Web Crawler Project Report

Erik Dziekonski Bautista
https://github.com/ErikDz/CSE305_project

1 Introduction

This project consists of a Web Crawler, it aims to crawl any website and get an index of all its pages. The project itself has been compartmentalized in different files, each serving a unique purpose in the process. We will now take a brief look at the structure of the project and the different components of it, briefly explaining the role of each.

2 File Structure and Purpose

I've structured the crawler in the following way:

- `webpage_downloader.h` and `webpage_downloader.cpp`: These two files take care of downloading the raw HTML string of a website using the libcurl library. The `downloadWebpage` function takes a URL as input and returns the downloaded webpage (as raw HTML) as a string. It also handles redirects and in case there is any redirect-loop, it has a max-depth hardcoded in order to prevent this case.

- `link_extractor.h` and `link_extractor.cpp`: They take care of extracting links from a webpage. The `extractLinks` function takes the raw HTML of a webpage and a base domain as input and returns a vector(str) of extracted links. This is done through regex expressions. It also includes utility functions for normalizing links, extracting domains, and validating URLs. The reason why the base domain is passed, is so that we can deal with relative links, and we can prepend the domain.

- `webpage_parser.h` and `webpage_parser.cpp`: These files contain functions to extract the title and content summary from a webpage. The `extractTitle` function extracts the title from the HTML `<title>` tag, while the `extractContentSummary` function extracts the meta description as the content summary. Again, making use of regex expressions parsing the `<head>` of the HTML doc.

- `setlist.h` and `setlist.cpp`: These files define the `SetList` class, which is a data structure used to store the visited webpages. It provides thread-safe methods to check if a webpage has been visited and to insert a new webpage into the set.

- `stripedhashset.h` and `stripedhashset.cpp`: They define the `StripedHashSet` class, it is a concurrent hash table data structure which is used for storing all the visited webpages. On top of that, it supports multiple reader threads and a single writer thread, allowing efficient concurrent access.

- `crawler.h` and `crawler.cpp`: The heart of the crawler. They contain the main implementation of the crawler. The `Crawler` class takes care of the crawling process, including thread synchronization, URL queue management, and performance analysis. It uses all the previously mentioned files in order to crawl the websites.

- `tester.cpp`: Allows us to test functions to verify the functionality of the webpage downloader, SetList, and crawler components. Make sure nothing breaks during development.

- `main.cpp`: This file serves as the entry point of the program. It prompts the user for a starting URL. It then creates an instance of the `Crawler` class and runs the crawling process. It also automatically finds the optimal number of threads (increasing gradually) to test for performance comparison. Finally, it saves the website index to a file and performs performance analysis.

3 Features

I implemented the following features:

1. **Multithreaded Crawling**: The crawler uses multiple threads for crawling, thus it crawls through multiple web page simultaneously and at a faster rate than a single thread. In such a way, each thread, as well as the one that downloads a peculiar webpage, extracts links, and updates the URL queue simultaneously.

2. **Concurrent Data Structures**: I use synchronized data structures in order to work concurrently as a result of using multiple threads. The `SetList` class is a basic setbased data structure with multiple shared mutex lock, on the other hand, the `StripedHashSet` class is the concurrent hash table with striped lock (higher performance).

3. **Performance Analysis**: It records the time for downloading webpages, extracting the links, and updating the URL queue. The performance metrics are provided as a part of the crawling procedure output.

4. **Website Index Generation**: It forms a website index that contains: the URL of a visited webpage, its title, and the metadescription. Before the algorithm ends this index is saved into the JSON file and can be reused or processed in the future (or graphed as seen later).

5. **Link Extraction and Normalization**: It has also link extraction and normalization functions. It parses links from HTML of the webpages using regex and cleans for query parameters, fragments and handles relative links.

