values than Python multiprocessing throughout most configurations, indicating more effective utilization of parallel resources for the benchmark workload.

The corresponding speedup behavior is illustrated in Fig. 1(b).

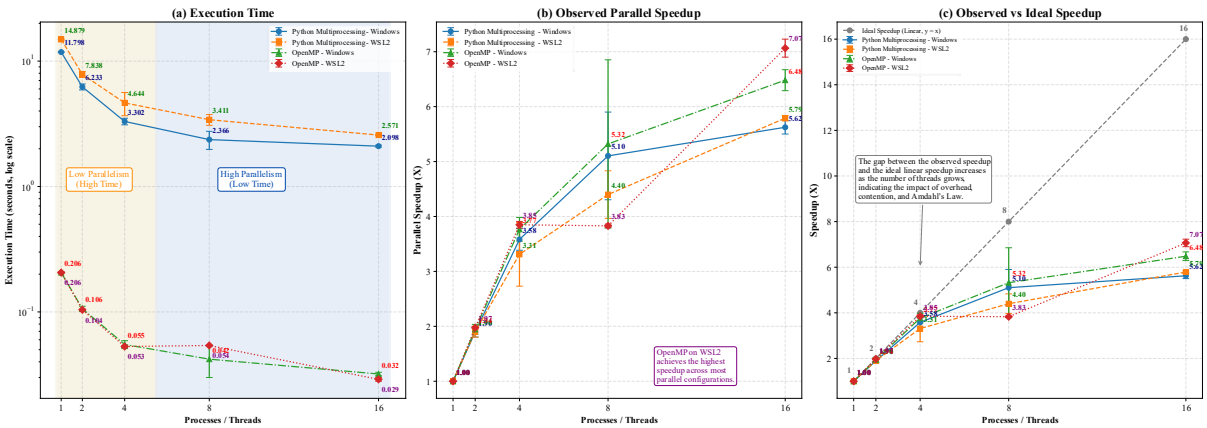

Fig. 1 Scalability Analysis of Python Multiprocessing and OpenMP under Windows and WSL2: (a) Execution Time, (b) Observed Parallel Speedup, and (c) Comparison Between Observed and Ideal Speedup. Error bars represent one standard deviation obtained from ten repeated executions

Fig. 1(c) highlights the increasing gap between observed and ideal scalability as the degree of parallelism increases, while Figs. 1(a) and 1(b) illustrate the corresponding execution-time reduction and speedup growth. Among the evaluated configurations, OpenMP on WSL2 achieved the highest observed speedup, while all implementations exhibited an increasing deviation from ideal linear scalability at larger thread counts.

The scalability trends observed in Fig. 1 provide additional context for the efficiency behavior reported in Table 3.

*name of corresponding author

Sinkron : Jurnal dan Penelitian Teknik Informatika
Volume 10, Number 3, July 2026
DOI : https://doi.org/10.33395/sinkron.v10i3.16262
e-ISSN : 2541-2019
p-ISSN : 2541-044X

Parallel Efficiency Results

The effectiveness of resource utilization was evaluated through parallel efficiency. The resulting efficiency values for all configurations are presented in Table 4.

Table 4 Parallel Efficiency Results (Mean ± Standard Deviation)

Processes / Threads	Python Windows (%)	Python WSL2 (%)	OpenMP Windows (%)	OpenMP WSL2 (%)
1	100.000 ± 0.000	100.000 ± 0.000	100.000 ± 0.000	100.000 ± 0.000
2	94.870 ± 4.718	95.164 ± 4.964	97.266 ± 4.523	98.703 ± 1.193
4	89.577 ± 4.889	82.797 ± 14.458	94.122 ± 5.363	96.191 ± 1.496
8	63.797 ± 9.997	54.996 ± 5.398	66.516 ± 19.152	47.865 ± 0.397
16	35.150 ± 0.765	36.172 ± 0.308	40.512 ± 1.193	44.157 ± 1.009

A gradual decline in efficiency can be observed as the degree of parallelism increases. Although higher process and thread counts produced greater speedups, the additional computational resources were utilized less efficiently. Among the evaluated implementations, OpenMP generally maintained higher efficiency values than Python multiprocessing, particularly in the lower and intermediate configurations.

