Crawling through Goodreads book lists using multithreading and multiprocessing techniques

*Marija Trpcevska[1], *Vladimir Zdraveski [2]
*Ss. Cyril and Methodius University in Skopje, Faculty of Computer Science and Engineering, Skopje, North Macedonia
[1] marija.trpchevska@students.finki.ukim.mk, [2] vladimir.zdraveski@finki.ukim.mk

Abstract- Today's Web represents a vast, complicated network of web pages linking to other parts of themselves or to one another, making the task of crawling through them much harder even if the often related process of scraping data from them is overlooked. The aim of this paper is to show that effective, fast crawling for individual purposes could be done with some decent know-how of multithreading/ multiprocessing techniques and a minimal code setup relative to its reusability, showing the concept at work by crawling Goodreads book lists. The performance boosts evident when compared to simpler, sequential crawling justifies the efforts that have been put in.

1 Introduction

With the advent of the World Wide Web, huge swaths of information found themselves being created, published and saved online with no indication of the massive transfer of human knowledge slowing down anytime soon. Such was the exponential growth of the Web that early search engines struggled to provide relevant search results and web crawls aiming to keep track of all available web pages consistently fell short of making a complete index (researchers Steve Lawrence and Lee Giles showed that no search engine indexed more than 16% of the Web in 1999[1]). Therefore, conceptualizing crawls in a constructive manner became necessary, with Edwards[2] noting *'Given that the bandwidth for conducting crawls is neither infinite nor free, it is becoming essential to crawl the Web in not only a scalable, but efficient way, if some reasonable measure of quality or freshness is to be maintained.'* Solutions to the problem arose just as quickly and are further elaborated upon in the Related Work section.

More recently, focused crawls are of interest to companies looking to search through the competition's websites and check for price changes, archive the Web in order to preserve Internet culture, create a mirror of a website for offline viewing etc. The goal hereby presented is to perform a focused crawl of Goodreads, an American social cataloging website and a subsidiary of Amazon that allows individuals to search its database of books, annotations, quotes, and reviews with the aim of extracting information. How the process is conducted is explained in the section Solution Architecture, giving an overview of the technologies used and the logic behind the code implementation. Reasons for using multithreading alone or combined with multiprocessing are considered. In Results, the code is tested with runs on a concrete list, a comparison is shown between both configurations done locally and on a remote server. In Conclusion, closing remarks and observations on future upgrades to the crawler are outlined.

2 Related Work

The first parallel crawler - WebCrawler - appeared in 1994, being able to download 15 links simultaneously. Soon however, the issue of size made huge crawls much too slow and computationally expensive. The need for an architecture that will scale hand in hand with the growth of the Web itself must have been acknowledged by early implementations of search engines as the seminal work of Google's Sergey Brin and Larry Page show them using a distributed system of page-fetching processes and a central database for coordinating the crawl, relying on an algorithm called PageRank to assess the probability that an URL will be visited by a real user and time the next crawls accordingly[3].

Having any single point of failure negates the benefits of other well-made aspects of the system, so naturally, major data structures were partitioned amongst nodes in Mercator and C-proc[4]. IBM's WebFountain was fully distributed and deployed MPI for message passing, achieving high freshness rates.[5] The next breakthrough in scalability came in the form of distributed hash tables to check discovered URLs for their presence in the database quickly; the master-slave dichotomy made way to P2P networks of crawlers as is the case in Polybot and UbiCrawler[6]. In recent years,

the idea of hosting the database remotely was implemented as per[7] which uses a combination of MapReduce programming and Azure pay-per-use hosting allowing companies interested in crawling to pay for resources proportionally.
The challenge remains to effectively perform crawling on RIAs (Rich Internet Applications) where going from one page to another using hyperlinks is replaced by users changing the DOM structure of the page by triggering user events, and redirection to another page is just an event instance. One such attempt exists[8] which describes multiple crawlers running on separate computers updating the state graph partially and informing each other when a new state has been found. On the other hand, Crawljax[9] uses multithreading for speeding up event-based crawling of an individual URL application. The crawling process starts with a single thread which initiates new ones when a state with more than one event is found, continuing the search from there.

