Computer Engineering &
Software Systems

Ain Shams University
Faculty of Engineering

Parallel Histogram Equalization of Gray Scale Images

Under Supervision of

Dr. Tamer Mostafa

&

Eng. Beshoy

Submitted By:

Ahmed Khaled Saad Ali Mekheimer
ID: 1809799
(Team Leader)

Mohammed Makram Mahrous
ID: 19P2645

Noureldien Khaled
ID: 18P5722

Ashraf Mohamed Almolakab
ID: 12P2018

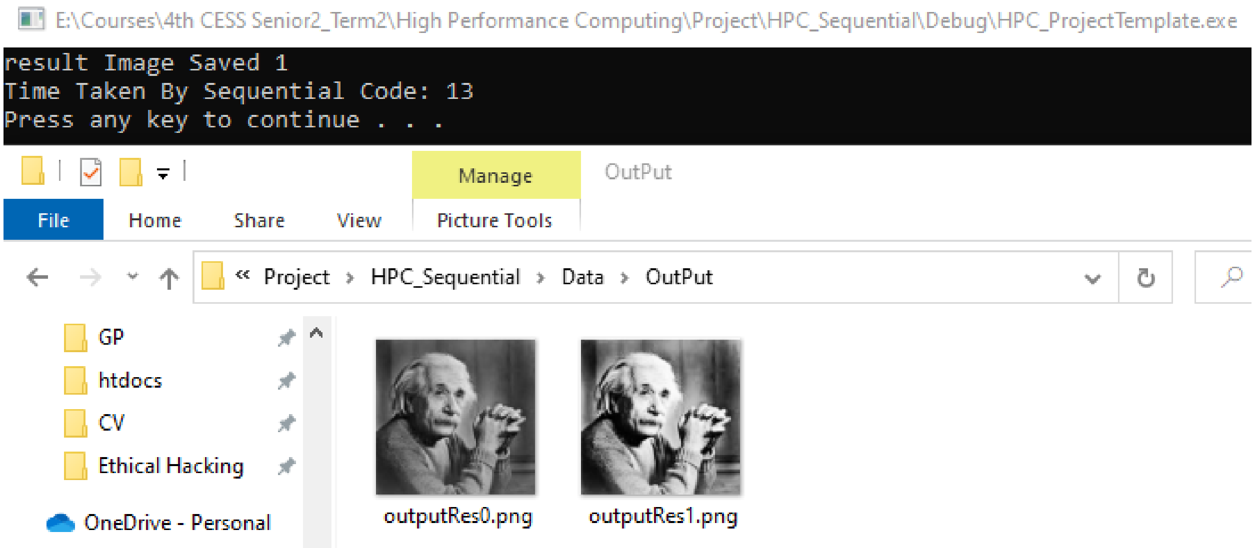
1. Sequential Code

Implementation & Description (In comments)

```
int main()
{
    int ImageWidth = 4, ImageHeight = 4;

    int start_s, stop_s, TotalTime = 0;

    System::String^ imagePath;
    std::string img;
    img = "..//Data//Input//test.png";

    imagePath = marshal_as<System::String^>(img);
    int* imageData = inputImage(&ImageWidth, &ImageHeight, imagePath);

    start_s = clock();

    //Array to count Number of Pixels' Intensities
    int number_of_pixels[256];
    for (int i = 0; i < 256; i++) {
        number_of_pixels[i] = 0;
    }

    //Array for probability of each pixel intensity
    double probability[256];

    //Array for Cumulative Values of Probabilities
    double cumulative_probability[256];

    //Array to Scale "cumulative_probability" to 0-255
    double scaled_cumulative_probability[256];

    //Array to Floor "scaled_cumulative_probability" values
    int floored_round[256];

    //                                                              STEP 1
    //Count number of pixels associated with each pixel intensity
    for (int i = 0; i < (ImageWidth * ImageHeight); i++)
        number_of_pixels[imageData[i]]++;
```

```
    //                                                              STEP 2
    //Calculate Probability of each pixel intensity in the image matrix
    for (int i = 0; i < 256; i++)
        probability[i] = (double)((double)number_of_pixels[i] /
(double)(ImageWidth * ImageHeight));

    //                                                              STEP 3
    //Calculate Cumulative Probability
    cumulative_probability[0] = probability[0];
    for (int i = 1; i < 256; i++)
        cumulative_probability[i] = probability[i] + cumulative_probability[i -
1];

    //                                                              STEP 4
    //Change Intensity range to 0-255
    //Scaling to 0-255 & Flooring
    for (int i = 0; i < 256; i++)
    {
        scaled_cumulative_probability[i] = cumulative_probability[i] * 256;
        floored_round[i] = floor(scaled_cumulative_probability[i]);
    }

    //                                                              STEP 5
    //Mapping the Original Image values to Floor Round values to have the Final
Output Image
    //Here we will prepare "imageData" values to be mapped by values we have from
"floored_round"
    for (int i = 0; i < ImageWidth * ImageHeight; i++)
        imageData[i] = floored_round[imageData[i]];

    stop_s = clock();
    TotalTime += (stop_s - start_s) / double(CLOCKS_PER_SEC) * 1000;

    //Converting "imageData" values to the Final Output Image using the given
"Create Image" function
    createImage(imageData, ImageWidth, ImageHeight, 1);
    cout << "Time Taken By Sequential Code: " << TotalTime << endl;

    free(imageData);
    system("pause");
    return 0;
}
```

Input & Output



So, Sequential code takes 13 ms.

2. MPI Code

Implementation & Description (In comments)

```
int main()
{
    int ImageWidth = 4, ImageHeight = 4;

    int start_s, stop_s, TotalTime = 0;

    System::String^ imagePath;
    std::string img;
    img = "..//Data//Input//test.png";


    imagePath = marshal_as<System::String^>(img);
    int* imageData = inputImage(&ImageWidth, &ImageHeight, imagePath);
    //Here the clock will include the time of MPI Initialization which adds time
to all processors
    start_s = clock();

    MPI_Init(NULL, NULL);
    //Here the clock starts after MPI Initialization which will lead to a much
fewer time
    //start_s = clock();
    int myrank, mysize;

    MPI_Comm_size(MPI_COMM_WORLD, &mysize);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

    // Local dynamic array from "imageData" for each processor (To be assigned in
Scatter)
    int* local_imageData = new int[(ImageHeight * ImageWidth)/mysize];

    //We are going to scale to 0-255 because maximum value in "imageData" was 255
    //So, we will count number of values of "imageData" which are 0-255 in the
below dynamic array
    //This array is local for each process
    int* local_no_pixels = new int[256];
    for (int i = 0; i < 256; i++) {
        local_no_pixels[i] = 0;
    }

    //This array will contain the sums of all "local_no_pixels" of processros
    int all_no_pixels[256] = { 0 };
```

```
    //This variable is for each processor, it will take its value Scattered from "all_no_pixels"
    int* local_counted_pixels = new int[256];
    for (int i = 0; i < 256; i++) {
        local_counted_pixels[i] = 0;
    }

    //Probability dynamic array for each processor that has probability of its assigned numbers
    double* local_probability = new double[256 / mysize];

    //All probability values will be Gathered in this static array
    double all_probability[256] = { 0 };

    //Cumulative Probability for each value (0-255)
    double all_cumulative_probability[256] = { 0 };

    //Local Cumulative dynamic array for each processor from "all_cumulative_probability"
    //Each processor will Scale its array then Floor it
    double* local_cumulative = new double[256];

    //Local floor values for "local_cumulative" for each processor
    int* local_floor_round = new int[256 / mysize];

    //This array has all the Floors to values (0-255)
    //The array will be used to map the Original Image values to Floor Round values to have the Final Output Image Values in an array
    //This array Gathers all "local_floor" arrays of all processors
    int floor_round[256] = { 0 };

    //                                                      STEP 1
    //Count number of pixels associated with each pixel intensity

    //Here we will Scatter the Original Image's array values on all processors, each has array "local_imageData"
    MPI_Scatter(imageData, (ImageHeight*ImageWidth)/mysize , MPI_INT, local_imageData, (ImageHeight * ImageWidth) / mysize, MPI_INT, 0, MPI_COMM_WORLD);


    //Here each processor will count the number of values (0-255) in its "local_no_pixels" array
```

