Performance analysis of a Non-Concurrent Crawler, a Concurrent Crawler (Python and Go), and a script using asyncio and aiohttp

[1]Kannadasan, [2]P.Manohar Venkat, [3]R.Joshitha

[1]Assistant Professor Senior, [2]Student, [3]Student
SCOPE,
Vellore Institute of Technology, Vellore, India

***Abstract*: Developing a concurrent crawler using Python and Go and comparing its performance with a single-threaded crawler Due to the issues that Python has with concurrency due to GIL (Global Interpreter Lock), the default Python scrapers, and crawlers are relatively slow. The first version will have no concurrency and will just request each website one at a time. The second version makes use of concurrent futures' thread pool executor, allowing me to send concurrent requests by making use of threads. This paper will solve these issues by using asyncio and aiohttp, which will allow concurrent requests to be made via an event loop. Then see if altering the implementation language has any effect on the speed of retrieval, internal processing, or memory needs for conducting the crawl. Google introduced GO to allow for increased concurrency.**

***Index Terms*: Web Crawler, GIL (Global Interpreter Lock), asyncio, aiohttp, Python scrapers.**

I. INTRODUCTION

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. 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 e-mail addresses (usually for spam).

Concurrent crawlers can utilize multi-handling or multi-threading. Each procedure or string works like a consecutive crawler, aside from they share information structures: wilderness and archive. Shared information structures must be synchronized (bolted for concurrent composes).

The asyncio module gives a structure that spins around the occasion circle. An occasion circle essentially sits tight 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 has a few circle executions accessible to it. The module will default to the one well on the way to being the best product 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 do 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 handler. 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 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 an 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.

II. PROBLEM STATEMENT

Python would seem the perfect language for writing web scrapers and crawlers. Libraries such as BeautifulSoup, Requests, and lxml give programmers solidAPIss to make requests and parse the data given back by web pages.

The only issue is that default Python web scrapers and crawlers are relatively slow. This is due to the issues that Python has with concurrency due to the languages GIL (Global Interpreter Lock). Compared with languages such as Golang and implementations of languages such as NodeJS building truly concurrent crawlers in Python is more challenging.

This lack of concurrency slows down crawlers due to the scripts simply idling while they await the response from the webserver in question. This is particularly frustrating if some of the pages discovered are particularly slow.

III. OBJECTIVE

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 requests each of the websites one after the other. The second version i.e., concurrent futures make use of concurrent futures' thread pool executor allowing us to send concurrent requests by making use of threads. Finally, we are going to take a look at a version of the script using async io and aiohttp, allowing us to make concurrent required requests using the event loop.

