SERIAL AND PARALLEL IMPLEMENTATION OF WEB CRAWLERS

A PROJECT REPORT

Submitted by

Nakul Jadeja (19BCE0660)

Chirayu Rathi (19BCE2206)

Manan Jain(19BCE2183)

Course Code: CSE4001

Course Title: Parallel and Distributed Computing

Under the guidance of

Narayanamoorthy M

Associate Professor, SCOPE,

VIT, Vellore.

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

April, 2022

ABSTRACT

In today's day and age, the use of internet is growing rapidly. The World Wide Web provides a vast source of diverse information. Nowadays, people use search engines to explore large volumes of data easily and extract valuable information from the web. However due to the large size of the Web, searching through all the Web Servers and pages is not very feasible. Everyday a large number of web pages are added and the nature of information gets changed. Due to the extremely large number of pages present on the Web, the search engine depends upon crawlers for the collection of required pages. Web crawling is a very lengthy and time consuming process. It has been observed that the default Python scrapers and crawlers are relatively slow due to the issues that Python has with concurrency due to GIL (Global Interpreter Lock). To ease this task, we would try to build a concurrent crawler as parallel computing would improve the performance and decrease the time of execution by greater leaps than a non concurrent crawler. For this project, we will be trying to develop a concurrent crawler using Python and comparing its performance with a single threaded crawler. The first version is going to lack any concurrency and simply request each of the websites one after the other. The second version makes use of concurrent futures' thread pool executor allowing to send concurrent requests by making use of threads. Finally we will be enhancing the system using asyncio and aiohttp, allowing the user to make concurrent requests by means of an event loop. Then, we will be checking if changing the language for implementation has any kind of impact on the speed of retrieval, internal processing as well as memory requirements for performing the crawl. Essentially, this project revolves around developing a concurrent crawler using Python and performing comparative analysis based on the time they take to crawl a certain number of URLs.

1. Introduction

1.1. Web Crawlers

A non-concurrent or standard web crawler (also known as a web spider or web robot) is a program or automated script which browses the World Wide Web in a methodical, automated manner. This process is called Web crawling or spidering. Crawlers are primarily programmed so that browsing is automated for repetitive behaviour. To browse the internet and create an index, search engines use crawlers most frequently. Other crawlers search for various data types, such as RSS feeds and email addresses. The word crawler originates from the Internet's first search engine: the Web Crawler. "Bot" or "Spider" is also synonyms. Googlebot is the most well-known web crawlers. Many legitimate sites, in particular search engines, use spidering as a means of providing up-to-date data. Web crawlers are mainly used to create a copy of all the visited pages for later processing by a search engine that will index the downloaded pages to provide fast searches. Crawlers can also be used for automating maintenance tasks on a Web site, such as checking links or validating HTML code. Also, crawlers can be used to gather specific types of information from Web pages, such as harvesting email addresses (usually for spam).

1.2. Working of Web Crawlers

A crawler searches for web-based information that it assigns to certain categories, then indexes and catalogues it so that it can be retrieved and analyzed for the crawled information. Until a crawl is started, the operations of these software programs need to be created. In this way, every order is specified in advance. Then, the crawler immediately executes these instructions. With the results of the crawler, an index is generated, which can be accessed via output software. Depends on the basic instructions, the knowledge a crawler can obtain from the Site. Crawling is the process of exploration in which a team of robots (known as crawlers or spiders) is sent out by search engines to find new and modified content. Content can differ, such as a website, an image, a video, a PDF, etc., but the content is found by links regardless of the format.

Example: To find new URLs, Googlebot begins by fetching a few web pages and then follows the links on those web pages. The crawler is able to find new content and add it to their index called Caffeine, a vast database of discovered URLs, by hopping along this link path, to be later retrieved when a searcher finds information that is a good match for the content on that URL.

1.3. Non-Concurrent Web Crawlers

It is a sequential crawler which will crawl all the pages in a linear fashion. It is the most basic crawler.

1.4. Concurrent Web Crawlers

Concurrent crawlers can utilize multi-handling or multi-threading. Each procedure or string works like a consecutive crawler, aside from the share information structures: wilderness and archive. Shared information structures must be synchronized (bolted for concurrent composes). Speedup of factor of 5-10 is simple thusly.

1.5. Asyncio, Coroutine, Aiohttp

The asyncio module gives a structure that spins around the occasion circle. An occasion circle essentially sits tightly to something to occur and afterward follows up on the occasion. It is in charge of taking care of such things as I/O and framework occasions. Asyncio really has a few circle executions accessible to it. The module will default to the one well on the way to be the most productive for the working framework it is running under; be that as it may, you can unequivocally pick the occasion circle on the off chance that you so want. An occasion circle essentially says "when an occasion occurs, respond with capacity B". Think about a server as it sits tight for somebody to go along and request an asset, for example, a website page. On the off chance that the site isn't exceptionally prominent, the server will be inert for quite a while. At the point when a client stacks the website page, the server will check for and call at least one

occasion handlers. When those occasion handlers are done, they have to give control back to the occasion circle. To do this in Python, asyncio utilizes coroutines. A coroutine is a unique capacity that can surrender control to its guest without losing its state. A coroutine is a purchaser and an expansion of a generator. One of their huge advantages over strings is that they don't utilize especially memory to execute. Note that when you call a coroutine work, it doesn't really execute. Rather it will restore a coroutine protest that you can go to the occasion circle to have it executed either promptly or later on. Python 3.5 included some new syntax that enables developers to make offbeat applications and packages less demanding. One such package is aiohttp which is a HTTP customer/server for asyncio. Essentially it enables you to compose nonconcurrent customers and servers. The aiohttp bundle likewise underpins Server WebSockets and Client WebSockets.

1.6. Application of Crawlers

Creating an index is the classic aim of a crawler. Thus, the foundation for the work of search engines is crawlers. They scour the Web for content first and then make users' results available. Based crawlers, for example, when indexing, concentrate on existing, content-relevant websites. There are also web crawlers used for other purposes:

• Price comparison sites aim for web-based information on individual goods so that prices or data can be reliably compared.

• A crawler can collect publicly accessible company e-mail or postal addresses in the field of data mining.

• In order to collect data for page views, or incoming or outbound links, web analysis tools use crawlers or spiders.

• Crawlers, for example, news pages, serve to provide informationhubs with data.

The operating system reduces the simultaneous programmes to a specific sequence of activities (but not pre-determined, it is an on-the-fly operation and relies on the process

contingency). On single-core processors, a multitasking system works as well. Parallel execution ensures that the machine allocates the instructions to various core processors. In both singletask and multitasking systems, this can occur. The use of the word parallel and concurrent is also common in other domains, such as programming, but it has created some confusion with the sequential and concurrent meanings that are somehow related.

Web crawling is a critical technique for gathering information and staying up with the latest with the quickly growing Internet. A web crawler is a program, which consequently navigates the web by downloading reports and following connections from page to page. It is a mechanism for the web crawlers and other data searchers to accumulate information for ordering and to empower them to stay up with the latest. All web indexes inside utilize web crawlers to keep the duplicates of information a new. Web crawler is partitioned into various modules. Among those modules crawler module is the module on which web index depends the most on the grounds that it gives the most ideal outcomes to the web search tool. Crawlers are little projects that 'peruse' the web for the internet searcher's sake, correspondingly to how a human client would pursue connects to achieve diverse pages. The motive of each data scientist and analyst is to draw inferences from the data gathered and thereby keep the world updated with the latest of happenings. Data is being generated at a high rate, with millions and millions of websites and zettabytes of data presently on the web, it is difficult to process the data. The need for an efficient algorithm to process gather and process the data is of utmost urgency.

2. Literature Survey

2.1. Survey of Related Work

S. No.	Title	Abstract
1	A Comparison of Open Source Web Crawlers for E-Commerce Websites	Web crawlers are valuable tools for extracting data from the Internet, such as text or images. It is interactive software that is able to collect data from websites. Websites for e-commerce are important areas for crawler applications. The paper focuses on summarising the methods of performance assessment of open source web crawlers, potential patterns in research and related gaps in research. In addition, a suggested method for the evaluation of the open source crawler has been presented.
2	Distributed Web Crawling over DHTs	In this paper, the design and implementation of distributed web crawler are presented. The distributed crawler harnesses the excess bandwidth and computing resources of clients to crawl the web. Nodes participating in the crawl use a Distributed Hash Table (DHT) to coordinate and distribute work. The implementation of the distributed crawler is using PIER, a relational query processor that runs over the Bamboo DHT and compares different crawl strategies on PlanetLab querying live web sources.
3	Web Crawler Scheduler based on Coroutine	The Coroutine crawler can perform crawler tasks more efficiently than the single-threaded and multi-threaded crawlers. At the same time, this profit is becoming greater. It is obvious when the number of tasks increases. In reality, the paper reported both CPU utilisation and memory consumption. Multi-threaded crawlers had the highest use in terms of CPU use, followed by the coroutine crawler, and the lowest was the single threaded crawler.
4	Design and implementation of a high-performance distributed Web crawler	This paper describes the design and implementation of a distributed Web crawler that runs on a network of workstations. The crawler scales to several hundred pages per

		second are resilient against system crashes and other events and can be adapted to various crawling applications.
5	PyBot: An Algorithm for Web Crawling	PyBot is a Web Crawler developed in Python to crawl the Web using Breadth-First Search (BFS). The success of the World Wide Web (WWW), which itself built on the open internet, has changed the way how humans share and exchange information and ideas. With the explosive growth of the dynamic Web, users have to spend much time just to retrieve a small portion of the information from the Web. The birth of search engines has made human lives easier by simply taking them to the resources they want. A web crawler is a program used by search engines to retrieve information from the World Wide Web in an automated manner.

2.2. Problem Definition

Developing a concurrent crawler using Python and Go and performing comparative analysis based on the time they take to crawl a certain number of URLs.

