Bilkent University

Computer Engineering

CS 425 – Parallel Computing

**Project 3 Report**

**Due Date: 7 December 2017**

**Name: Gülsüm Güdükbay (21401148)**

**Description of Sequential Face Recognition Code**

1. Allocated pictures and “training\_sets” arrays. Pictures array is an array for every person’s every picture, which is 18 x 20. Every person’s every picture is 202 by 182, so pictures array is a 4D array. “training\_sets” array is a k\*18 x 202 x 182 array, which is a 3D array. For each person and for each picture of each person, “read\_pgm\_file” method of the “utils.h” file is called to populate the pictures array from the txt files. The “utils.c” file is modified so that zero padding is done while reading the file (by extending and sliding the indices by one in each dimension). The “training\_sets” array is also populated using the same method.

//2d arrays of pictures for each picture for each people

int \*\*\*\*pictures = (int\*\*\*\*)malloc(n \* sizeof(int\*\*\*));

int \*\*\*training\_sets = (int\*\*\*)malloc((k \* n) \* sizeof(int\*\*));

for(int i = 0; i < n; i++)

pictures[i] = (int\*\*\*)malloc(p \* sizeof(int\*\*));

1. “histograms” array is a 2D array which contains the histogram of each training set. For each training set, “create\_histograms” method is called. This method creates the histogram by traversing each pixel and for each of them, compares the pixel with its 8 neighbors clockwise. Then, where each center pixel is smaller than its neighbor, that pixel is set to 1, otherwise 0. The 8 pixels are converted to a decimal value from binary in a cyclic clockwise fashion. Then, the histogram is generated by incrementing the decimal value-th index of the current histogram by one for each pixel.

//create histograms of training sets

**int**\*\* histograms = (**int**\*\*)**malloc**(k \* n \* **sizeof**(**int**\*));

**for**(**int** i = 0; i < k\*n; i++)

{

histograms[i] = (**int**\*)**malloc**(hist\_size \* **sizeof**(**int**));

**create\_histogram**(histograms[i], training\_sets[i], 202, 182);

}

**void** **create\_histogram**(**int**\* hist, **int**\*\* img, **int** num\_rows, **int** num\_cols)

{

**int** cur\_sum = 0;

**for**(**int** i = 0; i < 256; i++)

hist[i] = 0;

**for**(**int** i = 1; i < num\_rows-1; i++)

{

cur\_sum = 0;

**for**(**int** j = 1; j < num\_cols-1; j++)

{

cur\_sum = 0;

**if**(img[i][j] < img[i-1][j-1])

cur\_sum += 128;

**if**(img[i][j] < img[i-1][j])

cur\_sum += 64;

**if**(img[i][j] < img[i-1][j+1])

cur\_sum += 32;

**if**(img[i][j] < img[i][j+1])

cur\_sum += 16;

**if**(img[i][j] < img[i+1][j+1])

cur\_sum += 8;

**if**(img[i][j] < img[i+1][j])

cur\_sum += 4;

**if**(img[i][j] < img[i+1][j-1])

cur\_sum += 2;

**if**(img[i][j] < img[i][j-1])

cur\_sum += 1;

hist[cur\_sum]++;

}

}

}

1. For each person’s each test image, the distance between the test image’s histogram and each training set’s histogram is calculated. For each person’s each test image, the minimum distance is found and the index of it is returned as the closest person id for that test image. This is the “find\_closest” method.

int find\_closest(int \*\*\*histograms, int num\_persons, int num\_training, int size, int \*

test\_image)

{

double\* distances = (double\*)malloc(num\_training \* num\_persons\* sizeof(double));

double min\_number = DBL\_MAX;

int min\_index = 0;

for(int i = 0; i < num\_training\*num\_persons; i++)

{

distances[i] = distance( (\*histograms)[i], test\_image, size);

if(distances[i] < min\_number)

{

min\_number = distances[i];

min\_index = i;

}

}

free(distances);

return min\_index / num\_training;

}

1. Then, the true person ids and closest person ids for each person’s each test image is compared, and the number of inequalities are counted to find the error numbers.