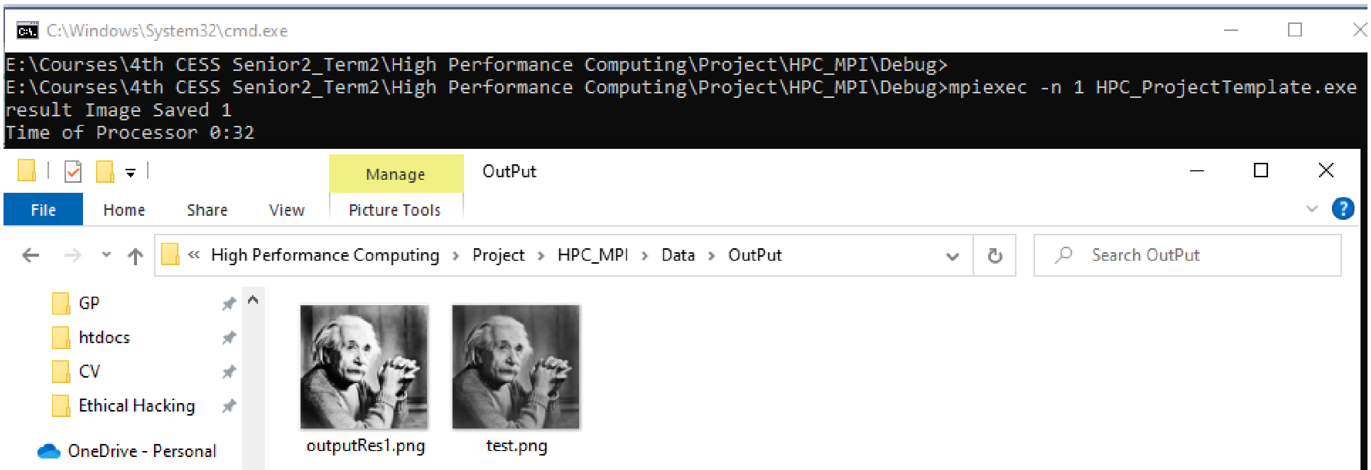
```
    //So, this means that in Index 0 of "local_no_pixels" we have how many
times the number "0" was in "local_imageData" and
    //so on for other numbers until 255
    for (int i = 0; i < (ImageHeight * ImageWidth) / mysize; i++)
        local_no_pixels[local_imageData[i]]++;

    //Now we need to SUM all "local_no_pixels" arrays of processors and reduce
them in one array
    MPI_Reduce(local_no_pixels, &all_no_pixels, 256, MPI_INT, MPI_SUM, 0,
MPI_COMM_WORLD);

    //We will Scatter Number of Pixels array to local arrays for each processor
    //Remember they are all 256 element which is the number we Scale to
    MPI_Scatter(&all_no_pixels, 256 / mysize, MPI_INT, local_counted_pixels, 256
/ mysize, MPI_INT, 0, MPI_COMM_WORLD);

    //                                                      STEP 2
    //Calculate Probability of each pixel intensity in the image matrix

    //Each processor will calculate the probability of its "local_no_pixels" in a
"local_probability" array
    //Probability=Number of pixels/Total Number of Pixels
    //Total Number of Pixels is the Image's number of pixels which is (width X
height)
    for (int i = 0; i < 256 / mysize; i++)
        local_probability[i] = (double)((double)local_counted_pixels[i] /
(double)(ImageWidth * ImageHeight));

    //We will gather all "local_probability" arrays of processor in one array
    MPI_Gather(local_probability, 256 / mysize, MPI_DOUBLE, &all_probability, 256
/ mysize, MPI_DOUBLE, 0, MPI_COMM_WORLD);


    //                                                      STEP 3
    //Calculate Cumulative Probability

    //Since we have all probability values in one array, we will calculate all
their cumulative values in one array also
    //All done by Processor 0
    //Note: Why didn't we do this in parallel and used one processor?
    //Because its faster to calculate Cumulative once by one processor, rather
than all processors calculate the same cumulative value
    if (myrank == 0)
    {
        all_cumulative_probability[0] = all_probability[0];
```

```
        for (int i = 1; i < 256; i++)
            all_cumulative_probability[i] = all_probability[i] +
all_cumulative_probability[i - 1];
    }

    //Scatter values of cumulative on processor, each processor will have the
same number of values in its local array
    MPI_Scatter(&all_cumulative_probability, 256 / mysize, MPI_DOUBLE,
local_cumulative, 256 / mysize, MPI_DOUBLE, 0, MPI_COMM_WORLD);

    //                                                          STEP 4
    //Change Intensity range to 0-255

    //Each processor will Scale the cumulative values by multiplying by 256
    //Each processor will put the value in its "local_probability" array
    //Then each processor will Floor the probability values in its "local_floor"
array
    //Now each processor has "local_floor" array which is the mapping of numbers
(0-255)
    for (int i = 0; i < 256 / mysize; i++)
    {
        local_probability[i] = local_cumulative[i] * 256;
        local_floor_round[i] = floor(local_probability[i]);
    }

    //We will Gather all Floor values from processors to be in "floor_round"
array which will be used to map values from Input Image to Output Image
    MPI_Gather(local_floor_round, 256 / mysize, MPI_INT, floor_round, 256 /
mysize, MPI_INT, 0, MPI_COMM_WORLD);


    //                                                          STEP 5
    //Mapping the Original Image values to Floor Round values to have the Final
Output Image
    //Here we will prepare "imageData" values to be mapped by values we have from
"floor_round"
    //So then we convert "imageData" values to the Final Output Image using the
given "Create Image" function
    //This is done better sequentially using one processor
    if (myrank == 0)
    {
        for (int i = 0; i < ImageHeight * ImageWidth; i++)
            imageData[i] = floor_round[imageData[i]];
    }
```

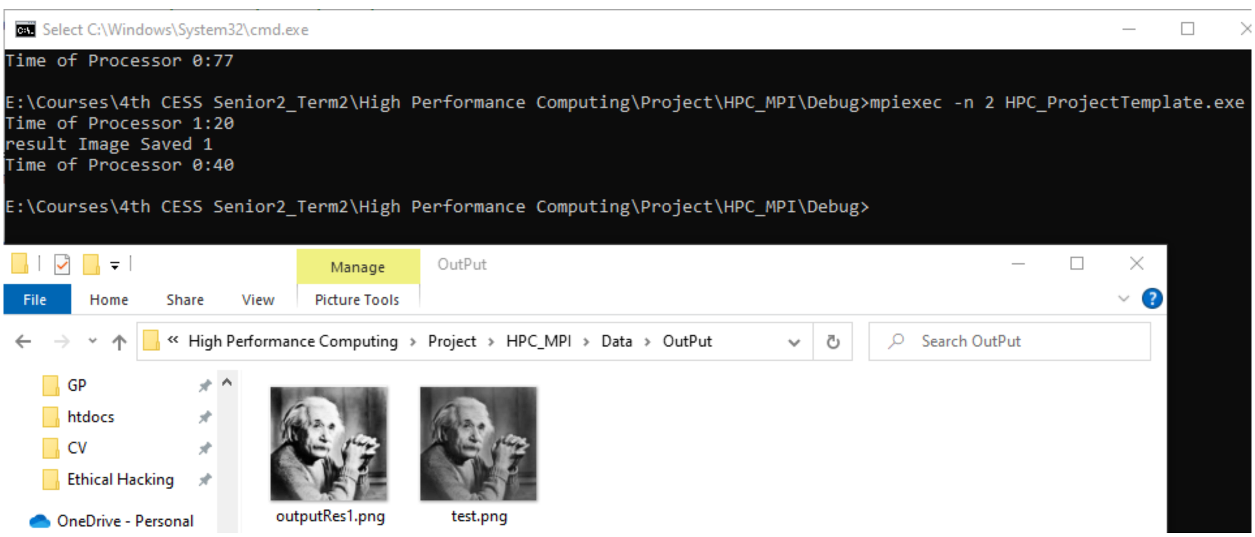
```
        stop_s = clock();
        TotalTime += (stop_s - start_s) / double(CLOCKS_PER_SEC) * 1000;

    //Since "image_Data" array values is only at Processor 0 so it is the only
one that can create the image
    if (myrank == 0)
        createImage(imageData, ImageWidth, ImageHeight, 1);

    cout << "Time of Processor " << myrank << ":" << TotalTime << endl;

    free(imageData);
    MPI_Finalize();
    return 0;
}
```

