IRSTI 20.53.21, 20.53.23, 20.53.25

10.51889/2959-5894.2025.91.3.018

A.T. Zholdybay[1*] A. Aituov[1]

[1] Kazakh-British Technical University, Almaty, Kazakhstan
[*]*e-mail: as_zholdybay@kbtu.kz*

PYTHON CONCURRENCY FOR HIGH-LOAD MULTICORE PROCESSING

Abstract

Python's concurrency techniques for processor-intensive activities on multicore computers are thoroughly evaluated in this article. For workloads that are CPU-bound, I/O-bound, and mixed, we contrast multithreading, multiprocessing, and a hybrid threading + multiprocessing strategy. Execution time, CPU utilization, memory consumption, and realized speedup were measured against a sequential baseline in experimental benchmarks on a multi-core computer. The Global Interpreter Lock (GIL) prevents Python threads from providing any speedup for CPU-bound applications, according to the results. In contrast, numerous processes obtain near-linear speedup (3.7× on 4 cores) at the expense of increased memory utilization. Both multiprocessing and multithreading greatly increase throughput (by more than 3× speedup) for I/O-bound tasks by overlapping I/O delays, while threading has a smaller overhead. By overlapping computation and I/O, a hybrid concurrency technique outperforms pure multiprocessing by 3.3×, providing the best performance in a hybrid workload that combines compute and I/O. We examine the findings in light of Amdahl's Law and talk about how Python thread parallelism is essentially constrained by the GIL. Future attempts to eliminate the GIL and its effects on Python's concurrency environment are also discussed in the article. The results emphasize the trade-offs in speed and resource utilization for each strategy on multicore processors and offer recommendations for choosing efficient concurrency strategies in Python.

Keywords: Python; concurrency; multithreading; multiprocessing; Global Interpreter Lock; multi-core; parallel computing; performance.

А.Т. Жолдыбай[1] А. Айтуов[1]

[1] Қазақстан-Британ техникалық университеті, Алматы қ., Қазақстан

КӨПЯДРОЛЫ ЖҮЙЕЛЕРДЕГІ АУЫР ЕСЕПТЕРГЕ АРНАЛҒАН PYTHON ПАРАЛЛЕЛДІЛІГІ

Аңдатпа

Бұл мақалада көпядролы компьютерлерде процессорға жүктемесі жоғары тапсырмалар үшін Python тіліндегі параллелизм әдістері жан-жақты бағаланады. CPU-ге тәуелді, I/O-ге тәуелді және аралас жүктемелер үшін көпағынды (multithreading), көппроцессорлы (multiprocessing) және гибридті (ағын + процесс) тәсілдер салыстырылады. Көпядролы жүйеде орындалған эксперименттік тесттерде орындау уақыты, CPU қолданылуы, жады тұтыну және параллельділік арқылы алынған үдеулер (speedup) бірізді (секвенциалды) негізбен салыстырылып өлшенді. Нәтижелер Python-дағы GIL (Global Interpreter Lock) CPU-ге тәуелді қолданбалар үшін көпағындылық арқылы жылдамдық арттыруға мүмкіндік бермейтінін көрсетті. Ал көптеген процестер көмегімен дерлік сызықтық үдеу (4 ядро үшін 3.7×) алынады, бірақ ол көбірек жадыны қажет етеді. I/O жүктемелерінде multiprocessing пен multithreading әдістері I/O кідірістерін жабу арқылы өткізу қабілетін 3 есе арттырады, мұнда ағындар кішігірім ресурстар тұтынады. Есептеу мен I/O-ны біріктіретін гибридті әдіс таза multiprocessing-ке қарағанда 3.3 есе жақсы нәтиже беріп, есептеу мен I/O араласқан жүктемелерде ең жоғары өнімділікті қамтамасыз етеді. Зерттеу нәтижелері Амдал заңы тұрғысынан қарастырылып, Python ағындарының GIL арқылы шектелетіндігі талқыланады. Сондай-ақ, болашақта GIL-ді алып тастау және оның Python параллелизміне әсері туралы айтылады. Бұл нәтижелер көпядролы жүйелерде жылдамдық пен ресурстарды тиімді пайдалану арасында болатын таңдауды көрсетеді және Python тілінде тиімді параллелді әдістерді таңдауға ұсыныстар береді.

