Riley Aitken
15928464

ENCE360 Assignment Report

Downloader.c Algorithm

The program first creates a ‘Context’ structure, which in of itself contains two queues (one for ‘todo’ tasks that are waiting to be collected by some worker thread, and one for storing the results of these tasks upon completion) and an arbitrary number of threads (the exact number is specified on the command line). The aforementioned queues are then allocated (dynamically), and the number of threads as specified are created using ‘pthread_create’.
Each worker thread upon creation starts execution in the specified ‘worker_thread’ start routine, which defines worker behaviour throughout the entirety of the program’s execution. Within this function, the worker obtains a task from the ‘todo’ queue, calls the associated function for completing the obtained task and updates the ‘result’ aspect of the task with the resultant data. This ‘completed’ task is then put onto the ‘done’ queue, waiting for collection from the main program, where it will be further processed. The worker thread will then attempt to obtain another completable task from the ‘todo’ queue, and loop again.

Now to describe the algorithm within the ‘main’ function of the program; the real meat and potatoes. The program is passed a text file, with a number of ‘tasks’ on separate lines; in this context, they are URLs to fetch data from. The program maintains a ‘work’ variable, which is initialised to 0 and for each ‘line’/task in the parsed file is incremented. For each parsed line in the file, a ‘Task’ object is created by processing the text, and this ‘Task’ object is placed on the ‘todo’ queue to be consumed by worker threads. Of note is the next conditional statement, which checks if the value of ‘work’ exceeds the number of specified threads. If it does, then ‘work’ is decremented and new tasks are not processed from the input file until a blocking function call ‘wait_task’ has returned. This function involves the processing of some completed task on the ‘done’ queue and ‘freeing’ the associated ‘Task’ object from dynamic memory/the heap. It is important to note that this function is always called by the ‘main’ thread of execution, and that processing of completed tasks is not distributed amongst the various threads that have been created in the program. This design choice results in less performance than if such processing was distributed and not performed by a singular thread.

Of note is the fact that this conditional statement checks ‘work’ against the specified number of threads, and not the actual lengths of the two queues, each of which were allocated to two times the number of threads in the initialisation portion of program execution. I assume this was a design decision to ensure the correctness of the program, though it means that both queues are not being utilised to their utmost potential and as such, this may be a performance trade-off.
The entirety of the specified text file is parsed, task by task, as described in the past two paragraphs. Once this is done, ‘work’ now contains the number of tasks that are yet to be completed and processed. A loop is ran through, where the ‘wait_task’ function is called and ‘work’ is decremented until it reaches zero, which signifies that there are no more tasks left to

complete and process. As before, this loop of obtaining complete tasks and processing them is handled entirely by the main thread, which is a potential performance issue.
All associated, dynamically allocated memory is finally freed from the heap.

Multithreaded Design Choice

Downloader.c describes a 'team model' for its worker threads, where each worker thread has access to the same 'context' (and both associated queues) and can perform a task/make a request to a URL independent of one another. Each thread obtains 'tasks' from the 'todo' queue, which is managed by the main/dispatcher thread. The main thread is the dispatcher thread, since it can be interpreted that the input file acts as a feed of requests/tasks and the main thread then delegates these tasks to the 'todo' queue to be consumed by the worker threads. However, mentioned before, the fact that the worker threads do not completely process a task (they only make the HTTP request and place the completed task and resultant data on the 'done queue) suggests less than optimal performance, since it is the 'dispatcher' thread that does the final processing and clean up of completed tasks. If this last minute processing and clean up (writing the results to an output file and freeing all allocated memory) was performed by a worker thread, the program may experience a performance boost.

Performance Analysis

The table below compares the performance of 'downloader.c', which utitlises my implementations of 'http.c' (the module for making GET requests) and 'queue.c' (which specifies a queue structure for managing task scheduling amongst threads) against a precompiled 'downloader' found in the 'bin' folder of the assignment resources. Each of the tests were run 3 times, and the average time of these three tests are seen in each cell below.

