

# Parallelization of Web Crawler with Multithreading

J-Component Report

submittedby

TEAM MEMBERS:

**Fardeen Shaikh 17BCE0796**

**Jasleen Saggu 17BCE2335**

**Soumi Hazra 17BCE0446**

*in partial fulfillment for the award of the degree of* **B.Tech**

in

**COMPUTER SCIENCE ENGINEERING**

Under the guidance of

**Faculty: Prof. Balamurugan R**

# School of Computing Science and Engineering

Winter Semester 2019-20

1



# School of Computer Science and Engineering

**DECLARATION**

I hereby declare that the J Component report entitle “Parallelization of Web Crawler with Multithreading**”** submitted by me to Vellore Institute of Technology, in partial fulfillment of the requirement for the award of the degree of **B.Tech** in **Computer science and engineering** is a record of bonafide undertaken by me under the supervision of **Prof. Balamurgan R.** I further declare that the work reported in this report has not been submitted and will not be submitted, either in part or in full, for the award of any other degree or diploma in this institute or any other institute or university.

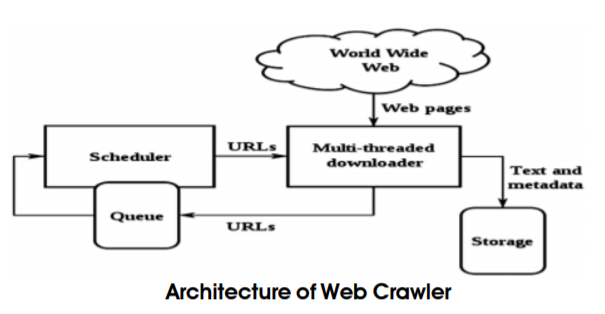
2

1. **ABSTRACT**

A parallel crawler is a crawler that runs multiple processes in parallel. The goal is to maximize the download rate while minimizing the overhead from parallelization and to avoid repeated downloads of the same page. By default, Python programs are single threaded. This can make scraping an entire site using a Python crawler extremely slow. Crawlers consume resources on visited systems and often visit sites without approval. Issues of schedule, load, and "politeness" come into play when large collections of pages are accessed. Mechanisms exist for public sites not wishing to be crawled to make this known to the crawling agent. The number of Internet pages is extremely large; even the largest crawlers fall short of making a complete index. For this reason, search engines struggled to give relevant search results in the early years of the World Wide Web, before 2000. Today, relevant results are given almost instantly. The number of possible URLs crawled being generated by server-side software has also made it difficult for web crawlers to avoid retrieving [duplicate content](https://en.wikipedia.org/wiki/Duplicate_content). Endless combinations of [HTTP GET](https://en.wikipedia.org/wiki/HTTP) (URL-based) parameters exist, of which only a small selection will actually return unique content. A Web crawler starts with a list of [URLs](https://en.wikipedia.org/wiki/Uniform_Resource_Locator) to visit, called the seeds. As the crawler visits these URLs, it identifies all the [hyperlinks](https://en.wikipedia.org/wiki/Hyperlink) in the pages and adds them to the list of URLs to visit, called the [crawl frontier](https://en.wikipedia.org/wiki/Crawl_frontier). URLs from the frontier are [recursively](https://en.wikipedia.org/wiki/Recursion) visited according to a set of policies. If the crawler is performing archiving of [websites](https://en.wikipedia.org/wiki/Website), it copies and saves the information as it goes. Crawlers usually perform some type of [URL normalization](https://en.wikipedia.org/wiki/URL_normalization) in order to avoid crawling the same resource more than once. The term URL normalization, also called URL canonicalization, refers to the process of modifying and standardizing a URL in a consistent manner. There are several types of normalization that may be performed including conversion of URLs to lowercase, removal of "." and ".." segments, and adding trailing slashes to the non-empty path component. Through this project, we will demonstrate web crawling of multiple child node in one go from the parent node using the concept of multithreading in python program.

