Major Task

CSE455, High-Performance Computing

Date: 9/ 5 / 2024

Group 10

Aya Ahmed Gamal	19P1689
Karen Alber Farid	19P8948
Shaimaa Mohamed	19P7484
Rana Mohamed	19P8994
Alyeldeen Khaled	19P1512

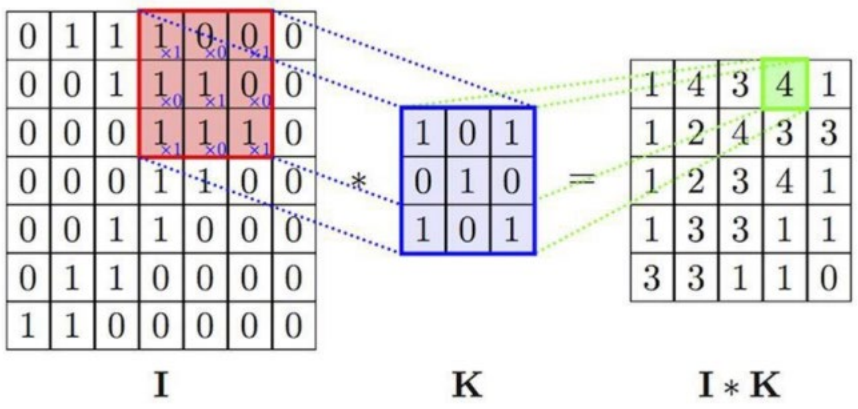
Table of Contents

Project #1: Low Pass Filter

Description:

It is used to make images appear smoother. Low pass filtering smooths out noise. It allows low frequency components of the image to pass through and blocks high frequencies. Low pass image filters work by convolution which is a process of making each pixel of an image by a fixed size kernel.

Convolution operation

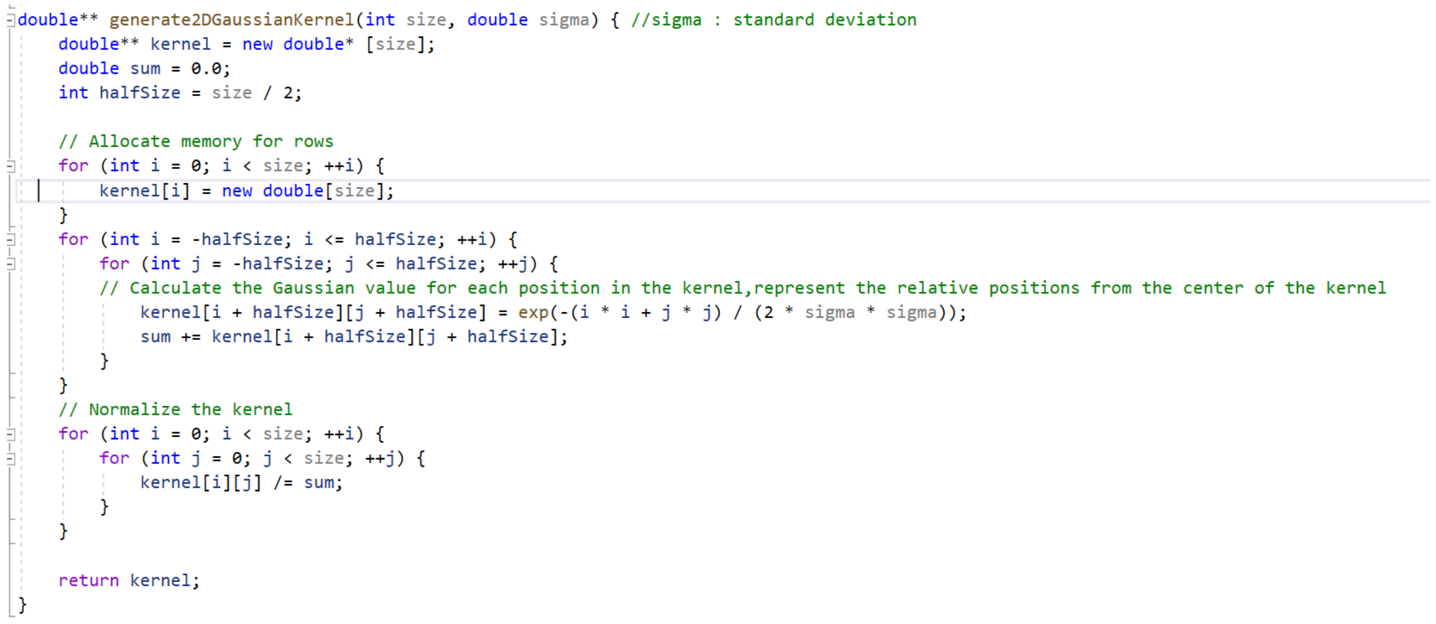


We assume Image is I, the kernel is K. If we applied the filter on the red region at I, the result will be computed by aligning the kernel onto the image part then doing basic multiplication between the aligned elements as: (1*1 + 0*0 + 0*1) + (0*1 + 1*1 + 0*0) + (1*1 + 1*0 + 1*1) = 4. Every kernel-based filter has its kernel, the low pass filter has this as a kernel:

1/9	1/9	1/9
1/9	1/9	1/9
1/9	1/9	1/9

Part One: Sequential

1. Generate a 2D gaussian filter



```cpp
double** generate2DGaussianKernel(int size, double sigma) { //sigma : standard deviation
    double** kernel = new double* [size];
    double sum = 0.0;
    int halfSize = size / 2;

    // Allocate memory for rows
    for (int i = 0; i < size; ++i) {
        kernel[i] = new double[size];
    }
    for (int i = -halfSize; i <= halfSize; ++i) {
        for (int j = -halfSize; j <= halfSize; ++j) {
        // Calculate the Gaussian value for each position in the kernel,represent the relative positions from the center of the kernel
            kernel[i + halfSize][j + halfSize] = exp(-(i * i + j * j) / (2 * sigma * sigma));
            sum += kernel[i + halfSize][j + halfSize];
        }
    }
    // Normalize the kernel
    for (int i = 0; i < size; ++i) {
        for (int j = 0; j < size; ++j) {
            kernel[i][j] /= sum;
        }
    }

    return kernel;
}
```

In the field of image processing we make an extensive use of Gaussian filtering as it is employed to lessen an image's noise. We will create a 2D Gaussian kernel using the gaussian distribution equation:

$$G(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

Where, y is the distance along vertical axis from the origin, x is the distance along horizontal axis from the origin.

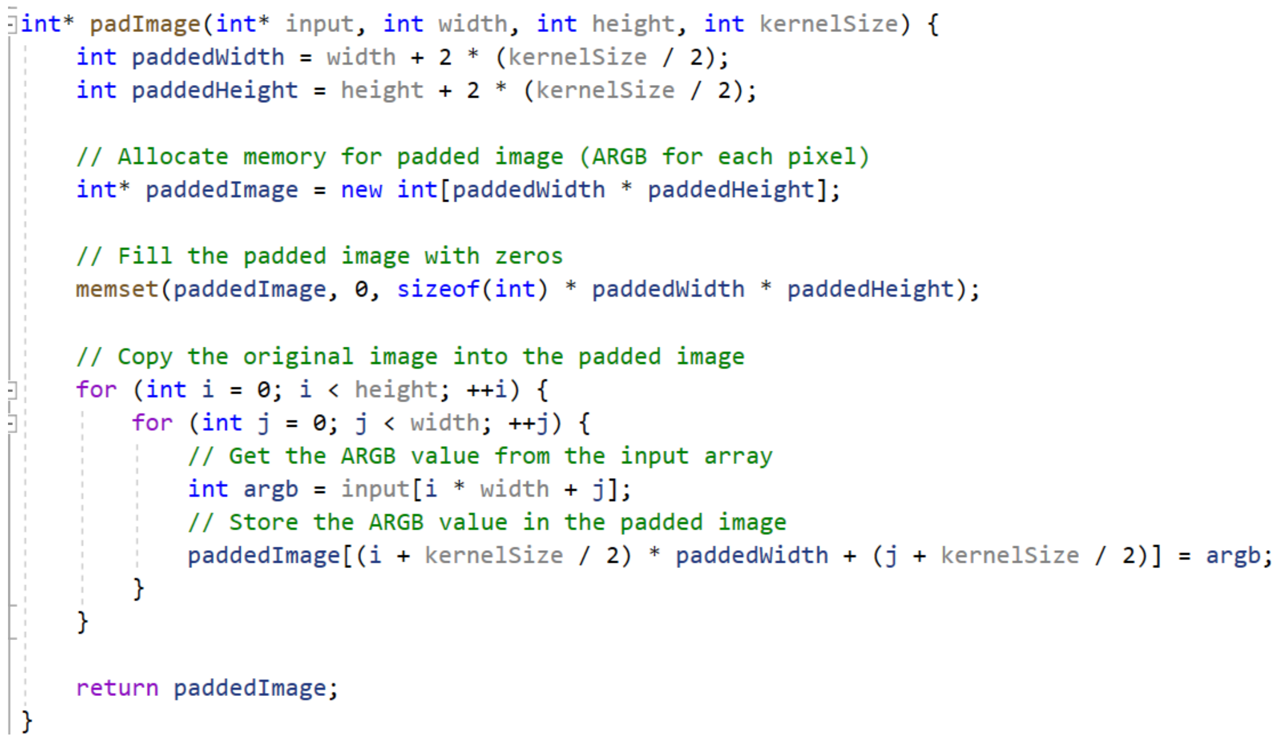
Our function steps:

1. It dynamically allocates memory for a 2D array (kernel) of size **size x size**.
2. It initializes a variable **sum** to keep track of the sum of all values in the kernel.
3. It calculates the half size of the kernel.
4. It allocates memory for each row of the kernel.
5. It loops through each element of the kernel in form of rows (i) and columns(j) and calculates the Gaussian value for each position. The Gaussian value is computed using the formula mentioned above where x and y represent the relative positions from the center of the kernel, and sigma is the standard deviation.
6. It normalizes the kernel by dividing each element by the sum of all elements.
7. It returns the generated Gaussian kernel.

If the size is 3*3 and sigma is 1 , the result will be 2D array like this :

```
kernel = [
   [0.04, 0.18, 0.04],
   [0.18, 0.64, 0.18],
   [0.04, 0.18, 0.04]
]
```

2. Padding to the image



```
int* padImage(int* input, int width, int height, int kernelSize) {
    int paddedWidth = width + 2 * (kernelSize / 2);
    int paddedHeight = height + 2 * (kernelSize / 2);

    // Allocate memory for padded image (ARGB for each pixel)
    int* paddedImage = new int[paddedWidth * paddedHeight];

    // Fill the padded image with zeros
    memset(paddedImage, 0, sizeof(int) * paddedWidth * paddedHeight);

    // Copy the original image into the padded image
    for (int i = 0; i < height; ++i) {
        for (int j = 0; j < width; ++j) {
            // Get the ARGB value from the input array
            int argb = input[i * width + j];
            // Store the ARGB value in the padded image
            paddedImage[(i + kernelSize / 2) * paddedWidth + (j + kernelSize / 2)] = argb;
        }
    }

    return paddedImage;
}
```

Padding is essential in image processing because it ensures that the kernel can be applied to all pixels in the image, including those at the edges, without causing boundary effects or losing information.