**for**(**int** i = 0; i < n; i++) //traverse each person

{

**for**(**int** j = k; j < p; j++) //traverse each test image for each person

{

**int**\* test\_img\_hist = (**int**\*)**malloc**(hist\_size \* **sizeof**(**int**));

**create\_histogram**(test\_img\_hist, pictures[i][j], rows, cols);

//find closest person ids for each person, for each test image

closest\_indices[i][j-k] = **find\_closest**(&histograms, n, k, hist\_size, test\_img\_hist);

**free**(test\_img\_hist);

}

}

**int** errors = 0;

**for**(**int** i = 0; i < n; i++) //traverse each person

{

**for**(**int** j = 0; j < p-k; j++) //traverse each test image for each person

{

**printf**("%d.%d.txt\t%d %d\n", i+1, j+k+1, closest\_indices[i][j]+1, i+1);

errors += (closest\_indices[i][j] != i);

}

}

**printf**("Accuracy: %d errors out of %d test images.\n", errors, n \* (p-k));

**Profiling Results Using gprof**

Sequential

***k = 10***

Flat profile:

Each sample counts as 0.01 seconds.

**% cumulative self self total**

**time seconds seconds calls ms/call ms/call name**

79.49 0.60 0.60 360 1.66 1.66 create\_histogram

10.69 0.68 0.08 180 0.45 0.45 find\_closest

7.35 0.73 0.06 32400 0.00 0.00 distance

2.67 0.75 0.02 1082 0.02 0.02 read\_pgm\_file

0.00 0.75 0.00 541 0.00 0.00 dealloc\_2d\_matrix

Call graph

granularity: each sample hit covers 2 byte(s) for 1.33% of 0.75 seconds

**index % time self children called name**

<spontaneous>

[1] 100.0 0.00 0.75 main [1]

0.60 0.00 360/360 create\_histogram [2]

0.08 0.00 180/180 find\_closest [3]

0.06 0.00 32400/32400 distance [4]

0.02 0.00 1082/1082 read\_pgm\_file [5]

0.00 0.00 1/541 dealloc\_2d\_matrix [6]

------------------------------------------------------------------------------

0.60 0.00 360/360 main [1]

[2] 79.3 0.60 0.00 360 create\_histogram [2]

------------------------------------------------------------------------------

0.08 0.00 180/180 main [1]

[3] 10.7 0.08 0.00 180 find\_closest [3]

------------------------------------------------------------------------------

0.06 0.00 32400/32400 main [1]

[4] 7.3 0.06 0.00 32400 distance [4]

------------------------------------------------------------------------------

0.02 0.00 1082/1082 main [1]

[5] 2.7 0.02 0.00 1082 read\_pgm\_file [5]

0.00 0.00 540/541 dealloc\_2d\_matrix [6]

------------------------------------------------------------------------------

0.00 0.00 1/541 main [1]

0.00 0.00 540/541 read\_pgm\_file [5]

[6] 0.0 0.00 0.00 541 dealloc\_2d\_matrix [6]

Index by function name

[6] alloc\_2d\_matrix [5] dealloc\_2d\_matrix [7] find\_closest

[2] create\_histogram [4] distance [3] read\_pgm\_file

The above tables show the percentage of total running time of the program used by each function (main ([1]), create\_histogram ([2]), read\_pgm\_file ([3]), distance ([4]), dealloc\_2d\_matrix ([5]), alloc\_2d\_matrix ([6]), find\_closest ([7])). Also, the tables show the cumulative time taken as each function is being executed, the duration of each function separately, the number of times each function is called, average number of milliseconds each function spends per call, average number of milliseconds each function and its descendants spend per call, name of each function, the indices of each function, the percentage of the total time spent in each function and its children, total amount of time propagated into each function by its children, number of times each function is called and its children were called.

As seen in the tables, most of the time taken was by the create\_histogram function with 79.49%, which was called 360 times. The duration of this function is 1.45 ms in average and it is more than other function’s durations, because it processes each pixel in O(n2) and 200x180x360 elements are processed. Then, the file reading function read\_pgm\_file was the second with 10.69%, which was called 378 times. The most called function was the distance function with 32400 calls, however its duration is too less to be considered in terms of percentage.

Sample Output of Sequential Code for k=10

1.11.txt 1 1

1.12.txt 1 1

1.13.txt 1 1

1.14.txt 1 1

1.15.txt 1 1

Accuracy: 0 errors out of 180 test images.

