Cs426 - Project 3

Alba Mustafaj

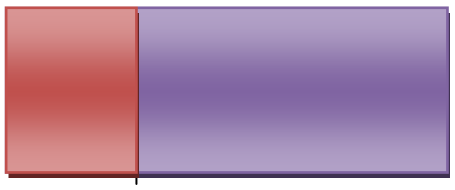
21500009

Table of Contents

1.1 Serial Part

1.1.1 Implementation Details (C++ used)

To avoid extra allocation and deallocation I have declared a 3D array for all the histograms including training and testing one . and initialized all its elements to 0. The testing histograms are specified latter by k, which will serve as the boundary index for the testing and training histograms. For instance, if k = 3, the red part of the below rectangle represents the training histograms part and the purple are the testing ones.



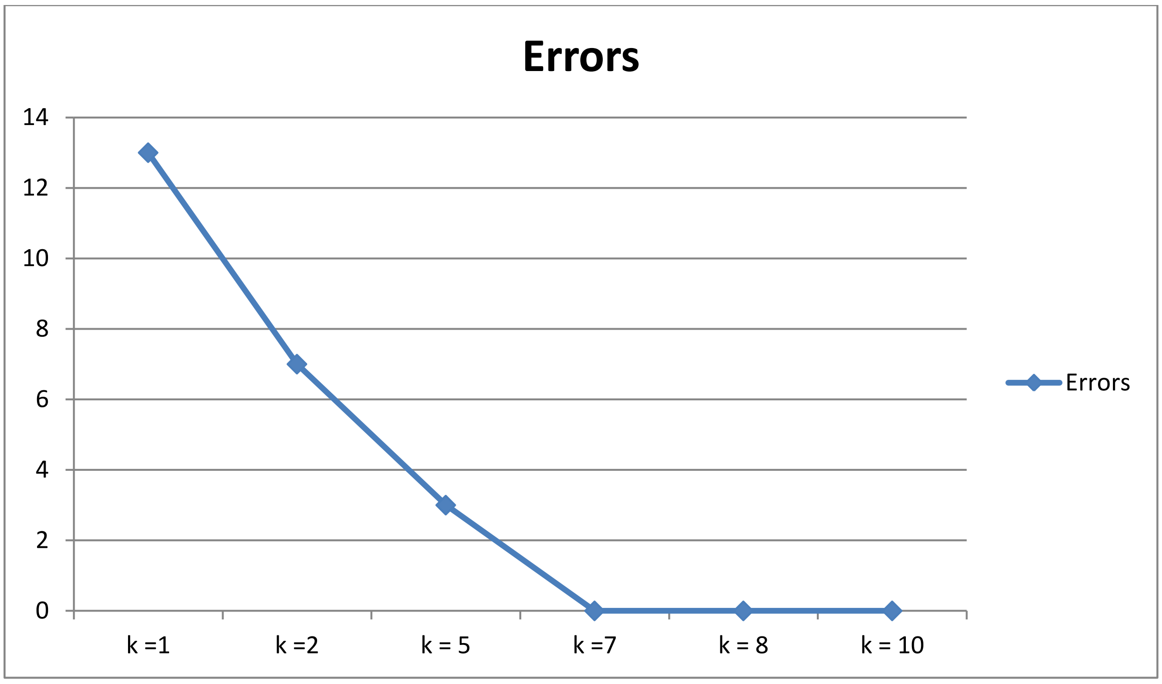
For reading all the files I use a double for loop, and assign a string accordingly to obtain the right names (i.e 1.1.txt) which are passed to read_pgm_file function. In order to obtain an easier calculations I allocate another matrix with indexes 2 greater than the image so that I initialize the borders with 0 and put the data of the image inside it. This enhanced matrix is passed to create_histogram function which generates a histogram for each of the images and is implemented as following:

I use a smallMatrix[3][3] to temporarily store the numbers 1 and 0 before making them a decimal. Then, in 2 nested while loops where I control the numer of rows and columns, I make the necessary comparisons between the current pixel and its surroundings and store the 1 or 0 into the smallMatrix. After the comparison, the number is converted into decimal and I add 1 in the 3D array at index represented by the decimal number I obtained (x and y indexes for the 3D array are specified by counters in the while loops) .

