ENCE360 Assignment

Multi-threaded HTTP Downloader Program

Reka Norman (21967425)

1. Algorithm Analysis

The algorithm in downloader.c performs the following steps:

- The given number of threads are created and setup to process tasks from the todo queue until there are no more tasks remaining.
- The program loops through all the URLs in the given URL file, and for each one:
    - Gets the number of tasks and the maximum chunk size to be used (the number of tasks is taken to be equal to the number of threads, and the chunk size is equal to the total file size divided by the number of tasks).
    - Creates the new tasks and adds them to the todo queue. Each task is responsible for downloading a particular byte range of a particular file.
    - The tasks added to the queue will be processed by the worker threads, which will perform the partial file download, place the downloaded data into the task's buffer and add the task to the done queue.
    - The main thread removes the tasks from the done queue one at a time, and writes the downloaded file data in each task to a partial download file.
    - The main thread merges the partial download files by copying the contents of each into a new file one by one, and deletes the partial download files.

This algorithm is very similar to the dispatcher/worker model described in the notes. The main thread acts as the dispatcher, since it reads the file download requests from the URL file, and dispatches these as tasks to the worker threads by placing them in the todo queue. The worker threads then perform the actual work of downloading the files. However, the dispatcher/worker model doesn't cover the fact that the main thread has to process each of the tasks again once the workers are finished with them, in order to merge the partial download files. This is the most inefficient part of the algorithm, since it acts as a bottleneck where the main thread is doing all the work and the worker threads are idle. Possible improvements to the algorithm which remove the need to merge files are discussed in Section 2.3. There are also aspects of the team model visible in the algorithm, in the sense that all the worker threads are equal. They each have the same functionality which allows them to process any task in the queue, with no thread specialisations.

2. Performance Analysis

2.1. Impact of Number of Threads

To test the effect of the number of threads on the download time, a URL file containing 10 of the large files (each a few megabytes in size) was downloaded using different numbers of threads. Large files were used so that the overhead times such as program setup would be less significant. The timing was done using the linux “time” command, and the results are shown in Figure 1. It is hard to tell exactly what the relationship between number of threads and download time is, since there are many variations in the download time which are caused by other factors. These include network-related factors such as the data rate on the network links being used, both locally and on the server side, as well as other programs running on the same machine while the timing is being done. To reduce the effect of these variations, each trial was repeated five times and the results were averaged.