Illustration to the function above :

1. It calculates the dimensions of the padded image by adding twice the kernel's half size to the width and height of the original image. This ensures that the kernel can be centered on any pixel in the original image.
2. It allocates memory for the padded image.
3. It fills the padded image with zeros, initializing it.
4. It copies the original image into the padded image, leaving a border around the original image equal to the kernel's half size. This effectively pads the original image with zeros.

Suppose we have a 3x3 input image:

[1, 2, 3]

[4, 5, 6]   and a kernel size of 3x3

[7, 8, 9]

padded image will be 5*5 :

[0, 0, 0, 0, 0]

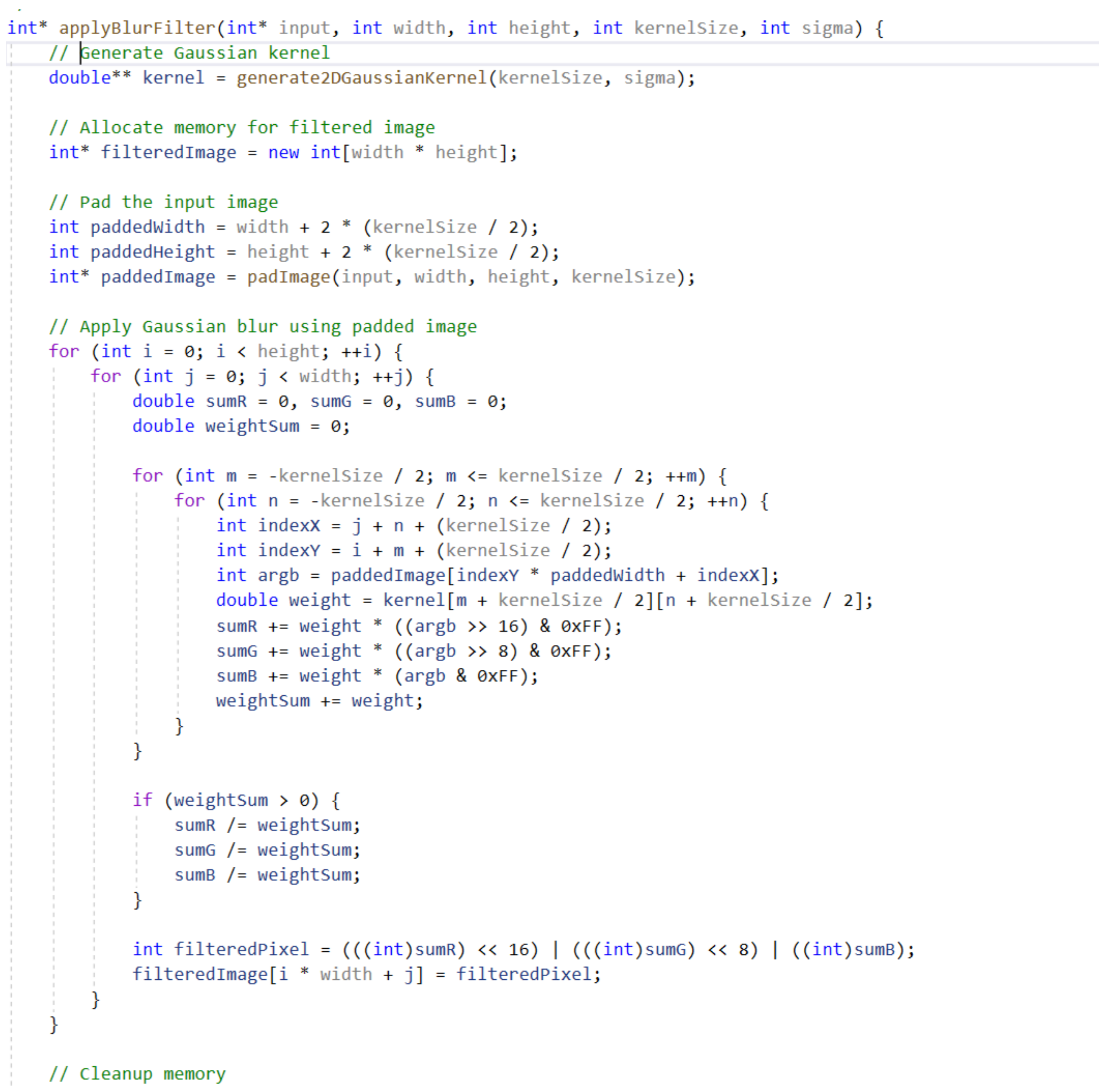
[0, 1, 2, 3, 0]

[0, 4, 5, 6, 0]

[0, 7, 8, 9, 0]

[0, 0, 0, 0, 0]

3. Apply the filter on image



```
int* applyBlurFilter(int* input, int width, int height, int kernelSize, int sigma) {
    // Generate Gaussian kernel
    double** kernel = generate2DGaussianKernel(kernelSize, sigma);

    // Allocate memory for filtered image
    int* filteredImage = new int[width * height];

    // Pad the input image
    int paddedWidth = width + 2 * (kernelSize / 2);
    int paddedHeight = height + 2 * (kernelSize / 2);
    int* paddedImage = padImage(input, width, height, kernelSize);

    // Apply Gaussian blur using padded image
    for (int i = 0; i < height; ++i) {
        for (int j = 0; j < width; ++j) {
            double sumR = 0, sumG = 0, sumB = 0;
            double weightSum = 0;

            for (int m = -kernelSize / 2; m <= kernelSize / 2; ++m) {
                for (int n = -kernelSize / 2; n <= kernelSize / 2; ++n) {
                    int indexX = j + n + (kernelSize / 2);
                    int indexY = i + m + (kernelSize / 2);
                    int argb = paddedImage[indexY * paddedWidth + indexX];
                    double weight = kernel[m + kernelSize / 2][n + kernelSize / 2];
                    sumR += weight * ((argb >> 16) & 0xFF);
                    sumG += weight * ((argb >> 8) & 0xFF);
                    sumB += weight * (argb & 0xFF);
                    weightSum += weight;
                }
            }

            if (weightSum > 0) {
                sumR /= weightSum;
                sumG /= weightSum;
                sumB /= weightSum;
            }

            int filteredPixel = (((int)sumR) << 16) | (((int)sumG) << 8) | ((int)sumB);
            filteredImage[i * width + j] = filteredPixel;
        }
    }

    // Cleanup memory
```

Illustration

- **Apply Gaussian Blur using Padded Image**:
    - We iterate over each pixel in the original input image.
    - Initializing Sums and Weight Sum:

      We initialize variables to hold the weighted sums of the color channels (Red, Green, Blue) and a variable to store the sum of weights. These will be used to compute the average color value for the pixel.
    - For each pixel, we iterate over the kernel centered at the current pixel's position. This involves looping over the rows and columns of the kernel then we calculate the indices in the padded image corresponding to the current position in the kernel. This allows us to access the pixel values from the padded image.
    - We retrieve the ARGB color value of the pixel from the padded image at the calculated indices.
    - Computing Weighted Sums:
      We compute the weighted sum of the color channels (R, G, B) by multiplying the pixel color values with the corresponding weights from the Gaussian kernel.

      ```
      double weight = kernel[m + kernelSize / 2][n + kernelSize / 2];
      sumR += weight * ((argb >> 16) & 0xFF);
      sumG += weight * ((argb >> 8) & 0xFF);
      sumB += weight * (argb & 0xFF);
      ```
    - We accumulate the weights to calculate the total weight applied to the pixel.
    - Normalizing Sums:
      After processing all neighboring pixels within the kernel, we normalize the weighted sums by dividing them by the total weight applied to the pixel.
    - Combining Channel Sums:
      We combine the normalized sums of the color channels (R, G, B) to obtain the filtered pixel value.

- Finally, we assign the computed filtered pixel value to the corresponding position in the filtered image.

Continue on the above example :

```
kernel = [
   [0.04, 0.18, 0.04],
   [0.18, 0.64, 0.18],
   [0.04, 0.18, 0.04]
]
paddedImage = [
   [0, 0, 0, 0, 0],
   [0, 1, 2, 3, 0],
   [0, 4, 5, 6, 0],
   [0, 7, 8, 9, 0],
   [0, 0, 0, 0, 0]
]
```

For the pixel at (0, 0)

```
weight = kernel[0][0] = 0.04
argb = paddedImage[0][0] = 0
sumR += 0.04 * ((0 >> 16) & 0xFF) = 0
sumG += 0.04 * ((0 >> 8) & 0xFF) = 0
sumB += 0.04 * (0 & 0xFF) = 0
```

for the pixel (0,1)

weight = kernel[0][1] = 0.18

argb = paddedImage[0][1] = 0

sumR += 0.18 * ((0 >> 16) & 0xFF) = 0

sumG += 0.18 * ((0 >> 8) & 0xFF) = 0

sumB += 0.18 * (0 & 0xFF) = 0

for the pixel (0,2)

weight = kernel[0][2] = 0.04

argb = paddedImage[0][2] = 0

sumR += 0.04 * ((0 >> 16) & 0xFF) = 0

sumG += 0.04 * ((0 >> 8) & 0xFF) = 0

sumB += 0.04 * (0 & 0xFF) = 0

we will get values of pixels (0,0),(0,1) ,(0,2)

