Parallel Computing

CS 426

Sec 01

Project 3

Syed Sarjeel Yusuf

21402744

Due: 07/12/2017

Serial implementation **3**
Function: create_histogram 3
Function: distance 3
Function: find_closest 4
Function: main 4

Profiling gprof of Serial Implementation **5**
Execution with k = 1 5
Execution with k = 10 6

Parallel Implementation **7**
Function: create_histogram 7
Function: distance 8
Function: main 9

Profiling gprof of Parallel Implementation **10**
Execution with number of threads = 1 10
Execution with number of threads = 4 11

Graph Plots **12**
Accuracy Graphs 12
Execution Times in Varying k 13
Execution Times in Varying Number of Threads 14

Serial implementation

NB: *IMGW, IMGH, NUMOFPEOPLE, NUMOFFILE* are global variables with the first two holding the dimensions of the image matrix and the latter two indicating sample sizes

Function: *create_histogram*

The purpose of this function was to create the histogram of a single image matrix, irrespective of whether the image is a test image or training image. The parameters taken are as described in the project prompt, with the same parameter names.

The result of the function is the *hist* array which is one dimensional and is passed to the function as a parameter. All elements are initially set to 0. The function then iterates through the columns and rows starting from element [1,1] in the matrix, all the way to element [n-1,m-1] of the matrix. This is where n and m are the numbers of elements in the row and columns of the *img* matrix respectively.

The algorithm then sets the $[i][j]^{th}$ element as the middle value and then compares it with all its neighbouring value in the *img* matrix. The neighbours are numbered from 1 to 8 going clockwise, and this is maintained using the *count* variable. If the middle value is less than the neighbor element then another variable, *decimal* is incremented by 2 to the power of the neighbour value. This is directly processing the binary value of the matrix operation to a decimal value.

Then after the decimal value is calculated, it corresponds to the index of the *hist* array histogram. The array thus holds the number of times the decimal, which naturally ranges from 0 to 255, occurs in processing the image matrix. The function thus returns this array.

Function: *distance*

The purpose of the function is to compute the distance vector between the histograms of two images. The parameters passed are the same as that described in the project prompt. The formula used to compute is as in *Eqn 1*.

$$\text{distance(a,b)} = \sum_{i=0}^{d} \frac{1}{2}\frac{(a_i - b_i)^2}{a_i + b_i}$$

Eqn 1.

Since both histograms are of the same length, the function simply iterates over the elements of the histogram from *i = 0* to *i* = |*a*|, and performs the mathematical operations of *Eqn 1*. The final value returned is in the variable *sum*.

Function: *find_closest*

The function is aimed at finding the matching person label of a test image, according to the training set data passed as a parameter. All parameters passed to the function are similar to that described in the project prompt, except the first parameter. The *training_set* parameter is not passed as a triple integer pointer but rather an integer array with three dimensions.

The function iterates over the *training_set* matrix whose dimension is *NUMOFPEOPLE* X *NUMOFFILE*. It gets the histogram corresponding to person *sampPerson* and file *sampImg*, along with the histogram of the test image and passes it as parameters to the *distance* function, which then returns a distance value between the two vectors.

The value returned, perceived as a similarity value, is then compared with the *min* value, which is set to an arbitrarily large value. If the procured similarity value is less than *min* then the variable is set to the similarity value and the *match* person is then set to the *sampPerson* value, indicating the closest label match so far. Eventually at the end of the function the *match* variable is returned.

Function: main

The function first takes the k value from the *argv[1]*, which thus indicates the separation between the training and test datasets. Thus begins the Training phase of the code. In constructing the training dataset the code iterates from the the *i = 0* to *i < NUMOFPEOPLE* and an inner loop *j = 0* to *j < NUMOFFILE - k*. This thus goes through the demarcated training files of all the people present in the sample.

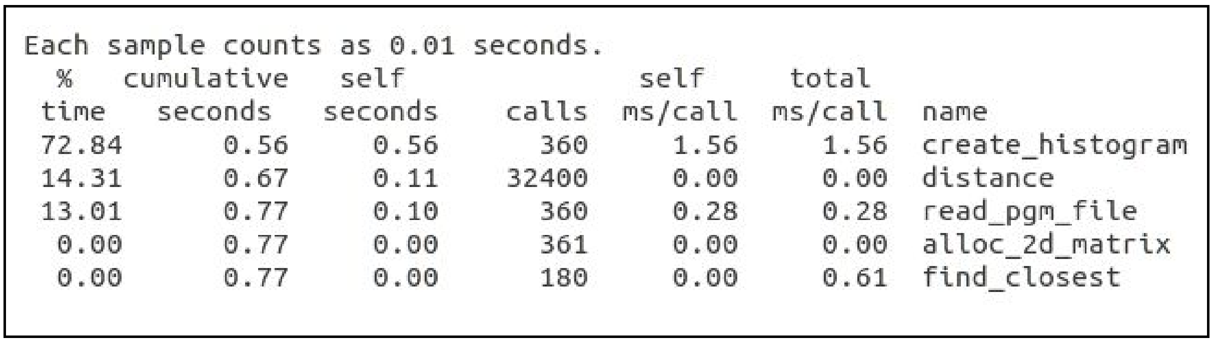
The filename of the training file *j* of person *i* is concatenated using the *sprintf* function, which is then passed into the *read_pgm_file* along with all other necessary parameters. The resultant read matrix is the passed into the *create_histogram* function along with the position in the training data set matrix, *hist*, where the achieved histogram will be stored. This occurs for all *j < k* values in the loop.

Similarly the test data set is also achieved. This is for all *j <= k* values in the iteration. Hence at the end of the Training phase we not only have all the training data processed but also the test data that shall be used. This thus transitions into the Testing phase.

In the Testing phase the iteration is similar to that of the Training phase, but *j* starts from the value of *k*. Thus the respective algorithms from the test data set is retrieved and passed into the function *find_closest* along with all other respective parameters such as the training data set matrix. The value that is returned can be interpreted as the person to which the test image belongs to. This returned value is then compared with the real label of the image's histogram and the accuracy, as variable *acc*, thus calculated accordingly. Finally the time taken, and accuracy are printed in the stated format.

