FACE DETECTION USING PYTHON MULTIPROCESSOR

PROJECT REPORT

For

Parallel and Distributed Computing (CSE4001)

Slot: B1

Submitted By

Name	Registration Number
Aarush Bhat	19BCE0564
Jyotir Aditya	19BCE0846

Under the guidance of

DR. Gopichand G

(SCHOOL OF COMPUTER SCIENCE AND ENGINEERING)

Dec, 2021

FACE DETECTION USING PYTHON MULTIPROCESSOR

1. ABSTRACT

Face detection using OpenCV with haar-like cascade classifier is a very common and faster method to implement. But for massive faces in one frame and long videos, it is very time consuming and slow processing. In this project, we are representing a method for parallel processing for multiple face detection and recognition in images with many faces, long videos, videos with many faces, and real time where images can be obtained from live video and compare the time with single processing. To detect the face in mentioned scenarios, we are using haar-like Cascade Classifier dataset "*haarcascade_frontalface_default.xml*" which is a xml file with features nodes of frontal face. This dataset is officially provided by OpenCV. For parallelization of the detection of multiple faces, we are using Python multiprocessor to make the training and testing faster. We also compare the execution time with single processor in the mentioned scenario.

Keywords – Face detection, Haar-like classification, Multiprocessing, Speedup, OpenCV, Video processing, Live frame processing.

2. INTRODUCTION

Face detection is a computer technology being used in a variety of applications that identify human faces in images and videos. Face detection also refers to the psychological process by which humans locate and attend to faces in a visual scene.

Face detection is one of the popular and most common topics of discussion. Face detection is used to take better photos by camera or videos. Face detection is used as preprocessing for face recognition of someone. Face detection is also used for face unlock on devices.

For face detection, haar-like classifier is a very common and fast method. With the help of OpenCV in python, we can easily implement Haar-like Cascade Classifiers and detect faces in images, videos or live streaming. But for long videos and massive faces in pictures or faces, the process becomes slow. But the process can be made efficient and faster using multiprocessing.

In this project, we are going to implement the multiprocessing in python and make the detection faster. We are going to compare the speed or execution time by implementing single processing and multiprocessing in the cases of image, video and live stream.

3. CONTRIBUTION OF THE WORK

Using only a single processor in a multiprocessing device is a kind of drawback or less efficient as other cores are idle and the work is time consuming. Parallelizing the program distributes the work among the processes and saves time.

In our work, we have taken three different scenarios of input data i.e., image, video and live stream, and have tested the time of execution with respect to utilization of single core process and two core, four cores, six cores and eight cores of multicore process. The result obtained is then compared and the conclusion is carried out on the basis of that. In the scenario of image, we have used different images with different numbers of faces. In the scenario of video, we have used different videos of different size, length, resolution and number of frames with in crowded and gathering modes. In the live scenario, we used the webcam of our laptops when I was alone and when I was with my family members.

4. RELATED WORKS

[1] Histogram of Oriented Gradient is used for extraction of the eigen faces. Proposed algorithm is 8.75% more accurate than the face recognition with PCA based face recognition algorithm. [2] This is based on a neural network which takes the input gray value of the face image. Proposed algorithm can give personal identity authentication very well which is a unique feature of humans and very hard to duplicate. [3] This paper is based on face recognition principal component analysis where eigenfaces are detected reducing the dimensionality of feature vectors. This algorithm is a very easy, simple and fast approach to implement. [4] The algorithm extracts the deep features using the 1^{st} 5 layers if Alexnet and an SVM is trained for each window size pixel to detect faces of the particular size. Then detections from all SVMs are pooled together and a single bounding box is output given by the detector. The proposed system can detect faces in mobile devices not having CUDA enabled GPUs. [5] This algorithm recognizes both front face and side face. And the gray value of the pixels in pixel matrix is replaced by the median value of its neighborhood sampling value and then the statistical histogram is obtained after extracting feature value by sub blocks. The recognition accuracy of LBPH algorithm is very high compared to other algorithms. [6] The paper is a description about the face recognition by Holistic approach where the algorithm takes the face image as inputs and extract the feature parameters. [7] The paper uses PCA MPI algorithm to reduce the time consumed by PCA. The proposed algorithm improves execution time upto 25x in training and 5x in recognition phase. [8] In this paper, a real time face recognition from a live video is proposed where the efficiency of face recognition is improved by combination of eigenface method using Haar-like features to detect both of eye and face, and Robert cross edge detector to locate human face positions for fast response time, Principal component Analysis is used to reduce the dimensionality of the training dataset. Faces of each person in a live video can be detected with high accuracy of 92% success rate. [9] This paper is a survey about 3D face recognition. The person can be recognized even in dim lights and with variant facial position and expression. [10] The proposed idea consists of multiple view-specific fast LAB cascade for extremely quick face proposal, multiple MLP cascade for candidate window verification, and a unified fine MLP with shaped-indexed features for accurate face detection. [11] In this proposal, the RCNN is improved by combining a number of strategies like feature concentration, model pretraining, hard negative mining, multi scale training and the proper calibration of key parameters. It is the best model in terms of ROC curves among all methods on FDDB benchmarks. [12] This paper not only focuses on accuracy but also on time factor and the time factor is also considered as the major problem in the real time environment. The recent architecture is represented by multi-core CPUs and manycore GPUs that provide the possibility to perform various tasks by parallel processing. It is easy to use and mass scanning of people.

5. EXISTING SYSTEM DESCRIPTION

There are many systems and technologies already existing for parallel face recognition systems. One of them is "*Parallel Architecture for Face Recognition using MPI*". In this proposal, PCA is used to reduce the dimension of extracted features by projecting images in new face space. Since PCA is a time consuming process due to its highly intensive computation nature, this paper has proposed two different architectures to accelerate training and testing phases of PCA algorithm by exploiting the benefits of distributed memory architecture. These architectures are for finding the face of a single person after extracting and comparing. Fig1. shows the general idea of the PCA system.