IV. COMPARISON

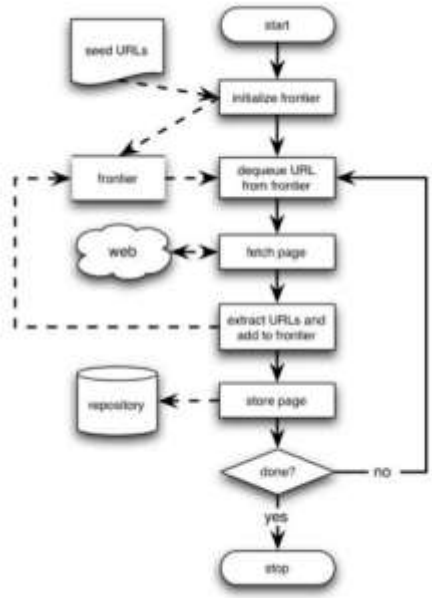


Fig 1: - Sequential Crawler

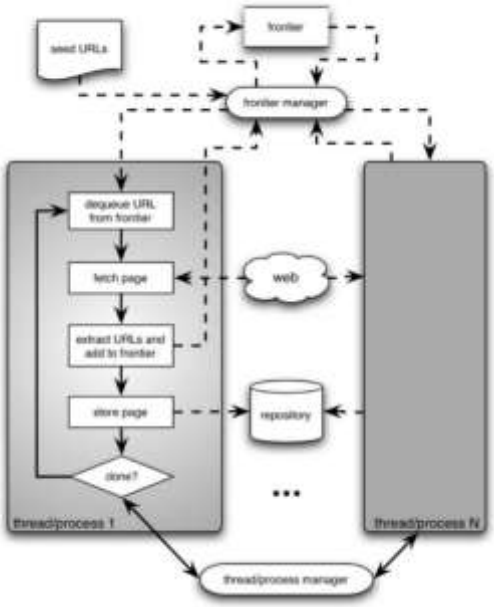


Fig 2: - Concurrent Crawler

Throughout the project, we have used python as the common ground for comparison and we have tried to optimize the code in python. However, we keep forgetting that python is a scripting language and it is dynamically typed so it is obviously slow to compile and therefore its runtime performance will be quite poor. For this purpose, we have tried to compare the performance of python by building a crawler in another language: Go. Go is much faster and its performance is almost comparable to that of C. Additionally, it has built-in libraries that can be used to make HTTP requests, making it much faster than Python. Additionally, Goroutines support in-built concurrency, adding more to the robustness and efficiency of the Go code. We would then analyze the time differences between all the codes and understand patterns and prove the conjectures.

V. LITERATURE REVIEW

1. ***IN [CGM 2002], the authors discuss various criteria and options for parallelizing the crawling process.***
   A crawler can either be centrally managed or distributed. A crawler can be planned to overlook the cover of pages that are downloaded while dealing with network load or the other way around. The creators characterize the nature of a crawler as its capacity to download "vital" pages previously others. For a parallel crawler, this metric is critical as each slithering procedure concentrates just on neighborhood information for checking pages as imperative. The writers notice that circulated crawlers are more profitable than multithreaded crawlers or independent crawlers on the checks of adaptability, proficiency, and throughput. If system scattering and system stack decrease are done, parallel crawlers can yield great outcomes. Their framework uses the memory of the machines and there is no plate to get to. They didn't store their information to demonstrate the handiness of a parallel creeping framework. Their objective was a little arrangement of news locales.
2. ***Mercator is a scalable and extensible crawler, now rolled into the Altavista search engine.***
   The authors of [HNM 1999] discuss implementation issues to be acknowledged for developing a parallel crawler like traps and bottlenecks, which can deteriorate performance. They talk about the upsides and downsides of various coordination modes and assessment criteria. The creators abridge their work with great execution measurements. In a nutshell, they agree that the correspondence overhead does not increment directly as more crawlers are included, the throughput of the framework increments straightly as more hubs are included and the nature of the framework, i.e., the capacity to get "essential" pages first, does not diminish with increment in the number of crawler forms. Fetterly and others in their work [FMN 2003] amassed their information utilizing the Mercator crawler. They slithered 151 million HTML pages totaling 6.4 TB over a time of 9 days.

3. ***In [LKC 2004], the authors implement a distributed system of crawlers in a point-to-point network environment.***
   Data to be communicated to and from amongst systems are stored in dynamic hash tables (DHTs). The crawler is an on-the-fly crawler. It brings pages guided by a client's pursuit. Notwithstanding when the hunt triggers the slithering undertaking,

there is no pruning of URLs and prohibition pages dependent on the substance. This can result in low-quality results returned. They execute URL-redirection among hubs to adjust the heap however the criteria for redirection of URLs are vague. The utilization of memory because of DHTs and the coordination isn't recorded. Be that as it may, they guarantee to rea ach download speed of around 1100 kBps every second with 80 hubs.

4. ***Authors of [PSM 2004] have implemented a location-aware system where globally distributed crawlers fetch pages from servers that are nearest to them.***
   The framework is strong to disappointments yet it has a long strength time and the element is hindering the association of information. At the point when a crawler goes down, the spaces it was slithering are exchanged for different crawlers. These outcomes in a substantial cover of information. An overwhelming heartbeat convention is expected to pick the closest crawler. Also, a crawler closer to many Web servers may be over-burden while a server at somewhat more separation might sit inert.