Profiling gprof of Serial Implementation

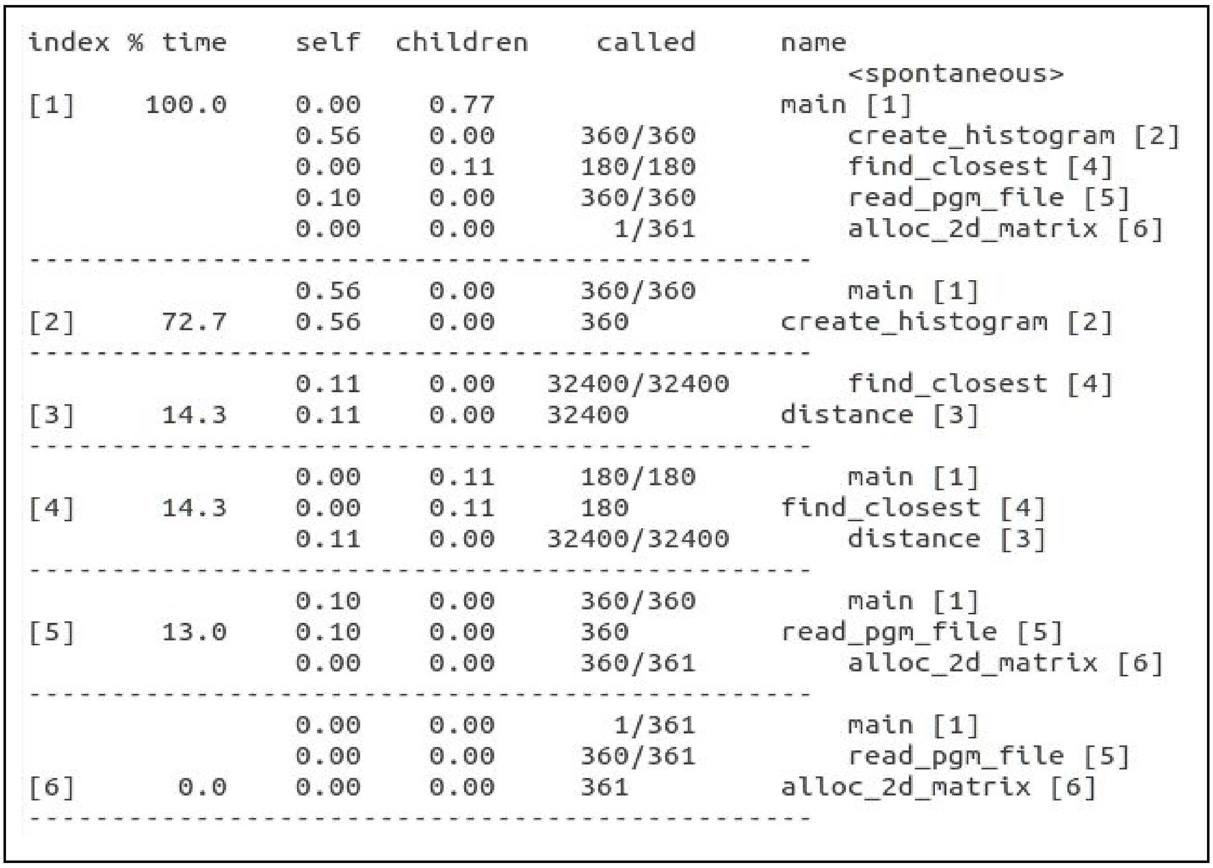
Execution with k = 1



```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 72.84      0.56     0.56      360     1.56     1.56  create_histogram
 14.31      0.67     0.11    32400     0.00     0.00  distance
 13.01      0.77     0.10      360     0.28     0.28  read_pgm_file
  0.00      0.77     0.00      361     0.00     0.00  alloc_2d_matrix
  0.00      0.77     0.00      180     0.00     0.61  find_closest
```

Table 1

As can be seen from *Table* 1, functions *create_histogram* and *distance* are the costliest in regards to runtime, with the later taking more than half of the run time, around more than 72%. The next costly function is the *distance* function with 14.31% of the total time consumption. Hence it can be seen that these two functions should primarily be parallelized. Also the *find_closest* function is the least costliest, out of the other functions, where the other functions are from the util.c file, which shall not be parallelized.

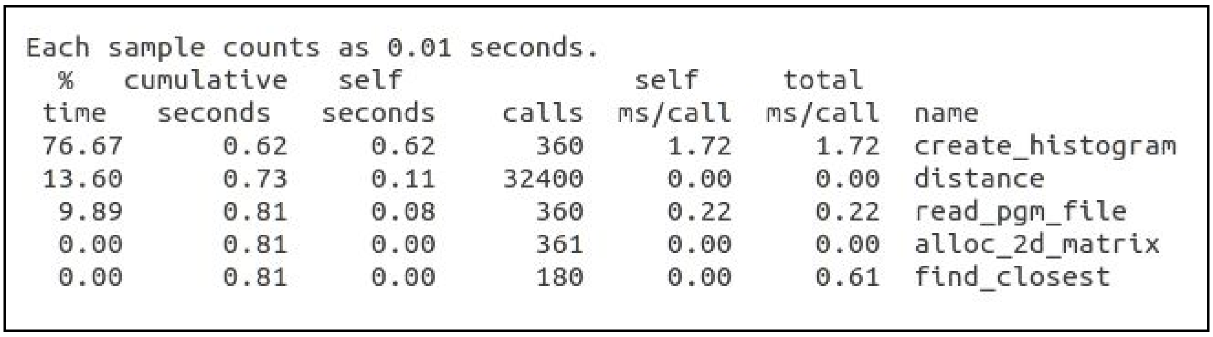


```
index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.00    0.77                 main [1]
                0.56    0.00     360/360         create_histogram [2]
                0.00    0.11     180/180         find_closest [4]
                0.10    0.00     360/360         read_pgm_file [5]
                0.00    0.00       1/361         alloc_2d_matrix [6]
-----------------------------------------------
                0.56    0.00     360/360         main [1]
[2]     72.7    0.56    0.00     360         create_histogram [2]
-----------------------------------------------
                0.11    0.00   32400/32400       find_closest [4]
[3]     14.3    0.11    0.00   32400         distance [3]
-----------------------------------------------
                0.00    0.11     180/180         main [1]
[4]     14.3    0.00    0.11     180         find_closest [4]
                0.11    0.00   32400/32400       distance [3]
-----------------------------------------------
                0.10    0.00     360/360         main [1]
[5]     13.0    0.10    0.00     360         read_pgm_file [5]
                0.00    0.00     360/361         alloc_2d_matrix [6]
-----------------------------------------------
                0.00    0.00       1/361         main [1]
                0.00    0.00     360/361         read_pgm_file [5]
[6]      0.0    0.00    0.00     361         alloc_2d_matrix [6]
-----------------------------------------------
```

Table 2

The call tree of the program can be seen in *Table 2*, which describes how a function calls another function, along with how many times the function was called. As can be seen the main function is delayed greatly by the *create_histogram* function. As can be seen, at most a function is called is 32400 times which is the *distance* function. The value corresponds to width of the image by the height of the image. The costliest function is called 360 times, which is basically the number of people in the dataset multiplied by the number of samples of each person.

