École Polytechnique

Bachelor of Science

Concurrent and Distributed Computing

CSE305 - Project Report

Authors:
Yubo Cai
Junyuan Wang

Supervisor:
Eric Goubault

Teaching Assistant:
Pogudin Gleb
Lafont Ambroise

June 14, 2024

GitHub Repository

Please find our code in the following GitHub repository: link.

1 Introduction

The objective of this project is to develop a parallel web crawler capable of efficiently indexing Wikipedia or a similar multi-page website. The web crawler will systematically visit web pages, collect links on the web pages, and collect statistical data to create an "index" of pages. This process is commonly used by search engines to index web content for easy retrieval.

The project involves two major components:

1. **Multithreaded Crawler**: This component will utilize multiple threads to visit web pages concurrently. Each thread will fetch a URL from the queue, download a webpage, extract all the links, and add them to a queue for further crawling. The implementation will use libcurl for handling HTTP requests and data extraction.
2. **Concurrent Data Structure**: To manage the list of already visited pages and ensure thread safety, a concurrent data structure will be used. A simple Hash-set and a Striped Hash-set are implemented to allow multiple threads to access and update the data structure simultaneously. This data structure serves as a cache and storage for indexing URLs.

Additionally, we analyze the performance of the crawler under different circumstances and collect data on bottlenecks.

The following sections will detail the deployment, design, implementation, and analysis of the parallel web crawler.

2 Deployment

Environment Requirements:
This project is only tested on the Ubuntu X86 system. However, we included all the source code of external libraries, "Libcurl" and "Cpplog", allowing for compilation on other platforms. **CMake**, **g++**, and **OpenSSL** are **required** to be installed on the machine.

Compilation & Usage Instructions:

1. Creating a build folder under the root directory of the project and entering this folder.

```
mkdir build && cd build
```

2. Running the following command to compile the project:

```
cmake .. && make
```

3. If there is no compilation error, the executable files should be located in the **build** folder named **ParallelCrawler**. Start this program by simply running the following command and passing the parameters. (Be sure to have the right permission)

```
./ParallelCrawler --url=https://example.com
```

Parameters:

- `--url`: Passing the URL you want to crawl. This parameter is **necessary**.
- `--threads`: The number of threads of crawler. Optional. The default is 8 threads.
- `--output`: Path to the output index file. Optional. The default is "found_urls.txt" under the current directory.
- `--max-urls`: It limits the number of URLs to crawl. If the limit is reached, the crawler will automatically stop. Optional. The default is 0, which means no limit.
- `--max-time`: It limits the time (in seconds) to crawl. If the limit is reached, the crawler will automatically stop. Optional. The default is 0, which means no limit.
- `--debug`: Verbose Level. A value between 0 and 1. Set to 1 will have more debug logs. Optional. The default is 0.
- `--cache-type`: Select data structure for cache. A value between 0 and 1. 0 is a simple hash set and 1 is a striped hash set. Optional. The default is 1.
- `--lab-machine`: Set to 1 if you are running on a lab machine or facing an SSL CA problem. Optional. Default is 0. **If set to 1, be sure to put the ca-bundle. crt in the build folder**.

Example: Crawling the Wikipedia English Pages and set the time limit to 60 seconds.

```
./ParallelCrawler --url=https://en.wikipedia.org/wiki/Main_Page
↪  --threads=32 --output=found_urls.txt --max-urls=0 --max-time=60
↪  --debug=1 cache-type=1
```

3 Implementation

In our project, each module is separated into different source code files in folder **src**. It helps to maintain clarity and extensibility. For example, if needed, one can implement different link extractors or queue schedulers with different strategies in their own class.

3.1 Crawler

The module of the Crawler integrates all other classes and spawns crawling threads. The Crawler class performs multi-threaded web crawling starting from a specified URL. It initializes other modules and takes parameters for the start URL, number of threads, file path for storing URLs, maximum number of URLs, and maximum crawl time. The start method logs the beginning of the process, initializes URL storage and scheduling, and spawns worker threads to fetch URLs, process responses, and extract and filter links. Worker threads handle HTTP redirects and statuses, updating the active thread count and visited URLs. The crawler monitors progress, ensuring limits are not exceeded, and stops when limits are reached or no URLs remain. It then joins worker threads and saves found URLs to a file, providing real-time logging throughout the process. This setup allows the Crawler to efficiently handle large-scale web scraping tasks within specified constraints.