3 Solution Architecture

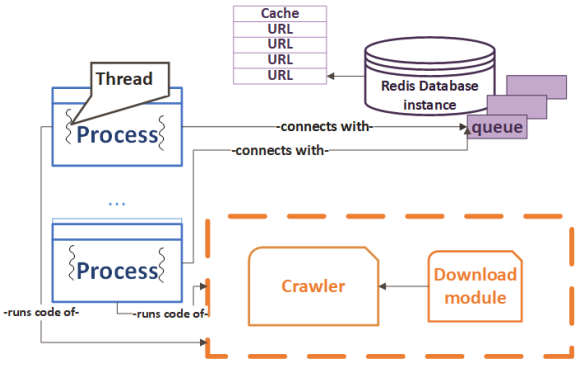

Figure 1: Block diagram representation

The programming language chosen for development is Python as it has multitude of simple-to-use libraries that could be of use in the process and an already-established community around web crawling.
The crawler itself would consist of a download class which would download links from a queue, keeping a separate cache to check if they have been visited before, check if the last visit completed successfully (and if not, retry the failed requests only a couple of times before giving up -as some errors such as HTTP request errors with a 4XX code are unresolvable -) or perform the initial visit keeping in mind to have a time gap between requests so as to be fair to the server and avoid having the IP address blocked (a concern when addressing the same server multiple times in a short interval). The downloading module would use proxies as an additional safety precaution. If there's an error-free cache hit, the cached result is used, otherwise the downloaded page is added to the cache and used. An additional depth value is defined to prevent spider traps (dynamically generated pagination tables and calendars that have many links keep the crawler stuck investigating links that have no intrinsic information). If a maximum depth number of links were followed to get to the current one, it is assumed that the crawl has entered such a trap and further inquiry into its own links is halted. A good practice is to check for the existence of a robots.txt file when requesting a page- crawlers need not conform to the guidelines outlined for them in it, but avoiding pages where crawlers (or some subtypes) are persona non grata is a respectful and risk-free way to go about things. If a page has no such rerstrictions, it is assumed that crawling is permissible.
In anticipation of large crawls, the cache is an instance of a Redis- a NoSQL database which uses a key-value dictionary and is able to store complex data structures. Additionally, it lends itself to easier scaling compared to traditional relational databases, being able to run on the cloud or a cluster of servers.
The rationale behind incorporating parallelism in a web crawler lies in the waiting time incurred when a sequential program has to wait for the page to be received from a server. Using multiple threads or processes means that while one thread/process is waiting for the page to be received and downloaded, a context switch to a free one will avoid wasting CPU cycles. In a multithreaded crawler, one can set the maximum number of threads and within that limit continue to create threads if the link queue is not empty. To make matters easier, if a thread happens to check the queue and it is empty, it is immediately terminated. Should the remaining threads add newly discovered links to the queue in the meantime, provided that the maximum number of threads is not reached, a new thread will be dispatched. In the event that all threads are processed, they will be set to sleep so that CPU execution can be focused elsewhere.
A multithreaded module could be expanded to support multiprocessing as well since one can make use of Redis to store the link queue in the database instead of the local memory so other processes can contribute to the same crawl. The queue would take the form a list, but also use a helper set structure to check for link uniqueness as well as if it's visited already (duplicates are a hindrance both in the queue as well as in the database entries; luckily, such issues can be easily solved or are a non-issue in the latter case as Redis's dictionary does not support duplicates). A hash table is used to map the URL of the link to the predetermined depth previously spoken about.
After these preparations, the crawler can be called from multiple processes, the number of which can be passed

as an argument or take the number of cores on the computer. All the processes are started and their termination is waited on. An even bigger performance boost could potentially be seen if the crawl is distributed across servers all pointing to the same Redis instance. Such an idea is only hampered by the limited computing resources that different providers of platforms as a service give to free tiers; that is to say, none of them provide the ability to use more than one server or virtual machine. However, the Heroku cloud platform will be used to test out the multithreaded, multiprocessing crawler on more cores than the home computer allows since it has Redis support (an instance of which will be created on the Redis Cloud free tier from Redis Ltd.).

The aim of the crawler is not to scrape data per se, but an illustrative example will be made using the BeautifulSoup library for parsing through HTML using a CSS selector-like logic. After getting the whole document tree, using the specific relationships between elements in the hierarchy, or examining the attributes of an element will return a list of matches whose text-content contains the scraped information needed. The results are then exported in a .txt file and analysed for correctness.