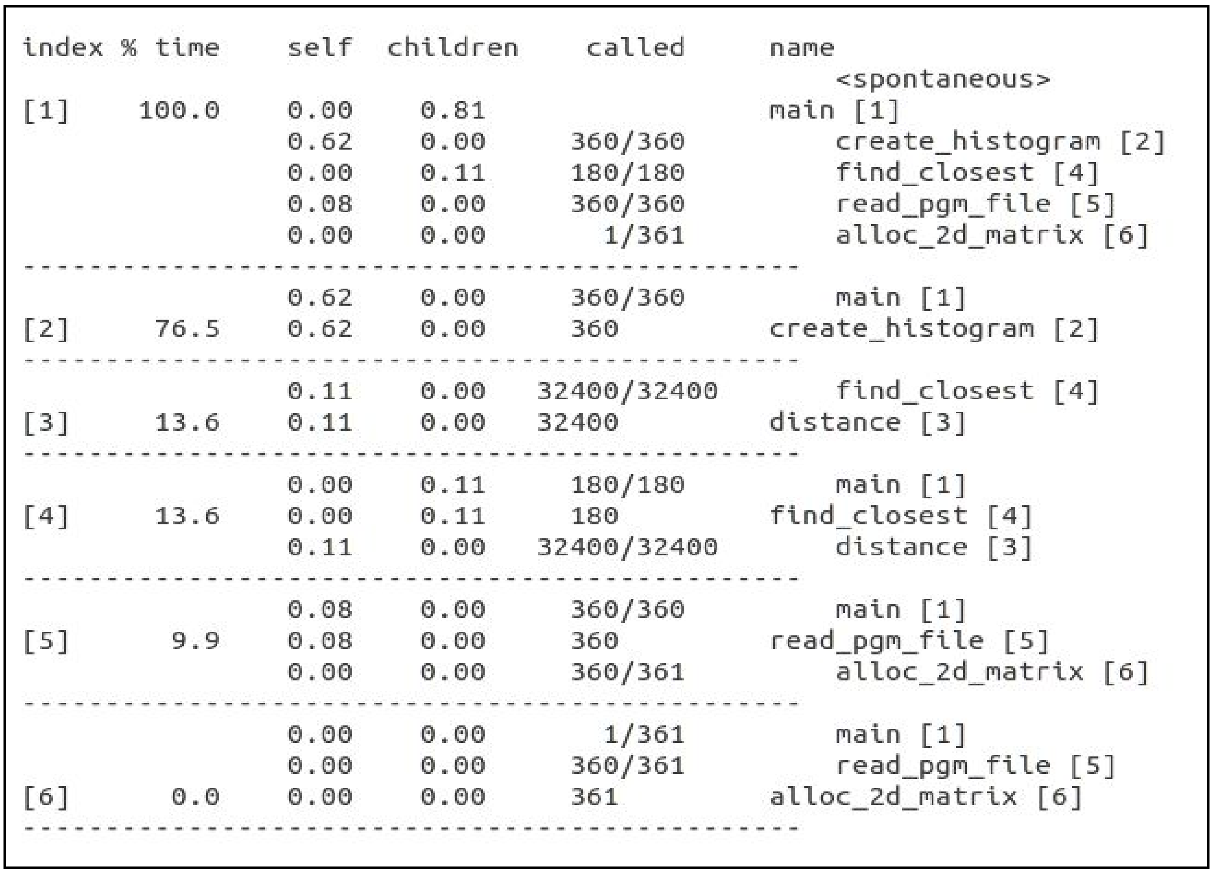
Execution with k = 10



```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total           
 time   seconds   seconds    calls  ms/call  ms/call  name    
 76.67      0.62     0.62      360     1.72     1.72  create_histogram
 13.60      0.73     0.11    32400     0.00     0.00  distance
  9.89      0.81     0.08      360     0.22     0.22  read_pgm_file
  0.00      0.81     0.00      361     0.00     0.00  alloc_2d_matrix
  0.00      0.81     0.00      180     0.00     0.61  find_closest
```

Table 3

As can be seen from *Table 3* similar results have been obtained regarding the time costs of the function to *Table 1*. This reiterates the call for *create_histogram* and *distance* to be parallelised.



```
index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.00    0.81                 main [1]
                0.62    0.00     360/360         create_histogram [2]
                0.00    0.11     180/180         find_closest [4]
                0.08    0.00     360/360         read_pgm_file [5]
                0.00    0.00       1/361         alloc_2d_matrix [6]
-----------------------------------------------
                0.62    0.00     360/360         main [1]
[2]     76.5    0.62    0.00     360         create_histogram [2]
-----------------------------------------------
                0.11    0.00   32400/32400       find_closest [4]
[3]     13.6    0.11    0.00   32400         distance [3]
-----------------------------------------------
                0.00    0.11     180/180         main [1]
[4]     13.6    0.00    0.11     180         find_closest [4]
                0.11    0.00   32400/32400       distance [3]
-----------------------------------------------
                0.08    0.00     360/360         main [1]
[5]      9.9    0.08    0.00     360         read_pgm_file [5]
                0.00    0.00     360/361         alloc_2d_matrix [6]
-----------------------------------------------
                0.00    0.00       1/361         main [1]
                0.00    0.00     360/361         read_pgm_file [5]
[6]      0.0    0.00    0.00     361         alloc_2d_matrix [6]
-----------------------------------------------
```

Table 4

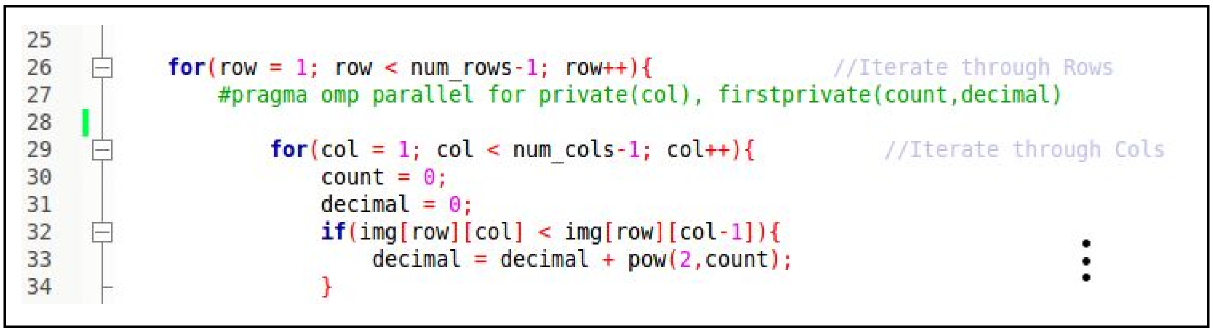
Parallel Implementation

The following functions were parallelized:

- *create_histogram*
- *distance*
- main

Function: *create_histogram*

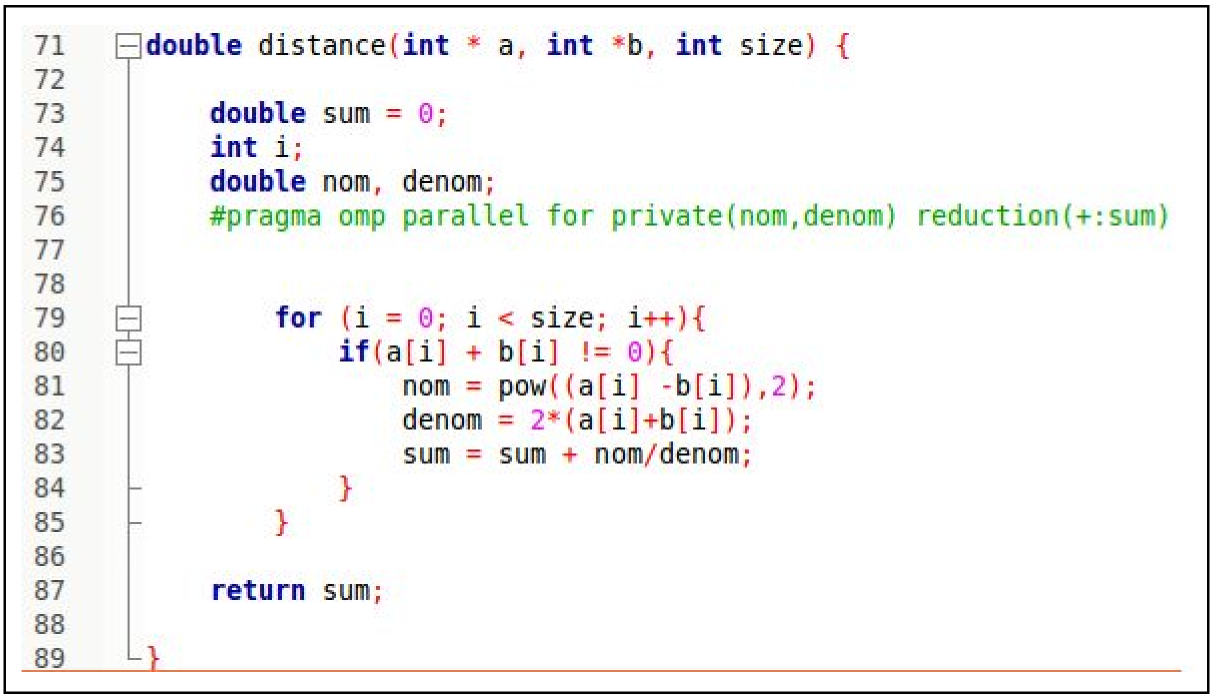
From *Fig 1,* it can be seen that the pragma implemented parallelize the *for* loop that iterates through the columns. Hence the *col* variable is kept private, and the *count* and *decimal* variables are made firstprivate so that they take their initial value in each thread. Since in the serial implementation, the two *for* loops would normally iterate rowxcol times, the parallelization reduces the number of iterations required to technically rowx1, where the pragma splits the *col* iterations upon the threads.



```

    for(row = 1; row < num_rows-1; row++){              //Iterate through Rows
        #pragma omp parallel for private(col), firstprivate(count,decimal)

            for(col = 1; col < num_cols-1; col++){          //Iterate through Cols
                count = 0;
                decimal = 0;
                if(img[row][col] < img[row][col-1]){
                    decimal = decimal + pow(2,count);
                }
```

Fig 1

Function: *distance*

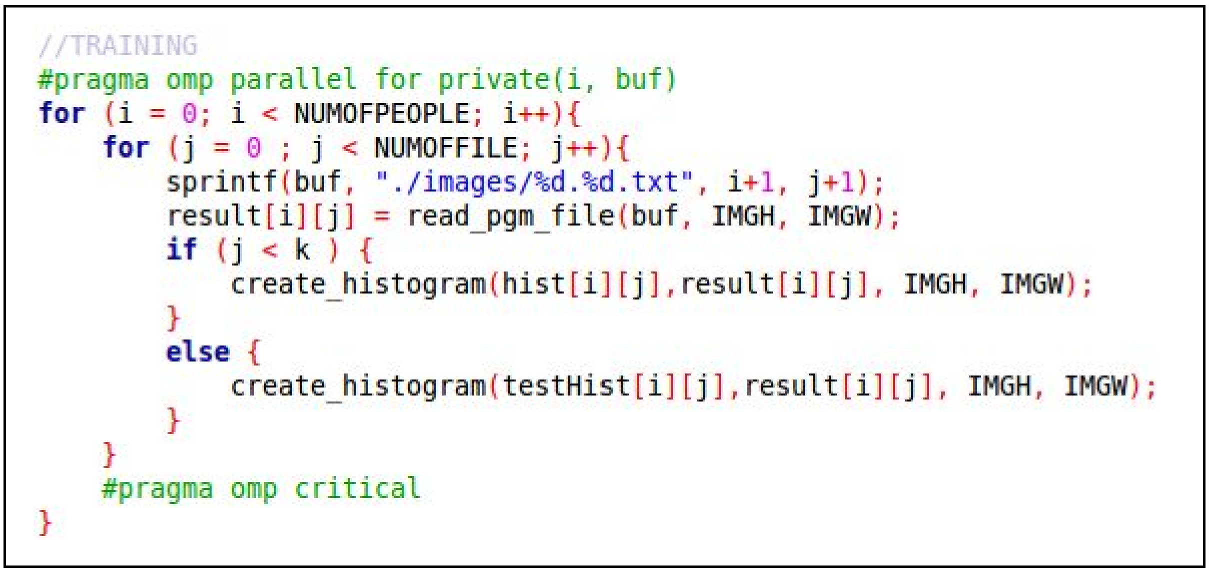


```
double distance(int * a, int *b, int size) {

    double sum = 0;
    int i;
    double nom, denom;
    #pragma omp parallel for private(nom,denom) reduction(+:sum)


        for (i = 0; i < size; i++){
            if(a[i] + b[i] != 0){
                nom = pow((a[i] -b[i]),2);
                denom = 2*(a[i]+b[i]);
                sum = sum + nom/denom;
            }
        }

    return sum;

}
```

Fig 2

As it has been established that the distance function is the second most costliest function, the whole function has been parallelized. This has been done by a reduction parallelization, which splits the for loop onto threads and reduces the overall *sum* variable using the summation reduction pragma. The loop could also have been split, without the reduction, and *sum* kept as shared. However, this may lead to read and write conflicts and thus the reduction pragma is used. As can be seen *nom* and *denom* have been kept private to ensure that their values are not overwritten.

