Alper Şahıstan
21501207

# CS426-Parallel Computing Project 3 Report

## 1. Implementation Details

**For the serial part** the three functions that were specified in the assignment were implemented. Those are **create_histogram**, **distance** and **find_closest**. The main function first reads in the k value as the input. Then training phase begins where for each person's first k picture histogram of size 256 integers is calculated. This calculation is done with **create_histogram** function. After training, test begins where for each person's remaining 20-k photos.Testing is done with distance function being called for each histogram in the training data. The closest distance encountered returns that persons id as a prediction for that test image.
The **create_histogram** function uses 3 embedded for loops. First 2 for loops iterate over the image the most inner one uses an static int array to calculate histogram index that is to be incremented. The static array is type of int and holds 16 values(8 pairs) that correspond to clockwise offsets from the center when taken two-by-two. As Figure-1 illustrates this inner for loop is applied in a sliding window fashion without any border of the window going out of the photo.

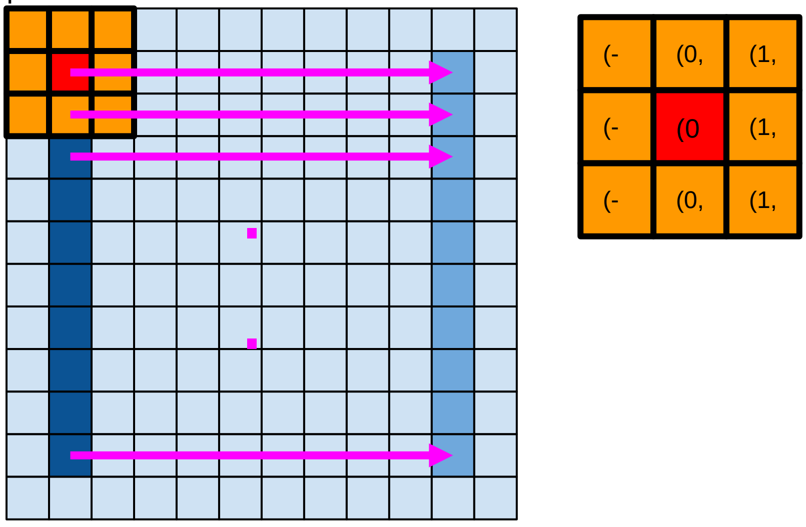


**Figure-1** Sliding window over the photo

The **distance** function uses a single for loop to evaluate distance between two histograms. Using the formula:

$$\sum_{i=0}^{d} \frac{1}{2} \frac{(a_i-b_i)^2}{a_i+b_i}$$

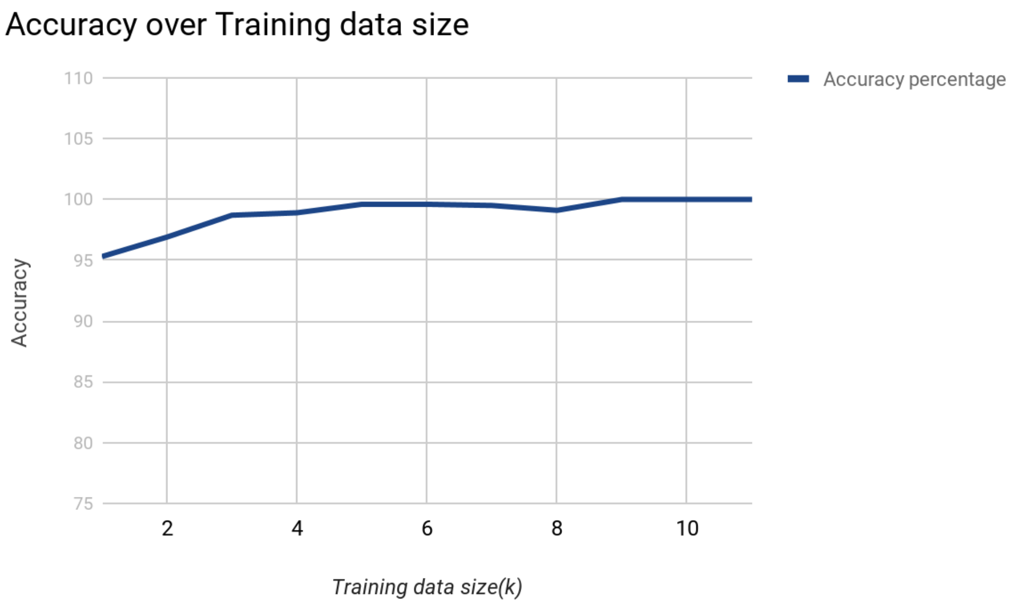
The **find_closest** function utilizes two embedded for loops to go over all of the training data histograms comparing a single test image histogram using **distance** function. The index of recorded min distance is returned.