A proof of concept will be made by attempting to crawl through some of Goodreads's many lists of books as it's a crawl-friendly website, it provides a robots.txt file for guidance and has a structured representation of data which could be easily reused in the scraping phase from one list to another.

4 Results

The initial link sent for all test executions will be the first page of Goodreads's list 'Books That Everyone Should Read At Least Once', which comes at about 100 pages that need to be downloaded. For a link to be added to the queue, it has to match the regex expression of HTML anchor tags which have a .bookTitle class associated with them and an href attribute which contains a path in the form of /book/show/(some numbers).(book title) and no query/fragment parts after that. This is in order to only download pages containing information about books (for example, `https://www.goodreads.com/book/show/5130.Island`).

Because Redis Ltd. does not officially support Windows, using the Windows subsystem for Linux, the cache and queue are set up through Kali distro.

First, a sequential version of the crawler is tested in order to gauge the baseline download speed, measuring the time taken from the start to the end of the execution. Afterwards, multithreaded-only runs are executed for different thread counts. Finally, multithreaded and multiprocessed crawls are assessed for performance, utilizing an Intel Core i5-2500 with 4 cores and 4 threads per core and a 50 Mbps download speed. The runs are made in one sitting in order to eliminate the possibility of a variable load network slowing certain runs if made at different times of day and with as few background processes as possible.

Script	No. threads	No. proc.	Time(s)	Compare	Errors
Sequential	1	1	315.380	1	N
Threaded	2	1	314.404	1.003	N
Threaded	4	1	320.210	0.985	Y
Threaded	8	1	308.160	1.023	N
Threaded	16	1	318.312	0.990	Y
Process	1	2	161.509	1.953	N
Process	2	2	164.559	1.917	N
Process	4	2	171.036	1.848	N
Process	1	4	194.060	1.625	Y
Process	2	4	221.042	1.427	Y
Process	4	4	193.521	1.630	Y

Table 1: Execution times comparison (local)

The obtained results are an excellent example of the dichotomy between theoretical expectations and practical implementation in code. Namely, threaded runs show an equivalence in performance to the sequential run. This could very well be blamed on the Python Global Interpreter Lock (GIL)- a preventative technique allowing only one thread to execute Python code at a time. Therefore, when multiple threads are dispatched in a core, they quickly switch between each other, and since there are I/O intensive operations taking about 2-3 seconds to finish, the execution is reduced to threads waiting in a queue to download one link. So, additional software overhead is incurred for little benefit.

Even though multiple cores can't be used to run threads in parallel, using multiple processes is acceptable as there is one GIL per process, so process-level parallelism is fully achievable. As results show, some configurations get to nearly two times speedup, with an average factor from multiprocessing runs of x1.733. All of the errors encountered are in the form `'HTTPSConnectionPool(host='www.goodreads.com',port=443):Readtimedout.(readtimeout=60)'` (with 60 seconds being the timeout that was set on the socket manually), but one such exception per run was detected (in the case of two threads and four processes, three timeouts were experienced). This indicates a delay on the server's side which is yet another indeterminate variable at play and can't be regulated. Not having any 'unreachable network' errors indicates an optimal bandwidth.

Many of the libraries used in the project such as `urllib` and `socket` for handling network requests,

`re` for regular expression matching, `zlib` used for compressing the HTML of the pages before inserting them in the cache are C extensions and as such those code sections can run in parallel with one active Python thread, so pure Python code sections are not in the majority. The crawl is I/O-bound too, so CPU tasks are kept minimal.

After having downloaded the pages and their HTML code accordingly, they are taken in a for loop from the cache, a BeautifulSoup parser is initialized and the page is scraped for the book's title, the author's name and the book's overall rating. This is made easy by Goodreads' excessive use of id attributes, so the scrape is very targeted and returns a result with certainty. All the information is then written to a text file. The output is as shown:

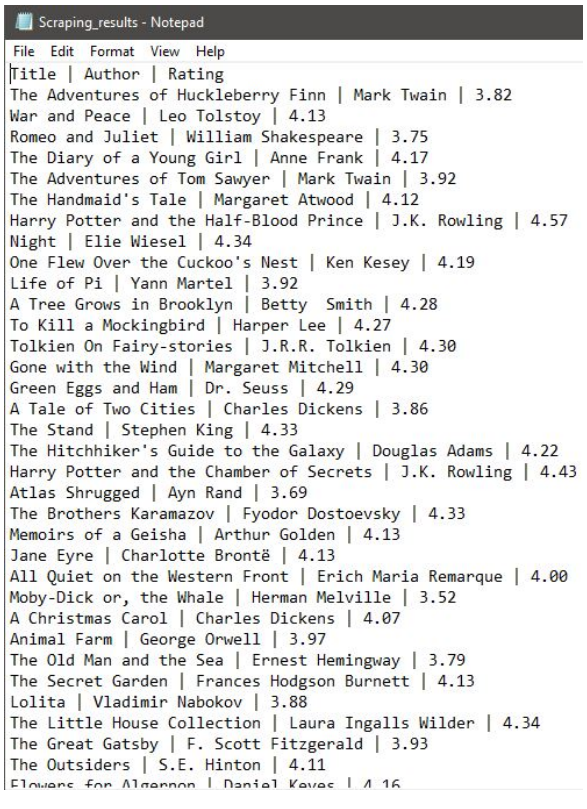

Figure 2: Scraped data from the downloaded pages

Faced with less than thrilling performance boosts, a Heroku account is made with the intention of running the threaded, multiprocessing crawler on a remote server because they have more bandwidth, more computing power and less background processes. Since only the crawler is deployed on Heroku, it does not share the cores with other apps and utilizes 4 physical cores and 4 hyperthreads, making the virtual CPU count equal to 8 with a 512MB RAM. This app container specification, a 'dyno' in Heroku's parlance, can only be instantiated once for free tier accounts, so the testing involves changing the Procfile with a certain configuration (for example, `worker: python Threaded_multiprocessing_crawler.py 4 16`), committing the change and pushing it to Heroku through Git for each test.

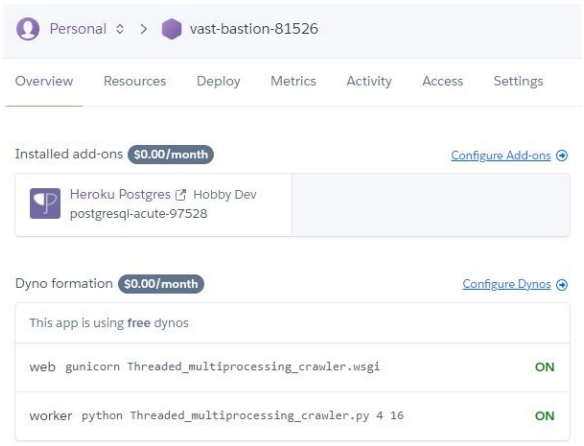

Figure 3: User interface showing the crawler running on Heroku

A separate Redis database instance on Redis Cloud allows for 30 MB of storage with maximum of 30 connections. Since the crawl is not large, the capacity usage stays well below the threshold.

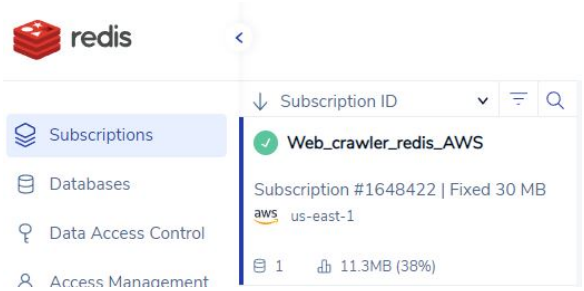

Figure 4: User interface showing the Redis database on AWS cloud services

Knowing the pitfalls encountered during local runs, the configurations chosen to be tested are ones in which the thread count is lower while having the process count vary on a greater range. Some entries in the Errors column have an asterisk to them indicating that while no error messages come directly from the server (usually, not getting a page from it will result with pasted HTML in the console log with a header tag containing "The latest request to the Goodreads servers took too long to respond. We have been notified of the issue and are looking into it"), the worker dyno informs of a download error and a random-length HTML comment is pasted afterwards. The remark about all the runs being made in one sitting holds in this case as well.

