Optimizing Face Recognition and Recovery Algorithms through Parallelization

CSCI-4320 Parallel Programming Project

Sharat Val
Computer Science Dept.
Rensselaer Polytechnic Institute
Troy NY USA
vals@rpi.edu

Danny Zheng
Computer Science Dept.
Rensselaer Polytechnic Institute
Troy NY USA
zhengd3@rpi.edu

ABSTRACT

This report presents a comprehensive benchmarking study evaluating the performance impacts of utilizing CUDA, MPI (Message Passing Interface), and MPI I/O (Input/Output) on parallelized algorithms in the context of high-performance computing (HPC) environments. The study focuses on assessing the efficiency of parallelized algorithms when executed on GPUs (Graphics Processing Units) using CUDA and using MPI for communication and coordination. Additionally, the utilization of MPI I/O for optimized data input and output operations is analyzed.

This study centers on the implementation and evaluation of parallelized facial recognition and recovery algorithms within the AiMOS Supercomputer. Our investigation delves into the impact of utilizing MPI-based parallelization in conjunction with CUDA-enabled GPU acceleration as opposed to conventional CPU-based implementations. This aims to enhance parallelization and expedite the program's performance beyond what is achieved with CPU alone.

The benchmarking experiments are conducted using synthetic workloads to assess the practical implications of adopting these technologies. These synthetic workloads are processed, and the results of the recognition and recovery algorithms are outputted to a file for viewing with the use of MPI I/O. Performance metrics such as execution time, speedup, and overall cycles are measured and analyzed to provide insights into the strengths and limitations of each approach.

The findings highlight the significant performance benefits of leveraging CUDA for GPU computing and MPI for distributed memory parallelism in HPC applications. Moreover, the use of MPI I/O demonstrates improvements in data throughput and I/O performance for large-scale simulations. These results contribute to the understanding of best practices in deploying parallel algorithms on modern HPC architectures, facilitating both accelerated execution and streamlined testing for these algorithms.

1 Introduction

In today's digital age, face recognition technology has emerged as a critical with applications such as security and surveillance. One of the key challenges in face recognition is ensuring accurate identification under varying conditions such as changes in lighting, facial expressions, poses, and occlusions.

Moreover, in important applications like law enforcement and security, the ability to recover identity information from partial or degraded facial images is of utmost importance. This necessitates the development of robust face recovery algorithms capable of reconstructing facial features from incomplete or corrupted data [4].

Additionally, the speed and efficiency of face recognition systems are paramount, especially in applications where real-time processing is essential. Rapid identification of individuals in surveillance videos, access control systems, and crowd monitoring applications requires fast and efficient face recognition algorithms capable of handling large volumes of data in real time. Achieving high throughput while maintaining accuracy is a challenging task that necessitates optimization of computational resources and algorithmic efficiency.

The core of the research involves the implementation and testing of parallelized facial recognition and recovery algorithms. We compare the computational speedup achieved with MPI-based parallelism against traditional CPU-based implementations. The addition of CUDA-enabled GPU acceleration to the MPI program is also explored to further enhance parallelization and execution speed. By using large synthetic workloads, we aim to display the performance improvements of utilizing CUDA and MPI to these algorithms. In this paper we only use the nearest

neighbor algorithm [9] for face recognition and linear interpolation [8] for face recovery, but we would also like to note the importance of a parallelized testing suite for any algorithm being used. This is done through the task of reporting the testing results in parallel with MPI I/O.

2 Code & Algorithms

In this section we will discuss the sequential implementation of the face recognition and recovery algorithms, as well as how the algorithms are tested. Then we will explain how we parallelized sections of the code using MPI, CUDA, and MPI I/O.

