Parallel Face Recognition using LBPH

Shubham Gupta, Anish Satpati, Tamoghno Kandar and Tushar Dhingra

***Abstract*—A parallel implementation of the LBPH algorithm for face recognition is developed and studied. parallel implementation has been developed using the OpenMP API. Studies related to the time of execution of the parallel program while varying the active parallel processes is done. Further the MPI library is incorporated into the study and a hybrid program using both shared and distributed memory paradigm is written and similar studies are conducted.**

I. Introduction

Due to limits in transistor size, number of cores on a chip, number of chips on a mother board, etc it is not always possible to speedup the program with just a better machine. Hence parallel computing is used run sections of the code in parallel thus reducing the runtime without needing a much better machine thus making the process economically beneficial along with reduction in time. Here in the following project we have tried to develop a parallel implementation of the Linear Binary Pattern Recognition (LBPH) face recognition algorithm. This algorithm is computationally simple and also offers lot of opportunities for parallelization.

Initially an all OpenMP based parallel implementation is written and studied. Some effects of wrong and redundant parallelizations are also discussed. Subsequently a more sophisticated OpenMP + MPI based hybrid program is written for parallel execution. Similar timing based studies are done to gauge the effectiveness of the parallel execution. Effect of variation of problem size on the execution time and accuracy of the algorithm is also incorporated into the study and the results are presented with appropriate insights and suggestions for improvement. Some shortcomings of the study and topics for further discussion and analysis have also been discussed.

II. Methodology

A. The LBPH algorithm

Linear Binary Pattern Histogram is an algorithm which uses local binary patterns to summarise the local structure of the image. These local patterns are unique for a person, like their facial geometry, some skin irregularities, etc. It is basically a texture analysis operator which is computationally simple with high discriminative power and robustness.

The working of the algorithm is as follows. Thresholding of the pixel values of neighbouring pixels using the centre pixel value is done into a binary 1 or 0. These binary digits are then concatenated to form a number which is subsequently converted into a decimal number and stored in the centre pixel value. This is done for all the pixels to generate a new image. The pixel values of this new image is now used to generate a histogram with a bin size of one for all the possible values. This histogram is now stored as a feature vector which is almost unique to the image.

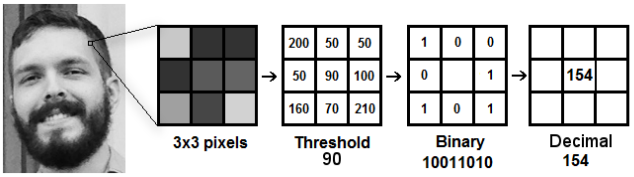

Fig. 1. Depection of the algorithm

B. Implementation

For implementation of the algorithm we have used dataset from [1]. We have selected 34 people from this dataset and taken 30 images of each individual for the dataset. The images are intially preproceesed and converted to text files with the pixel values using a Python script. This is done since it is not convenient to read image files directly into C++ while we can easily read a text file. All the images are converted to text files and stored in the directly where the C++ code can read them directly. These text files are used for the remainder of the study as a proxy for the images.

For implementation of the algorithm a window is slid over the whole image matrix. This window is a small matrix which stores the values after thresholding and then assigns the binary number to the centre and appends the number to the correct histogram bin.

This is done for all the images used on the training set and a dataset containing the feature vector of the images is made, which is a 3 dimensional matrix, with the first

index as the ID of the individual, the second index for the different images of the individual and the third index for the elements of the feature vector/histogram. Now this dataset is stored. For the testing part the images are again converted into corresponding histograms/vectors. Then the closest vector is found in the training set using some distance metric. The metric we have used in our project is the chi-squared distance metric which is:

$$\chi^2 = \frac{1}{2}\sum_{i=1}^{n}\frac{(x_i - y_i)^2}{(x_i + y_i)}$$

Using this distance metric the closest histogram is found in the training set formed and this is how face recognition is performed.

C. Parallelization

Initially a serial code for the algorithm was written and tested, which was then further parallelized. The parallelization of the code was done in steps. Firstly a pure OpenMP based parallel code was written for the algorithm. The parallel sections included the loop for creating the training dataset and the loop for finding the closest member in the training set for a test image. Some other initialisation and memory allocations and deallocation loops were also made parallel. This was the first rudimentary parallel code written for the algorithm which was further improved.

