Project report: Parallel Crawler

Evdokia Gneusheva and Vilius Tubinas

9 June 2023

Contents

1 Introduction

In our project, we implement a multithreaded crawler using libcurl library. The crawler follows webpage links via multiple threads, each of which visits links, downloads the webpages, and extracts the links from gathered data. The parsed webpages are represented by the Webpage structure. Those Webpage structures are stored in our hash table StripedHashSet.

The concurrency of the project is twofold - firstly, multiple threads are used to download webpage data, parse the links and store them. Secondly, each thread uses the **multi_curl** library to download multiple links simultaneously. To be more prcise, **multi_curl** takes multiple links which have to be visited and then visits them concurrently using a single thread. This is achieved by processing data of some links, while waiting for others to answer on the web.

The project **(https://github.com/evdokia0/Parallel-Crawler)** contains 6 files:

- **hash_table.h**: This file contains a thread-safe data structure called StripedHashSet for storing and managing Webpage objects. It is based on a textbook example as suggested in the exercise prompt and it utilises buckets to layout the objects.
- **crawler.h**: This file contains the implementation of our multithreaded web crawler that retrieves webpages using libcurl, parses them for URLs, and stores the pages' information in StripedHashSet. It also contains the CompareFirst, EqualFirst and WriteFunctionData struct.
- **webpage.h**: This file defines a Webpage struct that represents a webpage.
- **main.cpp:** This main function launches a web crawler on a specified set of initial URLs, storing the crawled webpages in StripedHashSet, and then outputs all crawled webpages, the total count of unique webpages, and the time taken for the entire process. Main function passes arguments to the crawler to modify its behavior (set number of threads, number of levels, etc.)
- **tests.h**: The tests file contains some basic tests that were use to check the functionality of the hash set.

In Section 2, we describe the data structures we use. We start by explaining the Webpage struct implemented in the file **webpage.h**, the StripedHashSet class implemented in the file **hash_table.h** and CompareFirst, EqualFirst and WriteFunctionData struct implemented in the very beginning of the file **crawler.h**.

In Section 3, we provide a detailed explanation of how we implemented the parallel crawler, a fundamental component of our project. Here, we will only talk about the file **crawler.h**.

In Section 4, we analyze the performance of the crawler based on the number of threads and other parameters, as well as the time spent on different parts of the code.

2 Data Structures

2.1 Webpage

The **Webpage struct** serves as a basic entity for the Parallel Crawler. Each Webpage object represents a webpage that has been or will be visited by the crawler, storing its own URL **'url'** (string), its parent URL **'parent'** (string), external links **'externalLinks'** (set of strings) to explore further, and its current level in the crawling process **'current_level'** (integer) to keep track of the exploration depth.

2.2 StripedHashSet

The hash set **StripedHashSet** [2] stores Webpage objects. It provides concurrent and thread-safe operations on these Webpage objects. For every bucket operation, it locks the corresponding mutex in the **'locks'** vector, ensuring that only one thread can access a given bucket at any given time, allowing multiple threads to operate on the same **StripedHashSet** without "colliding".

StripedHashSet class

- **Data members:** It consists of two private vector data members - **'table'** and **'locks'**. The **'table'** is essentially a hash table, where each item is a list of items of generic type 'T', called a bucket. The **'locks'** vector is used for synchronization, holding mutexes corresponding to the buckets in the 'table'. The constructor initializes the size of the **'table'** and **'locks'** vectors to a specified **capacity**.
- **Contains Method:** This method checks if a given item is present in the hashset.
- **Add Method:** This method is used to add an item to the hashset. It locks the corresponding bucket, checks if the item is already present, and if not, it adds the item to the end of the list. Tthis method re-sizse the table when the number of elements in a given bucket exceeds the total number of buckets. This is done to keep the table as square as possible in order to reduce computational time of other operations.
- **Remove Method:** This method removes a given item from the hashset. It locks the corresponding bucket, finds the item in the list and removes it if found.
- **Add_links Method:** This method adds a set of external links to a given webpage if it exists in the hashset.
- **Resize Method:** This method resizes the hashset by doubling its capacity. It locks all the buckets, rehashes all the items in the old table, and inserts them into a new table of double the size. It also doubles the number of locks.
- **Get_bucket Method:** This method returns a list of items that belong to the same hash bucket as the given item.
- **Get_count Method:** This method returns the total count of all items in the hashset.
- **Get Method:** This method returns an element from the hash table, based on a given url. If such object does not exists, it returns a null pointer.

- **Print_all Method:** This method outputs the contents of the entire hash table to the console, specifying which bucket each element belongs to.

