CSE305 Project - Final Report

Maika Edberg

June 15, 2022

To Introduce. Throughout the course CSE305 Concurrent and Distributed Computing, we have explored topics related to concurrency. Through coding multi-threaded programs, we've discussed the implementation of concurrent data structures, as well as the challenges involved in Concurrent Programming– namely race conditions, deadlocks and non-deterministic behaviour. As the final project for this course, we chose the Parallel Crawler. In short, the goal is to index a large page (such as Wikipedia) by extracting all links from the webpage and crawling through each to extra their links, and so on.

In this report, we detail briefly the implementation, discuss the behavior as the number of threads vary, and analyze the time taken in each part of the code.

Implementation. To implement this parallel crawler, we use the libcurl library [1]. In essence, we create multiple threads which all work over a shared concurrent queue. First, we start with the original link. We follow this link by requesting to download it on libcurl, then locating references of the form:

$$< a > href = \text{“}\{link\}\text{”} * < a\backslash >$$

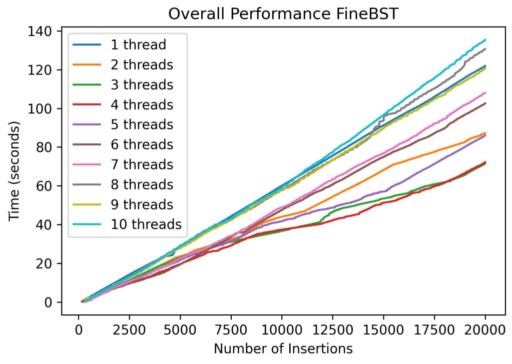
Then we push all the links that we have found here onto the concurrent queue and add it to our thread safe list of links. Since libcurl is thread safe but has no internal thread synchronization [2], we use concurrent data structures used in course to ensure thread safety. Namely, we used SafeUnboundedQueue (TD 4), SetList (TD 5) and FineBST (TD 6). We finish when we run out of links to follow or we reach a predefined bound on the list of links. The full code is available here.

We have faced some issues regarding the libcurl library. As we are working with https requests using SSL, sometimes superfulous issues occur; a full list can be found here. This proved to be one of the difficulties, but generally editing some files were sufficient to fix the issues.

Behaviour Across Threads. Despite the difficulties that we get with multi-threaded programming, it is worth it because of the performance gain we get. Here, we will study the improvement derived from having multiple threads. The code to obtain the plots are available here.

[1] https://curl.se/

[2] https://curl.se/libcurl/c/threadsafe.html



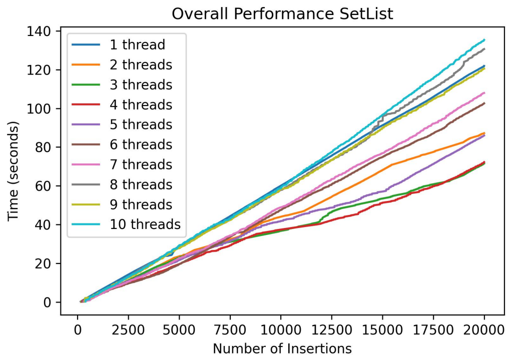
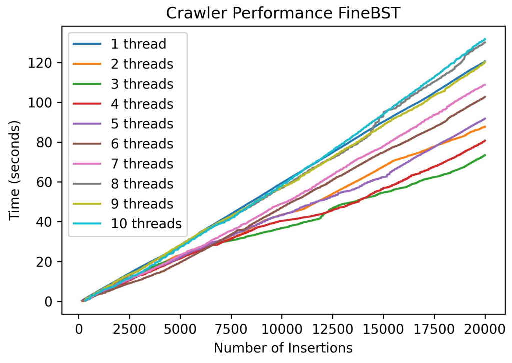


Figure 1: Overall Performance Across Threads



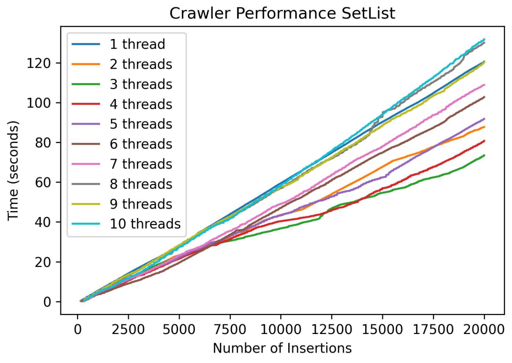


Figure 2: Crawler Performance Across Threads

See figure 2 for the results. In all figures, "Overall" refers to the time spend total, while "Crawler" refers to the time spent crawling the website. Note that this is not a deterministic look at the performance, as many factors could have gone into this, including the other work that was happening on the machine as we waited for the code to run. However, we clearly see an improvement. The most efficient in this situation, either three threads or four, cut the time by around half compared to the sequential version. The machine used for this experiment probably had only three or four cores available at the time to be fully utilized. Now, as the number of threads increase, we see the improvement starts to taper off. In all cases, having nine or ten threads is worse than having a single thread. Even if theoretically we may be working with ten threads, it is not physically possible with the machine, thus the extra cost of creating more threads is not compensated.