## INTRODUCTION

Python is a great language for writing web scrapers and web crawlers. Libraries such as BeauitfulSoup, requests and lxml make grabbing and parsing a web page very simple. By default, Python programs are single threaded. This can make scraping an entire site using a Python crawler extremely slow. We must wait for each page to load before moving onto the next one. Python supports threads which though not appropriate for all tasks, can help us increase the performance of our web crawler. A parallel crawler is a crawler that runs multiple processes in parallel. The goal is to maximize the download rate while minimizing the overhead from parallelization and to avoid repeated downloads of the same page. To avoid downloading the same page more than once, the crawling system requires a policy for assigning the new URLs discovered during the crawling process, as the same URL can be found by two different crawling processes. The Web comprises of voluminous rich learning content. The volume of ever growing learning resources however leads to the problem of information overload. A large number of irrelevant search results generated from search engines based on keyword matching techniques further augment the problem. Keeping in view the volume of content a significant crawler of semantic knowledge can be built on demonstrated multi-threaded unfocused crawler implemented in this project.



## REQUIREMENTS

Anaconda Navigator, Spyder, BeautifulSoup, Requests, URLlib

## KEYWORDS

Multithreading, Parallelization, Crawler, Unfocused, Concurrent Crawling

## LITERATURE SURVEY

Web crawlers follow links in webpages and automatically download the page[1]. Crawling is the most basic as well as the most powerful web search procedure. A design and framework of a multithreaded web crawler has been proposed. The end result of the process is a collection of web pages. The experiment demonstrates how this process has better performance. It is possible to increase the performance of a search by understanding the user’s need and the relevance of the document.

The computer network is vast and act as a single huge network for data and message transportation across vast distances. The World wide web has millions of pages linked and is ever growing. Hence, they continuously keep track of the web an find new webpages.

This system called the semantic web works on shared meaningful knowledge representation. It proposes to make an AI application that will make the content of the web meaningful to the system. This will make the crawler must more efficient by crawling only those pages which has relevance to the users requirements[2].

Initially a seed URL is given input by the user. Crawler then fetches the webpage from the interne. Contents of the page are parsed and the URL are scraped and stored in Database. If the links are to be traversed, the links are clicked and passed onto the web crawlers. Crawler then searches the keyword if present in the page fetched using the URL. If present, it calculates the number of times it is present in the page. A brief evaluation study of the crawler in an uncontrolled and practical environment is the main objective of this research paper.[3]

The World Wide Web (or simply the web) is a vast, prosperous, superior, readily available and suitable source of information and its users are growing very fast nowadays. To extract information from the

web, Search engines are used that access web pages as per the requirement of the users. Most of the data in the web is unmanaged so it is not possible to use the entire web at once in a single attempt, so search engines use the web crawler. Web crawler is an important part of search engine. This is a program that navigates the web and downloads reference to web pages.[4]

Search engines run multiple instances of crawlers on wide spread serversGet various information from them. Web crawler crawls from page topage in world wide. Get the webpage, load the page content and index itto find the engines database.This paper presents a systematic study of the web Crawler. The study ofweb crawler is very important because properly designed web crawler always gives good yield.Web crawler are an efficient component of the search engine. It is the core member responsible for searching the web and indexing the traversed webpages. It starts with a few seed pages, travels the web using the links in the webpages and inserts new link in the system database[5]

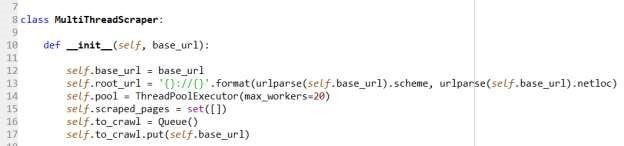
In the paper ―Finding Near-Duplicate Web Pages: A Large-Scale Evaluation of Algorithms‖ the author stated that Broder et al.'s [7] shingling algorithm and Charikar's [6] random projection based approach are considered \state-of-the-art" algorithms for finding near-duplicate web pages. Both algorithms were either developed at or used by popular web search engines. They compare the two algorithms on a very large scale, namely on a set of 1.6B distinct web pages.