3. Overview of the Work

3.1. Objectives of the Project

For this project, we are trying to instil concepts of concurrency into web crawlers and to develop a concurrent versions of a crawler using Python and comparing its performance with the single threaded crawler. Hence the main agenda is to write three different versions of the same script. The first version i.e. the non-concurrent crawler is going to lack any concurrency and simply request each of the websites one after the other. The second version i.e. concurrent futures makes use of concurrent futures' thread pool executor allowing us to send concurrent requests by making use of threads. Then, we are going to take a look at a version of the script using asyncioand aiohttp, allowing us to make concurrent requests by means of an event loop.

3.2. Software Requirements

- **Operation System:** Linux- Ubuntu 16.04 to 17.10, or Windows 7 to 10
- **Chrome Version:** 79.0 or above or Safari Version 11.1.2 or upgraded
- **IDLE:** Python v3.10 (preferable)

3.3. Hardware Requirements

- **Processor:** x86 64-bit CPU (Intel / AMD architecture)
- **Ram: 2 GB or more**

3.4. Methodology

3.4.1 Sequential crawler

The first step in the project was to write the code for a sequential crawler which will crawl all the pages in a linear fashion. The crawler would fetch the pages starting from a seed page and rank those pages based on the cosine similarity of the web page with the query. The time for computing for crawling and ranking the pages is computed using time.clock() function in the time module. The start and the end time are noted at the beginning and the end of the code to note down the required time for the computation.

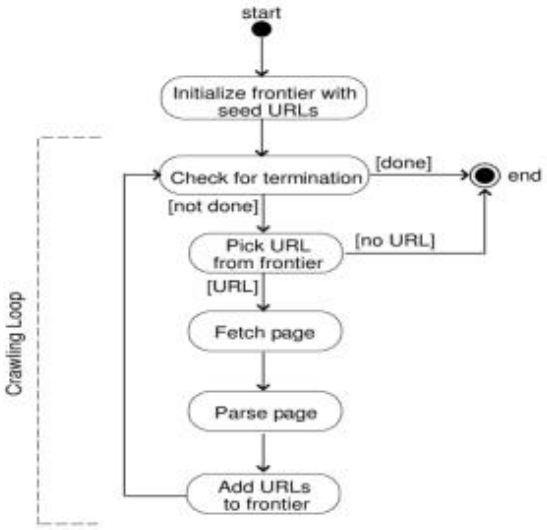


Figure 3.5 Flow chart of sequential crawler

Similarly, a code for the concurrent crawler is written to note down the changes in the required time by parallelizing time-consuming events, especially blocking I/O events such as requesting the HTML documents using http requests. Such events take time and during this time, the processor is not doing any fruitful work. However, executing this part in parallel helps in reducing the overall latency and increases throughput because, by the time one page is fetched, the crawler also makes a concurrent request to the next page, and while the first page is being compared with the query using cosine similarity measures, other pages have lined up in the queue. Hence, the total latency for the operation is:

In the case of sequential crawler, we have the case:

Therefore, in this case, we are increasing the FLOPS of the whole process by a factor of K. However, this is a rough estimate, and does not include external acting factors like slow or fluctuating internet connection. This cannot be factored in, and the above observation is the best-case scenario. To test this conjecture, we will try and implement the code using Python's Concurrent.futures. This module is present in Python's standard library and it will make concurrent requests using ThreadPoolExecutor, which basically works by creating separate threads for each request, and the crawler performs the required computation whenever it is done with one thread. This will supposedly increase the performance of the crawler. The time will be calculated in the same way as the sequential crawler. Next, we intend to use asyncio andaio http libraries to make concurrent requests by means of an event loop which will offer

us increased power and even better performance. According to the literature available online, it is said that asyncio and aiohttp are much more robust and useful, especially because of Python's slow runtime. To test this conjecture, we also plan to write a code using asyncio and aiohttp.

3.3.2 EVENT LOOP AND ASYNCIO

The event loop is central to the execution of the asynchronous functions. The event loop starts with registering or calling or canceling the coroutines. A coroutine is an asynchronous function in Python. It involves using the keyword asyncbefore its definition. The event loop may also involve building secure transport for communication between client and server or in other cases building transports to just communicate with another program. The main function of the event loop is to delegate the asynchronous functions to a pool of threads. Asyncio uses the event loop to asynchronously execute a coroutine. Using Python's standard asynciolibrary, and a package called "aiohttp", fetching a URL in a coroutine is very direct.

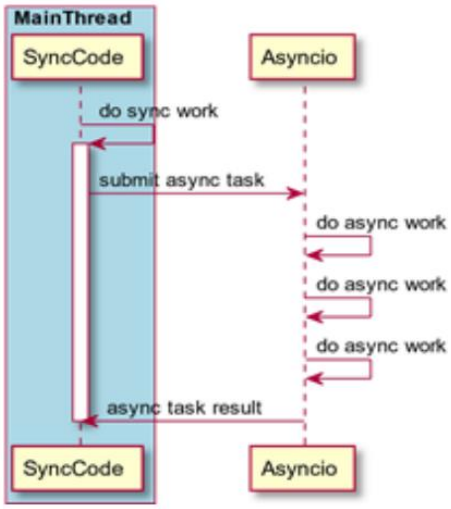


Figure 3.6 Event loop

The crawl method comprises all that our main coroutine must do. It is the worker coroutines that get URLs from the queue, fetch them, and parse them for new links.

Each worker runs the workcoroutine independently:

```
@asyncio.coroutine

def work(self):

 while True:

 url, max_redirect = yield from self.q.get()
```

```
# Download page and add new links to self.q.

yield from self.fetch(url, max_redirect)

self.q.task_done()
```

This will keep executing asynchr() till all the tasks are executed. In the scenario of the crawler, the number of tasks created are equal to the maximum number of threads provided by the function. These tasks are added to an asyncio queue, and then the loop will execute these tasks by crawling the pages until all of them are complete. If any of the tasks is waiting for the response from the web servers, then the loop can schedule other functions to the remaining threads. Following that the loop is closed. Aiohttp is used to create client sessions, which can then be used to get the HTTP response from web servers. For this code, get_bodyI(url), get_results(url) and handle_tasks(task_id, work_queue) are asynchronous functions.

3.3.3 Concurrent.futures

ThreadPoolExecutor is another way to execute functions on a pool of threads available. This is relatively simple and the number of threads can be set simply by creating an instance of ThreadPoolExecutor and setting the max_workers equal to the minimum of a number of URLs and the maximum number of threads. The instance of ThreadPoolExecutor will then submit tasks to the thread pool and these will be executed. These tasks will be functions.

```
Executor = ThreadPoolExecutor(max_workers =3)

Executor.submit(sample())
```

4. System Design

4.1. Algorithm

For a sequential crawler which will crawl all the pages in a linear fashion:

- The crawler would fetch the pages starting from a seed page and rank those pages based on the cosine similarity of the web page with the query.
- The time for computing for crawling and ranking the pages is computed using time.clock() function in the time module.
- The start and the end time are noted at the beginning and the end of the code to note down the required time for the computation.

For the concurrent crawler to note down the changes in the required time by parallelizing time-consuming events:

- Blocking I/O events such as requesting the HTML documents using http requests.
- The events take time and during this time, the processor is not doing any fruitful work.
- Executing this part in parallel reduces the overall latency and increases throughput
- By the time one page is fetched, the crawler also makes a concurrent request to the next page.
- While the first page is being compared with the query using cosine similarity measures, other pages have lined up in the queue.

We will try and implement the code using Python's concurrent.futures.

- The module is present in Python's standard library and it will make concurrent requests using ThreadPoolExecutor, which basically works by creating separate threads for each request
- The crawler performs the required computation whenever it is done with one thread.
- This will supposedly increase the performance of the crawler.
- The time will be calculated in the same way as the sequential crawler.