In the next stage the algorithm was parallelized using hybrid OpenMP + MPI programming. MPI processes were used to parallelize the main training loop. Hence now each process will have a subset of the training set formed and then all the processes will send their subset to the MASTER process which will now run the test image. The closet distance was still OpenMP parallel. The loop for creating the histogram which slides the window over the entire image was nor parallelized using OpenMP. This code was subsequently discarded as the need for sharing the whole training set was felt to be unnecessary, so instead each process had its own subset of the training set. The test image was uploaded to all the processes and each process shared with the MASTER the best scored and the corresponding ID with the MASTER. The MASTER then compared all the values and assigned the lowest one to the test image. This vastly reduced the amount of data shared between the processes hence speeding up the process.

In the final stretch of the study the accuracy and timing based on the number of training examples was carried out. In this the number of images per ID is varied and the accuracy and time of execution studied.

III. Results and Analysis

All the timings reported are an average of 5 consecutive execution times. All the results have been generated using the GCC 8.5 compiler and Open MPI v4.1.1 library.

The results for the timing study for the pure OpenMP base parallel code have been done on an quadcore i5 8^{th} generation machine with 8GB of RAM with hyperthreading enabled on an Ubuntu 18.04 operating system. In Table I we can see the results generated by pure OpenMP based implementation. Since the time of executions is not very large some system noise also effects the results.

TABLE I
TIMING AND SPEEDUP FOR OPENMP CODE

Number of OpenMP threads	Time	Speedup
1	4.151	1.00
2	2.140	1.94
4	1.172	3.54
6	0.807	5.14
8	0.777	5.34

Here we can see a very typical speedup in the program. Figure 2 shows the speedup vs number of threads curve. The graph slowly flattens as the number of threads increase. This is due to the maximum possible speedup as per Amdahl's law and also due to increase in the parallel overheads of thread management.

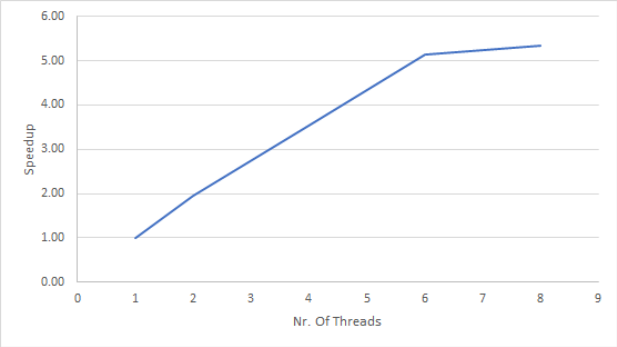

Fig. 2. Speedup achieved in pure OpenMP implementation

Another study that was done using this was that how can redundant and wrongly parallelised code can affect the performance. In this study the one of the functions which was called inside the training loop was also OpenMP parallelised. Hence we get two redundant parallel loops and the inner parallel sections does not have any available threads to spawn and just adds to the overheads causing a slowdown as seen in the figure 3

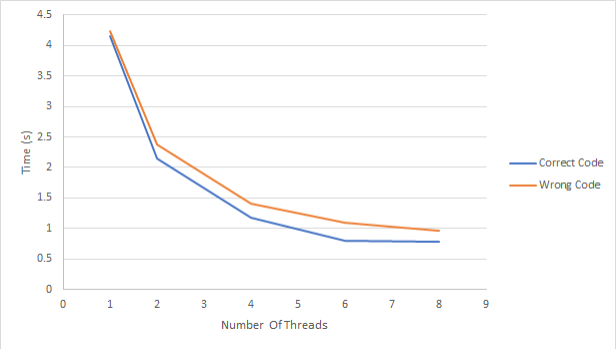

Fig. 3. Effect of wrongly written parallel implementation

The subsequent results are generated on a hexa core i7 9^{th} generation 64 bit machine with 16 GB of RAM and hyperthreading enabled . The code is run on Ubuntu 18.04 on VMWare workstation where the virtual environment is provided with 8GB of RAM and 8 cores. Since the results have been generated on a different machine direct comparison is not possible. However trends found in the studies can still be compared. The results for the OpenMP+MPI hybrid program were as seen in Tables II and III. we can see that the decrease in execution time, and hence the speedup is larger when MPI processes are increased. This phenomenon maybe because of two reasons. The first and foremost reason is that a major part of the code was parallelized using MPI and hence more amount of parallelism is linked with MPI than OpenMP and hence more speedup is seen with more MPI processes. The second reason maybe the parallel overhead of the two paradigms. It maybe so that the parallel overhead of the OpenMP thread management is more than that of the MPI message passing and hence we see smaller speedup. The second reason is a hypothesis

