Implementation of parallel web crawlers for extracting informations from social networks

Nikola Kanevche
nikola.kanevche@students.finki.ukim.mk
Ss. Cyril and Methodius University
Faculty of Computer Sciences and Engineering
Skopje, North Macedonia

1 Related work

Satinder Bal Gupta [8] in his work from 2012 states that web crawlers are not perfect and defines their issues. When the data we want to extract is big, single-thread web crawlers are not that effective, they are not fast enough [7].

One of the solutions of this problem is parallel BFS (Breadth First Search)[9]. Breadth-first search (BFS) is a fundamental graph primitive frequently used as a building block for many complex graph algorithms. Beamer in his work [1] writes about two parallelizing approaches of this algorithm. The first one is Parallel Top-down BFS and the second one is Parallel Bottom-up BFS. In the work[2], Beamer, Asanovic and Patterson in 2012 propose a hybrid approach that is advantageous for low-diameter graphs, which combines a conventional top-down algorithm along with a novel bottom-up algorithm.

Parallel crawlers architecture are very important part of this problem. Junghoo Cho and Hector Garcia-Molina in their work from 2002 [5] are talking about some parallel crawler architectures. A parallel crawler consists of multiple crawling processes, which are referred to as C-proc's. When all C-proc's run on the same local network and communicate through a high speed interconnect (such as LAN), it is called an intra-site parallel crawler. When C-proc's run at geographically distant locations connected by the Internet (or a wide area network), it is called a distributed crawler. When there is a central coordinator that logically divides the Web into small partitions (using a certain partitioning function) and dynamically assigns each partition to a C-proc for download, it is a dynamic assignment.

Parallel crawlers are ideal for big social networks with a lot of data and information. Duen Horng Chau, Shashank Pandit, Samuel Wang, Christos Faloutsos in their work from 2007 supported by the National Science Foundation under Grants No. IIS-0326322 IIS-0534205 [4] are talking about parallelizing crawlers for social networks with parallelizing via Centralized Queue.

In their work from 2011[3], about crawling Facebook, Salvatore Catanese, Pasquale De Meo, Emilio Ferrara, Giacomo Fiumara and Alessandro Provetti are suggesting that they used parallel codes for faster extracting information, because they were working with a huge amount of data.

Junghoo Cho, Hector Garcia-Molina, Lawrence Page in 1988 have published a work [6] in which are considering the URL ordering as a way to get the most important web pages first. This can be a good way to speed up the crawler as well.

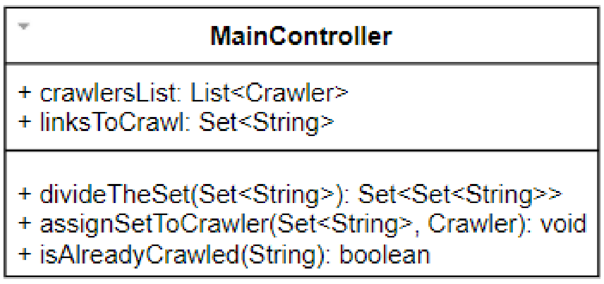
In 2010 Sharma, Gupta and Agarwal in 2010 have published a work where they are describing their architecture for parallel crawler. [10] PARCAHYD is designed to be scalable parallel crawler. This paper has enumerated the major components of the crawler and their algorithmic detail has been provided.

2 Solution architecture

Parallel crawlers can be managed in a lot of different ways, and there is a number of possible solution architectures for problem called "Parallelized web crawling".

In this work we will use the intra-site parallel crawler. In the intra-site parallel crawler architecture every crawler is located on the same machine and the same network. The set of URLs we want to extract information from is divided into subsets and every crawler gets a smaller part of the whole URL set.

As mentioned, target URLs are firstly placed in some data structure (it can be list, set, map or something else, we will discuss that later) and the Main Controller is in charge of creating Crawlers and assigning each of them a piece of whole set. The Main Controller is the kernel of this solution and it controls every Crawler (if some Crawler falls, the Main Controller will get the task to solve that problem). One of the biggest problems of parallel Crawlers is crawling the same URL plenty of times. I.E. When one Crawler finds an URL link in the previously given link (inter-link), the Main Controller should decide whether to crawl that URL, or not (because some URLs can be present in more different web sites). To solve that problem, every crawler, whenever it finds some inter-link, contacts the Main Controller and the MC should check if that URL has been crawled or not (in the Main Controller there is a set of already crawled URLs). When the checking is over, if the link has not been crawled yet, the MC allows the Crawler to do that, but if the link has been previously crawled, MC tells the Crawler not to extract information from that link. With this solution, we avoid the ”multiple times crawled URLs” - problem.