3.2 Striped Hash Set

The StripedHashset class is a thread-safe hash set that uses striped locking to allow concurrent access. It initializes with a specified number of stripes, each protected by its own mutex. The class supports inserting, removing, and checking for elements, dynamically resizing when the load factor exceeds a threshold. Each operation locks the relevant stripe to ensure thread safety. Additionally, it includes methods for calculating the load factor, getting the size, and saving the set to a file. This design ensures efficient and safe operations in high-concurrency environments. This class is used as the cache and storage for visited or found URLs.

3.3 Other Modules

There are a few other small modules, including link extractor, URL fetcher, URL scheduler, and URL store. Link extractor and URL scheduler are just simple implementations. In link extraction, we use regular expressions to search for the **"href"** or other keywords. We also determine if the links extracted are relative or absolute and handle them in different cases.

For the URL scheduler, we implement a simple thread-safe queue based on BFS. URL store module is a middleware adapting the striped hash set or other data structures to our crawler. It stores the cache and index of the website. The URL fetcher is an implementation to handle HTTP requests. It utilizes Libcurl to download the webpage for our crawler.

3.4 External Libraries

We use two external libraries in our project. The first one is Libcurl as indicated before. It encapsulates methods for handling HTTP requests easily. The second one is CppLog. This is a thread-safe logging library, which provides user-friendly and light-weight logging to our crawler.

4 Analysis

4.1 Testing Environment

In this report, the experiments are done on a specific testing environment since these times may vary depending on different hardware configurations or operating systems. The result is highly influenced by the Internet connection quality.

- **CPU**: Intel Core i5-11600K @ 3.90GHz, 6 cores 12 threads.
- **RAM**: DDR4 48GB 3200MHZ
- **Virtual Machine OS**: Ubuntu 22.04 x64 for Windows Linux Subsystem 2
- **Host OS**: Windows 11 23H2
- **Cmake**: 3.22.1
- **G++**: g++ (Ubuntu 11.4.0-1ubuntu1 22.04) 11.4.0

The program is running on Hyper-V-based virtual machines. Therefore, the performance might be influenced due to the virtualization. The **O3** optimization flag is enabled.

4.2 Performance of Crawler

We firstly performance improvements of different numbers of threads. We set a time limit of 5 minutes and analyze how many URLs are fetched and how many URLs are found. The total time can be beyond 5 minutes because we wait for all the threads to finish the current URL and end them safely. Anyway, if the time limit is reached, the workers will not get new URLs from the queue and only finish the current job.

Threads	Visited URLs	Found URLs	Total Time (s)
1	1722	443001	300.211
2	3305	627975	300.419
4	5834	1057311	300.663
8	8219	1224248	335.53
16	13394	1603218	302.668
32	15672	1737654	305.206
64	15678	1758751	345.732

Table 1: Performance with Different Number of Threads using Striped Hash-set

From Table 1, it is evident that setting the thread count to 16 is the most efficient for finding the maximum number of URLs. When the thread count is set to 32 or 64, Wikipedia returns an HTTP 429 (Too Many Requests) error. This results in increased throttling and significantly reduces the crawler's performance. Meanwhile, under our environment, by setting thread number to 16, the crawler will use over 90% of CPU resources. It is meaningless to increase the thread number further.

We also tested the cache using a basic thread-safe hash set, implemented from `std::unordered_set`. From Table 2, we observe no significant performance difference when comparing the two different data structures. The overhead associated with locking and unlocking operations likely accounts for the observed performance trend. Due to the high frequency of sequential insert and search operations, this overhead negates the time savings achieved by reducing contention in the striped hash set.

