ISSN 2007-9737

Parallelization Strategy Using Lustre and MPI for Face Detection in HPC Cluster: A Case Study

Hugo Eduardo Camacho Cruz, Julio Cesar González Mariño,
Jesús Humberto Foullon Peña

Universidad Autónoma de Tamaulipas,
Laboratorio de Optimización y Altas Prestaciones / Facultad de Medicina e Ingeniería en Sistemas Computacionales de Matamoros,
Mexico

{hcamachoc, jmarino}@docentes.uat.edu.mx, foullon@gmail.com

Abstract. The hardware requirements in object detection systems make these applications a challenge in their development given the high consumption of processing and memory they require for their execution. The detection of certain characteristics; in the case of a face, the profile, as well as the lighting, the distances and the numbers of objects are factors that influence the proper functioning and performance of these implementations. This paper presents an alternative to solve part of this problem through a parallelization strategy using Luster and MPI-IO for face detection in the HPC Cluster. We compare Dlib and OpenCV with our alternative based on the Viola-Jones algorithm called *Facedetector_MPI.* The tests were executed in HPC cluster with 7 nodes (1 metadata server, 2 object storage server and 2 and 4 lustre clients) and we used images since 4 to 148 faces. The results showed an important reduction in the read time of the image file compared with OpenCV of about 50% when the files are bigger to the stripe size(>1MB). Better is the increase obtained in processing around double compared with Dlib in the large images(>2Mpx) without greatly affecting the hit rate in the face detection.

Keywords. Lustre, MPI, face detector, Viola-Jones algorithm, OpenCV, parallel.

1 Introduction

Nowadays, computer object detection systems on digital images or videos are increasing, thanks to the multiple applications that can be given to them in our daily lives, from detecting the type of clothing of a person, the furniture of a house, even the kind of pets or plants around us. A case in the detection of objects is the detection of faces, some implementations such as those found in [1, 2] can detect and recognize faces, however their privacy makes them not easy to study.

During the last years a wide variety of algorithms in face detection have been developed and published, among those most cited for their comparison in performance and accuracy are those based on the work of Viola-Jones [3] and neural networks [4]. The algorithm proposed by Viola-Jones, is based on extracting Haar-like features that vary in size (width and height). Depending on the sum of dark or light areas, you can identify different parts of the face such as eyes, nose, lips, among others.

Proposals, such as those found in [5], combine HOG (Histogram of Oriented Gradient) and SVM (Support Vector Machine) to obtain precision and speed in the detection of faces. In order to improve the detection of faces, the use of convolutional neural networks (CNN) has been implemented [6], having the detection at an equal speed to the one based on HOG, nevertheless the limitations continue because it requires running on a GPU, given the need for greater computing power because the size of the image is scaled to be able to perform detection on small faces.

We must consider that the face detection rate depends on certain characteristics, that is, if there is a poor lighting condition, if it moves away, or even if part of it is obstructed; being factors that influence the consumption of computational resources such as processing and memory becoming an inconvenience for some devices or equipment.

Although part of these requirements can be met by the implementation of multiple technologies such as GPUs, this is not available to everyone because of their high costs.

In the literature, there are some works that highlight the parallelization of applications. There is alternatives available are the use of shared memory programming libraries [7], similar to the one used by OpenCV [8], where workload is distributed in the multiple CPU cores of a node through threads. Another type of parallelization is the distributed memory in which the libraries of the message passing interface (MPI) are used [9]; a program is replicated in multiple nodes of a computer cluster, and each one of them executes a specific task where the preliminary result is sent to the master node to show the final result. The inclusion of many processors allows operations to be executed in less time. It can also take advantage of parallel storage systems, as is the case of[10, 11], these systems distribute the data in the multiple storage devices of a cluster in order to get much faster I/O operations. However, most of the works found related to face detection implement the parallelism at the level of cpu or gpu using a single node.

In this paper, we present a parallel strategy for face detection on an HPC cluster. Our proposal uses the OpenCV open source computer vision and machine learning library [12], on which an optimization alternative based on the Viola-Jones algorithm was developed. This motivates us and allows us to establish the guideline to take advantage of the existing hardware, taking to a greater level the locality of data on a computer cluster through the Lustre file system, as well as reducing the execution times in the detection of faces through distributed processing by the implementation of the MPI message passing interface. Therefore, we try to answer the following research question:

It is possible to solve the requirements of memory and processing in the detection of faces exploiting the existing hardware through the increase of parallelism without compromising the hit rate? The rest of this article is organized as follows: Section 2 includes the data and methods used. Section 3 presents and discusses the results found, and, finally, in section 4 we present the conclusions.