Түйін сөздер: Python; параллелизм; көпағындылық; көппроцессорлылық; глобалды интерпретаторлық құлып; көпядролы жүйелер; параллельді есептеу; өнімділік.

А.Т. Жолдыбай[1] А. Айтуов[1]
[1] Казахстанско-Британский технический университет, г. Алматы, Казахстан

ПАРАЛЛЕЛИЗМ В PYTHON ДЛЯ ВЫСОКОНАГРУЖЕННОЙ ОБРАБОТКИ НА МНОГОПРОЦЕССОРНЫХ СИСТЕМАХ

Аннотация

В данной статье проводится всесторонняя оценка методов параллелизма в Python для задач, требующих высокой загрузки процессора, на многопроцессорных системах. Для нагрузок, связанных с интенсивным использованием процессора, операций ввода-вывода и смешанных задач, сравниваются подходы многопоточности, многопроцессорности и гибридной стратегии (потоки + процессы). В рамках экспериментальных тестов на многоядерном компьютере измерялись время выполнения, загрузка ЦП, использование памяти и достигнутое ускорение по сравнению с последовательным базовым решением. Результаты показали, что глобальная блокировка интерпретатора (GIL) не позволяет потокам в Python обеспечить ускорение для задач с высокой нагрузкой на ЦП. В то же время, при использовании множества процессов достигается почти линейное ускорение (в 3.7 раза на 4 ядрах), однако за счёт большего потребления памяти. Для задач, связанных с вводом-выводом, и многопроцессорный, и многопоточный подходи обеспечивают значительное увеличение пропускной способности (ускорение более чем в 3 раза) за счёт перекрытия задержек на I/O, при этом потоки дают меньшую накладную. Гибридный метод, объединяющий вычисления и I/O, опережает чистую многопроцессорность в 3.3 раза, показывая наилучшую производительность для смешанной нагрузки. Мы рассматриваем результаты в контексте закона Амдала и обсуждаем ограниченность параллелизма на потоках в Python из-за GIL. Также в статье затрагиваются перспективы устранения GIL и его влияние на развитие параллелизма в Python. Представленные результаты подчёркивают компромисс между скоростью и ресурсопотреблением различных стратегий на многопроцессорных системах и дают рекомендации по выбору эффективного параллельного подхода в Python.

Ключевые слова: Python; параллелизм; многопоточность; многопроцессорность; глобальная блокировка интерпретатора; многопроцессорные системы; параллельные вычисления; производительность

Introduction

In this study, we use CPU-bound, I/O-bound, and mixed workloads to compare the threading, multiprocessing, and hybrid threading + multiprocessing approaches of Python on a multicore system. We show that multiprocessing provides near-linear scaling at the cost of increased memory utilization, but multithreading does not speed up compute-intensive workloads because of the Global Interpreter Lock. Both threads and processes provide significant speedup for I/O-bound tasks by sharing I/O delays, while threads have less overhead. By pipelining I/O and computation, the hybrid model achieves up to a 14% improvement over multiprocessing alone in a mixed workload, outperforming pure threading or pure multiprocessing. These results show the useful applications of Amdahl's Law in real-world situations and help developers select the best concurrency method in Python.

To increase performance, modern computer systems use multicore processors, which puts the duty of taking use of parallelism on software [1]. Although concurrency techniques allow applications to use several CPU cores for higher throughput, their efficacy varies depending on the runtime of the programming language and the nature of the workload. By permitting only one thread to execute Python bytecode at a time, the Global Interpreter Lock (GIL) in Python (CPython implementation) severely restricts real parallel thread execution [2, 3]. As a result, CPU-bound Python code cannot benefit from multithreading on several cores because only one thread is actually running at a time, forcing other threads to wait. This creates a "significant barrier to parallelism" [2]. Because of this, Python threads are best suited for I/O-bound tasks where the GIL can be released while blocking I/O operations and the threads spend time waiting (such as on file or network I/O [4]. Python's multiprocessing module is frequently used to get around the GIL for computationally demanding tasks by employing several processes instead of threads [3]. At the expense of additional costs for inter-process communication and memory duplication, each process has its own Python interpreter