Input & Output

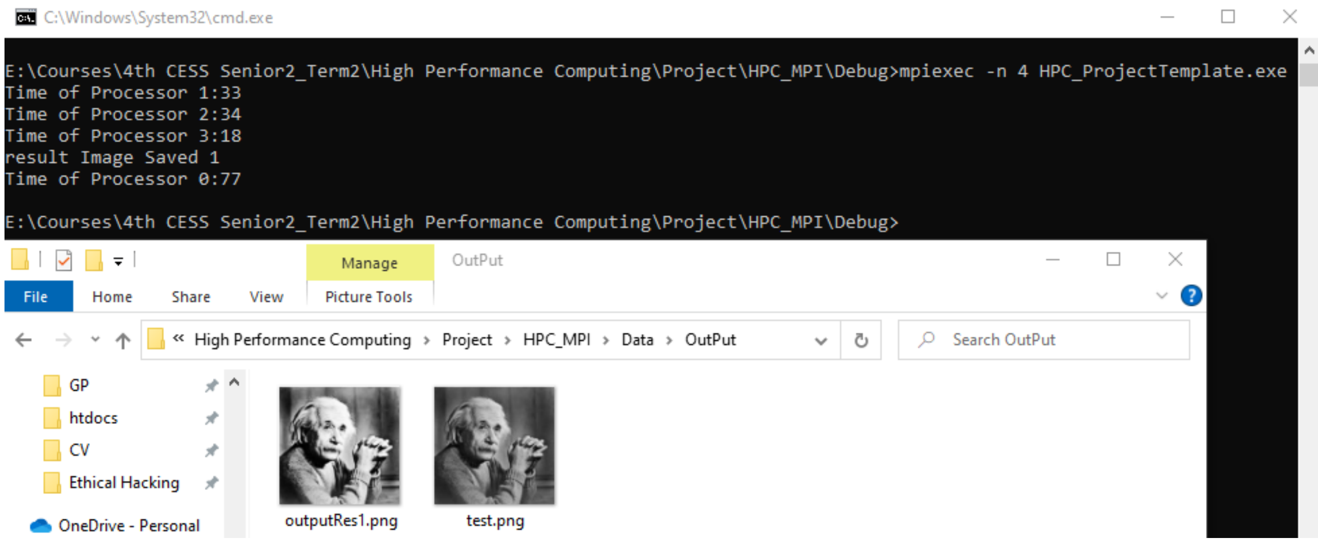
For 1 Processor: Time: 32 ms



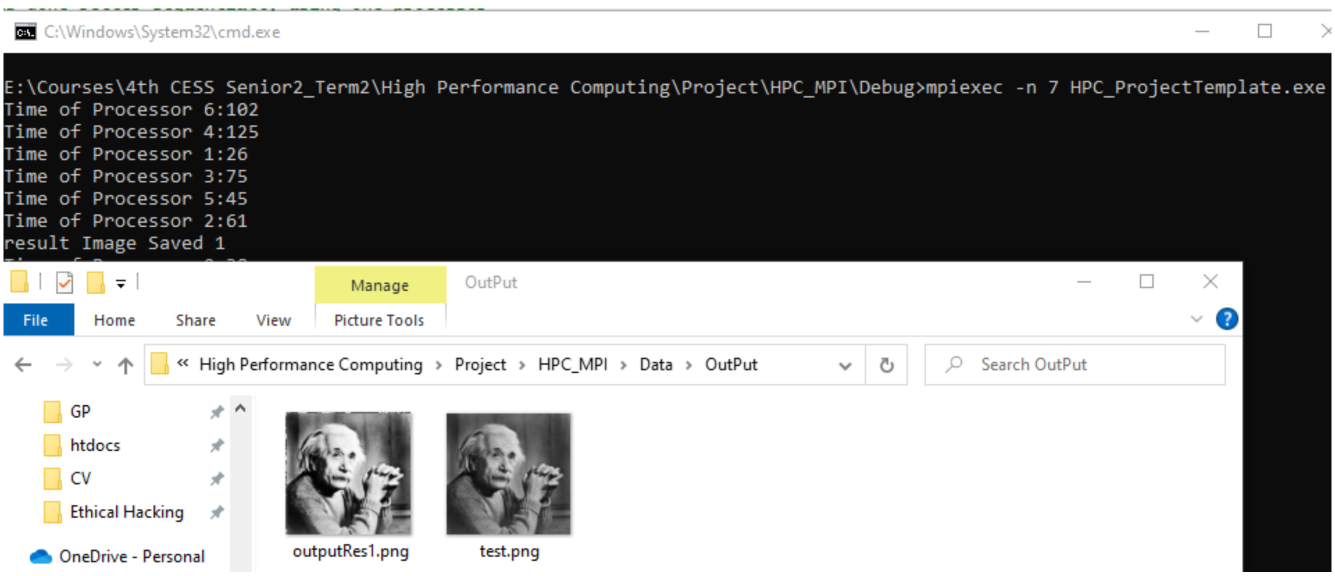
For 2 Processors: Longest Processor time: 40 ms



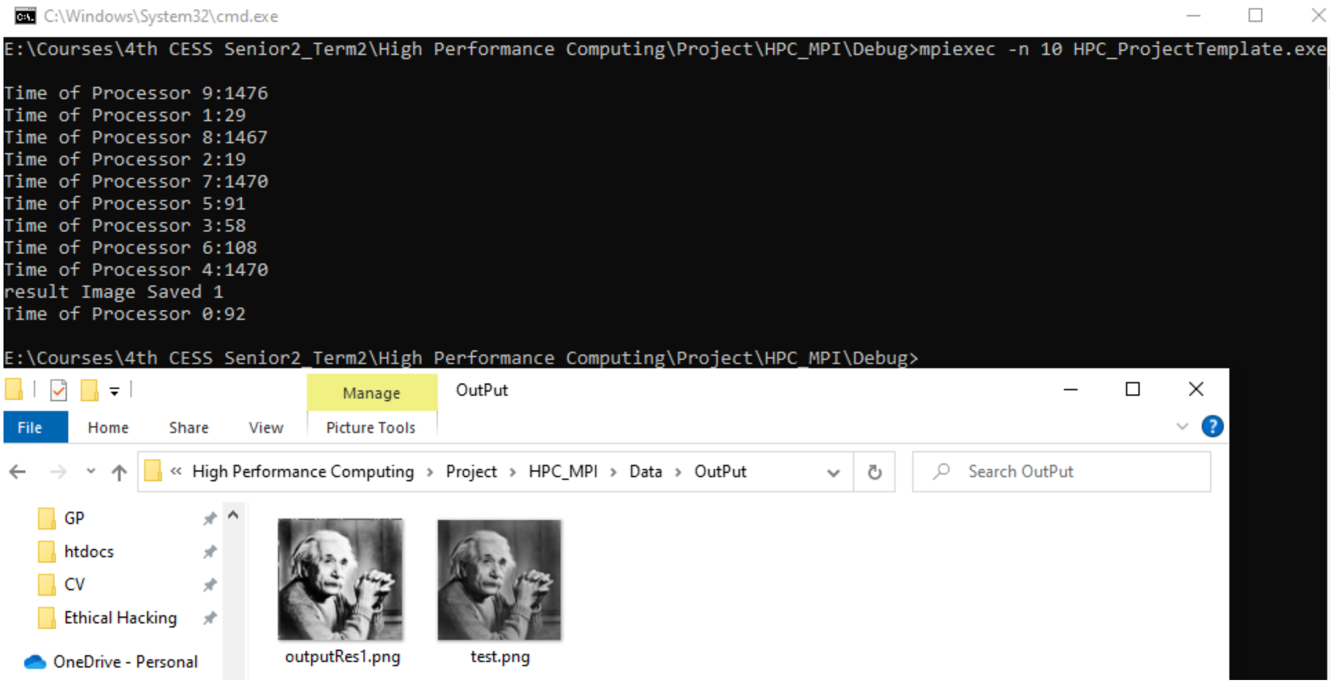
For 4 Processors: Longest Processor time: 77 ms



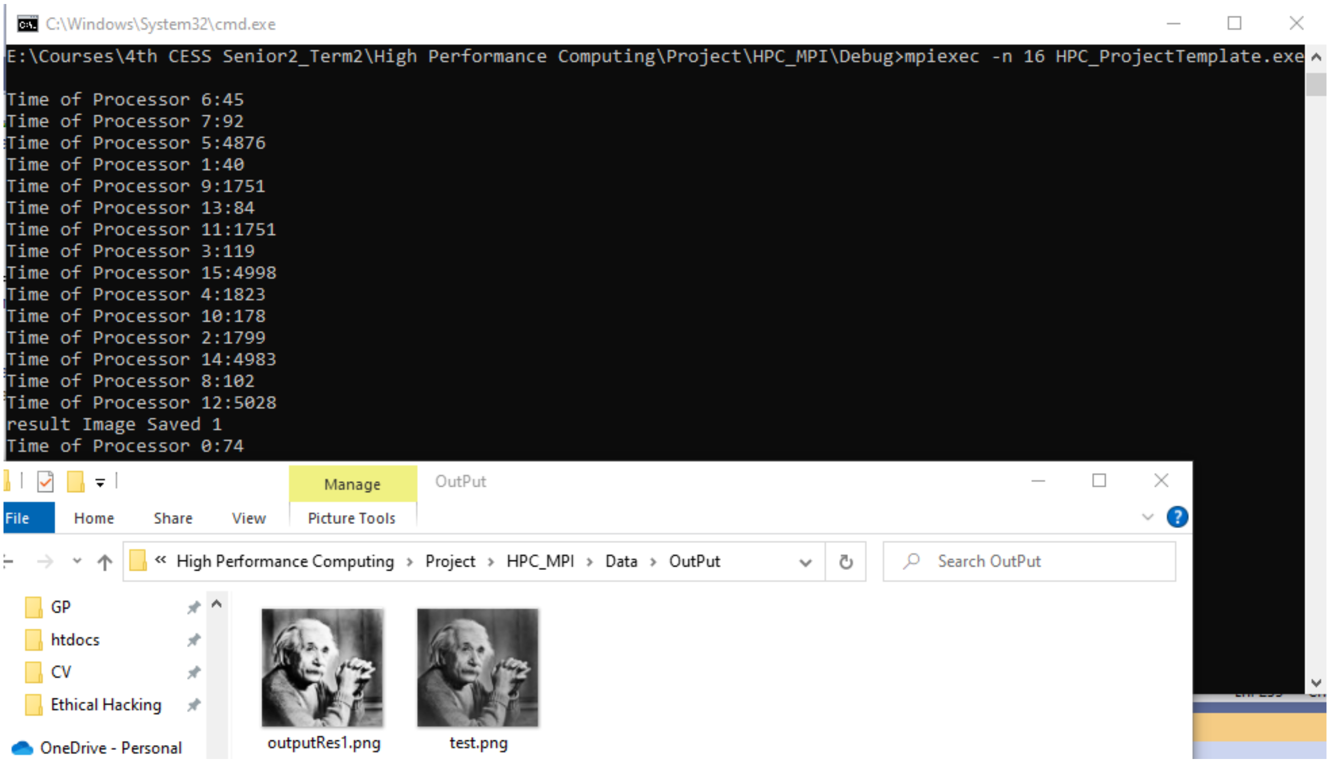
For 7 Processors: Longest Processor time: 125 ms