5. ***A recent paper, [JHH 2006] was discovered late during my research but it is worth mentioning here as the architecture of my system and their system have a lot in common.***
   In their work, the authors describe Stanford's WebBase system which is the academic version of Google! repository and search engine before it was commercialized in the late 1990s. The segment of intrigue is the crawler module. They actualize 500 procedures devoted to the errand of slithering. The way toward creeping is parallelized among the machines. A site with a completely qualified area name is taken as a nuclear unit for the creeping procedure and it ought not to be slithered by more than one machine for a slithering cycle. The creators state great focus on why a site ought to be treated as autonomous parts. For instance, it makes the administration of crawlers substantially simpler requiring little coordination among crawlers and the URL information structures can be put away in the fundamental memory. One can fortify site-particular creeping arrangements as well; like booking the season of the slither. In any case, they slither every one of the locales in the meantime (9 A.M. to 6 P.M.PST). This can be enhanced where the area of servers is mulled over. One can slither the destinations around evening time as per the area of servers or amid the small long stretches of mornings when the servers expect a lesser load. Amid the day, one can creep in a non-meddlesome manner with an expansive deferral between sequential gets. The creators allude to this deferral as affability delay and it ranges from 20 seconds for littler sites to 5 seconds for bigger sites. The kindness stops and cordiality approaches like regarding the robot's rejection convention assume a noteworthy job in the plan of a framework. The framework has an edge of 3000 pages to be crept per site, on the multi-day. The crawlers slither the pages which have a place with one space at any given moment. There are two kinds of slithering strategies portrayed. One, there is a procedure committed to creep each site and a procedure that slithers various destinations at the same time. The creators talk about focal points and tradeoffs between two procedures. Their framework actualizes a mix of the two sorts of procedures. The framework interprets area names as IP addresses for the seed URLs. IP locations of locales are utilized to get more pages from destinations. The creators express that on the off chance that one doesn't resolve area names, one could encounter an execution hit and exactness misfortune because of high DNS inquiries. Be that as it may, the strategy falls flat if there are different virtual servers actualized by facilitating administrations or if a space trades specialist organization. Moreover, robots.txt documents on the destinations facilitated on virtual servers can't be recovered. This and different components settle on the decision of settling a space name amid the beginning of a slither more costly and troublesome. To crawl, the creator's reason that regarding a site as one unit lifts the parallelization of a slithering procedure.

   Using a single process for the task of crawling in her work [SNC 2006], Patankar crawled 1 million pages in nine days. Data was loaded into the main memory from the tables and several queues were used to represent the frontier. HTML pages and images were crawled from five .edu domain

VI. NOVELTY

The project plots a detailed competitive analysis of various types of crawlers which thereby infers its efficiency. Go having highly advanced concurrent tools proved out to be way faster than Python (using asyncio and aiohttp), normal concurrent crawlers available in the market, and sequential crawlers. This project provides an in-depth analysis of the above-mentioned languages and brings out their performance analysis in such networking programs.

The project explains a great deal astof why sequential crawlers fail to match concurrent crawlers, as the job is not divided amongst various threads. It also illustrates why Go performs better than other programming languages when it comes to concurrency and mining data from the web. Go with its enhanced tools for concurrency proves to be fast, much more efficient, and robust. The above problem not being addressed to the common public in dummy terms has put Go in the shadow.

VII. EXISTING METHOD

The Comparison of performance with other systems - the data for other systems is taken from [SNC 2006]

Crawler	Distributed Needle	Needle	Google	Mercator	Internet Archive
Year	2006	2006	1997	2001	-
Machine	Intel P4 1.8 GHz, 1 GB RAM, 225 GB Hard disk	Configuration Intel P4 1.8 GHz, 1 GB RAM, 225 GB Hard disk	-	4 Compaq DS20E 666 MHz, Alpha servers	-
Data structures for URLs	Perl Scalar	Queue	-	Memory Hash table, disk sorted list	Bloom Filter per domain
DNS Solution	-	Stored locally in database	Local Cache	Custom	-
Programming Language	Perl	Perl	C++/ Python	Custom JVM	-
Parallelism per machine	1	1	-	100	64
# of crawling processes in the system	5-6	1	500	-	-
System size	5-6	1	-	4	-
Number of pages	1 Million	1 Million	24 Million	151 Million	100 Million
Crawl Rate (pages/sec)	3.85	1.7	48	600	10
Effective Crawl Rate (pages/second) = crawl rate/ (system size x parallelism)	0.77	1.7	-	600/400 = 1.5	10/64 = 0.15

Table 1 Comparison with other systems

VIII. Proposed Methodology

The first step in the project was to write the code for a sequential crawler that will linearly crawl all the pages. 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. Similarly, a code for a 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 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:

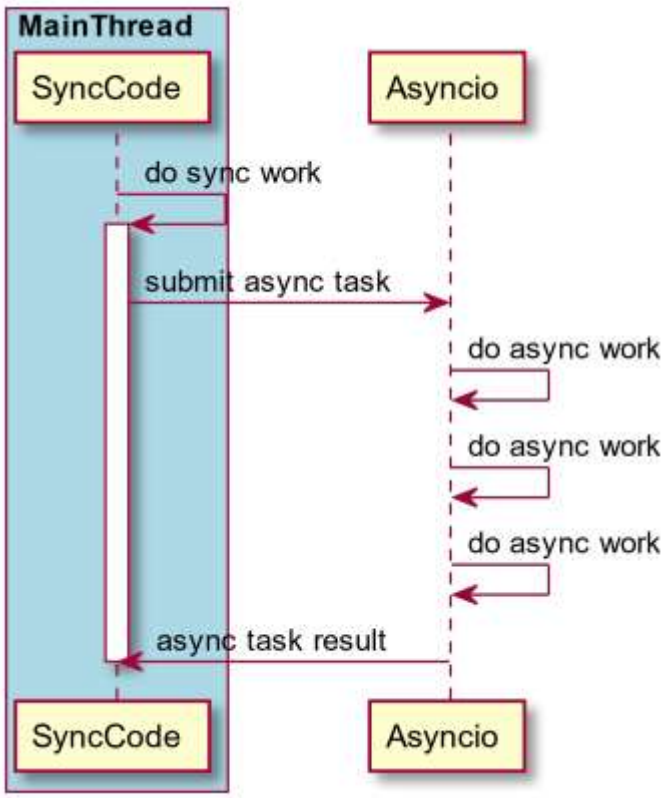
$$Latency = Time\ required\ to\ fetch\ one\ page + negligible\ time\ to\ process\ all\ requests$$
$$Effective\ Latency \approx Time\ required\ to\ fetch\ one\ page$$

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

$$Latency \approx k \times Time\ required\ to\ fetch\ one\ page$$

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 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 and aiohttp libraries to make concurrent requests using 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.

The event loop is central to the execution of the asynchronous functions. The event loop starts with registering or calling or canceling the co-routines. A coroutine is an asynchronous function in Python. It involves using the keyword async before its definition. The event loop may also involve building a 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.



An event loop is created by:
loop = asyncio.get_event_loop()

An asynchronous function can be executed asynchronously as shown below:

```
async def asynchr():
   print ("This is asynchronous")
loop = asyncio.get_event_loop()
loop.run_until_complete(asynchr())
loop.close()
```

This will keep executing asynchr() till all the tasks are executed. In the scenario of the crawler, the number of tasks created is 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 a 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.

IX. RESULTS

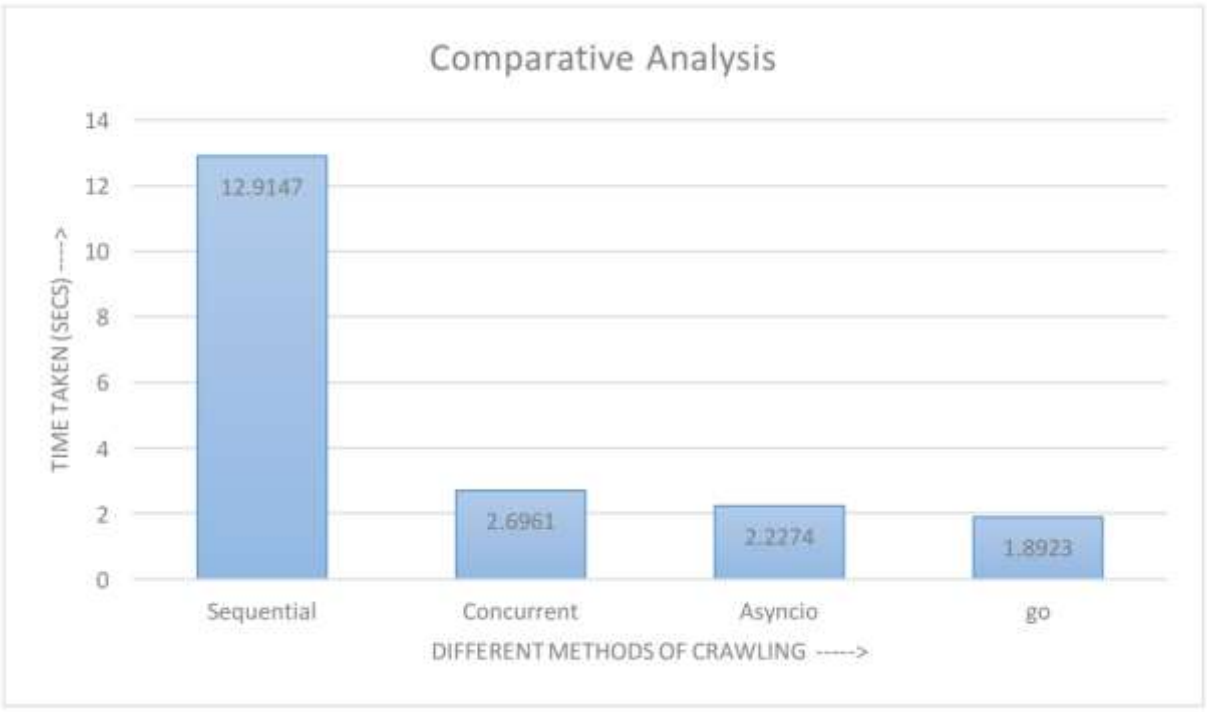
The observed time taken to crawl the given set of pages sequentially is 12.9147 seconds.