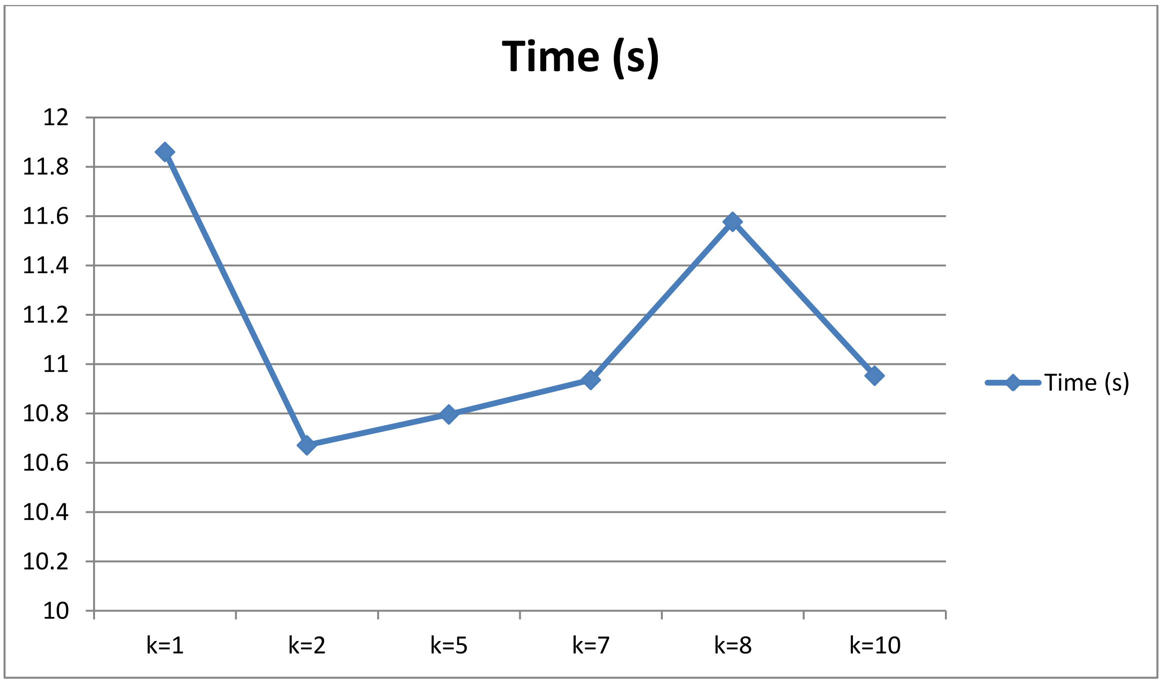
I deallocate the images and the enhanced image matrix before exiting the 2 nested for loops as they are not needed anymore and I will work only with the histograms. In another pair of for loops where the outer is bounded by the number of Ids and the inner starts from k and is bound by the numberOfPictures for 1 person (20) I keep the count of number of tests and the correct results and I call find_closest .Here the tests array is the chunk of the above mentioned 3D holding all the histograms. This chunk has i (the outer for loop count) and k (the index where test histogram starts) for indices. find_closest function is implemented as follows:

I make a 2D array which will contain the comparison results and I populate it by calling distance function (has mathematical calculations as described in the project pdf) inside 2 nested for loops which keep track of the id and the position the comparison value is put corresponds to the person with the same id. Furthermore, I find the smallest value in the 2D array holding the distances and return the Id to which it corresponds. Before returning I deallocate the array as it is not needed anymore. After this process is completed for all test histograms, I return the results and deallocate the 3D matrix.

1.1.2 Graphs



Errors
14
12
10
8
6
4
2
0
k =1
k =2
k = 5
k =7
k = 8
k = 10
Errors



Time (s)
12
11.8
11.6
11.4
11.2
11
10.8
10.6
10.4
10.2
10
k=1
k=2
k=5
k=7
k=8
k=10
Time (s)

1.1.3 Gproof & Discussion

As depicted in the Errors graph, initially for k = 1 there are 13 errors and as k increases testing gives better results (for k = 5 there are 3 errors out of 270 tests). Furthermore, for k >= 7 there are no errors which demonstrates that the training set is enough to determine accurate results.

From the Time graph it can be deduced that the time variation for the serial part is approximately 10-11 seconds with small variance of 1-1.5 seconds (when k =1) . These difference in results can be attributed to comparison time (bounded by number of tests) as it is different for different k, as well as rounding errors in the time function. Furthermore, the similarity in the results can be attributed to the time taken to compute the histograms, which requires dynamic allocation and comparisons in order to obtain the decimals which are put inside the histograms (this process is the same for all cases regardless of k).