For 10 Processors: Longest Processor time: 1470 ms



For 16 Processors: Longest Processor time: 5028 ms



3. OpenMP Code

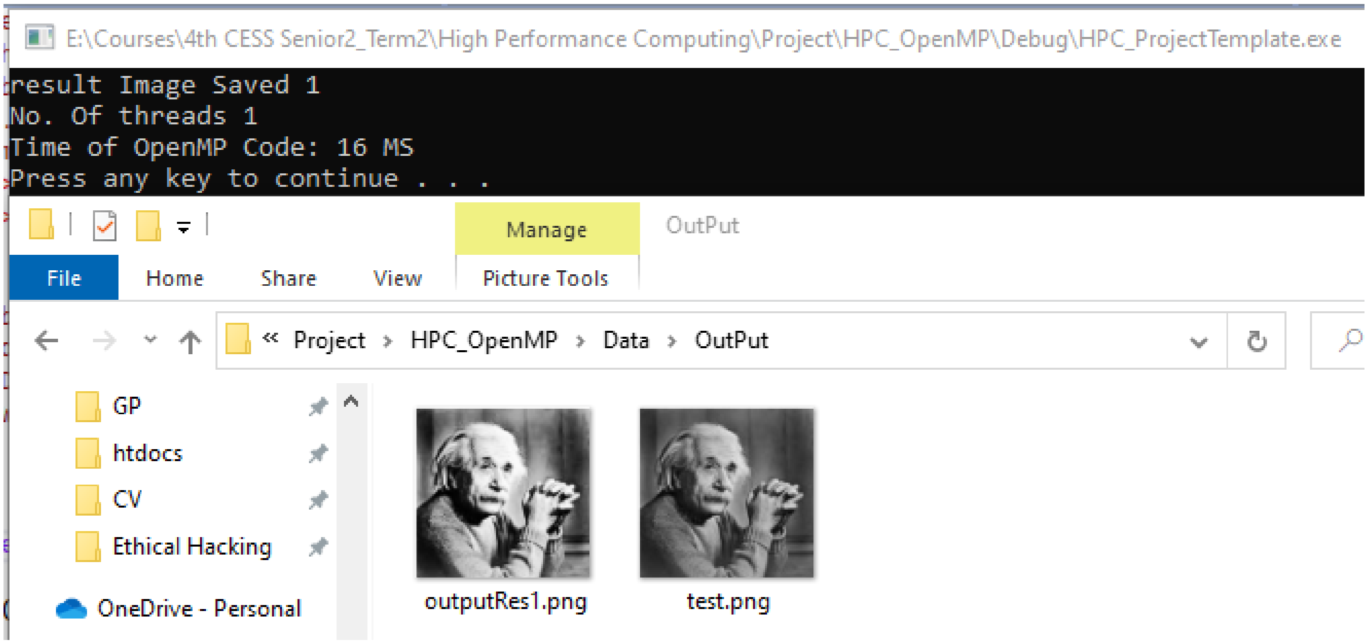
Implementation & Description (In comments)

```
int main()
{

    System::String^ imagePath;
    std::string img;
    img = "..//Data//Input//test.png";

    imagePath = marshal_as<System::String^>(img);
    int* imageData = inputImage(&ImageWidth, &ImageHeight, imagePath);

    start_s = clock();
    /*****************************************************************/
    // initialize needed arrays
    int number_of_pixels[256];
    double probability[256];
    double cumulative_probability[256];
    double scaled_cumulative_probability[256];
    int floored_round[256];
    int number_of_pixels_global[256] = { 0 }; // Shared array to store the final
result
    omp_set_num_threads(4);        //sets the number of threads to be used in the
parallel region
    //starts a parallel region with shared array number_of_pixels_global
#pragma omp parallel shared(number_of_pixels_global)
    {
        int local_number_of_pixels[256] = { 0 }; // Private array for each thread

        //distributes the iterations of the following loop among the threads in
the parallel region.
#pragma omp for
        for (int i = 0; i < (ImageHeight * ImageWidth); i++) {
            // Each thread updates its own local array by incrementing the count
for the intensity value of the corresponding pixel in the image.
            local_number_of_pixels[imageData[i]]++;
        }

        //collect the values of each thrads from local_number_of_pixels to  be
stored in number_of_pixels_global
        for (int i = 0; i < 256; i++) {
            //allow one thrad to write at a time to avoid data race conditions
#pragma omp atomic
```

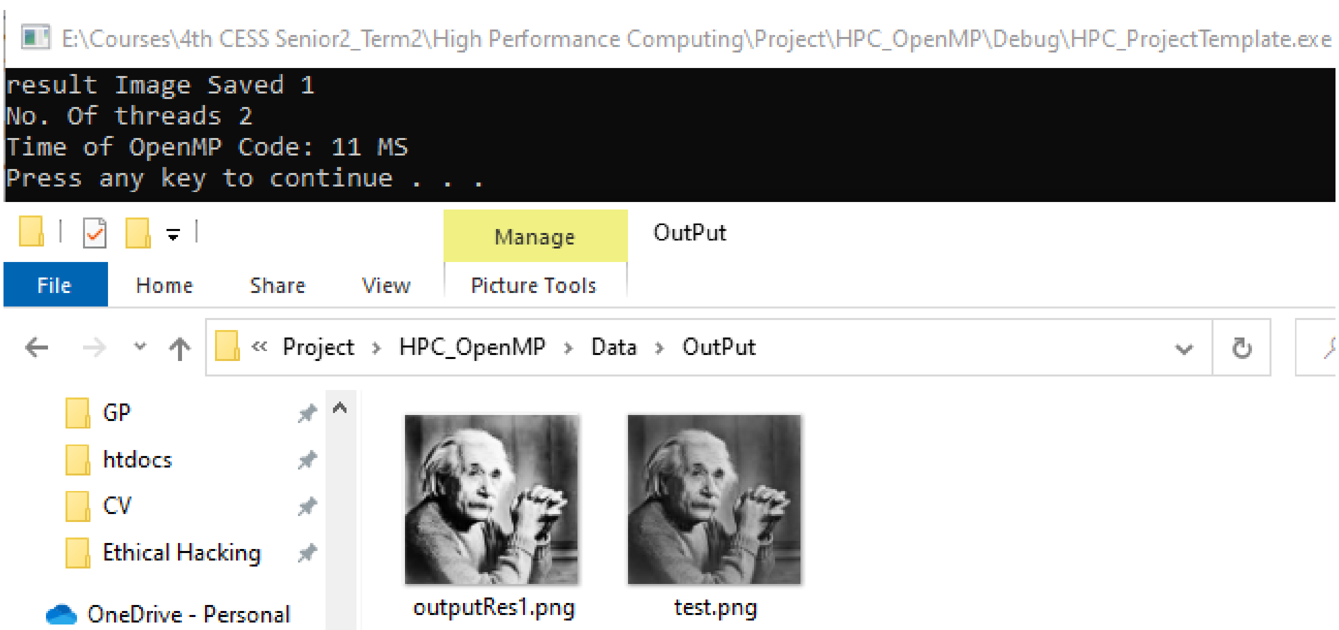
```
                number_of_pixels_global[i] += local_number_of_pixels[i];
            }
    }
    //calculates the probability of each intensity value in parallel
#pragma omp parallel for

    for (int i = 0; i < 256; i++)
        probability[i] = (double)((double)number_of_pixels_global[i] /
(double)(ImageWidth * ImageHeight));
    //calculates the cumulative probabilities by summing the probabilities of
previous intensity values.so its sequentional
    cumulative_probability[0] = probability[0];
    for (int i = 1; i < 256; i++) {
        cumulative_probability[i] = probability[i] + cumulative_probability[i -
1];

    }
    //scales the cumulative probability values by multiplying them by 256 in
parallel
#pragma omp parallel  for
    for (int i = 0; i < 256; i++)
        scaled_cumulative_probability[i] = cumulative_probability[i] * 256;
    //applies the floor function to round down the scaled cumulative
probabilities in parallel
#pragma omp parallel for
    for (int i = 0; i < 256; i++)
        floored_round[i] = floor(scaled_cumulative_probability[i]);
    //replaces the intensity values of the image with the corresponding
floored  values in parallel
#pragma omp parallel for
    for (int i = 0; i < ImageWidth * ImageHeight; i++)
        imageData[i] = floored_round[imageData[i]];
    /***************************************************************************
*/
    stop_s = clock();
    TotalTime += (stop_s - start_s) / double(CLOCKS_PER_SEC) * 1000;
    createImage(imageData, ImageWidth, ImageHeight, 1);
    cout << "time: " << TotalTime << " MS" << endl;
    free(imageData);
    system("pause");
    return 0;

}
```

