THE USA JOURNALS
THE AMERICAN JOURNAL OF ENGINEERING AND TECHNOLOGY (ISSN – 2689-0984)
VOLUME 06 ISSUE12

PUBLISHED DATE: - 10-12-2024
DOI: - https://doi.org/10.37547/tajet/Volume06Issue12-06 **PAGE NO.: - 51-58**

RESEARCH ARTICLE **Open Access**

USING ASYNCHRONOUS PROGRAMMING IN PYTHON TO IMPROVE APPLICATION PERFORMANCE

Kokalko Mykola
Senior Software Engineer at Uvik, Hollywood, FL, USA

Abstract
Asynchronous programming in Python is a powerful tool for improving application performance by effectively managing multitasking. The main focus is on the use of the async and await keywords, as well as the asyncio, ThreadPoolExecutor and ProcessPoolExecutor libraries, which play an important role in organizing multitasking processes. This is especially true for applications related to I/O operations, such as web servers and APIs. The use of asynchronous programming allows you to eliminate thread locks and optimize query processing, which makes applications more responsive and scalable. The article provides specific examples of using the asynchronous approach in practice, including parallel task execution and resource management. In addition, the study demonstrates that the introduction of asynchronous technologies helps to reduce infrastructure costs, ensuring high throughput and stable operation even under high load conditions. Asynchronous programming stands out for its flexibility and the ability to create high-performance systems that are capable of handling a large number of simultaneous I/O operations.

Keywords Asynchronous programming, Python, asyncio, ThreadPoolExecutor, performance, parallelism, multitasking, optimization.

INTRODUCTION

Asynchronous programming is becoming one of the key approaches to optimizing application performance in the modern era of technological advancement. With the increasing volume of data and the growing number of users interacting with systems in real-time, the need to process multiple requests concurrently has emerged. Traditional methods of sequential task execution are not always capable of handling high loads, leading to application slowdowns and reduced efficiency. In this context, asynchronous technologies have gained particular relevance as they allow for the simultaneous execution of multiple operations without blocking the program's main thread.

The relevance of this topic is driven by the growing demand for high-performance applications capable of handling large data volumes and supporting simultaneous interactions with numerous users. Asynchronous programming is one solution that enables developers not only to speed up task execution but also to reduce server resource costs by efficiently managing parallel processes. This is especially crucial for developing web services, APIs, analytics systems, and applications operating in real-time.

The purpose of this study is to explore methods and tools for asynchronous programming in Python, as well as to analyze their practical application for enhancing application performance.

METHODS

In a broad sense, asynchronous programming in Python involves executing requests without blocking while awaiting their completion. Asynchronous programming in Python can be implemented using various methods, some of which are particularly relevant for concurrency in Python [1].

Hunt J. argues [2] that asynchronous programming in Python provides an efficient model for handling multiple tasks that can run concurrently without blocking the main execution of the program. Asynchronous tasks involving input-output operations can be effectively implemented with `asyncio.sleep()`. It is important to note that the completion order of functions may vary depending on how the asyncio event loop schedules tasks. This behavior is crucial for task management optimization, which is especially relevant when working with asynchronous iterators.

A comparison of asynchronous and multithreading approaches highlights fundamental differences. Asynchronous programming allows only one part of the program to execute at any given moment. For example, if function `fn1()` is temporarily paused (e.g., waiting), it does not block the entire program, allowing other functions, such as `fn2()`, to operate during that time. Thus, the system efficiently uses CPU resources, reallocating time to other tasks. In multithreading, however, all functions, such as `fn1()`, `fn2()`, and `fn3()`, run simultaneously and can execute in parallel without waiting for each other's completion [2].

Ganji M., Alimadadi S., and Tip F. emphasize [3] that in Python, the keywords `async` and `await` play a critical role in asynchronous programming.

In Hielscher M. M.'s article [4], the development of LABS—a modular, open-source Python solution—is discussed. The software is designed for automation and process scheduling in laboratory settings, specifically for managing automated synthesis via a web interface. The article explores how the modular structure enables configuration and management of processes, ensuring flexibility and scalability.

In Potashov A. V.'s work [5], the specifics of using Python for backend development of high-load web applications are presented. The author highlights the importance of selecting appropriate tools and architectural solutions to optimize the handling of large data volumes and ensure stability under high loads. The focus is on the use of asynchronous technologies and parallel computations.