## METHODOLOGY

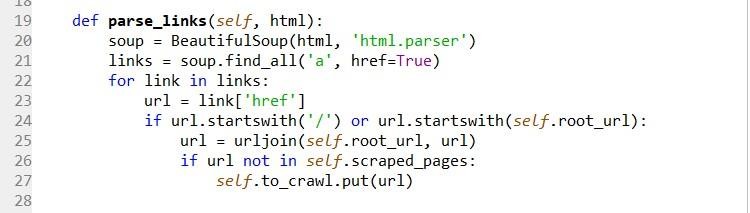
An adopted methodology for web crawling is when a query is passed to search engine by user, web crawler starts then from a set of seed pages. Page downloader fetches the webpage for a particular URL. The downloaded page is passed onto the extractor which extracts contents and parses them to get the links. Frontier starts indexing all the received links. The goal is to maximize the download rate while minimizing the overhead from parallelization and to avoid repeated downloads of the same page. To avoid downloading the same page more than once, the crawling system requires a policy for assigning the new URLs discovered during the crawling process, as the same URL can be found by two different crawling processes. It works like an ackerman function where the key methods of a crawler are packed within a function that is called for requests.

## IMPLEMENTATION

**Setting Up Our Class**

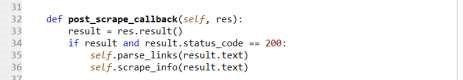
Initialize the class to create a crawler which only takes one argument. Pass the start URL as an argument, and from this use urlparse from the urllib.parse library to pull out the sites homepage. This root URL is going to be used later to ensure that our scraper doesn’t end up on other sites.We also initialise a thread pool. We are later going to submit ‘tasks’ to this thread pool, allowing us to use a callback function to collect our results. This will allow us to continue with execution of our main program, while we await the response from the website.

**Parsing Links and Scraping**

Next, we write a basic link parser. Our goal here is to extract all of a sites internal links and not to pull out any external links. We can then check whether this link is relative (starting with a ‘/’) or starts with our root URL. If this is the case we can then use URL join to generate a crawlable URL and then put this in our queue provided we haven’t already crawled it.

**Defining Our Callback**

This function will execute after the previous function has completed and will be passed the result of our previous function as an argument.

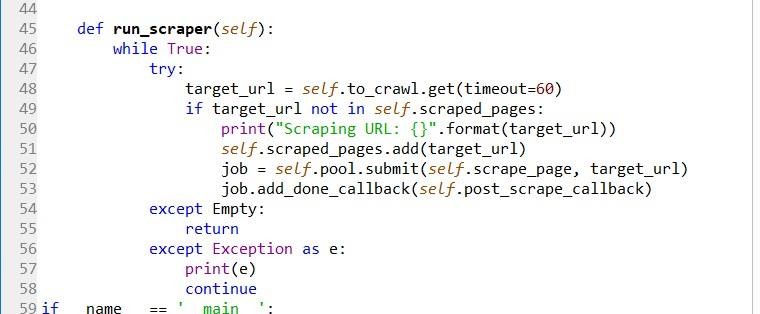


**Scraping Pages**

Define a function which will be used to scrape the page. This function is very simple and simply takes a URL and returns a response object if it was successful. Otherwise we return ‘None’. By limiting the amount of CPU bound work we can increase the overall speed of our crawler.

**Run Scraper Function**

The run scraper function brings all of our previous work together and manages our thread pool.



1. **PYTHON CODE:**

import requests

from bs4 import BeautifulSoup

from queue import Queue, Empty

from concurrent.futures import ThreadPoolExecutor

from urllib.parse import urljoin, urlparse

class MultiThreadScraper:

def \_\_init\_\_(self, base\_url):

self.base\_url = base\_url

self.root\_url = '{}://{}'.format(urlparse(self.base\_url).scheme, urlparse(self.base\_url).netloc)

self.pool = ThreadPoolExecutor(max\_workers=20)

self.scraped\_pages = set([])

self.to\_crawl = Queue()

self.to\_crawl.put(self.base\_url)

def parse\_links(self, html):

soup = BeautifulSoup(html, 'html.parser')

links = soup.find\_all('a', href=True)

for link in links:

url = link['href']

if url.startswith('/') or url.startswith(self.root\_url):

url = urljoin(self.root\_url, url)

if url not in self.scraped\_pages:

self.to\_crawl.put(url)

