ENCE360 Assignment

Josiah Craw
35046080

October 18, 2019

1 Algorithm Analysis

1.1 Describe the Algorithm

The algorithm from lines 228-264 initially moves through each URL within the input file. It then proceeds to get the URL from the file by replacing the end character with a null terminator. The URL and the number of threads is then passed into get_num_tasks. This function destermins the number of tasks required to complete the download given the number of threads. The byte size is then returned. The number of tasks is used to loop through and add that number of tasks to the todo task queue. This queue contains tasks which hold the URL and the min and max range for one task. This separates the file into parts using the byte ranges. Next, the context containing the todo and complete queues is passed into wait_task. This function runs through all of the tasks in the queue and processes them using http_get_content. The files are then merged and the file fragments are deleted.

This algorithm is similar to the worker assignment algorithm that has been used in class.

1.2 Improvements

This algorithm could be improved by deleting the file fragments as they are merged into the final file. Running through all the files twice is inefficient. Splitting the files within threads would not necessarily increase efficiency however, this could help making the system if an internet failure occurs. For this improvement to be effective on running the code would have to check if the downloads had already occurred then catch up to the most recently downloaded file. Another improvement could be merging and deleting the files in a thread while starting the next download in the background. Alternatively the file fragments could be merged as they are downloaded individually.

2 Preformance

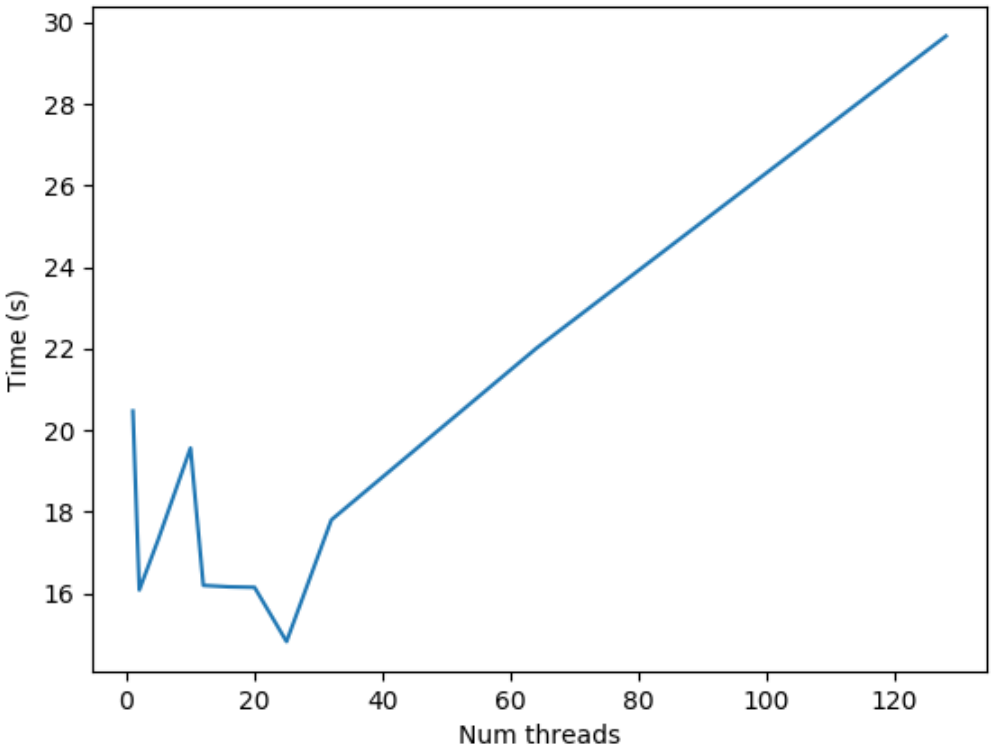

Figure 1: Graph showing number of threads to time for large.txt

Table 1: Table showing threads to time in large.txt

Num Threads	Time (s)
1	20.48
2	16.09
5	17.34
10	19.56
12	16.19
16	16.16
20	16.15
25	14.82
32	17.80
64	22.01
128	29.67