In the work by Wilkes M. and Wilkes M., "Parallelization and Async" [9], the specifics of parallel and asynchronous programming in real-world applications are examined. The authors emphasize the advantages of using parallelism to improve program execution efficiency and provide examples of solving complex computational tasks.

The practical application of FastAPI described in [6] demonstrates Python's asynchronous capabilities to ensure fast and efficient responses, making it suitable for high-load applications. The "Databases" project on GitHub [8] offers asynchronous database support for Python, easily integrating with frameworks like FastAPI and AIOHTTP. This enables non-blocking database operations, which is crucial for maintaining responsiveness in asynchronous applications. The development of real-time applications using Python and WebSocket is discussed in an article on the Habr platform [10], describing how asynchronous programming paradigms are essential for real-time data handling between clients and servers, ensuring low latency in communication.

The resource "Asynchronous Processing in System Design" [11] provides an overview of asynchronous processing concepts in system architecture. It highlights the benefits of task

decomposition and concurrent operation processing to enhance system throughput. The source [12] describes the use of asyncio.Queue for inter-task communication in asynchronous applications, explaining how queues can coordinate work between producer and consumer coroutines to create efficient data processing pipelines.

Collectively, the reviewed literature underscores the significance of asynchronous programming in modern Python development. The integration of asynchronous frameworks and tools broadens the possibilities for creating high-performance, scalable, and responsive applications. Ongoing research and practical implementations continue to expand horizons and address the challenges associated with asynchronous code execution.

RESULTS AND DISCUSSION

Asynchronous programming, due to its flexibility and efficiency, has become one of the key tools for creating high-load and scalable systems. When a function is declared with the `async` modifier, it becomes a coroutine, capable of suspending and resuming execution as needed. The `await` operator is used to call other coroutines and manage parallel operations efficiently within an application. Table 1 below reviews the application of `async` and `await` operators.

Table 1. Application of async and await operators [3]

Aspect	async	await
Function Type	Marks a function as a coroutine.	Used within a coroutine to call other coroutines.
Execution Control	Can suspend and resume the coroutine during execution.	Suspends the current coroutine, allowing other tasks to run.
Purpose	Essential for asynchronous programming.	Used to wait for the completion of asynchronous operations.
Event Loop	Allows the event loop to switch to other tasks while waiting for operations.	Enables the event loop to handle other tasks during waiting.
Parallelism Management	Ensures efficient management of parallel processes.	Provides seamless handling of asynchronous operations.
Code Usage	Indicates that a function can be asynchronous.	Used within a coroutine to control the flow of asynchronous code.
Coroutine Interaction	Can define coroutines.	Used to call other coroutines within a coroutine.
Code Readability	Helps write readable asynchronous code.	Enhances code readability by managing asynchronous tasks.
Maintainability	Simplifies code by abstracting low-level concurrency constructs.	Improves code maintainability through efficient handling of

		asynchronous tasks.

The Asyncio library offers high functionality and flexibility, which has led to its inclusion in Python's standard library set. The programming language also integrated the `async` and `await` keywords, which are used to explicitly denote asynchronous code. These key constructs help to avoid confusion between asynchronous functions and generators. The `async` keyword is placed before a function declaration with `def`, indicating the asynchronous nature of the function. The `await` keyword signals that program execution is suspended until the coroutine completes. An example using the `async` and `await` keywords is presented below:

```
import asyncio
import aiohttp

urls = ['http://www.google.com', 'http://www.yandex.ru', 'http://www.python.org']

async def call_url(url):
    print(f'Starting request to {url}')
    response = await aiohttp.get(url)
    data = await response.text()
    print(f'{url}: data size {len(data)} bytes: {data}')
    return data

futures = [call_url(url) for url in urls]

loop = asyncio.get_event_loop()
loop.run_until_complete(asyncio.wait(futures))
```

In this code, the function marked as `async` returns a coroutine when called, which is executed asynchronously. The asynchronous nature of this method allows the program to await the completion of each request without blocking the main thread, significantly speeding up data processing [4].

The following Table 2 examines the potential of asynchronous programming in Python to enhance application performance.

