Bilkent University

Department of Computer Engineer

CS426 Parallel Computing

Project 3 Report

Serhat Aras

21401636

1 Implementation and design choices

1.1 Problem Definition

This project aims to implement a Face Recognition Algorithm using Local Binary Patterns, this simple procedure is used for variety of purpose in texture analysis, namely gesture and face recognition applications utilize this algorithm. Data used in the project are a derivation of "Collection of Facial Images: Grimace"[1] Dataset. These pre-processed 180x200 pixel sized images are used both in the train and the test cycles in the LBP Face Recognition implementation.

In one image, the LBP Face Recognition algorithm is calculating local features for each pixel using its 8-Neighborhood Pixel values and its own pixel value. In this comparison, if a neighbor's value is greater than the value of the central Pixel of the kernel, value of the index that corresponds to the neighbor's index is updated as 1 in the 8-digit binary representation of the kernel. Otherwise its value is corresponds to 0. This 8-digit binary representation later converted to a decimal representation of kernel's LBP value in which it's going to determine the index of this value in the frequency histogram of the Image.

1.2 Implementation

Implementing Local Binary Patterns divides this project into 2 major parts, sequential and parallel implementation of the same problem. Each problem is literally executing the same algorithm to some extend except the parallel version includes OpenMP Library which enables LBP Face Recognition system to run on parallel using pragmas' of the OpenMP. In this section, this report states the common factors on each implementation for simplicity.

In the first phase of the LBP Face Recognition System of the 2 implementation, the dataset must be readed from the "./images" folder at the root directory of the project. In order to read, the util.c used which is provided by the TA. 2 function is handles dataset reading. These are; *alloc_2d_matrix(int r, int c)* and *read_pgm_file(char * file_name, int h, int w)* function. Note that not all data is readed at the beginning, the algorithm reads an image when it needs to.

The system requires Image data to train itself and later, to test its correctness, therefore “k value” which corresponds to how much images are going to be used to train for one person in LBP Face Recognition System is used and this value is provided by user as console argument.

To train the LBP Face Recognition System, for each person in the dataset, k number of their images are used to calculate their frequency histogram. This processed discussed earlier in the this section. To create histograms, the system calls create_histogram(int * hist, int ** img, int num_rows, int num_cols) function which calculates LBP Frequency Histogram for an image and updates an integer array named hist in the function parameters. This histogram values are are stored for each image and used in later to predict the image owner.

After each of the (18*k) images’ histograms are calculated, the system tries to predict [18*(20-k)] remaining images. Prediction includes 2 functions, *distance(int *a, int *b, int size)* which finds finds the distance between two vectors and *find_closest(int ***training_set, int num_persons, int num_training, int size, int *test_image)* which finds the closest histogram for test image's histogram from training set Returns person id of the closest histogram. To predict the person, the LBP Face Recognition System executes the following algorithm.

[1] For each test image:
- **[1.1] Calculate the histogram value,**
- **[1.2] Find a histogram with minimum distance from the training set.**
  - **[1.2.1] For each histogram in the training set**
  - **[1.2.2] Calculate distance between test image’s histogram and training image**
- **[1.3] Return the ID of the person that has the closest distance.**

For paralleling the system, the openMP pragmas are used in strategic location in the code. This locations are determined based on the characteristics of the section. For example, the bottleneck in the algorithm is caused by the function from the *create_histogram()* function.The pragmas used in this methods are used to parallelize the nested for loops. Pragmas that featured in the implementation are:

In the *create_histogram()* method, The ***#pragma omp parallel for collapse(2)*** is used for paralleling the 2 nested for loop and speed up the O(n^2) algorithm just as in Matrix Matrix Multiplication. Also ***#pragma omp atomic*** is used to let only the master thread to update the

histogram counts since multiple threads has possibility to increment the value of the same array element which results in a race condition, this incrementation must be handled one thread at a time in order to prevent it and this OpenMP pragma prevents this.

While bearing the same approach, *find_closest()* function features 2 pragmas also. These are ***#pragma omp parallel for*** used as in the find_closest() method and ***#pragma omp atomic*** while incrementing the distance value. Since multiple threads has possibility to increment the value of the distance variable which results in a race condition, this incrementation must be handled one thread at a time in order to prevent it and this OpenMP pragma prevents this.