2 Data and Methods

The task of detecting a face begins through access to the image that is distributed in small pieces in the Lustre. Then once the image is accessed, the master or principal node (rank 0) has the task of dispersing fragments of the image in the processing nodes through the MPI. Each of these processes executes our implementation (Facedetector_MPI). Finally, face detection is carried out and the result is sent back to the principal node, which is responsible of showing the total number of faces detected. Our work consists of 4 main steps: the implementation of an HPC Cluster, the data location, the operation of Facedetector_MPI, and the detection of faces.

2.1 HPC Cluster Architecture

The first step is the implementation of an HPC cluster. Initially it was built with the following characteristics: 7 nodes with Intel Core i7 processors at 3.6 Ghz, 12GB of DDR4 memory, SATA III hard disk at 2 TB at 7200 RPM and average reading speed at 156 MB/s, a GigabitEthernet network card and a 24-port Gigabit Ethernet switch. CentOS version 7 was implemented as the operating system. As well as the xfs local file system and the Lustre parallel file system in conjunction with ext4 (figure 1)

The xfs was used for the cases where the CPU implementations that by default come with the Dlib [5] and OpenCV frameworks are compared; It is noteworthy that here there is no modification to the applications, so the image is stored and processed in a single node, thus preventing access to remote servers.

Part of our parallel strategy is the data location, therefore we define the use of Lustre as a parallel file system and likewise, a collection of images[1] of multiple sizes was used. The following configuration was implemented:

- 1 Metadata and Management Server (MDS/MGS).
- 2 Object Storage Server (OSSs)
- 1, 2 and 4 Clients
- Stripe Size of 1MB.

ISSN 2007-9737

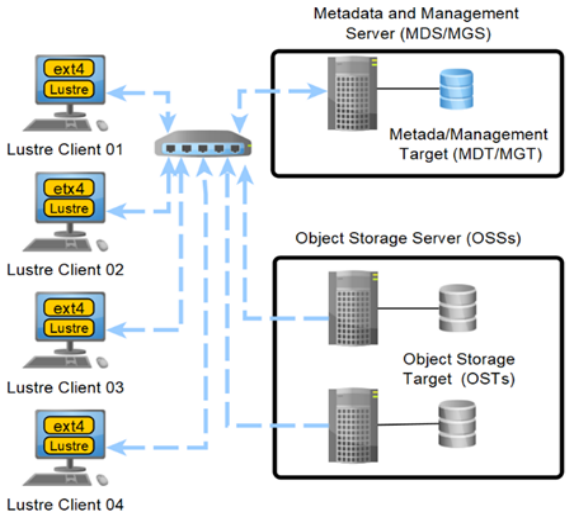


Fig. 1. HPC Cluster Architecture

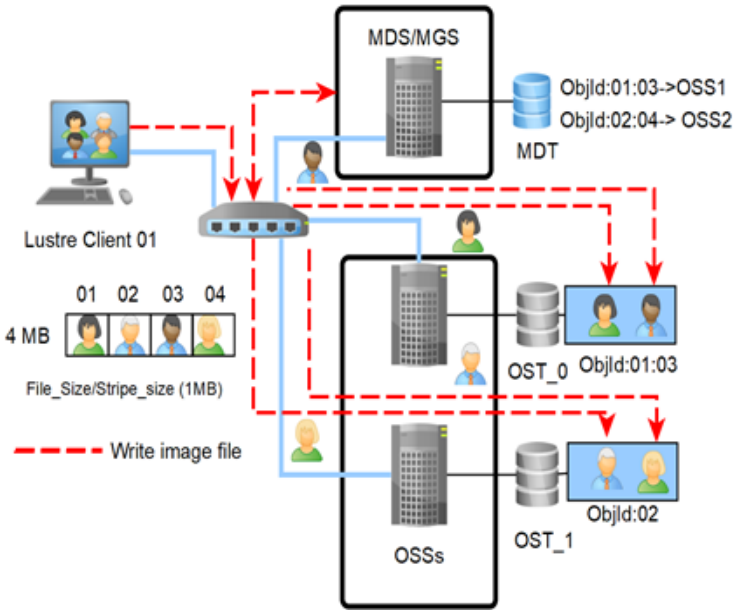


Fig. 2. Write image file

2.2 Data Location

In this second step, the goal is to distribute the image across multiple data servers. Lustre operations are implemented using a client/server model where clients send Input/Output operation initially to metadata servers to obtain the parameters of the objects (offset, size, id and others) as well as the type of operation (write/read) to be executed.

If a write operation is requested, lustre allows the image located in the client's local buffer to be sent in parallel across multiple object storage targets (OSTs) in a round-robin fashion (Figure 2).

That is, an image can be divided into multiple pieces (1) that will be sent and stored in different OSTs within the luster system.