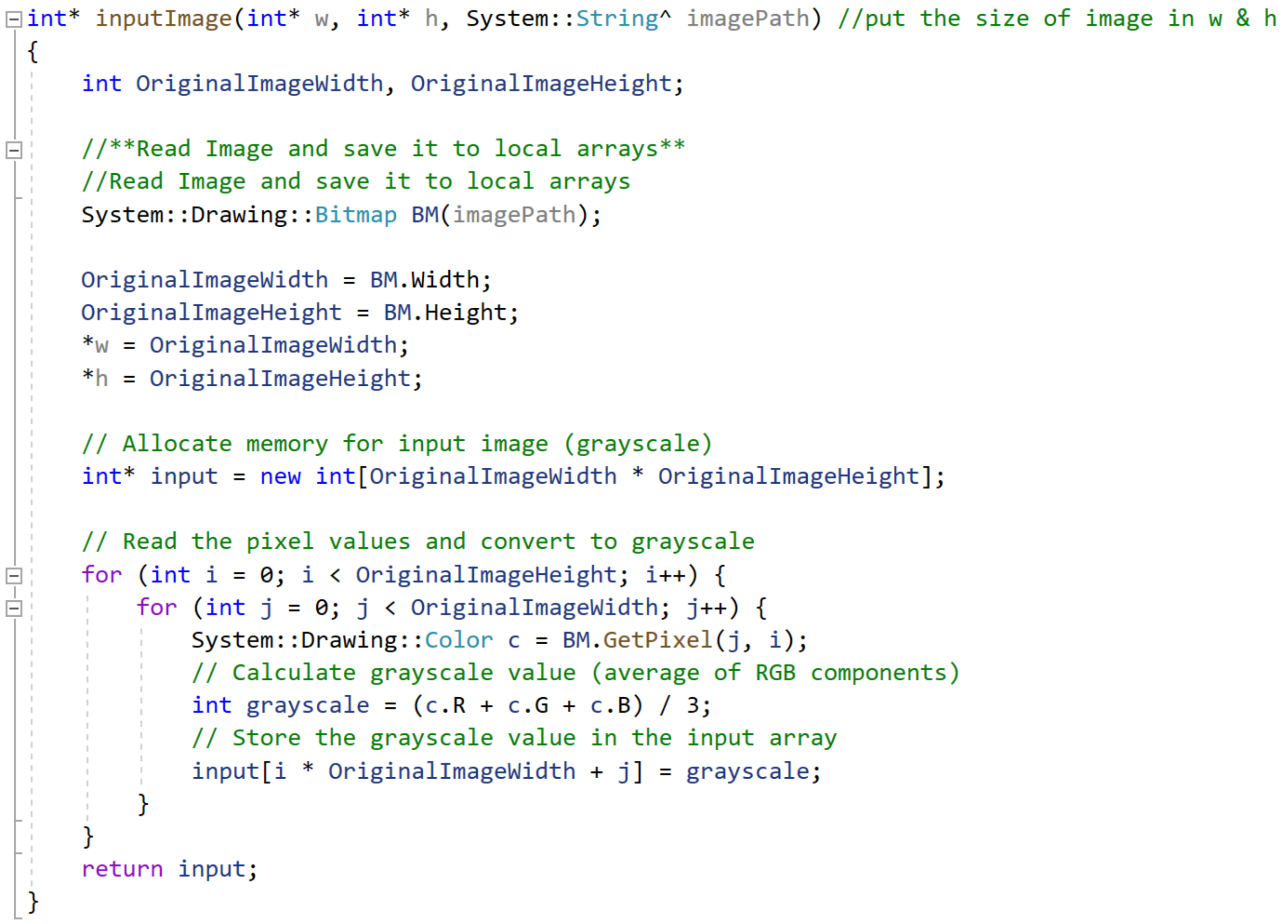
(1,0),(1,1),(1,2)

(2,0),(2,1),(2,2)

Then get the weighted sum as mentioned in the function above, and each pixel in the filtered image will have the value after making the convolution process and rgb sum of each pixel.

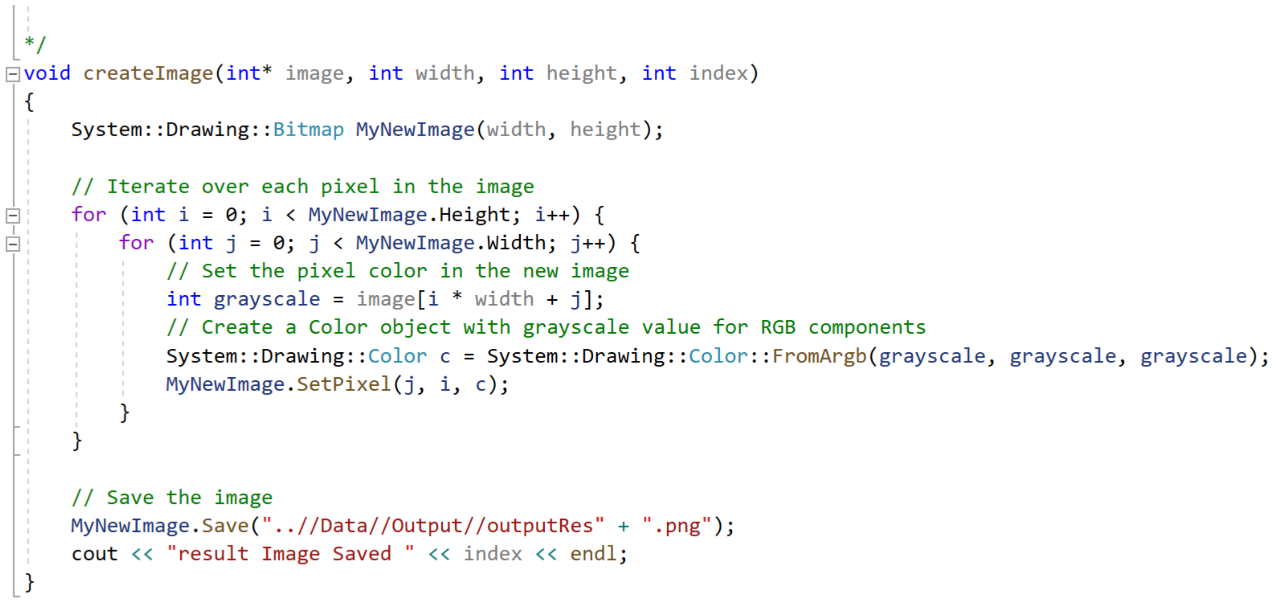
Input Image Function

The function only takes each image path and process on the image to make the input in form of pixels to deal with , also it calculates the greyscale value by averaging the rgb pixels.



```cpp
int* inputImage(int* w, int* h, System::String^ imagePath) //put the size of image in w & h
{
    int OriginalImageWidth, OriginalImageHeight;

    //**Read Image and save it to local arrays**
    //Read Image and save it to local arrays
    System::Drawing::Bitmap BM(imagePath);

    OriginalImageWidth = BM.Width;
    OriginalImageHeight = BM.Height;
    *w = OriginalImageWidth;
    *h = OriginalImageHeight;

    // Allocate memory for input image (grayscale)
    int* input = new int[OriginalImageWidth * OriginalImageHeight];

    // Read the pixel values and convert to grayscale
    for (int i = 0; i < OriginalImageHeight; i++) {
        for (int j = 0; j < OriginalImageWidth; j++) {
            System::Drawing::Color c = BM.GetPixel(j, i);
            // Calculate grayscale value (average of RGB components)
            int grayscale = (c.R + c.G + c.B) / 3;
            // Store the grayscale value in the input array
            input[i * OriginalImageWidth + j] = grayscale;
        }
    }
    return input;
}
```

Create Image function



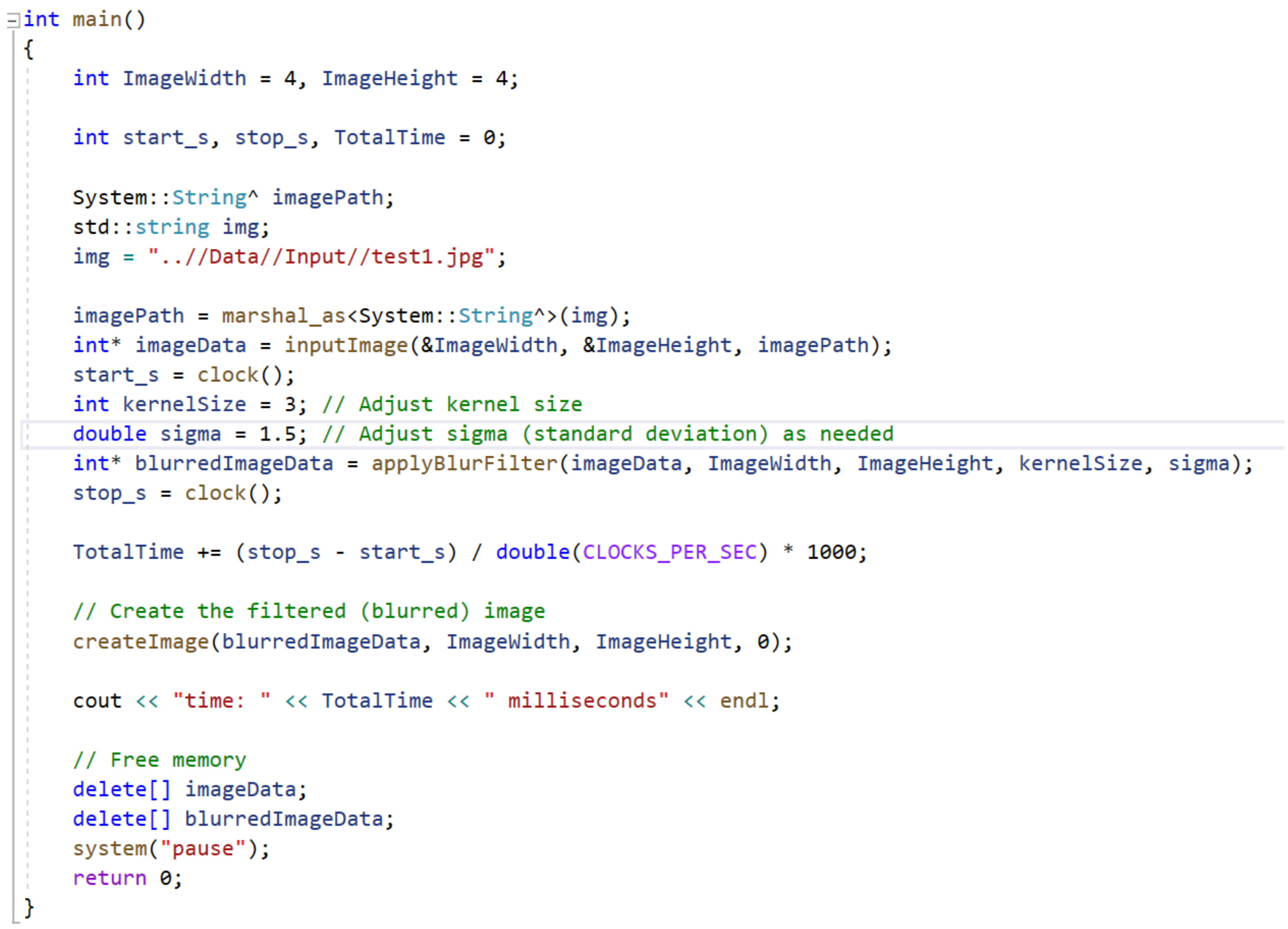
```
*/
void createImage(int* image, int width, int height, int index)
{
    System::Drawing::Bitmap MyNewImage(width, height);

    // Iterate over each pixel in the image
    for (int i = 0; i < MyNewImage.Height; i++) {
        for (int j = 0; j < MyNewImage.Width; j++) {
            // Set the pixel color in the new image
            int grayscale = image[i * width + j];
            // Create a Color object with grayscale value for RGB components
            System::Drawing::Color c = System::Drawing::Color::FromArgb(grayscale, grayscale, grayscale);
            MyNewImage.SetPixel(j, i, c);
        }
    }

    // Save the image
    MyNewImage.Save("..//Data//Output//outputRes" + ".png");
    cout << "result Image Saved " << index << endl;
}
```

- This function creates a new bitmap image using the provided pixel values, width, and height.
- It initializes a System::Drawing::Bitmap object with the specified width and height.
- Iterating over each pixel in the image, it retrieves the grayscale value from the input pixel array.
- Using the grayscale value, it creates a System::Drawing::Color object and sets the corresponding pixel in the new image.
- Finally, the new image is saved to our output directory .