Global Operator
The operator "==" defines the comparison operation of two webpage objects based on their URLs.

2.3 (Helper) CompareFirst and EqualFirst

We implemented the **CompareFirst stuct** and the **EqualFirst stuct** so that **unordered_set** can hash elements, check if two pairs are equal and order them based only on the first member of the pair, instead of both members. These structures are necessary, since during the process of crawling each link is stored in a pair with their parent url, but parent url should not be taken into account when checking if that link has already been visited.

- **CompareFirst:** This structure provides a way to hash pairs by hashing only the first value in the pair.
- **EqualFirst:** This structure provides a way to compare pairs by only comparing the first value in the pair.

2.4 (Helper) WriteFunctionData

We implemented the **WriteFunctionData struct** in order to be able to pass multiple variables to the write function through a single pointer, as required by the **libcurl library**.

This struct contains five members:

- **url:** The URL of the webpage that is currently being crawled.
- **parent:** The parent URL from which this URL was extracted.
- **HashSet:** A pointer to a StripedHashSet object that stores all the Webpage structures.
- **current_level:** The current depth level of the webpage being crawled.
- **only_local:** A boolean flag indicating whether the crawler should only crawl links within the same domain as the initial URLs (true), or whether it should crawl any discovered link (false).
- **new_urls:** A pointer to a set storing all the new URLs that are extracted from the current webpage.

3 Parallel Crawler

3.1 Overview

In this part we are going to talk about our implementation of the parallel crawler itself. The crawler works in a batch-by-batch fashion, each time taking a specified number of links from the link pool, distributing them to the desired number of threads, executing these threads, and adding the newly found found links to the pool. In other words, it operates in **multiple threads**, each working on a subset of URLs. The URLs to be crawled are stored in **std::unordered_set** data structures, each pair consisting of a URL and its parent URL. The crawler uses the **libcurl** library [1] to make HTTP requests and fetch webpage data. The **libcurl's 'multi' interface**, used within each thread, speeds up the process by allowing a thread to explore webpages while waiting on other webpages to load. The crawler parses the webpages to find all the links, which are then stored and further crawled in the next iteration. We store the links as **Webpage objects** in our **StripedHashSet**. We make sure to prevent re-crawling the same URL (by using sets as well as the **contain** method of **StripedHashSet** and to keep track of the **crawling level depth**. We also created a set of **parameters** to customize crawler's behavior such as maximum number of links per thread and etc.

Here's a step-by-step overview of how the Parallel Crawler works:

1. **Initialization:** The program starts with the **crawl_webpage** function, which takes as input a set of URLs to be crawled, the number of URLs, maximum wait time for network operations, a hash set to store the crawled webpages, and some optional parameters to customize the crawler's behavior. The function also initializes several unordered sets to keep track of URLs that are currently being checked and are newly discovered.

2. **Link extraction and web page crawling:** The core of the web crawling happens inside a while loop. For each iteration, the current URLs are divided across multiple threads using **std::thread**. The links are given to threads in an arbitrary order, but the number of links each thread should take is determined by:

$$N_{links} = \max(N_{links}/N_{threads}, N_{max\_links})$$

   where N_{links} is the number of links currently in the link pool, $N_{threads}$ is the number of threads (pre-set by the user) and N_{max_links} is the maximum number of links allowed per thread (also pre-set by the user).

   Each thread calls the **process_links** function, which uses the **libcurl** library to send HTTP requests to each URL. If the number of total URLs in the pool is larger than all of the threads can take, the URLs are either left for the next while loop iteration or disregarded (this is decided by flag **disregard_leftovers**).

3. **Data processing:** The received HTML is processed by the **cb** (callback) function, which is called by **libcurl** every time data is received from a server. This function extracts all links from the received HTML using regular expressions, unifies their format (e.g. makes relative links non-relative by adding the root domain), and adds these links to the set of new URLs.

4. **Page creation:** The **cb** function also creates a Webpage object for each URL (that is currently being explored), storing the URL, the set of links that have been found, the current crawl depth, and the URL of the parent page. The Webpage object is then added to the StrippedHashSet. If such object has already been added to the StrippedHashSet, it adds new links to the existing Webpage object (this can happen when webpages have a lot of content and cb is called multiple times for one URL).

5. **Thread synchronization:** Once all threads have finished processing their URLs, they are joined back into the main thread. The new URLs discovered by all threads are merged together, and the process repeats.

6. **End condition:** The crawling stops either when there are no more URLs to process, the maximum crawl depth (**max_level**) is reached or the total number of URLs to crawl (kept by **std::atomic<int> counter**) has reached the desired number.