def scrape\_info(self, html):

return

def post\_scrape\_callback(self, res):

result = res.result()

if result and result.status\_code == 200:

self.parse\_links(result.text)

self.scrape\_info(result.text)

def scrape\_page(self, url):

try:

res = requests.get(url, timeout=(3, 30))

return res

except requests.RequestException:

return

def run\_scraper(self):

while True:

try:

target\_url = self.to\_crawl.get(timeout=60)

if target\_url not in self.scraped\_pages:

print("Scraping URL: {}".format(target\_url))

self.scraped\_pages.add(target\_url)

job = self.pool.submit(self.scrape\_page, target\_url)

job.add\_done\_callback(self.post\_scrape\_callback)

except Empty:

return

except Exception as e:

print(e)

continue

if \_\_name\_\_ == '\_\_main\_\_':

s = MultiThreadScraper("https://vit.ac.in/")

s.run\_scraper()

1. **OUTPUT**

Scraping URL: <https://vit.ac.in/>

Scraping URL: [https://vit.ac.in](https://vit.ac.in/)

Scraping URL: <https://vit.ac.in/about-vit>

Scraping URL: <https://vit.ac.in/about/vision-mission>

Scraping URL: <https://vit.ac.in/about/leadership>

Scraping URL: <https://vit.ac.in/about/administrative-offices>

Scraping URL: <https://vit.ac.in/about/Sustainability>

Scraping URL: <https://vit.ac.in/about/true-green>

Scraping URL: <https://vit.ac.in/about/community-outreach>

Scraping URL: <https://vit.ac.in/about/communityradio>

Scraping URL: <https://vit.ac.in/about/infrastructure>

Scraping URL: <https://vit.ac.in/about/ranking-and-accreditation>

Scraping URL: <https://vit.ac.in/all-news-archieved>

Scraping URL: <https://vit.ac.in/all-events>

Scraping URL: <https://vit.ac.in/national-institutional-ranking-framework-nirf>

Scraping URL: <https://vit.ac.in/mhrdugc>

Scraping URL: <https://vit.ac.in/about/news-letter>

Scraping URL: <https://vit.ac.in/academics/home>

Scraping URL: <https://vit.ac.in/programmes-offered-2019-20>

Scraping URL: <https://vit.ac.in/schools>

Scraping URL: <https://vit.ac.in/academics/ffcs>

Scraping URL: <https://vit.ac.in/academics/library>

Scraping URL: <https://vit.ac.in/academics/centers>

Scraping URL: <https://vit.ac.in/academics/iqac>

Scraping URL: <https://vit.ac.in/academics/transcripts>

Scraping URL: <https://vit.ac.in/academics-feedback>

Scraping URL: <https://vit.ac.in/admissions/overview>

Scraping URL: <https://vit.ac.in/admissions/programmes-offered>

Scraping URL: <https://vit.ac.in/all-courses/ug>

Scraping URL: <https://vit.ac.in/all-courses/pg>

Scraping URL: <https://vit.ac.in/admissions/research>

Scraping URL: <https://vit.ac.in/admissions/international>

Scraping URL: <https://vit.ac.in/viteee>

Scraping URL: <https://vit.ac.in/placements/overview>

Scraping URL: <https://vit.ac.in/placements/Superdreamoffers>

Scraping URL: <https://vit.ac.in/placements/dreamoffers>

Scraping URL: <https://vit.ac.in/placements/internship>

Scraping URL: <https://vit.ac.in/placements/statistics>

Scraping URL: <https://vit.ac.in/placements/placement-records>

Scraping URL: <https://vit.ac.in/placements/PAT-Office>

Scraping URL: <https://vit.ac.in/InternationalRelations>

Scraping URL: <https://vit.ac.in/internationalrelations/itp>

Scraping URL: <https://vit.ac.in/internationalrelations/partneruniversities>

Scraping URL: <https://vit.ac.in/internationalrelations/SAP>