  Note : We change the image format from this function , the last line considering the saving part

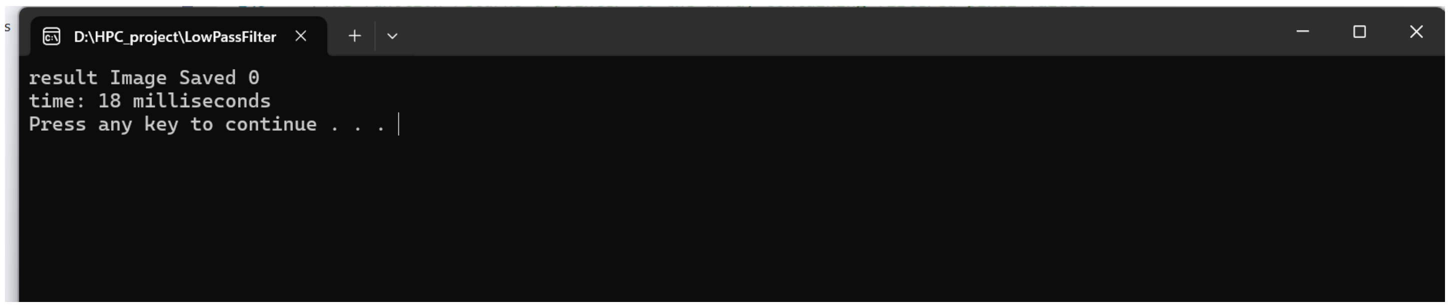
  MyNewImage.Save("..//Data//Output//outputRes" + ".png");

4. Sequential results



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

    int start_s, stop_s, TotalTime = 0;

    System::String^ imagePath;
    std::string img;
    img = "..//Data//Input//test1.jpg";

    imagePath = marshal_as<System::String^>(img);
    int* imageData = inputImage(&ImageWidth, &ImageHeight, imagePath);
    start_s = clock();
    int kernelSize = 3; // Adjust kernel size
    double sigma = 1.5; // Adjust sigma (standard deviation) as needed
    int* blurredImageData = applyBlurFilter(imageData, ImageWidth, ImageHeight, kernelSize, sigma);
    stop_s = clock();

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

    // Create the filtered (blurred) image
    createImage(blurredImageData, ImageWidth, ImageHeight, 0);

    cout << "time: " << TotalTime << " milliseconds" << endl;

    // Free memory
    delete[] imageData;
    delete[] blurredImageData;
    system("pause");
    return 0;
}
```

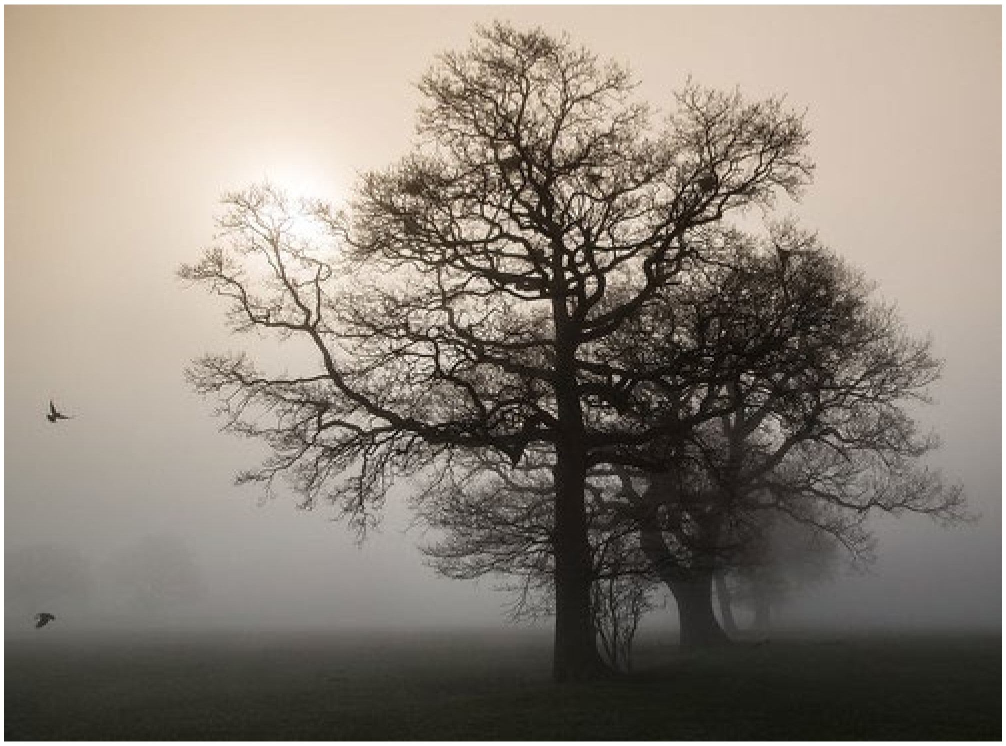
Output



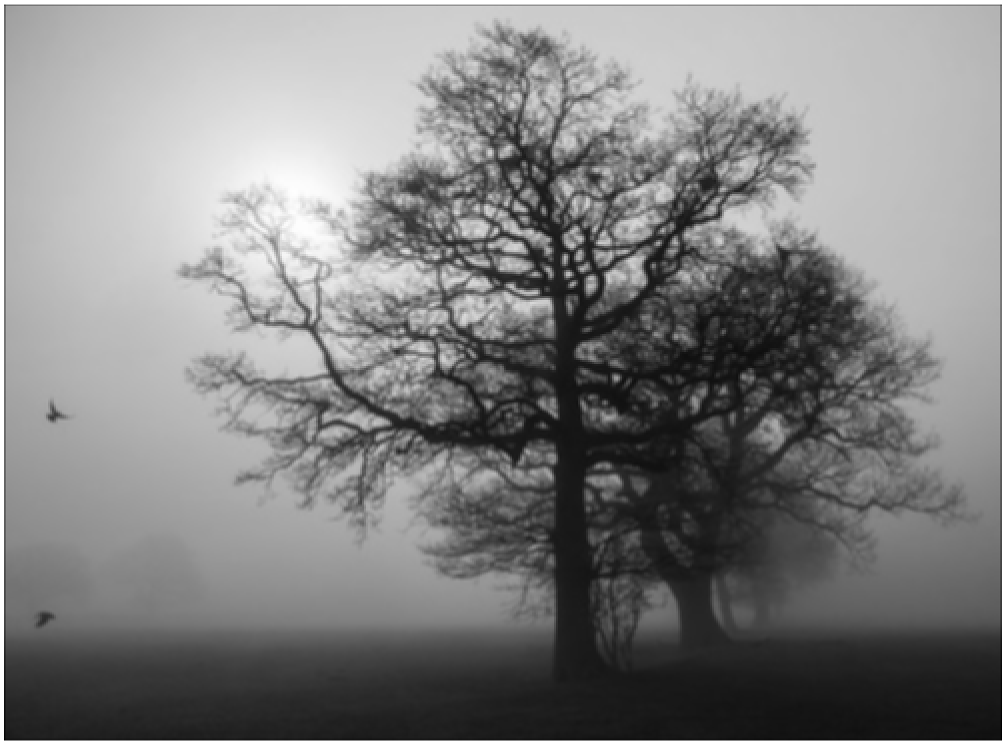
D:\HPC_project\LowPassFilter

```
result Image Saved 0
time: 18 milliseconds
Press any key to continue . . .
```

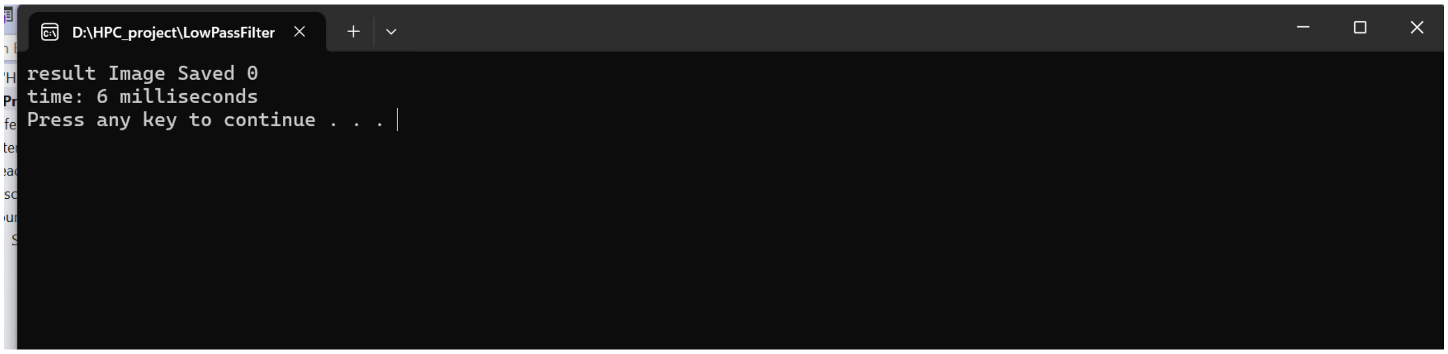
Input1



Output1

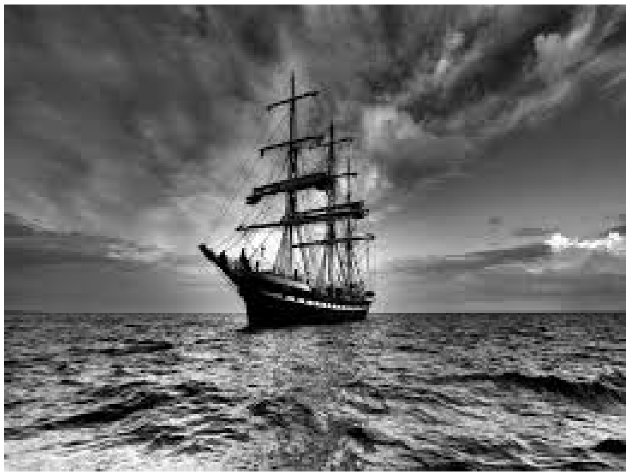


Trying another image

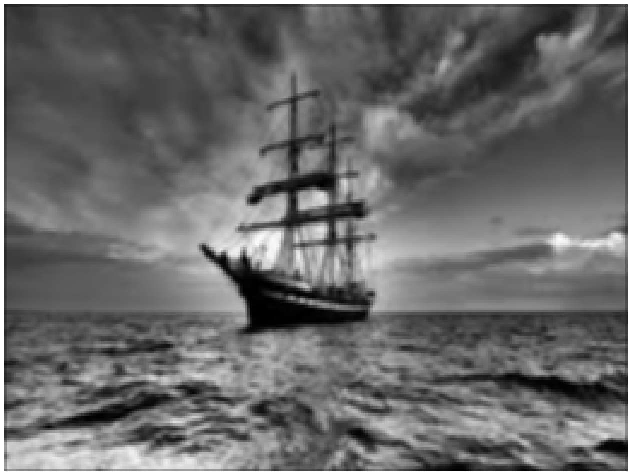