Function: *main*



```
//TRAINING
#pragma omp parallel for private(i, buf)
for (i = 0; i < NUMOFPEOPLE; i++){
    for (j = 0 ; j < NUMOFFILE; j++){
        sprintf(buf, "./images/%d.%d.txt", i+1, j+1);
        result[i][j] = read_pgm_file(buf, IMGH, IMGW);
        if (j < k ) {
            create_histogram(hist[i][j],result[i][j], IMGH, IMGW);
        }
        else {
            create_histogram(testHist[i][j],result[i][j], IMGH, IMGW);
        }
    }
    #pragma omp critical
}
```

Fig 3

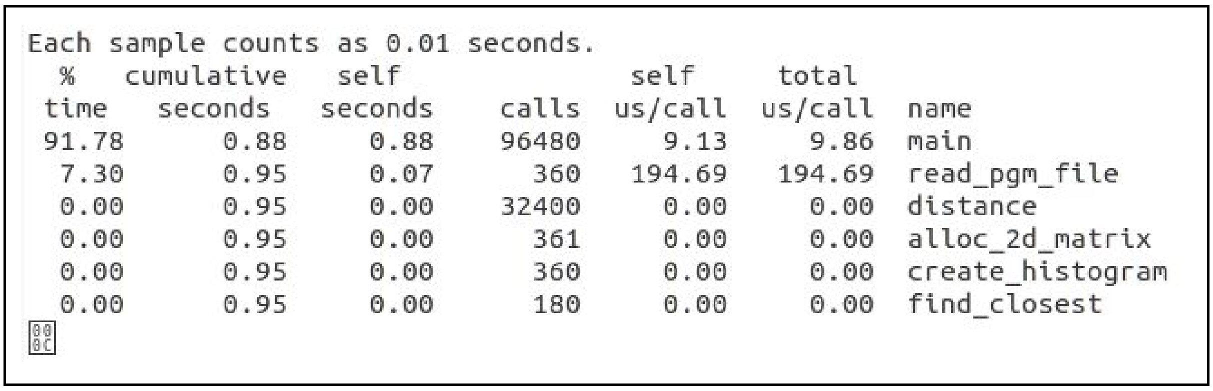
As can be seen from *Fig 3* the pragmas have been inserted in an attempt to optimize the training phase of the program. This implies parallelizing the reading of files and reduce the number of times a single thread calls the *creat_histogram* function which is seen to be the most time consuming function. Therefore a *parallel for* pragma is set to distribute the number of people upon the threads, so that each thread is responsible for reading the files of one or more people. This reduces the load of reading the files of all the people on a single thread. Also the variables *i* and *buf* have been kept private to ensure that they are not over-written and also the matrices being utilized are shared. There is no worry of data dependency or over-writing while creating the training dataset variable *hist* as the *i* and *j* values are distinct on each thread.

The last pragma inserted is the *critical* pragma which ensures that all the threads complete their task before the Training phase begins. This is vital for the program as a successful training phase can only be achieved once the training data is present.

Profiling gprof of Parallel Implementation

NB: Value of k is set to 10 while number of threads are varied.

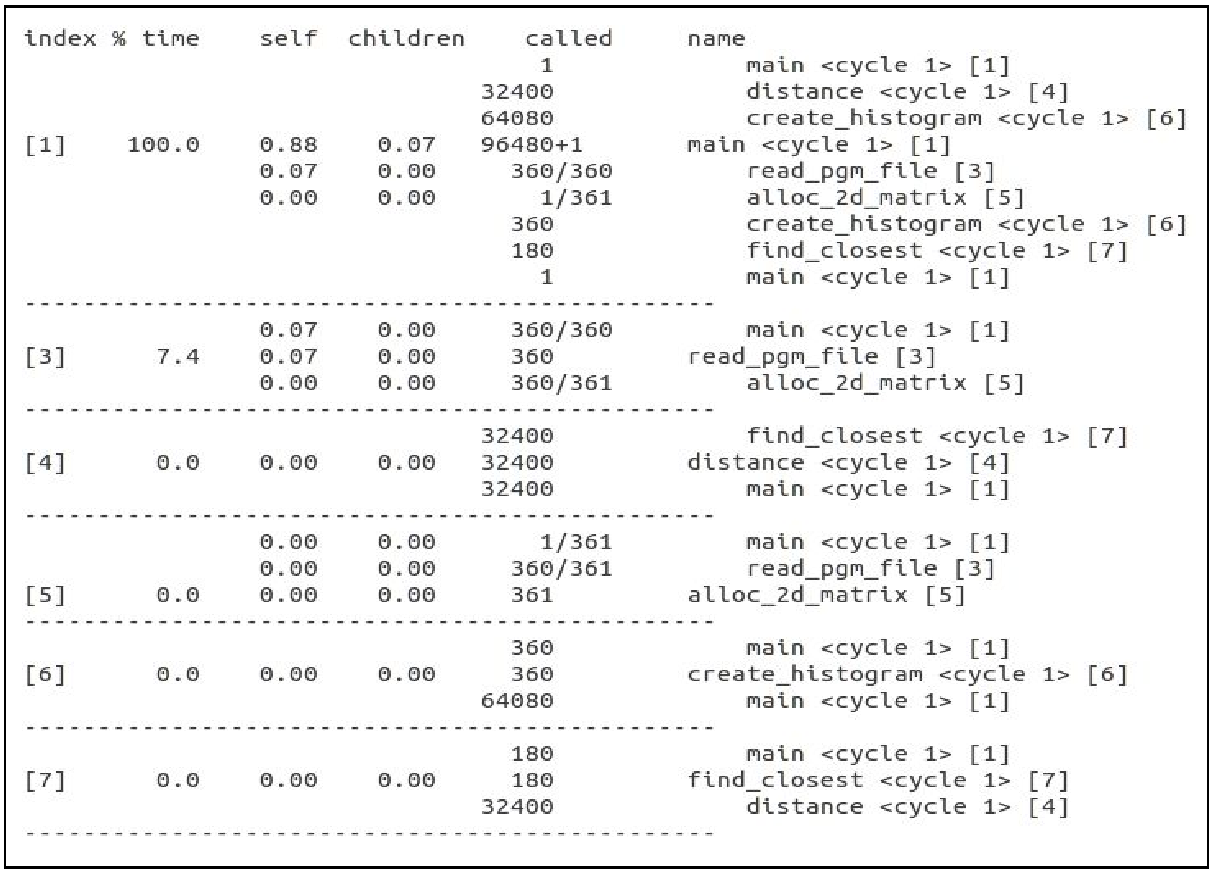
Execution with number of threads = 1



```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
 91.78      0.88     0.88    96480     9.13     9.86  main
  7.30      0.95     0.07      360   194.69   194.69  read_pgm_file
  0.00      0.95     0.00    32400     0.00     0.00  distance
  0.00      0.95     0.00      361     0.00     0.00  alloc_2d_matrix
  0.00      0.95     0.00      360     0.00     0.00  create_histogram
  0.00      0.95     0.00      180     0.00     0.00  find_closest
```

Table 5

As can be seen from *Table 5* The *create_histogram* and *distance* functions are now some of the least time costing functions. On the other hand, the main function has increased in percentage of cost and this illustrates that the reading time and computation time within the main function is relatively high after parallelization.