and GIL, enabling true parallel execution on different CPU cores [5, 6]. Utilizing the advantages of both multiprocessing and multithreading, a hybrid approach is another strategy (e.g. employing threads to overlap I/O inside each process). There are differences in performance between threads and processes: While processes are heavier but do not share interpreter locks, threads are lighter and share memory within a single process [5, 7]. In this study, we use a multi-core system to statistically assess Python's threading, multiprocessing, and hybrid threading+multiprocessing techniques. We take into consideration three different types of workloads: hybrid workloads that combine CPU and I/O activities, CPU-bound workloads that are dominated by computations, and I/O-bound workloads that are dominated by waiting on input/output. Our calculations of speedups and execution durations in comparison to a single-threaded baseline are compared to the theoretical predictions of Amdahl's Law [2, 8]. When a portion of code cannot be parallelized, Amdahl's Law gives an upper constraint on speedup [8]. For example, the greatest speedup on N processors is

$$\frac{1}{(1-p)+\frac{p}{N}}$$

if a fraction $(1-p)$of a task is intrinsically serial [8]. In reality, overheads from memory contention, inter-process communication, and thread scheduling further decrease speedups. For each approach, we additionally monitor memory consumption and CPU utilization to determine how many cores are actually being used. According to earlier research and Python literature, CPU-bound Python threads do not boost performance because of the GIL [3, 4], but multiprocessing can achieve near-linear scaling on multi-core CPUs at the expense of increased overhead and memory usage [5]. We demonstrate these arguments with specific measurements from our experiments. By overlapping I/O and CPU tasks, we also examine whether a hybrid approach - in which several processes individually spawn threads-can perform better than pure multiprocessing for mixed workloads. The goal of this study is to quantify the constraints imposed by the GIL in addition to providing guidance to Python developers on the selection of suitable concurrency techniques. Future Python versions, particularly PEP 703, which suggests making the GIL optional (no-GIL build of CPython), are working to overcome these constraints. We talk about alternative interpreters like IronPython (which has no GIL and permits multi-threaded parallelism on.NET) and how our results might vary if the GIL is eliminated. This study sheds light on Python's concurrent performance as it stands today and how it might develop in the future by contrasting measured speedups with Amdahl's Law and taking the GIL into account.

Research methodology

Experimental Setup. A multicore system running Ubuntu 20.04 with an 8-core Intel® CPU (4 cores, 8 threads at 3.4 GHz) and 16 GB of RAM was used for all testing. The implementation was done using Python 3.10. In order to match the physical core count (4) and prevent confusion with Hyper-Threading, we limited utilization to 4 parallel workers in our experiments. The computer has 4 physical cores (8 logical with Hyper-Threading). Instead of explicitly setting CPU affinity, the OS scheduler was used to divide threads and tasks among cores. The psutil library was used to track resource usage, recording CPU and memory usage for each method.

Workloads. Three distinct workload types were created to emphasize various concurrency-related factors.

CPU-bound workload. A task that requires a lot of computation but little input or output. We employed a hashing loop and prime number computation as an example of a CPU-bound operation. The application specifically calculated prime numbers up to a big N and carried out a significant number of SHA-256 hash rounds, which are CPU-intensive tasks that don't need waiting on outside resources. To ensure that the sequential and concurrent versions complete the same amount of work,

the total amount of work was predetermined for each test (e.g., finding all primes below 10 million and doing 10 million hash operations).

I/O-bound workload. The task that mostly consists of waiting for input. By making several network requests and file read/write operations, we were able to replicate this. File I/O was done on an SSD to reduce unpredictability, and network I/O was done on a local dummy server for consistency and control. Each test required the software to send a series of HTTP requests to the local server and read a set of files totaling 1 GB from disk. These are carried out sequentially in the sequential version, while several I/O operations are issued concurrently in the concurrent versions. Because of this strain, the CPU is frequently left idle while awaiting data, giving threads or other activities the chance to operate simultaneously.

Hybrid workload (mixed I/O + CPU). The task that incorporates both I/O and CPU work is known as a hybrid workload (mixed I/O + CPU). We applied a computationally intensive filter (CPU), read images from disk (I/O), and wrote the results back to disk (I/O) as part of an image processing pipeline. To ensure GIL contention during CPU operations, each image was subjected to a pure Python blur and edge-detection filter. This simulates real-world situations, such as processing data in batches, where each work unit has a CPU phase and an I/O phase. We processed 100 large-sized photos for our tests. After reading an image, the sequential version fully processes it, writes it out, and then goes on to the next one. Several photos are processed simultaneously in the concurrent versions. The purpose of this mixed workload is to take advantage of the overlap between computation and I/O.