2.1 Code & Algorithms: Sequential

First, the program reads all the rows in the input csv file. The csv file is ordered such that there are 4097 numbers separated by commas where the first number of each row indicates the label (which person the image is of) for that image and the rest of the 4096 numbers represent pixel values from 0-255. Therefore, each row represents the 64x64 pixel information for each image. In the original data set, there are images of 40 different people with 10 different photos of each of these 40 people, amounting to 400 total images. In order for training and testing, we take 40 distinct images, each of a different person, and have 360 distinct images for training and 40 distinct images for testing. One thing to note is that these images are in grayscale. Here is an example of what the images look like without any processing:

Figure 1: Face Images without any pre-processing

The first step in processing the data is to normalize each image such that the Euclidean Distance of each image is equal to 1. The reason we do this step is because normalization has been proven to aid in the classifying power of the nearest neighbor algorithm, which is what is used to classify our images [1]. After the normalization is done, we mean center each column amongst the images. Essentially using all our training data, we find the mean value of each pixel location (4096 in total) and subtract each images pixel value by that corresponding mean. This achieves having a mean of 0 across the columns of all training images used. This step is important as mean centering can lead to more accurate distance metrics. In nearest neighbor algorithms, the distance between data points is typically used as a measure of similarity. By centering the data, you ensure that the distance metric is not influenced by differences in means across features, resulting in more accurate similarity measurements [2]. Here is what the images look like after pre-processing:

Figure 2: Face Images after pre-processing (Image normalization and feature mean centering)

After reading and processing the input of the training images file, we apply the same methods to the testing data except instead of calculating the column means of the testing data for mean centering, we just mean center by using the column means gathered from the training data. By using the column means of the training data on the testing data, you ensure that the pre-processing steps applied to the testing data are consistent with those applied to the training data. This consistency helps prevent introducing bias or discrepancies between the datasets [3].

The next step is then to loop through each of the testing images and pass it through the nearest neighbor algorithm. The algorithm works by comparing a given test image with all the training images by calculating the Euclidean distance between each training image and the test image to find the smallest distance. The training image that produces the smallest distance to the testing image will be the matching image. The algorithm should work in a way such that the training image picked should have the same label as the testing image. Additionally, results are printed out into an output file in the format: “Train-Image: 1, Test-Image: 0, Predicted-Label: 1, Correct” or “Train-Image: 161, Test-Image: 22, Predicted-Label: 18, Correct-Label: 23, Wrong”. This is done to get a deeper understanding on the performance results of the nearest neighbor algorithm. Across all our runs we typically get a success rate of 95%. Here are the training images that got matched to the testing images of Figure 2:

Figure 3: Matched Training Images of Figure 2

Lastly, we test out the efficacy of the linear interpolation occlusion recovery algorithm. We first go through each testing image and randomly remove around 50% of the pixels in an image. Here is what the occluded images of Figure 2 look like:

Figure 4: Occluded Images of Figure 2

We then recover each occluded image by using the linear interpolation algorithm. For each missing pixel, this algorithm finds the closest non-missing pixel to the left, right, up, and down of its location, and then the missing pixel gains the value of the average of the non-missing pixel values around it. Here is what the occlusion recovered images of Figure 4 look like:

Figure 5: Occlusion Recovered Images of Figure 4

We then pass these occlusion-recovered images into the nearest neighbor algorithm and see whether we get a match of the same person. As before, we have a separate output file that records more detailed results about which training/testing images matched correctly and incorrectly. Finally, across our several runs of the sequential version we have seen our occlusion recovery matching get a success rate of around 93%.

2.2 Code & Algorithms: Parallelized (CUDA, MPI, MPI I/O)

In this subsection we will discuss the changes made to the Sequential version in order to parallelize the program. With the use of MPI, we get each rank to read an equal portion of the input file and have each rank only keep track of their own section of training images. When determining the number of ranks we used multiples of 360 because each of our testcases will have a multiple of 360 number of lines for both the training and testing dataset. We duplicated training/testing images to generate larger datasets. This is done solely for measuring the performance of the program and will not affect the actual success rates of the algorithms. Another important point to note is that since MPI I/O cannot read line by line, but byte by byte, we had to modify the input files to each be three digits long in order to have each line be the same number of characters. This allows us to properly use offsets for each rank to navigate to their section of the input file and start reading in the information. We read in using MPI_File_seek (to get each rank to go to their correct section) and then MPI_File_read.