In the same manner, *find_closest()* function uses the same approach with the *create_histogram()*. The ***#pragma omp parallel for collapse(2)*** used in the paralleling 2 nested for loop and inside these loops, another OpenMP statement, ***#pragma omp critical*** pragma is used while updating the value of the distance variable which used in the comparison of each person pairs histograms,. This multiple access and updates may result in a race condition if it's attempted to be updated by multiple threads. This pragma is used to prevent this.

The analysis of the regions to determine the where to insert these pragmas are derived from the analysis of the GPROF output. The main motive is to aid the heavily bottle necked areas by dividing the work between threads. This analysis is discussed in a section later in the report.

1.2.1 Sequential Version Sample Outputs

Some of the selected outputs are displayed below for both parallel and sequential part. Note that this is not the full output of the system.

1.2.1 Sequential Version Sample Outputs

- ***For k=1***

1.2.txt 1 1
1.3.txt 1 1
1.4.txt 1 1
…..
18.17.txt 13 18
18.18.txt 18 18
18.19.txt 18 18
18.20.txt 18 18
Accuracy: 329 correct answers for 342 tests
Sequential time: 1491.61 ms

- ***For k=5***

1.6.txt 1 1
1.7.txt 1 1
1.8.txt 1 1
…..
18.17.txt 18 18
18.18.txt 18 18
18.19.txt 18 18
18.20.txt 18 18
Accuracy: 267 correct answers for 270 tests
Sequential time: 1585.46 ms

- ***For k=10***

1.11.txt 1 1
1.12.txt 1 1
1.13.txt 1 1
…..
18.17.txt 18 18
18.18.txt 18 18
18.19.txt 18 18
18.20.txt 18 18
Accuracy: 180 correct answers for 180 tests
Sequential time: 1648.63 ms

1.2.2 Parallel Version Sample Outputs

- ***For k=1, # of Cores 1***

…..
18.17.txt 13 18
18.18.txt 18 18
18.19.txt 18 18
18.20.txt 18 18
Accuracy: 329 correct answers for 342 tests
Parallel time: 1564.12 ms
Sequential time: 0.77 ms

- ***For k=5, # of cores 8***

….
18.17.txt 18 18
18.18.txt 18 18
18.19.txt 18 18
18.20.txt 18 18
Accuracy: 267 correct answers for 270 tests
Parallel time: 1522.45 ms
Sequential time: 0.74 ms

- ***For k=10, # of cores 16***

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
1.16.txt 1 1
1.17.txt 1 1
…..

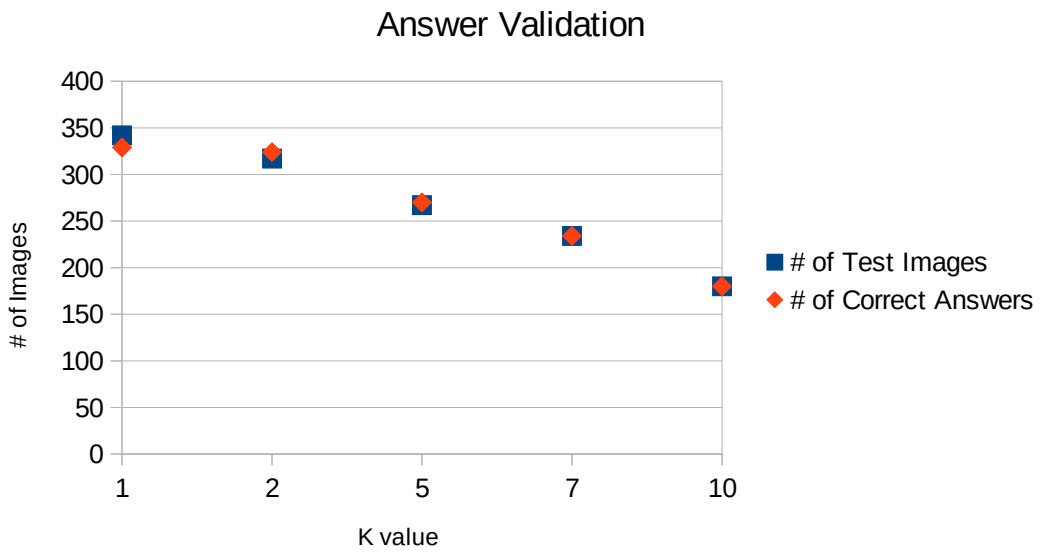
18.15.txt 18 18
18.16.txt 18 18
18.17.txt 18 18
18.18.txt 18 18
18.19.txt 18 18
18.20.txt 18 18
Accuracy: 180 correct answers for 180 tests
Parallel time: 1628.72 ms
Sequential time: 0.92 ms

More examples are provided with the submitted data. Please see the serhat_aras.output for full combination of the outputs.

