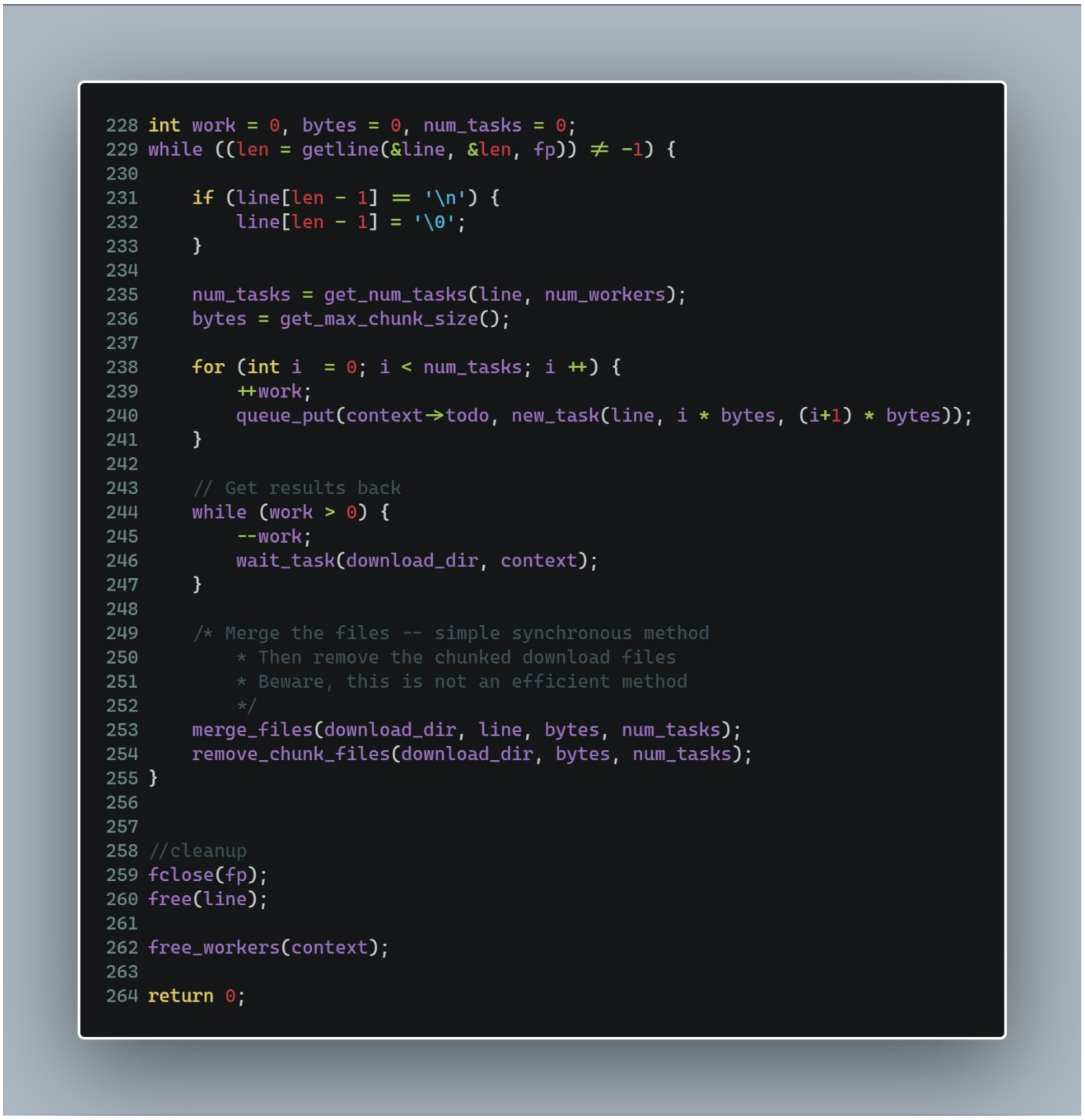
ENCE360 Assignment Report

## 1 Algorithm Analysis



```c
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;
```

| | |
|---|---|
| L228 | Declares variables:<br>`work`: stores the number of tasks which are incomplete<br>`bytes`: stores the maximum chunk size to be downloaded<br>`num_tasks`: stores the total number of tasks for the download |
| L229 | For each line in the opened file, it stores the string `line`, and the length of the line in `len`. |
| L231-233 | Ensures that the string stored in `line` has a null terminator in the place of a newline character, if a newline character exists in `line`. |
| L235 | Gets the number of tasks it will take to do a complete download of the given URL stored in `line`. |
| L236 | Gets maximum size of a chunk to be downloaded, which forms a partial download of the resource located at the given URL. |

