

## Programming Paradigm with time comparison

This report documents the implementation process for four programming paradigms: Synchronous, Multiprocessing, Multithreading, and Asynchronous. Each paradigm is used to fetch 100 UUIDs from a URL and the execution time is measured.

### Code snippet of the timer function:

The below timer function is a decorator that measures how long a function takes to run on average. It takes two arguments: number and repeat. It uses `timeit` to record execution times and prints the average. We should decorate a function with `@timer(number, repeat)` in synchronous, multiprocessing, multithreading and asynchronous file to get the timing.

```
program_timer.py X
program_timer.py > timer
1  import timeit
2
3
4  def timer(number, repeat):
5      def wrapper(func):
6          runs = timeit.repeat(func, number=number, repeat=repeat)
7          print(sum(runs) / len(runs))
8
9      return wrapper
```

### 1. Synchronous Paradigm

#### Code Snippet:

```
sync.py X
sync.py > main
1  import requests
2
3  from program_timer import timer
4
5  URL = "https://httpbin.org/uuid"
6
7  def fetch(session, url):
8      with session.get(url) as response:
9          print(response.json()["uuid"])
10
11 @timer(1, 1)
12 def main():
13     with requests.Session() as session:
14         for _ in range(100):
15             fetch(session, URL)
```

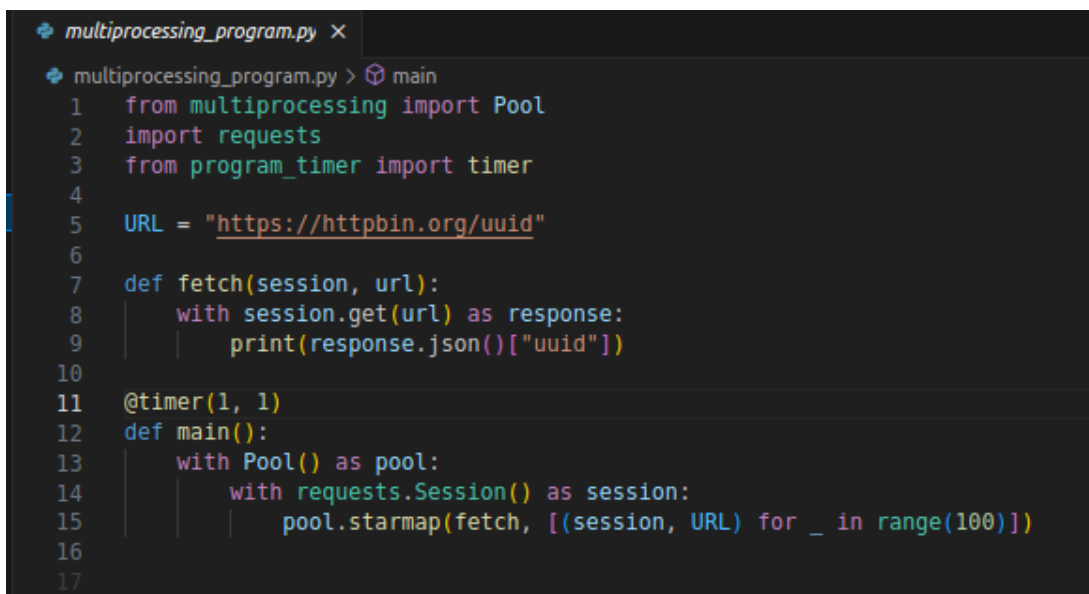
The `fetch()` function in the above snippet retrieves a UUID from the specified URL using a requests session. The `main()` function iterates 100 times, calling `fetch` for each iteration to retrieve all UUIDs. The `@timer(1, 1)` decorator above `main` measures the time taken to execute the entire `main` function, including all 100 fetches.

### Output:

The average time it took to fetch 100 UUIDs using synchronous approach is 32.337910691003344.

## 2. Multi-processing Paradigm

### Code snippet:

A screenshot of a code editor window titled 'multiprocessing\_program.py'. The code is as follows:

```
1 from multiprocessing import Pool
2 import requests
3 from program_timer import timer
4
5 URL = "https://httpbin.org/uuid"
6
7 def fetch(session, url):
8     with session.get(url) as response:
9         print(response.json()["uuid"])
10
11 @timer(1, 1)
12 def main():
13     with Pool() as pool:
14         with requests.Session() as session:
15             pool.starmap(fetch, [(session, URL) for _ in range(100)])
16
17
```

