Colt Campbell
CSCE 313_504
October 27, 2013

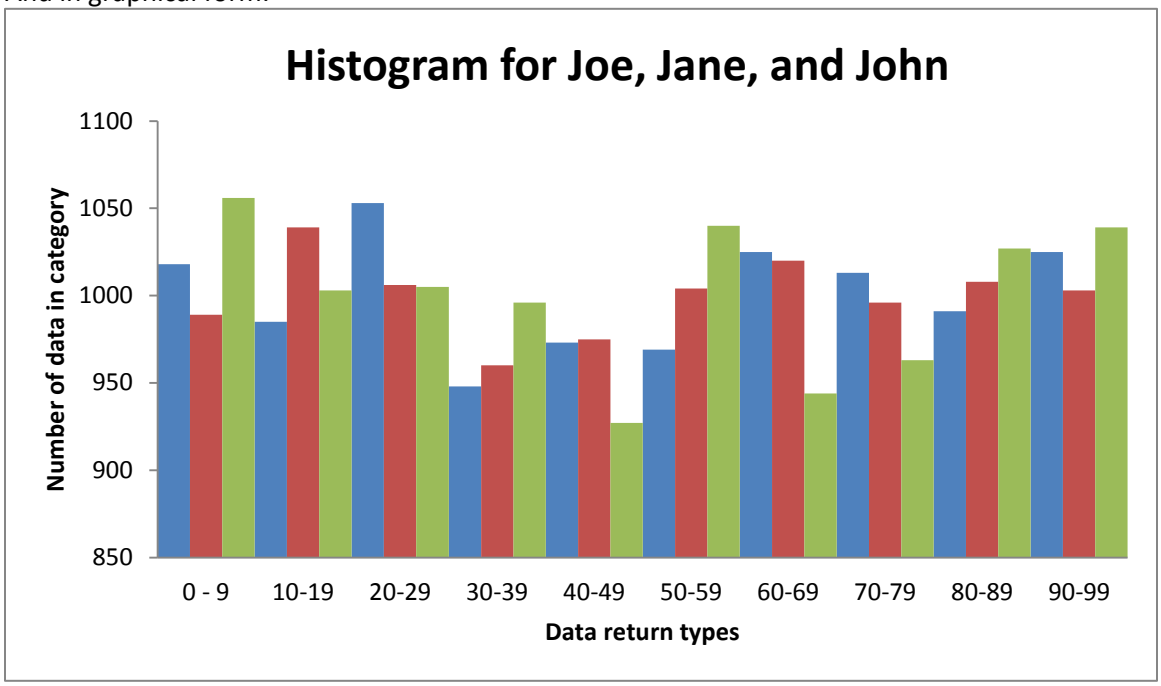
MP3 Report

General Performance Evaluation:

For the program, the general performance seems to be working very well. There were a few things that I did to help improve the performance as far as the histogram and statistics threads go in the form of having all three of the histograms print out at the same time (as shown below), and the way that the threads were implemented, I believe was in tune with how the program should be running. There are several helper functions to make the program more efficient that act as the thread starters and action handlers. The worker thread handler is used to tell a worker thread what to do and then when it is used as much as it should be, it is killed off using the "quit" request send to the channel that is passed as the worker's channel to communicate with the data server. The statistics thread helps with keeping track of the statistics by retrieving all of the data out of the response buffers and putting them into the correct stat threads for each of Joe, Jane and John. The request thread just helps in putting data into a request. Everything is then built up from pthread_create and then destroyed later by joining with the kill thread that uses the manually written kill function telling all of the threads to terminate, essentially. After everything is terminated, memory is 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 (done on the compute.cse.tamu.edu server):

```
----------------------------------------
Data    Joe     Jane    John
0-9     1018    989     1056
10-19   985     1039    1003
20-29   1053    1006    1005
30-39   948     960     996
40-49   973     975     927
50-59   969     1004    1040
60-69   1025    1020    944
70-79   1013    996     963
80-89   991     1008    1027
90-99   1025    1003    1039
----------------------------------------
```

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.

In all, it does not seem to have a specific pattern of returning which number to whom, except for the fact that all of the returns average around 1,000 per each number range per person. And in total, each person always gets 10,000 number returns (their returned data as is supposed to happen). And since there are 10,000 requests, and each number range has 10 numbers in it, and there's an average of about 1,000 counts for each number range, it is safe to say that each number appears about 10 times per person when the request size is 10,000 per person and there are a total of 100 different numbers to return. I believe this means that the program is working as it is supposed to work. The performance of the system could be improved by adding the worker thread creation into one thread (thereby having less overall threads to deal with). Aside from that, the program would be difficult to further improve. The code is compartmentalized through the use of the functions that are used for pthread_create to start the threads and it is decently fast with near linear run time. The usleep command at the end of the main function does skew the recorded performance time a bit but it is required in order to help make sure the termination of all of the channels and threads completes.

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 10 worker threads. The time it took in microseconds to completely handle all 10,000 requests in microseconds as well as statistical data on the distribution of numbers is shown here (taken directly from the terminal):

B = 500, W = 10:

10000 requests done in 11,991,635 microseconds.

(inserted commas after for better readability)

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 20, instead of 10 (an increase in worker threads). The output of that run is here:
B = 500, W = 20:
10000 requests done in 6,580,615 microseconds.
(again, inserted commas after for better readability)

Clearly, the time went down about 5 million microseconds by simply increasing the amount of worker threads per person to 20 rather than 10. So, if we increase the number of worker threads, it should become faster because more threads are handling more information at once, allowing the 10,000 requests to get done relatively quickly.
To further support this claim, I will continue increasing the worker thread amount by 10 until we hit W=60.
B = 500, W = 30:
10000 requests done in 4,749,476 microseconds.

B = 500, W = 40:
10000 requests done in 4,358,251 microseconds.

B = 500, W = 50:
10000 requests done in 4,196,508 microseconds.

And as we can see, without even doing another run for W = 60, the worker thread amount increasing began to stifle out really increasing performance of the overall program after about 30 worker threads for each person. At the point of worker threads being 30, the program is getting to where the number of worker threads is no longer seriously increasing performance because there are too many threads to handle concurrently (the sheer number of threads actually creates overhead). So, we do want a high number of threads, but not too high, so that the program stays at optimal performance. But, it is interesting to note what happens when I change the buffer size with constant number of worker threads.

B = 600, W = 30:
10000 requests done in 5,269,146 microseconds.
B = 700, W = 30:
10000 requests done in 4,789,567 microseconds.
B = 800, W = 30:
10000 requests done in 4,770,807 microseconds.

As a result of these runs, it appears that with increasing buffer sizes the performance very slightly increases as well, but not to a significant extent. So, with the optimal 30 worker threads, and a good buffer size of 500, the program can keep at peak performance.

Buffer size and worker thread increases do seem to have an effect on the performance of the system. The number of worker threads increasing does have a point to which the performance of the program will actually stop increasing because of limitations of the system hardware and concurrent function (probably like the same idea behind have more than about ten cores in a computer can begin to actually slow down performance rather than speed it up).