Sequential time: 2.661891

**Description of Parallel Face Recognition Code**

1. The algorithm in the sequential face recognition is kept the same. Pragmas were added in order to parallelize the face recognition code. Following steps show where and why the pragmas were inserted to determined places.
2. At first, #pragma omp parallel for statements were inserted into each function’s loops (create\_histogram, distance and find\_closest). However, when the program was compiled and ran with multiple threads, the execution time was very high due to the overhead of parallelizing. Looking at the profiling results with ompP (OpenMP Profiler) (gprof only profiles the main method of the OpenMP applications since it does not comprehend the paralellizations, that is why ompP is used for profiling), the overhead was significantly high, which increased the execution times to more than 6 seconds. Not only overhead, but also the critical sections such as the sum part in the distance function increased the execution time due to the threads waiting each other. If critical sections were not indicated, race conditions happened.
3. Since creating the histograms take a lot of time due to the number of pixels processed (looking at the serial gprof results), parallelizing any stuff related to the create\_histogram would be beneficial. Since I did not use any nested parallelism, I had to make a decision between parallelizing the loop inside the create\_histogram method or parallelizing the loop that I call that method in the main method. After I tried both of them, parallelizing the loop in the main method where the create\_histogram method is called decreased the execution time, because of the overhead associated with the parallelized for loop of the row pixels in the create\_histogram method. The overhead is due to the create\_histogram method being called 18(k-20) times in a nested for loop, not the loop in which the training set histograms are created (most significantly)

This pragma overall parallelizes the creation of histograms among threads for each training set. So if the number of threads is 4, each thread calls the create\_histogram method k\*18/4 times. Here, all variables were shared.

**#pragma omp parallel for shared(histograms, training\_sets, rows, cols)**

**for(int i = 0; i < k\*n; i++)**

**{**

**create\_histogram(histograms[i], training\_sets[i], rows, cols);**

**}**

1. The most important decision was made when parallelizing the nested loop in the main method, which traverses each person and each test image of each person to create the histograms and to find the closest person id for each test image. At first, the outer loop was parallelized, however this did not ensure an optimal granularity, since it was a coarse grain due to the inner loops and the calls to create\_histograms method p-k times in each inner loop. That is why, the inner loops are parallelized for each person, which also decreased the execution time significantly. Here, pictures, histogram, closest\_indices, rows and cols variables were shared.

**for(int i = 0; i < n; i++) //traverse each person**

**{**

**#pragma omp parallel for shared(pictures, histograms, closest\_indices, rows, cols)**

**for(int j = k; j < p; j++) //traverse each test image for each person**

**{**

**int\* test\_img\_hist = (int\*)malloc(hist\_size \* sizeof(int));**

**create\_histogram(test\_img\_hist, pictures[i][j], rows, cols);**

**//find closest person ids for each person, for each test image**

**closest\_indices[i][j-k] = find\_closest(&histograms, n, k, hist\_size, test\_img\_hist);**

**free(test\_img\_hist);**

**}**

**}**

1. Lastly, counting the errors was parallelized using #pragma omp parallel for reduction. This was due to the fact that the summation was performed 18\*(20-k) times and in terms of duration, this benefited the execution time by decreasing it. Every variable was shared here, and the reduction was made on the errors variable, in which summation was performed on.

**for(int i = 0; i < n; i++) //traverse each person**

**{**

**#pragma omp parallel for reduction(+:errors)**

**for(int j = 0; j < p-k; j++) //traverse each test image for each person**

**{**

**errors += (closest\_indices[i][j] != i);**

**}**

**}**

**printf("Accuracy: %d errors out of %d test images.\n", errors, n \* (p-k));**

**Profiling Results Using gprof**

Flat profile:

Each sample counts as 0.01 seconds.

% cumulative self self total

time seconds seconds calls ms/call ms/call name

76.22 0.51 0.51 358 1.43 1.43 create\_histogram

13.45 0.60 0.09 540 0.17 0.17 read\_pgm\_file

10.46 0.67 0.07 32068 0.00 0.00 distance

0.00 0.67 0.00 542 0.00 0.00 dealloc\_2d\_matrix

0.00 0.67 0.00 541 0.00 0.00 alloc\_2d\_matrix

0.00 0.67 0.00 180 0.00 0.39 find\_closest

Call graph (explanation follows)

granularity: each sample hit covers 2 byte(s) for 1.49% of 0.67 seconds

**index % time self children called name**

[1] 100.0 0.00 0.67 main [1]

0.51 0.00 358/358 create\_histogram [2]

0.09 0.00 540/540 read\_pgm\_file [3]

0.00 0.07 180/180 find\_closest [5]

0.00 0.00 542/542 dealloc\_2d\_matrix [6]

0.00 0.00 1/541 alloc\_2d\_matrix [7]

--------------------------------------------------------------------------

0.51 0.00 358/358 main [1]

[2] 76.1 0.51 0.00 358 create\_histogram [2]

--------------------------------------------------------------------------

0.09 0.00 540/540 main [1]

[3] 13.4 0.09 0.00 540 read\_pgm\_file [3]

0.00 0.00 540/541 alloc\_2d\_matrix [7]