3.2 Libcurl Library and threading

We use the **libcurl** library, which is a library for transferring data with URLs. We also use libcurl's **'multi' interface** which allows the crawler to explore webpages and extract links from them while waiting on other webpages to load. The entire process of crawling is done concurrently using multiple threads.

- **init Function:** This function initializes a **curl easy handle** with various options, including the URL to fetch, the function to call when data is received, and the user data to pass to that function. It is a convenient way to add all of the needed flags to the **curl easy handle** to configure the scraping process. The function adds this handle to a **curl multi handle**, which allows multiple **easy handles** (i.e., multiple HTTP requests) to be managed at once.

- **process_links():** This function is the main workload of each thread. It initializes a **curl multi handle** and adds all the URLs it is responsible for to this handle. Then, it repeatedly calls **curl_multi_perform()** and **curl_multi_wait()** until all HTTP requests are complete.

- **cb Function:** This is the callback function that is called when data is received from a curl easy handle. It extracts all links from the received data, and adds them to the new URLs set. It also updates the HashSet with the new webpage data. The methods for link extraction will be addressed in the upcoming section.

- **crawl_webpage Function:** This is the main function of the web crawler. It works by creating multiple threads and distributing the initial set of URLs across these threads. Each thread calls **process_links()** on its set of URLs, which fetches each URL and extracts the links from its content. The function then takes the new URLs discovered by each thread, combines them into a new set, and starts a new round of multi-threaded processing. This process repeats until the maximum depth level or other criterion specified before has been reached.

3.3 Link Extraction

Once the data is obtained, we need to extract all the links from it, and here's how we do it.

- **extract_links Function:** This is the main function. It extracts all the links present on a webpage. It returns a set of new links that have not been checked yet. It checks if URLs are absolute or relative, and whether the URLs belong to the same domain as the root URL, if the **only_local** flag is true. It also avoids already checked URLs and unifies the format of found URLs.
- **extract_domain Function:** This is a helper function that extracts the domain from the URL string using regular expressions.
- **compare_domains Function:** This is a helper function that compares two domains.

The link extraction mainly happens in the **cb callback function** and the **extract_links** function. In the **cb function**, after the page data is received, **extract_links** is invoked to identify and retrieve all hyperlinks present on the page. After that, these new URLs are inserted into a shared set **new_urls**, which helps keep track of all the URLs that have to be processed. The function also updates the links of the parent webpage in the hash set, creating the link graph of the crawled web.

3.4 Parameters

We added a list of parameters to customize the behaviour of our crawler and analyze it.

- **numthreads:** The number of threads the program should use.
- **max_links_per_thread:** The maximum number of links that each thread can process at once.
- **max_level:** The maximum depth of crawling, where 1 means only the initial URLs are crawled, 2 means their links are crawled as well, and so on.
- **only_local:** A boolean parameter that controls whether to only crawl links within the same domain as the initial URLs or to crawl any discovered link.
- **max_wait:** The maximum time to wait for all threads to complete in seconds.
- **disregard_leftovers:** A boolean parameter to control whether the function should disregard any URLs that were not processed by the time the maximum wait time has passed.

4 Results and Observations

4.1 Analyzing the behavior of our crawler depending on the number of threads involved

Initialization

- In this section, we'll examine the results from testing our web crawler on Wikipedia. Three key factors were changed during the tests: the number of threads used and the quantity of links to crawl.
- For clarity, these are the values used in our tests:
  - Threads: 1, 2, 4, 8, 16, 32
  - Total links: 100, 200, 500 crawled (i.e. 7000, 14000, 35000 discovered links)
- We started the tests with a small number of links due to time constraints.
- **It's important to note that we counted a link as 'crawled' only if we downloaded its HTML, gathered all child links, and added it to our record, or 'HashSet'. In other words, the number of links discovered by the crawler is significantly higher than the number of links crawled. For example, when our crawler "crawls" 100 links, in total it discovers and appends as children around 7000 links. This relation was noticed to be proportional, with 200 links crawled leading to discovery of around 14000 unique links total and 500 links leading to discovery of around 35000.**
- Certain additional parameters remained constant during the tests. The 'only_local' variable was set to true, which ensured that we examined only local links. The 'disregard_leftovers' was set to false, meaning we didn't omit any remaining data. We also controlled the depth of the crawling process using the 'max_level' variable, setting it to a high value of 1000 to make sure it was never reached during our tests.
- All tests ran on an Omen 15 laptop, connected to an Eduroam network. The laptop has the following specifications:
  - Processor: 10th Generation Intel Core i7-10750H, 6 cores, 2.6 GHz base frequency, up to 5 GHz with Intel Turbo Boost Technology.
  - Graphics: NVIDIA GeForce RTX 3070 with 1290MHz Boost Clock and 100W Max Graphics Power. It also includes NVIDIA Max-Q Technologies such as Dynamic Boost 2.0, Resizable BAR.
  - RAM: 16 GB DDR4-2933 SDRAM (2 x 8 GB)
  - Storage: 512 GB Intel SSD
  - Operating System: Windows 10 Home.
- The 'error rate', seen in the tables below, accounts for times when a link didn't respond and couldn't be crawled. In these cases, the crawler marks that link as an error. We've also provided the standard deviation for the times because we've averaged the results of three separate runs. This helps us make a more reliable estimate of how well the crawler performs.