For the normalizing portion we use CUDA in order to get each thread to normalize 1 row for all the rows that are in 1 rank. For the column means we get each rank to calculate their sum for a given column, then utilize MPI_Allreduce to add up the sums for all other ranks and then divide that total sum by the total number of rows found in the input file. This way each rank is able to find the column mean of the whole training dataset, despite only reading a portion of the dataset. After each rank has access to the means of each of the 4096 columns, we utilize CUDA again in order for each thread to access a given element per rank and subtract the corresponding mean from that value.

This process is repeated for the testing dataset, like the sequential version. However, while each rank only keeps track of their portion of the training images, each rank will keep track of all the testing images. We still utilize MPI I/O's offsets to get each rank to read only their portion of the testing data, but then utilize the MPI_Allgather function to combine these portions into 1 big array so that each rank has access to every testing image.

Like the sequential version, we run the nearest neighbor algorithm on each testing image, but this time each rank only has a couple of training images. In order to choose the

correct image for the case where the smallest distance for 1 rank is bigger than the smallest distance for another rank, we utilize MPI_Allreduce once more in order to find the global minimum distance. We then set the expected index to -1 for the ranks where their local min distance is greater than the global min distance, since this means that there is another rank that has a better match than the one found by this rank. We tried utilizing CUDA for calculating the Euclidean distance, however we noticed slower run times as each thread would have to work on a whole row. Additionally, if we made each thread work on an element, we would have had to utilize shared memory to keep track of the sum. However, by doing this we would have needed to implement mutexes for the critical section which would have ultimately made the main part of the code sequential. It is because of these reasons we did not utilize CUDA for calculating the Euclidean distance.

We utilize MPI I/O for the results printed in the output file for the nearest neighbor algorithm. Like the input, each rank will go to their specific offset and print out their results for their training images. We have three types of output: “Train-Image: 1, Test-Image: 0, Predicted-Label: 1, Correct”, “Train-Image: 161, Test-Image: 22, Predicted-Label: 18, Correct-Label: 23, Wrong” or “Train-Image: -1, Test-Image: 22, Found Better”. We do this by using MPI_File_write_at. We also utilize MPI_Reduce to calculate the total number of correctly classified images amongst all the ranks. The success rate for these stays at the same 95%.

For occluding the images, we utilize CUDA and MPI. We make each rank work on a portion of the testing images array, and then utilize CUDA in order for each thread to work on the element level to randomly remove the pixel value or not. We utilize curand_init for the random number, so the randomly generated numbers will vary based on how many ranks are used. Just like the sequential version about half the pixel values for an image will be removed randomly. After, each rank occludes their portion of testing images, they then do occlusion recovery on those same portions of testing images. For the occlusion recovery we utilize CUDA on the element level for each thread with a missing pixel value to find the 4 closest non-missing pixel values in the image and assign the average to itself. After, this is done we utilize MPI_Allgather just like last time for each rank to have an array of all the occlusion recovered images. This is followed up by running the nearest neighbor algorithm, printing out the results to a separate output file using MPI I/O, and printing out the success rate to standard output. For the parallel version since the random numbers vary based on how many ranks used, we have seen the success rate for occlusion recovery vary from 89% to 94%.