--------------------------------------------------------------------------

0.07 0.00 32068/32068 find\_closest [5]

[4] 10.4 0.07 0.00 32068 distance [4]

--------------------------------------------------------------------------

0.00 0.07 180/180 main [1]

[5] 10.4 0.00 0.07 180 find\_closest [5]

0.07 0.00 32068/32068 distance [4]

--------------------------------------------------------------------------

0.00 0.00 542/542 main [1]

[6] 0.0 0.00 0.00 542 dealloc\_2d\_matrix [6]

--------------------------------------------------------------------------

0.00 0.00 1/541 main [1]

0.00 0.00 540/541 read\_pgm\_file [3]

[7] 0.0 0.00 0.00 541 alloc\_2d\_matrix [7]

--------------------------------------------------------------------------

Index by function name

[7] alloc\_2d\_matrix [6] dealloc\_2d\_matrix [5] find\_closest

[2] create\_histogram [4] distance [3] read\_pgm\_file

The above tables show the percentage of total running time of the program used by each function (main ([1]), create\_histogram ([2]), read\_pgm\_file ([3]), distance ([4]), find\_closest ([5]), dealloc\_2d\_matrix ([6]), alloc\_2d\_matrix ([7])).

As seen in the tables, most of the time taken was again by the create\_histogram function with 76.22% The duration of this function is 1.43 ms in average and it is more than other function’s durations, because it processes each pixel in O(n2) and 200x180x360 elements are processed. Then, the file reading function read\_pgm\_file was the second with 13.45%, which was called 540 times. The most called function was the distance function with 32068 calls, however its duration is too less to be considered in terms of percentage. These percentages and durations are less than sequential, but not too much because of the overheads of creating and destroying threads. Because of the fact that gprof does not comprehend the parallelized regions of a program, ompP was utilized to further profile the program and look at the overheads associated with parallelized regions.

For thread count 4 and k=10, the parallel profiling results were as follows:

----------------------------------------------------------------------

---- ompP General Information --------------------------------

----------------------------------------------------------------------

Start Date : Thu Dec 07 19:31:24 2017

End Date : Thu Dec 07 19:31:24 2017

Duration : 0.28 sec

Application Name: unknown

Type of Report : final

User Time : 0.52 sec

System Time : 0.00 sec

Max Threads : 4

ompP Version : 0.8.99

ompP Build Date : Dec 7 2017 12:54:01

PAPI Support : not available

----------------------------------------------------------------------

---- ompP Region Overview ------------------------------------

----------------------------------------------------------------------

PARALLEL LOOP: 3 regions:

\* R00001 lbp\_omp.c (148-152)

\* R00002 lbp\_omp.c (156-165)

\* R00003 lbp\_omp.c (179-183)

----------------------------------------------------------------------

---- ompP Callgraph ------------------------------------------

----------------------------------------------------------------------

Inclusive (%) Exclusive (%)

0.28 (100.0%) 0.01 ( 2.25%) [unknown: 4 threads]

0.01 ( 4.70%) 0.01 ( 4.70%) PARLOOP |-R00001 lbp\_omp.c (148-152)

0.26 (92.92%) 0.26 (92.92%) PARLOOP |-R00002 lbp\_omp.c (156-165)

0.00 ( 0.13%) 0.00 ( 0.13%) PARLOOP +-R00003 lbp\_omp.c (179-183)

----------------------------------------------------------------------

---- ompP Overhead Analysis Report ---------------------------

----------------------------------------------------------------------

Total runtime (wallclock) : 0.28 sec [4 threads]

Number of parallel regions : 3

Parallel coverage : 0.27 sec (97.75%)

Parallel regions sorted by wallclock time:

Type Location Wallclock (%)

R00002 PARLOOP lbp\_omp.c (156-165) 0.26 (92.92)

R00001 PARLOOP lbp\_omp.c (148-152) 0.01 ( 4.70)

R00003 PARLOOP lbp\_omp.c (179-183) 0.00 ( 0.13)

SUM 0.27 (97.75)

Overheads wrt. each individual parallel region:

Total Ovhds (%) = Synch (%) + Imbal (%) + Limpar (%) + Mgmt (%)

R00002 1.04 0.47 (45.11) 0.00 ( 0.00) 0.24 (22.88) 0.00 ( 0.00) 0.23 (22.23)

R00001 0.05 0.02 (37.68) 0.00 ( 0.00) 0.01 (20.27) 0.00 ( 0.00) 0.01 (17.42)

R00003 0.00 0.00 (97.51) 0.00 ( 0.00) 0.00 (33.29) 0.00 ( 0.00) 0.00 (64.21)