Input & Output

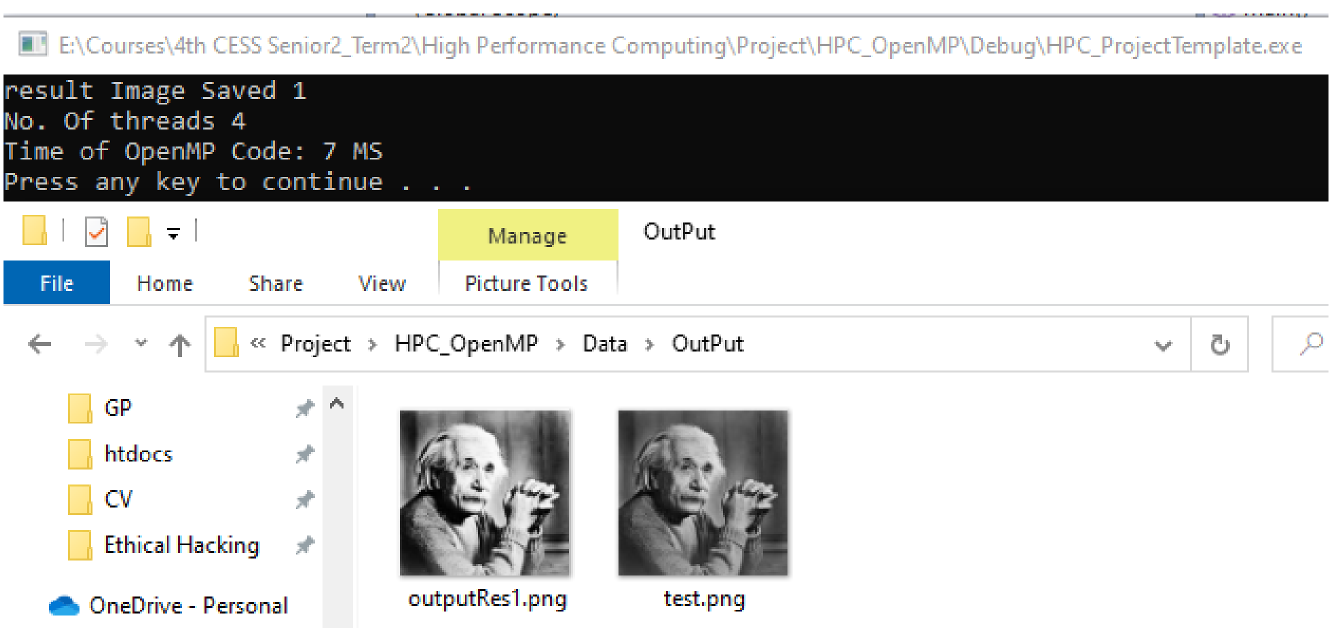
For 1 thread: Time is 16 ms



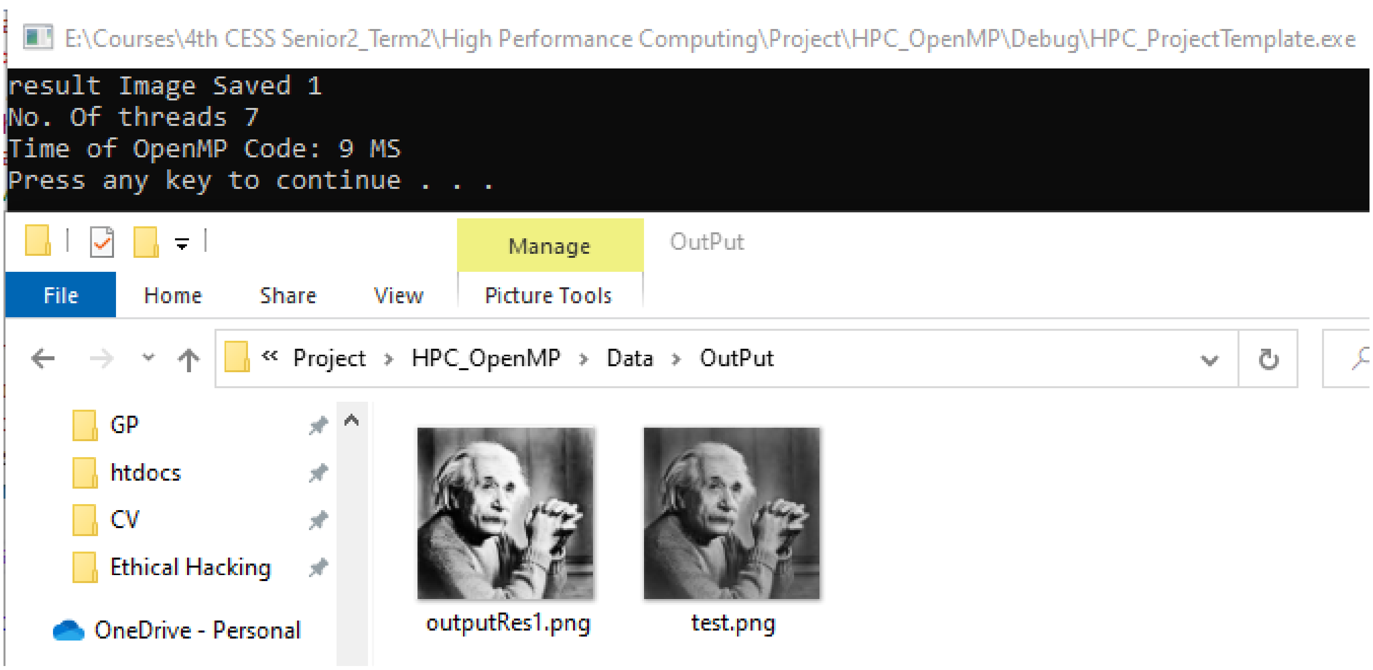
For 2 threads: Time is 11 ms



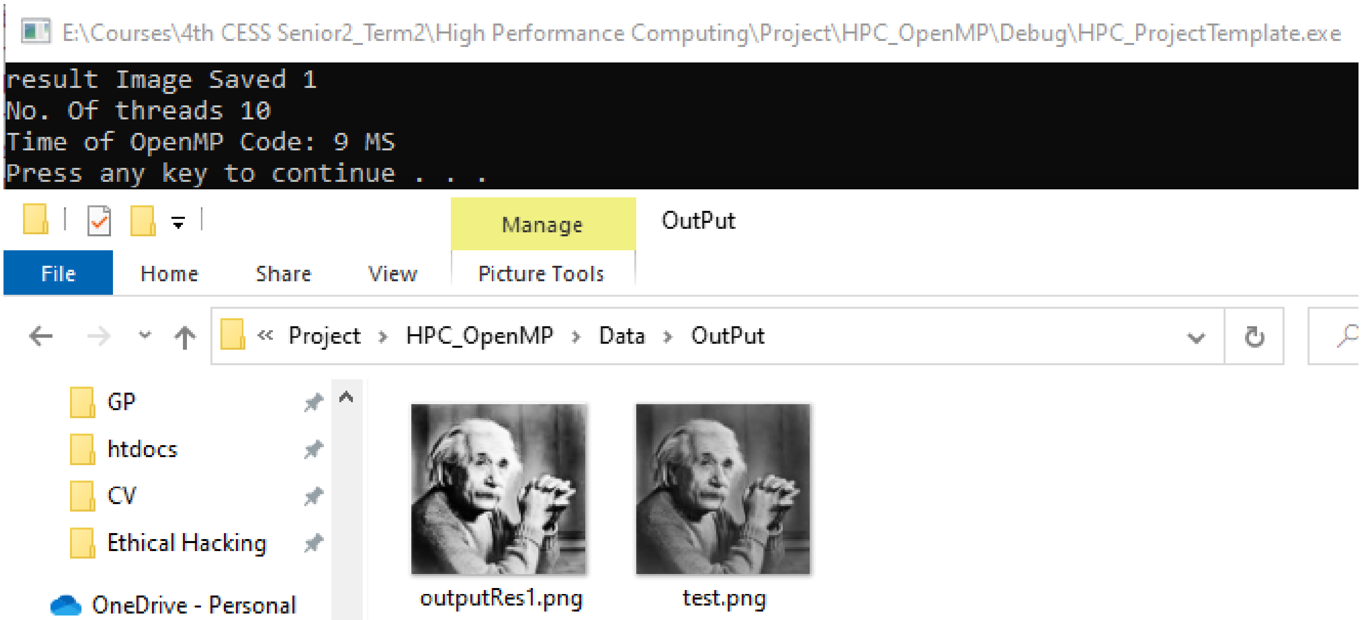
For 4 threads: Time is 7 ms



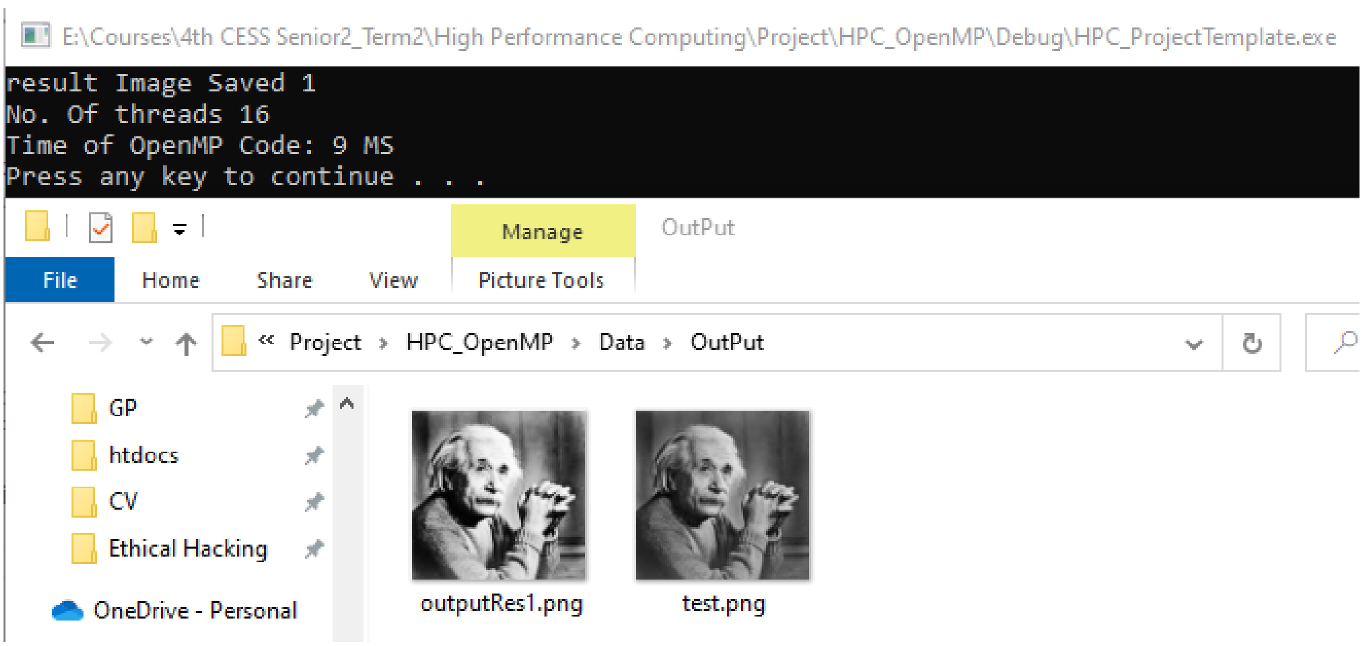
For 7 threads: Time is 9 ms



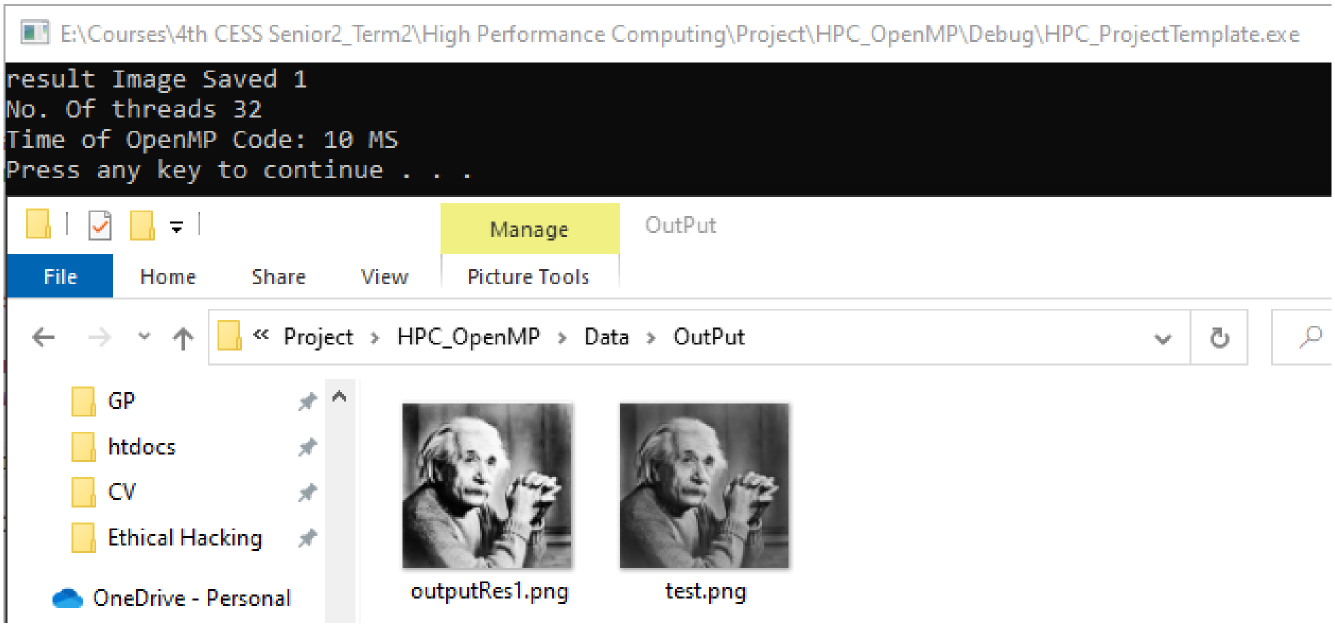
For 10 threads: Time is 9 ms



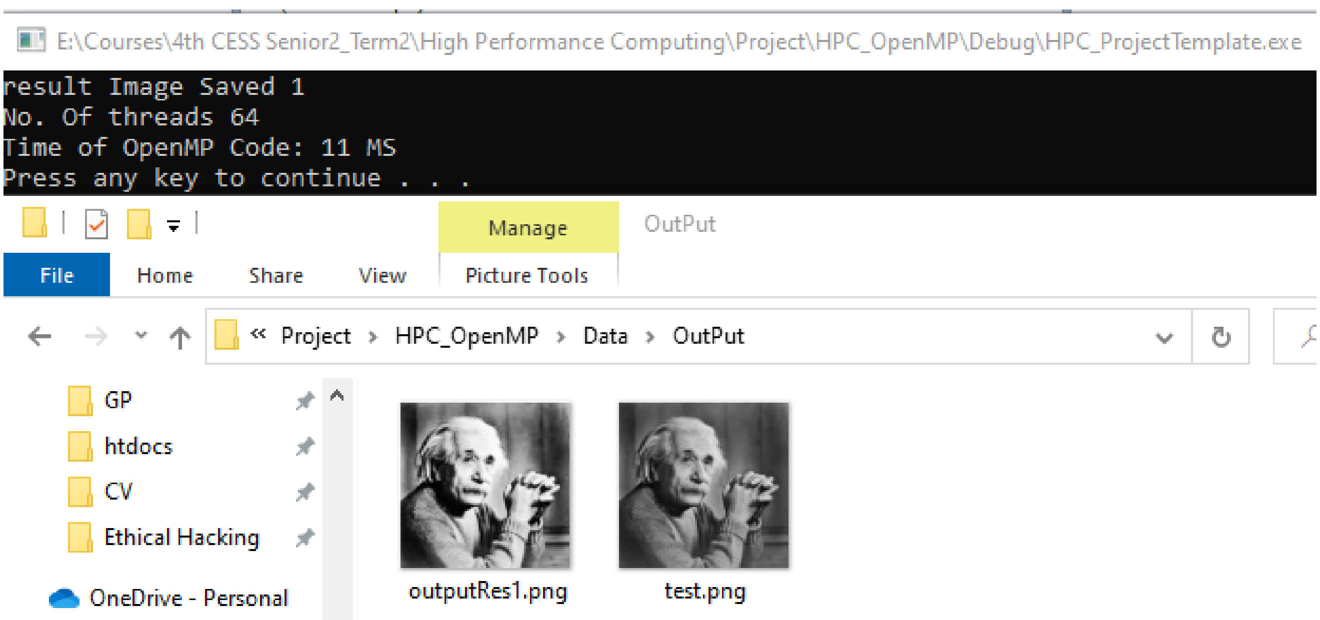
For 16 threads: Time is 9 ms



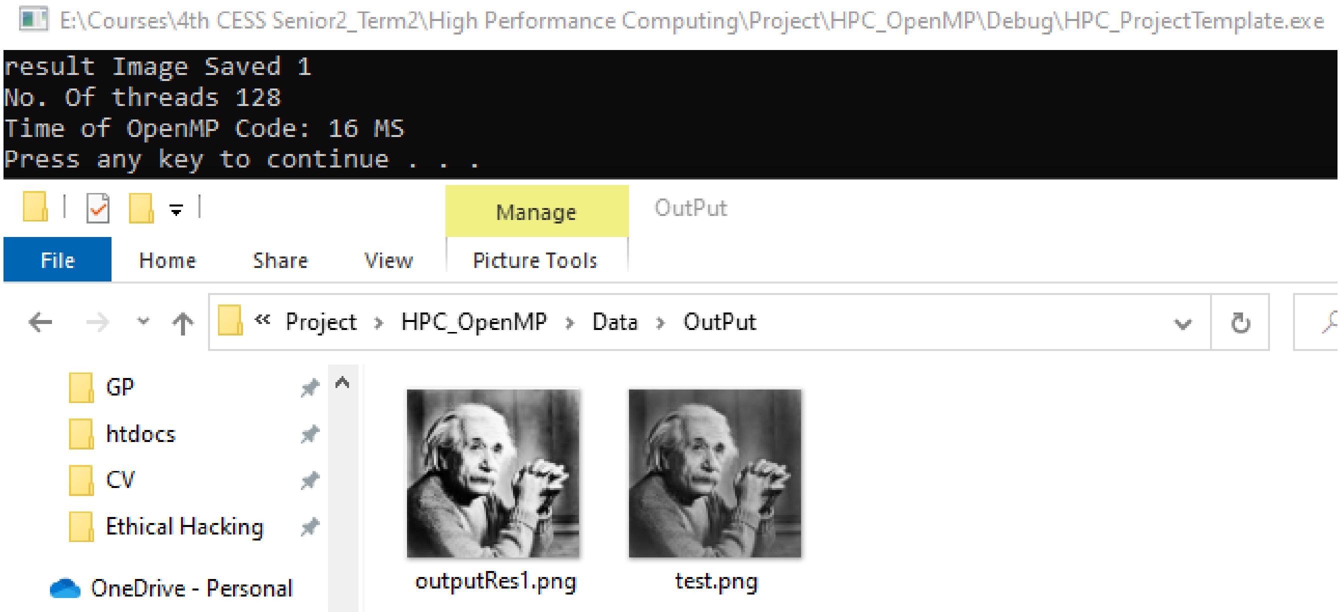
For 32 threads: Time is 10 ms



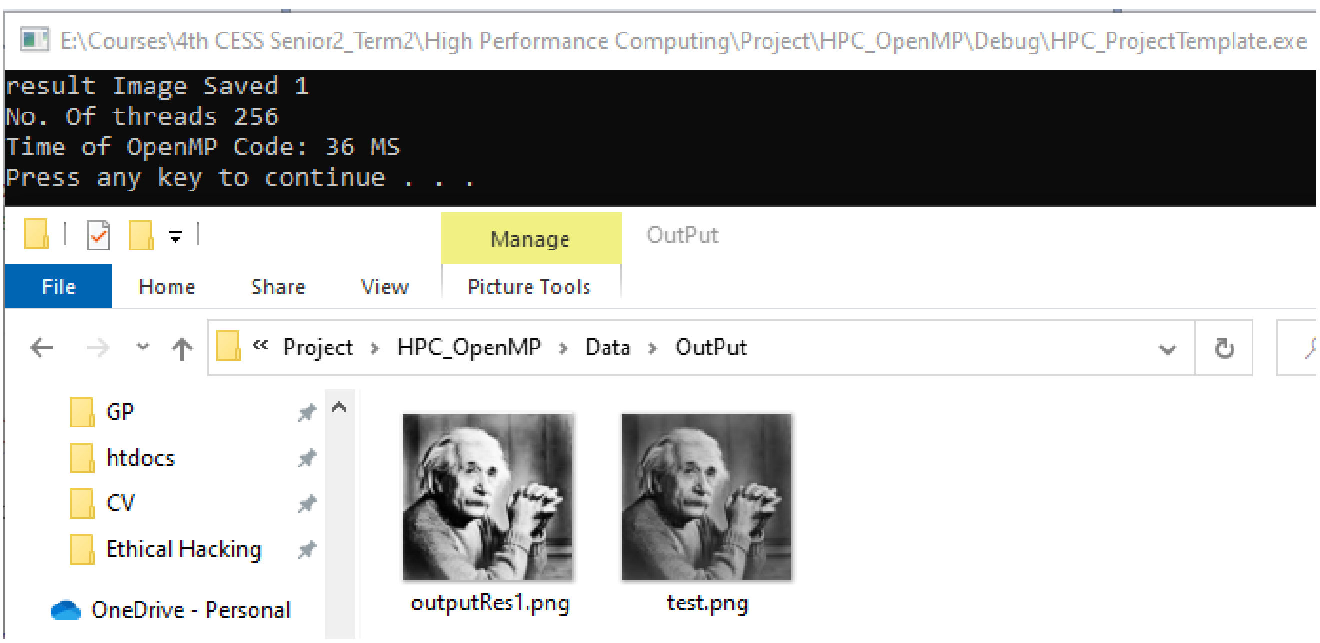
For 64 threads: Time is 11 ms