The observed time taken to crawl the given set of web pages using concurrent futures is 2.6961 seconds which is much lesser than the time taken in a sequential crawler.

The observed time taken to crawl the given set of web pages using asyncio and aiohttp script is 2.2274 seconds which is comparatively lesser than the time taken by a normal concurrent crawler. Using it may be a little more complicated giving us increased power and much more enhanced performance.

```
C:\gocode\src\crawler>go run init.go
2018/11/13 11:31:16 ||  [Processing] Spawning subroutines :  10
2018/11/13 11:31:16 ||  [Processing] Fetching page content :  http://codeforces.com/profile/zacker-22
2018/11/13 11:31:16 ||  [Processing] Fetching page content :  https://archive.org/details/opensource_movies
2018/11/13 11:31:16 ||  [Processing] Fetching page content :  http://codeforces.com/profile/jheel4
2018/11/13 11:31:16 ||  [Processing] Fetching page content :  https://httpbin.org/ip
2018/11/13 11:31:16 ||  [Processing] Fetching page content :  http://example.com
2018/11/13 11:31:16 ||  [Processing] Fetching page content :  http://codeforces.com/profile/the_pritisher
2018/11/13 11:31:16 ||  [Processing] Fetching page content :  http://codeforces.com/
2018/11/13 11:31:16 ||  [Processing] Fetching page content :  https://www.codechef.com/
2018/11/13 11:31:16 ||  [Processing] Fetching page content :  http://codeforces.com/profile/cipher.
2018/11/13 11:31:16 ||  [Processing] Fetching page content :  http://codeforces.com/profile/nikhilsaboo
2018/11/13 11:31:17 ||  [Processing] Writing to the file :  crawler\example.com\index.html
2018/11/13 11:31:17 ||  [Success] Crawled page :  http://example.com
2018/11/13 11:31:17 ||  [Processing] Writing to the file :  crawler\.html
2018/11/13 11:31:17 ||  [Processing] Writing to the file :  crawler\profile\nikhilsaboo.html
2018/11/13 11:31:17 ||  [Processing] Writing to the file :  crawler\profile\zacker-22.html
2018/11/13 11:31:17 ||  [Processing] Writing to the file :  crawler\profile\jheel4.html
2018/11/13 11:31:17 ||  [Processing] Writing to the file :  crawler\profile\the_pritisher.html
2018/11/13 11:31:17 ||  [Processing] Writing to the file :  crawler\profile\cipher.
2018/11/13 11:31:17 ||  [Processing] Writing to the file :  crawler\ip.html
2018/11/13 11:31:17 ||  [Success] Crawled page :  http://codeforces.com/profile/nikhilsaboo
2018/11/13 11:31:17 ||  [Success] Crawled page :  http://codeforces.com/profile/zacker-22
2018/11/13 11:31:17 ||  [Success] Crawled page :  https://httpbin.org/ip
2018/11/13 11:31:17 ||  [Success] Crawled page :  http://codeforces.com/profile/jheel4
2018/11/13 11:31:17 ||  [Success] Crawled page :  http://codeforces.com/profile/the_pritisher
2018/11/13 11:31:18 ||  [Success] Crawled page :  http://codeforces.com/profile/cipher.
2018/11/13 11:31:18 ||  [Success] Crawled page :  http://codeforces.com/
2018/11/13 11:31:18 ||  [Processing] Writing to the file :  crawler\.html
2018/11/13 11:31:18 ||  [Success] Crawled page :  https://www.codechef.com/
2018/11/13 11:31:18 ||  [Processing] Writing to the file :  crawler\details\opensource_movies.html
2018/11/13 11:31:19 ||  [Success] Crawled page :  https://archive.org/details/opensource_movies
2018/11/13 11:31:19 ||  [Status] Failed urls :  []
2018/11/13 11:31:19 The time taken to crawl the given web pages: 1.8923421265
```

The observed time taken to crawl the given set of web pages using go is 1.8923 seconds. Go doesn't have a third-party dependency for concurrency which can play a major role in increasing its run time speed.