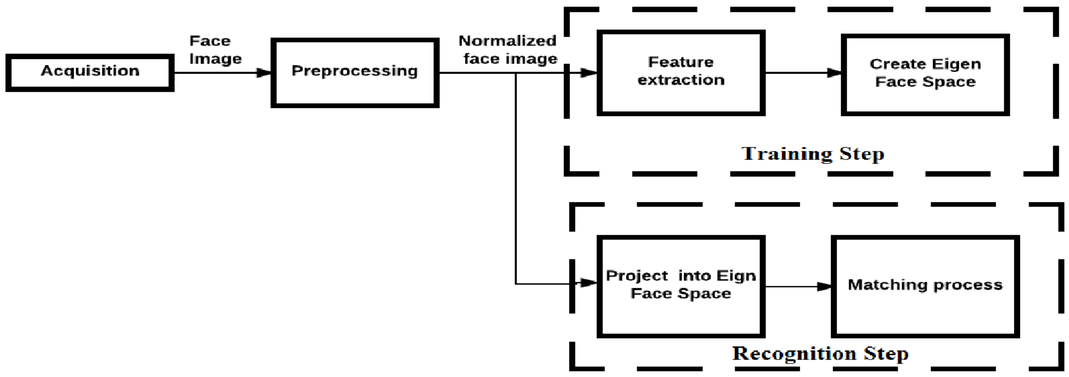


Fig. 1. PCA Recognition System

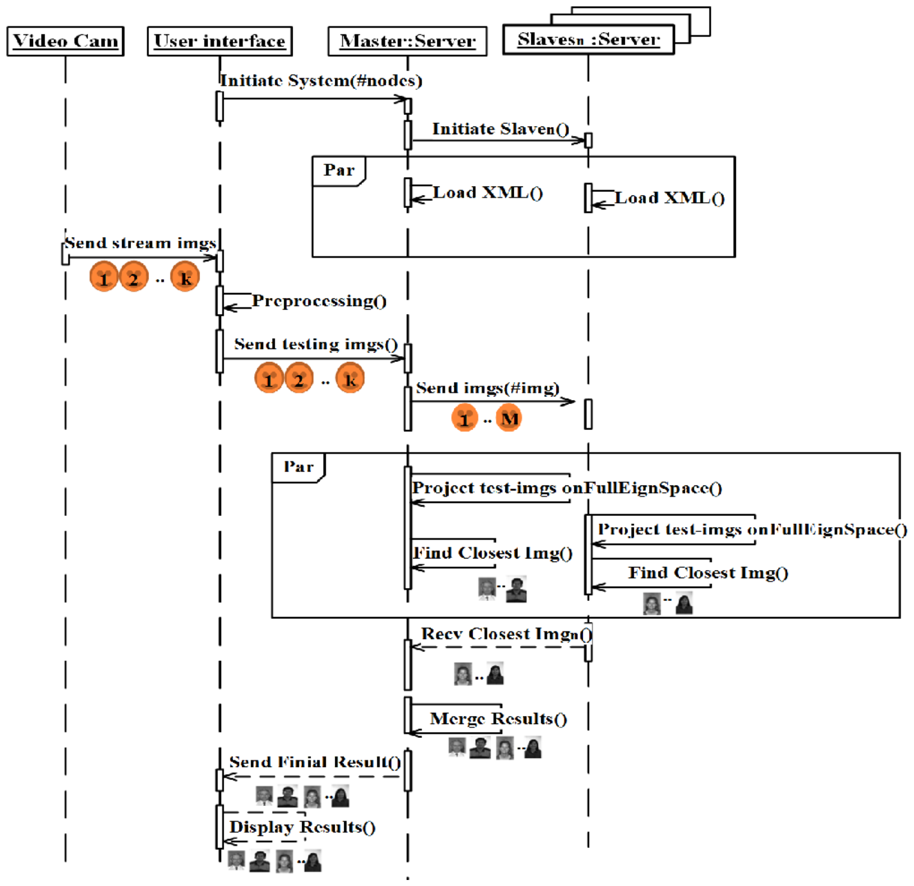


Fig 2. Sequence diagram to recognize 1 test image vs 1000 images in training DB

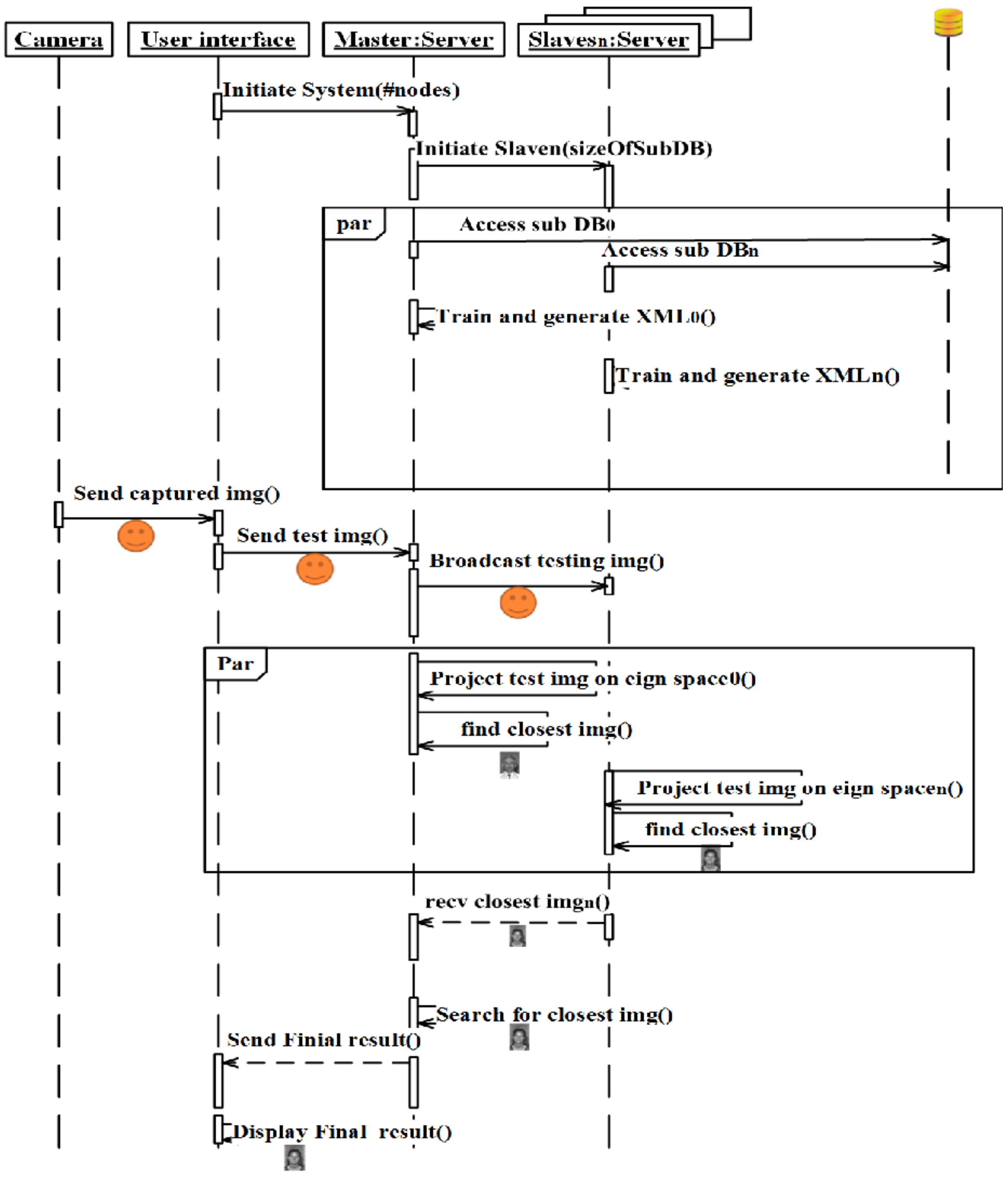


Fig. 3. Sequence diagram to recognize stream of testing images in training DB

6. DATA SET DESCRIPTION

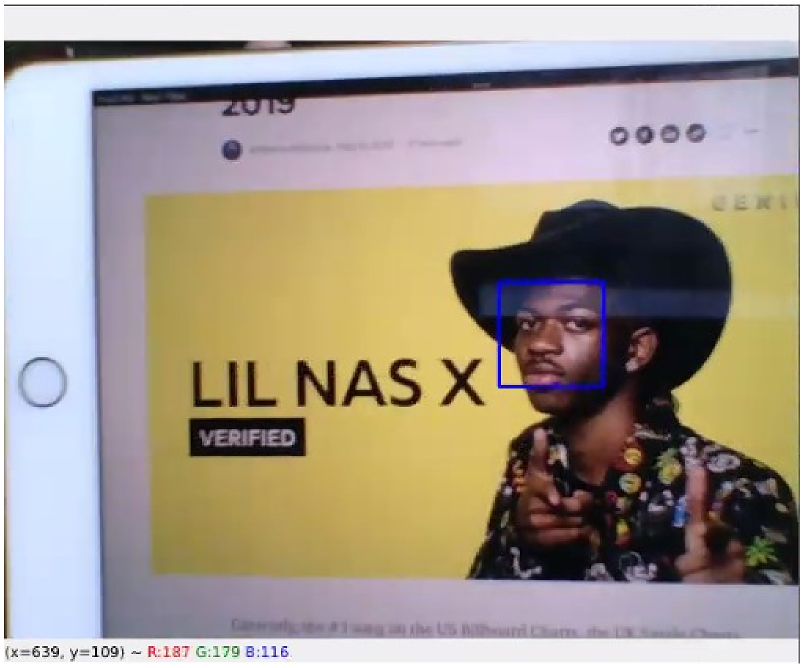
Trained Data set: *haarcascade_frontalface_default.xml* is a dataset provided by OpenCV. This xml file contains features of Faces. Since there are very large number of features, they are grouped into different stages with internalNodes and leafValues.

Link:[*https://github.com/opencv/opencv/blob/master/data/haarcascades/haarcascade_frontalface_default.xml*](https://github.com/opencv/opencv/blob/master/data/haarcascades/haarcascade_frontalface_default.xml)

Image Input Data:

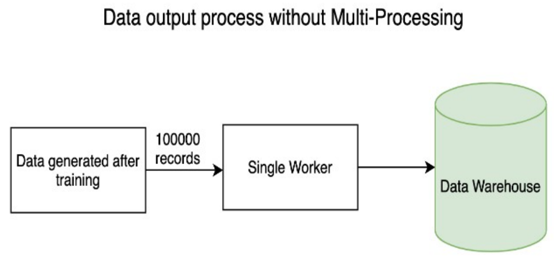