2 Performance Comparison & Observations

For the performance overview, the following graphs are prepared and discussed in the following subsections. .

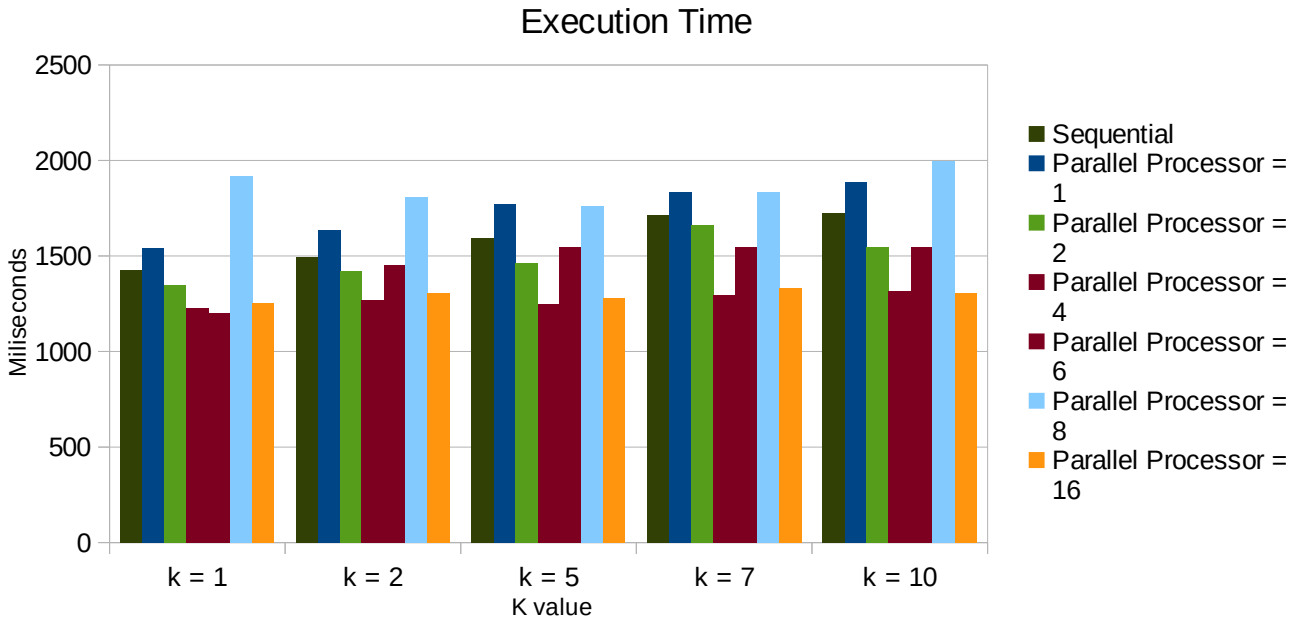
2.1 Correctness of the LBP Face Recognition Algorithm



# Of Images	342	324	270	234	180
# of Correct Predictions	329	317	267	234	180

The graph above shows the relative values of the number of test images and the number of correct assumptions resulting at usage of the algorithm. As stated in the graphs, number of correct predictions are converging to the real value at the k greater or equal to 6, the graph contains k = 7 and 10 to demonstrate this. The most wrong guesses happens at k=1 for both of the parallel and sequential cases. The number of wrong predicted image is 13. The algorithm tends to decrease the number of wrong guesses if the train data increased. The number of guesses appeared to be not related to the number of cores in the parallel execution.

2.2 Execution time of the LBP Face Recognition Algorithm



The above graph show the execution time of all possible scenarios requested in the project manual. As observed from the graph, the value of the K variable is highly affects the running time of the algorithm, however, effects of the number of cores for parallel execution also another factor for execution time when considering the parallel algorithms analysis. The most demanding part of the algorithm is createHistogram() function. After paralleling the create_histogram(), distance() and find_closest() functions, the profiler shows that the main part is now holds the most of the execution time and the old bottlenecks are eliminated from the application.