| | |
|---|---|
| L238-241 | It puts `num_tasks` tasks on the queue in context`->todo`. Each task will download a chunk of the resource. This forms the dispatcher portion of the dispatcher/worker model – it gets the next request and puts it on the queue for the workers to process. For each task put onto the queue, the counter `work` is incremented. |
| L244-247 | Until there are no more tasks on the `todo` queue, `wait_task` gets completed tasks from the `context->done` queue, and saves the downloaded buffer (which is a partial download of the resource) to disk, with the name being the integer representing the first downloaded byte of the entire partial download. |
| L253 | Merges the partially downloaded files into a single file, which is a modification of the URL. |
| L254 | Deletes the partial chunk files. |
| L259 | Closes the open file (which stored the list of URLs). |
| L260 | Frees the allocated memory, which was allocated by `getline`. |
| L262 | Frees all the worker threads. |
| L264 | Returns `0`, telling the operating system that no errors occurred. |

The algorithm used here is a combination of the dispatcher/worker model, and the team model.

The dispatcher/worker model is illustrated by:

- the use of a parent thread dispatching work onto the `todo` queue, in L238-241
- the use of workers waiting for work (the `worker_thread` function passed to `pthread_create`)

The team model is illustrated by the fact that each of the spawned threads are equal, and each thread processes the task on its own. In comparison, a pure dispatcher/worker model has the ability to have specialised workers, and for specific requests to be dispatched to specific workers.

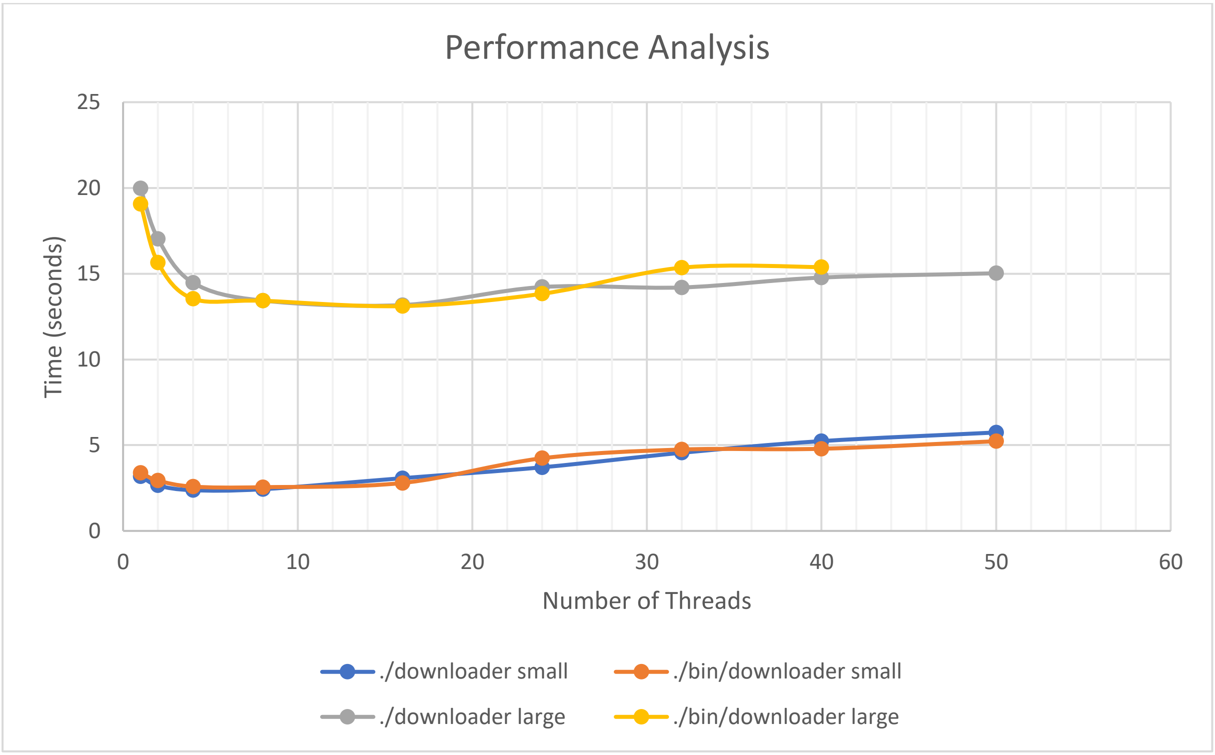
Additionally, the downloader uses the producer/consumer model for synchronising access to the fixed queues. This is done by using semaphores and mutexes.

