Colt Campbell
CSCE 313_504
November 16, 2013

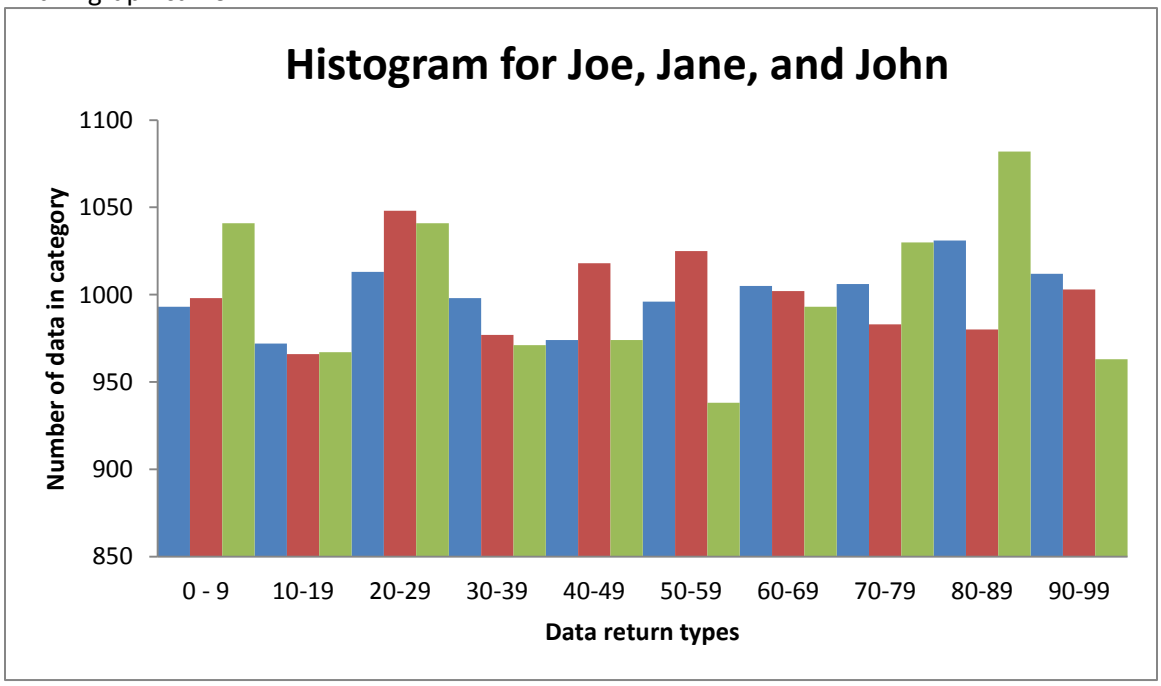
MP4 Report

General Performance Evaluation:

This program is the same as MP3 in every way except for how the worker threads are now handled. Keeping in tune with the way the program should be running, as many channels as requested workers are created within the single event_handler_thread and kept going until all of the numbers of requests for each person are fulfilled. After everything is terminated, memory is still given back to the CPU with the delete command for the buffers, and then a histogram is printed of the resulting statistics. An example of the output of the end histogram is displayed below (again done on the compute.cse.tamu.edu server):

Data	Joe	Jane	John
0-9	993	998	1041
10-19	972	966	967
20-29	1013	1048	1041
30-39	998	977	971
40-49	974	1018	974
50-59	996	1025	938
60-69	1005	1002	993
70-79	1006	983	1030
80-89	1031	980	1082
90-99	1012	1003	963

And in graphical form:



Where Joe is the first column of each triple-set, Jane is the second, and John is represented by the third column.

The program overall operates and outputs basically the same things in the same manner that the code for MP3 did, with the exception that there is a marked difference in performance speed. The performance speed almost entirely comes from the primary event_handler_thread function that does most of the work of the program, and as such, is a function of its general speed, which seems to be on the order of about 3nw, where n is the number of requests and w is the amount of worker threads, and thus about linear run time speed. Looking through the code, I can see no obvious ways to improve the current code. I actually tried to take the nested for loop inside of the while loop in my event_handler_thread (the one that sets the file descriptors in readset) out of the while loop, but that caused problems, of which I commented extensively on in the program code. Essentially, it boiled down to that loop being a necessity inside the while loop directly before select is called because select system calls actually modify certain information bits about the file descriptors that must be reset when seeing which are ready to read again. Actual measurements of the performance are further explored below.