Scraping URL: <https://vit.ac.in/admissions/international/overview>

Scraping URL: <https://vit.ac.in/research>

Scraping URL: <https://vit.ac.in/research/academic>

Scraping URL: <https://vit.ac.in/research/centers-list>

Scraping URL: <https://vit.ac.in/research/sponsored-research>

Scraping URL: <https://vit.ac.in/campuslife/overview>

Scraping URL: <https://vit.ac.in/campuslife/fests>

Scraping URL: <https://vit.ac.in/campuslife/studentswelfare>

Scraping URL: <https://vit.ac.in/campuslife/Sports>

Scraping URL: <https://vit.ac.in/campuslife/hostels>

Scraping URL: <https://vit.ac.in/campuslife/startups>

Scraping URL: <https://vit.ac.in/campuslife/healthservices>

Scraping URL: <https://vit.ac.in/campuslife/hostelsfee>

Scraping URL: <https://vit.ac.in/campuslife/otheramenities>

Scraping URL: <https://vit.ac.in/detailview/green-vit>

Scraping URL: <https://vit.ac.in/academics/coe>

Scraping URL: <https://vit.ac.in/centers/asc>

Scraping URL: <https://vit.ac.in/guest-house>

Scraping URL: <https://vit.ac.in/hotels-in-vellore>

Scraping URL: <https://vit.ac.in/internal-complaints-committee>

Scraping URL: <https://vit.ac.in/redressal>

Scraping URL: <https://vit.ac.in/sites/default/files/SBST_Freshers_App.rar>

Scraping URL: <https://vit.ac.in/anti-ragging-committee>

Scraping URL: <https://vit.ac.in/capability-enhancement-scheme>

Scraping URL: <https://vit.ac.in/sites/default/files/FormatGuidelines.doc>

Scraping URL: <https://vit.ac.in/alumni_progression>

Scraping URL: <https://vit.ac.in/instruction>

Scraping URL: <https://vit.ac.in/ariia-award>

Scraping URL: <https://vit.ac.in/school-advanced-sciences-sas/stat-thon-ws-2020>

Scraping URL: <https://vit.ac.in/school-advanced-sciences-sas/math-thon-2020>

Scraping URL: <https://vit.ac.in/school-electrical-engineering-select/2-nd-virtual-conference-advances-electric-drives-process>

Scraping URL: <https://vit.ac.in/general/faculty-development-program-%E2%80%93-%E2%80%9Cautopilot-%E2%80%93-%E2%80%9D>

Scraping URL: <https://vit.ac.in/BTechAdmissions/>

Scraping URL: <https://vit.ac.in/detailview/vit-wishes-warm-%E2%80%98happy-birthday%E2%80%99-our-honourable-chancellor>

Scraping URL: <https://vit.ac.in/vit-institution-eminence-ioe>

Scraping URL: <https://vit.ac.in/files/MBA_online_interview/Instructions-to-candidates-for-MBA-online-interview.pdf>

Scraping URL: <https://vit.ac.in/files/MBA_online_interview/MBA-2020-Admissions-online-interview-candidates-date-and-time-schedule.pdf>

Scraping URL: <https://vit.ac.in/ranking>

Scraping URL: <https://vit.ac.in/academics/InternationalRelations>

Scraping URL: <https://vit.ac.in/admission-overview>

Scraping URL: <https://vit.ac.in/all-news>

Scraping URL: <https://vit.ac.in/mtech-mca-admissions>

Scraping URL: <https://vit.ac.in/vit-e-learning-portal-offers-more-courses-its-students>

Scraping URL: <https://vit.ac.in/vit-team-donates-3d-printed-face-shields-medical-college>

Scraping URL: <https://vit.ac.in/vit-donates-%E2%82%B9125-crore-cmprf>

Scraping URL: <https://vit.ac.in/galleries>

Scraping URL: <https://vit.ac.in/video>

Scraping URL: <https://vit.ac.in/campus-hostel/hostels>

Scraping URL: <https://vit.ac.in/sites/default/files/Student-Code-of-Conduct.pdf>

Scraping URL: <https://vit.ac.in/iprcell>

Scraping URL: <https://vit.ac.in/sites/default/files/footer_menu_doc/FormatGuidelines%20%283%29.doc>