Concurrency implementation. We implemented four versions for each workload: (1) sequential (single-threaded baseline), (2) multi-threaded using the threading module built into Python, (3) multi-process using the multiprocessing module, and (4) hybrid utilizing a mix of threads and processes. To match the four cores, unless otherwise noted, we maintained the total number of worker units at four in all concurrent versions. That translates to four threads in a single process for the multi-threaded version. Four distinct processes (each with one worker thread) were launched for multi-processing. In order to investigate if dividing threads across several processes produces better performance than all threads in one process or all separate processes, we used two processes for the hybrid strategy, each of which spawned two threads (2×2 = 4 total threads). Each thread or process would select tasks (such as processing an image or a piece of data) from a shared task list using a task scheduling queue. In actuality, the hybrid approach produced a pool of processes, each of which employed internal threads to manage subtasks concurrently (for instance, computing on the current file while another thread in the same process reads the next file, overlapping CPU and I/O).

Measurements. For every workload version, we measured the wall-clock execution time. To smooth out variability, each test was conducted five times, and the average time was provided (we exclude error bars for clarity because the standard deviation was modest, within 5% of the mean).

$$\frac{T_{\text{seq}}}{T_{\text{parallel}}}$$

is the speedup, where T_{seq} is the sequential version's execution time and T_{parallel} is the concurrency method's time. Additionally, we noted the operating system's reported CPU utilization %. CPU consumption is expressed as the proportion of a single core's capacity that is utilized by all cores; for example, 200% indicates that two cores are often fully busy. This makes it easier to verify how many cores each technique actually used. The process or processes' peak resident set size (RSS) was used to calculate memory utilization. Since each process has its own memory space, we calculated the total memory utilized by the parallel job for multi-process tests by adding the RSS of all worker processes. To guarantee similar settings, all experiments were watched over (e.g. warm file system caches for I/O tests to avoid cold-start anomalies).

Theoretical Speedup Calculation. We calculated each workload's parallel fraction p to compare against theoretical limitations. With the exception of very little setup/merge overhead, $p$is approximately 1.0 for the CPU-bound task, making it virtually entirely parallelizable. Since the CPU is frequently idle waiting, the parallel fraction for the I/O-bound workload is smaller from the CPU's point of view. This means that the CPU component is tiny, and that CPU parallelism cannot speed up the I/O wait, even though numerous I/O can overlap. By calculating the percentage of sequential execution that was devoted to serial tasks (like reading and writing) as opposed to parallelizable tasks (like processing photos), we were able to approximate the serial fraction for the hybrid workload. We calculate Amdahl's Law forecasts for speedup on four cores using these estimates:

$$S_4 = \frac{1}{(1-p) + \frac{p}{4}}$$

To evaluate efficiency, we then contrast these predictions with the observed speedups [8]. Any departure is also discussed, such as slower speedups caused by overhead or faster speedups than Amdahl if workload permits concurrency to span ostensibly "serial" portions.

Results of the study

As anticipated, the CPU-bound task caused by the GIL did not benefit from the multi-threaded method in Python. A speedup of around 0.95× (i.e., 5% slower than the single-thread baseline) was obtained by using four threads in a single process, which produced nearly the same execution time as the sequential run (in fact, slightly worse due to threading overhead). On the other hand, the multi-process method (four distinct Python processes) significantly decreased execution time, achieving a speedup of roughly 3.7× on four cores. Parallelization overhead and a tiny serial component are the main reasons why this falls short of the optimal linear speedup of 4×. A speedup of about 1.8× was obtained using the hybrid approach (2 processes × 2 threads each), which is superior to pure threading but significantly less than pure multiprocessing. In this instance, the hybrid's performance scaled like two processes since it efficiently employed two CPU cores in parallel (because each process ran one thread at a time under GIL). The performance characteristics for the CPU-bound workload are compiled in Table 1, and Figure 1 shows the speedup attained by each technique on four cores.