3 Results

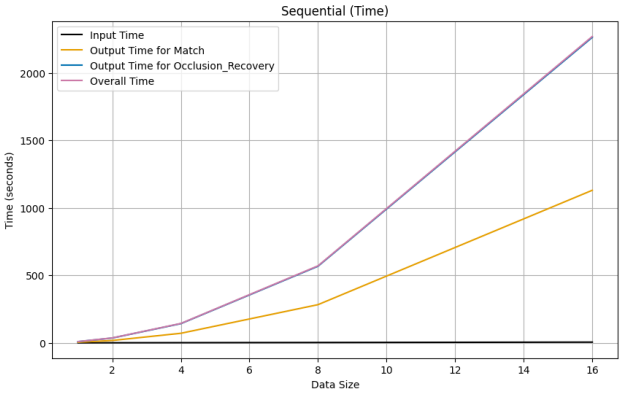
3.1 Data Collection

We conducted a series of comprehensive investigations to evaluate the performance in different scenarios. These investigations included the Sequential Study, Strong Scale Study with MPI Only, Strong Scale Study with MPI and CUDA, and Weak Scale Study with MPI and CUDA. To assess performance metrics accurately, we employed clock cycle measurements and execution time analyses across program operations. Clock cycle calculations were made feasible by leveraging the Power9 processors of the AiMOS Supercomputer, our HPC setting.

Our analysis encompassed key program operations such as reading training and testing data, writing nearest neighbor results to an output file (including algorithm runtime), and complex tasks like occluding images, recovering them, executing nearest neighbor, and outputting results to another file. Furthermore, we calculated comprehensive clock cycles and execution times, factoring in overhead elements such as data normalization and mean centering. Using the collected data, we developed graphical representations to visualize overhead across the four test cases. Additionally, we calculated relative speedup values for each test case relative to their base configurations of Rank/Data Size 1.

3.2 Performance Results

To evaluate the performance improvement achieved by implementing MPI and CUDA, it was essential to establish the baseline results using the sequential version of the algorithm. We conducted experiments using data sizes comprising 1, 2, 4, 8, and 16 data sets, with each data set containing 360 images, to establish this baseline (Figure 6). For the Strong Scale Study using MPI exclusively, we employed a data size containing 16 data sets and systematically increased the number of MPI ranks utilized by the algorithm from 1 to 36 (Figure 7). In the Strong Scale Study integrating both MPI and CUDA, we maintained a data size of 16 data sets and concurrently varied the number of MPI ranks and block size on a single GPU. This involved scaling from 1 rank with a block size of 1 thread to 36 ranks with a block size of 1024 threads (Figure 8). For the Weak Scale Study incorporating MPI and CUDA, we utilized a fixed block size of 1024 threads and expanded both the number of GPUs/ranks and the data size (Figure 9).



Data Size	Speed Up
1	1.000000
2	0.256787
4	0.065363
8	0.016479
16	0.004143

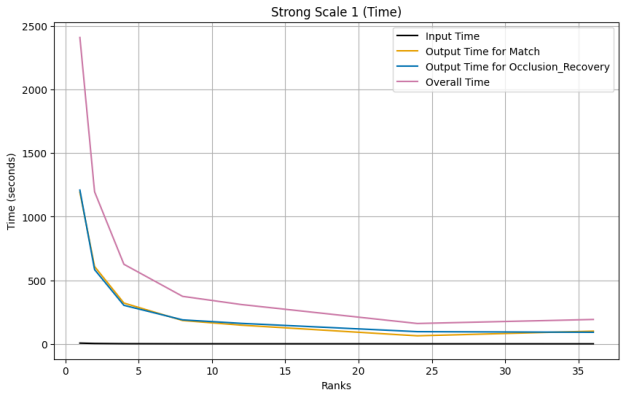
Data Size	Input Time	Output time Match	Output time Occlusion	Overall Time
1	0.347571	4.434848	9.021585	9.408033
2	0.701058	17.756024	35.858856	36.637524
4	1.383284	71.011663	142.373176	143.936009
8	2.769965	283.144698	567.784130	570.921170
16	5.556700	1130.824359	2264.301142	2270.743474

Data Size	Input Cycle	Output cycle Match	Output Cycle Occlusion	Overall Cycle
1	216817130	2271788925	4621358929	4858104177
2	440780164	9091660547	18360692487	18841231875
4	769962888	36362066315	72907450428	73769366083
8	1479302210	144983571042	290731951575	292399222325
16	2880609463	578985446896	1159329308313	1162663387450