- Lastly, we set a 30-second wait time for the curl library when it tries to get a response from a webpage. If there's no response within 30 seconds, the library flags the webpage as an error. This wait time is likely why we have large standard deviations in some tests.

Results

Threads	Links visited	Error rate	Average time (s)	Standard Deviation	Links per thread
1	100	0	78.7506	22.1764	1
1	200	0.002	155.4150	22.9205	1
1	500	0.001	462.8483	67.7778	1
2	100	0.017	68.1706	39.7567	1
2	200	0.025	91.7683	16.6374	1
2	500	0.039	329.3893	19.5396	1
4	100	0.030	25.1917	1.8763	1
4	200	0.007	54.3329	3.8934	1
4	500	0.038	151.9037	24.6543	1
8	100	0.003	14.6727	0.8044	1
8	200	0.008	32.9747	5.2062	1
8	500	0.029	129.4574	29.7209	1
16	100	0.030	9.9518	1.4952	1
16	200	-0.002	20.0925	2.2222	1
16	500	0.015	77.4863	6.5872	1
32	100	-0.010	17.8489	17.9890	1
32	200	0.002	26.8035	14.8027	1
32	500	0.015	73.9640	65.5573	1

Table 1: Computational time for different number of threads and links (1 link per thread)
Each time is an average of 3 runs

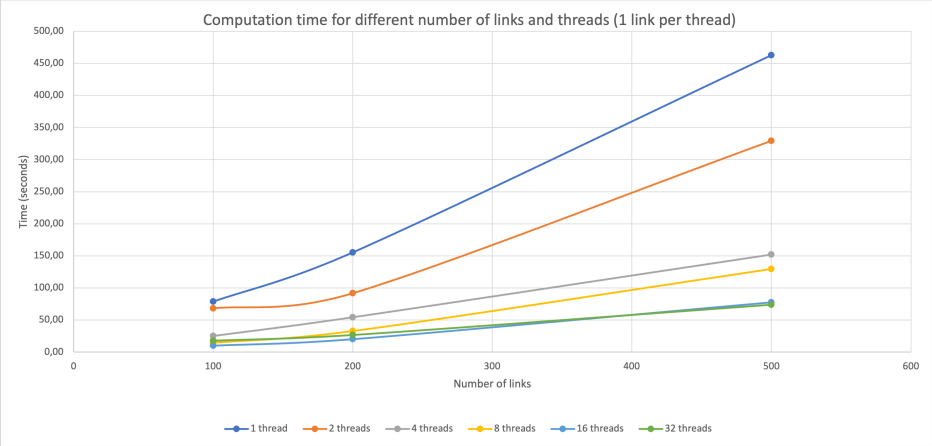

Figure 1: Plot of computational time depending on number of threads (1 link per thread)

Obseravtions

- To begin, the average time taken to crawl the websites reduces as the number of threads increases, across all three quantities of links (7000, 14000, 35000 discovered links). This is in line with the expected behavior, since parallelization using multiple threads generally leads to faster execution. When using 1 thread, the crawling times are the longest. By increasing the number of threads to 16 or 32, we see a notable decrease in the average time taken to crawl the same number of websites.

- Interestingly, this relationship isn't linear, and it seems to decrease gradually as we approach a higher number of threads. For instance, the jump in efficiency from 2 to 4 threads is

significant, but the reduction in time from 16 to 32 threads is minor. In fact, for 100 links, there is a slight increase in time when we go from 16 to 32 threads. We suppose that there may be a limit to the efficiency gains from thread increase, due to factors like the cost of managing more threads or the hardware limits of the computer.

- Analyzing the different total link counts, we notice that the effect of the thread number is not uniform across them. The impact of increasing the number of threads is more prominent with a larger number of links. This makes sense because as the amount of work increases (i.e., more links to crawl), the benefits of parallel processing become more apparent.