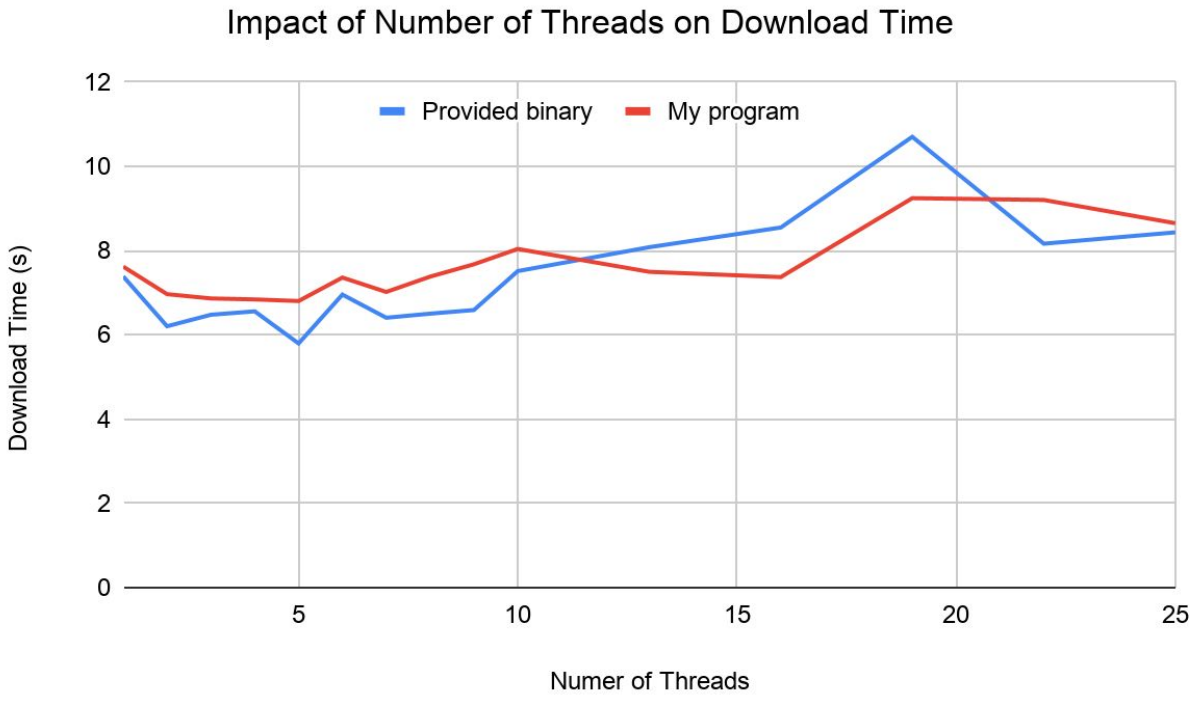

Figure 1 - The relationship between the number of threads and the download time.

The optimal number of threads, both for the program written and for the provided pre-compiled binary, appears to be around 4 or 5. However, the number of threads does not seem to have a very significant effect on the download time, so it is difficult to find a clear pattern in the results. As the number of threads increases above about 7, the download time starts to increase. This is probably due to the overhead associated with using more threads. For example, more threads need to be initialised and given resources. There are more HTTP requests and responses made in total, so there is more overhead associated with HTTP headers. A larger number of partial

download files are needed, so there is an additional overhead related to creating and deleting these. There will be more context switching time required between the threads, and more threads will be contending for the shared queue, so it is more likely that threads will block while waiting for the mutex to be available. As the number of threads becomes large, these overheads outweigh the benefits of parallel processing. Also, the testing was performed on a machine with a single four-core CPU, so no more than four threads will ever be running truly simultaneously.

In comparison to the provided binary, the performance of the program which was written is similar, suggesting that the implementation works correctly. Both the programs perform best when the number of threads is around 2 to 6, and then gradually become slower as the number of threads is increased. The precompiled-binary seems respond more significantly to changing the number of threads, with the download time at around 19 threads being much longer. However, it is likely that this variation is due to some unrelated factor as described above. For a more accurate analysis, the tests would need to be performed in a more controlled environment.

2.2. Impact of File Size

The effect of file size on the download time was tested by downloading several files of different sizes from both small.txt and large.txt, and timing how long the download took. For each trial, 10 threads were used and only a single file was downloaded each time. Each trial was repeated three times and the download times averaged, to reduce the variations due to other factors such as the network speed and other processes running during testing. The results of these tests are shown in Figure 2.