At four threads, OpenMP on WSL2 achieved an efficiency of 96.191%, indicating highly effective utilization of the available processing resources. In contrast, efficiency values at 16 processes or threads decreased to approximately 35–44% across all implementations.

CPU Utilization Results

CPU utilization was measured through dedicated monitoring experiments to evaluate processor resource usage during benchmark execution. The corresponding CPU-utilization measurements are summarized in Table 5.

Table 5 CPU Utilization Results (Mean ± Standard Deviation)

Processes / Threads	Python Windows (%)	Python WSL2 (%)	OpenMP Windows (%)	OpenMP WSL2 (%)
1	7.29 ± 0.71	6.21 ± 0.03	8.20 ± 1.62	7.28 ± 0.84
2	12.90 ± 0.44	12.32 ± 0.10	13.16 ± 2.08	13.95 ± 0.76
4	24.62 ± 0.51	22.50 ± 1.06	25.24 ± 4.25	24.65 ± 0.51
8	42.13 ± 3.50	41.10 ± 1.38	34.92 ± 9.44	37.35 ± 7.05
16	84.44 ± 2.99	92.44 ± 0.88	85.13 ± 15.78	97.31 ± 2.18

The CPU-utilization trends observed across different process and thread configurations are illustrated in Fig. 2(a).

CPU utilization increased steadily as additional processes and threads were introduced. Low-parallelism configurations utilized only a small fraction of the available logical processors, whereas high-parallelism configurations approached full utilization of the multicore platform.

The highest utilization was observed for OpenMP on WSL2, reaching 97.31% at 16 threads. In general, WSL2 achieved slightly higher utilization levels than native Windows, particularly in the largest process and thread configurations.

The relationship between processor utilization and parallel efficiency across different levels of parallelism is further illustrated in Fig. 2.

*name of corresponding author

Sinkron : Jurnal dan Penelitian Teknik Informatika
Volume 10, Number 3, July 2026
DOI : https://doi.org/10.33395/sinkron.v10i3.16262
e-ISSN : 2541-2019
p-ISSN : 2541-044X

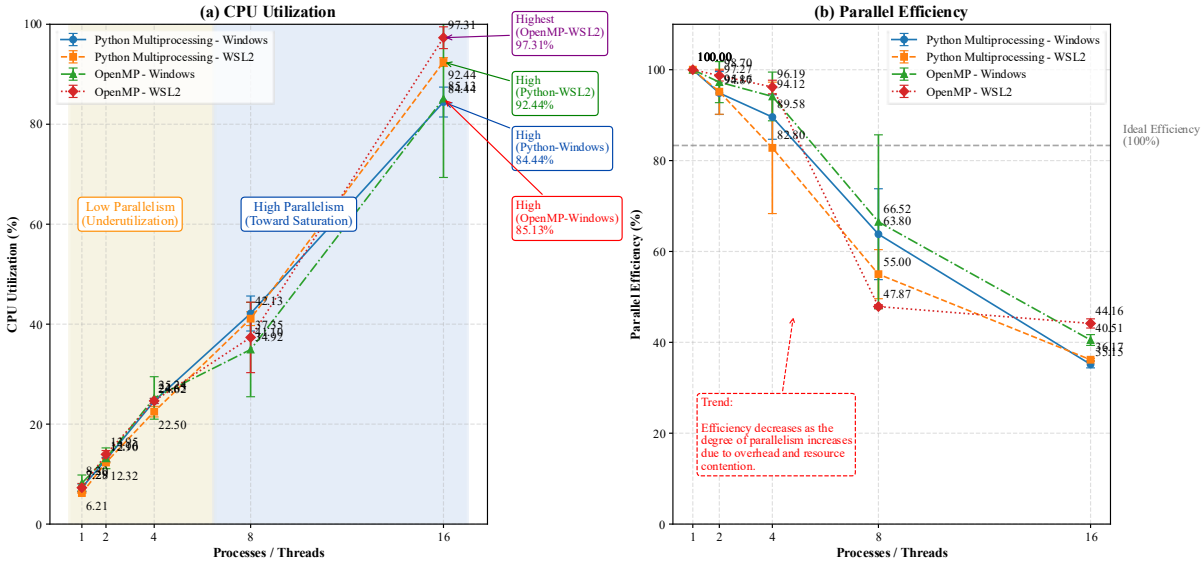

Fig. 2 Resource Utilization Analysis of Python Multiprocessing and OpenMP under Windows and WSL2: (a) CPU Utilization and (b) Parallel Efficiency. Error bars represent one standard deviation obtained from ten repeated executions

Fig. 2 shows that higher processor utilization does not necessarily translate into proportional efficiency gains, particularly at larger process and thread counts. Fig. 2(b) further summarizes the efficiency behavior observed under increasing levels of parallelism.