```
result Image Saved 0
time: 6 milliseconds
Press any key to continue . . .
```

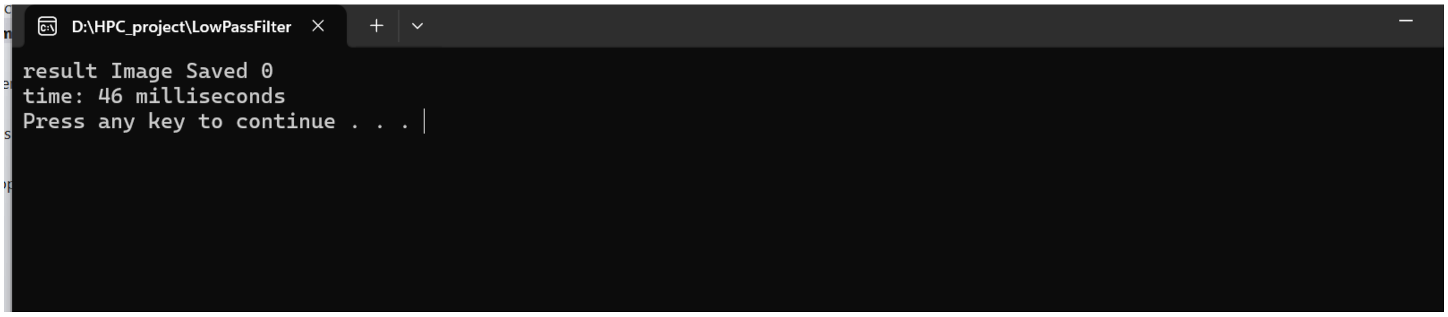
Input2



Output2

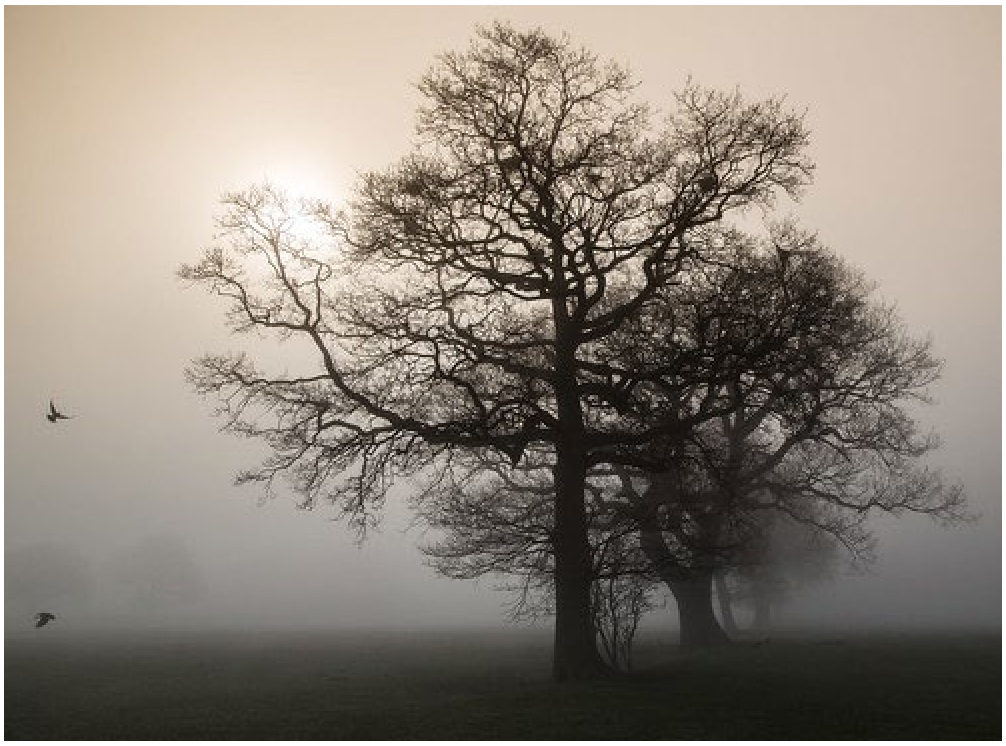


Changing the kernel to b 5*5 and sigma to be 2

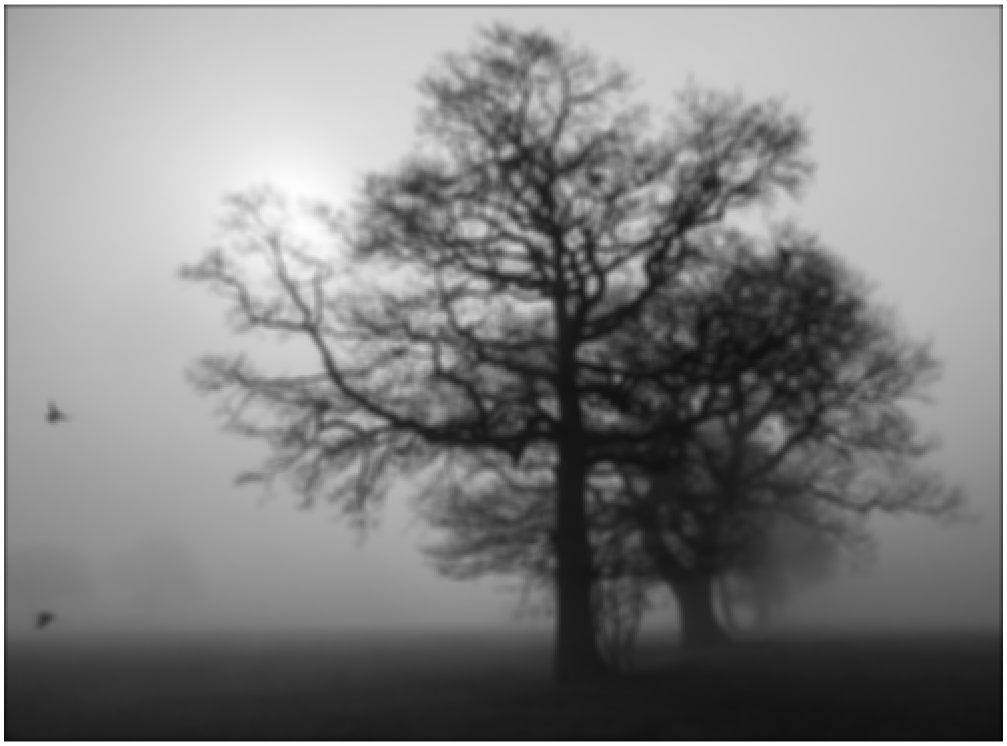


