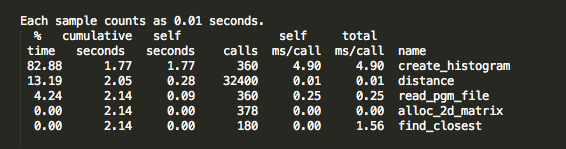
## Implementation Details & Gprof Profiling Outputs

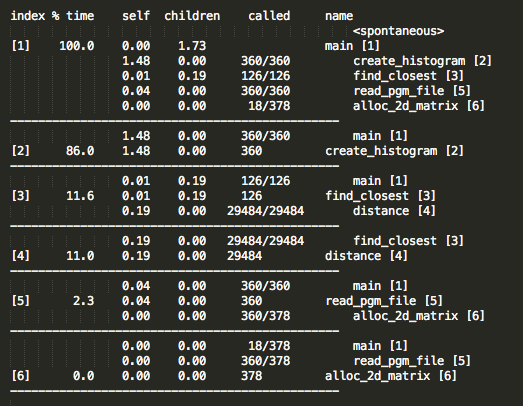
Program reads each training image into a two dimensional array and creates its histogram. After creating histogram, there is no need for image itself. Then it does the same thing for all test images. Histogram for each test image is compared to training samples and person who has the closest match with test histogram is given as predicted result.

Gprof result for “./lbp\_seq 10” is shown here:



Time column show total percentage of running time spent by this function. Self seconds shows total time in seconds spent by each function. Cumulative seconds is total time of a function and all the functions above it. So cumulative seconds column of top method is overall running time of the program. self ms/call column shows running time in milliseconds of respective function for each invocation. Total column includes running time per invocation of descendants of that function. Here self and total columns are equal for create\_histogram because create\_histogram does not call any other method. It uses all specified time itself.

This shows that “create\_histogram” takes most of the running time. We analyze call graph to learn relationship between these functions.



main function calls create\_histogram, find\_closest, read\_pgm\_file and alloc\_2d\_matrix functions at different times. Create\_histogram function is called 360 times and it took 1.48 seconds in total. Read\_pg\_file function takes second place in running time with 0.04 seconds. Find\_closest does not take much time itself but some descendants that it invoked took 0.19 seconds in total. This is a significant cost. We can look at third row to learn which function takes 0.19 seconds. Apparently it is distance function. Distance function is called 29484 times. It is a good candidate for being parallelized as its runs are independent and it is called a lot of times.

We can see that “create\_histogram” method uses 86% of running time and it does not call any other function at all. Therefore it is clear that we need to parallelize execution of this method.

“create\_histogram” is called at two places in the program. First when reading training samples and second when reading test images. I added “#pragma omp for” in the beginning of for clause which calls “create\_histogram” for training images. This pragma runs the for clause in separate threads. Each thread takes a portion of index space. In this program, this for iterates “k” times. Therefore the for with pragma iterates approximately “k/number\_of\_threads” times for each thread.

I used parallelized for with reduction option where I read and predict test samples. Required processor string is “#pragma omp for reduction(+:correct\_answers)”. This statement reduces individual “correct\_answers” variables computed in each thread to one variable in master thread. One “correct\_answeres” variable for all threads is not feasible because addition operation is not atomic. Race conditions may have caused wrong computations. Therefore to prevent this situation, either specifying addition operation as atomic via “#pragma omp atomic” or reduction operation after parallelized for is necessary. I used the latter because a large number of atomic operations seriously hurts execution performance.

Additional parameters for “pragma omp for” are “schedule, ordered, private, firstprivate, lastprivate, shared, reduction, collapse and nowait”

## Accuracy of Results

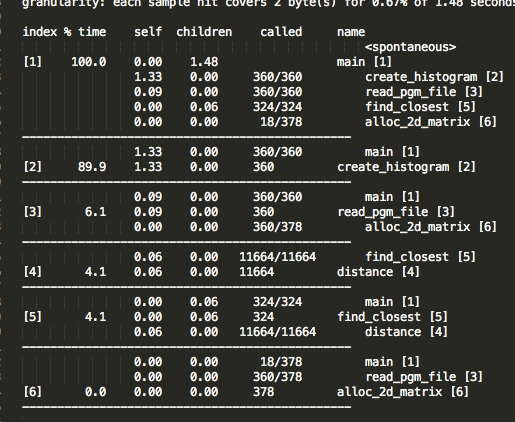
## Execution Times

Training data size versus execution time

Below is execution time of parallel algorithm with different number of threads. As you can see parallel version with 1 thread is slightly slower than serial version because threading brings its own overhead. However when we increase number of threads execution time decreases very fast.

## Discussion

After applying parallelization, following gprof results are received for the same task of which gprof results were discussed above.



As can be seen from call graph, create\_histogram is significantly faster in parallel version. Also time taken by distance function is reduced to 33% percent of its previous value. Parallelizing creating histograms brings significant time reduction because it is computation intensive job with no dependencies between jobs. Also parallelizing reading from files is a good idea because disk input output is a slow operation. While waiting for a file operation to respond, program can initiate other file operations. At the end we reduced program execution times to 50% of its serial execution time. This can be seen from execution time graphics mentioned above.