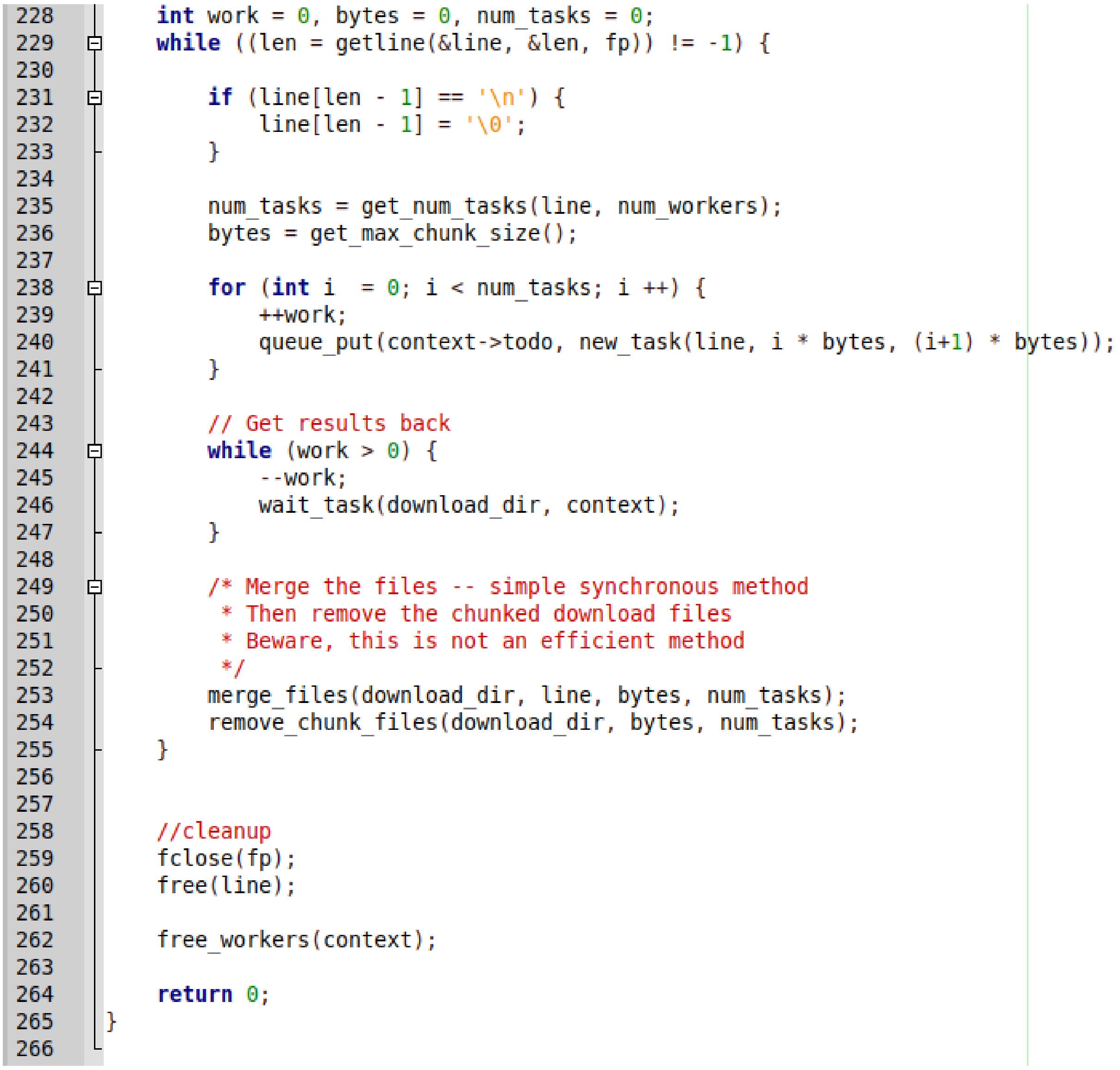
ENCE360 Assignment

Hao Li 83838861

Algorithm analysis



```
228     int work = 0, bytes = 0, num_tasks = 0;
229     while ((len = getline(&line, &len, fp)) != -1) {
230 
231         if (line[len - 1] == '\n') {
232             line[len - 1] = '\0';
233         }
234 
235         num_tasks = get_num_tasks(line, num_workers);
236         bytes = get_max_chunk_size();
237 
238         for (int i  = 0; i < num_tasks; i ++) {
239             ++work;
240             queue_put(context->todo, new_task(line, i * bytes, (i+1) * bytes));
241         }
242 
243         // Get results back
244         while (work > 0) {
245             --work;
246             wait_task(download_dir, context);
247         }
248 
249         /* Merge the files -- simple synchronous method
250          * Then remove the chunked download files
251          * Beware, this is not an efficient method
252          */
253         merge_files(download_dir, line, bytes, num_tasks);
254         remove_chunk_files(download_dir, bytes, num_tasks);
255     }
256 
257 
258     //cleanup
259     fclose(fp);
260     free(line);
261 
262     free_workers(context);
263 
264     return 0;
265 }
266 
```

Line 228: Initializing the number of threads, and the max chunk size to be downloaded. ‘work’ as a temporary variable to record how the tasks going, it is floating between 0 - ‘num_tasks’.

Line 229 - 233: Reading the download list line by line until all lines have been read, each line is treated as a URL, delete the line feed at the end of a URL.

Line 235 - 236: Assigning ‘num_tasks’ to be a number of threads. ‘bytes’: It is a number of bytes need to be downloaded per task, perhaps the actual size downloaded by a task is smaller than this number, but it is never going to be larger than that. It is counted by sending a HEAD request to get the Content-length of the file that needs to be downloaded, then get the ceiling value of (Content-length / num_tasks) to avoid missing data.