Considering the results of the Gprof, the following can be said about the implementation:

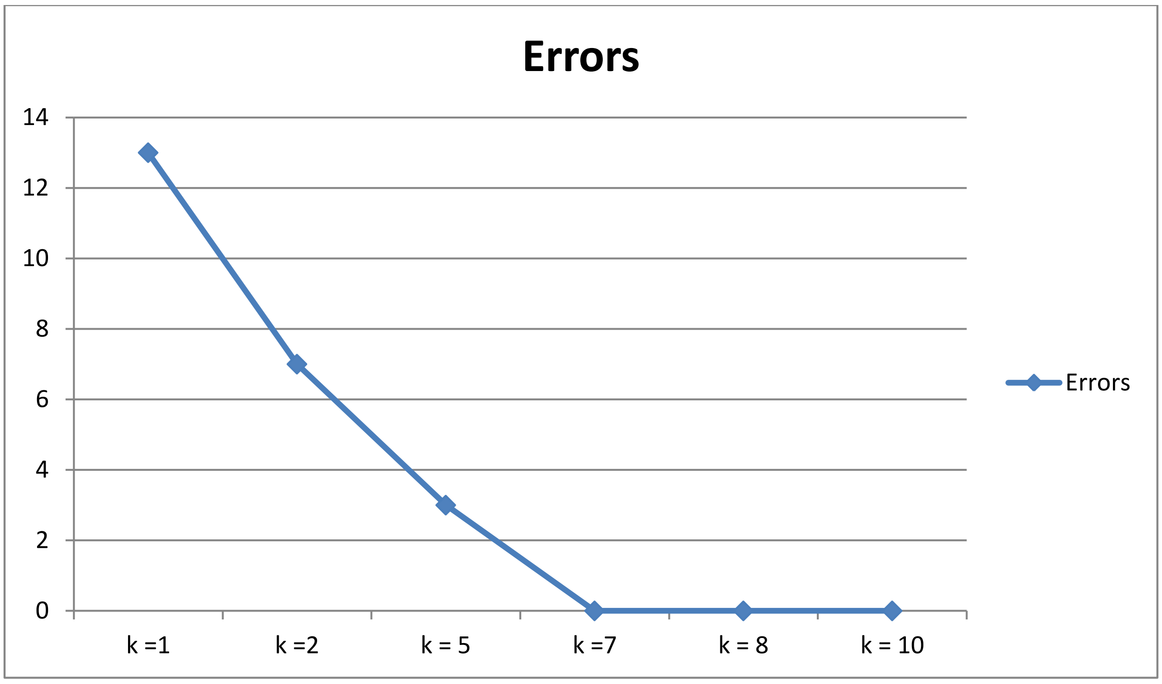
As stated above from the graph results, the time elapsed create histograms is the same in all cases. Concretely it takes 24.88% of the time and is called 360 (18*20) to make the histograms. Cumulative seconds corresponding to it are 3.33. Calling main which is done 1 time takes 4.06 cumulative seconds, thus 3.35 % of the time. The function for reading from the files into arrays is executed 360 times and takes 1.67 % of the total time which is 4.13 cumulative s. Distance and power function whose calls vary for different nr of 4 have their own part in the overall time. The other functions (deallocate, allocate, find closest) do not take a significant % of time, thus the issue is with the above mentioned functions. A good optimisation would be in parallelising the part of creating histograms, comparing, reading the files and finding distance between vectors.