Overheads wrt. whole program:

Total Ovhds (%) = Synch (%) + Imbal (%) + Limpar (%) + Mgmt (%)

R00002 1.04 0.47 (41.92) 0.00 ( 0.00) 0.24 (21.26) 0.00 ( 0.00) 0.23 (20.66)

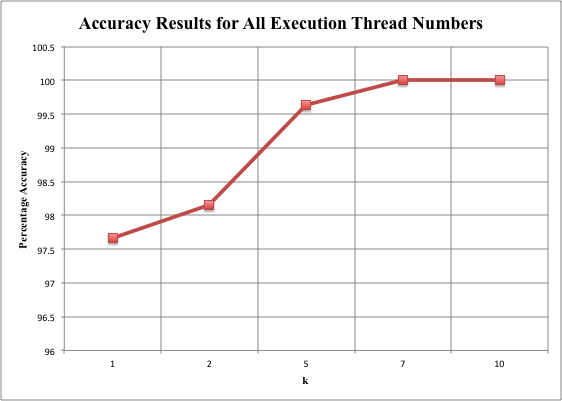
R00001 0.05 0.02 ( 1.77) 0.00 ( 0.00) 0.01 ( 0.95) 0.00 ( 0.00) 0.01 ( 0.82)

R00003 0.00 0.00 ( 0.12) 0.00 ( 0.00) 0.00 ( 0.04) 0.00 ( 0.00) 0.00 ( 0.08)

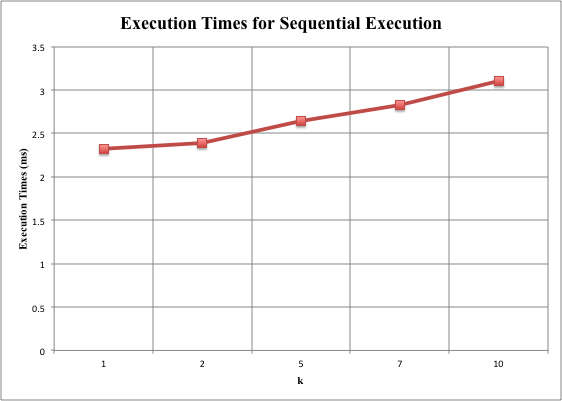
SUM 1.09 0.49 (43.81) 0.00 ( 0.00) 0.25 (22.26) 0.00 ( 0.00) 0.24 (21.56)

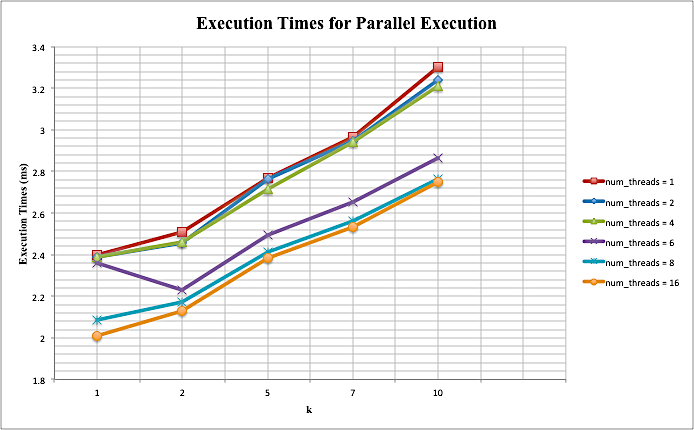
If we look at the other profiling results for different number of threads, as the number of threads go larger than 4 or smaller than 4, the execution time increases and the total overhead increases significantly as it is set larger. The highest execution time is taken by the parallel region of the nested loop in the main method to find the closest indices. This is due to the amount of processing. The not-so-fine granularity helps decrease the overhead of creating many threads and load balancing. The total parallel coverage is 97.75%.

**Results**



The above graph shows the accuracy results for each execution type and k values. As the number of threads change, the accuracy results do not change. That is why, only one mutual graph is provided for the accuracy results. As seen in the graph below, the execution times for sequential execution increase linearly as the k values increase. This is due to the increase in the number of training sets and the histogram comparison number increasing linearly too.





The graph above shows the execution times for each number of threads and for each k values for each number of threads. As seen above, the execution time in executing the program with 16 threads is the best one, since the program benefits from parallelizing and optimal granularity. The execution times increase linearly as k increases and the worst execution times are when there is no parallelization (num\_threads = 1). The graph shows that parallelizing really decreases the execution time, which proves the hypothesis that as there is optimal parallelization, the execution time decreases.