Line 238 - 241: Putting the URL and the corresponding download ranges into a queue, ‘work’ is increasing until it reaches the number of threads during the loop.

Line 244 - 247: Inside the while loop, every time the ‘count’ decreases by one, start to download one piece of file with a corresponding range from the queue until the ‘count’ is 0.

Line 253: After downloading all chunk files, to read all chunk file's contents and merge them all into a final file, in my case, I named the final filename is the page name of a URL.

Line 254: Clean all downloaded chunk files to save more system memory after merging.

Line 259: After downloading all the files from the download list, close the txt file.

Line 260: Free the allocated memory for URL.

Line 262: Free the queue’s content.

This is similar to the producer-consumer problem mentioned in lecture notes, there is a queue to handle the ‘producer’ and ‘consumer’ in both of them. Nevertheless, there are some slight differences between them. The real producer-consumer problem is synchronization, producer generates data and consumer consumes the data at the same time. Our assignment is to put all data into the queue, then consume it all in order.

Improvement: The way to improve it is roughly combining the two loops from line 238 – 247 in the beginning. However, the default number of threads becomes one after combining, because it follows the order inside the loop, put one into the queue then download one from the queue. It should modify the download mechanism, to make sure producing and consuming synchronously, and prevent deadlock is necessary.

Performance analysis

Threads	small.text (in Second)	large.text (in Second)
1	3.3584	20.2013
2	2.8049	18.3276
4	2.4822	16.5979
8	2.5706	15.3269
12	2.9341	15.6760
16	3.5269	16.1467
32	5.4371	20.5263
64	7.2088	24.8965
128	11.4381	30.2795