## 1.1 Improvements for the downloader model

One possible improvement is for each thread to write to its own file upon completion of a partial download, instead of `wait_task` doing this on the main thread. Merging on the main thread introduces a bottleneck, such as when multiple tasks are completed and waiting to be processed by `wait_task`, but only one is being processed at that point in time. However, this may introduce collisions if the same file was downloaded at the same time by multiple threads, though this can be mitigated.

Another alternate possible improvement would be to have each thread append to the file as it receives each chunk. However, this may negate the benefits of using buffered I/O as it could increase kernel calls.

## 2 Performance Analysis



The times are the result of running each binary-thread-file combination five times, ignoring the two outliers, and taking the mean of the three interior times[1].

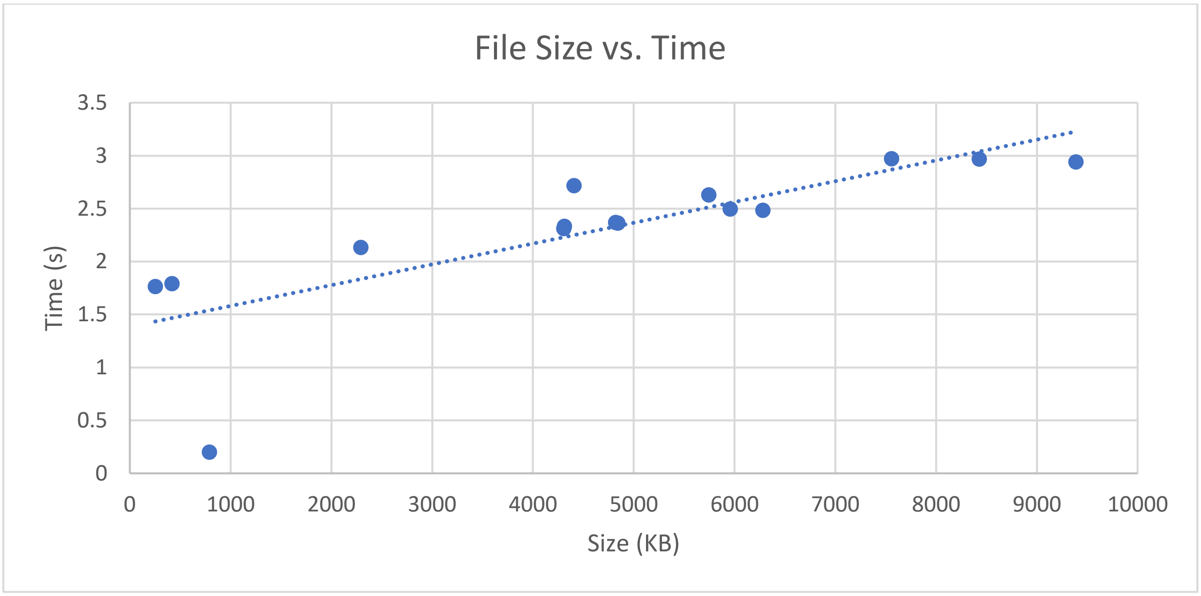
As shown in the plot above, the optimal number of threads is either 4 or 8. This makes sense, since the above processes were run on a 4-core processor, which had 8 threads. Having less than 4 threads will result in an underutilisation of the multi-processing ability of the CPU, thus decreasing the performance. Increasing the number of threads will increase the amount of context-switching, which introduces additional overhead. Additionally, increasing the number of threads has the potential to overload the caches, causing the processor to evict data to slower storage, which further increases the amount of time for context-switching.