```
result Image Saved 0
time: 46 milliseconds
Press any key to continue . . .
```

On the same image

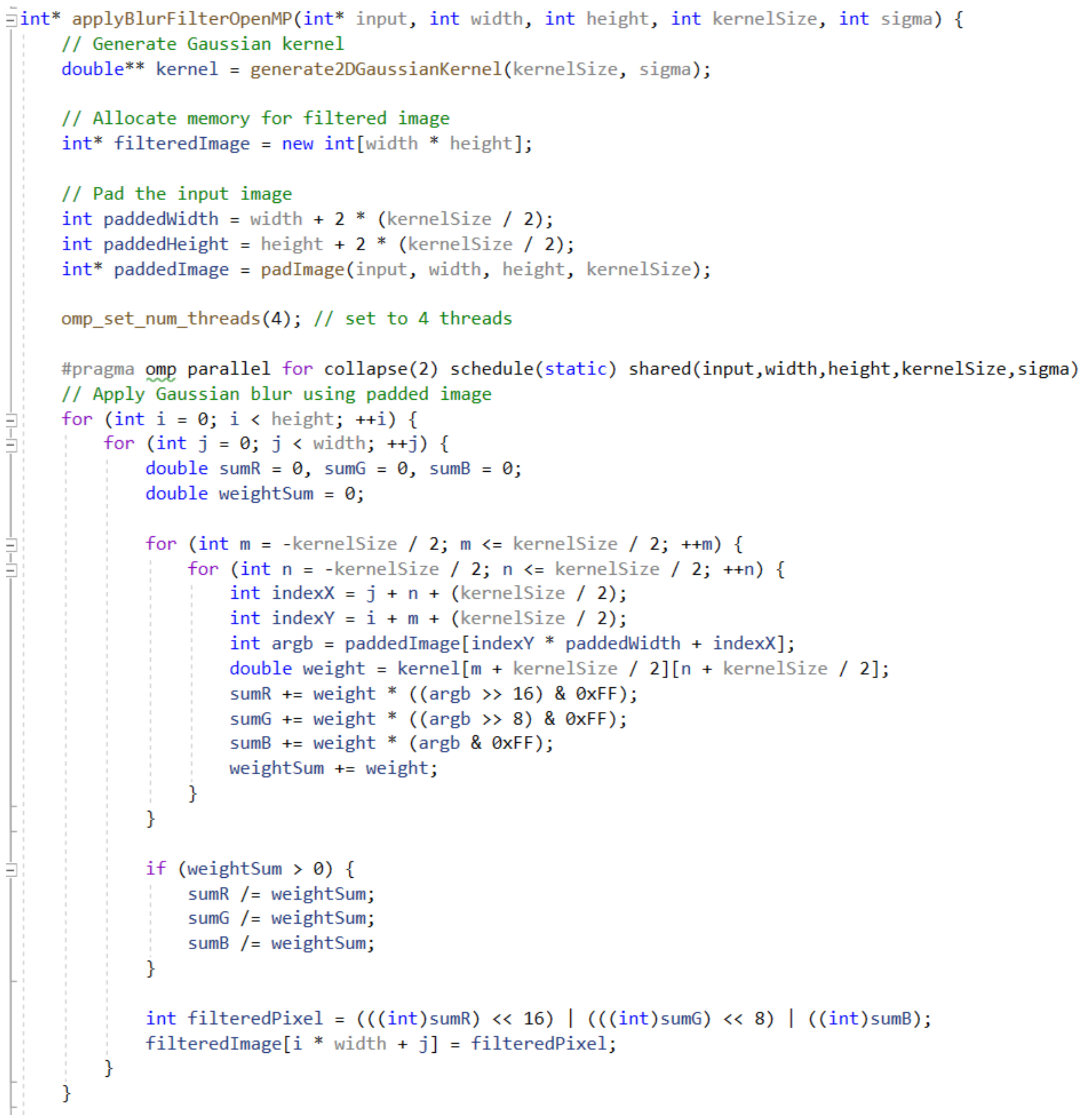


Output image is much blurred than before



Part 2: Parallel using openmp

we will use the same function above but adjusting in it



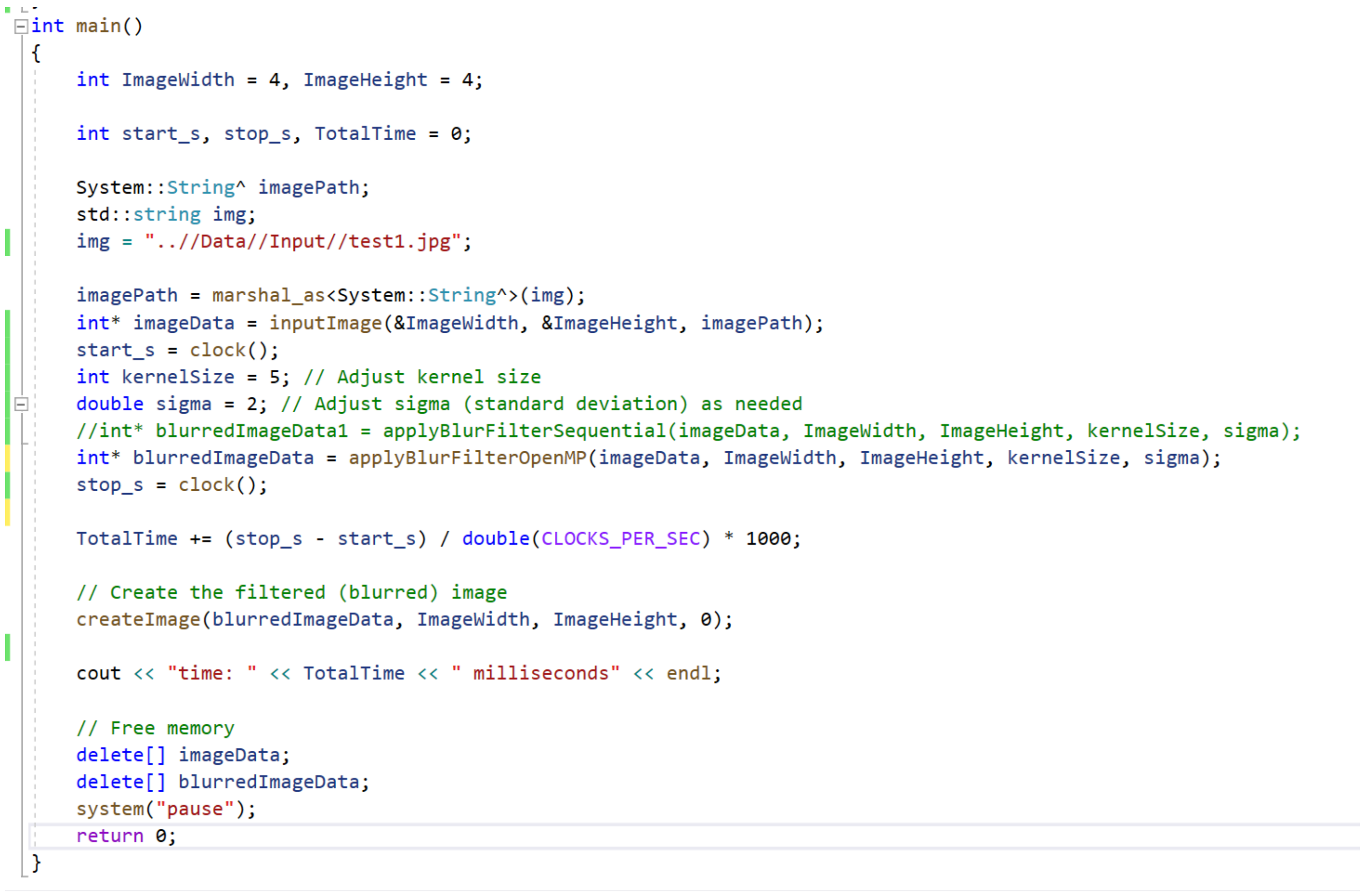
```
int* applyBlurFilterOpenMP(int* input, int width, int height, int kernelSize, int sigma) {
    // Generate Gaussian kernel
    double** kernel = generate2DGaussianKernel(kernelSize, sigma);

    // Allocate memory for filtered image
    int* filteredImage = new int[width * height];

    // Pad the input image
    int paddedWidth = width + 2 * (kernelSize / 2);
    int paddedHeight = height + 2 * (kernelSize / 2);
    int* paddedImage = padImage(input, width, height, kernelSize);

    omp_set_num_threads(4); // set to 4 threads

    #pragma omp parallel for collapse(2) schedule(static) shared(input,width,height,kernelSize,sigma)
    // Apply Gaussian blur using padded image
    for (int i = 0; i < height; ++i) {
        for (int j = 0; j < width; ++j) {
            double sumR = 0, sumG = 0, sumB = 0;
            double weightSum = 0;

            for (int m = -kernelSize / 2; m <= kernelSize / 2; ++m) {
                for (int n = -kernelSize / 2; n <= kernelSize / 2; ++n) {
                    int indexX = j + n + (kernelSize / 2);
                    int indexY = i + m + (kernelSize / 2);
                    int argb = paddedImage[indexY * paddedWidth + indexX];
                    double weight = kernel[m + kernelSize / 2][n + kernelSize / 2];
                    sumR += weight * ((argb >> 16) & 0xFF);
                    sumG += weight * ((argb >> 8) & 0xFF);
                    sumB += weight * (argb & 0xFF);
                    weightSum += weight;
                }
            }

            if (weightSum > 0) {
                sumR /= weightSum;
                sumG /= weightSum;
                sumB /= weightSum;
            }

            int filteredPixel = (((int)sumR) << 16) | (((int)sumG) << 8) | ((int)sumB);
            filteredImage[i * width + j] = filteredPixel;
        }
    }
```

How parallelism using openMp is done?

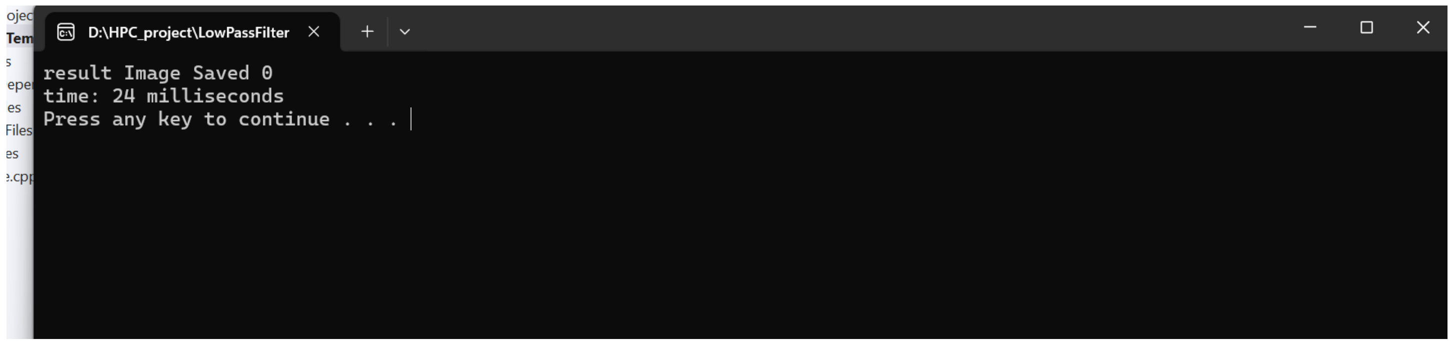
1. **Specifying the Number of Threads**: OpenMP provides a mechanism to specify the number of threads to be used for parallel execution. You can set the number of threads using **omp_set_num_threads() function**.
2. **Parallel Region**: The parallel region is delineated by the **#pragma omp parallel for** directive. This directive tells the compiler to distribute the iterations of the following loop among multiple threads.
3. **Collapse Directive**: The **collapse(2)** clause collapses the nested loops into a single loop, allowing for parallel execution of both loops.
4. Single Loop: After collapsing, there is a single loop that iterates over the 2D grid of the image pixels. Each iteration of this single loop represents a unique combination of row and column indices in the original nested loops.
5. **Static Schedule**: The **schedule(static)** clause specifies that the iterations should be divided into chunks of equal size and distributed statically among the threads.

Parallelism: the pragma omp parallel directive with the collapsed loop instructs the OpenMP runtime to distribute the iterations of the collapsed loop among multiple threads for parallel execution. Each thread will be assigned a subset of iterations to process concurrently.

Test Cases



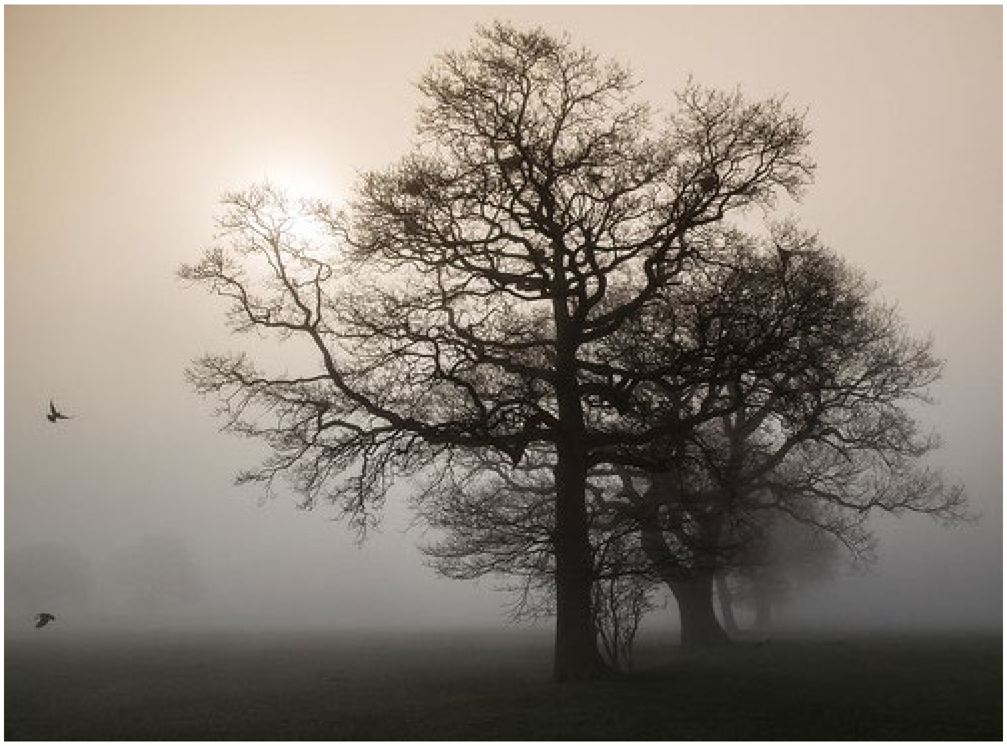
```
int main()
{
    int ImageWidth = 4, ImageHeight = 4;

    int start_s, stop_s, TotalTime = 0;