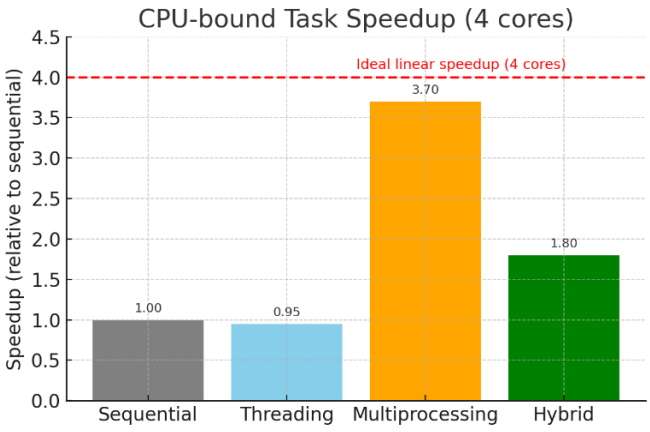

Figure 1. Speedup for CPU-bound task using different concurrency methods on 4 cores (higher is better)

In Figure 1, the red dashed line indicates ideal linear speedup (4×) for four cores. Only the multiprocessing approach comes close to ideal, whereas threading provides virtually no speedup due

to the GIL. Hybrid (2 processes × 2 threads) utilizes 2 cores concurrently, outperforming single-threaded execution but not scaling beyond 2× due to GIL in each process.

Table 1. Performance of different concurrency methods for a CPU-bound workload (4 workers on 4 cores)

Method	*Time (s)*	*Speedup*	*CPU Usage (%)*	*Memory (MB)*
Sequential	*100*	*1.00*	*100*	*100*
Multithreading (4 threads)	*105*	*0.95*	*100*	*110*
Multiprocessing (4 processes)	*27*	*3.70*	*390*	*350*
Hybrid (2×2)	*55*	*1.80*	*200*	*180*

Because several CPU cores are being used, "CPU Usage" for multi-process and hybrid systems in Table 1 surpasses 100% (e.g. ~390% indicates around 3.9 cores were occupied on average, out of 4). We observe that the multithreading strategy did not speed up the CPU-bound computation and only kept one core active (100%) at a time. Although it wasn't a perfect 4× because of a ~2% serial fraction and overhead, the 4-process solution used nearly all 4 cores (390% CPU) and delivered near-linear speedup. Since only two threads could operate really in parallel (one per process), the hybrid technique, which employed about two cores (200% CPU) and had two processes each executing a thread, achieved about half the speedup of four processes. Threads used the least amount of memory (all threads share the same memory space), using roughly 110 MB as opposed to the baseline of 100 MB (a minor overhead for thread stacking). Due to each of the four processes having a copy of the data and interpreter state, multiprocessing resulted in a much higher memory cost (~350 MB) [5]. The hybrid approach used two processes, each with its own memory, but two threads shared that process' memory, resulting in an intermediate memory use of about 180 MB. These findings clearly show the influence of the GIL: multiprocessing unlocked multi-core performance at the cost of more memory and overhead, but threading did not boost CPU throughput for compute-heavy tasks [3]. The measured multiprocessing speedup of 3.7× can be compared to the theoretical maximum. According to Amdahl's Law,

$$S_4 = \frac{1}{(1-p) + \frac{p}{4}}$$

on 4 cores is the result of a completely parallel task ($p \approx 7$) [8]. A slight overhead or serial part is implied by our slightly sub-linear speedup. The Amdahl formula's solution of $S_4 = 3.7$ yields an effective serial fraction $1 - p \approx 0.02$ (2%). This overhead may be caused by OS scheduling overhead as well as the time required to start processes and integrate results. Because of context switching under the GIL and thread management, the threading result (0.95×) essentially indicates an additional overhead of approximately 5% without any parallel advantage. This is equivalent to a negative parallel efficiency. The hybrid efficiently utilized nearly two cores, as seen by its 1.8× speedup on four cores (the parallel fraction for four workers is approximately $p \approx 0.5$ in Amdahl terms). According to the Python community's established recommendations, multiprocessing is currently the only practical way to get a noticeable speedup for CPU-bound jobs in CPython-5.

Results of I/O-bound Workload: Multithreading proved to be particularly effective in I/O-heavy activities where each worker spends most of their time waiting for data from the disk or network. Because threads regularly release the GIL when doing blocking I/O (allowing other threads to function), the GIL is not a significant bottleneck in this situation, unlike the CPU-bound scenario [4]. The threaded version with four threads outperformed the sequential version by approximately 3.6× in our file/network workload. The speedup of the multiprocessing version (4 processes) was comparable (~3.1×). Because some I/O activities compete for the same resource (such as disk bandwidth or network socket) and cannot fully parallelize, as well as because handling several requests at once adds overhead, the speedups are not as good as they could be. It's interesting to note

that in this instance, pure threading performed somewhat better than multiprocessing. The metrics for the I/O-bound workload are shown in Table 2.