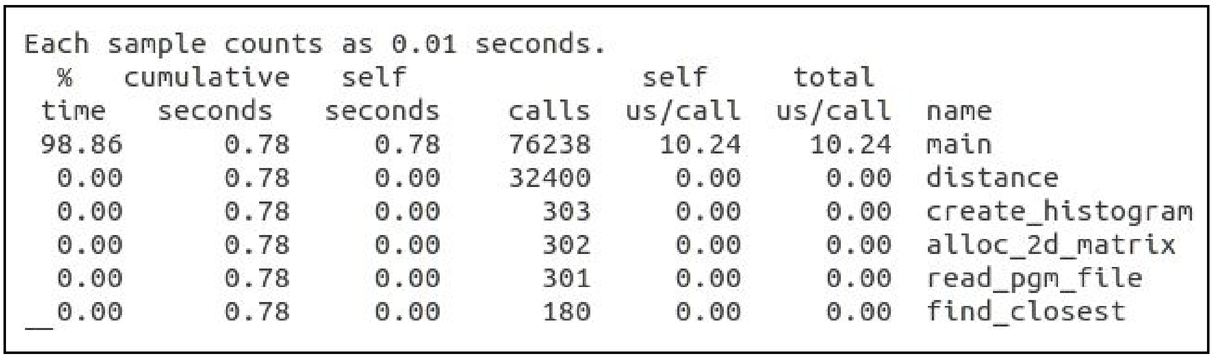
```
index % time    self  children    called     name
                                     1           main <cycle 1> [1]
                                 32400           distance <cycle 1> [4]
                                 64080           create_histogram <cycle 1> [6]
[1]    100.0    0.88    0.07    96480+1      main <cycle 1> [1]
                0.07    0.00     360/360         read_pgm_file [3]
                0.00    0.00       1/361         alloc_2d_matrix [5]
                                   360           create_histogram <cycle 1> [6]
                                   180           find_closest <cycle 1> [7]
                                     1           main <cycle 1> [1]
-----------------------------------------------
                0.07    0.00     360/360         main <cycle 1> [1]
[3]      7.4    0.07    0.00     360         read_pgm_file [3]
                0.00    0.00     360/361         alloc_2d_matrix [5]
-----------------------------------------------
                                 32400           find_closest <cycle 1> [7]
[4]      0.0    0.00    0.00    32400        distance <cycle 1> [4]
                                 32400           main <cycle 1> [1]
-----------------------------------------------
                0.00    0.00       1/361         main <cycle 1> [1]
                0.00    0.00     360/361         read_pgm_file [3]
[5]      0.0    0.00    0.00     361         alloc_2d_matrix [5]
-----------------------------------------------
                                   360           main <cycle 1> [1]
[6]      0.0    0.00    0.00     360         create_histogram <cycle 1> [6]
                                 64080           main <cycle 1> [1]
-----------------------------------------------
                                   180           main <cycle 1> [1]
[7]      0.0    0.00    0.00     180         find_closest <cycle 1> [7]
                                 32400           distance <cycle 1> [4]
-----------------------------------------------
```

Table 6

As can be seen from *Table 6* which depicts the call tree of the program, the function calls are totaled overall, and are not represented thread wise. Hence the values are similar to that of *Table 4.*