Reliability Analysis

To assess experimental consistency, coefficient of variation (CV) and 95% confidence interval statistics were calculated from ten repeated executions. The overall reliability indicators are summarized in Table 6.

Table 6 Reliability Analysis Based on Coefficient of Variation and 95% Confidence Interval

Platform	Avg. CV (%)	Avg. 95% CI Width (%)
Python Windows	5.80	8.30
Python WSL2	6.97	9.97
OpenMP Windows	8.48	12.13
OpenMP WSL2	1.24	1.77

The reliability statistics indicate relatively low variability across repeated executions. All evaluated platforms produced average CV values below 10%, suggesting good repeatability and stable benchmark behavior.

OpenMP on WSL2 exhibited the highest stability, with an average CV of only 1.24% and a confidence-interval width of 1.77%. The relatively low variability observed across all platforms supports the reliability of the benchmark results presented in Tables 2–5 and indicates that random runtime fluctuations had only a limited influence on the reported performance metrics.

The relatively narrow confidence intervals further indicate that the observed performance differences between platforms are unlikely to be caused by random runtime fluctuations. These results support the statistical robustness of the repeated-measurement methodology adopted in this study.

Consequently, the observed execution-time, speedup, efficiency, and CPU-utilization trends can be interpreted with a high degree of confidence.

DISCUSSIONS

Scalability and Amdahl's Law

As shown in Fig. 1(c), the measured speedup deviates more and more from the ideal linear speedup as the number of threads increases. The time of execution was significantly reduced with the addition of processes and threads, but the acceleration achieved was not proportional to the increase in computational resources. The highest measured speedup for OpenMP on WSL2 was 7.065 at 16 threads, which is still well below the theoretical speedup of 16.

This behavior is in agreement with Amdahl's Law which states that the maximum speedup achievable by a parallel application is limited by the remaining serial part of the workload (Amdahl, 1967; Poolla & Saxena, 2023).

*name of corresponding author

e-ISSN : 2541-2019
p-ISSN : 2541-044X

Table 4 provides additional support for this observation in terms of efficiency results. Efficiency was close to 100% for low number of threads, but decayed slowly with increased parallelism, reaching 44.157% for OpenMP on WSL2 at 16 threads. The results suggest diminishing returns as additional computational resources are allocated beyond the most efficient parallel configuration.

Locally, an exception to this trend was observed for OpenMP on WSL2, where the speedup decreased slightly from 3.849× at four threads to 3.829× at eight threads (Table 3). This small regression may be associated with the transition from fully independent physical-core execution to resource sharing under Hyper-Threading, where two logical threads compete for the execution units of a single physical core, incurring scheduling overhead that momentarily outweighs the benefits of more threads. Another transient efficiency drop at the physical-core boundary has been observed in other OpenMP scalability studies (Martell et al., 2022; Santos-da-Silva et al., 2025).

The reliability analysis is shown in Table 6, which indicates low variability across repeated runs, suggesting that the observed scalability patterns are reproducible and not just due to random runtime variations.

Hyper-Threading and Processor Utilization

The experimental platform has 8 physical cores and 16 logical threads provided by Hyper-Threading technology. Results show that the performance keeps improving beyond 8 execution units, but the magnitude of the improvement diminishes further with increased thread counts. This trend is reflected in the efficiency results, where all implementations show a significant drop at 16 processes/threads.

Hyper-Threading takes advantage of the resources of a single physical core to run multiple execution contexts, thus increasing processor utilization. As shown in Table 4, the CPU utilization for Python multiprocessing was 84.44% and 92.44% for Windows and WSL2, respectively, and 85.13% and 97.31% for OpenMP. But the logical threads share the execution resources of the same physical core and cannot provide the same performance benefit as additional physical cores. This characteristic is also reflected in the efficiency decline observed at higher levels of parallelism (Serpa et al., 2023).