2.3 GPROF Profiler Analysis

GPROF is a tool to analyze the execution of the applications. Using this GCC-Profiler, the developer can examine the number of function calls from each subroutine within the program with the load and execution time data. In this project, we introduced the GPROF since the Implementation of the parallel section aims to ease the load of the execution by paralleling the heavy tasks to multiple threads and cores and execute the program simultaneously. To achieve this, after we generate and run a executable of the LBP Face Recognition system, we call the *prof -b lbp_omp gmon.out* terminal command. This generates a gmon.out file which contains the profile information, afterwards, the profile data is used to generate a readable text file. An example to the gprof output is provided below.

******If the '-b' option is given, gprof doesn't print the verbose blurbs that try to explain the meaning of all of the fields in the tables. This is useful if you intend to print out the output, or are tired of seeing the blurbs.***

********---> Paralel Execution for

********------> K = 1

********------> OMP_NUM_THREADS=6

Flat profile:

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self us/call	total us/call	name
92.66	0.50	0.50				main
5.56	0.53	0.03	360	83.39	83.39	create_histogram
1.85	0.54	0.01	360	27.80	27.80	read_pgm_file
0.00	0.54	0.00	4005	0.00	0.00	distance
0.00	0.54	0.00	720	0.00	0.00	alloc_2d_matrix
0.00	0.54	0.00	720	0.00	0.00	dealloc_2d_matrix
0.00	0.54	0.00	342	0.00	0.00	find_closest

Call graph

granularity: each sample hit covers 2 byte(s) for 1.85% of 0.54 seconds

index	% time	self	children	called	name
					<spontaneous>
[1]	100.0	0.50	0.04		main [1]
		0.03	0.00	360/360	create_histogram [2]
		0.01	0.00	360/360	read_pgm_file [3]
		0.00	0.00	4005/4005	distance [4]
		0.00	0.00	360/720	dealloc_2d_matrix [6]
		0.00	0.00	342/342	find_closest [7]

		0.03	0.00	360/360	main [1]
[2]	5.6	0.03	0.00	360	create_histogram [2]
		0.00	0.00	360/720	alloc_2d_matrix [5]
		0.00	0.00	360/720	dealloc_2d_matrix [6]

		0.01	0.00	360/360	main [1]
[3]	1.9	0.01	0.00	360	read_pgm_file [3]
		0.00	0.00	360/720	alloc_2d_matrix [5]

		0.00	0.00	4005/4005	main [1]
[4]	0.0	0.00	0.00	4005	distance [4]

		0.00	0.00	360/720	create_histogram [2]
		0.00	0.00	360/720	read_pgm_file [3]
[5]	0.0	0.00	0.00	720	alloc_2d_matrix [5]

		0.00	0.00	360/720	create_histogram [2]
		0.00	0.00	360/720	main [1]
[6]	0.0	0.00	0.00	720	dealloc_2d_matrix [6]

		0.00	0.00	342/342	main [1]
[7]	0.0	0.00	0.00	342	find_closest [7]

The function selected to be paralled is determined based on the execution time compared to overall execution. As discussed at the execution time analysis, create_Histogram() callls are basically took the longest portion of the execution time. Also, the number of K has a direct correlation with the number of Distance function calls. Since each created threads are going over the same training set, if we increase the number of elements in this set, we directly increase the number of elements to compare. As a result of this algorithm, the bottleneck happens for each create_Histogram() function calls which runs on O(n^2). After we paralleled this function, the cumulative execution time of the create_Histogram() function decreases significantly compared to the sequential version. Most of the heavy work can be observed on main function. This call hierarchy can be examined in the profiler since the number of calls to create_Histogram() hardly changes but due to the changing execution time of the cumulative time, bottleneck for the application can be observed easily.

3. References

[1] https://cswww.essex.ac.uk/mv/allfaces/grimace.html

[2] http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.456.1094&rep=rep1&type=pdf

Hardware Info of the Computer:

Processor	Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz
Memory	16381MB
Graphic Card	NVIDIA Gtx1070 (8 GB)
Operating System	Ubuntu 19.04
Kernel Version	5.0.0-13-generic
GCC Comp.	Using built-in specs.