A) Graph of overall time in seconds

B) Speed up, Time, and Cycles Results

Figure 6: Sequential Study Results



Ranks	Speed Up
1	1.000000
2	2.014011
4	3.849115
8	6.446078
12	7.785924
24	15.027790
36	12.553565

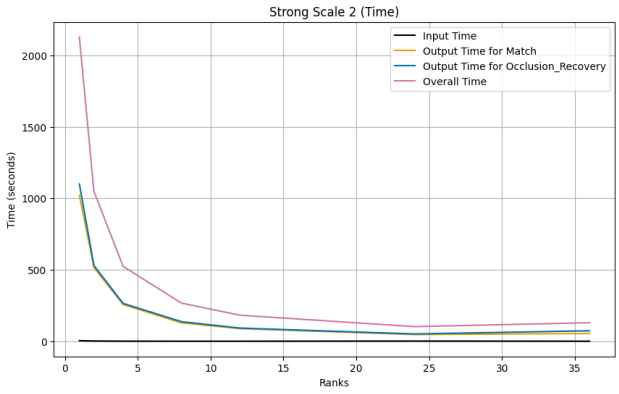
Ranks	Input Time	Output time Match	Output time Occlusion	Overall Time
1	6.177315	1193.032451	1209.441405	2409.715509
2	3.338429	608.348952	584.207091	1196.475654
4	1.508627	321.089132	303.186765	626.044008
8	0.858442	183.812734	188.988526	373.826598
12	0.770899	147.206449	161.340415	309.496411
24	0.631973	63.339155	96.184359	160.350623
36	0.477137	100.069423	91.216050	191.954672

Ranks	Input Cycle	Output cycle Match	Output Cycle Occlusion	Overall Cycle
1	3162875402	610856072406	619257758881	1233821708920
2	1709326178	311486529329	299125372780	612618829927
4	772429693	164403905006	155237517027	320546739467
8	439524908	94115710527	96765799116	191406513884
12	394697366	75372575480	82609427356	158468201341
24	323568343	32430883388	49248264458	82102654627
36	244281890	51237672262	46704547588	98284879821

A) Graph overall time in seconds

B) Speed up, Time, and Cycles Results

Figure 7: Strong Scale Study (SSS) with MPI only Results



Ranks	Speed Up
1	1.000000
2	2.029941
4	4.058677
8	7.950822
12	11.633693
24	20.694740
36	16.344770

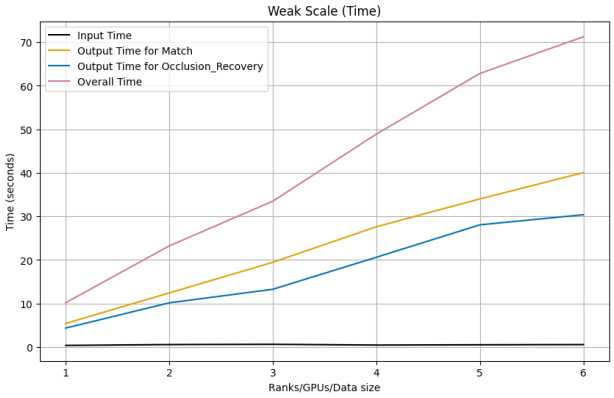
Ranks	Input Time	Output time Match	Output time Occlusion	Overall Time
1	4.840002	1020.651005	1100.464469	2126.536528
2	2.423882	515.267023	529.545661	1047.585273
4	1.207609	257.286887	265.117990	523.948250
8	0.732944	128.819828	137.518203	267.461203
12	0.781197	88.808308	92.713739	182.791181
24	1.950585	47.486703	51.600696	102.757343
36	0.704368	54.876166	73.803348	130.105017

Ranks	Input Cycle	Output cycle Match	Output Cycle Occlusion	Overall Cycle
1	2478182586	522598370024	563464840028	1088838932596
2	1241066931	263829368132	271140370005	536389382366
4	618308295	131737197745	135746908661	268274362515
8	375265765	65958255038	70411957962	136945310517
12	399967464	45471587159	47471234888	93592654003
24	998725150	24314118815	26420558514	52613770179
36	360620478	28097733914	37788838087	66616472441