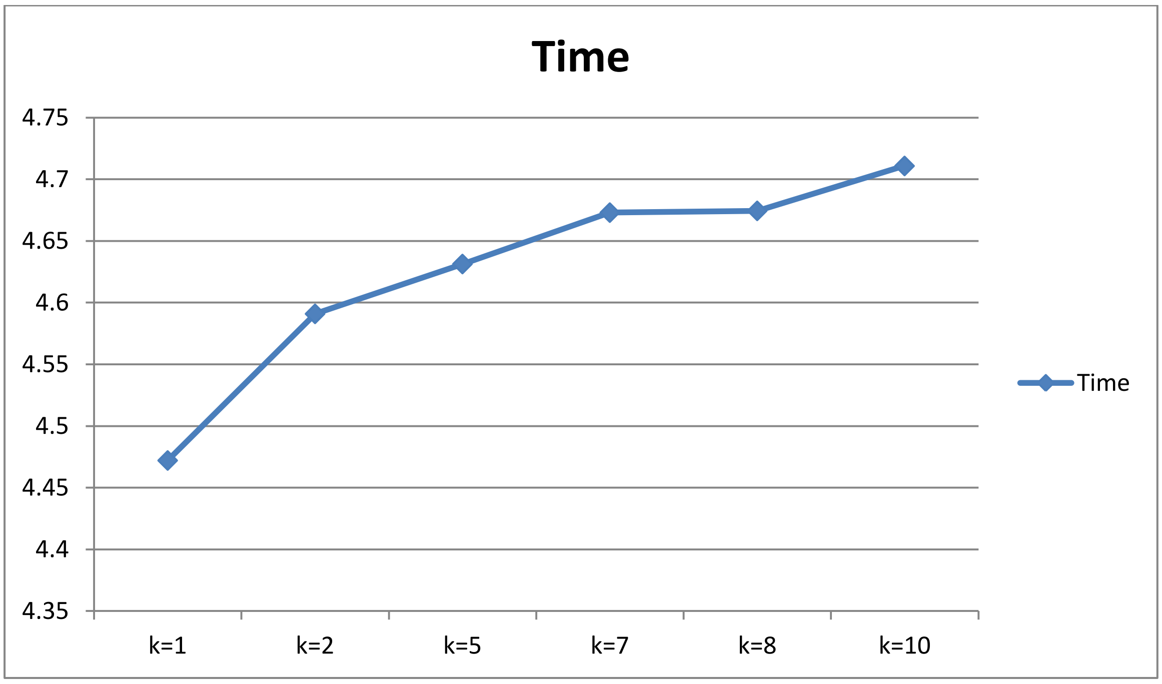
1.2 Parallel Part

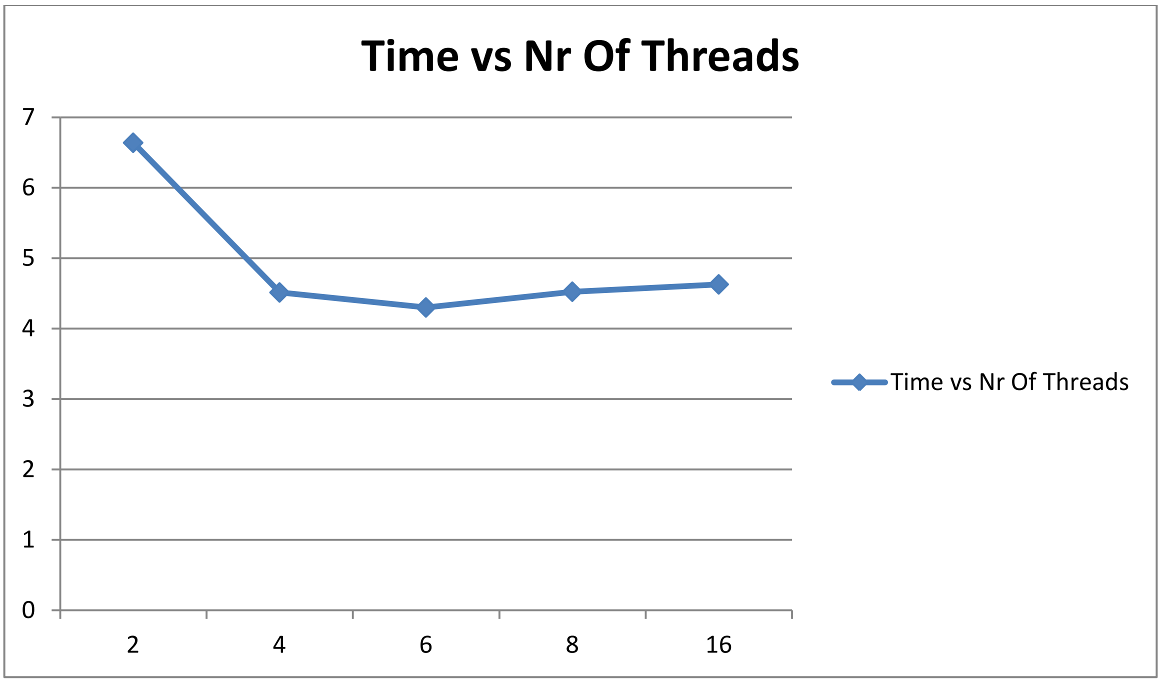
1.2.1 Implementation Details

Being that there was no harmful data dependency which could affect badly the program when trying to parallelise the tasks, no change was made to the overall logic and implementation explained in the first section. The parallel statements were put in the tasks which were most time consuming, which was clear after doing gproof part. Thus I put pragma omp's as following: Firstly, an optimisation was needed for reading the files as that function is called 360 times, thus a parallelising statement was put before the two nested loops inside which image read and enhanced image matrix was being created. Another important issue within those loops is creating the histograms which is as well called 360 times. Being that together in the serial implementation they take approximately 26.5% of the time, in the parallelised version an obvious improvement will be obtained. Furthermore, another #pragma omp for is put before the 2 nested for loops inside which the find_closest function is called which as well calls distance function. Although from the gproof results they don't take a substantial % of the overall time elapsed, there is improvement in the performance. A more clear view of the improvement is depicted in the graphs.

1.2.2 Graphs


Errors
14
12
10
8
6
4
2
0
k =1
k =2
k = 5
k =7
k = 8
k = 10
Errors


Time
4.75
4.7
4.65
4.6
4.55
4.5
4.45
4.4
4.35
k=1
k=2
k=5
k=7
k=8
k=10
Time



1.2.3 Gproof & Graph Discussion

As illustrated in the errors graph, the performance in terms of accuracy remains the same, the program runs correctly and the number of errors is the same as in the serial part. Regarding the Time graph, the time elapsed for different k's for the same nr of thread is observed. Concretely, results increasing slightly between 4.4-4.75 were obtained and as k increases the time elapsed increases as well. This can be attributed to a bigger number of training histograms each of which is to be compared with a test histogram, thus leading to a big number of computations. Logically being that less tests are done when k increases, the time should decrease, however the compare function does not take a considerable amount of time. Thus, this slight variation in time can be attributed to small errors in time calculating functions, the terminal and the pc.

Regarding the graph Time vs Nr Of threads, these tests corresponds to k=1. Initially for 2 threads a bigger time is required for the program and as the number of threads increases, the time elapsed decreases (as shown in the graph). However, for a large nr of threads (i.e 14, 16) a different behaviour was observed as the time elapsed increases slightly rather than being smaller than that for less nr of threads used. This can be due to rounding errors as well as the big number of threads is no longer required as the necessary optimisation is already done. To have a better insight on these results the following