Figure 1

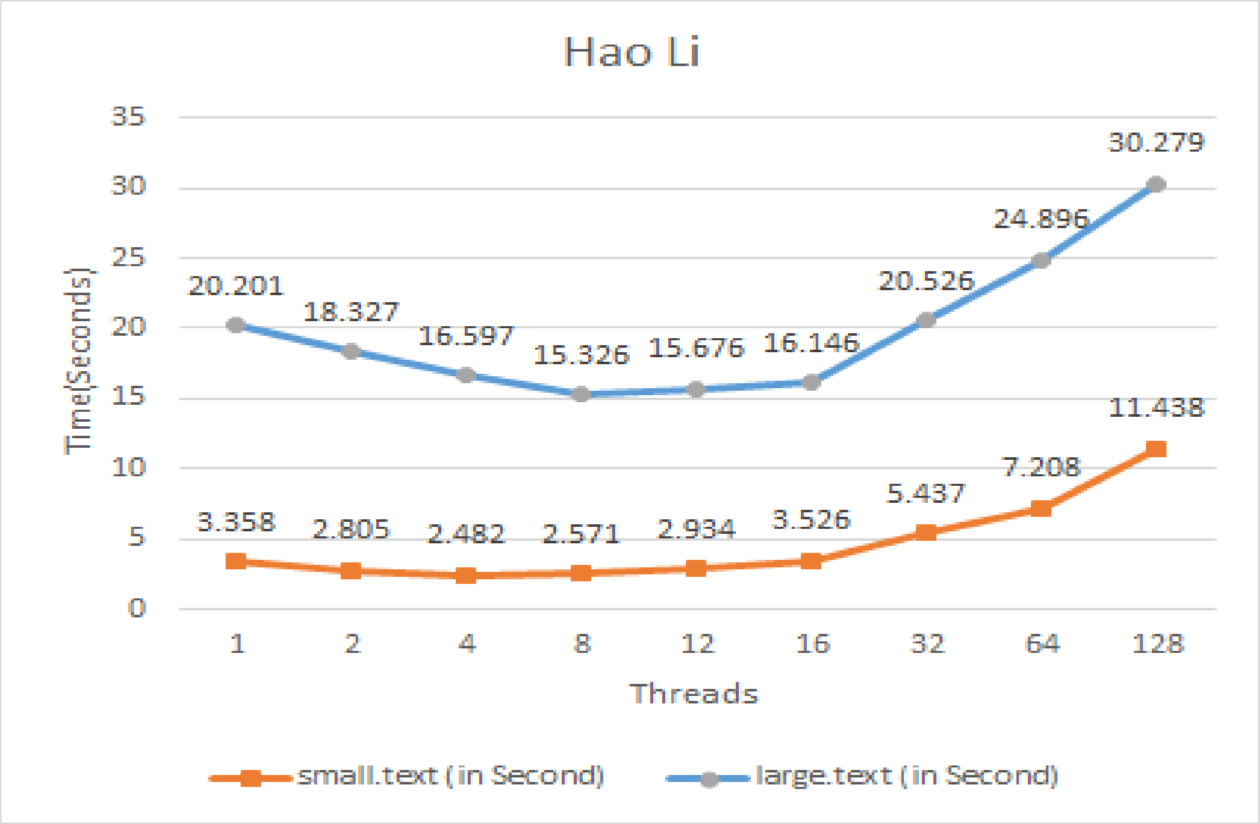


Figure 1-1

Threads	small.text (in Second)	large.text (in Second)
1	2.9434	19.7719
2	2.6871	18.5034
4	2.3566	16.3598
8	2.6104	14.2567
12	2.9684	15.2810
16	3.5233	16.2544
32	4.6702	20.2369

