ECE 459: Programming for Performance
Assignment 1

Sean Byungyoon Kim

January 28, 2019

Part 0: Resource Leak

The resource leak was caused by the write_png_file function, where the write_struct that was created was not being destroyed when it was finished being used. I fixed it by destroying the struct at the end of the function by using png_destroy_write_struct.

Another issue occured where still reachable blocks were found by valgrind within the curl library. The curl_easy_init was not being cleaned up properly with curl_easy_cleanup. To fix this issue, curl_global_init and curl_global_cleanup were used.

Part 1: Pthreads

My code is thread-safe because of the use of mutexes that locks resources that are shared between threads when they are being used. The threads share 3 resources between them: output_buffer, received_fragments and received_all_fragments. All other variables/resources that the threads use are localized within each individual thread so function calls involving those variables can be considered to be thread/safe. There are no race conditions within the code as they are handled by the mutexes that lock/unlock shared resources accordingly. I ran experiments on an Intel i7-4720HQ CPU @ 2.60 GHz. It has 4 physical cores and 8 virtual CPUs. Tables 1 and 2 present my results.

	Time (s)
Run 1	62.189
Run 2	59.052
Run 3	24.792
Average	48.678

Table 1: Sequential executions terminate in a mean of 48.678 seconds.

	N=4, Time (s)	**N=64, Time (s)**
Run 1	20.585	32.248
Run 2	56.865	46.553
Run 3	12.521	27.496
Average	29.990	35.432

Table 2: Parallel executions terminate in a mean of 32.711 seconds.

Part 2: Nonblocking I/O

Table 3 presents results from my non-blocking I/O implementation. I started 64 requests simultaneously.

	Time (s)
Run 1	24.208
Run 2	30.011
Run 3	28.255
Run 4	27.106
Run 5	38.908
Run 6	41.815
Average	31.717

Table 3: Non-blocking I/O executions terminate in a mean of 31.717 seconds.

Discussion. At 64 requests, the non-blocking I/O implementation ran the fastest when compared to the parallel and sequential executions, although the runtimes between the non-blocking and parallel implementations was fairly small. This result is expected as the parallel and non-blocking implementation should see similar performance results and still be faster than the sequential implementation.

Part 3: Amdahl's Law and Gustafson's Law

I measured the user runtime of the entire program (with 4 threads) and the user runtime of the parallel portion of the program by using time ./paster_parallel and the clock() function respectively. Than I subtracted the time taken for the parallel portion from the user total runtime. Over 3 runs, it gave an average of 2.47 seconds or the sequential portion of paster_parallel.

However, Amdahl's Law does not apply to paster_parallel because it breaks the assumption that the runtime of the program can be measured accurately. Since the program receives PNG fragments randomly, the runtime can wildly vary as it'll take different amounts of time for the program to receive all the fragments it needs to build the PNG. As well, server inconsistencies can act as overhead that negatively affects the performance of the program which breaks the assumption of Amdahl's Law that overhead doesn't matter. Therefore, Amdahl's Law does not apply to paster_parallel.

Gustafson's Law does not apply to paster_parallel either because of the random nature that the program receives the fragments from the server. Similar to Amdahl's Law, it becomes difficult to measure the runtime of the program when the program receives fragments randomly whiel also potentially experiencing server errors,