Moreover, shared processor resources, such as cache hierarchies and execution units, may introduce contention among logical threads, further reducing efficiency with the number of concurrent execution contexts (Dai et al., 2025; Serpa et al., 2023). This trend is also shown in Fig. 2, where the CPU utilization keeps growing in Fig. 2(a) and the parallel efficiency decreases in Fig. 2(b) with the increase of the thread counts.

Windows and WSL2 Performance Behavior

The experimental results indicate that WSL2 generally achieved higher speedup and CPU utilization than native Windows, particularly for OpenMP workloads. The highest utilization was observed for OpenMP on WSL2, reaching 97.31% at 16 threads, which was accompanied by the highest measured speedup of 7.065.

Although scheduler activity was not directly profiled, the results suggest that WSL2 was able to utilize the available logical processors more effectively during highly parallel execution. This observation is supported by the consistently higher utilization values obtained under WSL2 in the larger process and thread configurations. Furthermore, OpenMP on WSL2 achieved the highest speedup (7.065) and the highest efficiency (44.157%) among the evaluated 16-thread configurations.

Consequently, WSL2 exhibited slightly better scalability characteristics than native Windows for the benchmark workload evaluated in this study, although the underlying scheduler behavior and operating-system-level resource management mechanisms were not directly measured. Similar performance differences between execution environments have been reported in studies evaluating virtualized and containerized computing platforms (Aqasizade et al., 2025; Sobieraj & Kotyński, 2024).

Parallel Overhead and Performance Limitations

In all the configurations considered, OpenMP consistently performed faster than Python multiprocessing. The execution time was reduced from 11.798 s to 2.098 s on Windows and 14.879 s to 2.571 s on WSL2 using Python multiprocessing. Using OpenMP, the execution time was reduced from 0.206 s to 0.032 s and 0.029 s on Windows and WSL2, respectively.

This is mainly because of the underlying execution models. Due to the use of separate processes, the Python multiprocessing library incurs the overhead of process creation, memory management, and inter-process communication (Piñeiro & Pichel, 2026). In contrast, OpenMP uses lightweight, shared memory threads that generally reduce communication and synchronization requirements (Dagum & Menon, 1998).

The reduced efficiency seen at increased process and thread counts indicates the effect of synchronization costs, resource contention, and cache-related constraints that are typical in multicore systems (Martell et al., 2022). Contending for shared cache resources between logical threads may have been a factor in efficiency degradation at higher levels of parallelism, even though cache locality and memory-bandwidth utilization were not directly measured. This behavior is evident in Fig. 2, where Fig. 2(a) shows CPU utilization approaching saturation while parallel efficiency declines in Fig. 2(b). The gap between these metrics means that a higher processor utilization

*name of corresponding author

e-ISSN : 2541-2019
p-ISSN : 2541-044X

does not always translate into a proportional performance gain, especially when new threads compete for shared execution resources.

Since the experimental platform contains a single processor socket and a single NUMA node, NUMA-related effects are expected to be negligible in the present study. However, the synchronization overheads, contention on shared resources, and the serial portion of the workload probably explain the non-linear scaling behavior of the Python multiprocessing and OpenMP implementations.

CONCLUSION

Key Findings

This study evaluated the performance of Python multiprocessing and OpenMP on native Windows 11 and WSL2 Ubuntu 24.04 using a CPU-intensive floating-point benchmark. The performance was assessed based on execution time, speedup, parallel efficiency, CPU utilization and statistical reliability measures.

The results indicate that both approaches utilize multicore execution. However, OpenMP outperformed Python multiprocessing in all the tested configurations. The highest speedup of 7.065 was achieved by OpenMP on WSL2 at 16 threads. Overall, WSL2 demonstrated slightly better scalability and processor utilization than native Windows, particularly for OpenMP workloads.

Scalability and Resource Utilization Insights

Despite the performance improvement by increasing the number of processes and threads, the observed speedup was less than the theoretical linear speedup, and the efficiency decreased at higher levels of parallelism. These results show diminishing returns from parallel overhead, synchronization costs, resource contention and the remaining serial portion of the workload.