MainController

+ crawlersList: List<Crawler>
+ linksToCrawl: Set<String>

+ divideTheSet(Set<String>): Set<Set<String>>
+ assignSetToCrawler(Set<String>, Crawler): void
+ isAlreadyCrawled(String): boolean

Figure 1: Class for the Main Controller

Because we will divide the original URLs into smaller parts, one solution for the data structure used for storing every URL needed for crawling is a simple set (e.g Java TreeSet). Using a TreeSet we will get every URL ordered alphabetically. After that, the MC will take care of dividing original set into smaller parts and assigning each of them to specific Crawler, using some assigning algorithm.

Assignment can be done in multiple ways. One of them is simply assigning by order number. E.g First Crawler will get the first subset, the second will get second subset, etc. In our solution we will be using the random assignment, the Main Controller will randomly choose one of the Crawlers that still does not have a subset and will assign the current subset to the chosen Crawler.

For that purpose, every Crawler will get a unique number (ID), from 1 to number of Crawlers we are using. Afterwards, the Main Controller will divide the original set into n subsets (where n is the number of Crawlers), and each crawler will get m/n URLs (where m is the number of URLs contained in the original set). The order of crawlers is not important; what is important is that each crawler will get exactly one random subset from the original set.

Number of Crawlers can vary, and it depends on the number of links we want to extract information from. We can implement some function to calculate how many crawlers we need in order to get the optimal solution, but that will be discussed in one of the other sections.

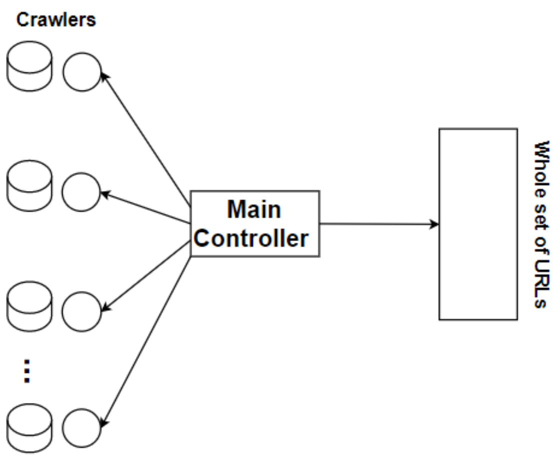
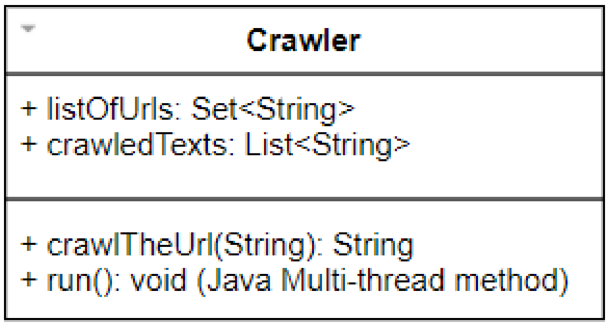


Figure 2: Assigning URLs to crawlers

After dividing and assigning the URLs to each of the Crawlers, every Crawler starts to extract information from its URL, independently of other Crawlers. Using Java Multi-thread, Every Crawler implements the interface Threadable and runs in own thread. With this solution, we get parallelized crawling system. Every Crawler has its own storage to place the extracted information. When all the Crawlers finish their job, the information that they crawled is stored in global Database, and it is ready for further processing (that is not the interest of this work).


Crawler

+ listOfUrls: Set<String>
+ crawledTexts: List<String>

+ crawlTheUrl(String): String
+ run(): void (Java Multi-thread method)

Figure 3: Class for the Crawler

The final architecture consists a whole set of URLs we want to extract information from, Main Controller, Crawlers that are using the Java Multi-thread to speed up the whole process of crawling and a database in which we store the crawled information. Main Controller works the same as showed in Figure 1 and Figure 2. Crawler works the same as showed in Figure 3.