- Regarding the error rate, it fluctuates with the change in thread number, though it's not directly proportional or inversely proportional to the number of threads. The error rate is quite random since it appears from web pages not answering the crawler's download request within 30 seconds.

- Analyzing the standard deviation, we notice that it gets significantly lower as the number of threads increases. We believe that most of the variation in time can be attributed to network connection issues - either pages answering after a long wait or not answering at all. As the number of threads is increased, one malfunctioning page has a less severe effect on the overall performance of the crawler and that is why we observe a more stable computational time.

- Considering the above observations, it seems that using 16 threads provides an optimal balance between computational time and resource allocation. It significantly reduces the time to crawl compared to 1 thread but does not add much overhead compared to 32 threads. Specifically, for crawling 200 links (i.e. 14000 discovered links), it took an average of around 20 seconds with a relatively low standard deviation, indicating consistent results.

4.2 Analyzing the behavior of our crawler depending on the number of links per threads

Initialization

Since we have used **multi interface** of libcurl library, it allows one thread to process multiple links concurrently and thus to speed up the crawling process even further (i.e. overall, we use multiple threads + each thread is now able to process multiple links simultaneously). Essentially, as we increase the links per thread, this increases the workload for each individual thread, which means that each thread has more data to process before it can complete. So, it would be interesting to analyze the performance of the crawler based on the number of links processed per thread. We will analyze it for 1, 2, 4, 8, 16, 32 and 64 links per thread for multiple threads.

Results

Threads	Links visited	Error rate	Average time (s)	Standard Deviation	Links per thread
1	100	0.05	82.8277	16.9336	2
1	200	0.037	246.5547	69.5794	2
1	500	0.037	674.7383	44.7547	2
2	100	-0.007	136.6040	12.4774	2
2	200	0.020	341.1293	169.9298	2
2	500	0.033	852.2800	70.3828	2
4	100	0.027	126.3207	14.0417	2
4	200	0.002	251.4180	57.8083	2
4	500	0.019	707.3463	32.9920	2
8	100	-0.007	108.6093	17.6085	2
8	200	0.012	108.2820	28.0336	2
8	500	0.015	501.3080	86.0541	2
16	100	-0.010	65.5780	50.1427	2
16	200	-0.003	161.0120	31.0683	2
16	500	0.013	358.6383	43.6072	2
32	100	-0.010	35.1215	1.0106	2
32	200	0.000	103.7034	20.9698	2
32	500	0.013	227.5483	1.8929	2

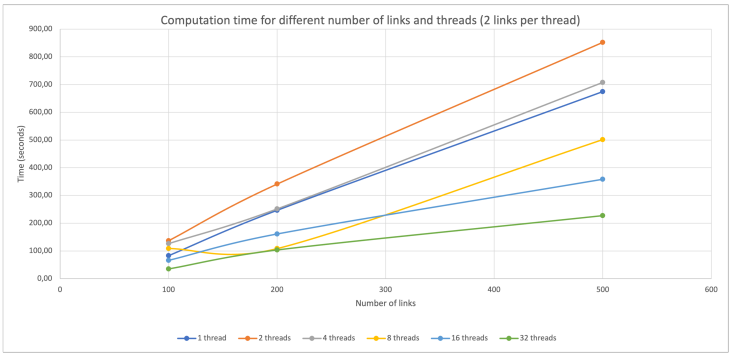

Figure 2: Computation time for different number of threads and links (2 links per thread)
Each time is an average of 3 runs

Threads	Links visited	Error rate	Average time (s)	Standard Deviation	Links per thread
1	100	0.0467	97.4651	17.2422	4
1	200	0.010	164.9810	19.1176	4
1	500	0.022	395.2533	36.7035	4
2	100	0.003	86.3572	35.4936	4
2	200	0.025	184.6927	6.0519	4
2	500	0.023	395.6083	63.4690	4
4	100	-0.007	50.7377	16.2115	4
4	200	-0.002	149.3567	43.0724	4
4	500	0.019	355.5217	33.9988	4
8	100	-0.010	85.3994	16.9600	4
8	200	-0.005	119.9104	45.1830	4
8	500	0.017	272.3013	33.5978	4
16	100	-0.010	24.9853	17.2095	4
16	200	-0.005	69.3599	27.8823	4
16	500	0.012	158.7823	61.0665	4
32	100	-0.010	15.3594	16.4105	4
32	200	-0.005	68.2133	1.9349	4
32	500	0.011	166.4593	52.4752	4