Once the image is located on disk, it can be accessed through a read operation directly from the OSTs:

$$chunk\_img(x) = \frac{Size_{img}}{Unit_{stripping}}. \quad (1)$$

2.3 Facedetector_MPI Description

a. Reading Image

The third step aims to describe the application developed. Our tool is responsible for carrying out an operation of reading the image to the object servers, this request is made in grayscale (figure 3) regardless of whether it is color or not. It is noteworthy that we chose to perform the operation using a single 8-bit channel instead of the three channels (RGB) used by other tools; since it allows us to avoid any inconvenience when manipulating the pixels, in addition to reducing the access time to it. Remember that an image is just an array where each cell represents a pixel, so reading the image is saved in a variable of that type.

b. Scaling Image

Once the image is loaded in gray scale, the size is extracted in pixels (width and height). Subsequently, one of the contributions that improve our tool is the implementation of a function called scalar_img(). This function is responsible for scaling the image to a factor of 2 when the image is small (≤1920 and ≤1080).

This allows the face detector to search for faces around 30x30 pixels allowing to reduce the search time and improving the level of accuracy. The second occurs when the images are of a larger size (≥4096 X ≥2160), then a factor of 0.5 is used, which will reduce the image to 50% of its original size. If none of the above conditions are met, the image remains unchanged. A look at the scaling algorithm can be seen in the algorithm 1.

c. Image Dispersion

A further contribution on this work is to allow an increase in the parallelism at the process level. The use of the message passing interface library is implemented for this matter; that is, the application is replicated on the multiple processing nodes.

The algorithm 1. scalar_img description

```
1.  scalar_img(image)
2.  {
3.   /*small image 1920X1080*/
4.   if(width<=1920 and
5.   height<=1080)
6.   {
7.  resize(image size factor of
8.   2 in W and H)
9.    } /*big image*/
10.    else if(width>=4096 and
11.          height>2160)
12.          {
13.     resize(image size factor of 0.5
14.          in W and H)
15.          }else{
16.               /*original size*/
17.               Return image;
18.               }
19.  return image;
20. }
```

The division of the image (size_img/ num_procs) is given horizontally (figure 4a). This is due to the fact that such division allows to maintain a greater number of traits or particular characteristics such as the eyes and nose of a face compared to a vertical division (figure 4b), where the features or signs of the faces can be affected by the cuts of the image (figure 5), which is translated into an increase of false negatives. In other words the faces are not detected even when they are present so the success rate is in the detection of faces is reduced.

A scatter routine (figure 6) is used to designated root process (lustre client 01) sending data to all lustre clients (processes) in a communicator, in order to sends chunks of an image to each node to process the chunk of the received image.

2.4 Face Detection

In this step, the objective is to explain how face detection is carried out. It takes advantage of the multi-threading offered by MPI, a big advantage of MPI is the ability to specify noncontiguous accesses in memory, so the parallelization is even

greater. Each of the nodes executes the face detection algorithm on the piece of image that was assigned to it.

The Haar-like feature classifier was used for front faces. It is noteworthy that such a classifier is included in the OpenCV implementation and was previously trained with multiple images.

It should be noted that the process for training a classifier is that the sample images called positive include these characteristics, as well as different pixel sizes.

In addition, to complement the training it is necessary to use images that do not contain faces. For the extraction of the Haar characteristics, each of these has a unique value that is obtained by subtracting the sum of the light parts from the sum of the dark parts, with which eyes, nose and lips can be identified.

To reduce the computational cost required, an integral image [3] is implemented, like Crow's summed area table [13]. Wherein, the value of i (x, y) in the summed area table is the sum of all the pixels to the left and above (x, y), see equation (2):

$$ii(x,y) = \sum_{\substack{x' \le x \\ y' \le y}} i(x',y') \ . \tag{2}$$

After loading the classifier and generating the matrix where the image is stored, a new vector is created where the detected faces will be stored. In order to calculate the minimum requirements to preserve the possible faces detected, the minimum and maximum size required (30x30) is defined. The algorithm detects faces of different sizes on the image and draws a box on the detected face.

A face outside the indicated parameters is ignored. The preliminary result is sent to the master or main node to show the result (figure 7) through a gather routine which takes elements from each clients (processes) and gathers them to the root process. Subsequently these are ordered by the rank of the process that each of them received.