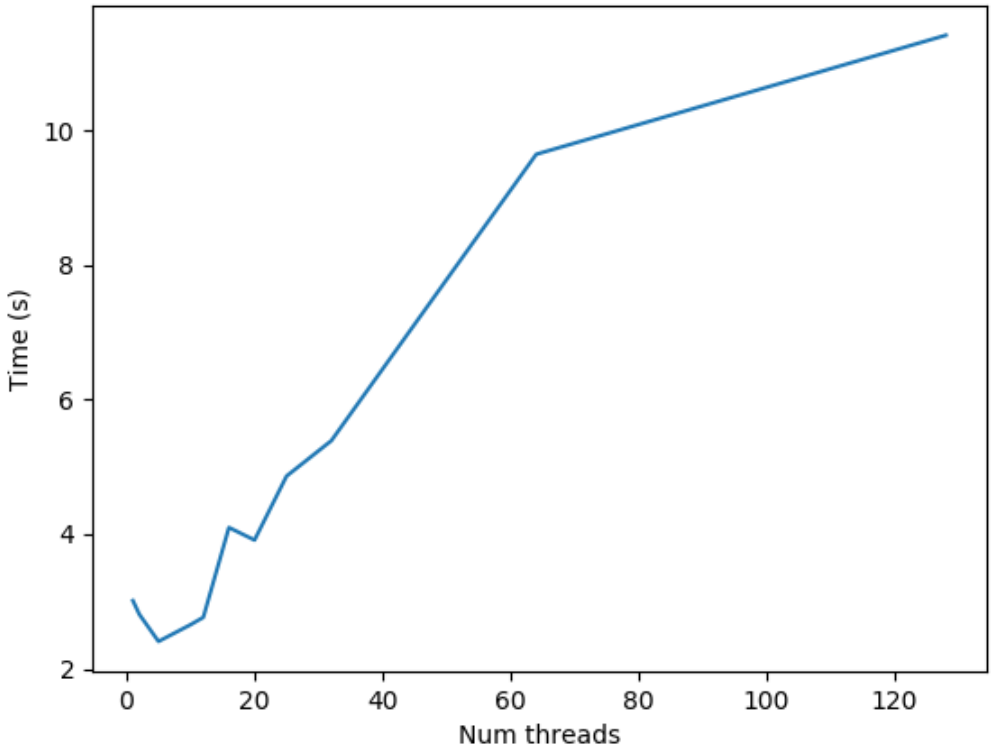

Figure 2: Graph showing number of threads to time for small.txt

Table 2: Table showing threads to time in small.txt

Num Threads	Time (s)
1	3.01
2	2.80
5	2.40
10	2.65
12	2.76
16	4.09
20	3.91
25	4.86
32	5.38
64	9.64
128	11.41

The data above was generated using a python script and matplotlib to plot. The timeing was done within python using time.time() this simplified the data taking

process. For the two files, large.txt and small.txt different thread numbers were optimal for these files. For large.txt 25 threads appeard to be best in testing. In small.txt 5 threads was fastest.

As the number of threads increases the time taken to download increases, most likey due to the merge time increasing as well as the extra network overhead in sending so many packets.

As the file size increases the file downloads slower, but a larger number of threads is better for larger files as the chunks are still large. This is shown in the data as the optimal threads for larger files is higher.

The prformance could be optimised if the files were directly written to the file instead of file fragements then merged. Or if the file fragements were merged and deleted together rather than seperatly.