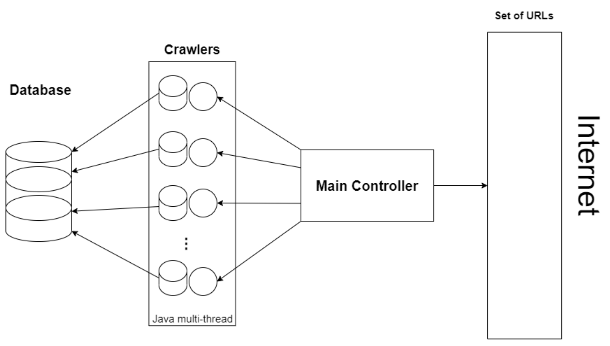


Figure 4: Final Architecture with all components

3 Results

The first testing dataset contains about 3500 links, each of them is from same website. With sequential crawling (1 Thread) the time consumed for finishing this problem is about 20 minutes(1200 seconds).

Because of that, adding more Threads is essential if we want to speed up the whole process.

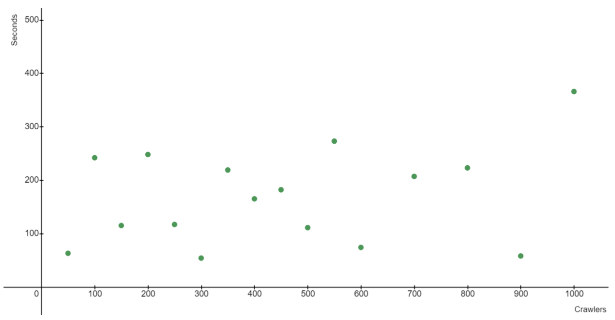
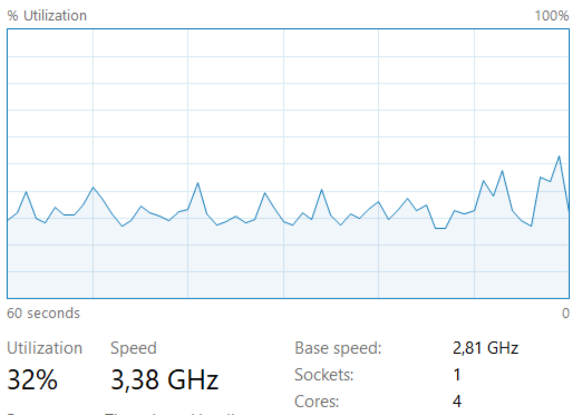


Figure 5: Y seconds needed for X crawlers.

As we can see, with Java multi-thread, the whole process is much faster. The average time for crawling with n crawlers is about 3 minutes which is almost 7 times better than the sequential solution. Also, if we choose the most optimal number of crawlers we can get up to 20 times faster solution.



Utilization 32%
Speed 3,38 GHz
Base speed: 2,81 GHz
Sockets: 1
Cores: 4

Figure 6: CPU Utilization using single thread

In terms of CPU usage, again we can see a huge difference between sequential and parallel crawling. When using 1 thread, in average, only about 1/3 of CPU is used.

On the other side, using multi thread, much more capacity of the CPU is used, in average about 90% of CPU is used with more threads.

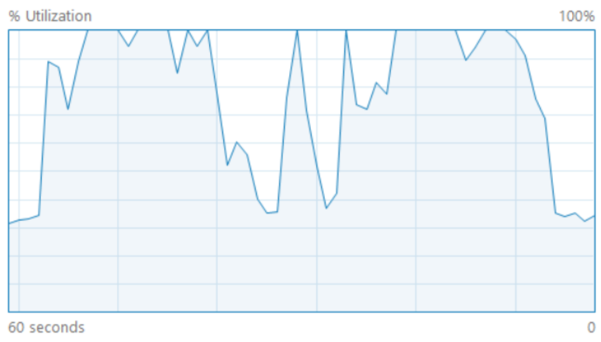


Figure 7: CPU Utilization using multi thread

The second testing dataset contains about 7000 links, from 2 separate websites. Sequential solution for this number of links in this case threw an Out of memory error after 1 and a half hour of working. It was consuming about 93% of RAM capacity.