Scraping URL: <https://vit.ac.in/campus-category/Grievance> Cell

Scraping URL: <https://vit.ac.in/contactus>

Scraping URL: <https://vit.ac.in/sites/default/files/history.png>

Scraping URL: <https://vit.ac.in/sites/default/files/vit_history%20%281%29%20%281%29.jpg>

Scraping URL: <https://vit.ac.in/sites/default/files/vit_history%20%282%29%20%282%29.jpg>

Scraping URL: <https://vit.ac.in/sites/default/files/vit_history%20%283%29%20%282%29.jpg>

Scraping URL: <https://vit.ac.in/sites/default/files/vit_history%20%284%29%20%282%29.jpg>

Scraping URL: <https://vit.ac.in/sites/default/files/vit_history%20%285%29%20%282%29.jpg>

Scraping URL: <https://vit.ac.in/sites/default/files/vit_history%20%286%29%20%282%29.jpg>

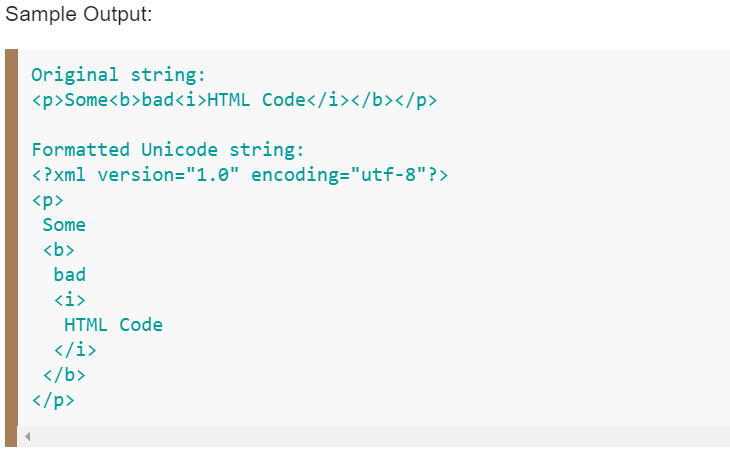
​

1. **DETAILS OF LIBRARIES INCLUDED**

**Beautifulsoup4:**

Beautiful Soup is a Python library for pulling data out of HTML and XML files. It works with your favorite parser to provide idiomatic ways of navigating, searching, and modifying the parse tree. It commonly saves programmers hours or days of work.





**Extract content within a tag with BeautifulSoup**



**concurrent.futures**

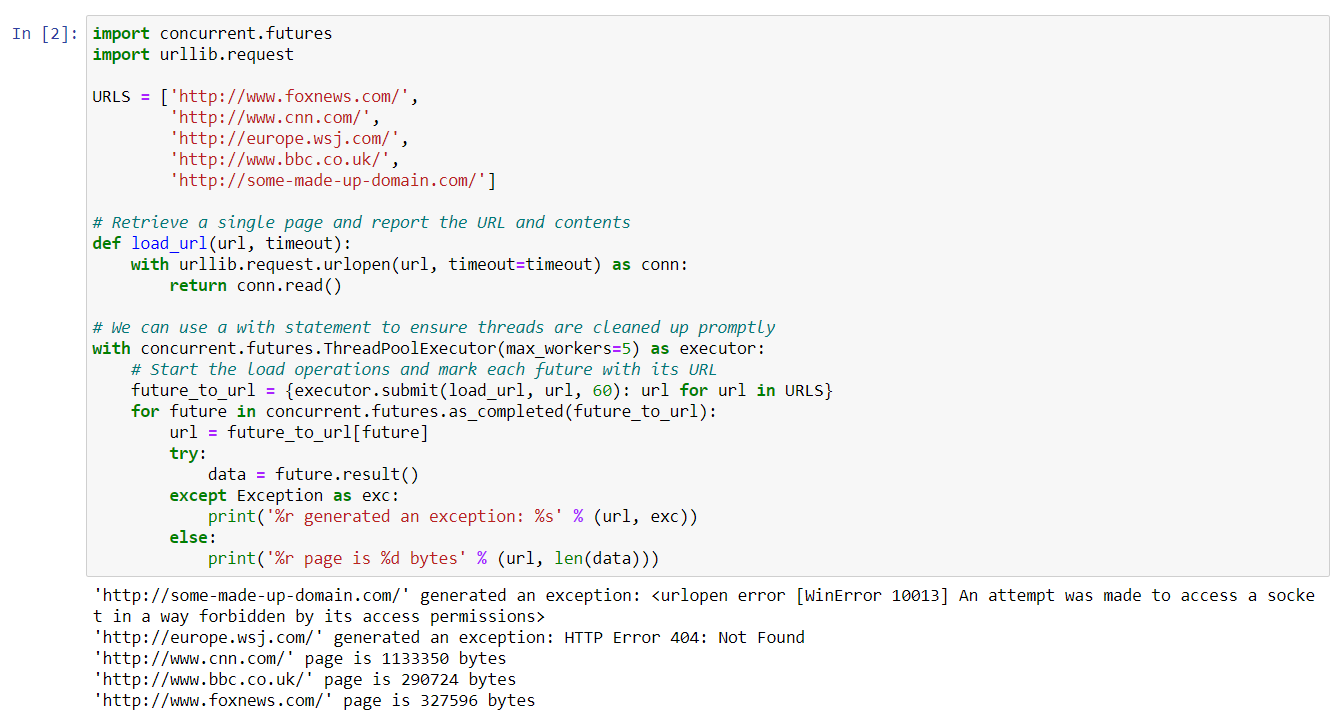
The concurrent.futures module provides a high-level interface for asynchronously executing callables.