TABLE II
EXECUTION TIME FOR THE HYBRID CODE

OpenMP Threads MPI Processes	1	2	4	8
1	4.388	2.316	1.727	1.577
2	2.305	1.688	1.671	
4	1.381	0.989		
8	1.003			

which can be further tested by running this program in an actual distributed memory system. Running on multiple nodes connected by an interconnect, which slows down the message passing and hence we may see a different trend than what is observed here. Lastly we can see that the maximum speedup observed here is only 4.44 times as opposed to 5.34 times in Table I. This again maybe

TABLE III
SPEEDUP FOR THE HYBRID CODE

OpenMP Threads MPI Processes	1	2	4	8
1	1.00	1.89	2.54	2.78
2	1.90	2.60	2.63	
4	3.18	4.44		
8	4.37			

because of different machines used to run the programs or maybe due to added parallel overheads, which can be be found by doing both the studies on the same machine.

Lastly we see the effect of training set size on the accuracy and time of execution. The trend is what is expected. Initially as we decrease the size of the training set the dip in execution tie is large but it flatlines after sufficient amount of decrease. This is again due to two reasons. First one being dominance of parallel overheads at lower problem sizes. Second one is even though I am reducing my training set size but my test data is constantly the same. Also serial sections of the code also affect this.

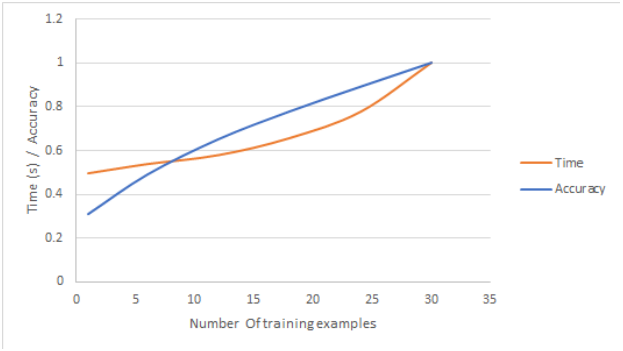

Fig. 4. Effect of training set size on execution time and accuracy

The trend observed in accuracy is also very typical. Initially the drop in accuracy due to reduction in training set is less and it increases further as the training set size is reduced further. This is simply because initially even though training samples have reduced enough samples are available for a obtain the general characteristics of the image But as the size reduces beyond a limit the samples are no longer enough to obtain the general characteristics and hence large drop in accuracy is observed.

IV. OUTLOOK

The study done here is a preliminary study and can be extended further. Running on a machine with more number of available cores will help us see behavior much more clearly. As explained above running on a

distributed memory machine will help see the actual effect of MPI on the program. The algorithm can also include GPU based parallel programming to speedup the program even more. The algorithm itself can be also extended to give better results like The image can be divided into sections and the LBPH algorithm is applied to each section separately and the final histograms are concatenated to form the feature vector. This helps preserve some of the spatial information of the pixel data along with the texture information. Lager datasets can be deployed for training and testing giving more robust results.

V. Summary

This is the project done as a part of the ME766 High Performance Scientific Computing course. Here Face recognition using the LBPH algorithm was implemented in a parallel paradigm. The algorithm is first written as a serial implementation which is subsequently parallelized. First a pure OpenMP based parallel program written and analysed based on the executions time. Here we also analyse the effect that wrong and redundant parallel code can have on the performance.

Next a hybrid OpenMP + MPI based code is written for parallel execution of the code. A similar timing and speedup study is done for this program by varying both the number of MPI processes and the OpenMP threads and the results are analysed. Here we make an untested hypothesis that MPI in this case has lower parallel overheads than OpenMP. This hypothesis is subject to further testing before being stated as a result. In the end trends in accuracy and times of execution is studied based on the size of the training set used. A decline in both is seen as expected with the trend in rate of decline as expected and explained in previous section.

VI. Contribution

Shubham Gupta - writing the OpenMP + MPI hybrid program and doing the timing study. Also helped in presentation and report writing.

Anish Satpati - Helped in writing the serial implementation as well as the pure OpenMP based parallel code. Wrote the script for image to text file conversion.

Tamoghno Kandar - Helped in writing the serial implementation as well as the pure OpenMP based parallel code along with report writing.

Tushar Dhingra - Helped in doing the timing study for the pure OpenMP code. Also helped with the presentation and report writing.

References

[1] Gary B. Huang et al. *Labeled Faces in the Wild: A Database for Studying Face Recognition in Unconstrained Environments*. Tech. rep. 07-49. University of Massachusetts, Amherst, Oct. 2007.