Resource-utilization analysis has also shown that higher CPU utilization does not lead to proportional performance benefits. Reliability analysis revealed low variability in repeated runs, supporting the consistency of the reported results.

Practical Implications

The results indicate that OpenMP is more suitable for scientific computing and other CPU-intensive workloads requiring high performance and scalability. Python multiprocessing remains a practical option when rapid development and integration with the Python ecosystem are prioritized.

The results also show that WSL2 can be an effective platform for parallel-computing development on Windows systems. Furthermore, the best compromise between speedup and efficiency is generally reached when the number of processes or threads is in tune with the number of available physical cores.

Limitations and Future Work

This study was limited to a single hardware platform, CPU-bound floating-point workloads, and a comparison of Windows and WSL2. Furthermore, cache locality, memory-bandwidth effects, and low-level hardware events were not explicitly profiled and may affect scalability behavior at higher process and thread counts.

Future work may include native Linux environments, additional processor architectures, memory-bound and I/O-intensive workloads, distributed-memory approaches like MPI and modern Python parallel frameworks. Low-level profiling techniques are additionally useful to gain further insights into scheduling behavior, synchronization overhead, cache utilization and memory-access characteristics in multicore systems.

The primary contribution of this study is the integration of performance, resource-utilization, and reliability metrics within a unified benchmark methodology for multicore scalability assessment. Unlike previous studies that focus on individual programming models or execution environments, this methodology enables a comprehensive evaluation of process-based and thread-based parallelism across Windows and WSL2.

REFERENCES

Amdahl, G. M. (1967). Validity of the single processor approach to achieving large scale computing capabilities. *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference on - AFIPS '67 (Spring)*, 483. https://doi.org/10.1145/1465482.1465560

Aqasizade, H., Ataie, E., & Bastam, M. (2025). Experimental assessment of containers running on top of virtual machines. *IET Networks*, *14*(1). https://doi.org/10.1049/ntw2.12138

Arboleda, F. J. M., Arias, M. R., & Riveros, J. A. H. (2023). Performance of Parallelism in Python and C++. *IAENG International Journal of Computer Science*, 579–591.

Arjona, A., Finol, G., & López, P. G. (2023). Transparent serverless execution of Python multiprocessing applications. *Future Generation Computer Systems*, *140*, 436–449. https://doi.org/10.1016/j.future.2022.10.038

*name of corresponding author

e-ISSN : 2541-2019
p-ISSN : 2541-044X

Blandino, H. O., & Meneses, E. (2022). A Comparative Evaluation of Parallel Programming Python Tools for Particle-in-Cell on Symmetric Multiprocessors. In C. J. and O. C. and G. G. Navaux Philippe and Barrios H. (Ed.), *High Performance Computing* (pp. 1–15). Springer International Publishing.

Chapman, Barbara., Jost, Gabriele., & Pas, R. van der. (2008). *Using OpenMP : portable shared memory parallel programming*. MIT Press.

Ciccozzi, F., Addazi, L., Asadollah, S. A., Lisper, B., Masud, A. N., & Mubeen, S. (2023). A Comprehensive Exploration of Languages for Parallel Computing. *ACM Computing Surveys*, *55*(2), 1–39. https://doi.org/10.1145/3485008

Dagum, L., & Menon, R. (1998). OpenMP: an industry standard API for shared-memory programming. *IEEE Computational Science and Engineering*, *5*(1), 46–55. https://doi.org/10.1109/99.660313

Dai, F., Hossain, M. A., & Wang, Y. (2025). State of the Art in Parallel and Distributed Systems: Emerging Trends and Challenges. *Electronics*, *14*(4), 677. https://doi.org/10.3390/electronics14040677

Fink, Z., Liu, S., Choi, J., Diener, M., & Kale, L. V. (2021). *Performance Evaluation of Python Parallel Programming Models: Charm4Py and mpi4py*. https://doi.org/10.1109/ESPM254806.2021.00010

Georges, A., Buytaert, D., & Eeckhout, L. (2007). Statistically rigorous java performance evaluation. *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, 57–76. https://doi.org/10.1145/1297027.1297033

Gonnord, L., Henrio, L., Morel, L., & Radanne, G. (2023). A Survey on Parallelism and Determinism. *ACM Computing Surveys*, *55*(10), 1–28. https://doi.org/10.1145/3564529

Hunold, S., & Kraßnitzer, K. (2023). *A Quantitative Analysis of OpenMP Task Runtime Systems* (pp. 3–18). https://doi.org/10.1007/978-3-031-31180-2_1

Jain, R. (1991). *The Art of Computer Systems Performance Analysis: Techniques For Experimental Design, Measurement, Simulation, and Modeling, NY: Wiley*.