4.2. Architecture Diagrams

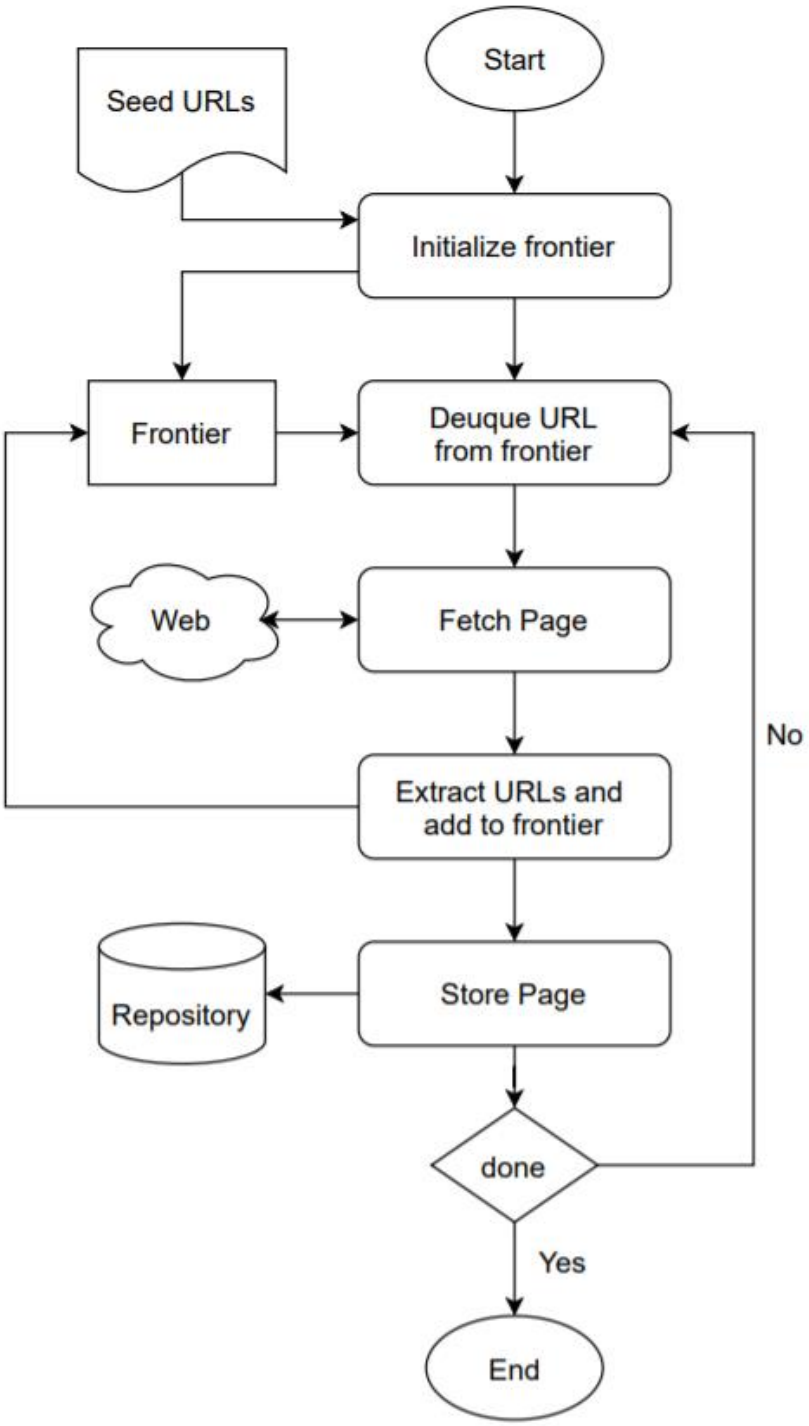


Fig.: Sequential Architecture

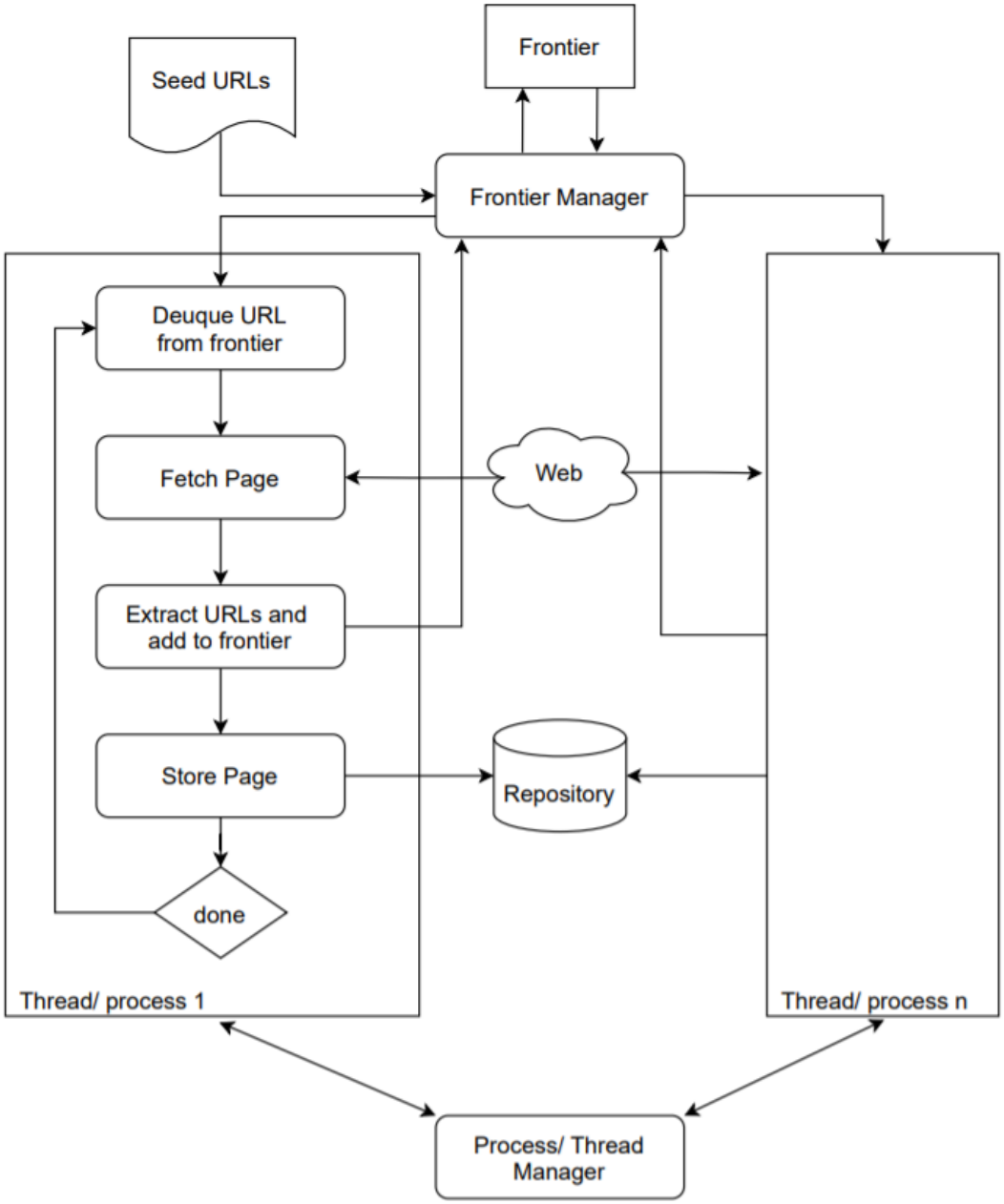


Fig.: Concurrent Architecture

5. Implementation

5.1. Description of Modules/Programs

5.2. Source Code

Non-concurrent crawler

```
import requests
import time
from bs4 import BeautifulSoup

class GILBlocking(object):

    def __init__(self, url_list):

        self.urls = url_list
        self.results = {}

    def __make_request(self, url):
        try:
            r = requests.get(url=url, timeout=20)
            r.raise_for_status()
        except requests.exceptions.Timeout:
            r = requests.get(url=url, timeout=60)
        except requests.exceptions.ConnectionError:
            r = requests.get(url=url, timeout=60)
        except requests.exceptions.RequestException as e:
            raise e
        return r.url, r.text

    def __parse_results(self, url, html):

        try:
            soup = BeautifulSoup(html, 'html.parser')
```

```
            title = soup.find('title').get_text()
        except Exception as e:
            raise e

        if title:
            self.results[url] = title

    def wrapper(self, url):
        url, html = self.__make_request(url)
        self.__parse_results(url, html)

    def get_results(self):
        for url in self.urls:
            try:
                self.wrapper(url)
            except:
                pass

if __name__ == '__main__':
    q=time.time()
    scraper = GILBlocking(["https://timesofindia.indiatimes.com/defaultinterstitial.cms",
"https://www.theguardian.com/international",
"https://www.apple.com/in/?afid=p238%7CsdUuvv563-
dc_mtid_187079nc38483_pcrid_474706300243_pgrid_109516736379_&cid=aos-IN-kwgo-
brand--slid---product-",
"https://www.samsung.com/in/",
"https://www.amazon.jobs/en-gb/"])
    scraper.get_results()
    q1=time.time()
    for key,val in scraper.results.items():
        print(key+": "+val)
    print("The time taken to crawl the given web pages is: "+str(q1-q))
```

Concurrent crawler

```
from concurrent.futures import ThreadPoolExecutor
import time
import requests
from bs4 import BeautifulSoup

class ConcurrentListCrawler(object):

    def __init__(self,url_list, threads):
        self.urls = url_list
        self.results={}
        self.max_threads = threads
        print("Number of threads "+str(self.max_threads))

    def __make_request(self, url):
        try:
            r=requests.get(url=url, timeout=20)
            r.raise_for_status()
        except requests.exceptions. Timeout:
            r = requests.get(url=url, timeout=60)
        except requests.exceptions.ConnectionError:

            r = requests.get(url-query, timeout=60)

        except requests.exceptions.RequestException as e:
            raise e

        return r.url, r.text

    def __parse_results(self, url, html):
        try:
            soup = BeautifulSoup(html, 'html.parser')
            title = soup.find('title').get_text()
```

```
        except Exception as e:

            raise e

        if title:
            self.results[url]=title

    def wrapper(self, url):

        url, html=self.__make_request(url)
        self.__parse_results(url, html)

    def run_script(self):
        with ThreadPoolExecutor(max_workers=min(len(self.urls),self.max_threads)) as
Executor:
            jobs=[Executor.submit(self.wrapper, u) for u in self.urls]

if __name__ =='__main__':
    q=time.time ()
    example =
ConcurrentListCrawler(["https://timesofindia.indiatimes.com/defaultinterstitial.cms",
"https://www.theguardian.com/international",
"https://www.apple.com/in/?afid=p238%7CsdUuvv563-
dc_mtid_187079nc38483_pcrid_474706300243_pgrid_109516736379_&cid=aos-IN-kwgo-
brand--slid---product-",
"https://www.samsung.com/in/",
"https://www.amazon.jobs/en-gb/"],5)
    example.run_script()
    q1=time.time()

#print(example, results)
    for key, val in example.results.items ():
      print (key+": "+val)
    print("Time taken to crawl the given web pages: "+str(q1-q))
```

Asyncio and Aiohttp

```
import asyncio
import aiohttp
from bs4 import BeautifulSoup
import time
import logging
class AsnycGrab(object):
    # Constructor function for Async Grab.
    # Initializes the URLS of an object of Async Grab to the list of URLS P
    # called Name of the List of URLS: url_list
    # Maximum number of threads are set by the paraneter: max_threads The results of the
crawl aro added to a dictionary: results
    def __init__(self, url_list, max_threads):
        self.urls = url_list
        self.results = {}
        self.max_threads = max_threads
    # -parse_results will use Beautifulsoup which is a scraping module
    # Beautiful Soup will scrape the content of the html document mentioned
    def __parse_results(self, url, html):
        try:
            soup = BeautifulSoup(html, 'html.parser')
            title = soup.find('title').get_text()
        except Exception as e:
            raise e
        if title:
            self.results[url] = title

    # getbody (set.url) is an asynchronous coroutine that will create a cli The session will fetch
the URL with timeout set as 3 asynchronously.
    # get body (self.url) is an asynchronous coroutine that will create a client session. The
session will fetch the URL with timeout set as 3s asynchronously The event loop can go on to
```

execute other threads while the coroutine anais for reading the Return values: URL for response and HTML content for body.

```
    # ...
    async def get_body(self, url):
        async with aiohttp.ClientSession() as session:
            async with session.get(url, timeout=30) as response:
                assert response.status == 200
                html = await response.read()
                return response.url, html
    # ...
```

get reaultsiself, url) is an asynchronous function that awaits for self.get body to return the After returning the HTML und URL, 1 returns 'Completed as a sign for completing the Crawling ..