A) Graph overall time in seconds

B) Speed up, Time, and Cycles Results

Figure 8: Strong Scale Study (SSS) with MPI and CUDA Results



A) Graph overall time in seconds

Ranks/GPUs/Data size	Speed Up
1	1.000000
2	0.437788
3	0.303634
4	0.207973
5	0.161856
6	0.142704

Ranks/GPUs/Data size	Input Time	Output time Match	Output time Occlusion	Overall Time
1	0.366870	5.411954	4.348610	10.169917
2	0.559949	12.422389	10.147059	23.230253
3	0.619696	19.458941	13.261161	33.494015
4	0.447872	27.623775	20.612566	48.900103
5	0.510282	34.017305	28.067392	62.833061
6	0.556370	40.062894	30.384583	71.265595

Ranks/GPUs/Data size	Input Cycle	Output cycle Match	Output Cycle Occlusion	Overall Cycle
1	187831156	2771025880	2226563933	5207199426
2	286684049	6360518914	5195492543	11894372621
3	317277037	9963379471	6789982124	17149631547
4	229301736	14143872516	10553999552	25037741792
5	261257810	17417510407	14371032424	32171731093
6	284858259	20513027005	15557525860	36489457311

B) Speed up, Time, and Cycles Results

Figure 9: Weak Scale Study (WSS) with MPI and CUDA Results

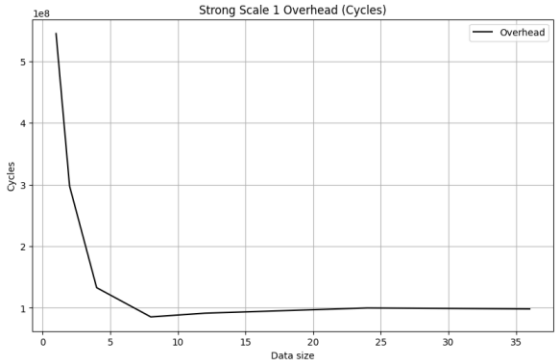
3.3 Result Analysis

In the sequential study, we observed an exponential increase in the overall runtime as the dataset size grew larger. This trend signifies the escalating computational demands encountered in processing larger datasets using a sequential approach.

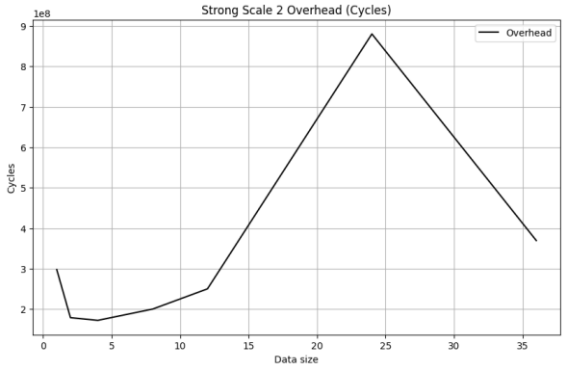
Transitioning to the Strong Scale Study (SSS) with MPI, we analyzed the impact of increasing the number of MPI ranks on the total runtime and computational cycles required for task completion. Notably, we observed an exponential decay in both runtime and cycles with each increment in the number of ranks, with the most significant improvement seen when transitioning from 1 rank to 2 ranks. The runtime reached its minimum at 24 ranks, beyond which further increases led to an increase in overall time. The speedup was 15.027 times its base. This rise can be attributed to the increasing overhead time outweighing the time saved by utilizing more ranks (Figure 10 A).