The results obtained will show the total number of faces detected (FD), as well as the possible false positives (FP) and false negatives (FN) of each image; This allows us to calculate the success rate (HR), using the following expressions (3, 4), where A is the number of successes:

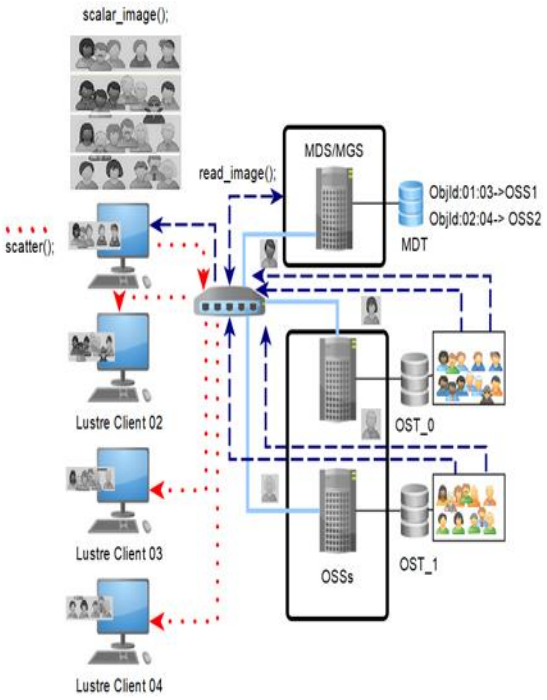


Fig. 3. Read Image file

Fig. 4. Image dispersion

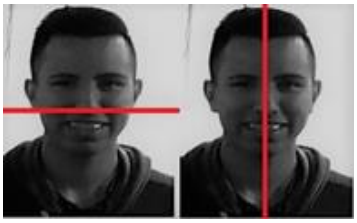


Fig. 5. Division of a face

$$A = FD - FP, \tag{3}$$

$$HR = \frac{\mathrm{A}}{\mathrm{TF}}. \tag{4}$$

3 Results and Discussion

The first objective has been to justify the inclusion of lustre to increase the location of data on remote servers. Thus, our second objective was to promote parallel processing on the multiple nodes of an HPC cluster in order to obtain results in face detection in a shorter time and without reducing the effectiveness rate.

To obtain the first results, in the case of Dlib and OpenCV, a single node running on the local XFS file system was used. In Facedetector_MPI, the Lustre configuration of 1 metadata server, 2 data servers, 1, 2 to 4 clients, as well as a 1MB stripe size was used. The samples of images used contain different file sizes for reading operations we consider accesses from 138KB to 20MB. To test the results, the average of 5 executions per image was obtained.

3.1 Image Read Time Performance

In the executions made in xfs (Dlib and OpenCV) the client accesses his local filesystem; There is therefore no remote access. In contrast, in executions in Lustre (Facedetector_MPI), the client accesses two remote servers (table 1).

As you can see the reading accesses with Lustre (figure 8 and 9) in the case of small files (<1MB) present a significant penalty against local accesses, so there is no significant improvement. The results shown in the figures do not reflect the advantage of accessing multiple servers in parallel. It should be noted that in the tests performed the servers only attend to the client's requests, so they are not running any other application.

Therefore, there is the conviction that to increase the performance to a better extent the implementation of a greater number of servers is necessary and this is noticed when the image size is larger than the stripping unit (1MB).

An increase in bandwidth is seen; This is due to the distribution of the image on 2 servers, since we will have multiple processes in parallel, simultaneously accessing part of the image, reducing access times by approximately 50% compared to the implementation in OpenCV and