The downloader and the provided downloader performed consistently similar, with a difference typically below 0.8 seconds. The one major difference was the inability of the provided downloader to run with 50 threads, whilst using `large.txt` as the source file of URLs.

[1] See timer.py

## 2.1 File Size

Download times appears to increase linearly with file size, as is logical. This is shown by the plot below:



The plot is fitted with a linear trendline. The $R^2$ value, including the outlier is 0.62. However, removing that outlier results in $R^2$ being 0.81. The outlier is an image served by Unsplash, which is hosted by the Fastly, a CDN.

## 2.2 Downloader Optimisations

One optimisation would be the removal of print statements. This would remove rendering by the terminal, as well as the need to flush buffers to pass to the terminal.

Another optimisation would be to use a distributed architecture, so that the downloader would run on multiple computers. This would reduce the amount of bottlenecking by the network and allow the utilisation of a greater number of threads. However, this approach has a number of limitations. For example, it may be necessary to move downloaded files to a single location. However, fundamentally, using a distributed system for the downloader would increase performance (if implemented well).

A further potential optimisation would be to increase the buffer size. This can decrease the CPU time, even if not improving the read time [1]. However, it would be best if other considerations and experiments were undertaken to find the optimal buffer size.

Another optimisation would be to dynamically allocate the number of threads, taking into account the ability of the system.

A further optimisation would be to ensure that chunk downloads, and thus buffer sizes are a power of 2, and the same as the disk block size. This means that the amount of multiple disk block reads and writes is reduced, and reads always use the full block.

Some performance optimisations were done, such as reducing the number of dynamic allocations and reallocations of memory.

## 2.3 File Merging and Removal

The suggested idea of removing files all at once was instead replaced by deleting the files as soon as they became unnecessary – i.e. after their contents were merged into the output file.

As stated earlier, a potential optimisation would be to have each thread to write to its own file upon completion of a partial download, instead of `wait_task` doing this on the main thread.

Another idea would be to have a continual rolling over of unused threads, so that the next file can start to be downloaded whilst the previous file is finishing up. However, this requires the ability to distinguish which output file each partial file corresponds to. This overlap would ensure that the queue was always full, and thus, the workers were always being productive.

One optimisation which was implemented was the use of a buffer with a length of `BUFSIZ`. `BUFSIZ` is chosen on each system in order make stream I/O efficient [2]. This reduced the need to dynamically allocate memory for scenarios where the downloader was attempting to allocate a large amount of space for an entire file, as the downloader was running on a single thread.

The current method of partially downloading every file means that large files can be downloaded with easy recovery from errors, due to the ability to pause and resume downloads. Though this adds the overhead of the HTTP header for every partial download, this method is optimal as it balances the need for reliability with the ability to download files fast. If the goal was to download many files fast, a potential optimisation could be to download multiple files simultaneously, each file on its own thread. However, for very large files this could be potentially dangerous and difficult, requiring large amounts of system resources. Overall, partially downloading every file is optimal.

## 3 References

[1] Javamex, “How big should my input stream buffer be?,” [Online]. Available: https://www.javamex.com/tutorials/io/input_stream_buffer_size.shtml. [Accessed 18 October 2019].

[2] GNU Project, “12.20.3 Controlling Which Kind of Buffering,” Free Software Foundation, [Online]. Available: https://www.gnu.org/software/libc/manual/html_node/Controlling-Buffering.html. [Accessed 18 October 2019].

[3] Factory.hr, 26 July 2018. [Online]. Available: https://medium.com/@factoryhr/http-2-the-difference-between-http-1-1-benefits-and-how-to-use-it-38094fa0e95b. [Accessed 2018 October 2019].