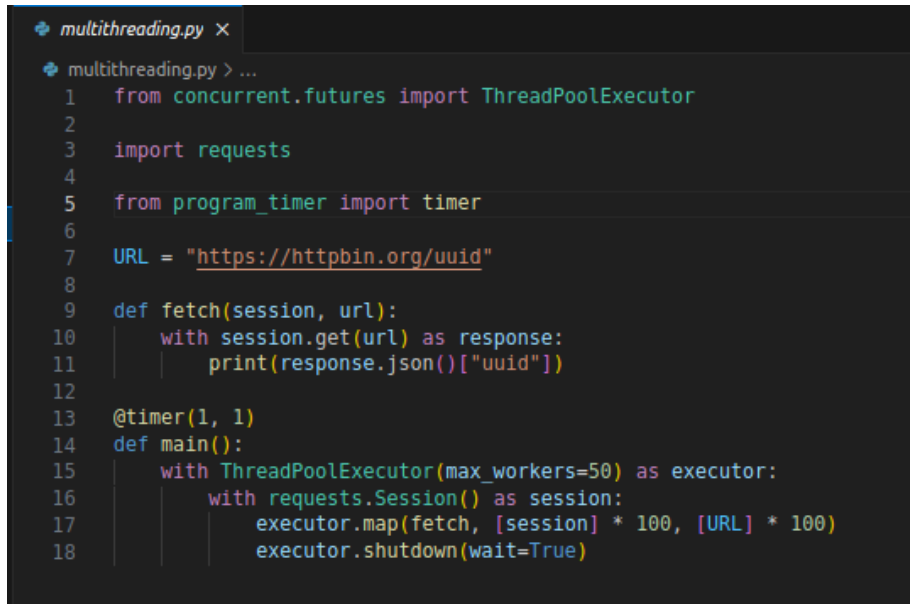
The above code uses the multiprocessing module to parallelize the execution of the `fetch` function across multiple processes. A pool of worker processes is created to handle tasks concurrently. The `starmap` function from multiprocessing is used to apply the `fetch` function to multiple arguments (session and URL) in parallel.

### Output:

Using multiprocessing approach time taken to fetch data is 19.38998791799895 seconds, which is fast as compared to the synchronous approach (which took 32.3379106 910 033 44 seconds).

### 3. Multi-threading Paradigm

#### Code snippet



```
multithreading.py x
multithreading.py > ...
1  from concurrent.futures import ThreadPoolExecutor
2
3  import requests
4
5  from program_timer import timer
6
7  URL = "https://httpbin.org/uuid"
8
9  def fetch(session, url):
10     with session.get(url) as response:
11         print(response.json()["uuid"])
12
13  @timer(1, 1)
14  def main():
15     with ThreadPoolExecutor(max_workers=50) as executor:
16         with requests.Session() as session:
17             executor.map(fetch, [session] * 100, [URL] * 100)
18     executor.shutdown(wait=True)
```

The above code uses multi-threading to execute multiple network requests concurrently, aiming to speed up the process. The `ThreadPoolExecutor` class manages a pool of threads for executing tasks asynchronously. The `map` method from `ThreadPoolExecutor` is used to apply the `fetch` function to multiple arguments (session and URL) concurrently.

#### Output:

Using Multi-threading the time taken to fetch data is 8.282894207000936 seconds, demonstrating significant improvement over the synchronous and multiprocessing approaches.

### 4. Asynchronous Paradigm

The below code shows asynchronous programming using `asyncio` and `aiohttp` to handle multiple network requests efficiently. This approach allows other tasks to run while waiting for network responses, optimizing performance for I/O bound tasks. In the program `async def` declares functions as asynchronous while `await` pauses execution within asynchronous functions for awaited operations.

### Code Snippet:

```
async.py 1 X
async.py > ...
1  import asyncio
2  import aiohttp
3
4
5  from program_timer import timer
6
7  URL = "https://httpbin.org/uuid"
8
9  async def fetch(session, url):
10     async with session.get(url) as response:
11         json_response = await response.json()
12         print(json_response["uuid"])
13
14  async def main():
15     async with aiohttp.ClientSession() as session:
16         tasks = [fetch(session, URL) for _ in range(100)]
17         await asyncio.gather(*tasks)
18
19  @timer(1, 1)
20  def func():
21     asyncio.run(main())
22
23
```

### Output:

The code using asynchronous paradigm executes the program to fetch a data in 2.6922175629988487 seconds, outperforming multithreading, multiprocessing, and synchronous approaches.

### Comparison table

Paradigm	Synchronous	Multiprocessing	Multithreading	Asynchronous
Execution Time (Approx.)sec	32.34	19.39	8.28	2.69

The table shows how long each paradigm takes to fetch the data. It can be seen that "asynchronous" and "multi-threading" are much faster than the other two methods for this kind of task.

### Advantages and Disadvantages:

Paradigm	Advantages	Disadvantages	Best for
<b>Synchronous</b>	Easy to understand, code is straightforward	Slow for I/O bound tasks (waits for each operation to complete), can lead to unresponsive applications	Simple tasks without complex I/O or multi-core processing needs
<b>Multiprocessing</b>	Efficient for CPU-bound tasks, utilizes multiple CPU cores for parallel processing	Overhead of process creation and management can outweigh benefits for I/O bound tasks, complex to implement for tasks with frequent I/O context switching between processes	Highly CPU-bound tasks when the workload requires significant processing power across multiple cores
<b>Multithreading</b>	Utilizes multiple CPU cores for parallel execution, improves performance for tasks with some CPU processing mixed with I/O	Overhead of thread creation and management, potential race conditions if shared data isn't carefully handled	I/O bound tasks with some CPU processing, especially on systems with multiple cores
<b>Asynchronous</b>	Excellent for I/O bound tasks because it doesn't block the main thread, efficiently utilizes a single CPU core by managing multiple tasks concurrently	Requires a different coding style compared to synchronous programming, can be more complex for tasks with dependencies between operations	I/O bound tasks, especially when efficient utilization of a single CPU core is crucial and responsiveness matters