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

It is easily evident from the above graph that the time taken by the concurrent crawler to crawl the given web pages (8) is much lesser when compared to the performance of the sequential crawlers. Go can give us an additional benefit in terms of performance.

A drastic difference in terms of time taken will be observed when we increase the number of web pages to be crawled. Go will give us additional concurrency benefits when the number of web pages is large.

We, therefore, conclude, that Go is much faster and its performance is almost comparable to that of C. Additionally, it has built-in libraries that can be used to make HTTP requests, making it much faster than Python. Additionally, Go routines support in-built concurrency, adding more to the robustness and efficiency of the Go code.

No. of URLs	No- Concurrency	Concurrent Futures	Asyncio & Aiohttp
5	4.021 secs	1.098 secs	1.3197 secs
50	79.2116 secs	28.82 secs	31.5012 secs
100	157.5677 secs	60.1970 secs	45.4405 secs

Table 2 Speed Comparisons

X. 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. Crawlers are utilized via web crawlers, for instance, to screen the volume of data on the Web. Yet, the size of the Web implies that far-reaching crawling devours a considerable measure of handling power, which normally implies building colossal server farms to control the product.

As the amount of data on the Internet keeps on developing, so does the subject of how to process everything and make it helpful.

Go makes it substantially less demanding to compose strong, organized applications, without yielding much in the execution method, as contrasted with C or C++. The superior is in extensive part because of the static gathering of the statically-composed Go code. A considerable measure of enhancements is conceivable when a compiler can do all the code review work heretofore, instead of the dynamic JS compiler work done amid runtime. The following conclusions can be drawn based on the results obtained as it clearly shows with the rising number of web pages the amount of time required for other crawlers to crawl through is significantly large.

This mining configuration is significantly more productive. Presently numerous threads (gophers) are working freely; accordingly, the entire activity isn't all on Gary. A gopher is finding the mineral, one mining the metal, and another purifying the ore—potentially all in the meantime.

With the end goal for us to bring this kind of usefulness into our code, we're going to require two things: an approach to make autonomously working gophers, and a path for gophers to discuss (send minerals) to one another. This is the place Go's simultaneousness natives come in: goroutines and channels

References

[1] [CGM 2002] J. Cho and H. Garcia-Molina, Parallel crawlers. In Proceedings of the Eleventh International World Wide Web Conference, 2002, pp. 124 - 135, http://oak.cs.ucla.edu/~cho/papers/cho-parallel.pdf. http://oak.cs.ucla.edu/~cho/papers/cho-parallel.pdf

[2] [HNM 1999] A. Heydon and M. Najork, Mercator: A scalable, extensible web crawler. World Wide Web, vol. 2, no. 4, pp. 219 -229, 1999., http://citeseer.nj.nec.com/heydon99mercator.html

[3] [BCS 2002] P. Boldi, B. Codenotti, M. Santini, and S. Vigna, Ubicrawler: A scalable fully distributed web crawler. In Proceedings of AusWeb02 - The Eighth Australian World Wide Web Conference, Queensland, Australia, 2002, http://citeseer.ist.psu.edu/boldi02ubicrawler.html

[4] [FMN 2003] D. Fetterly, M. Manasse, M. Najork, and J. Wiener. A large-scale study of the evolution of web pages. In Proceedings of the twelfth international conference on World Wide Web, Budapest, Hungary, pages 669-678. ACM Press, 2003. https://dl.acm.org/doi/10.1145/775152.775246

[5] [BYC- 2002] Baeza-Yates and Castillo, WIRE Project 2002, http://www.cwr.cl/projects/WIRE/

[6] [JHH 2006] 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 2006

[7] JGM 2003] J Cho and H Garcia-Molina, Effective Page Refresh Policies for Web Crawlers, ACM Transactions on Database Systems, 2003