In the SSS case combining MPI and CUDA, we observed a modest improvement in efficiency compared to using MPI alone. The overall runtime and cycles showed a similar trend to the previous study, exhibiting a relative speedup ranging from 13.3% to a maximum of 69.3%. This improvement aligns with the increased efficiency resulting from a larger block size, allowing the GPU to process the dataset more rapidly. Notably, a block size of 128 threads yielded the highest speedup at 69.3% relative to the SSS case with only MPI, with further increases in block size leading to diminished speedup. It had a speedup of 20.695 compared to the 1 rank test. It's crucial to highlight that while the overhead time increases in the SSS with MPI and CUDA due to larger block sizes, the overhead time decreases in the SSS using only MPI (Figure 10 B).

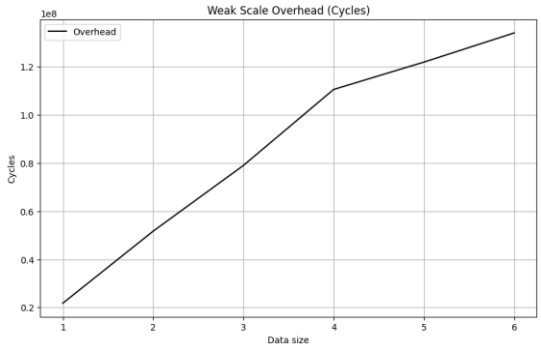
In the Weak Scale Study with MPI and CUDA, we observed a predominantly linear increase in overall runtime, with its speedup diminishing. This linear trend aligns with expectations, as the simultaneous increase in dataset size, MPI ranks, and GPUs introduces escalating program overhead, consequently increasing the overall runtime (Figure 10 C).



A) SSS w/ MPI Overhead cycles (scaled to 1e8 cycles)



B) SSS w/ MPI & CUDA Overhead cycles (scaled to 1e8)



C) WSS w/ MPI & CUDA Overhead cycles (scaled to 1e8)

Figure 10: Overhead Cycles Graph Results (Overhead Time Graph Results Follow Same Trend)

4 Summary and Future Work

Overall, it is clear that parallelizing both the facial recognition/occlusion recovery algorithms and testing of these algorithms yields faster execution times and fewer clock cycles. Through the use of MPI, to assign each process a portion of the images, we are able to quickly apply the preprocessing steps of normalization and mean centering with the help of message passing. We can then run the nearest neighbor algorithm on the test images for each process to find their local best solution and then find a global solution through the use of message passing. Additionally, with the help of CUDA, we utilize the GPU and use threads speed up the linear interpolation algorithm for each pixel to rapidly find its closest non-missing pixel neighbors for occlusion recovery.

Although we have shown the improvements gained by parallelization, there are still further improvements that can be made upon this program. As mentioned before, the norm2 calculation used for the nearest neighbor algorithm does not utilize CUDA because we noticed a decrease in execution times when we utilized it. We stated that this happened because we make each thread process a whole image (of the images that 1 process holds) instead of making each thread process a pixel, and summing up the squared differences in a shared memory variable. With modifying shared memory comes mutexes which can slow down parallelization efforts in order to have correctness. We would have liked to experiment with mcs locks instead just mutexes, to see if any overall improvement can be gained [5]. Additionally, we would have liked to look into more creative ways of addressing are with mutual exclusion through the use of algorithms [6].

Since we also wanted this program to work as a testing suite we would have liked to implement and test more facial recognition algorithms. One such classifier we would have liked to try is support vector machines or a simple convolutional neural network layer for occlusion recovery [7]. In the future, we would like to implement a more general testing suite, one that is able to be more flexible in the type of image it can process, rather than just being able to test grayscale 64x64 pixel images. We also hope to work with more than just 400 unique images, and not have to rely on duplicating images for the purpose of testing run times. Moreover, we would like to have a more diverse group of faces to work with. We need to ensure that the training images we use as our database for the nearest neighbor algorithm, do not favor certain races/genders over others, as this can give way to forms of discrimination [10].