handle tasks is also an asynchronous function that fetches the first task from the work If the result of self.get results(url) is Completed then we store it in zask status Els the exception is logged.

```
    # ...
    async def get_results(self, url):
        url, html = await self.get_body(url)
        self.__parse_results(url, html)
        return 'Completed'
    async def handle_tasks(self, task_id, work_queue):
        while not work_queue.empty():
            current_url = await work_queue.get()
            try:
                task_status = await self.get_results(current_url)
            except Exception as e:
                logging.exception('Error for {}'.format(current_url),exc_info=True)
```

#event loop(creates a queue and adds the tasks to the queen the loop will keep on running till all the taskS are executed.

```
    def eventloop(self):
        q = asyncio.Queue()
```

```
        [q.put_nowait(url) for url in self.urls]
        loop = asyncio.get_event_loop()
        tasks= [self.handle_tasks(task_id, q, ) for task_id in range(self.max_threads)]
        loop.run_until_complete(asyncio.wait(tasks))
        loop.close()
    #Main Function
if __name__ == '__main__':
    q=time.time()
    async_example = AsnycGrab([
          "https://timesofindia.indiatimes.com/defaultinterstitial.cms",
"https://www.theguardian.com/international",
"https://www.apple.com/in/?afid=p238%7CsdUuvv563-
dc_mtid_187079nc38483_pcrid_474706300243_pgrid_109516736379_&cid=aos-IN-kwgo-
brand--slid---product-",
"https://www.samsung.com/in/",
"https://www.amazon.jobs/en-gb/"],5)
    async_example.eventloop()
    q1=time.time()
    for key,val in async_example.results.items():
        print(str(key)+": "+str(val))
print("Time taken to crawl the pages: "+ str(q1-q))
```

5.3. Test cases

For test cases, we are going to have a different number of URLs ranging from 5 to 100. We are going to run all three crawlers across the URLs and find the total time taken for execution by each crawler.

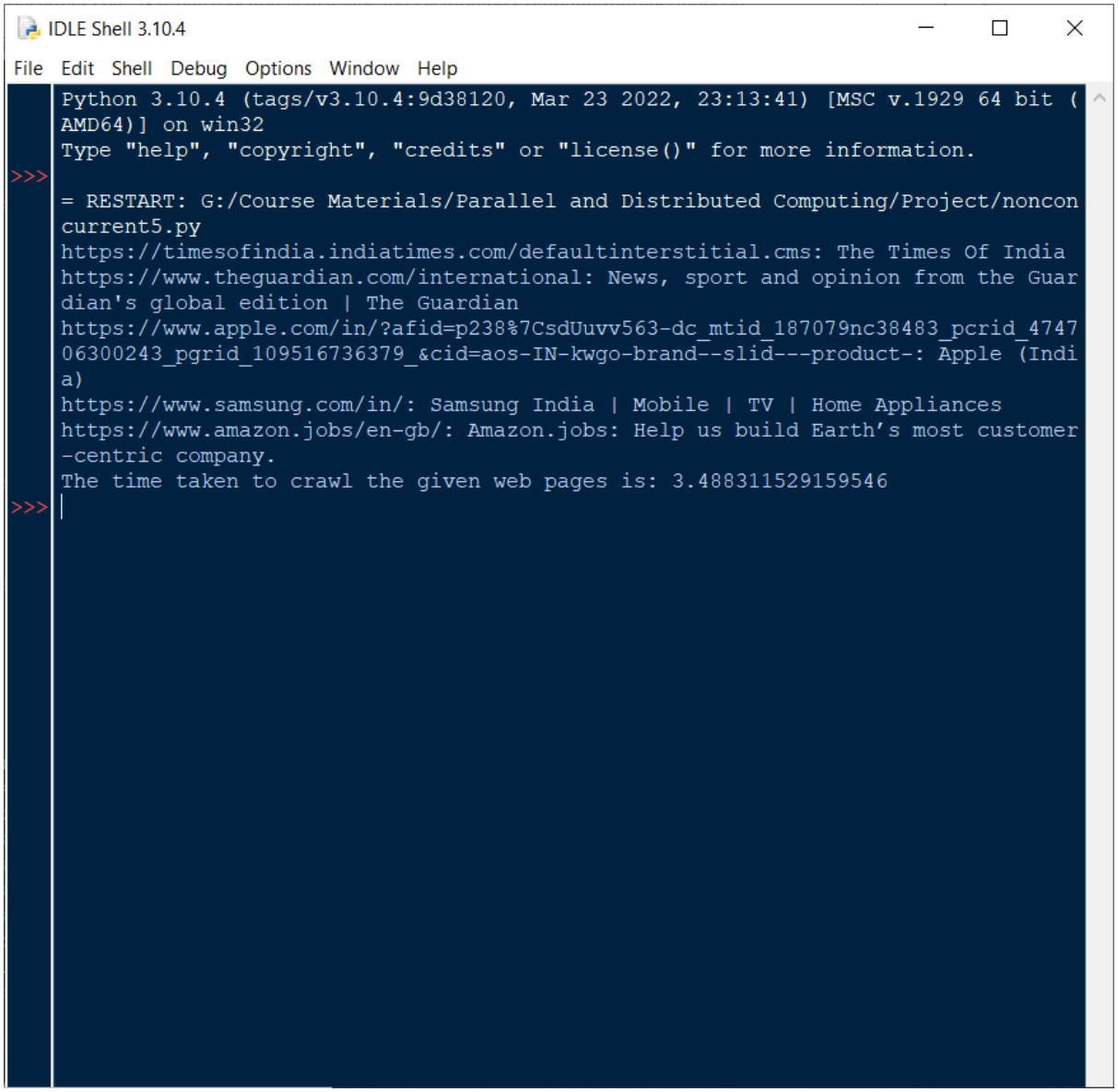
Test case number	Number of URLs
Test case 1	5
Test case 2	25
Test case 3	100