Video Input Data: Two videos, one with 162 frames, duration = 5 sec, Width = 2560, Height = 1440, and another with 3180 frames, 1:46, Width = 1280, Height = 720

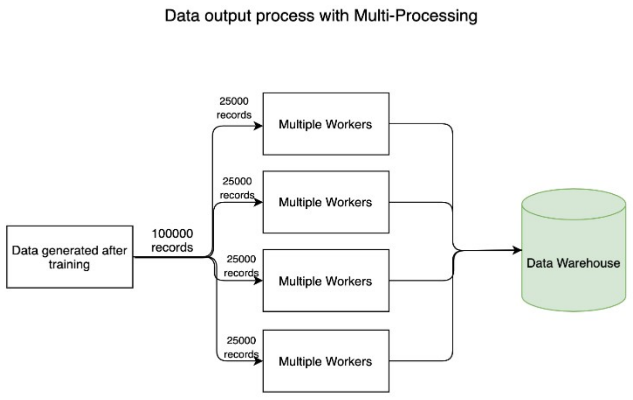
Live Input Data:



7. SYSTEM ARCHITECTURE AND MODULE WISE DESCRIPTION

COMPARATIVE ARCHITECTURE OF SINGLE AND MULTIPROCESSOR


Data output process without Multi-Processing
Data generated after training
100000 records
Single Worker
Data Warehouse


Data output process with Multi-Processing
Data generated after training
100000 records
25000 records
Multiple Workers
Data Warehouse

PYTHON MODULES

- **OpenCV:** OpenCV is a famous and most used library in python for image processing. For our project, we will be using
  - VideoCapture(args) to open image, video or cam;
  - read() to read frame,
  - cvtColor(frame, cv2.COLOR_BGR2GRAY) to convert BGR to Gray color,
  - CascadeClassifier('dataset//haarcascade_frontalface_default.xml’) to get haar-like classifier xml and detect faces.