One final note we would like to state in order to hammer in the importance of parallelizing these algorithms comes from the scientific journal entitled: “The visual human face super-resolution reconstruction algorithm based on improved deep residual network” [4]. The research paper focuses on advancing the field of face super-resolution (SR) by improving the architectural nuances of deep residual networks. The paper effectively discusses the implications of the architectural changes on the network's performance. The use of a global-local residual learning approach also contributes significantly to the model's robustness against overfitting and aids in better gradient flow during training, which in turn speeds up the optimization process. While the paper presents significant advancements, there are areas that could be improved or further explored, such as utilizing the power of HPC to work with more varied and larger datasets. The improvements suggested by the authors not only enhance the quality of super-resolved images but also offer a pathway towards faster training times without sacrificing output fidelity. Ultimately, this showcases the need for parallel computing and HPC to speed up these face recognition/recovery techniques.

We hope that this paper can serve as a basis for creating parallelized algorithms and testing suites to rapidly find which facial recognition/occlusion recovery algorithms perform the best.

ACKNOWLEDGMENTS

We thank Prof. Christopher Carothers and the Center for Computational Innovations (CCI) for making it possible for us to use AiMOS to perform the experiments described in this paper.

Sharat Val – Wrote code (Sequential version, MPI version, and MPI/CUDA version), Tested code, Wrote Sections 1,2,4 of the report.

Danny Zheng – Created datasets, Tested code, Collected data for runtime and clock cycles, Wrote results section and abstract for the report.

REFERENCES

[1] Dalwinder Singh and Birmohan Singh. 2020. Investigating the impact of data normalization on classification performance. *Applied Soft Computing* 97 (December 2020). DOI:http://dx.doi.org/10.1016/j.asoc.2019.105524

[2] Xueyi Wang. 2011. A fast exact K-nearest neighbors algorithm for high dimensional search using K-means clustering and triangle inequality. *The 2011 International Joint Conference on Neural Networks* (July 2011). DOI:http://dx.doi.org/10.1109/ijcnn.2011.6033373

[3] Qiong Wei and Roland L. Dunbrack. 2013. The role of balanced training and testing data sets for binary classifiers in bioinformatics. *PLoS ONE* 8, 7 (July 2013). DOI:http://dx.doi.org/10.1371/journal.pone.0067863

[4] Di Fan, Shuai Fang, Guangcai Wang, Shang Gao, and Xiaoxin Liu. 2019. The visual human face super-resolution reconstruction algorithm based on improved deep residual network. *EURASIP Journal on Advances in Signal Processing* 2019, 1 (July 2019). DOI:http://dx.doi.org/10.1186/s13634-019-0626-4

[5] John M. Mellor-Crummey and Michael L. Scott. 1991. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems* 9, 1 (February 1991), 21–65. DOI:http://dx.doi.org/10.1145/103727.103729

[6] Leslie Lamport. 1987. A fast mutual exclusion algorithm. *ACM Transactions on Computer Systems* 5, 1 (January 1987), 1–11. DOI:http://dx.doi.org/10.1145/7351.7352

[7] Wang Xiang. 2022. The research and analysis of different face recognition algorithms. *Journal of Physics: Conference Series* 2386, 1 (December 2022). DOI:http://dx.doi.org/10.1088/1742-6596/2386/1/012036

[8] S.M. Seitz and C.R. Dyer. 1995. Physically-valid view synthesis by image interpolation. *Proceedings IEEE Workshop on Representation of Visual Scenes (In Conjunction with ICCV'95)* (June 1995). DOI:http://dx.doi.org/10.1109/wvrs.1995.476848

[9] Xinyu Guo. 2021. A KNN classifier for face recognition. *2021 International Conference on Communications, Information System and Computer Engineering (CISCE)* (May 2021). DOI:http://dx.doi.org/10.1109/cisce52179.2021.9445908

[10] Fabio Bacchini and Ludovica Lorusso. 2019. Race, again: How face recognition technology reinforces racial discrimination. *Journal of Information, Communication and Ethics in Society* 17, 3 (August 2019), 321–335. DOI:http://dx.doi.org/10.1108/jices-05-2018-0050