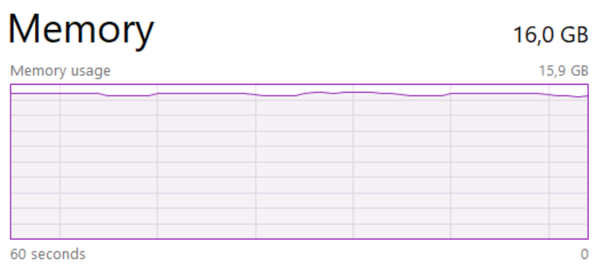
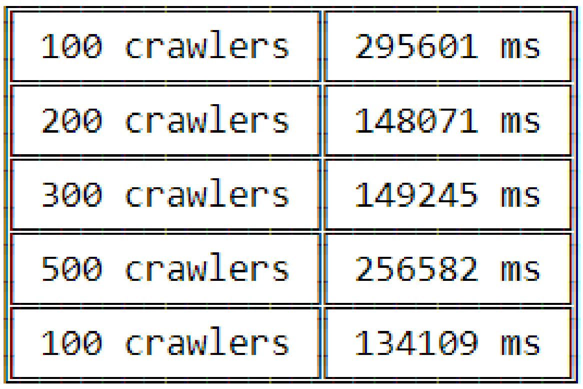


Figure 8: RAM Utilization using single thread

With more threads, the solution is more optimal and in average it finishes after about 6-7 minutes.



100 crawlers	295601 ms
200 crawlers	148071 ms
300 crawlers	149245 ms
500 crawlers	256582 ms
100 crawlers	134109 ms

Figure 9: Table about number of crawlers with their needed time to finish the whole task.

Multi thread solution (100 threads), uses about 70-80% of the RAM.

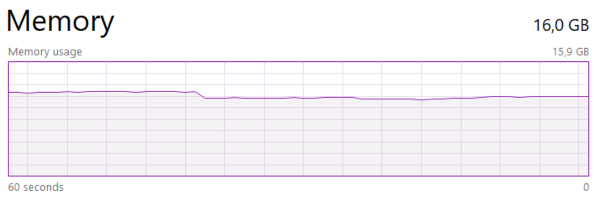


Figure 10: RAM Utilization using multi thread

As we can see, there is a huge improvement of the sequential solution, with parallelizing it. The optimization is more than 1000%.

References

[1] S. Beamer et al. "Distributed Memory Breadth-First Search Revisited: Enabling Bottom-Up Search". In: *2013 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum*. 2013, pp. 1618–1627. DOI: 10.1109/IPDPSW.2013.159.

[2] Scott Beamer, Krste Asanovic, and David Patterson. "Direction-optimizing breadth-first search". In: *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE. 2012, pp. 1–10.

[3] Salvatore A Catanese et al. "Crawling facebook for social network analysis purposes". In: *Proceedings of the international conference on web intelligence, mining and semantics*. 2011, pp. 1–8.

[4] Duen Horng Chau et al. "Parallel crawling for online social networks". In: *Proceedings of the 16th international conference on World Wide Web*. 2007, pp. 1283–1284.

[5] Junghoo Cho and Hector Garcia-Molina. "Parallel crawlers". In: *Proceedings of the 11th international conference on World Wide Web*. 2002, pp. 124–135.

[6] Junghoo Cho, Hector Garcia-Molina, and Lawrence Page. "Efficient crawling through URL ordering". In: (1998).

[7] Keyur Desai et al. "Web Crawler: Review of Different Types of Web Crawler, Its Issues, Applications and Research Opportunities". In: *International Journal of Advanced Research in Computer Science* 8.3 (2017).

[8] Satinder Bal Gupta. "The issues and challenges with the web crawlers". In: *International Journal of Information Technology & Systems* 1.1 (2012), pp. 1–10.

[9] Sungpack Hong, Tayo Oguntebi, and Kunle Olukotun. "Efficient parallel graph exploration on multi-core CPU and GPU". In: *2011 International Conference on Parallel Architectures and Compilation Techniques*. IEEE. 2011, pp. 78–88.

[10] AK Sharma, JP Gupta, and DP Agarwal. "Parcahyd: an architecture of a parallel crawler based on augmented hypertext documents". In: *International Journal of Advancements in Technology* 1.2 (2010), pp. 270–283.