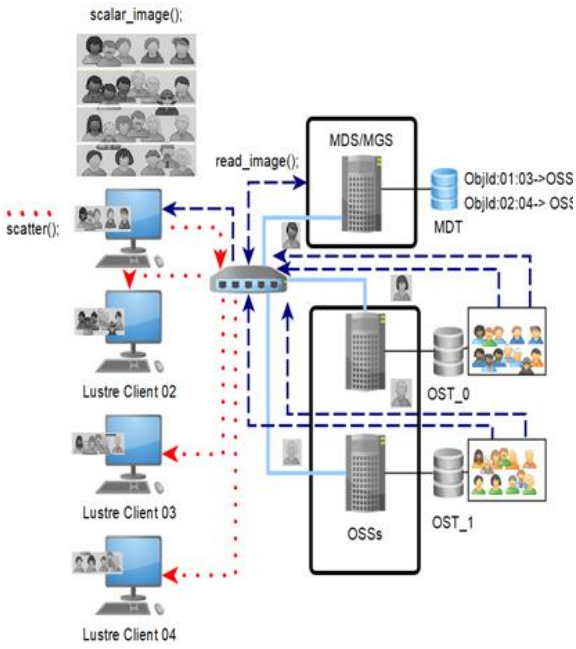


Fig. 6. Scatter routine

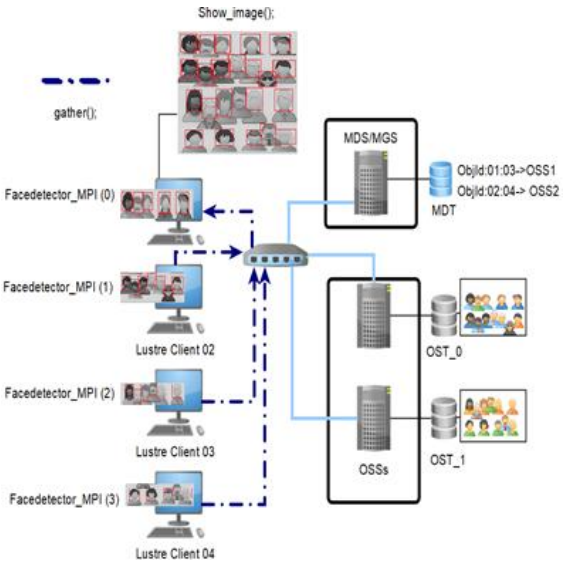


Fig. 7. Gather routine

Table 1. image read times

		Times (miliseconds)		
IMAGE	**FILE SIZE**	**DLIB**	**OPENCV**	**FACEDETECTOR_MPI_2N**
gpl01	138KB	4.4	15.55	14.61
gpl02	170KB	12.8	25.90	26.71
isum	207KB	5	17.98	12.00
gpl03	456KB	22.4	50.24	48.25
student01	636KB	10.6	8.55	14.50
student02	2.67MB	53.4	41.49	50.57
graduation	3.72MB	211.8	557.77	226.04
astro	6.84MB	389.4	1136.9	420.7
stair	19.9MB	376	1004.43	506.38

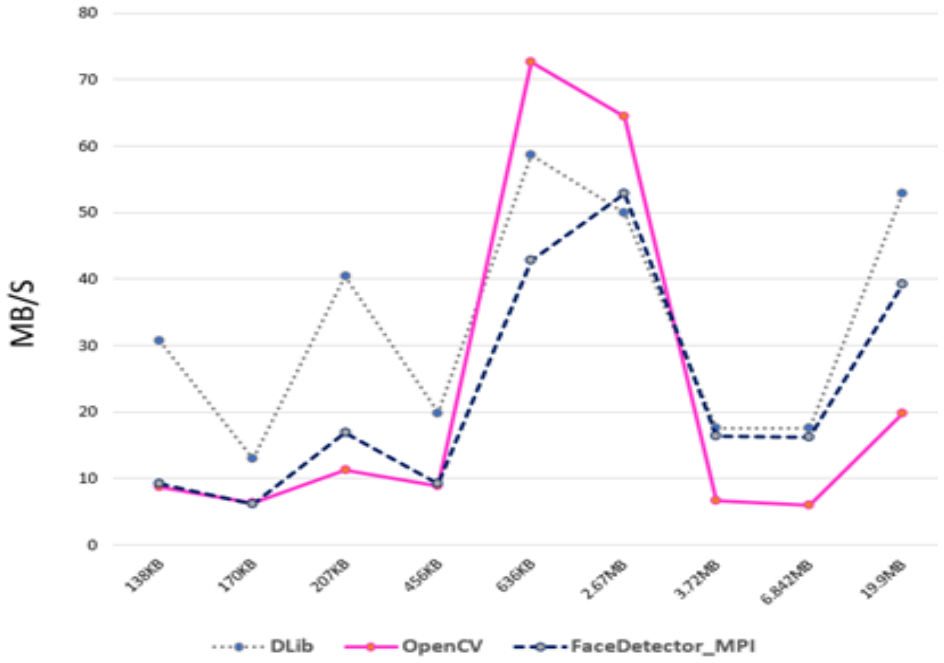


Fig. 8. Throughtput

being similar in Dlib although the latter was optimized at a level 3 (-O3).

Better still is the fact of being able to manage an image with different clients, because each one of them can access a part of the image managing to increase the parallelism even more and consequently a reduction in access times.

3.2 Face Detector Algorithm Performance

Table (2) shows the average execution times (5 per image) obtained in the face detection algorithms that comes by default in Dlib and OpenCV, as well as the results of the modified algorithm in Facedetector_MPI for 2 and 4 clients.