**For the parallel part,** same code is used with some openmp directives. After extensive profiling, data indicated that nearly 87% of the execution time was create_histogram function. Thus **omp parallel for collapse(2)** was applied to outer two for loops enabling parallel execution of those for loop iterations. Most inner loop was not paralyzed since it only contained an access to a critical section that requires thread private calculation of histogram index. Incrementation of histogram index and calculation of the index are protected from race condition using **omp atomic** directive. **omp parallel for** was applied again for the distance function yet it's inner formula calculation updates a thread shared variable so it was protected with **omp atomic** directive. Finally, the find_closest function's for loops where parallized with **omp parallel for collapse(2).** Again to avoid race conditions the updates to the thread shared min distance value and it's person id were put under a **omp ciritcal** directive to denote critical section. This section includes the if clause that checks the current distance to the global min value.

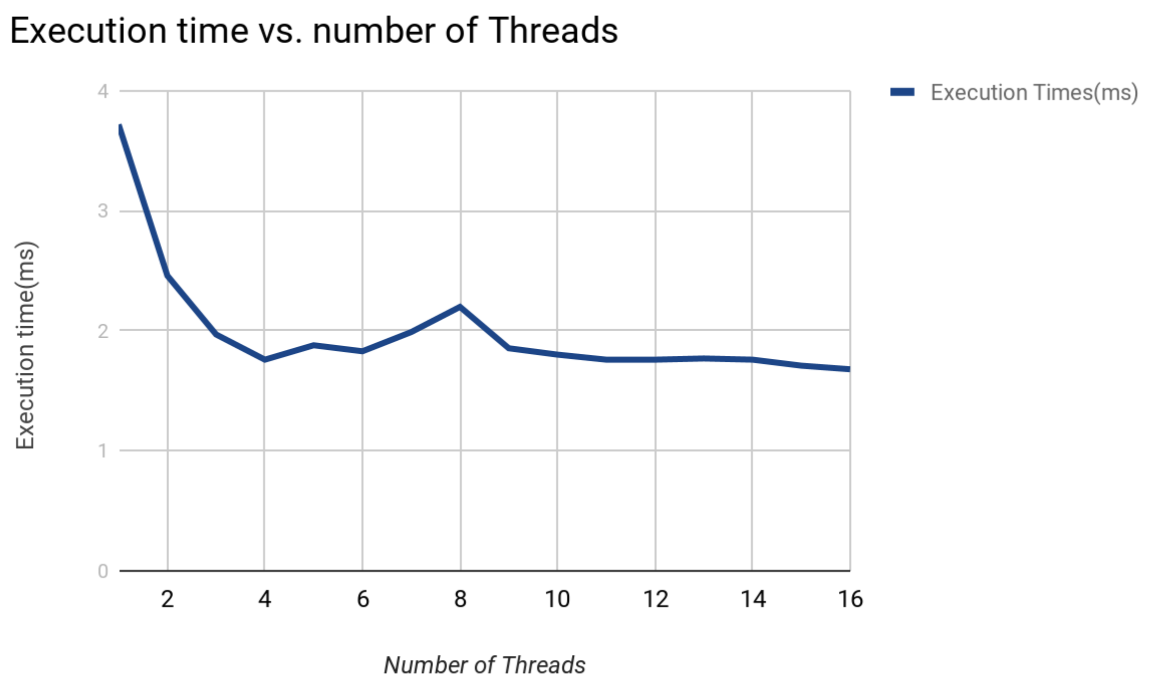
## 2. gprof Profiling output Details

As profiling of sequential code indicates approximately 87% of the time is taken by create_histogram function, approximately 5% of the time is taken by the distance function and again approximately 5% is file reads.
When one looks at the omp version of the code's profiling results nearly %97 of the time taken for each separate execution is in the main function. This is true for all number of threads with all of the different k values.

## 3. Benchmark Results



To display the data points better the graph starts from 75% rather than 0%.



## 4. Discussion

When we look at the profiling results for sequential code create_histogram, distance and file reads take most of the execution time this is due to dense for loops in the create_histogram function and its frequent calls from main the main function.
But when we look at the parallel profiling data we see that these percentages shifted to main function. This might indicate that parallelizing create_histogram and other function calls are effective on a scale such that main function that calls those functions take more time than the actual functions. As the in the 3.benchmark results indicate parallelizing this application returns some speed up of 2.21
As the second graph indicates most significant speedup is at 4 threads. This is due to number of cores that machine has that the tests were conducted on. Until 8 threads there is a curve with local minima at 4. After 8 threads one can observe that the increasing the number of threads will not achieve significant speed up. This is again due to number of physical and logical cores of the machine.
When looked at histogram results for accuracy one can observe that there is a fast climb to 98 percent at k = 5 but after k = 5 the accuracy barely increases just like a logarithmic function. At k = 9 accuracy hits 100% and after that it goes on as 100% since we have enough train data to compare with tests that is accurate.