- **Multiprocessing:** This is a python library to implement multiprocessing. Since there are so many methods to implement multiprocessing, we are going to explain a few related and used methods or functions.
  - Multiprocessing.cpu_count()
  - Pool(no_of_processors) to distribute the data to multiple processors (data parallelism).

- Process(targetFunction, args) to parallelize a function with arguments. This method is also used to divide the processor's works.

The Pool is used to create objects. The parallelization can be implemented using the following functions.

- map(func, inputDataList) to divide inputDataList into every processor as a single argument in the function func.
- Startmap(func, dataList) is similar to map but every element in dataList is unpacked as an argument.

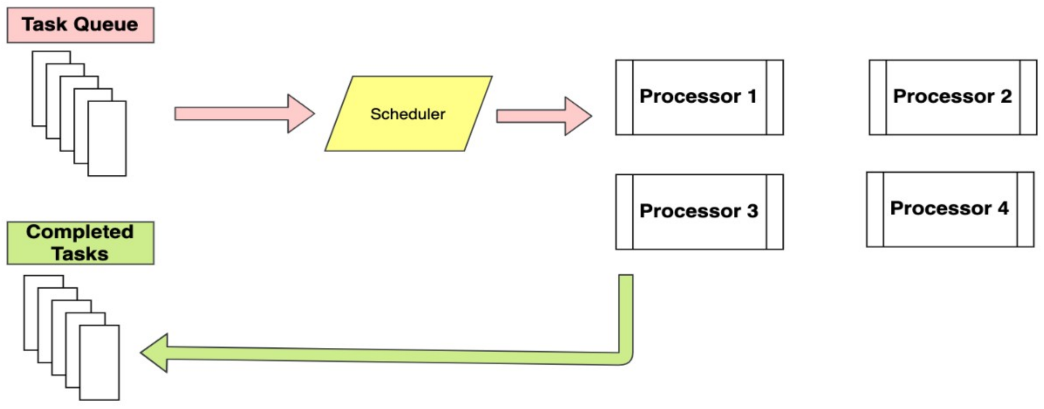
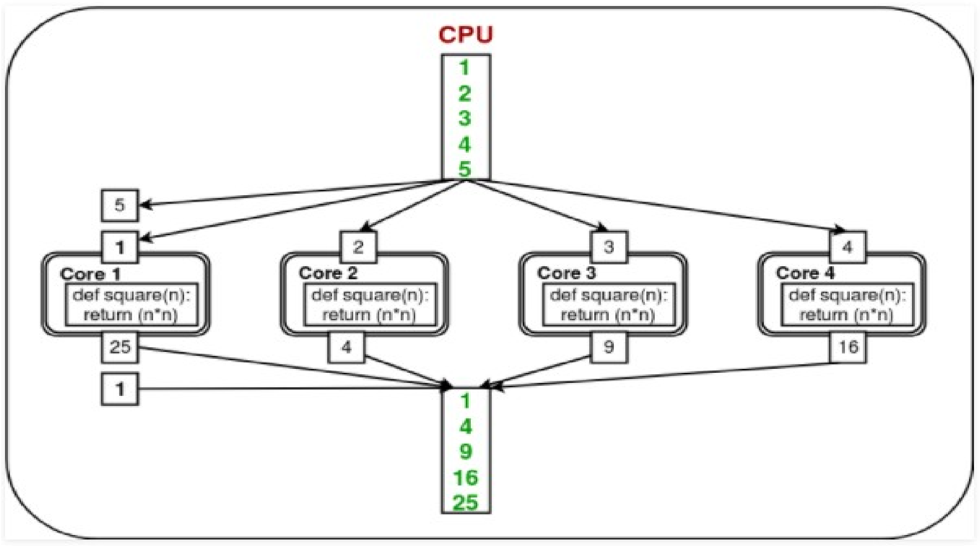


Fig. pool in python multiprocessing



Multiprocessing.Pool(4)

Pool.map(square, [1,2,3,4,5])

Fig. Multiprocessing

IMPLEMENTATION OF PARALLELIZATION

- To implement the parallelism in face recognition, the following strategies are used in different cases.

- **Image/ live stream:** for images with fewer faces, single processors are faster since creating threads for multiprocessing is a slower task. For massive faces in an image, the process of identifying faces by drawing squares is parallelized using concurrent.Future library. Live is similar to taking images as input continuously in real-time. So, the implementation is similar to the image.

- **Video:** video is a collection of images called frames and it streams to the last frame. So, parallelizing for video is done using data parallelism.

    - First split the video into a collection of frames.
    - Each collection is parallelized with different processors.
    - Merge the collections again.
    - Each collection can be multithreaded to make the process even faster.