ISSN 2007-9737

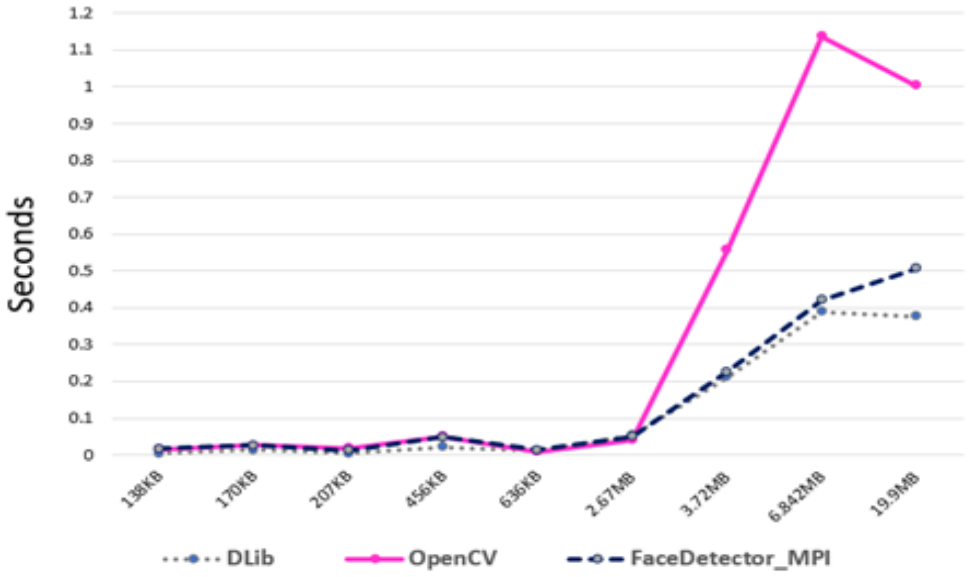


Fig. 9. Latency

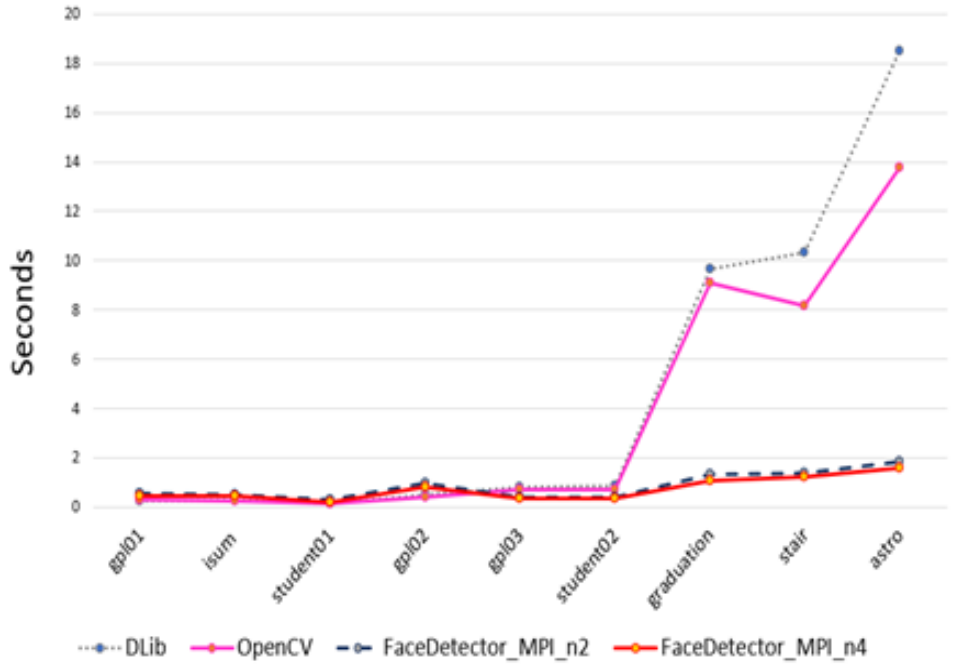


Fig. 10. Run time of face detector algorithm

Table 2. Run time

IMAGE	GEOMETRY (PIXELS)	DLIB	OPENCV	FACEDETECTOR _MPI_NP2	FACEDETECTOR _MPI_NP4
gpl01	960x651	247.2	271.67	572.33	480.30
isum	1008x756	292.2	259.29	528.55	464.24
student01	1024x369	149	162.28	289.39	222.03
gpl03	1280x1033	507.2	413.21	955.15	838.196
gpl02	1280x1652	800	688.58	412.338	354.466
student02	2048x1152	885.6	704.19	396.44	327.31
graduation	8204x3132	9653.8	9078.03	1344	1063.79
stair	5963x4608	10319.2	8169.36	1380.97	1209.41
astro	8048x6070	18483.2	13752.8	1833.57	1608.74

Table 3. Number faces detect

		Dlib			OpenCV			Facedetector mpi_n2			Facedetector mpi_n4		
	Total Faces	FD	FP	FN	FD	FP	FN	FD	FP	FN	FD	FP	FN
gpl01	4	2	0	2	3	0	1	3	0	1	4	0	0
gpl02	12	7	0	5	10	1	3	7	0	5	8	0	4
gpl03	15	2	0	13	6	0	9	13	1	3	11	0	4
student01	17	17	0	0	17	0	0	16	0	1	11	0	6
student02	17	17	0	0	17	0	0	16	0	1	15	0	2
astro	17	17	0	0	21	4	0	17	0	0	17	0	0
stair	18	18	0	0	26	8	0	18	0	0	14	1	5
isum	62	10	0	52	23	0	39	60	0	2	60	0	2
graduation.	148	149	1	0	163	16	1	149	2	1	125	0	23

When analyzing the results, we realize that there are some differences in the execution times of the algorithm. Figure 10 shows how the operations for smaller images in some cases are similar for the 3 implementations.