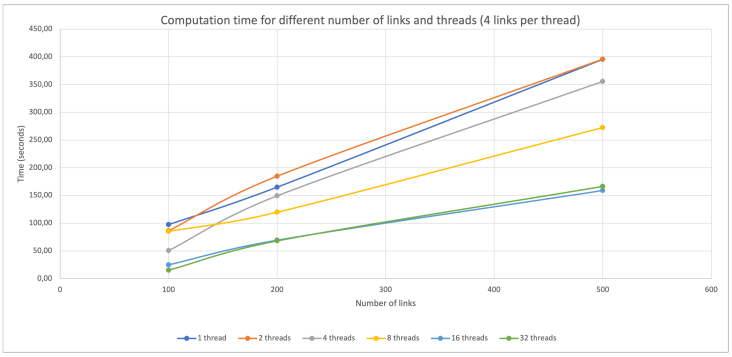

Figure 3: Computational time for different number of threads and links (4 links per thread) Each time is an average of 3 runs

Threads	Links visited	Error rate	Average time (s)	Standard Deviation	Links per thread
1	100	0.0067	46.1137	16.2899	8
1	200	0.0233	127.3338	32.1806	8
1	500	0.0367	174.4447	88.5472	8
2	100	-0.0067	22.8427	17.2034	8
2	200	0.0133	32.8814	16.6623	8
2	500	0.0227	188.8383	60.0329	8
4	100	-0.0067	8.9399	1.6884	8
4	200	-0.0017	66.4497	13.1781	8
4	500	0.0187	143.0784	43.5198	8
8	100	-0.0067	46.8863	16.5215	8
8	200	0.0000	32.3948	17.1534	8
8	500	0.0027	123.5557	64.9618	8
16	100	-0.0167	5.9132	0.2068	8
16	200	0.0000	21.6335	17.1595	8
16	500	0.0007	89.4793	15.3080	8
32	100	-0.0100	6.2966	0.3466	8
32	200	0.0000	50.3820	15.8092	8
32	500	0.0000	89.2837	16.4491	8

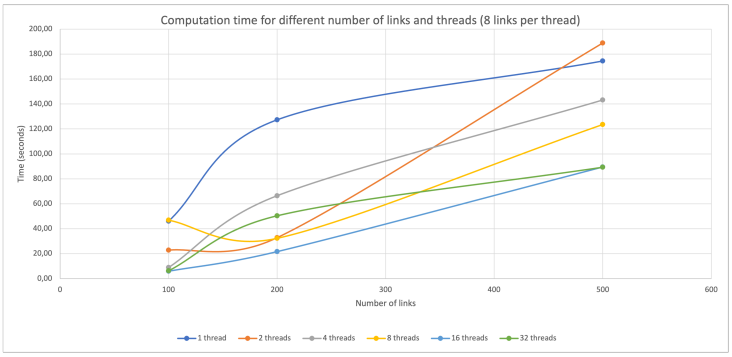

Figure 4: Computational time for different number of threads and links (8 links per thread) Each time is an average of 3 runs

Threads	Links visited	Error rate	Average time (s)	Standard Deviation	Links per thread
1	100	-0.0067	19.5965	0.4117	16
1	200	0.0067	44.2843	15.5068	16
1	500	0.0187	143.2717	37.4826	16
2	100	0.0000	22.5963	14.8652	16
2	200	0.0083	41.5964	15.5053	16
2	500	0.0007	86.8496	30.8461	16
4	100	-0.0100	19.9499	16.6635	16
4	200	-0.0033	46.8380	30.1508	16
4	500	0.0247	134.5876	46.3751	16
8	100	-0.0067	16.1820	16.2984	16
8	200	-0.0017	32.3902	16.8117	16
8	500	0.0047	83.2123	28.9789	16
16	100	-0.0033	6.4966	0.2255	16
16	200	-0.0017	24.3256	16.0478	16
16	500	0.0027	65.9982	17.7226	16
32	100	0.0000	6.4597	0.8461	16
32	200	0.0000	42.0309	1.6108	16
32	500	0.0000	52.8237	1.4693	16

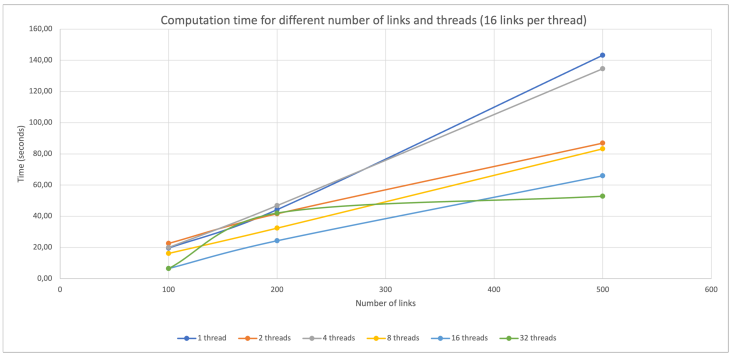

Figure 5: Computational time for different number of threads and links (16 links per thread) Each time is an average of 3 runs