Script	No. threads	No. proc.	Time(s)	Compare	Errors
Process	2	2	186.784	1.688	Y
Process	1	4	65.426	4.820	N*
Process	2	4	90.619	3.480	N*
Process	1	8	191.520	1.647	Y
Process	2	8	99.054	3.184	N
Process	4	8	44.973	7.013	N
Process	1	16	81.089	3.889	N*
Process	4	16	76.041	4.147	N
Process	8	16	149.897	2.104	N
Process	1	32	94.199	3.348	N
Process	8	32	76.661	4.114	N
Process	16	32	112.215	2.810	N

Table 2: Execution times comparison (Heroku)

Pleasingly, remote server executions indeed give the source code justice, with one run displaying x7 speedup factor and an average speedup of x2.984 across all runs. Seemingly, the upper limit of the speedup must lie somewhere in here because adding on more processes does not seem to nudge the execution times in a significant fashion, making the logarithmic curve performance measuring adheres to apparent. Additionally, the warning often given towards increasing the number of threads (more does not mean better) holds true, seeing performance decreases as more threads running less effectively are added (the logs even show loads from the cache, which means some threads are idle and access the queue for URLs which another thread has just finished downloading).
In all fairness, if the crawl was run at larger depths with a much larger queue many of the configurations above would shine through, however having limited storage quells such ambitions.
One final note: all the runs made of the crawler point to another variable that can't be regulated- OS scheduling. Some runs are doomed from the start and appear to run almost sequentially depending on when (or if) all threads receive the starting URL. Some runs show the threads synchronizing their execution, so downloads of the same URL may be sprinkled throughout the logs before some small server delay drifts them apart. This means that the time taken for a run of a particular configuration might be in the interval of ± 20 seconds from the one shown. Special care was given towards taking into consideration only the runs that did not emulate sequential downloading.

5 Conclusion and future work

It is doubtful that utilizing the same configurations, but with more processes/threads would lead to more than a marginal speed-up worth the extra processing power. The fact of the matter is that connecting to the remote database takes time as does getting a server response, finding all the links, scraping the results- it all must take up some interval per requested page which can't be further diminished and which adds up from the hundred or so links followed to some fixed chunk of time, giving a lower bound to the whole endeavor. What can be done however, is distributing the crawl across multiple physically separate servers, not just leveraging one server with multiple cores. That ambition must be paid for as previously mentioned, but disregarding that for a moment, there is no issue that would hinder lowering the time taken to some 10-20 seconds or so. The true benefit could be glimpsed with larger crawls, ones that aim to scrape whole websites- distributed crawling is precisely the method used by Googlebot to index the Web.
Now, crawls are not automated in the same way that Google's are, but the structure of the code reasonably permits adding a couple of features concerning sitemap checks, an algorithm dictating how often a page should be crawled through, including handlers for dead links/ outdated cache information and one is well under way towards a sophisticated crawler able to be successfully used in all sorts of contexts.

6 References

[1] Lawrence, S. and Giles, C.L., 2000. Accessibility of information on the web. intelligence, 11(1), pp.32-39.
[2] Edwards, J., McCurley, K. and Tomlin, J., 2001, April. An adaptive model for optimizing performance of an incremental web crawler. In Proceedings of the 10th international conference on World Wide Web (pp. 106-113).
[3] Brin, S. and Page, L., 1998. The anatomy of a large-scale hypertextual web search engine. Computer networks and ISDN systems, 30(1-7), pp.107-117.
[4] Heydon, A. and Najork, M., 1999. Mercator: A scalable, extensible web crawler. World Wide Web, 2(4), pp.219-229.
[5] Yi, J. and Niblack, W., 2005, April. Sentiment mining in WebFountain. In 21st International Conference on Data Engineering (ICDE'05) (pp. 1073-1083). IEEE.
[6] 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.
[7] Bahrami, M., Singhal, M. and Zhuang, Z., 2015, February. A cloud-based web crawler architecture. In 2015 18th International Conference on Intelligence in Next Generation Networks (pp. 216-223). IEEE.
[8] Mirtaheri, S.M., Zou, D., Bochmann, G.V., Jourdan, G.V. and Onut, I.V., 2013, October. Dist-ria crawler: A distributed crawler for rich internet applications. In 2013 Eighth International Conference on P2P, Parallel, Grid, Cloud and Internet Computing (pp. 105-112). IEEE.
[9] Mesbah, A., Van Deursen, A. and Lenselink, S., 2012. Crawling Ajax-based web applications through dynamic analysis of user interface state changes. ACM Transactions on the Web (TWEB), 6(1), pp.1-30.