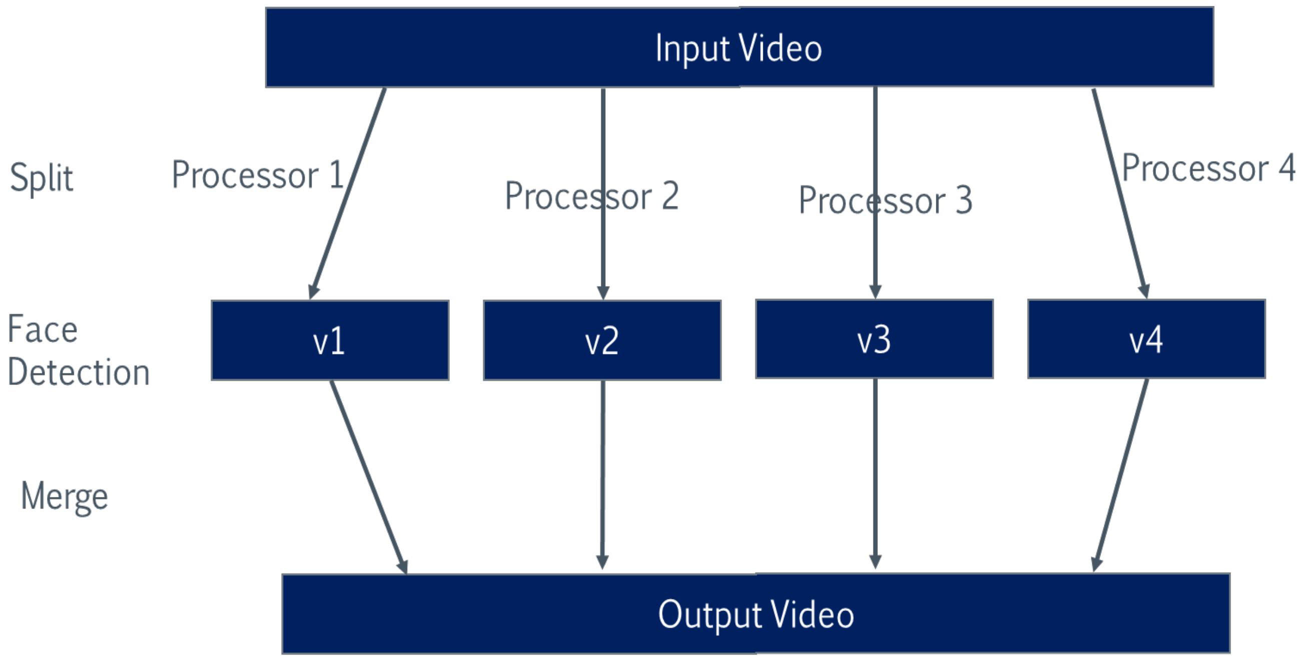


Fig. face detection in video

COMPARISON METHODOLOGY OF PERFORMANCE

The performance of the programs is compared based on total execution time and frame per second. In the scenario of image, execution time and no of faces per second are two comparison parameters. In the scenario of video, total execution time and frame per second are two comparison parameters. In the case of live streams, the total execution time is invalid so only frame per seconds. Since there is no total frame in the live scenario, we got execution time t of 10 frames and then calculated fps by fps = (10/t).

For calculating efficiency, the following formulas are used.

$$efficiency \in time = \frac{\frac{1}{t_{multicore}} - \frac{1}{t_{singlecore}}}{\frac{1}{t_{singlecore}}} = \frac{t_{singlecore} - t_{multicore}}{t_{multicore}}$$

$$efficiency \in fps = \frac{fps_{multicore} - fps_{singlecore}}{fps_{singlecore}}$$

8. RESULTS

The Code repository can be found here https://github.com/sloorush/parallel-face-detection.

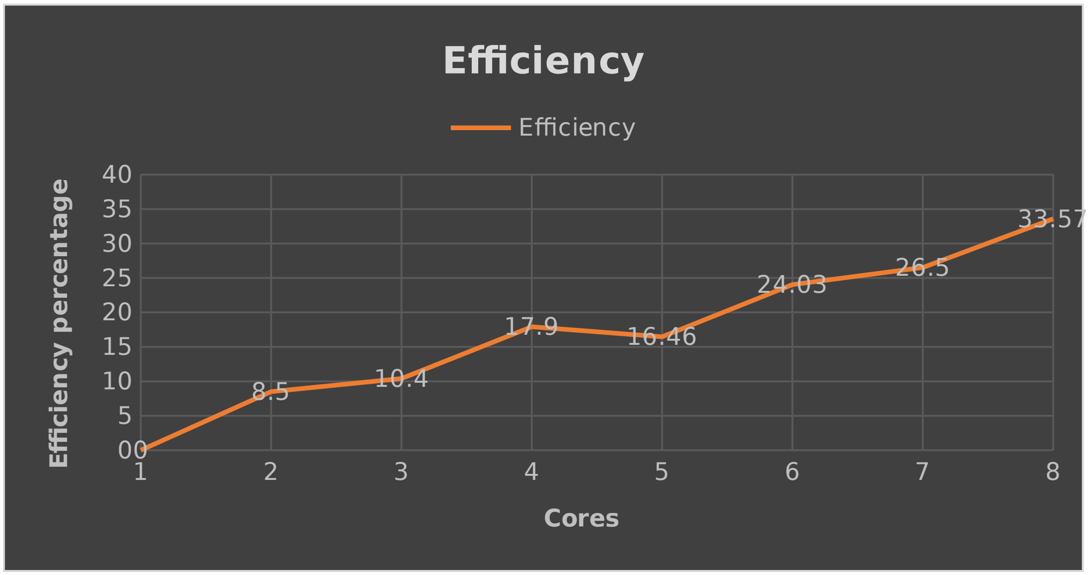
To compare the performance, we have configured the following setup.

PLATFORM DETAILS

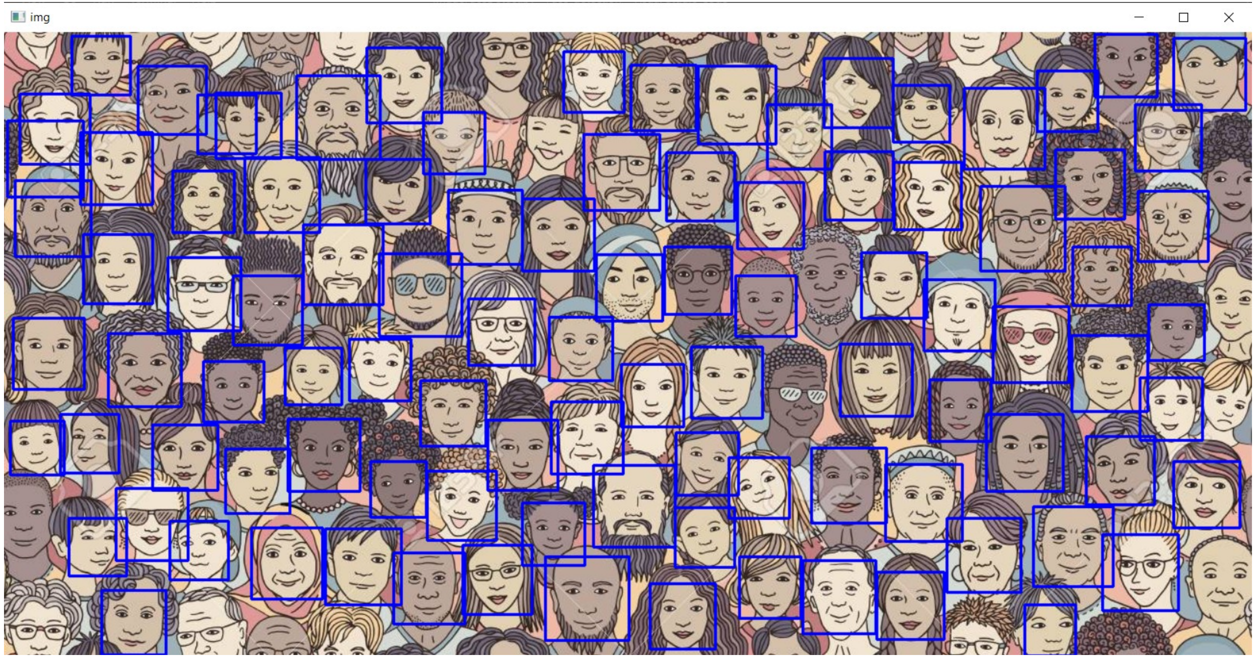
PC: Asus TUF fx505
Core: 8 cores
CPU: Ryzen 7
Graphics Card Type: Dedicated graphics card
Graphics Card Model: GeForce GTX1650
Graphics Card RAM: 4GB
RAM: 16GB
OS: Linux Mint 20.2 x86_64
Environment: Python 3.8.10
OpenCV Version: 4.5.2
Editor: Visual Studio Code