    System::String^ imagePath;
    std::string img;
    img = "..//Data//Input//test1.jpg";

    imagePath = marshal_as<System::String^>(img);
    int* imageData = inputImage(&ImageWidth, &ImageHeight, imagePath);
    start_s = clock();
    int kernelSize = 5; // Adjust kernel size
    double sigma = 2; // Adjust sigma (standard deviation) as needed
    //int* blurredImageData1 = applyBlurFilterSequential(imageData, ImageWidth, ImageHeight, kernelSize, sigma);
    int* blurredImageData = applyBlurFilterOpenMP(imageData, ImageWidth, ImageHeight, kernelSize, sigma);
    stop_s = clock();

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

    // Create the filtered (blurred) image
    createImage(blurredImageData, ImageWidth, ImageHeight, 0);

    cout << "time: " << TotalTime << " milliseconds" << endl;

    // Free memory
    delete[] imageData;
    delete[] blurredImageData;
    system("pause");
    return 0;
}
```



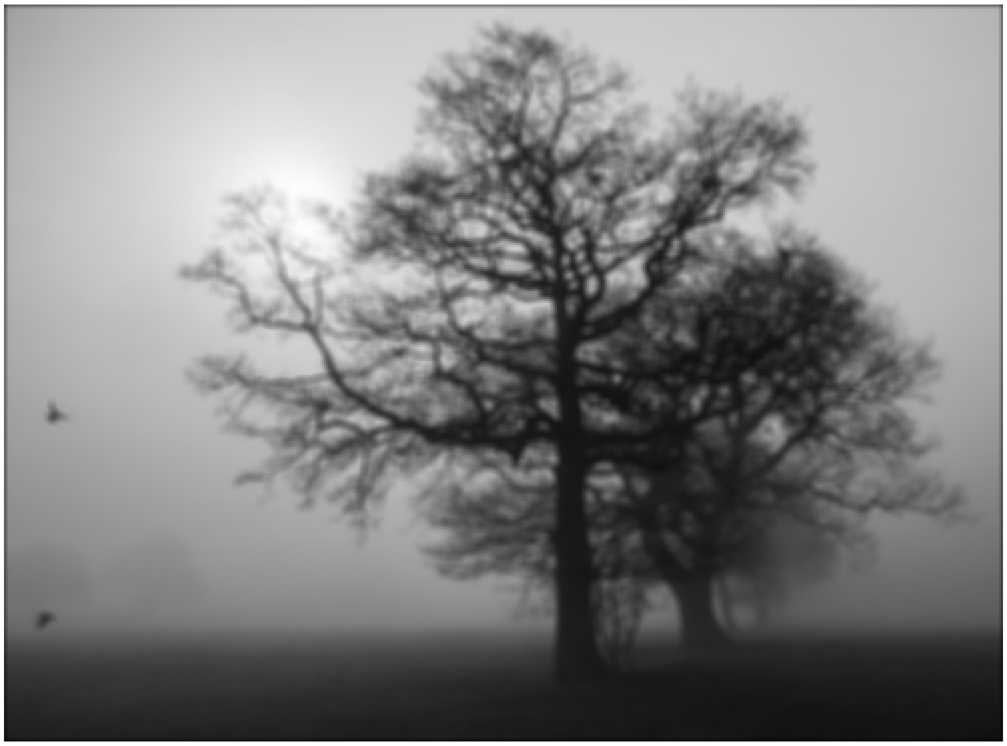
```
result Image Saved 0
time: 24 milliseconds
Press any key to continue . . .
```

1. Test case 1

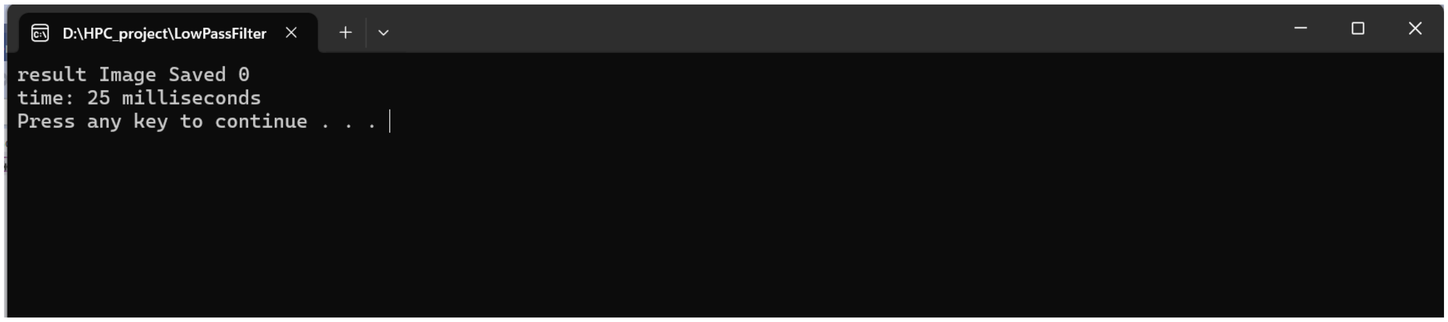
Input1



Output1

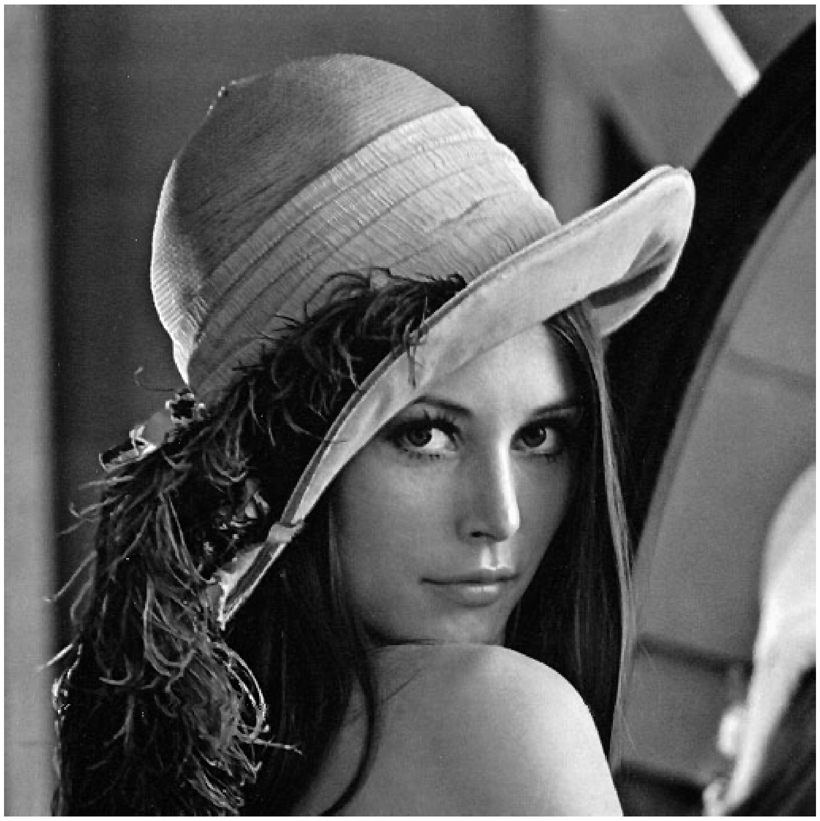


Test Case 2 : another input but .png image

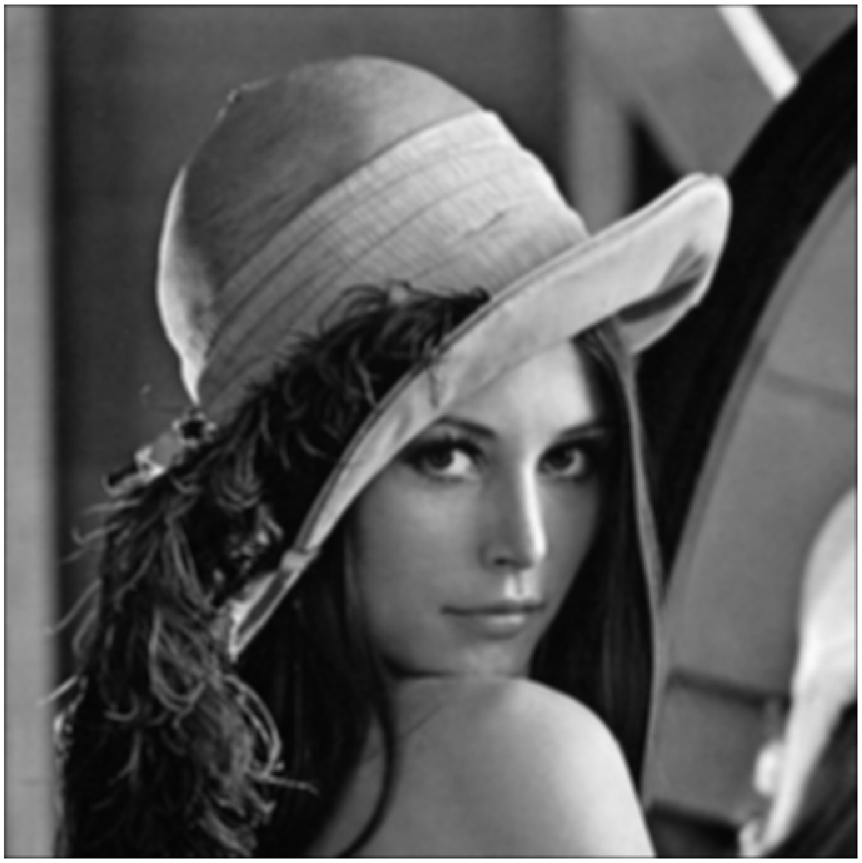