6. Output and Performance Analysis

6.1. Snapshots and Output

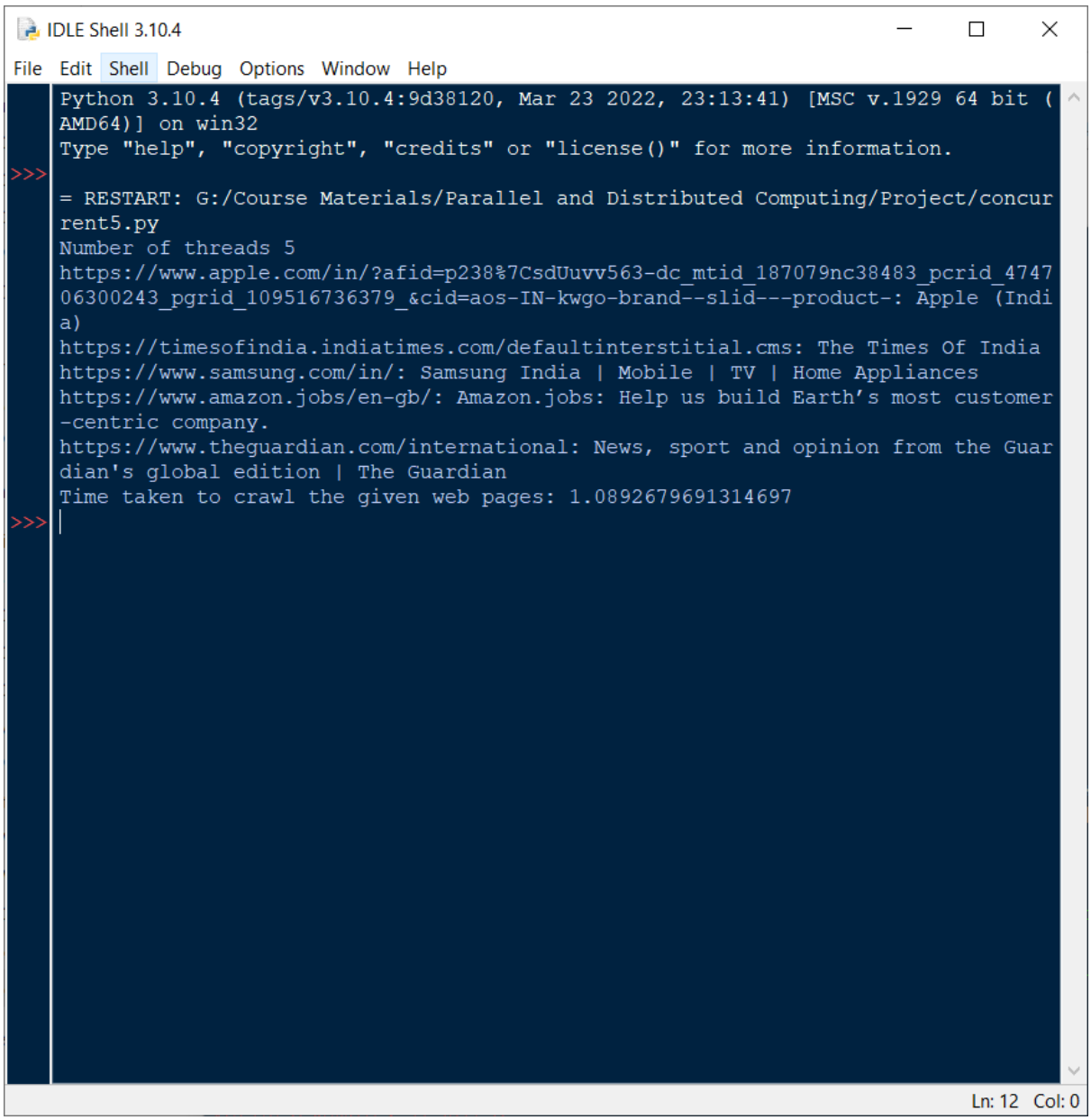
Number of URLs: 5

Non-concurrent



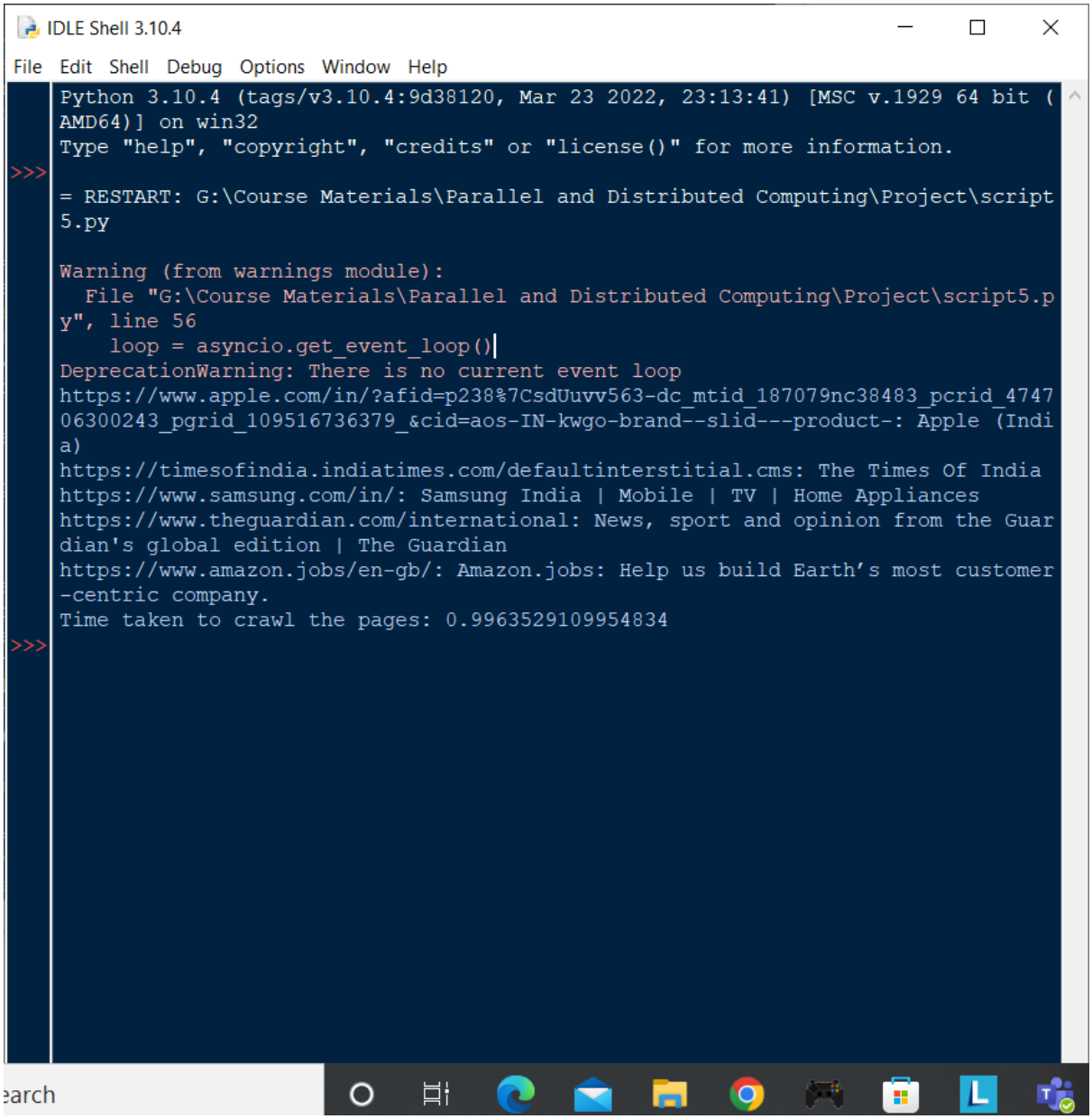
The observed time taken to crawl the given set of pages in a sequential manner is **3.488 sec**.

Concurrent



The observed time taken to crawl the given set of web pages using concurrent futures is 1.089 seconds.

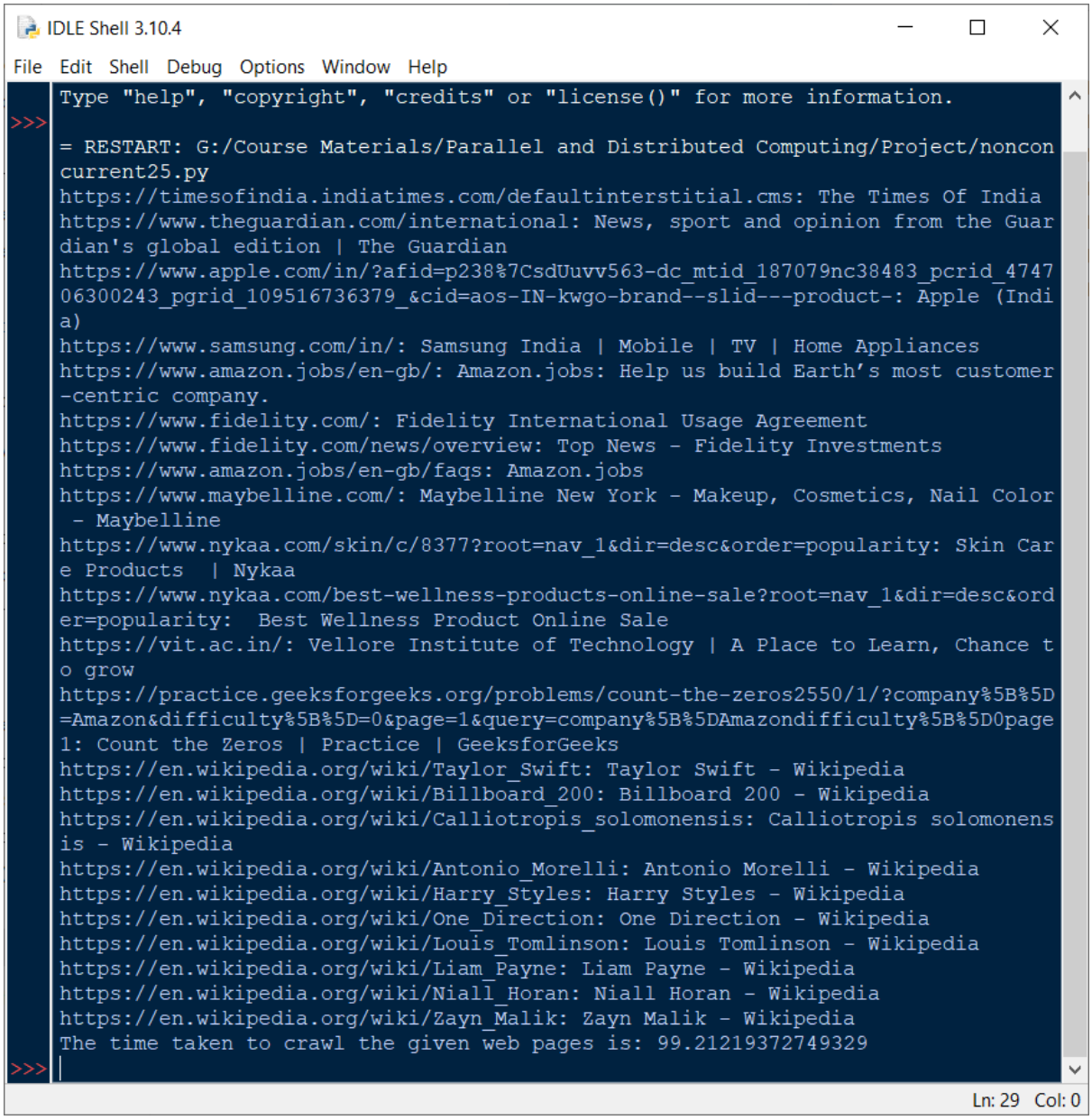
Script



The observed time taken to crawl the given set of web pages using asyncio and aiohttp script is 0.996 seconds which is comparatively lesser than the time taken by a normal concurrent crawler.