However, as the image grows, it is possible to appreciate how the execution time is considerably reduced in Facedetector_MPI.

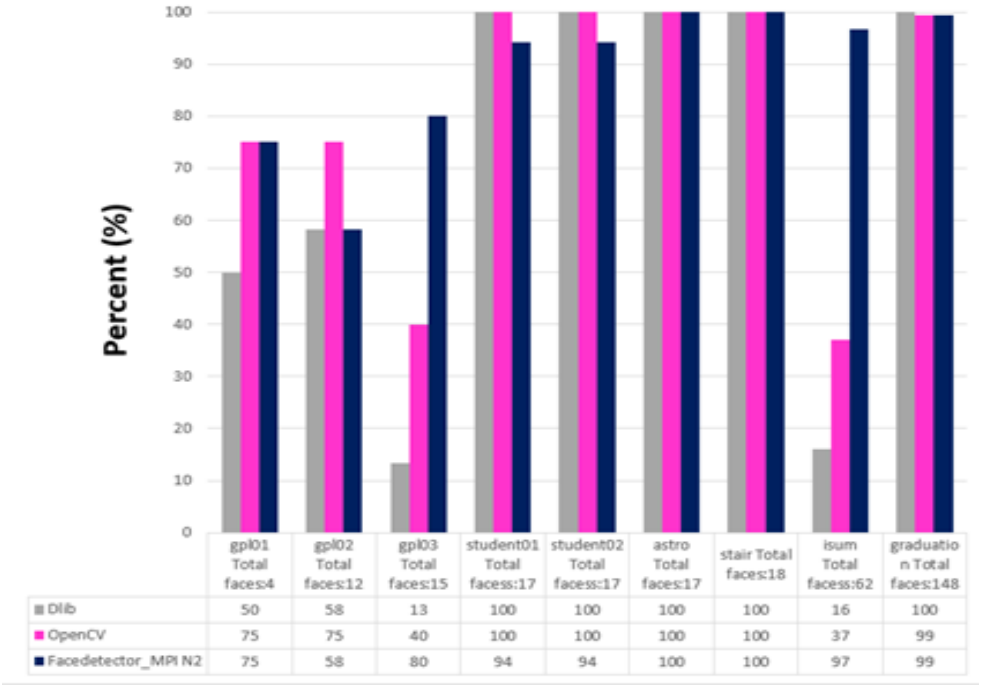
This increase in performance is achieved thanks to the interaction of multiple processes that execute in parallel the algorithm of face detection on the assigned piece of image. You might think that this increase in benefits compromises the success rate of the face detection algorithm, however this does not happen, and it is here that the implementation of scaling becomes relevant.

3.3 Hit Rate

We take some final measurements and add a comparison between Dlib, OpenCV and Facedetector_MPI with 2 and 4 clients. Table (3) shows the lowest number of faces in the images (TF) (4 to 128), as well as the results obtained in: total number of faces detected (FD), false positives (FP) and false negatives (FN) of each of the implementations.

If we look at the data, we can realize that even in some cases, the number of faces detected is greater (FP) with respect to TF and this in the case of OpenCV adds more time in the execution of the face detection algorithm.

This is because some characteristics have been found within the image that are interpreted as



	gpl01 Total faces:4	gpl02 Total faces:12	gpl03 Total faces:15	student01 Total facess:17	student02 Total facess:17	astro Total faces:17	stair Total faces:18	isum Total facess:62	graduation Total faces:148
Dlib	50	58	13	100	100	100	100	16	100
OpenCV	75	75	40	100	100	100	100	37	99
Facedetector_MPI N2	75	58	80	94	94	100	100	97	99

Fig. 11. Hit rate

faces, but they are not. To prevent this from happening in our implementation, we take advantage of the previously designed image scaling in order to better identify the characteristics of the faces and thus reduce the number of false detections.

Figure 11 clearly shows how the hit rate is similar in 5 (student01, student02, astro, stair and graduation) of the 9 cases having a hit rate between 94 and 100% over the 3 implementations, However, Facedetctor_MPI it is superior (gpl01, gpl03, isum) about 40 to 60% effective in 3 cases and only one (gpl02) shows a 20% reduction compared to the other implementations.

Although the results show certain differences, we can see how the added parallelism strategy improves performance and exploits existing computing resources, without compromising the effectiveness of face detection on an HPC cluster.