'Downloader' Version	**No.of Threads (rows) / Input File (columns)**	Small.txt	Large.txt	Test.txt
My 'downloader'	3 threads	12.5	18.58	0.167
	6 threads	12.55	19.1	0.148
	9 threads	11.83	18.23	0.137
	12 threads	15.1	18.65	0.151
	18 threads	9.34	17.95	0.163
	24 threads	9.12	16.32	0.167
	36 threads	9.85	16.75	0.169
	48 threads	14.17	17.37	0.182

bin/downloader	3 threads	15.21	19.2	0.222
	6 threads	11.5	21.38	0.160
	9 threads	12.136	18.1	0.165
	12 threads	11.35	18.202	0.205
	18 threads	11.5	17.21	0.161
	24 threads	13.94	16.55	0.168
	36 threads	14.935	16.1	0.160
	48 threads	12.5	16.61	0.173

During testing, no clear disparity between the two ‘downloader’ versions was observed, and any minor differences I believe would have been due to network variability and not differences in implementation.

For ‘bin/downloader’ when using the ‘large.txt’ file, it appears that increasing the number of threads stops increasing performance after 18-24 threads or so, though the results of my testing is far from conclusive. When using the ‘small.txt’ and ‘test.txt’ files, there doesn’t appear to be much difference when running the tests with a greater number of threads than say, 8, since there wasn’t a significant increase in performance when the number of threads began growing exponentially.

What is interesting is a disparity between the two ‘downloader’ versions when input is the ‘small.txt’ file (which involves making a larger number of HTTP requests, but receiving much less data on average than for the ‘large.txt’ file), and the number of threads is in the range of 18-36. I believe this due to my ‘http.c’ implementation not handling redirections or any other HTTP responses not simply including the payload in its response, since I can observe some requests receiving 0 bytes in response (when manually navigating to such URLs on the browser, I am automatically redirected to some other page, so I am not sure if these pages no longer exist or if they require authenticated access). As such, I believe the worker threads, when there are many of them, pass on these such requests as ‘completed’ much quicker in my implementation, than in the supplied ‘downloader’.

The ‘size’ of the input files (‘large.txt’ and ‘small.txt’) have effects on performance as you would expect. ‘small.txt’ has a longer list of tasks/URLs to request to, but individually the threads making these requests receive much less data than in the ‘large.txt’ URLs. It follows that as the number of threads would increase, the greater the rate that the tasks/requests are being consumed off of the queue and completed, especially in the case where large amounts of data are being downloaded; it would make sense to have as many such requests (receiving a large payload of bytes) in parallel at any given time. The observed data above suggests that 24 threads is optimal for the ‘large.txt’ file, and any more seems redundant, probably due to a network bottleneck. This number of threads also appears optimal for the ‘small.txt’ file as well.

It appears that an optimal number of threads seems to be around 24, though the testing undergone was far from conclusive. It appears that as the number of threads grows somewhat exponentially, the performance doesn’t change much. While this could be due to a network bottleneck, I believe that since ‘downloader.c’ allocates queues of length linear to the number of threads, this requires additional computation without the additional performance. In fact, due to the issues I outlined previously in this report, because the ‘dispatcher’ thread handles a significant amount of task processing besides actually making the requests, increasing the number of threads significantly doesn’t have an enormous impact because of the design of the algorithm used.

In my ‘http.c’ implementation, I establish a TCP connection with the associated server on all requests. So various threads may be establishing connections with the same server e.g. the University of Canterbury server at the same time. It may speed up the downloading process if the number of times threads have to connect to the server individually is reduced. This could be achieved by identifying a subdomain to which many files are to be downloaded from, and instead of passing a singular HTTP request to be sent to the server on each open connection, a number of ‘pages’ could be passed as a singular task for some thread, so a thread could make several requests over one established connection, instead of just one request per connection.

It is not apparent if a TCP connection is required to fetch the data from the server safely and correctly; if it is not, then use of a connectionless protocol such as User Datagram Protocol for making the requests to the server could speed up the downloading process. Much less processing is required since no connection has to be established before data transfer begins. However, UDP does not guarantee in-order delivery of packets and does not support error and flow control, so the ‘http.c’ file may have to be modified to handle the scenario when packets arrive out of sequence. The only error checking UDP has is a checksum, so little can be done error control wise other than requesting the incorrect packet (if this is even possible in this context).