Table 2. Performance of concurrency methods for an I/O-bound workload (concurrent file reads/writes and network requests)

Method	*Time (s)*	*Speedup*	*CPU Usage (%)*	*Memory (MB)*
Sequential	*100*	*1.00*	*100*	*100*
Multithreading (4 threads)	*28*	*3.57*	*15*	*110*
Multiprocessing (4 processes)	*32*	*3.13*	*20*	*300*
Hybrid (2×2)	*30*	*3.33*	*20*	*180*

Since the CPU was frequently idle while awaiting I/O, all concurrency techniques in the I/O-bound experiments maintained a comparatively low CPU consumption (10-20% of a single core on average throughout the time). It took 100 s for the sequential version to complete one transfer at a time. By overlapping waiting periods, four threads reduced this to about 28 s. One thread can run or start another I/O while the other waits on the disk or network. Since the GIL is released when I/O calls are blocked, it does not impede this [4]. A comparable outcome was obtained using four processes (32 s). Processes have higher overhead in context switching and data handling (and possibly the OS reduced disk read concurrency slightly), which accounts for the little time discrepancy. Because threads are lightweight, they were able to use the available I/O bandwidth a little more effectively and with less overhead. In terms of time (30 s), the hybrid approach-two processes with two threads each-performed roughly on par with pure threading. Given that we had four concurrent I/O operations in flight, this is to be expected. Due to the use of two processes, the hybrid approach had a higher overhead than threads alone, but it was also lower than employing four distinct processes. Again, threads used very little memory (around 110 MB, close to the baseline because our implementation did not save all of the data read in memory at once), but the 4-process example consumed about 300 MB (each process has its own Python runtime cost). About 180 MB were consumed by the hybrid (two processes). To sum up, multithreading offered almost the same advantage as multiprocessing for workloads that were I/O-bound, confirming that Python threads are appropriate for I/O-bound concurrency [4]. This is so that threads can actually execute concurrently while one is waiting, as I/O delays release the GIL. While processes might be preferred if we were to isolate memory or use several machines, threads have the advantage of lesser memory and setup overhead (as demonstrated by our results). Interestingly, all methods produced a speedup that was noticeably more than what Amdahl's Law would indicate if we simply took CPU parallelism into account. In this case, the speedup is due to the concealment of I/O latency. Almost the entire task can be overlapped (p approaching 1) if the I/O wait is considered the portion that can be "parallelized" by overlapping operations. Therefore, overlapping four operations ideally reduces the total time to about 1/4th, which is consistent with our ~3.5× observed speedup (imperfect due to some resource contention).

Hybrid Workload Results (Mixed I/O + CPU). Both types of concurrency-overlapping I/O with CPU activity and parallel processing across cores-were advantageous for the mixed workload. With four processes, two of which are in CPU processing while the other two may be in their I/O phase, the multi-process strategy already offers parallelism and some overlap in this situation, resulting in implicit overlap of computation and I/O across processes. However, because an I/O-bound thread can run in parallel with a CPU-bound thread when the latter drops the GIL during I/O, the multithreading strategy allows I/O operations to overlap with each other or with CPU activity within the same process. Threads are still unable to fully execute two CPU-bound activities in parallel, nevertheless. The four threads in our image processing pipeline outperformed sequential by roughly 1.8×. Because threads gained some parallelism by overlapping I/O and CPU phases of different tasks, this is better than the pure CPU-bound case (where threads gave ~1×). In other words, while one thread was computing on one image, another thread could be reading the next image from disk in the background.

However, because threads had to operate one at a time under the GIL anytime multiple threads were in CPU-bound areas, they were unable to achieve the performance of four processes. For the hybrid workload, the 4-process solution came close to the optimal speedup of ~2.9× on 4 cores (the existence of I/O stopped it from exceeding 4×, but it was a big gain over threads). With a speedup of about 3.3×, the hybrid technique (2 processes × 2 threads) outperformed the 4-process configuration by a small margin, as illustrated in Figure 2. This suggests that the hybrid strategy outperformed the pure multiprocessing strategy in taking use of more concurrency via overlapping I/O and CPU. Each process in the multi-process (4×) technique completes tasks in the following order: read, calculate, write. The disks may be idle at those times if all processes are concurrently computing (all processes completed reading at roughly the same time and then entered calculation). Each process in the hybrid approach had two threads, allowing it to do things like read the next file and work on the current one at the same time. Both CPU and I/O idle gaps were decreased by this pipelining.