Martell, V., Korochkin, A., & Rusanova, O. (2022). Comparative analysis of the effectiveness of using fine-grained and nested parallelism to increase the speedup of parallel computing in multicore computer systems. *System Research and Information Technologies*, (2), 45–60. https://doi.org/10.20535/SRIT.2308-8893.2022.2.03

Mustafidah, H., Geography, S., & Permatasari, S. N. C. (2019). Accuracy of the Neurons Number in the Hidden Layer of the Levenberg-Marquardt Algorithm. *International Journal of Recent Technology and Engineering (IJRTE)*, *8*(4), 2349–2353. https://doi.org/10.35940/ijrte.D8259.118419

Mustafidah, H., Pambudi, U. B., & Suwarsito, S. (2022). *Selection of the most accurate training algorithm in the backpropagation network based on the accuracy of data pattern matching*. 060012. https://doi.org/10.1063/5.0111243

Pasqualin, D. P., Diener, M., Du Bois, A. R., & Pilla, M. L. (2025). Thread and Data Mapping in Software Transactional Memory: an Overview. *International Journal of Parallel Programming*, *53*(3), 19. https://doi.org/10.1007/s10766-025-00796-1

Piñeiro, C., & Pichel, J. C. (2026). OMP4Py: A pure Python implementation of openMP. *Future Generation Computer Systems*, *175*, 108035. https://doi.org/10.1016/j.future.2025.108035

Poolla, C., & Saxena, R. (2023). On extending Amdahl's law to learn computer performance. *Microprocessors and Microsystems*, *96*, 104745. https://doi.org/10.1016/j.micpro.2022.104745

Ruys, W., Lee, H., You, B., Talati, S., Park, J., Almgren-Bell, J., Yan, Y., Fernando, M., Erez, M., Gligoric, M., Burtscher, M., Rossbach, C. J., Pingali, K., & Biros, G. (2025). Performance Characterization of Python Runtimes for Multi-device Task Parallel Programming. *International Journal of Parallel Programming*, *53*(2), 16. https://doi.org/10.1007/s10766-025-00788-1

Santos-da-Silva, F. H., Fernandes, J. B., Sardina, I. M., Barros, T., Xavier-de-Souza, S., & Assis, I. A. S. (2025). Auto-Tuning for OpenMP Dynamic Scheduling applied to Full Waveform Inversion. *Computers & Geosciences*, *202*, 105932. https://doi.org/10.1016/j.cageo.2025.105932

Schryen, G. (2024). Speedup and efficiency of computational parallelization: A unifying approach and asymptotic analysis. *Journal of Parallel and Distributed Computing*, *187*, 104835. https://doi.org/10.1016/j.jpdc.2023.104835

Serpa, M. S., Cruz, E. H. M., Diener, M., Lorenzon, A. F., Beck, A. C. S., & Navaux, P. O. A. (2023). Mitigating execution unit contention in parallel applications using instruction-aware mapping. *Concurrency and Computation: Practice and Experience*, *35*(17). https://doi.org/10.1002/cpe.6819

Sobieraj, M., & Kotyński, D. (2024). Docker Performance Evaluation across Operating Systems. *Applied Sciences*, *14*(15), 6672. https://doi.org/10.3390/app14156672

Tesser, R. K., & Borin, E. (2023). Containers in HPC: a survey. *The Journal of Supercomputing*, *79*(5), 5759–5827. https://doi.org/10.1007/s11227-022-04848-y

*name of corresponding author

e-ISSN : 2541-2019
p-ISSN : 2541-044X

Zhou, N., Zhou, H., & Hoppe, D. (2023). Containerization for High Performance Computing Systems: Survey and Prospects. *IEEE Transactions on Software Engineering*, *49*(4), 2722–2740. https://doi.org/10.1109/TSE.2022.3229221

*name of corresponding author