We have executed the python code and the input detail and output are shown in the table.

Image (98 faces)		
CPU CORES	**Average Execution Time**	**Efficiency compares to single core**
1	1.91 sec	0%
2	1.76 sec	8.5%
3	1.73 sec	10.40%
4	1.62 sec	17.9%
5	1.64 sec	16.46%
6	1.54 sec	24.03%
7	1.51 sec	26.5%
8	1.43 sec	33.57%

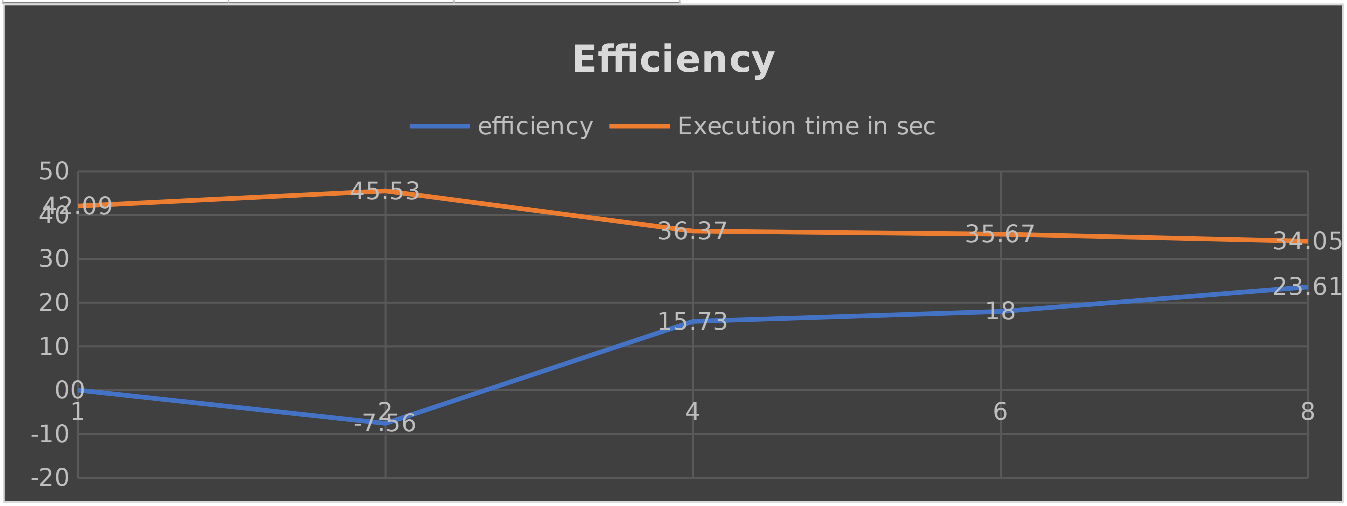


Output:

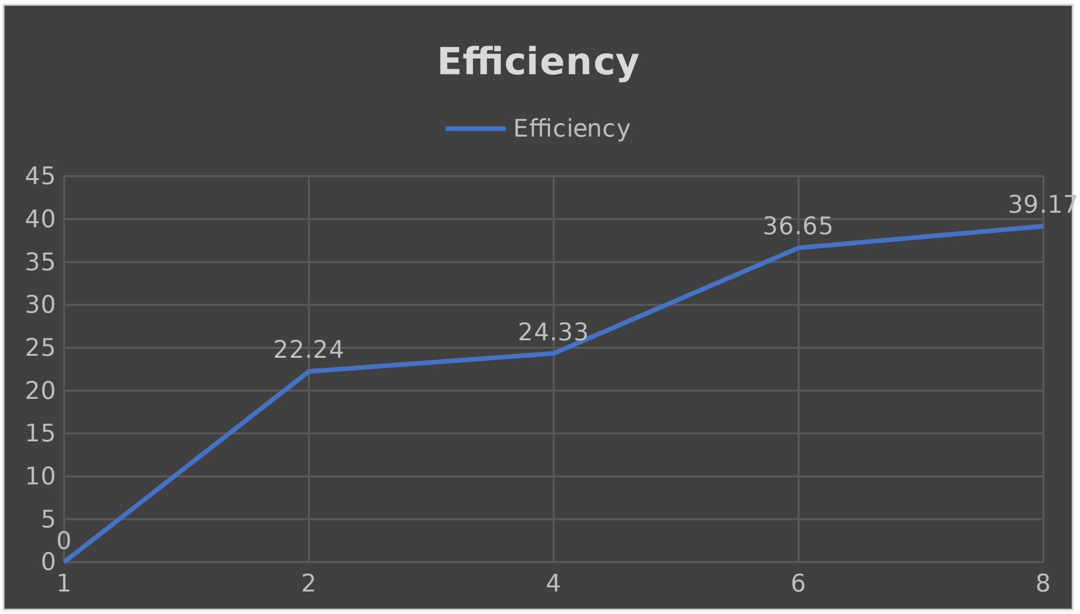


**Video
162 frames,
Duration = 5 sec,
Width = 2560, Height = 1440**		

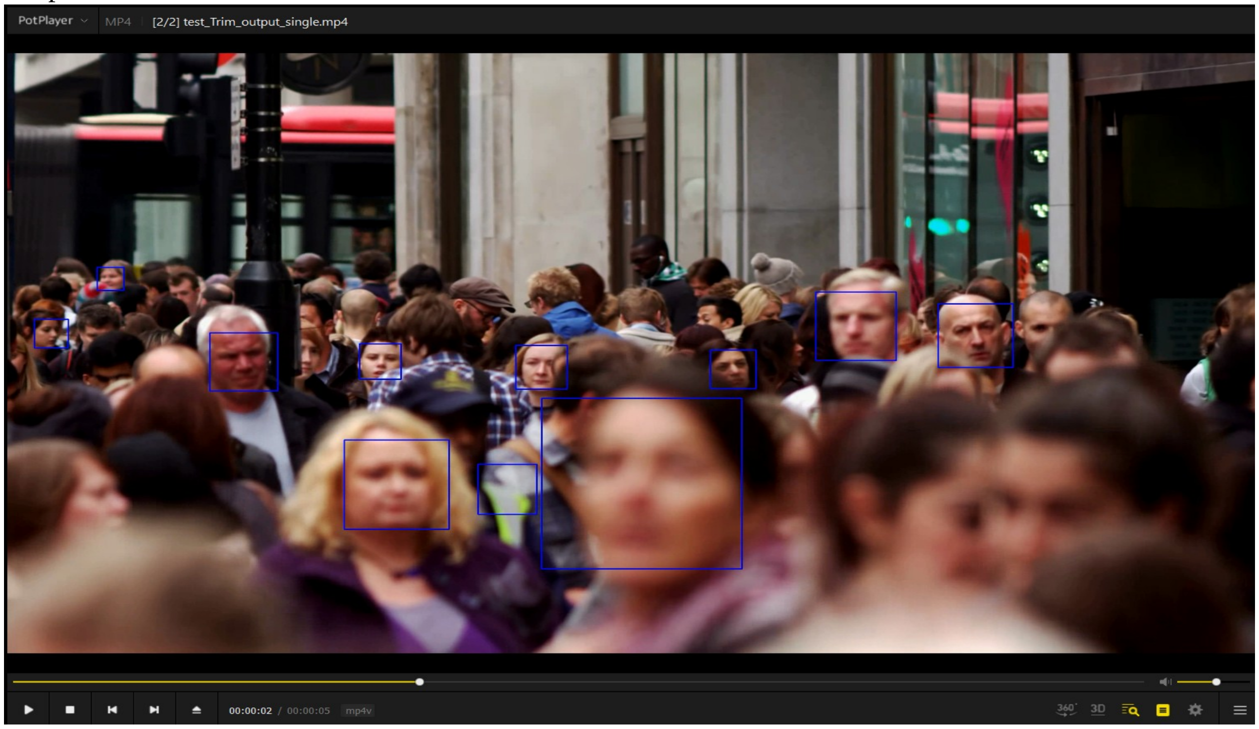
CPU CORES	**Average Execution Time**	**Efficiency compares to single core**
1	42.09 sec 3.84 fps	0%
2	45.53 sec 3.56 fps	-7.56%
4	36.37 sec 4.45 fps	15.73%
6	35.67 sec 4.54 fps	18.00%
8	34.05 sec 4.76 fps	23.61%



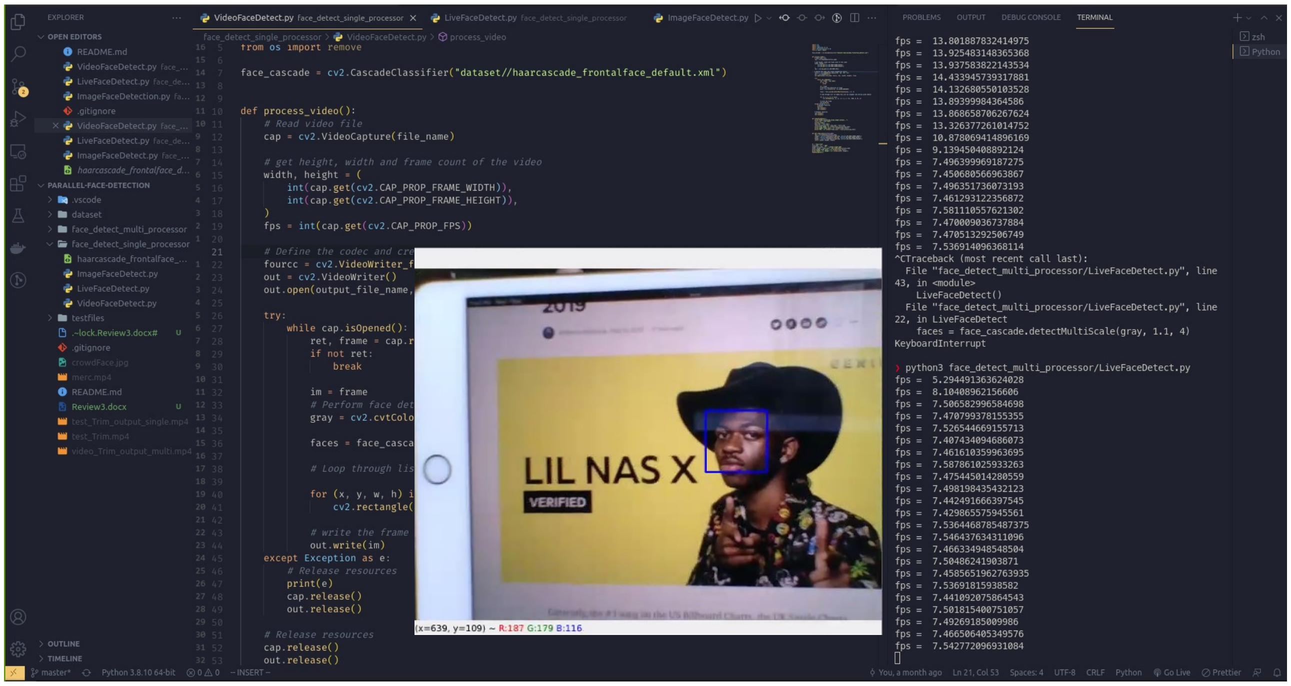
Video 3180 frames duration = 1:46, Width = 1280, Height = 720		
CPU CORES	**Average Execution Time**	**Efficiency compares to single core**
1	177.65 sec 17.9 fps	0%
2	145.33 sec 21.88 fps	22.24%
4	142.89 sec 22.25 fps	24.33%
6	130.00 sec 24.46 fps	36.65%
8	127.65 sec 24.46 fps	39.17%