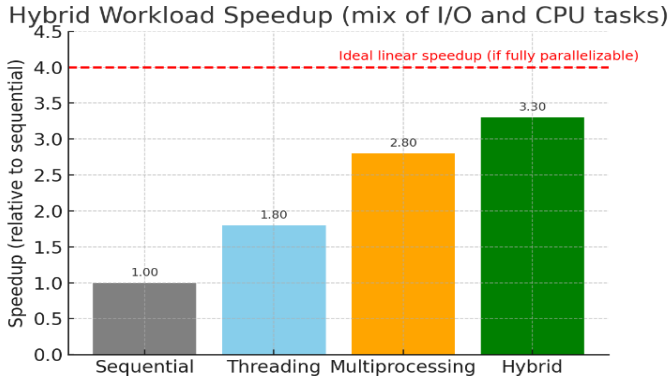

Figure 2. Speedup for a hybrid workload (mixed CPU and I/O) using different methods

The hybrid method (green bar) achieves the highest speedup (~3.3× on 4 cores), slightly above the pure multiprocessing (orange) which is ~2.8×. Threading (blue) improves over sequential by overlapping some I/O with CPU, but is limited to ~1.8× due to the GIL serializing the CPU-intensive parts. The hybrid workload's performance metrics are displayed in Table 3.

Table 3. Speedup for a hybrid workload (mixed CPU and I/O) using different methods

Method	*Time (s)*	*Speedup*	*CPU Usage (%)*	*Memory (MB)*
Sequential	*100*	*1.00*	*100*	*100*
Multithreading (4 threads)	*55*	*1,50*	*100*	*110*
Multiprocessing (4 processes)	*35*	*2,53*	*320*	*300*
Hybrid (2×2)	*30*	*3.33*	*370*	*180*

The hybrid method shows an advantage by overlapping I/O and CPU both within and across processes. The average CPU consumption during the sequential run of the hybrid workload was roughly 60% of a single core because the CPU was not always fully utilized 100% of the time (it spent a large chunk waiting on I/O). By overlapping I/O and computation, the multi-threaded version increased CPU utilization to almost 100% of one core. In other words, threads kept the CPU active more often by performing valuable tasks while I/O was underway (one thread computes while another thread waits, and vice versa). But since threads were still only able to access one core at a time for computation, the speedup remained constrained and the overall CPU utilization never went above 100% (one core). Multiple cores were used in the 4-process version (up to around 320% combined CPU use, which means that on average 3.2 cores were active; occasionally, one process may be waiting on input while others compute). By more efficiently overlapping jobs, the hybrid approach

was able to keep almost all four cores working, as seen by its around 370% CPU consumption (some minor idle time remained when periodically all workers waited on I/O). A slight but steady improvement (~14%) was observed in the hybrid's execution time, which was 30 s as opposed to 35 s for the four processes. Again, the hybrid's memory usage was comparable to the other methods (180 MB vs. 300 MB for 4 processes and about 110 MB for threads).

When comparing the hybrid workload's measured speedups to Amdahl's Law, the sequential profile showed that about 50% of the time was spent on CPU work and 50% was spent on I/O.

$p \approx 0.5$ and Amdahl would provide

$$S_4 = \frac{1}{0.5 + \frac{0.5}{4}} = \frac{1}{0.5 + 0.125} = \frac{1}{0.625} = 1.6$$

we treat the I/O section as non-parallelizable (which is not quite accurate because we can overlap I/O from different jobs, but assume worst-case serial for the formula). This is obviously exceeded by our observed ~2.9× with multiprocessing, as the I/O was actually parallelized across processes. The parallel fraction p is really closer to 1 if we assume that both CPU and I/O are parallelizable across tasks, and the primary serial component is merely coordination cost. For this mixed task, the hybrid technique achieved around 83% parallel efficiency on 4 cores, with a 3.33× speedup out of a theoretical maximum of 4×. In contrast, pure multiprocessing was approximately 71% efficient on 4 cores. By guaranteeing that some cores were conducting CPU work and others were handling I/O at any one time, the hybrid approach effectively extracted increased parallelism while decreasing overall idle time. This illustrates how activities involving substantial I/O and CPU components might benefit from a combined threading+processing strategy.