Threads	Links visited	Error rate	Average time (s)	Standard Deviation	Links per thread
1	100	-0,0067	32,6259	15,0986	32
1	200	0,0083	45,4625	18,4258	32
1	500	0,0420	106,7401	23,1550	32
2	100	-0,0100	24,8624	16,6923	32
2	200	0,0000	25,3834	2,9288	32
2	500	0,0220	76,4348	20,5828	32
4	100	0,0000	11,4627	0,9498	32
4	200	0,0017	32,1667	17,2356	32
4	500	0,0027	92,4773	2,4164	32
8	100	0,0000	8,2553	0,7916	32
8	200	0,0000	18,5980	2,1623	32
8	500	0,0000	61,2670	3,3799	32
16	100	0,0000	6,9866	0,2809	32
16	200	0,0000	26,9220	15,0648	32
16	500	0,0000	39,3840	16,9093	32
32	100	0,0000	27,0842	15,3761	32
32	200	0,0000	25,8811	15,9715	32
32	500	0,0000	74,5927	13,4585	32

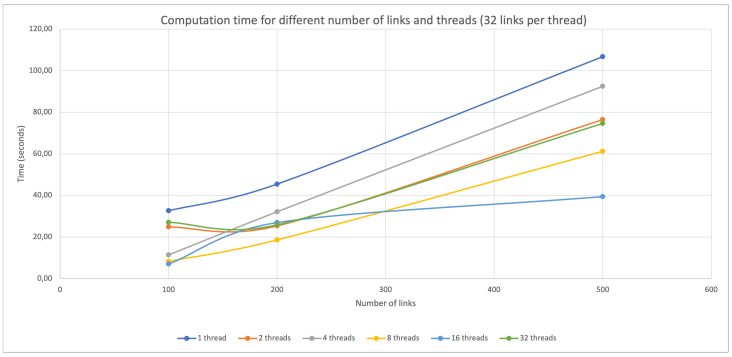

Figure 6: Computational time for different number of threads and links (32 links per thread) Each time is an average of 3 runs

Threads	Links visited	Error rate	Average time (s)	Standard Deviation	Links per thread
1	100	-0.0100	24.4952	1.2387	64
1	200	-0.0017	44.5570	3.9524	64
1	500	0.0053	130.1630	16.2764	64
2	100	0.0000	20.3798	0.3514	64
2	200	0.0000	42.1142	16.6971	64
2	500	0.0060	63.5752	13.4600	64
4	100	-0.0067	14.4997	1.2327	64
4	200	0.0000	25.4617	1.6668	64
4	500	0.0007	53.6311	16.9210	64
8	100	-0.0067	9.7021	0.8641	64
8	200	0.0000	20.9890	2.3730	64
8	500	0.0000	49.2364	18.7294	64
16	100	0.0000	10.1814	1.7703	64
16	200	0.0000	31.5968	16.7576	64
16	500	0.0000	68.9429	9.7975	64
32	100	-0.0033	6.8892	1.2185	64
32	200	0.0000	37.1179	15.8836	64
32	500	0.0000	59.8987	2.8331	64

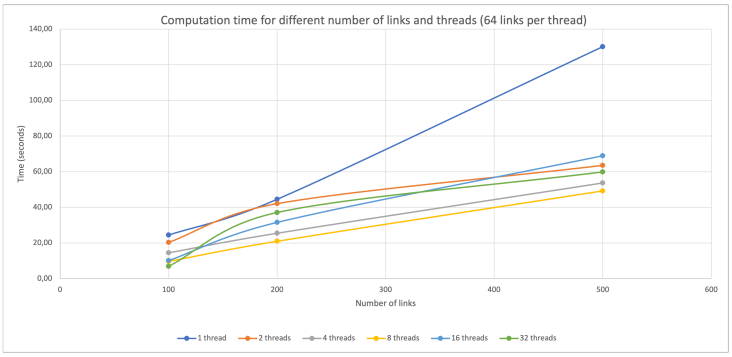

Figure 7: Computational time for different number of threads and links (64 links per thread) Each time is an average of 3 runs