The asynchronous execution can be performed with threads, using ThreadPoolExecutor, or separate processes, using ProcessPoolExecutor. Both implement the same interface, which is defined by the abstract Executor class.

**ThreadPoolExecutor**

ThreadPoolExecutor is an Executor subclass that uses a pool of threads to execute calls asynchronously.

Eg:



Executor Objects

class concurrent.futures.Executor

An abstract class that provides methods to execute calls asynchronously. It should not be used directly, but through its concrete subclasses.

submit(fn, \*args, \*\*kwargs)

Schedules the callable, fn, to be executed as fn(\*args \*\*kwargs) and returns a Future object representing the execution of the callable.

with ThreadPoolExecutor(max\_workers=1) as executor:

future = executor.submit(pow, 323, 1235)

print(future.result())

map(func, \*iterables, timeout=None, chunksize=1)

Similar to map(func, \*iterables) except:

the iterables are collected immediately rather than lazily;

func is executed asynchronously and several calls to func may be made concurrently.

The returned iterator raises a concurrent.futures.TimeoutError if \_\_next\_\_() is called and the result isn’t available after timeout seconds from the original call to Executor.map(). timeout can be an int or a float. If timeout is not specified or None, there is no limit to the wait time.

If a func call raises an exception, then that exception will be raised when its value is retrieved from the iterator.

When using ProcessPoolExecutor, this method chops iterables into a number of chunks which it submits to the pool as separate tasks. The (approximate) size of these chunks can be specified by setting chunksize to a positive integer. For very long iterables, using a large value for chunksize can significantly improve performance compared to the default size of 1. With ThreadPoolExecutor, chunksize has no effect.

Changed in version 3.5: Added the chunksize argument.

shutdown(wait=True)

Signal the executor that it should free any resources that it is using when the currently pending futures are done executing. Calls to Executor.submit() and Executor.map() made after shutdown will raise RuntimeError.

If wait is True then this method will not return until all the pending futures are done executing and the resources associated with the executor have been freed. If wait is False then this method will return immediately and the resources associated with the executor will be freed when all pending futures are done executing. Regardless of the value of wait, the entire Python program will not exit until all pending futures are done executing.

You can avoid having to call this method explicitly if you use the with statement, which will shutdown the Executor (waiting as if Executor.shutdown() were called with wait set to True):

import shutil

with ThreadPoolExecutor(max\_workers=4) as e:

e.submit(shutil.copy, 'src1.txt', 'dest1.txt')

e.submit(shutil.copy, 'src2.txt', 'dest2.txt')

e.submit(shutil.copy, 'src3.txt', 'dest3.txt')

e.submit(shutil.copy, 'src4.txt', 'dest4.txt')

**ThreadPoolExecutor**

ThreadPoolExecutor is an Executor subclass that uses a pool of threads to execute calls asynchronously.