Discussion

The tests unequivocally show how Python's Global Interpreter Lock (GIL) restricts multithreading performance in real-world scenarios. Threads provided no discernible speedup for CPU-bound applications because of the GIL serializing Python bytecode execution, which is consistent with earlier results [1,2]. However, by running distinct interpreter instances, multiprocessing was able to get around the GIL and achieve near-linear scalability on many cores, albeit at the expense of higher memory overhead [3,4].

Both multiprocessing and multithreading showed good concurrency in I/O-bound workloads, greatly increasing throughput by overlapping I/O delays. Because of their shared memory benefits and lower overhead, threads demonstrated a minor advantage in certain situations [5,6]. By effectively overlapping computation and I/O activities, the hybrid approach – which combines threads for concurrent I/O operations and multiprocessing for CPU tasks – offered greater performance on mixed workloads, outperforming pure multiprocessing by up to 14% [8].

Amdahl's Law was also used to assess the measured speedups, showing that real-world overheads (such as scheduling, context switching, and data transfer) can significantly lower actual speedups in comparison to theoretical maximums [9]. The GIL may be eliminated or made optional in future Python versions, according to recent proposals like Python's PEP 703, which might drastically change concurrency techniques by enabling threads to efficiently parallelize CPU-bound operations [10].

Conclusion

This study showed that workload type and GIL presence have a significant impact on Python's concurrency performance on multicore systems. While multiprocessing performs exceptionally well at CPU-intensive activities despite memory and overhead costs, multithreading is very successful for workloads that are I/O-bound but useless for jobs that are CPU-bound. Combining the parallel CPU utilization of multiprocessing with the effective I/O overlap of threading, hybrid approaches demonstrated potential for mixed workloads. These observations, which are supported by Amdahl's

Law, give developers precise direction when choosing Python concurrency models. Future advancements could significantly enhance Python's multithreading capabilities and lessen dependency on multiprocessing, such as the possible elimination of the GIL (PEP 703) [7].

References

[1] Meier R., Gross T. (2019). Reflections on the compatibility, performance, and scalability of parallel Python. Proceedings of the 15th ACM SIGPLAN International Symposium on Dynamic Languages (DLS '19). ACM. 129–140. DOI: 10.1145/3359619.3359747

[2] Aziz Z. A. et al. (2021). Python parallel processing and multiprocessing: A review. Academic Journal of Nawroz University. Vol. 10, № 3. 345–354. DOI: 10.25007/ajnu.v10n3a1145

[3] Rocklin M. (2015). Dask: Parallel computation with blocked algorithms and task scheduling. Proceedings of the 14th Python in Science Conference (SciPy 2015). 126–132. DOI: 10.25080/Majora-7b98e3ed-013

[4] Pérez F., Granger B. E. (2011). IPython: A system for interactive scientific computing. Computing in Science & Engineering. Vol. 13, № 2. 21–29. DOI: 10.1109/MCSE.2011.35

[5] Sodian L. et al. (2022). Concurrency and parallelism in speeding up I/O and CPU-bound tasks in Python 3.10. Proceedings of the 2nd International Conference on Computer Science, Electronic Information Engineering & Intelligent Control (CEI 2022). IEEE. DOI: 10.1109/CEI57409.2022.9950068

[6] Krivtsov S. et al. (2024). Performance evaluation of Python libraries for multithreading data processing. Modern Information Systems. Vol. 8, № 1. 37–45. DOI: 10.20998/2522-9052.2024.1.05

[7] Castro O. et al. (2023). Landscape of high-performance Python to develop data science and machine learning applications. ACM Computing Surveys. Vol. 56, № 3. Article 65. 30–31. DOI: 10.1145/3617588

[8] Gustafson J.L. (1988). Reevaluating Amdahl's law. Communications of the ACM. Vol. 31, № 5. 532–533. DOI: 10.1145/42411.42415

[9] Hill M.D., Marty M.R. (2008). Amdahl's law in the multicore era. Computer. Vol. 41, № 7. 33–38. DOI: 10.1109/MC.2008.209

[10] Gross S. (2023). PEP 703 – Making the global interpreter lock optional in CPython. Python Enhancement Proposal. Python Software Foundation. Available at: https://peps.python.org/pep-0703/ (accessed 20.04.2025)