6 Conclusions

The work done in this investigation define several important points. Since the study of different algorithms for face detection, until to find a better solution in the processing of images.

A limitation that we find to image processing is its high consumption in computational resources, leaving low-performance teams without the possibility of being able to efficiently execute this task.

The tests performed have allowed us to see the superiority that exists when using a large image since the results improve significantly thanks to the implementation of a greater number of OSTs, as well as the use of multiple lustre clients and the MPI interface.

Once the parallelization strategy for face detection on HPC cluster is made, we can conclude that the results are similar or in some cases better to the others implementations that do not exploit the parallelism to a higher level.

The reuse existing hardware is presented as a less expensive solution compared to the acquisition of new equipment and the implementation of GPUs.

Finally, we hope that this work helps to promote the use of HPC clusters through the comparison of different parallelization strategies

Acknowledgements

We thank the Programa para el Desarrollo Profesional Docente (PRODEP) for the support granted mentioned in the Official Letter No. 511-6/17/8212, and the Universidad Autónoma de Tamaulipas - Facultad de Medicina e Ingeniería en Sistemas Computacionales de Matamoros, all of them for providing the means to carry out this work.

References

1. **Schroff, F., Kalenichenko, D., & Philbin, J. (2015).** FaceNet: A Unified Embedding for Face Recognition and Clustering. *IEEE Computer Society Conference on Computer Vision and Pattern Recognition,* pp. 815–823.
2. **Taigman, Y., Yang, M., Ranzato, M., & Wolf, L. (2014).** DeepFace: Closing the Gap to Human-Level Performance in Face Verification. *IEEE Conference on Computer Vision and Pattern Recognition,* pp. 1701–1708.
3. **Viola, P. & Jones, M. (2001).** Rapid object detection using a boosted cascade of simple features. *IEEE Computer Society Conference on Computer Vision and Pattern Recognition,* Vol. 1, pp. 1–9. DOI: 10.1109/CVPR.2001.990517.
4. **Rowley, H.A., Baluja, S., & Kanade, T. (1998).** Neural network-based face detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence,* Vol. 20, No. 1, pp. 23–38. DOI: 10.1109/34.655647.
5. **King, D.E. (2009).** Dlib-ml: A machine learning toolkit. *The Journal of Machine Learning Research,* Vol. 10, pp. 1755–1758.
6. **Yu, S., Jia, S., & Xu, C. (2017).** Convolutional neural networks for hyperspectral image classification. *Neurocomputing,* Vol. 219, pp. 88–98.
7. **Chang, C.H., Lu, C.W., Yang, C.T., & Chang, T.C. (2016).** An approach of performance comparisons with OpenMP and CUDA parallel programming on multicore systems: OpenMP and CUDA performance comparisons on multicore systems. *Concurrency and Computation: Practice and Experience*, Vol. 28, No. 16, pp. 4230–4245. DOI: 10.1002/cpe.3829.
8. **Iordan, C., Jahre, M., & Natvig, L. (2015).** Tuning the victim selection policy of Intel TBB. *Journal of Systems Architecture,* Vol. 61, No. 10, pp. 58–591. DOI: 10.1016/j.sysarc.2015.07.004.
9. **Dinan, J., Balaji, P., Buntinas, D., Goodell, D., Gropp, W., & Thakur, R. (2016).** An implementation and evaluation of the MPI 3.0 one-sided communication interface. *Concurrency and Computation: Practice and Experience,* Vol. 28, No. 17, pp. 4385–4404. DOI: 10.1002/cpe.3758.
10. **Wang, L., Ma, Y., Zomaya, A.Y., Ranjan, R., & Chen, D. (2015).** A parallel file system with application-aware data layout policies for massive remote sensing image processing in digital earth. *IEEE Transactions on Parallel and Distributed Systems,* Vol. 26, No. 6, pp. 1497–1508. DOI: 10.1109/TPDS.2014.2322362.
11. **Moore, M., Farrell, P., & Cernohous, B. (2018).** Lustre lockahead: Early experience and performance using optimized locking. *Concurrency and Computation: Practice and Experience,* Vol. 30 No. 1, pp. 1–14. DOI: 10.1002/cpe.4332.
12. **Pulli, K., Baksheev, A., Kornyakov, K., & Eruhimov, V. (2012).** Realtime computer vision with OpenCV. *Communications of the ACM,* Vol. 55, No. 6, pp. 61–69. DOI: 10.1145/2184319.2184337.
13. **Crow, F.C. (1984).** Summed-Area tables for texture mapping. *ACM SIGGRAPH Computer Graphics,* Vol. 18, No. 3, pp. 207–212. DOI: 10.1145/800031.808600.

Article received on 01/11/2018; accepted on 02/10/2019.
Corresponding author is Hugo Eduardo Camacho Cruz.