Figure 2

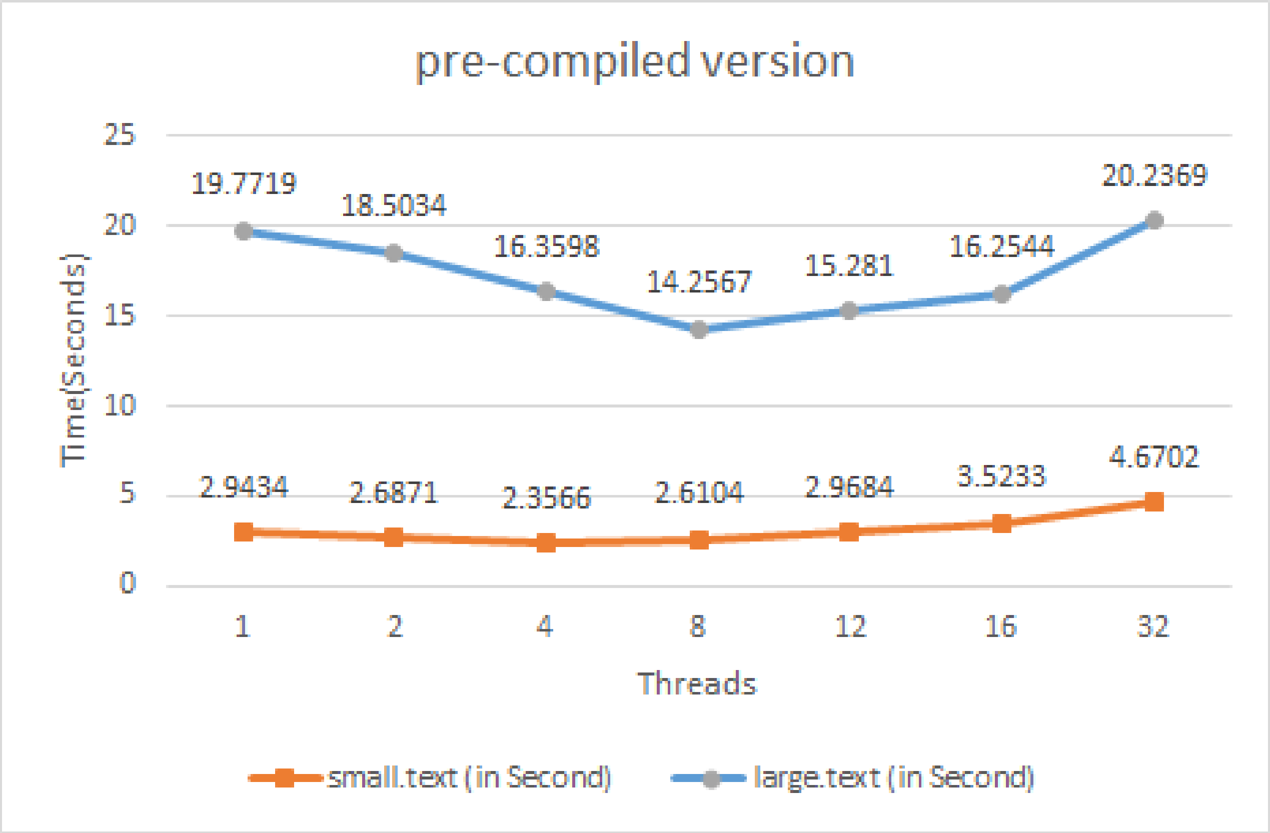


Figure 2-1

According to the performance test, I generated figure 1, figure 1-1, figure 2 and figure 2-2, which are tables and line charts showing how the number of threads impacts the performance in different file sizes. Figure1 and figure1-1 are showing my program running result, Figure2 and figure2-1 are showing bin/downloader running result.

In two types of testing, the optimal number of threads in small.text is 4, the optimal number of threads in large.text is 8. From single-threaded download to 4 or 8 thread downloads in different environments is getting better. Once the number of threads is larger than the optimal number of threads, the performance is getting degradation.

The download time is definitely getting longer while the file size is getting bigger. But it still shows more advantages of multi-threaded downloads than single-threaded download, the optimal multi-threaded downloads save 5 seconds in large.txt, 0.8 seconds in small.txt.

The current approach used in merging and removal is working for every size of the file, but it is not efficient every time. There are too many chuck files that have been downloaded when file size and the number of threads are both too big. It takes a long time to read all the chunk file's contents, combine them all to the final file then remove all chunk files. So according to the file size, it is necessary to choose the appropriate number of threads, it would maximize the performance of multi-threaded downloads.

The way to improve that is to create a final file first, download the same content as each chunk file every time, but instead of creating a chunk file, it directly writes the content into the final file in turn. We can save the reading time for each chunk file, writing time to the final file and removing all chunk files.