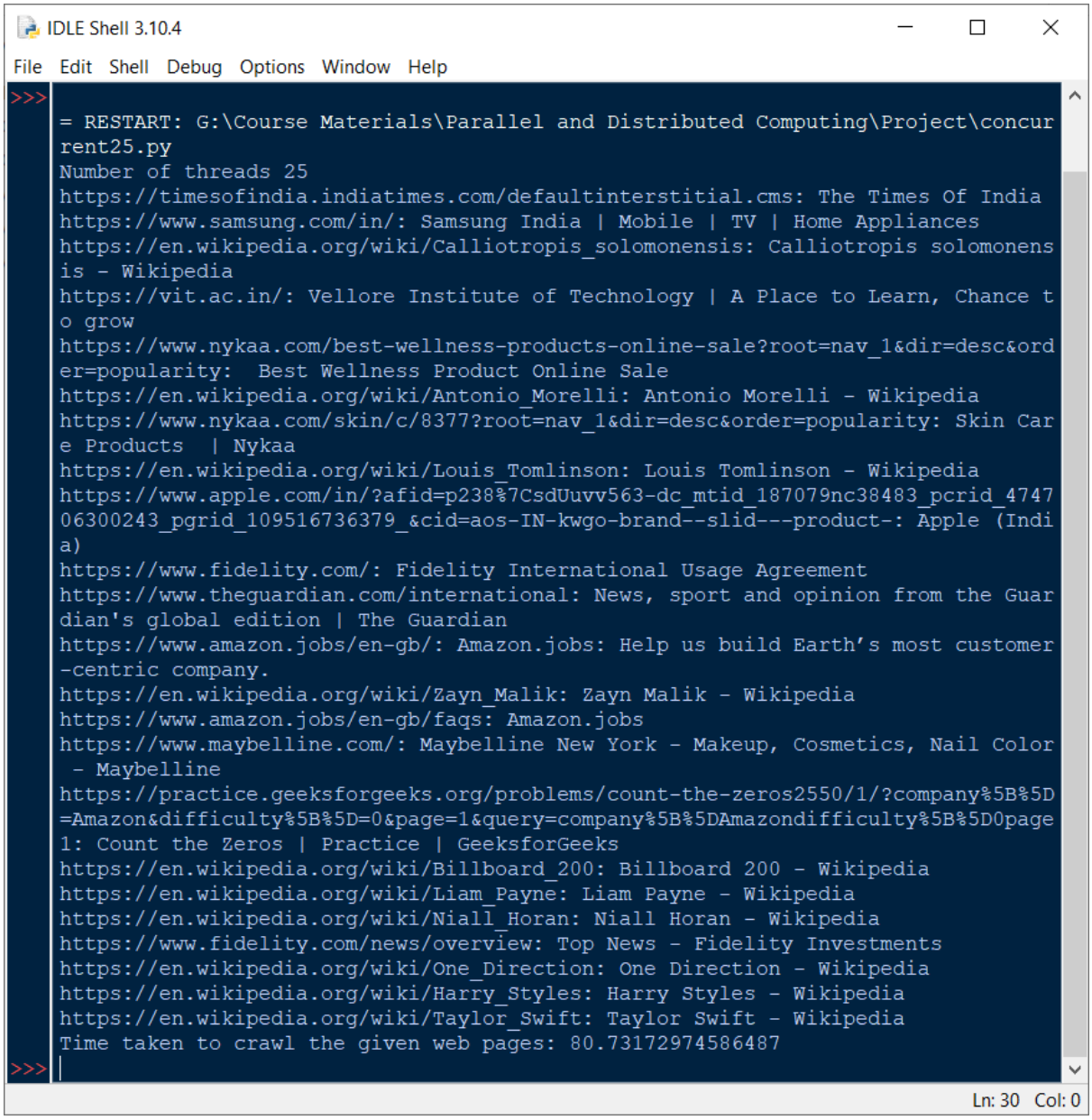
Number of URLs: 25

Non-concurrent



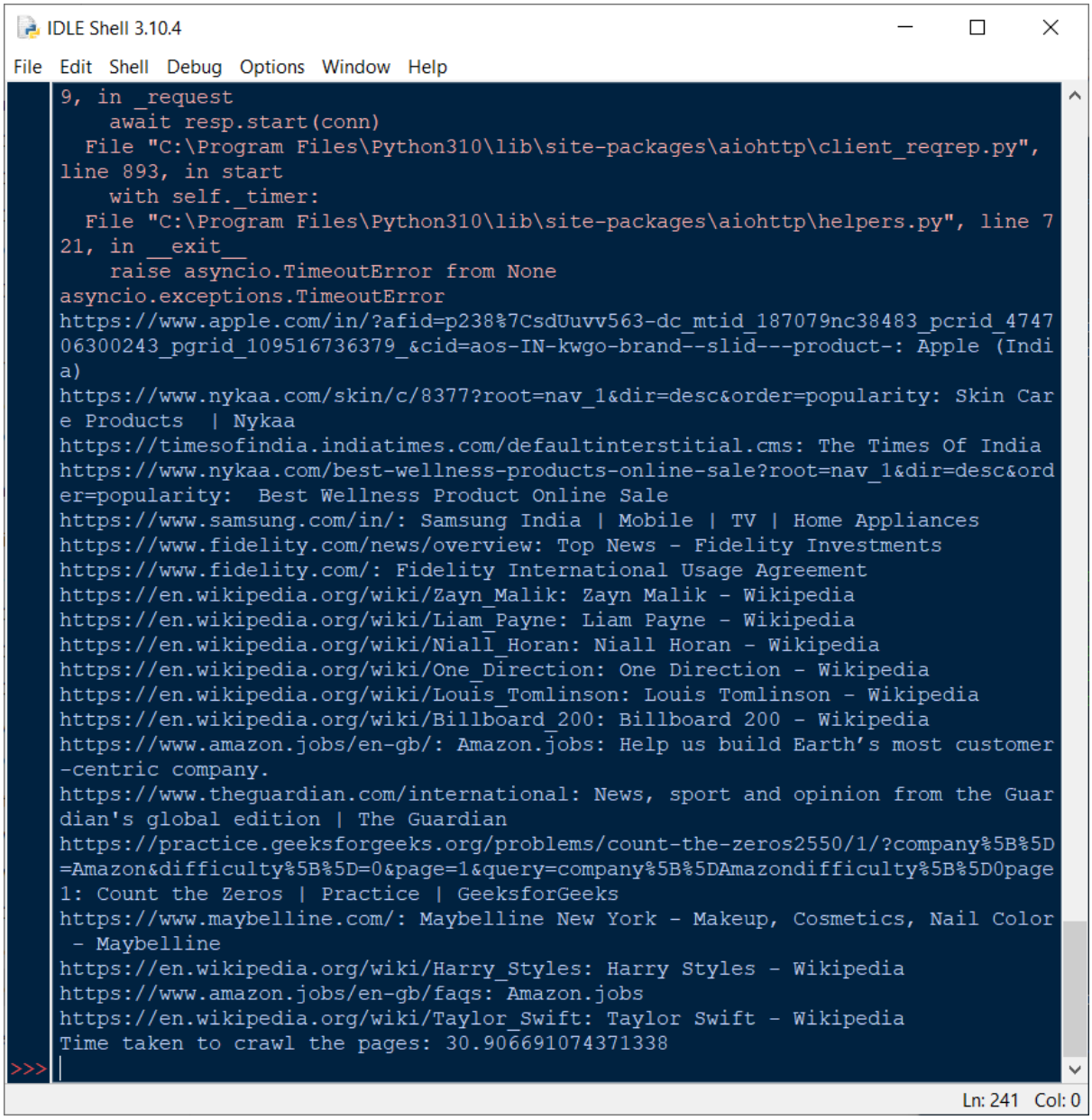
The observed time taken to crawl the given set of pages in a sequential manner is 99.212 seconds

Concurrent



The observed time taken to crawl the given set of web pages using concurrent futures is 80.732 seconds which is much lesser than the time taken in a sequential crawler.

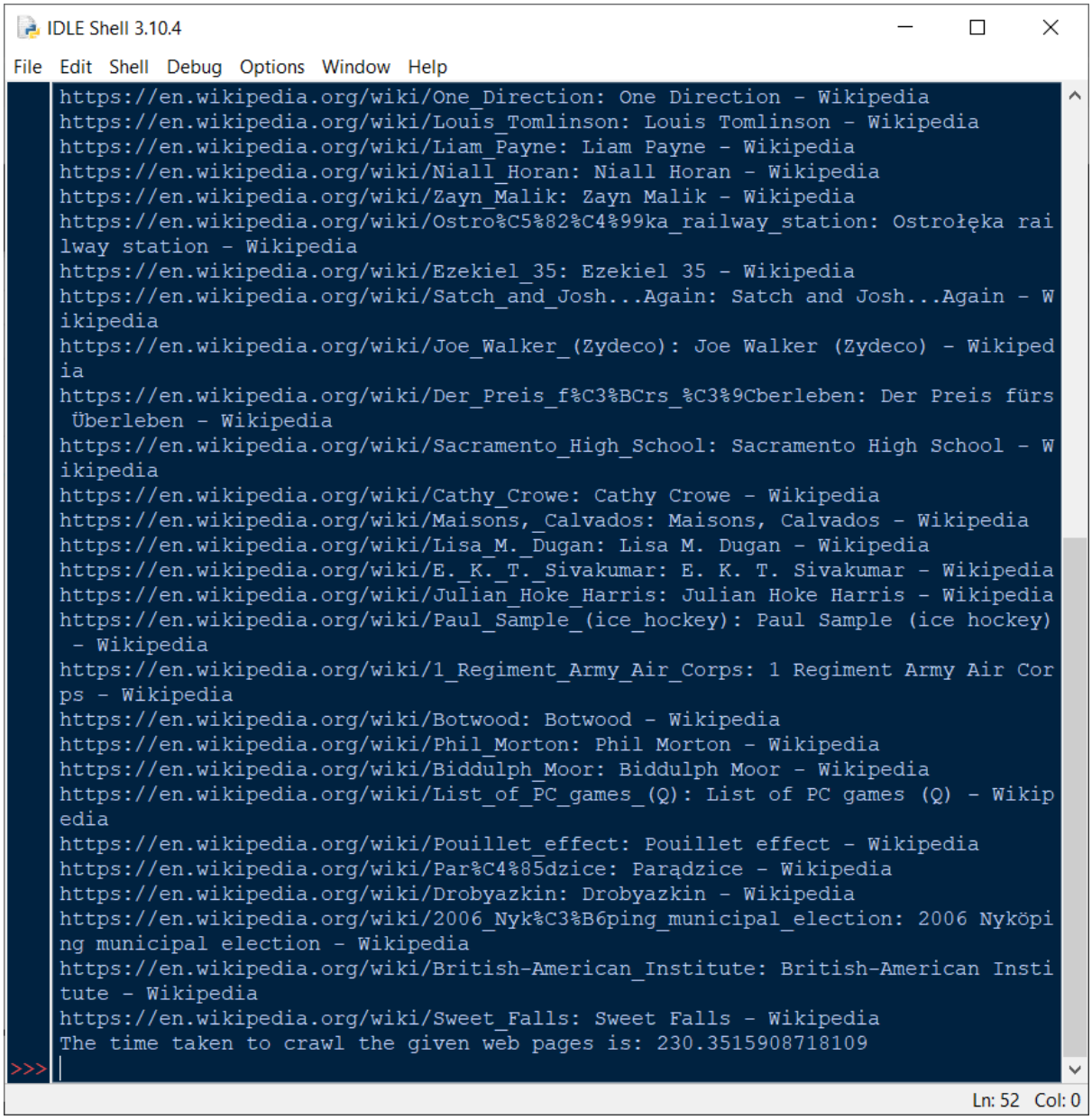
Script



The observed time taken to crawl the given set of web pages using asyncio and aiohttp script is 30.906 seconds which is comparatively lesser than the time taken by a normal concurrent crawler.