4 Performance Analysis and Heuristic for Optimal Number of Threads

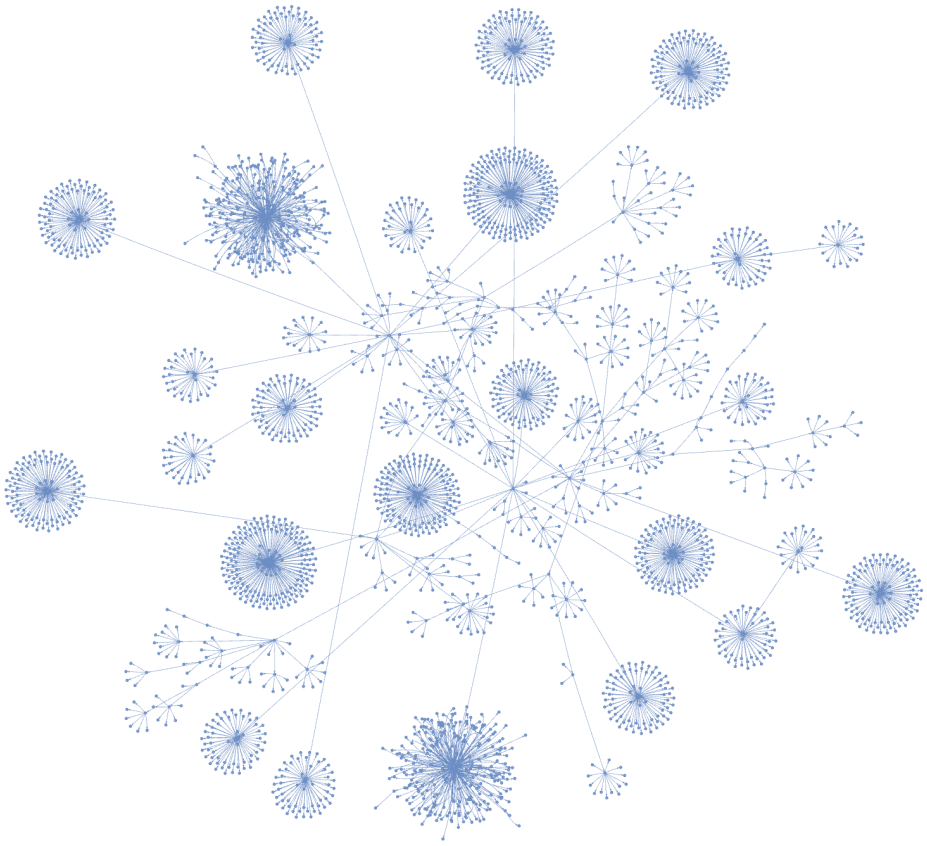

Figure 1: Force-directed graph of python.langchain.com (using the PyVis python library on the generated JSON)

4.1 Performance Analysis

The performance of the web crawler is assessed by measuring the time taken for different stages of the crawling process, specifically downloading webpages, extracting links, and updating the queue. For the test, we chose the website: `https://python.langchain.com/`

4.1.1 Performance Metrics

The key performance metrics captured in the `Crawler` class include:

- `download`: Time taken to download the webpage.
- `extract_links`: Time taken to extract links from the webpage.
- `update_queue`: Time taken to update the queue with new URLs.

4.1.2 Testing with 8 Threads

- **Download Time:** 499555 ms
- **Link Extraction Time:** 511572 ms
- **Queue Update Time:** 16796 ms

The crawling process with 8 threads completed with a total of 2807 pages visited in 247.8 seconds, with a performance of 11.3 pages/s.

4.1.3 Testing with 16 Threads

- **Download Time:** 465589 ms
- **Link Extraction Time:** 1068810 ms
- **Queue Update Time:** 49686 ms

The crawling process with 16 threads completed with a total of 2807 pages visited in 252.5 seconds, with a performance of 11.114 pages/s.

4.1.4 Optimal Number of Threads

As we can see, it is evident that using 8 threads provides the best performance. The optimal number of threads was determined by comparing the performance (pages per second) for different thread counts.

4.2 Heuristic for Choosing the Number of Threads

To determine the optimal number of threads for the web crawler, a heuristic approach was used. The process is detailed as follows (and automated in `main.cpp`!):

4.2.1 Initial Setup

The initial number of threads is set to the hardware concurrency level, i.e., the number of hardware threads supported by the system, retrieved using `std::thread::hardware_concurrency()`.

4.2.2 Performance Evaluation Loop

A loop goes through different thread counts, starting from the number of threads supported by the system and it increases in steps equal to the initial count, up to a maximum of four times the initial count.

- For each thread count, the crawler is initialized with `Crawler crawler(startUrl, numThreads);`.
- The crawling process is initiated with `crawler.run();`.

- Performance metrics are recorded and displayed
- The performance is calculated as the number of pages visited divided by the elapsed time (pages per second).

4.2.3 Determining the Best Performance

The best performance (as mentioned before, we're relying on the pages per second) is tracked during the iterations, and at the end, the number of threads that got the best performance is displayed. The loop terminates early if the speedup between successive iterations falls below a threshold (in this case a hardcoded 1.1×), indicating diminishing returns.

4.3 Reasons for Diminished Performance with More Threads

On the test website, increasing the number of threads from 8 to 16 resulted in diminished performance. Let us take a look at what might be the reasons for this:

4.3.1 Thread Management Overhead

As the number of threads increases, the overhead associated with managing these threads also increases. We can mainly see that thread management overhead occurs in:

- The `Crawler::run()` function, where threads are created and joined.
- The `Crawler::crawlLoop()` function, where each thread performs the main crawling tasks.
- The use of mutexes (`std::mutex` and `std::shared_mutex`) in the `queueMutex` and `visitedMutex` variables to protect shared resources, which can become contention points with increased threads.

4.3.2 Resource Contention

With more threads, the competition for shared resources such as network bandwidth, CPU cycles, and memory increases. The `downloadWebpage()` function can experience bottlenecks due to the limited network bandwidth when multiple threads attempt to download many webpages simultaneously. Almost the same thing with extracting links and updating the queue involve shared data structures protected by mutexes, which can become contention points as the number of threads grows.