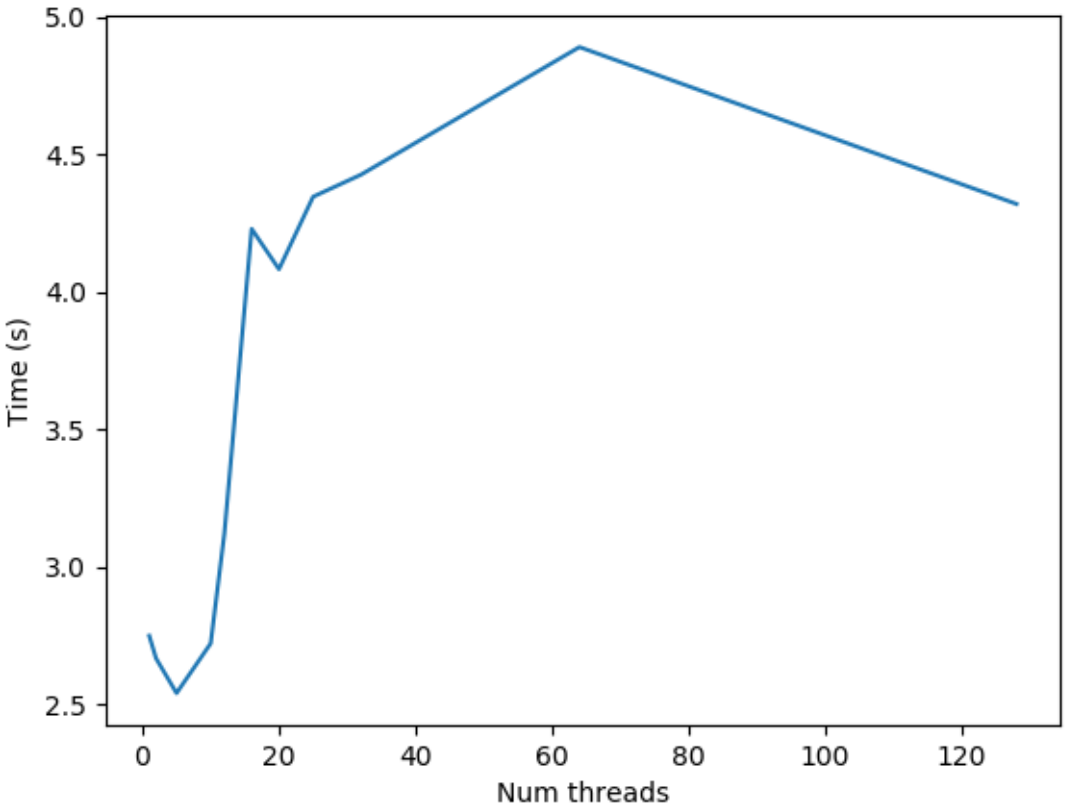

Figure 3: Graph showing the demo bin to number of threads in small.txt

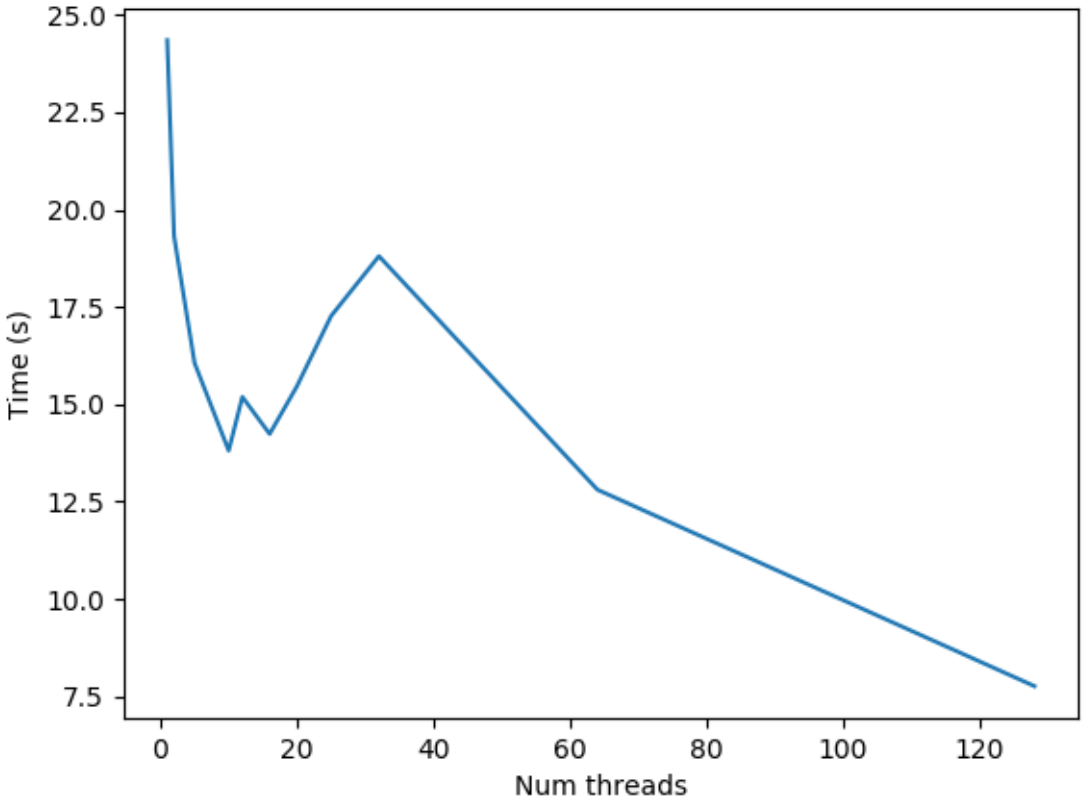

Figure 4: Graph showing the demo bin to number of threads in large.txt

For the higher threads the given code fails saying ERROR connect! showing an apparent reduction in time as threads increase however this is not accurate. However, the code is faster in the given bin for lower numbers.

The merge system goes through and reads then writes the file to the main using the min bytes rang as the name, the delete then does the same but deleting. This is not very good as all the files are gone through twice, which is inefficent. Improvments could be done by doing these operations together.