analysis of gproof and putting #pragma decisions is made: Being that the distance function is called in each comparison and it itself contains many calls of the power function which take a considerable time of time, I have inserted a parallelising statement before the for loop inside which the distance is computed for the histograms, thus making an improvement from the serial program. Furthermore, a pragma put for the foor loop which initializes the 3D array where the histograms will be stored speeds up the initialization of the array elements with 0. Another important part is reading from files and creating the histograms, thus a pragma is inserted there to speed up the process of creating the histograms. Although allocation and deallocation of the 3D matrix do not require a significant amount of time, I have also parallelized that task. Regarding the gpfrofs, I run for 2,4,8,16 threads and obtained the corresponding outputs. Analysing the calls made to the functions, especially create_histogram and read_pgm_file, the results for different number of threads were varying in a not clear manner. Being that gprof doesn't understand the intricacies of openmpi, these results can be attributed to this fact, as gpfrof gives the results without the accuracy that they are being computed in omp. For instance, in the case of 4 threads each thread gets 90 pgm_files to open and populate the matrixes, however gproof doesn't perceive well this behaviour, resulting to inaccurate results. However, by analyzing the cumulative timing in gpfrof txts for different threads, the time % taken by creating histograms reduces from more than 24% to less than 20% and reading the files time also reduces slightly although its % is not significant. However, for threadNr = 16 the gproof timing results are slightly smaller than that of sequential. This can be also seen by the graphs where the timing decreases as the nr of threads increases except of when the nr of threads is much greater (i.e 12, 14, 16).

2.General Discussion & Conclusions

All in all, it can be said that this project is a good example of task parallelizing with omp. Dealing with multiple data to read, compute and compare results in a high amount of time if the tasks are to be run sequentially. The serial version in this case requires about twice the time of the parallel version because in the latter, inserting pragmas for parallelizing the most time consuming tasks as reading from files and creating the histogram for each of 360 images. The performance in terms of correctness in my implementation is 0 errors for k >=7 for both parallel and serial versions. Regarding the time elapsed, the serial implementation takes about 10-11 seconds and the parallel one approximately half of the time. The improvement was obtained by looking at gprof results for the serial part and optimizing it such that different nr of threads take almost equal parts of the tasks and execute them in parallel, thus resulting in a better performance in terms of time.