4.3 Profiling

We use different profiling tools on Linux to analyze the bottleneck of the program. We use the **Perf** to get the runtime percentage of all the functions. We found that the most time-consuming part of our crawler is the regular expression search in link extractions. According to Fig. 1, nearly 90% of the runtime is on extracting links.

Threads	Visited URLs	Found URLs	Total Time (s)
1	1688	303282	301.116
2	3062	459099	301.618
4	4446	639560	301.109
8	8279	1084052	301.667
16	14068	1658181	302.281
32	14891	1773533	302.589
64	14762	1550026	475.498

Table 2: Performance with Different Number of Threads using Simple Hash-set

```
Samples: 3M of event 'cycles', Event count (approx.): 3220175588431, DSO: ParallelCrawler
  Children      Self  Command          Symbol
-   94.14%     0.00%  ParallelCrawler  [.] Crawler::worker_thread
   - 94.14% Crawler::worker_thread
      - 93.55% LinkExtractor::extract_links
         - 93.18% std::regex_search<__gnu_cxx::__normal_iterator<char const*, std::__cxx11
            - 93.18% std::__detail::__regex_algo_impl<__gnu_cxx::__normal_iterator<char co
               - 92.29% std::__detail::_Executor<__gnu_cxx::__normal_iterator<char const*,
                  + 87.50% std::__detail::_Executor<__gnu_cxx::__normal_iterator<char cons
                    2.97% __mcount_internal
                  + 1.75% __gnu_cxx::operator!=<char const*, std::__cxx11::basic_string<ch
```

Fig. 1: Perf Result

4.4 Further Improvements

Firstly, to improve the performance of link extraction, we can use the **RE2** library for high-performance regular expression matches. By switching to this library, the time consumption percentage can decrease to 5% from over 90%, shown by Fig. 2. The performance of the crawler is increased greatly shown by Table 3. However, since we do not have too much time to testing on this library, we put the implementation in the **dev** branch. For the main branch, we still use the **std::reg_search**. Secondly, we store all the found URLs in the memories without saving them into the disk and reclaiming the memories. It is possible to have a memory overflow if the website is too large. Therefore, we can implement another thread for saving the URLs continuously into the disk and reclaiming the memories.// Moreover, it is possible to implement a headless browser to adapt our crawler to a wider range of websites since some websites require Javascript to correctly render the content.

```
Samples: 1M of event 'cycles', Event count (approx.): 3857680086537
  Children      Self  Command          Shared Object      Symbol
-    1.06%     0.01%  ParallelCrawler  ParallelCrawler    [.] LinkExtractor::extract_links
   - 1.05% LinkExtractor::extract_links
      + 0.83% std::unordered_set<std::__cxx11::basic_string<char, std::char_traits<char>, s
```

Fig. 2: Perf Result using RE2

Threads	Visited URLs	Found URLs	Time (s)
1	1981	543455	300.083
2	4317	851420	300.211
4	7213	1367001	300.587
8	9870	1710885	300.741
16	14439	2435152	301.295
32	12634	3202352	301.765
64	13603	2285103	306.844

Table 3: Performance after using RE2

5 Conclusion

In this project, we successfully developed a multithreaded web crawler capable of efficiently indexing a large-scale website like Wikipedia. The project included designing and implementing key components such as the multithreaded crawler and a concurrent data structure for managing visited pages, utilizing libraries like libcurl for HTTP requests and CppLog for logging. Our performance analysis demonstrated the scalability and efficiency of our approach, with optimal performance achieved using 16 threads. We identified bottlenecks in the link extraction process and proposed further improvements, such as integrating the RE2 library for faster regular expression matching.

Overall, our parallel web crawler shows significant promise for large-scale web indexing tasks, providing a solid foundation for further enhancements and optimizations. The detailed performance analysis and profiling guided our understanding of the system's behavior under different conditions, paving the way for future work aimed at further improving the crawler's efficiency and robustness.

Future work could focus on refining the link extraction module, exploring alternative data structures for better concurrency management, and extending the crawler's capabilities to handle a wider variety of websites with different structures and contents.