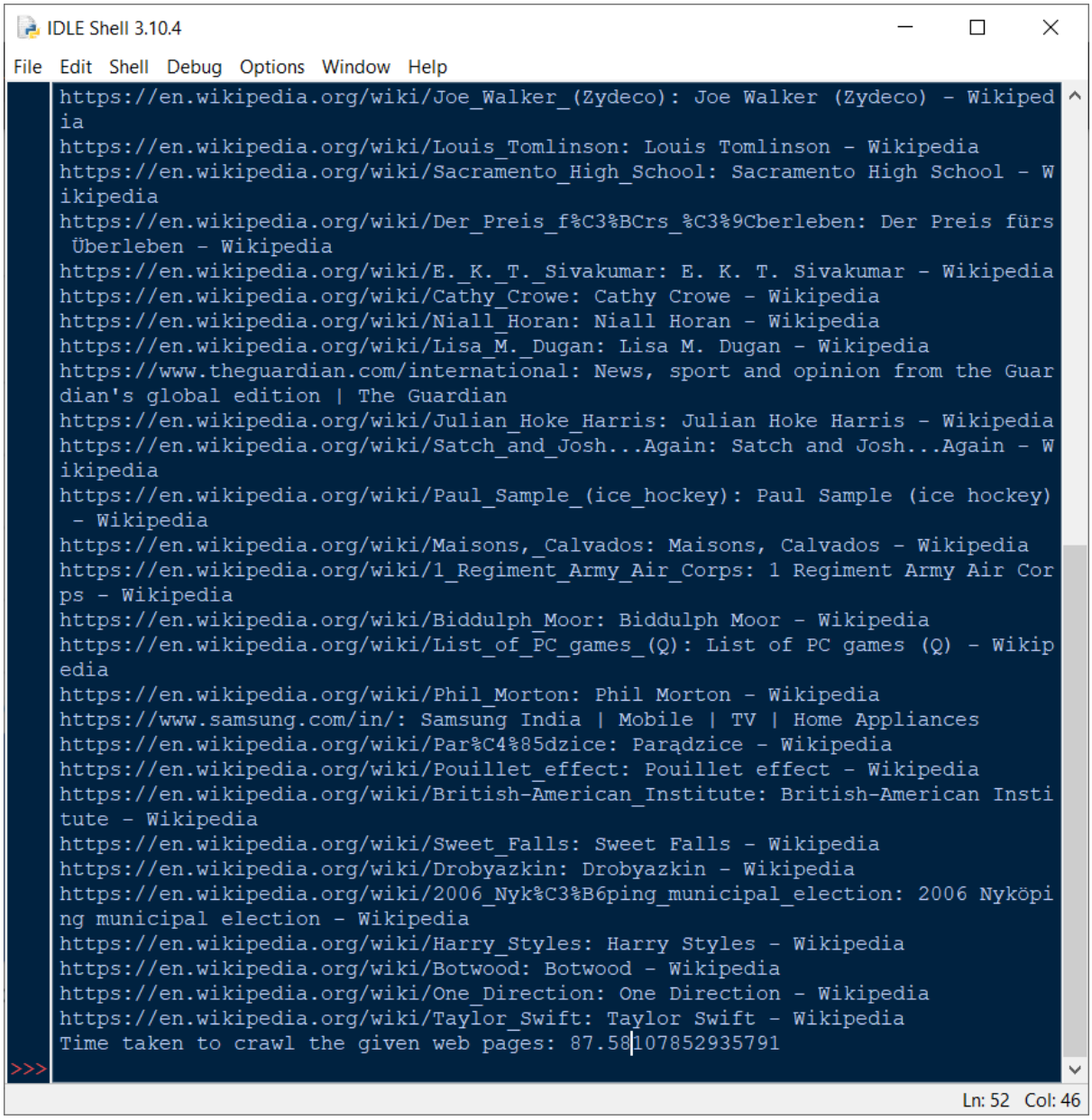
Number of URLs: 100

Non-concurrent



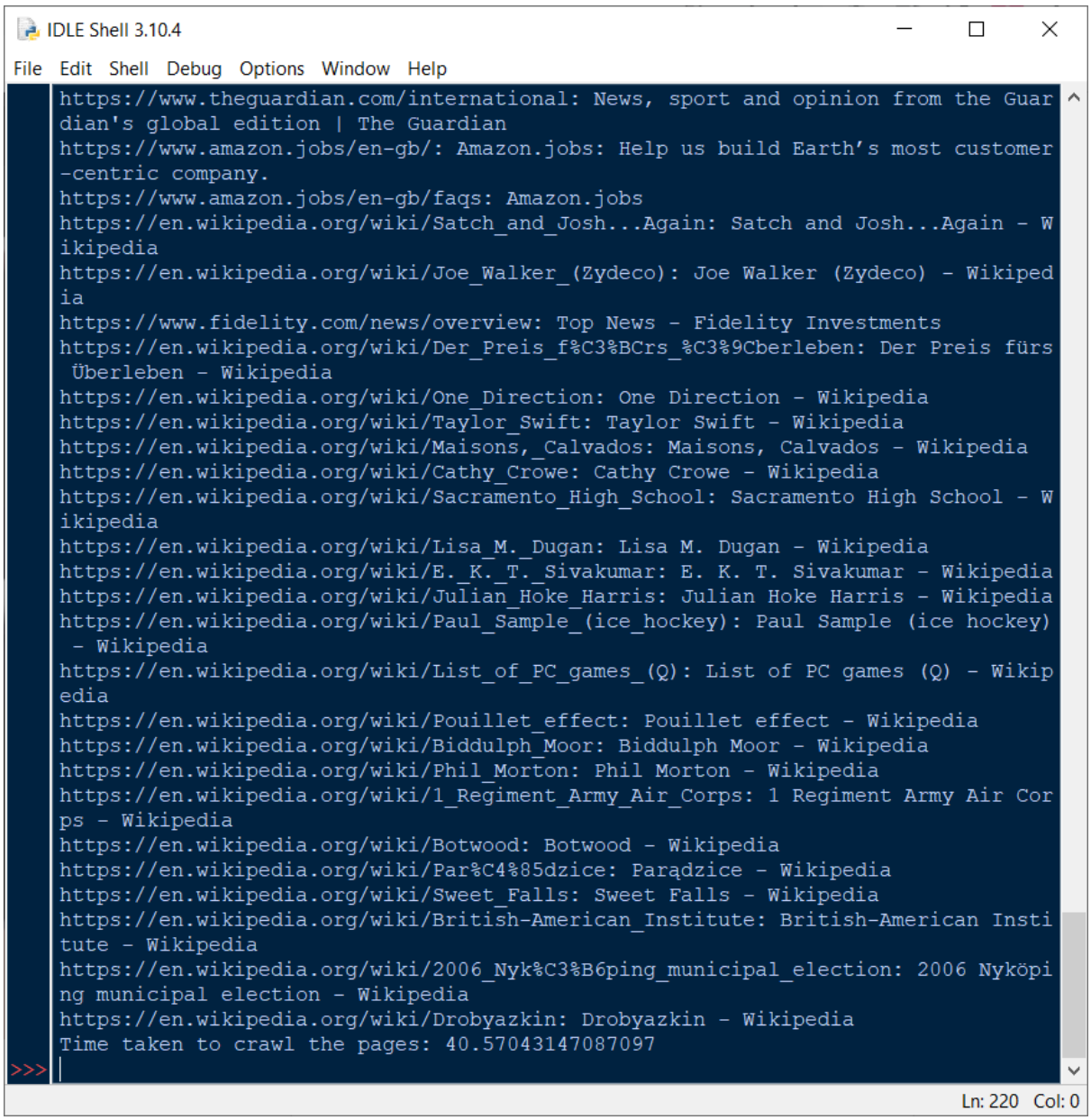
The observed time taken to crawl the given set of pages in a sequential manner is 230.252 seconds

Concurrent



The observed time taken to crawl the given set of web pages using concurrent futures is 87.581 seconds which is much lesser than the time taken in a sequential crawler.