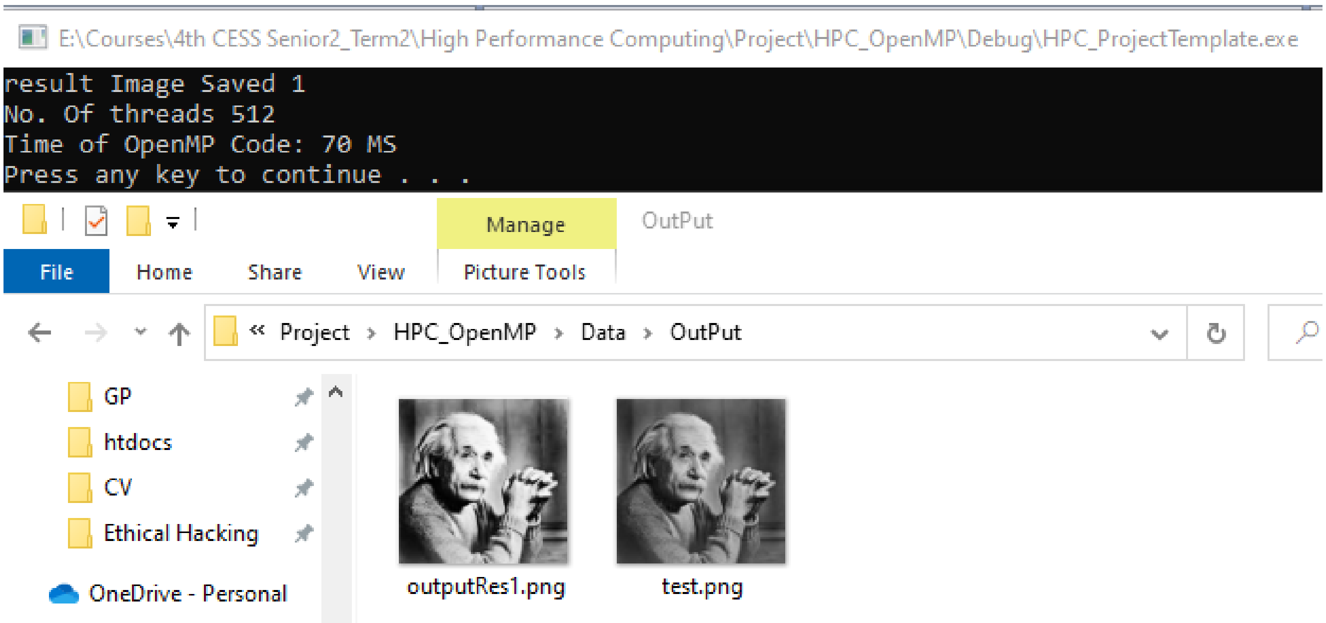
For 128 threads: Time is 16 ms



For 256 threads: Time is 36



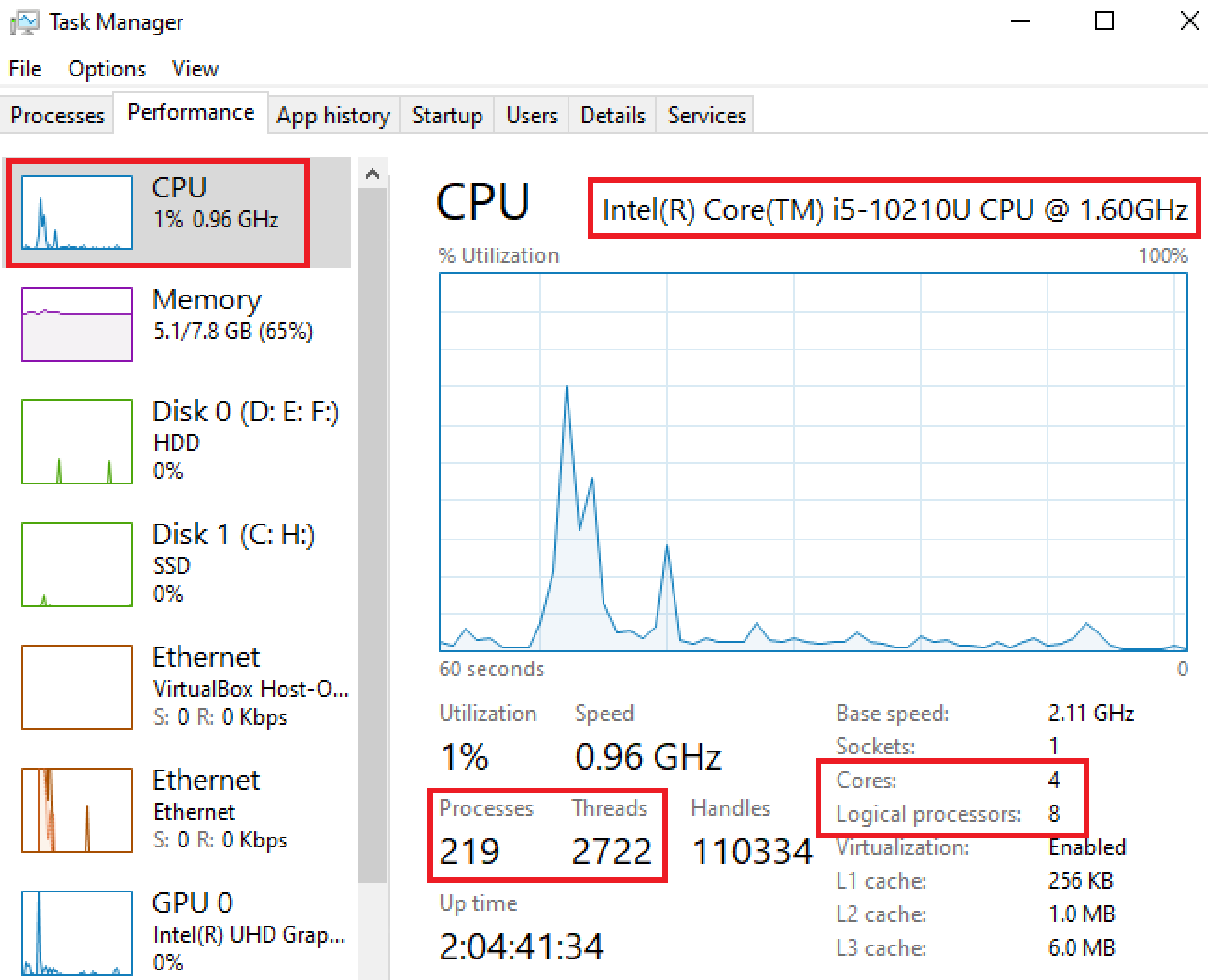
For 512 threads: Time is 70 ms



Comparison

Number Of Processors (for MPI) /Threads (for OpenMP)	1	2	4	7	10	16	32	64	128	256	512
Sequential Code Time (ms)	13	-	-	-	-	-	-	-	-	-	-
MPI Code Time (ms)	32	40	77	125	1470	5028	-	-	-	-	-
OpenMP Code Time (ms)	16	11	7	9	9	9	10	11	16	36	70

CPU Capabilities



Conclusion

Based on the comparison results, codes faster in time are:
1. OpenMP Code (4 processors used is the Best)
2. Sequential Code