Deadlocks can occur when the callable associated with a Future waits on the results of another Future. For example:

import time

def wait\_on\_b():

time.sleep(5)

print(b.result()) # b will never complete because it is waiting on a.

return 5

def wait\_on\_a():

time.sleep(5)

print(a.result()) # a will never complete because it is waiting on b.

return 6

executor = ThreadPoolExecutor(max\_workers=2)

a = executor.submit(wait\_on\_b)

b = executor.submit(wait\_on\_a)

And:

def wait\_on\_future():

f = executor.submit(pow, 5, 2)

# This will never complete because there is only one worker thread and

# it is executing this function.

print(f.result())

executor = ThreadPoolExecutor(max\_workers=1)

executor.submit(wait\_on\_future)

class concurrent.futures.ThreadPoolExecutor(max\_workers=None, thread\_name\_prefix='', initializer=None, initargs=())

An Executor subclass that uses a pool of at most max\_workers threads to execute calls asynchronously.

initializer is an optional callable that is called at the start of each worker thread; initargs is a tuple of arguments passed to the initializer. Should initializer raise an exception, all currently pending jobs will raise a BrokenThreadPool, as well as any attempt to submit more jobs to the pool.

Changed in version 3.5: If max\_workers is None or not given, it will default to the number of processors on the machine, multiplied by 5, assuming that ThreadPoolExecutor is often used to overlap I/O instead of CPU work and the number of workers should be higher than the number of workers for ProcessPoolExecutor.

New in version 3.6: The thread\_name\_prefix argument was added to allow users to control the threading.Thread names for worker threads created by the pool for easier debugging.

Changed in version 3.7: Added the initializer and initargs arguments.

Changed in version 3.8: Default value of max\_workers is changed to min(32, os.cpu\_count() + 4). This default value preserves at least 5 workers for I/O bound tasks. It utilizes at most 32 CPU cores for CPU bound tasks which release the GIL. And it avoids using very large resources implicitly on many-core machines.

ThreadPoolExecutor now reuses idle worker threads before starting max\_workers worker threads too.

1. **CONCLUSION**

When testing this script on several sites with performant servers, we were able to crawl several thousand URLs a minute with only 20 threads. Ideally, we would use a lower number of threads to avoid potentially overloading the site you are scraping. Although when crawling a heavy website like that of government, there are high risks of crawler being trapped on the website due to lack of semantic crawling.

However, it overcomes the wait for each page to load before moving onto the next one and help us increase the performance of our web crawler. A parallel crawler is a crawler that runs multiple processes in parallel. The goal is to maximize the download rate while minimizing the overhead from parallelization and to avoid repeated downloads of the same page to crawl multiple links using the concept of multithreading.

## REFERENCES

1. Chau, D.H., Pandit, S., Wang, S., Faloutsos, C. and Faloutsos, C., 2007, May. Parallel crawling for online social networks. In *Proceedings of the 16th international conference on World Wide Web* (pp. 1283-1284). ACM.
2. Castillo, C., 2005, June. Effective web crawling. In *Acm sigir forum* (Vol. 39, No. 1, pp. 55-56). Acm.
3. Thelwall, M., 2001. A web crawler design for data mining. *Journal of Information Science*, *27*(5), pp.319-325. [4]Shkapenyuk, V. and Suel, T., 2002, February. Design and implementation of a high-performance distributed web crawler. In *Proceedings 18th International Conference on Data Engineering* (pp. 357-368). IEEE.
4. Boldi, P., Codenotti, B., Santini, M. and Vigna, S., 2004. Ubicrawler: A scalable fully distributed web crawler. *Software: Practice and Experience*, *34*(8), pp.711-726.
5. M. S. Charikar. Similarity Estimation Techniques from Rounding Algorithms. In 34th Annual ACM Symposium on Theory of Computing (May 2002).
6. A. Broder, S. Glassman, M. Manasse, and G. Zweig. Syntactic Clustering of the Web. In 6th International World Wide Web Conference (Apr. 1997), 393-404.