Output:



Live:



```
face_detect_single_processor > VideoFaceDetect.py > process_video
from os import remove

face_cascade = cv2.CascadeClassifier("dataset//haarcascade_frontalface_default.xml")


def process_video():
    # Read video file
    cap = cv2.VideoCapture(file_name)

    # get height, width and frame count of the video
    width, height = (
        int(cap.get(cv2.CAP_PROP_FRAME_WIDTH)),
        int(cap.get(cv2.CAP_PROP_FRAME_HEIGHT)),
    )
    fps = int(cap.get(cv2.CAP_PROP_FPS))

    # Define the codec and cre
    fourcc = cv2.VideoWriter_f
    out = cv2.VideoWriter()
    out.open(output_file_name,

    try:
        while cap.isOpened():
            ret, frame = cap.r
            if not ret:
                break

            im = frame
            # Perform face det
            gray = cv2.cvtColo

            faces = face_casca

            # Loop through lis

            for (x, y, w, h) i
                cv2.rectangle(

            # write the frame
            out.write(im)
    except Exception as e:
        # Release resources
        print(e)
        cap.release()
        out.release()

    # Release resources
    cap.release()
    out.release()
```

```
fps =  13.801887832414975
fps =  13.925483148365368
fps =  13.937583822143534
fps =  14.433945739317881
fps =  14.132680550103528
fps =  13.89399984364586
fps =  13.868658706267624
fps =  13.326377261014752
fps =  10.878069414896169
fps =  9.139450408892124
fps =  7.496399969187275
fps =  7.450680566963867
fps =  7.496351736073193
fps =  7.461293122356872
fps =  7.581110557621302
fps =  7.470009036737884
fps =  7.470513292506749
fps =  7.536914096368114
^CTraceback (most recent call last):
  File "face_detect_multi_processor/LiveFaceDetect.py", line
43, in <module>
    LiveFaceDetect()
  File "face_detect_multi_processor/LiveFaceDetect.py", line
22, in LiveFaceDetect
    faces = face_cascade.detectMultiScale(gray, 1.1, 4)
KeyboardInterrupt

❯ python3 face_detect_multi_processor/LiveFaceDetect.py
fps =  5.294491363624028
fps =  8.10408962156606
fps =  7.506582996584698
fps =  7.470799378155355
fps =  7.526544669155713
fps =  7.407434094686073
fps =  7.461610359963695
fps =  7.587861025933263
fps =  7.475445014280559
fps =  7.498198435432123
fps =  7.442491666397545
fps =  7.429865575945561
fps =  7.5364468785487375
fps =  7.546437634311096
fps =  7.466334948548504
fps =  7.50486241903871
fps =  7.4585651962763935
fps =  7.53691815938582
fps =  7.441092075864543
fps =  7.501815400751057
fps =  7.49269185009986
fps =  7.466506405349576
fps =  7.542772096931084
```

9. CONCLUSION

The time taken by multi core to detect faces is less as compared to that by single core to do the same. The pooling in multiprocessing and concurrency of python distributes the data across the cores and hence the time reduces. The efficiency of the algorithms increases with increase in number of faces in image, number of faces in live stream, number of frames in video, the length of the video.

10. REFERENCES

[1] Dadi, H. S., & Pillutla, G. M. (2016). Improved face recognition rate using HOG features and SVM classifier. *IOSR Journal of Electronics and Communication Engineering*, *11*(04), 34-44.

[2] Zhi, H., & Liu, S. (2019). Face recognition based on genetic algorithm. *Journal of Visual Communication and Image Representation*, *58*, 495-502.

[3] Kaur, R., & Himanshi, E. (2015, June). Face recognition using principal component analysis. In *2015 IEEE international advance computing conference (IACC)* (pp. 585-589). IEEE.

[4] Sarkar, S., Patel, V. M., & Chellappa, R. (2016, February). Deep feature-based face detection on mobile devices. In *2016 IEEE International Conference on Identity, Security and Behavior Analysis (ISBA)* (pp. 1-8). IEEE.

[5] X. Zhao and C. Wei, "A real-time face recognition system based on the improved LBPH algorithm," 2017 IEEE 2nd International Conference on Signal and Image Processing (ICSIP), Singapore, 2017, pp. 72-76, doi: 10.1109/SIPROCESS.2017.8124508.

[6] ISSN 1013-5316; CODEN: SINTE 8

[7] Ibrahim, D. S., & Hamdy, S. (2017). Parallel architecture for face recognition using MPI. *International Journal of Advanced Computer Science and Applications (IJACSA)*, *8*(1), 425-430.

[8] Varun Pande, Khaled Elleithy, Laiali Almazaydeh. Parallel Processing for Multi Face Detection and Recognition. Department of Computer Science and Engineering University of BridgeportBridgeport, CT 06604, USA{vpande, elleithy, lalmazay}@bridgeport.edu

[9] Zhou, S., Xiao, S. 3D face recognition: a survey. Hum. Cent. Comput. Inf. Sci. 8, 35 (2018). https://doi.org/10.1186/s13673-018-0157-2

[10] Wu, S., Kan, M., He, Z., Shan, S., & Chen, X. (2017). Funnel-structured cascade for multi-view face detection with alignment-awareness. Neurocomputing, 221, 138-145.

[11] Sun, X., Wu, P., & Hoi, S. C. (2018). Face detection using deep learning: An improved faster RCNN approach. Neurocomputing, 299, 42-50.