Table 2. The use of asynchronous programming in Python to improve application performance [5]

Aspect	Description
Asynchronous Tasks (Coroutines)	Asynchronous programming uses coroutines that allow tasks to execute concurrently without blocking the main thread, reducing delays during input/output waits.

Event Loop	Asynchronous tasks operate within a single thread using an event loop, which manages task execution while minimizing context switching and overhead.
Non-blocking Input/Output (I/O)	Asynchronous programming enables resource management, such as database queries and network connections, without blocking other operations, enhancing performance in network applications.
Concurrency	Asynchronous programs allow multiple tasks to run simultaneously, efficiently using CPU time while waiting on I/O operations.
Support for asyncio Library	The built-in asyncio library provides convenient tools for managing asynchronous tasks, such as coroutine creation and event management.
CPU Usage Optimization	While asynchronous programming does not increase the number of available CPU cores, it allows for more efficient use of resources by avoiding idle time.
Minimization of Locks	Asynchronous programming reduces the need for thread locks, decreasing the likelihood of deadlocks and improving overall application stability and performance.
High-performance Network Support	Asynchronous programming is ideal for handling large numbers of network or API requests, as it manages connections and data without blocking.
Scalability	Asynchronous applications are easier to scale, as they can efficiently handle numerous concurrent connections and tasks.

One of the most popular applications of asynchronous programming is the development of high-performance web servers and APIs. Such servers handle thousands or even millions of requests per second, requiring efficient management of wait times for operations such as network requests or database interactions.

An example is the FastAPI framework, based on Python's asynchronous constructs, which has become a popular choice for building high-load REST APIs. By utilizing coroutines and an event loop, FastAPI can handle multiple connections simultaneously without blocking the main thread. For instance, when sending an HTTP request to retrieve data from a database, instead of waiting for a response, the system continues processing other requests, making the server significantly more responsive and efficient.

The practical benefit of using an asynchronous API is that server resources (CPU, memory) are utilized much more efficiently compared to traditional thread-based servers like Flask or Django, which require creating a new thread or process for each request. This reduces infrastructure costs and enables more users to be served on the same hardware [6].

The asynchronous framework aiohttp is used for building web clients and servers that support multiple simultaneous connections. Its architecture enables the creation of non-blocking HTTP services, which are especially beneficial for APIs handling incoming requests from other

systems. For example, microservices that work with multiple external APIs can efficiently manage requests without waiting for each to complete individually.

Asynchronous servers based on aiohttp can handle a large number of long-lived connections (e.g., WebSockets), which are used for real-time data exchange [7].

Asynchronous libraries for working with external systems, such as databases or other APIs, help to minimize delays caused by processing requests. In real-world applications where quick data retrieval and processing are crucial, asynchronous approaches significantly improve system performance and throughput.

Databases often represent a bottleneck in application performance. In the traditional approach, the application must wait for the database query to complete, blocking the thread and reducing the overall system performance. Asynchronous drivers like asyncpg (for PostgreSQL) and aiomysql (for MySQL) allow non-blocking query execution.

For example, in analytics systems where rapid processing of large data volumes is essential, asynchronous drivers enable multiple database queries to be executed in parallel without waiting for the previous ones to finish. This is particularly important for high-load applications, such as e-commerce platforms or business analytics systems, where minimizing wait time is critical for the overall functionality of the application [8].

Many modern applications integrate with various external services, such as payment systems, geolocation services, or other APIs. In such scenarios, asynchronous requests allow multiple requests to be sent and processed concurrently without waiting for each one to complete. For instance, the aiohttp framework enables multiple HTTP requests to be sent in parallel, gathering results as they complete without blocking program execution.

The practical benefit of this approach is that developers can build systems that efficiently integrate with external APIs without degrading the overall application performance. An example is a hotel booking system where the application sends requests to multiple external services to obtain data on available rooms and prices. Asynchronous programming minimizes wait times, enhancing the user experience [9].

Real-time applications, such as online chat systems, notifications, or games, actively use asynchronous libraries to support persistent connections with users. These applications need to maintain numerous active connections with minimal latency, making asynchronous approaches essential for building efficient systems.

Asynchronous libraries like websockets allow developers to create servers that handle WebSocket connections, supporting real-time, bidirectional communication with clients. This is particularly critical for applications requiring instant data transmission, such as trading platforms, online communication systems, or monitoring applications.