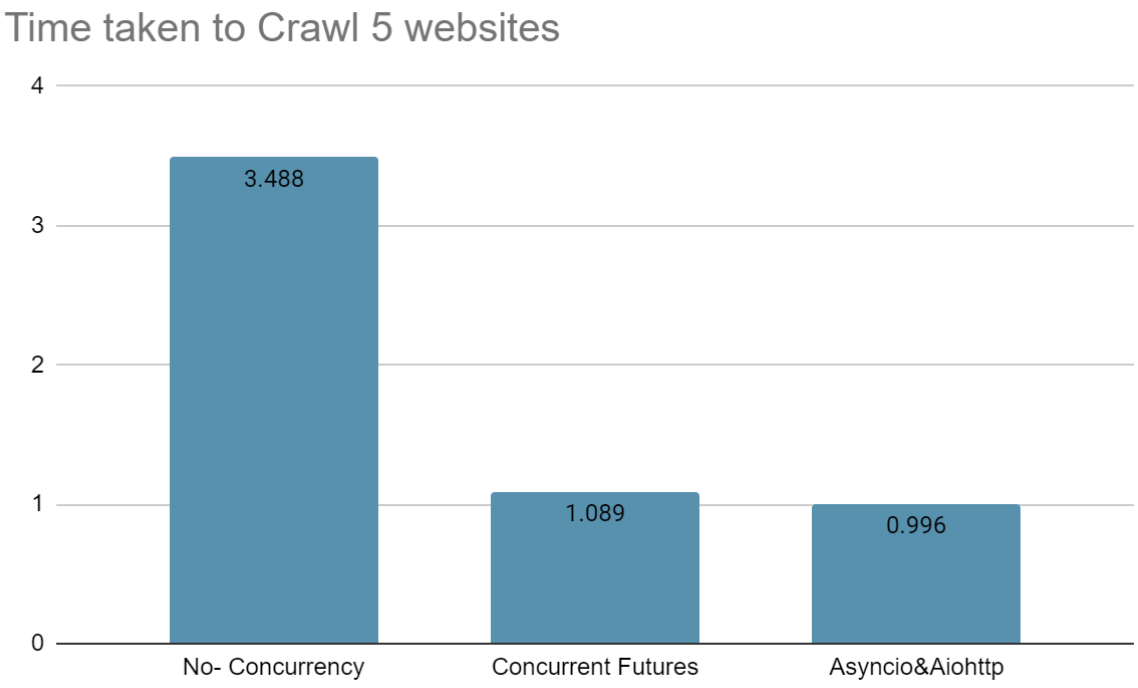
Script

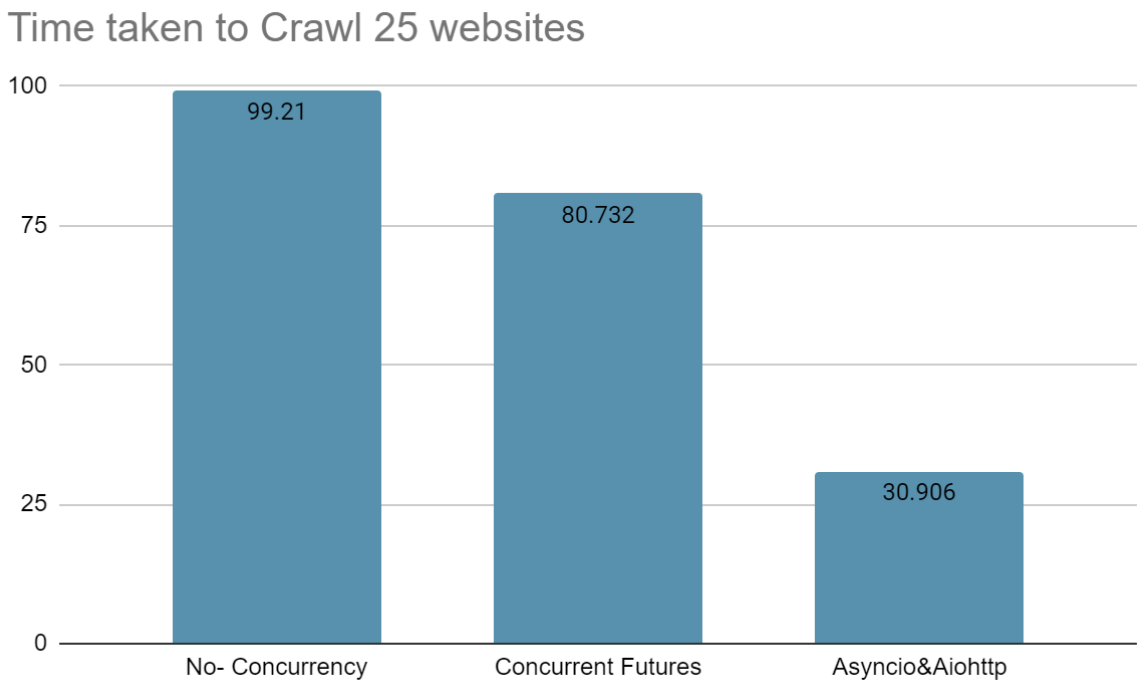


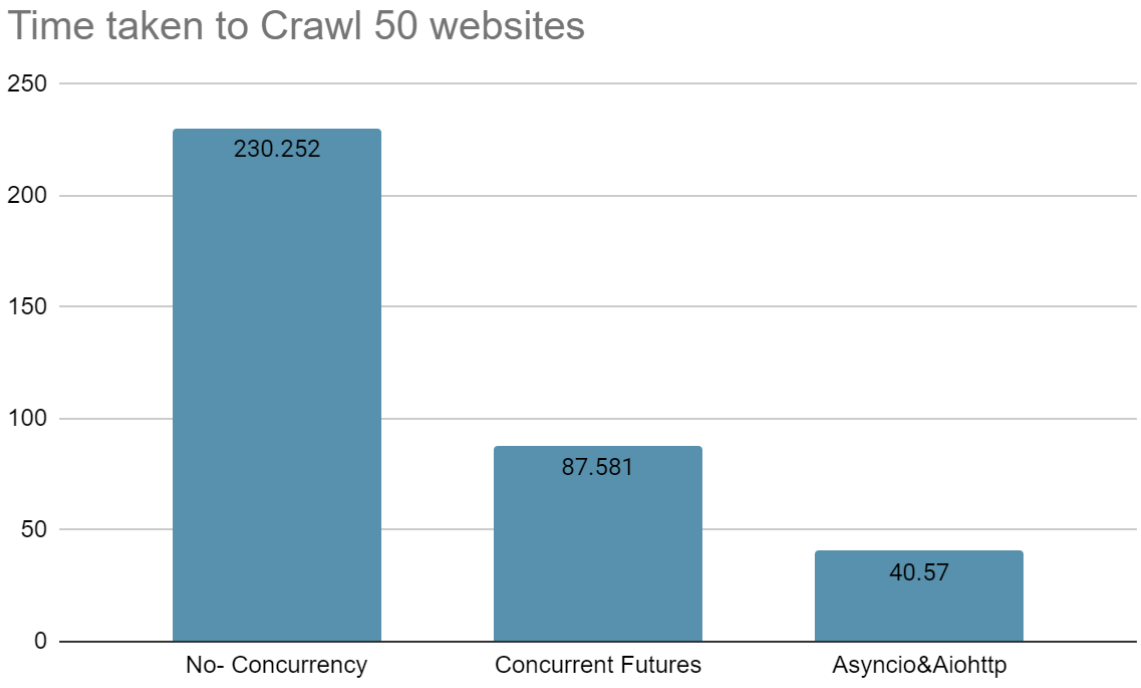
The observed time taken to crawl the given set of web pages using asyncio and aiohttp script is 40.570 seconds which is comparatively lesser than the time taken by a normal concurrent crawler.

6.2. Performance Comparison based on Performance Metrics

Graphical representation of the time taken to crawl the given web pages using different methods:




Time taken to Crawl 25 websites
100
75
50
25
0
99.21
80.732
30.906
No- Concurrency
Concurrent Futures
Asyncio&Aiohttp


Time taken to Crawl 50 websites
250
200
150
100
50
0
230.252
87.581
40.57
No- Concurrency
Concurrent Futures
Asyncio&Aiohttp

Comparing execution time for all methods with no of urls:

Input size(Number of URLs)	No-Concurrency	Concurrent Futures	Asyncio&Aiohttp
5	3.488s	1.089s	0.996s
25	99.21s	80.732s	30.906s
100	230.252s	87.581s	40.570s

7. Conclusion

Web crawlers are programmed to consequently visit pages on the Internet and can be utilized to record them and assemble bits of data from various pages. The main function of a crawler is to screen the volume of data on the Web. Yet, the size of the Web implies that far reaching crawling takes a considerably high amount of power and time. As the amount of data on the Internet keeps on developing, so does the subject of how to process everything and make it helpful. The web crawler made using the Asyncio and Aiohttp libraries is recommended as the best way as it is the fastest. Asyncio is also used as a foundation for multiple Python asynchronous frameworks that provide high-performance network and web-servers, database connection libraries, distributed task queues, etc. It is a perfect fit for IO-bound and high-level structured network code. These conclusions can be drawn based on the results obtained as it clearly shows with the rising number of webpages the amount of time required for other crawlers to crawl through is significantly large.

3.1. Future Directions

From enhancing better marketing activities to creating a good investment decision, web scraping is a boom to the current market and the future it holds. Without data, a brand may not be able to completely fulfill all the needs that their prospects are continuously exploring. Data is one of those assets that hold immense value if the modern lead driven B2B brand wants to achieve success and build healthy relationships with potential prospects. The most important factor required for the web crawlers is speed as no one wants the execution time to be more.

8. References

[1] D. Yang and P. Thiengburanathum, "A Comparison of Open Source Web Crawlers for E-Commerce Websites," 2020 Joint International Conference on Digital Arts, Media and Technology with ECTI Northern Section Conference on Electrical, Electronics, Computer and Telecommunications Engineering (ECTI DAMT & NCON), Pattaya, Thailand, 2020, pp. 200-205, doi: 10.1109/ECTIDAMTNCON48261.2020.9090772.

[2] Z. Wang, "Web Crawler Scheduler Based on Coroutine," 2019 International Conference on Intelligent Computing, Automation and Systems (ICICAS), Chongqing, China, 2019, pp. 540-543, doi: 10.1109/ICICAS48597.2019.00118.

[3] Leng, A.G.K., Kumar, R., Singh, A.K. and Dash, R.K., 2011, December. PyBot: An algorithm for web crawling. In 2011 International Conference on Nanoscience, Technology and Societal Implications (pp. 1-6). IEEE.

[4] Najork, M., 2009. Web Crawler Architecture

[5] [JHH 2016] J. Cho, H. G. Molina, T. Haveliwala, W. Lam, A. Paepcke, S. Raghavan and G. Wesley, Stanford WebBase Components and Applications, ACM Transactions on Internet Technology, 6(2): May 2016

[6] Patwa, A.M., 2006. DESIGN AND (Doctoral dissertation, The University of Colorado at Colorado Springs)

[7] [LKC 2014] B. T. Loo, S. Krishnamurthy, and O. Cooper, Distributed Web Crawling over DHTs. Technical Report UCB-CS-04-1305, UC Berkeley, 2004

[8] [PSM 2014] O. Papapetrou and G. Samaras, Minimizing the Network Distance in Distributed Web Crawling. International Conference on Cooperative Information Systems, 2004, pp. 581-596

[9] V. Shkapenyuk and T. Suel, "Design and implementation of a highperformance distributed Web crawler," Proceedings 18th International Conference on Data Engineering, San Jose, CA, USA, 2002, pp. 357-368, doi: 10.1109/ICDE.2002.994750.

[10] Najork, M.A. and Heydon, C.A., AltaVista Co, 2001. System and method for enforcing politeness while scheduling downloads in a web crawler. U.S. Patent 6,321,265.