3. MPI Code

For OpenMP code, starting with one processor isn’t the best approach because there is reduction and splitting of arrays on threads. However, from using 2-64 processors the code is faster, proving that threads worked on their assigned arrays perfectly to give better results.

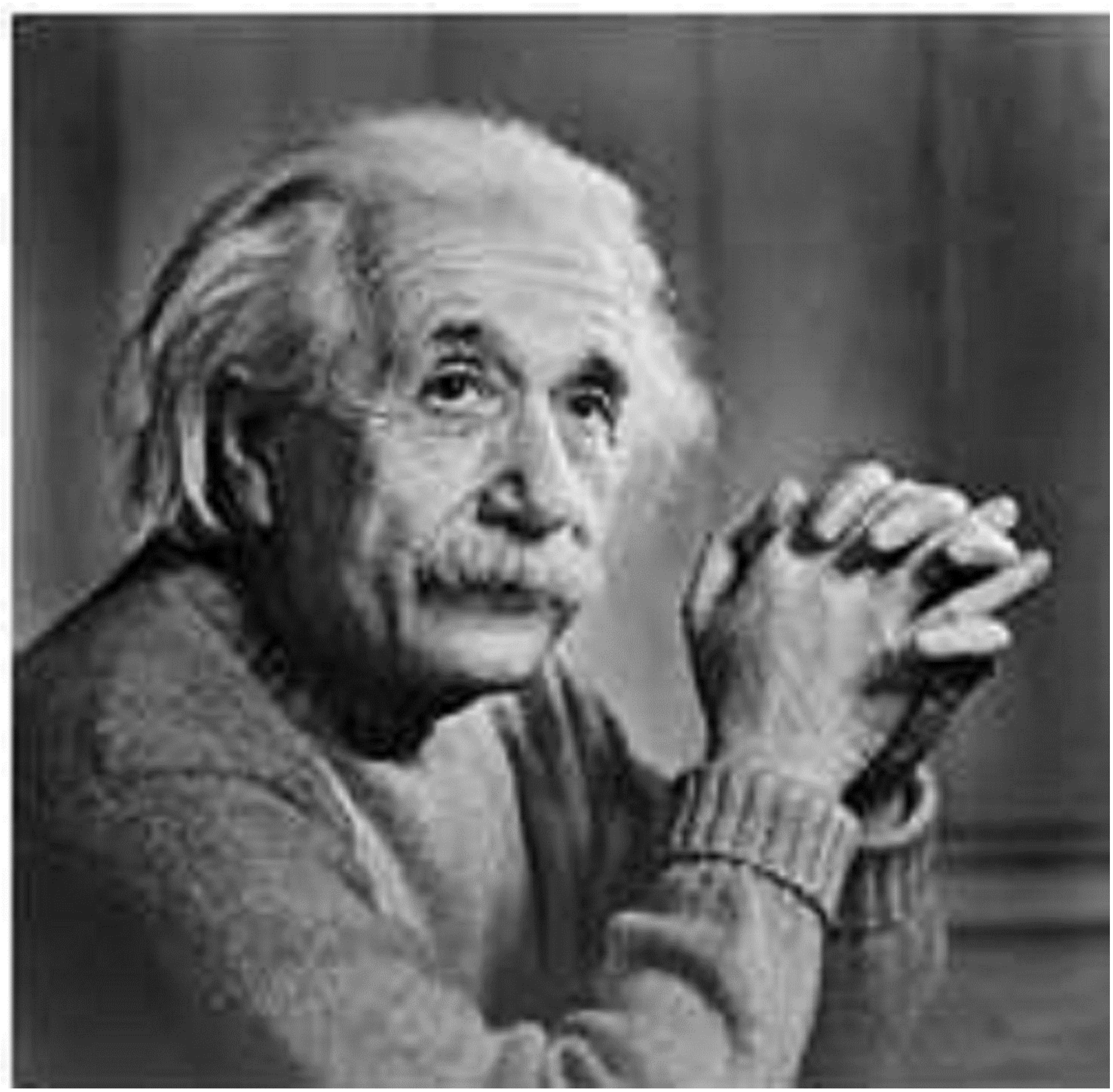
But OpenMP code compared to Sequential code isn’t that much of a difference, due to that the image used isn’t that hard to do it sequentially, but still OpenMP code performs slightly better and can perform faster when applied to a bigger pixelated image.