These approaches enable support for thousands of connections without the need to create a separate thread or process for each user. As a result, such applications can scale significantly better while maintaining minimal data transmission latency [10].

These mechanisms are also applied in real-time event-processing applications, such as monitoring systems or signal processing from IoT devices. For instance, in an industrial monitoring system collecting data from sensors, asynchronous approaches enable the parallel processing of numerous incoming messages and the execution of appropriate actions (e.g., notifications about

equipment malfunctions) [11].

In the field of Big Data processing, asynchronous methods are essential. In scenarios requiring the parallel processing of large volumes of data from external sources, asynchronous solutions can significantly speed up task execution. For example, when working with distributed file systems or data streams, asynchronous libraries allow for parallel request processing, minimizing idle time.

Asynchronous queue systems, such as asyncio.Queue, are widely used in applications that need to manage tasks coming from multiple sources. For example, in distributed data processing systems, asynchronous queues are used to transfer tasks between different system components, ensuring efficient load distribution among nodes. This increases throughput and reduces wait times [12].

Thus, asynchronous methods are indispensable for creating high-performance, scalable, and responsive systems.

CONCLUSION

The examination of asynchronous programming methods in Python has demonstrated its significant advantages in the context of developing high-performance and scalable applications. The use of key constructs `async` and `await`, along with specialized libraries like asyncio, enables effective multitasking and parallel operation management, ensuring smooth task execution without blocking the main thread. This is particularly important for systems that handle a large volume of I/O operations, where delays in data processing can negatively impact performance and user experience.

The practical application of asynchronous programming has shown that this approach can substantially improve application responsiveness and optimize server resource utilization. Web services, APIs, and real-time systems can handle numerous concurrent requests while avoiding issues associated with traditional multithreading. Asynchronous technologies also help reduce infrastructure load and ensure system scalability, a critical factor for modern business applications and services.

In conclusion, the adoption of asynchronous programming in Python application development significantly enhances their efficiency and resilience to high loads. This approach opens up extensive possibilities for creating complex and functional systems that can quickly adapt to changing operational conditions and meet user demands.

REFERENCES

1. signal — Set handlers for asynchronous events. [Electronic resource] Access mode: https://docs.python.org/3/library/signal.html (accessed 12.10.2024).
2. Hunt J. Concurrency with AsyncIO //Advanced Guide to Python 3 Programming. - Cham : Springer International Publishing, 2023. – pp. 493-503.
3. Ganja M., Alimadadi S., Tip F. Code coverage criteria for asynchronous programs //Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. - 2023. – pp. 1307-1319.
4. Hoelscher M. M. et al. LABS: Laboratory Automation and Batch Scheduling–A Modular Open Source Python Program for the Control of Automated Electrochemical Synthesis with a Web Interface //Chemistry–An Asian Journal. – 2023. – Vol. 18. – No. 14. – p. e202300380.
5. Potashov A.V. PYTHON for the server side of highly loaded web applications //BBK 1 N 34. – p. 2767.
6. FastAPI Basics. [Electronic resource] Access

mode: https://alex-podrabinovich.medium.com/html-%D0%BE%D1%81%D0%BD%D0%BE%D0%B2%D1%8B-fastapi-da782612d3e5(accessed 12.10.2024).

7. Welcome to AIOHTTP. [Electronic resource] Access mode: https://docs.aiohttp.org/en/stable /(accessed 12.10.2024).
8. Databases. [Electronic resource] Access mode: https://github.com/encode/databases (accessed 12.10.2024).
9. Wilkes M., Wilkes M. Parallelization and async //Advanced Python Development: Using Powerful Language Features in Real-World Applications. – 2020. – pp. 283-344.
10. Development of real-time applications with Python and WebSocket. [Electronic resource] Access mode: https://habr.com/ru/companies/otus/articles/770256 /(accessed 12.10.2024).
11. Asynchronous Processing in System Design. [Electronic resource] Access mode: https://tr-page .yandex.ru/translate?lang=en-ru&url=https%3A%2F%2Fwww.geeksforgeeks.org%2Fasynchronous-processing-in-system-design%2F(accessed 12.10.2024).
12. Asyncio Queue in Python. [Electronic resource] Access mode: https://superfastpython.com/asyncio-queue /(accessed 12.10.2024).