Threads	Links visited	Error rate	Average time (s)	Standard Deviation	Links per thread
1	100	0.0000	23.2084	0.2929	128
1	200	0.0000	49.4128	2.1505	128
1	500	0.0013	99.1209	2.8670	128
2	100	-0.0067	21.3176	3.4874	128
2	200	0.0000	40.8620	16.9445	128
2	500	0.0000	73.0594	18.3151	128
4	100	-0.0067	14.1486	1.0537	128
4	200	0.0000	35.7671	16.8358	128
4	500	0.0000	62.4866	13.5349	128
8	100	-0.0067	9.2496	1.1770	128
8	200	0.0000	29.0440	19.7348	128
8	500	0.0000	46.6068	6.5647	128
16	100	-0.0100	24.6118	15.8402	128
16	200	0.0000	25.0449	15.3519	128
16	500	0.0000	68.6247	31.1195	128
32	100	-0.0033	6.6091	0.7442	128
32	200	0.0000	34.4576	16.5884	128
32	500	0.0000	46.3425	18.0290	128

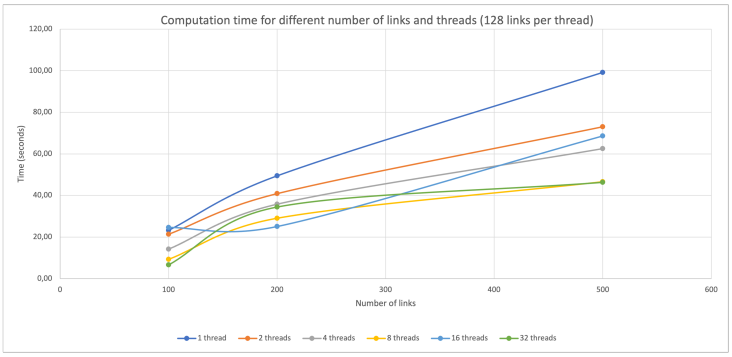

Figure 8: Computational time for different number of threads and links (128 links per thread) Each time is an average of 3 runs

Observations

- Generally, as the links per thread increase, we observe a decreasing trend in the average time. For example, for 8 threads with 500 links, the average time decreases from approximately 501 seconds for 2 links/thread to around 49 seconds for 64 links/thread.
- The error rate seems to be inconsistent as the number of links per thread increases. In some cases, there is a slight increase in error rate, while in others, it remains constant or even decreases slightly. This could be due to factors like the quality and availability of links or the reliability of the network connection. However, the overall error rate doesn't seem to be largely affected by the increase in links per thread, implying the crawler can handle more workload per thread without a significant rise in errors.
- Generally, the standard deviation decreases as the number of links per thread increases, which suggests that increasing the workload per thread leads to more consistent performance.

In summary, increasing the number of links per thread seems to improve the efficiency of the crawler by decreasing the average processing time and maintaining or slightly improving the error rate. It also leads to more consistent performance. This implies that it might be beneficial to increase the links per thread to improve efficiency and consistency.

To conclude, we observe that depending on the number of links visited (7000, 14000, 35000), the optimal number of threads is 16 or 32.

4.3 Computation breakdown

By using the apple profiler, we compiled the crawler (for Wikipedia) with with 16 links per thread and 16 threads and ran it for 500 links (i.e. 35000 discovered links). The test was done on a MacBook Pro which has 4 cores.

The profiler showed the distribution of times spent on each part of the code. The code ran for a total of 5 minutes and 27 seconds. The main function spent 94.9% of time initialising the crawl_webpage function and 3.9% of the time was spent initialising the StrippedHashSet. Upon initialisation each thread showed promising results in their breakdown - curl_multi_perform took up 99.9% of computational time, 10.7% of which was taken up by the cb function, which extracts links and appends elements to the hash table. The most significant amount of time within cb was spent on regex search, which is a library we use for link extraction.

The situation is different when the testing is done on a smaller amount of links. When running the profile for 100 (i.e. 7000 discovered links) links with all of the same parameters kept the same, the cb function takes up around 99,2% of computational time. The breakdown of the cb function is similar as before, with the most significant amount of time being spent on iterating through the HTML to found links.

Upon further inspection and testing we can see that in the computational breakdown curl_multi_perform always takes up around 99% of computational time. The factor which is different when running on small amount of links (or at the very beginning of the run with a large amount of link) is the distribution between waiting for network and processing of the data. When one thread deals with only a few links, the chunk of time, which is used for processing data becomes relatively more significant.

To sum up, the breakdown of computational times has shown promising results. The parts of the program which took up significant portions of time were the HTTPS requests themselves and

analysis of the links. Writting into the stripped hash set and checking if an element is contained in it did not cost a lot computation-wise indicating that our implementation is successful.

References

[1] Official libcurl library.

[2] Maurice Herlihy and Nir Shavit. The art of multiprocessor programming. *Chapter 13*, 2012.