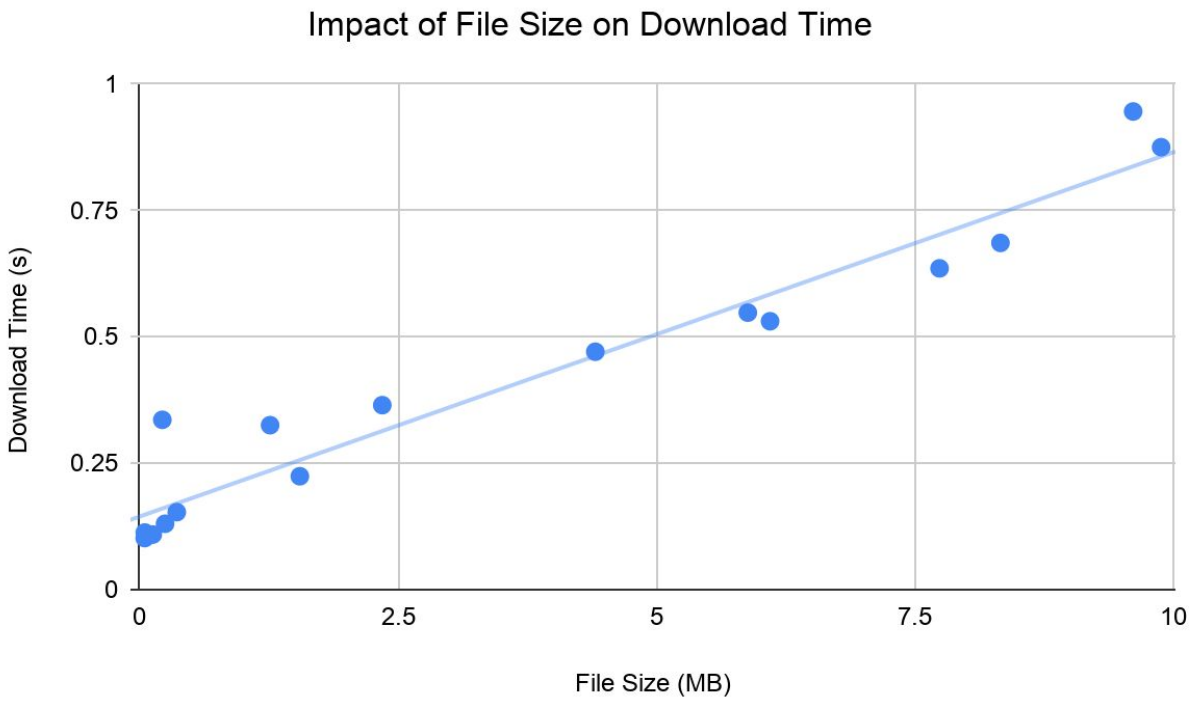

Figure 2 - The relationship between the file size and download time.

It can be seen that there is a roughly linear relationship between the file size and the download time, as expected. As the file size increases, there are two main factors which cause the download time to increase: the increased time taken to actually receive the file data from the server, as well as the increased time taken to merge the partial download files into a single file. There is no real way to reduce the actual download time from the server, however the merging time could be reduced to improve the performance of the program for large files. This is discussed in Section 2.3.

The y-intercept of roughly 0.15 s represents the overhead part of the download, which is not directly due to actually downloading the data, and is therefore constant regardless of the file size. This includes things such as setting up the program and initialising the threads. It is clear that for small files, this overhead time makes up a large proportion of the total download time, while for large files it is less significant.

2.3. File Merging and Removal

Currently, the partial download files are merged into a single file and then deleted using two synchronous functions (merge_files and remove_chunk_files). Since these are synchronous, all the work is done by the main thread, and increasing the number of threads will not improve the performance. Therefore, even if downloading the partial files from the server can be done quickly using multiple threads, there will be a bottleneck at the merging stage. The merging process is also very inefficient, since it requires copying the entire contents of each file from a partial download file to the final file. This copying involves reading and writing from and to files on disk, which are slow I/O operations.

To improve the program's performance, these unnecessary copying operations need to be removed. The file data received from the server should be written directly to the final file, rather than to temporary partial download files. However, if multiple threads are performing the download, they can't write to the same file simultaneously, as this would introduce race conditions. A possible solution would be to keep the partial downloads in main memory (stored in their Buffer structs) instead of writing them to temporary files in the wait_task function. A synchronous function could then take all the gathered partial download Buffers and write their contents to a single file in the correct order. This would still involve a bottleneck where the main thread is doing all the work of writing to the file, but there would be no unnecessary copying.

Another possible solution would be to have each thread completely download a single file. This would allow each thread to download its own file from the server and write its contents to disk without interfering with the other threads. Multiple files would be downloaded simultaneously, improving the overall download speed. There would be no partial download files, and therefore no merging operation would be needed. However, this approach would obviously only have the benefits of multithreading if multiple files are being downloaded.