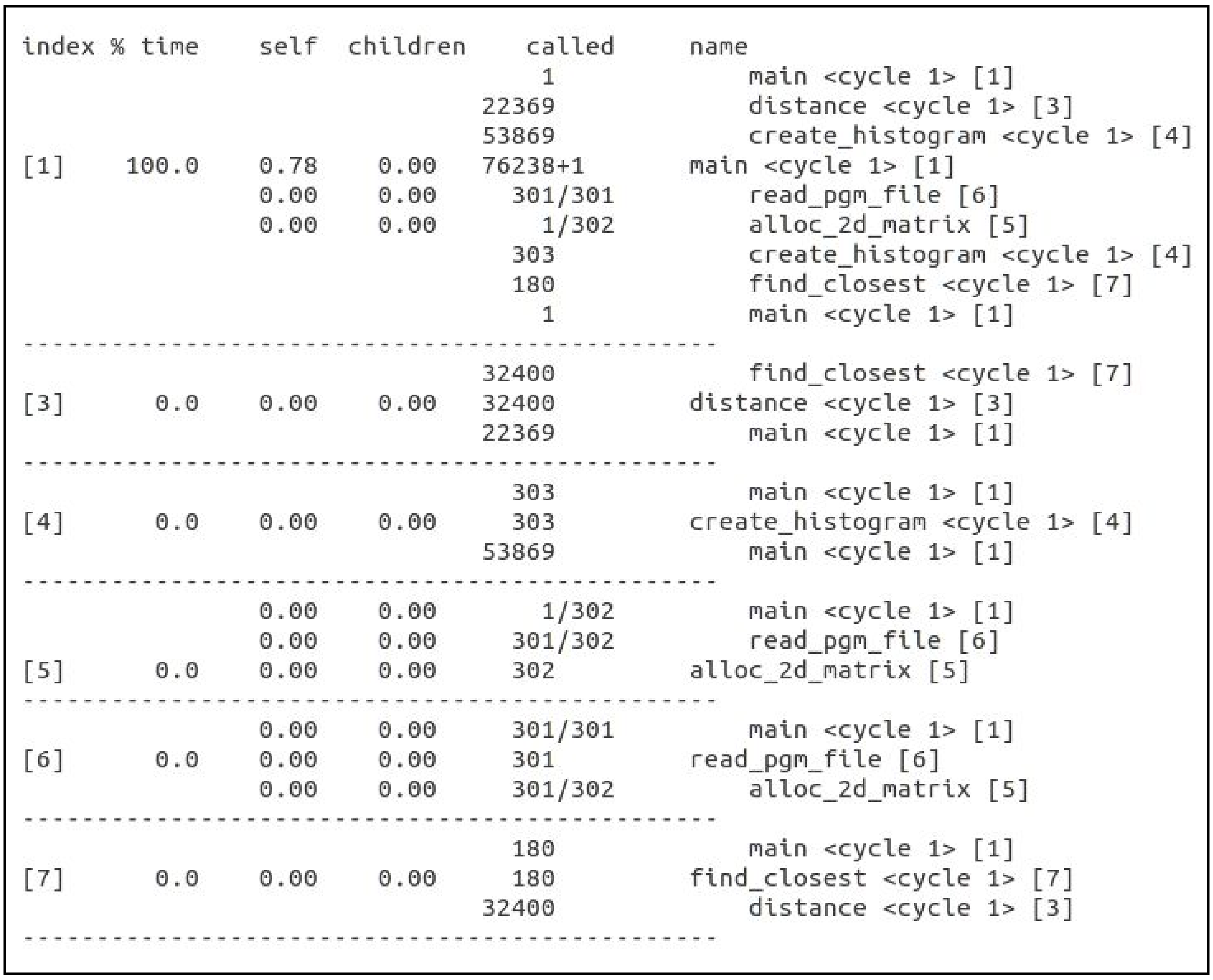
Execution with number of threads = 4

Similar results to the previous discussions.



```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
 98.86      0.78     0.78    76238    10.24    10.24  main
  0.00      0.78     0.00    32400     0.00     0.00  distance
  0.00      0.78     0.00      303     0.00     0.00  create_histogram
  0.00      0.78     0.00      302     0.00     0.00  alloc_2d_matrix
  0.00      0.78     0.00      301     0.00     0.00  read_pgm_file
__0.00      0.78     0.00      180     0.00     0.00  find_closest
```

Table 7

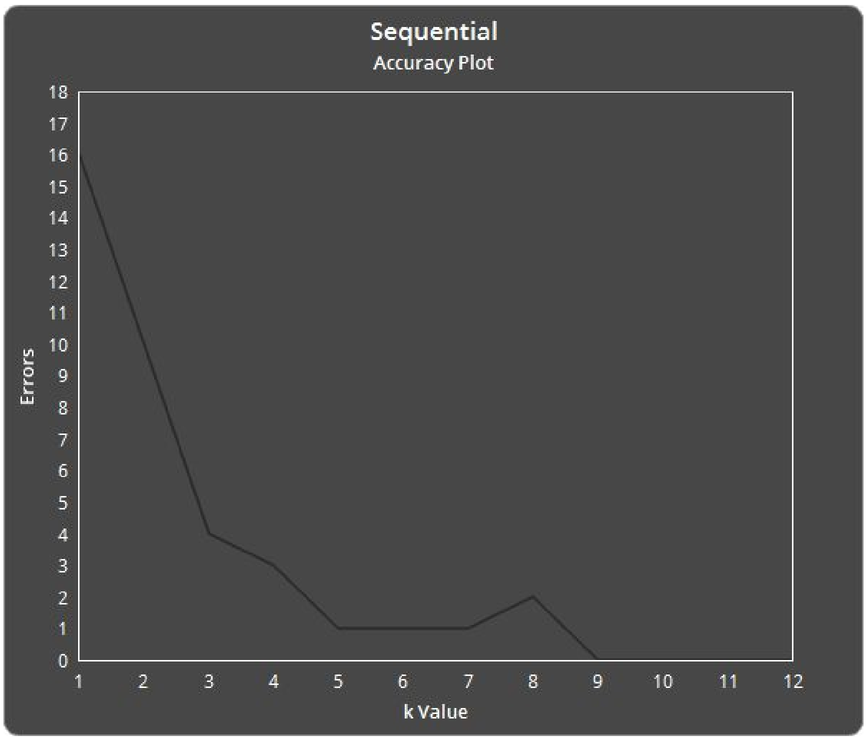


```
index % time    self  children    called     name
                                    1             main <cycle 1> [1]
                                22369             distance <cycle 1> [3]
                                53869             create_histogram <cycle 1> [4]
[1]    100.0    0.78    0.00    76238+1       main <cycle 1> [1]
                0.00    0.00     301/301          read_pgm_file [6]
                0.00    0.00       1/302          alloc_2d_matrix [5]
                                  303             create_histogram <cycle 1> [4]
                                  180             find_closest <cycle 1> [7]
                                    1             main <cycle 1> [1]
-----------------------------------------------
                                32400             find_closest <cycle 1> [7]
[3]      0.0    0.00    0.00    32400         distance <cycle 1> [3]
                                22369             main <cycle 1> [1]
-----------------------------------------------
                                  303             main <cycle 1> [1]
[4]      0.0    0.00    0.00      303         create_histogram <cycle 1> [4]
                                53869             main <cycle 1> [1]
-----------------------------------------------
                0.00    0.00       1/302          main <cycle 1> [1]
                0.00    0.00     301/302          read_pgm_file [6]
[5]      0.0    0.00    0.00     302          alloc_2d_matrix [5]
-----------------------------------------------
                0.00    0.00     301/301          main <cycle 1> [1]
[6]      0.0    0.00    0.00     301          read_pgm_file [6]
                0.00    0.00     301/302          alloc_2d_matrix [5]
-----------------------------------------------
                                  180             main <cycle 1> [1]
[7]      0.0    0.00    0.00      180         find_closest <cycle 1> [7]
                                32400             distance <cycle 1> [3]
-----------------------------------------------
```