SetList vs FineBST. As our project required keeping a concurrent directory of all the links found, we had to make a decision on which data structure to use. First, we used the SafeUnboundedQueue to queue all the links found to crawl them for later use. Then, we had a choice to make for the data structure to keep this concurrent directory: namely, FineBST or SetList. We implemented the SetList first, yet we realized that because we do not need to remove elements, it is theoretically better to use FineBST, as insertion with FineBST is $O(log(n))$ compared to SetList's $O(n)$. Thus, we implemented both to test whether this holds in practice. And indeed, as we take a look at figure 3 FineBST does mostly better than SetList. This gain is not so relevant within the context of the time of computation, however. This suggests that the working with the concurrent data structure is not so much a computational burden compared to the problem we are tackling.

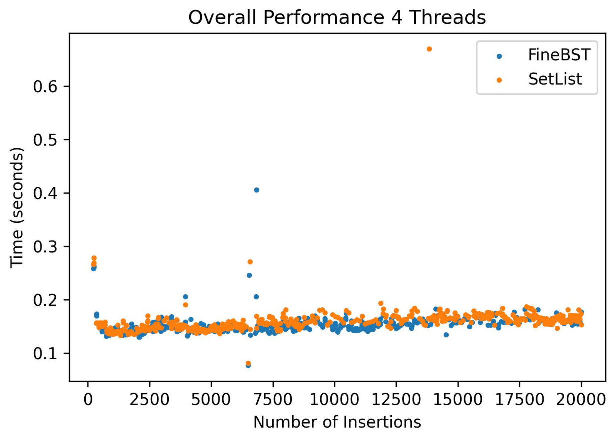
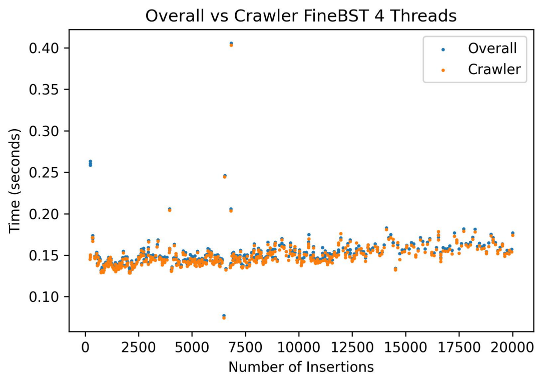


Figure 3: SetList vs FineBST



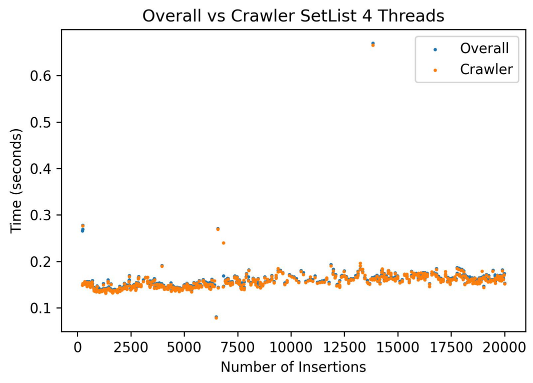


Figure 4: The Computational Bottleneck

The Computational Bottleneck. This brought us to question, what was taking so long? We decided now to take a look at how much time the regex pattern matching was taking, compared to the rest of project. That is, how much time are we spending looking at the raw html code of the website? We found the answer to be almost all the time, as seen in figure 4. No matter how many insertions and which data structures we use, we always end up spending a lot of time just looking at the html code. This seems to be the computational bottleneck that we cannot get over, as html files tend to be large and we have to string search through every character to extract the links.

To Conclude. We are able to see some performance gains from using concurrent methods—with four threads, we are able to cut the computation time in half compared to a single-threaded approach. This was still not sufficient to overcome some of the operations we have to do. Regardless of what methods we use, the string search for links in the websites take a tremendous amount of computational power. However, we still see the extra power that concurrent programming give us to solve problems like crawling which require heavy computations. With multi-threaded programming, we gain access to a new set of tools to solve many problems. This work has shown us that, despite some difficulties that may arise, we can experience performance gains if implemented properly.