Performance of program with varying numbers of worker threads and buffer sizes:

(All done on the compute.cs.tamu.edu server)

In the first main test run of the program, I used 10,000 requests, a buffer size of 500, and only 30 worker threads. The time it took in microseconds to completely handle all 10,000 requests in microseconds is shown here along with measurements from the MP3 code taken from my first report for comparison of performance and some new runs for that same code.(taken directly from the terminal; the insertion of commas is for readability of the time measurements):

MP4 Source ----
B = 500, W = 30:
10000 requests done in 10,632,933 microseconds.
MP3 Source ----
B = 500, W = 30:
10000 requests done in 5,730,035 microseconds.

(There is an obvious lapse between the MP3 code and the MP4 code for 30 worker threads.)

In the second main test run of the program, still using 10,000 requests, and a buffer size of 500, I changed only the worker thread amount to 50, instead of 30 (an increase in worker threads). The output of that run is here:

MP4 Source ----
B = 500, W = 50:
10000 requests done in 10,320,363 microseconds.
MP3 Source ----
B = 500, W = 50:
10000 requests done in 4,364,921 microseconds.

Interestingly, from the data we gather that the MP4 Code didn't experience a serious increase in performance from W = 30 to W = 50, but the MP3 code dropped a whole million microseconds off by having more threads.

MP4 Source ----
B = 500, W = 70:
10000 requests done in 10,332,568 microseconds.

MP3 Source ----
B = 500, W = 70:
10000 requests done in 4,187,164 microseconds.

MP4 Source ----
B = 500, W = 100:
10000 requests done in 10,383,145 microseconds.
MP3 Source ----
B = 500, W = 100:
10000 requests done in 4,253,031 microseconds.

Obviously, the performance speed had leveled off at around 10.3 million microseconds for MP4 source code and about 4.2 million microseconds for the MP3 source code. Since finding the performance cap for both of the variations of code, I wanted to see if smaller worker amounts would show me where an increase in performance happened for MP4. Starting with w = 5, and working up to w = 20, still comparing to MP3 run times:
MP4 Source ----
B = 500, W = 5:
10000 requests done in 27,962,699 microseconds.
MP3 Source ----
B = 500, W = 5:
10000 requests done in 24,033,532 microseconds.

MP4 Source ----
B = 500, W = 10:
10000 requests done in 17,062,058 microseconds.
MP3 Source ----
B = 500, W = 10:
10000 requests done in 12,918,972 microseconds.

MP4 Source ----
B = 500, W = 20:
10000 requests done in 11,608,581 microseconds.
MP3 Source ----
B = 500, W = 20:
10000 requests done in 7,534,640 microseconds.

From this, we see that about 30 worker threads are optimal for both the MP3 and MP4 code. Prior to that many worker threads, the data shows that both are slower, but still, MP4 is actually performing more weakly than MP3's code performance. To me, this seems concerning being as MP4 is supposed to be an improvement on MP3. I did notice that my MP3 code doesn't always close everything prior to outputting its calculated run time, whereas my MP4 does close everything, and perhaps that has something to do with why my MP4 is significantly slower than my MP3. I would think however, that at a theoretical level, the MP3 code would eventually be slower than the MP4 code because the use of too many threads would just create too much overhead, and slow the program drastically compared to having just one thread as MP4 does, with a bunch of request channels. Unfortunately, I cannot test that thought thoroughly because going over 100 threads/ channels could start getting to the point of the OS's limits as far as

number of files open. I don't believe there is much wrong with my MP4 code though, and I think if anything, something was simply wrong with my MP3 code (as like I said, it doesn't fully terminate everything prior to outputting its calculated run time). Or at least, something is more wrong with my MP3 code than in my MP4 code.

Thus far, we have seen that for MP4, the code does have a serious increase (in the order of about 17 million microseconds) in performance for worker thread numbers from 5 to 30, and so does MP3, and they both start capping off and in the case of MP3 for sure, start getting worse performance with worker threads of 100 or more. I suspect that buffer size will act similarly in MP4 as MP3, and will give slight increases in run time, but probably nothing hugely significant. Overall, it seems that my MP3 code just runs better than my MP4 code for the limitations of the system they are being ran on, but it could easily be a result of faulty thread handling in either of or both of the programs.