OpenMP code becomes slower as we increase number of processors from 128 processors, this is due to the reduction of summation calculated is split on much arrays that it takes more time to gather these values.

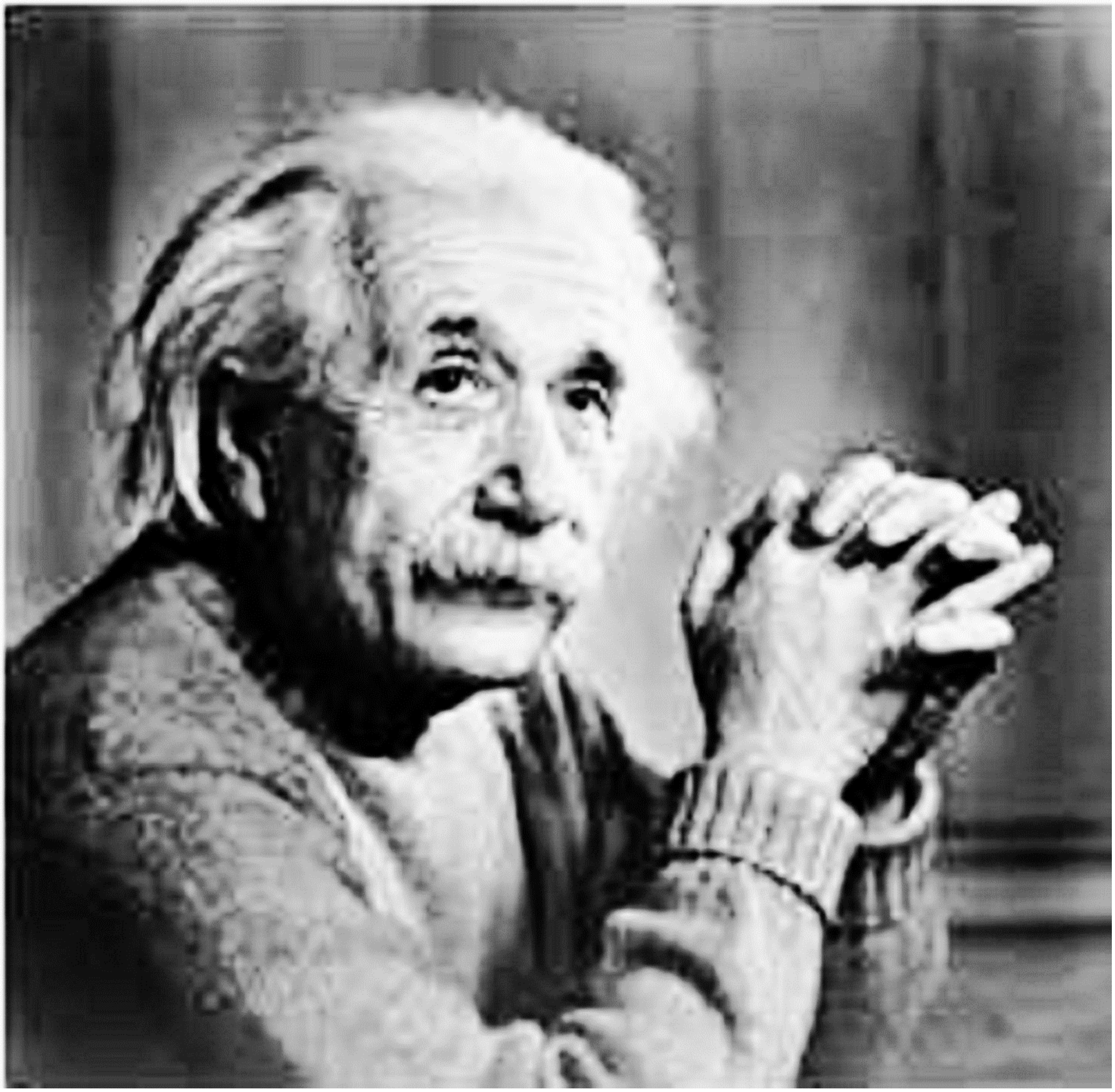
For MPI code, it seems that MPI approach isn’t suitable for 2 reasons:
1. The image isn’t big enough that it needs to be done in parallel, sequential is enough.
2. You will notice that MPI code tries at its best to parallel any possible code that can be parallel using “MPI_Scatter” & “MPI_Gather” but calling these methods multiple times cause a HUGE Communication Overhead which slows the code significantly.

Input & Output Images Tried On 2 Images

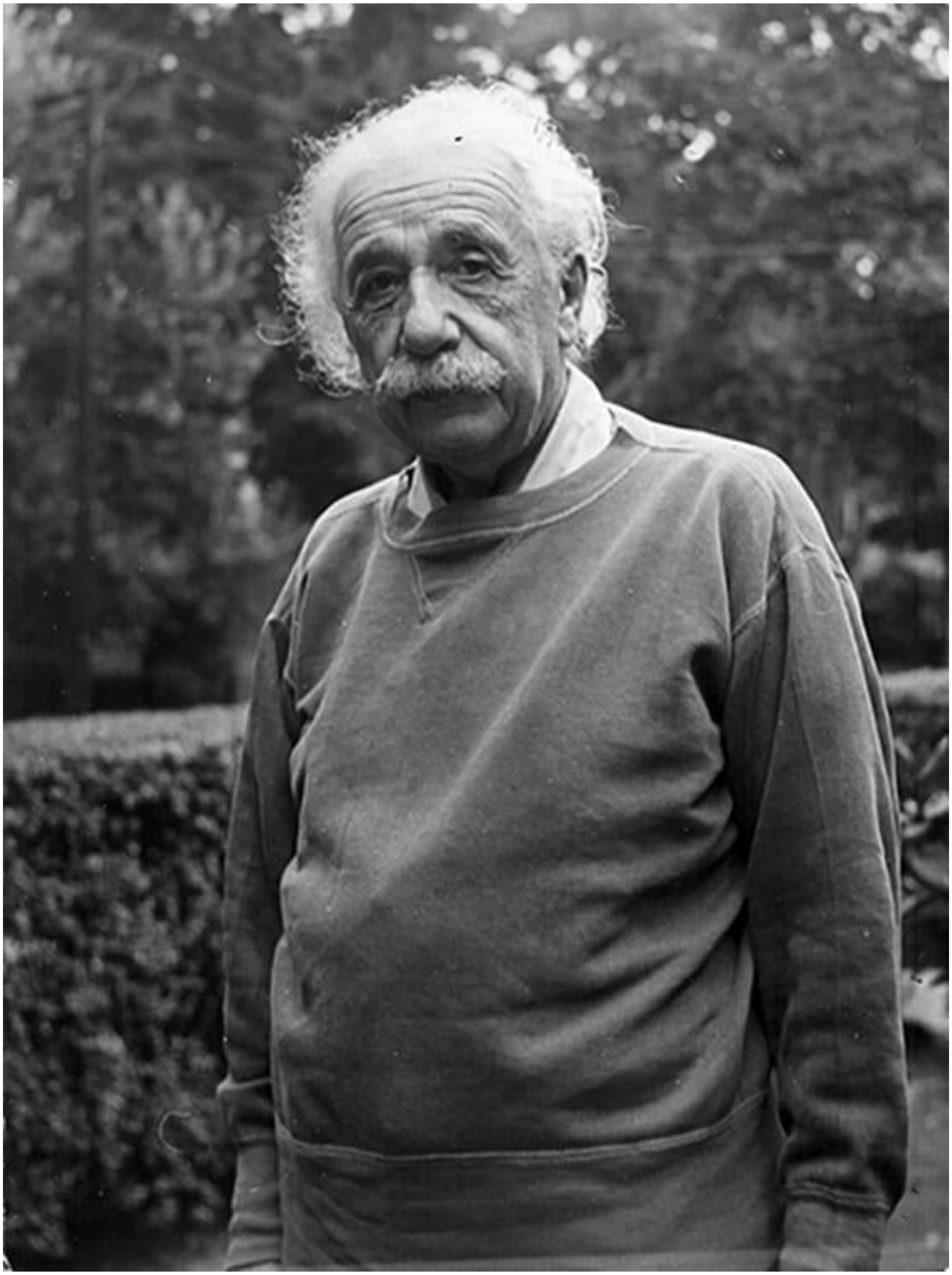
Input



Output



Input



Output