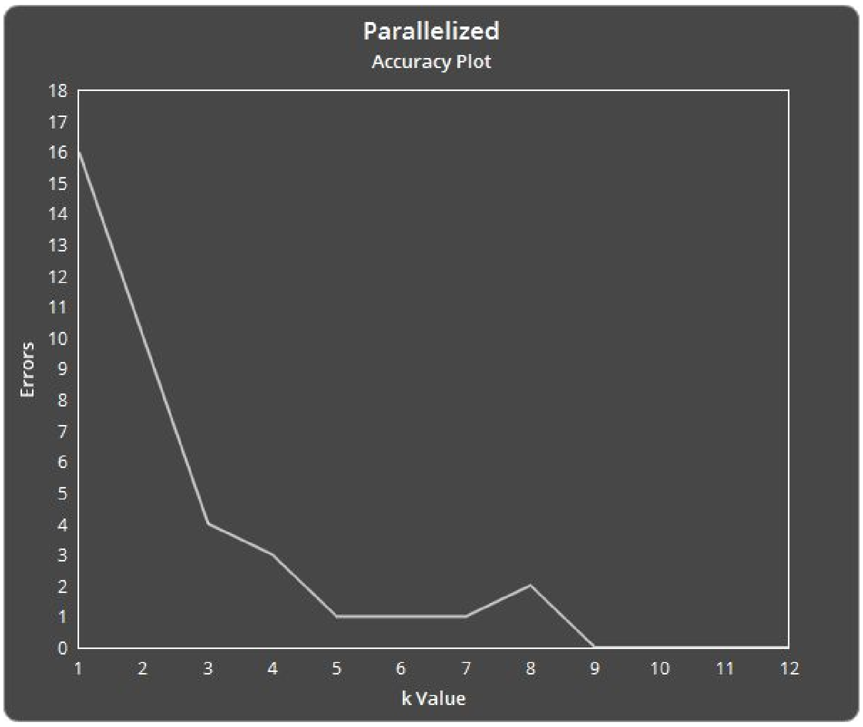
Table 8

Graph Plots

Accuracy Graphs



Plot 1

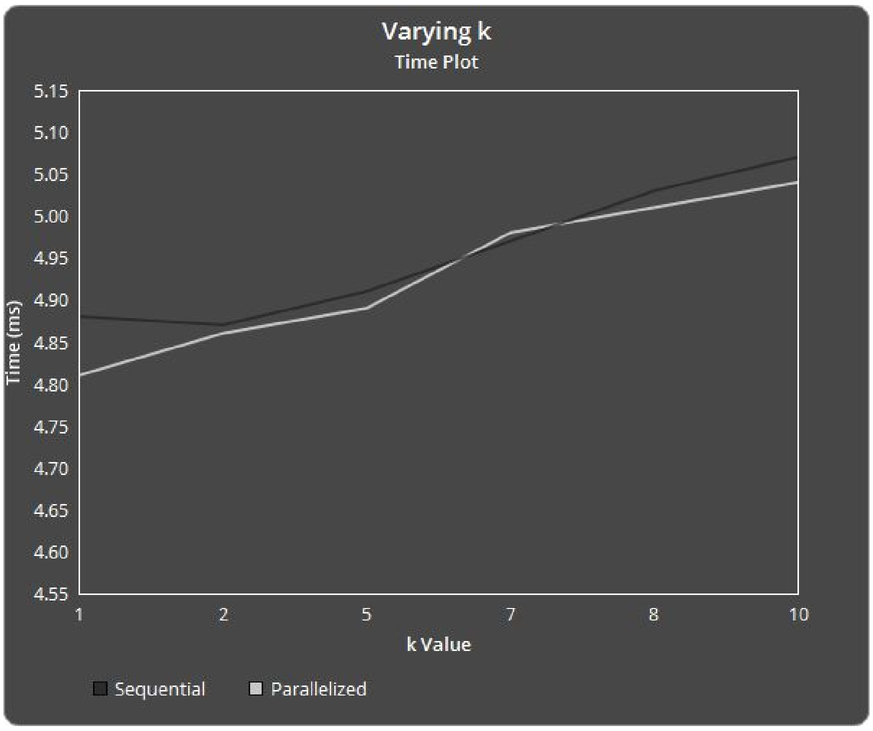


Plot 2

As can be seen from the two plots of accuracy, both the sequential and parallelized programs have the same accuracy. This is to be expected as the parallelization of the code is not meant to affect the accuracy of the lbp algorithm. By adding OMP pragmas, the functionality of the code is not to be altered, but instead its execution is to be optimized. Hence the same number of errors are witnessed in both the types of programs. It can be seen that by k = 9, the program achieves a 100% accuracy. Moreover, the trend of the errors is as expected, except for hen k = 8. This is because the more the training set, the higher is the expected accuracy. Thus it can be said that both the parallelized and sequential codes perform as expected.

Execution Times in Varying k

NB: Number of Threads set at 2 or Parallelization execution

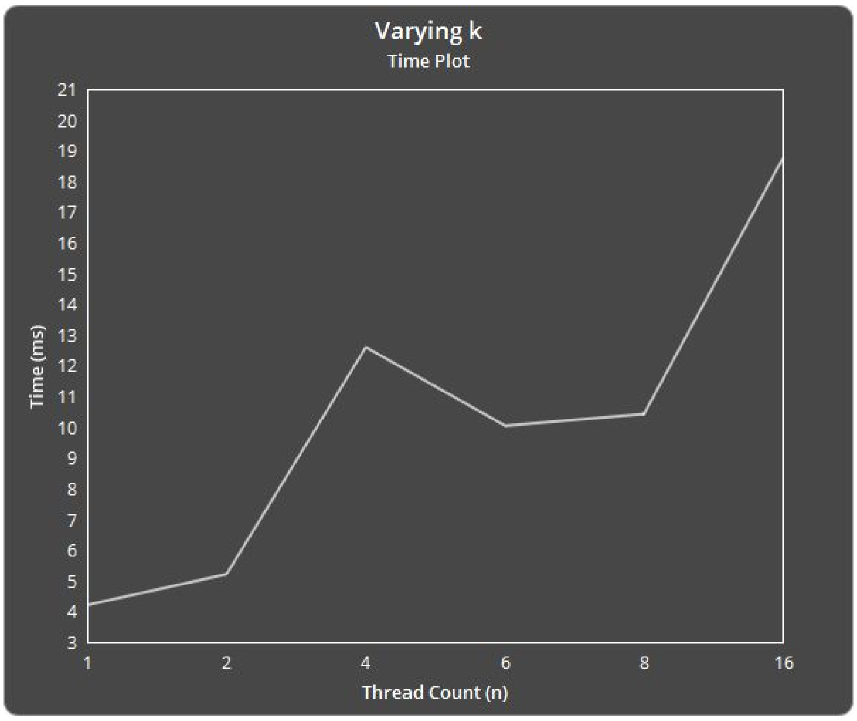


Plot 3

As can be seen the parallelized program does take a smaller time to execute, but the difference is in the 10th of milliseconds. Apart from an anomaly around k = 7, the parallel program does perform better than the sequential program. This can also be seen in the profiling of the programs as it is evident that the *create_histogram* function along with the distance function takes almost no time to execute. This has noteworthy effects on the performance of the program.

Execution Times in Varying Number of Threads

NB: k value set at 10



Plot 4

From *Plot 4* it can be seen that increasing the number of threads is having an overall adverse effect on the performance on the program. This is probably because of the overhead in caused in creating the threads in the first place. This needs to be noted as parallelized functions such as *create_histogram* and *distance* functions are called 180 and 32400 times at most *Table 6*. That means each time these functions are called threads are created. That leads to a large overhead. A solution to the issue could be to only parallelize the main function as seen in *Fig 3*. Otherwise the number of threads, being created leads to an overhead that outweighs the benefits of parallelization.