```
result Image Saved 0
time: 25 milliseconds
Press any key to continue . . .
```

Input2

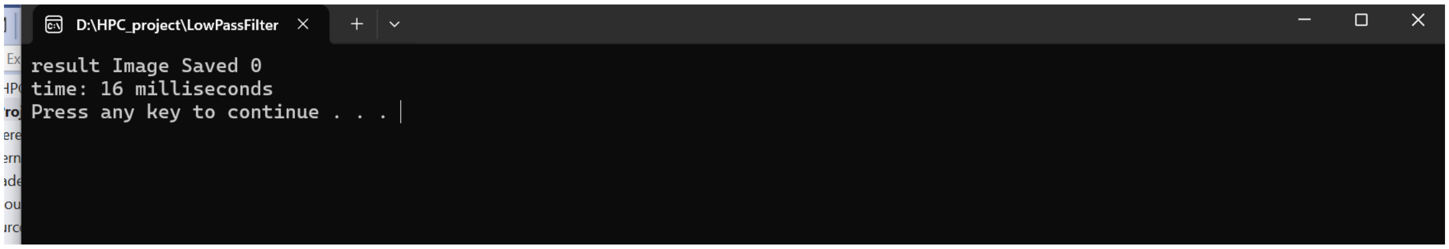


Output2



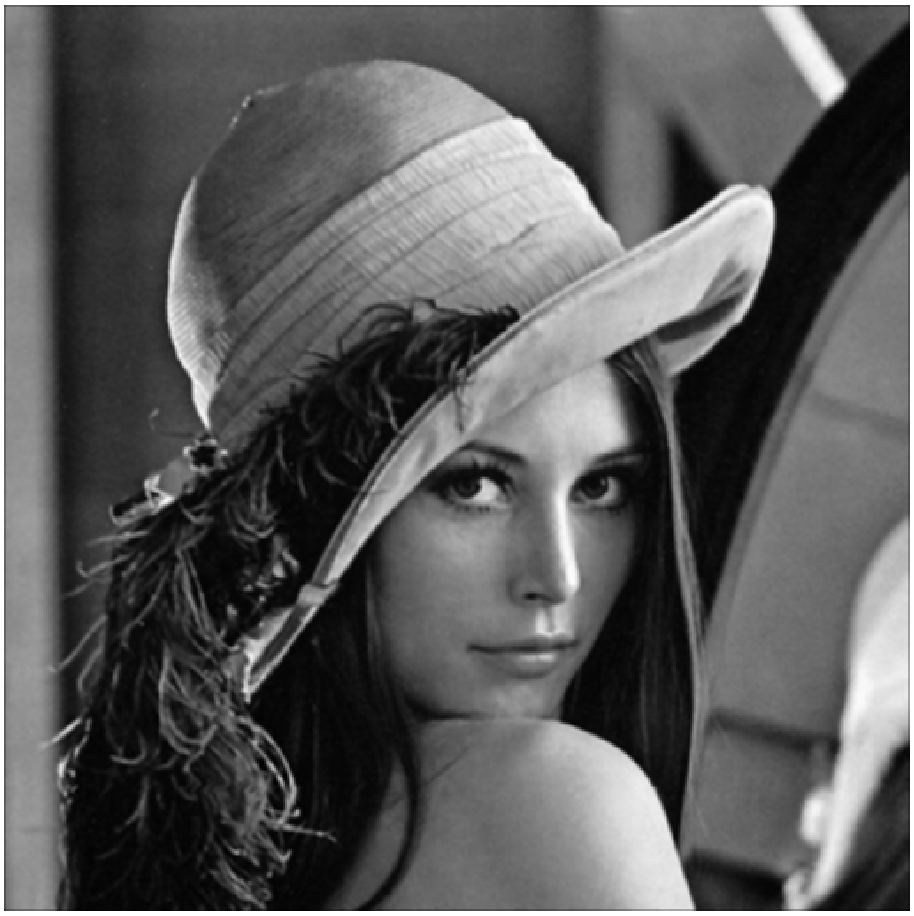
Test Case 3

Trying to increase the number of threads to be 8 and make kernel 3*3



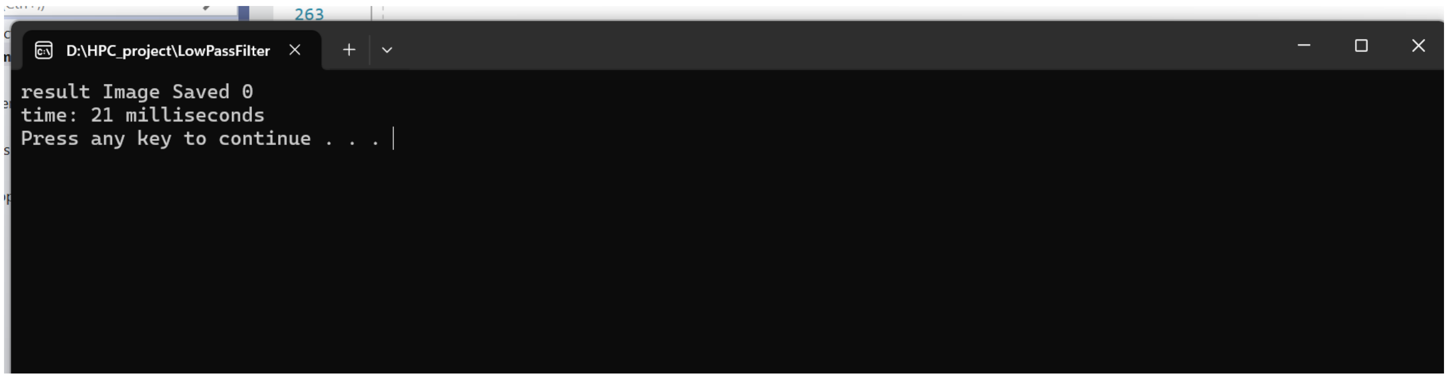
```
result Image Saved 0
time: 16 milliseconds
Press any key to continue . . .
```

Output



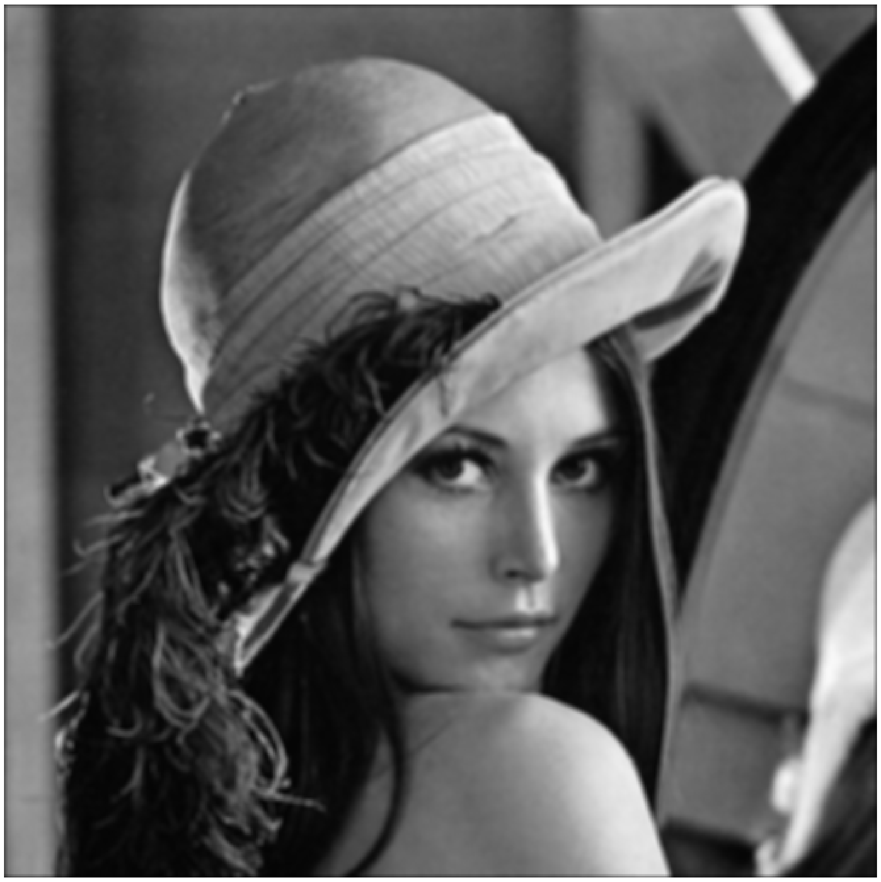
Test Case 4

Trying to make threads 8 and kernel 5*5 , sigma =2

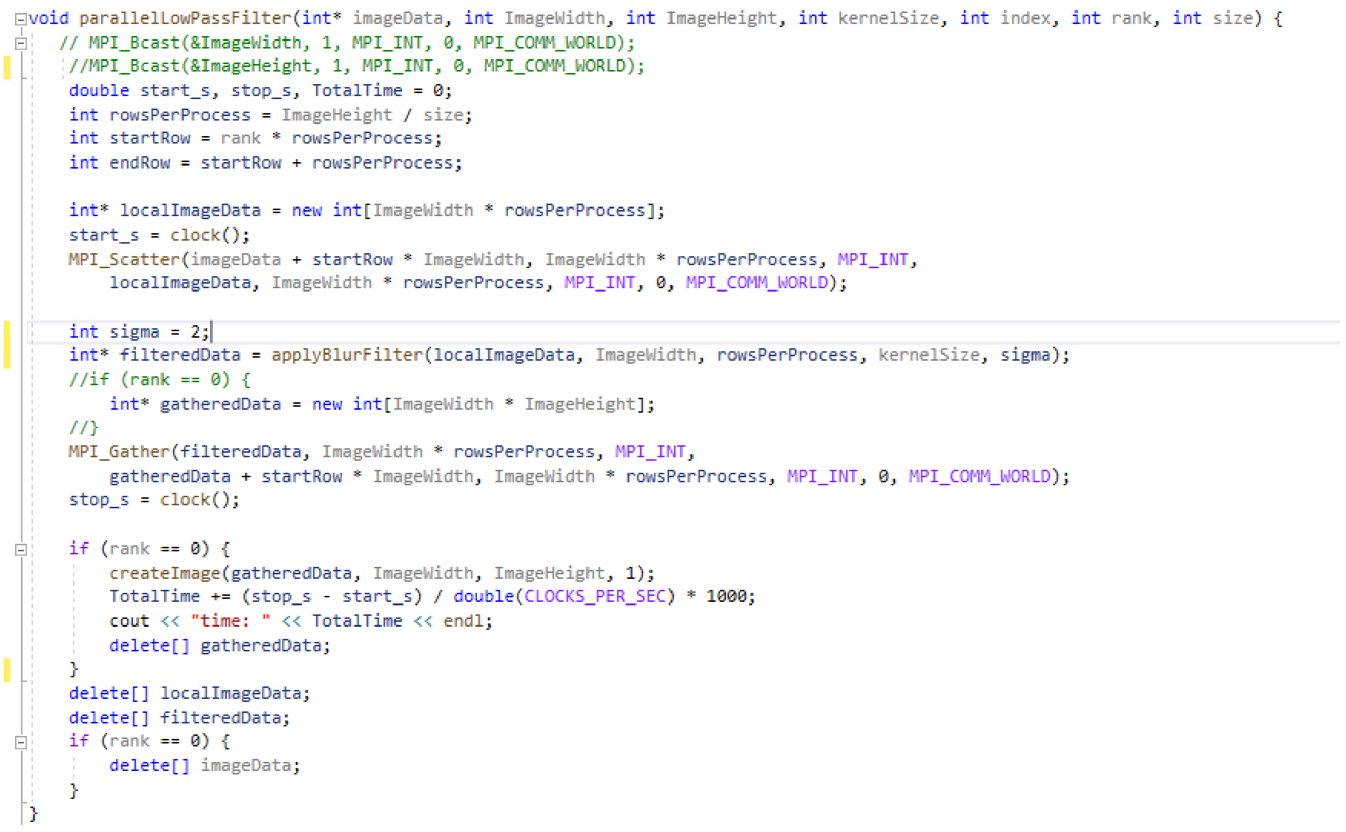


```
result Image Saved 0
time: 21 milliseconds
Press any key to continue . . .
```

Output

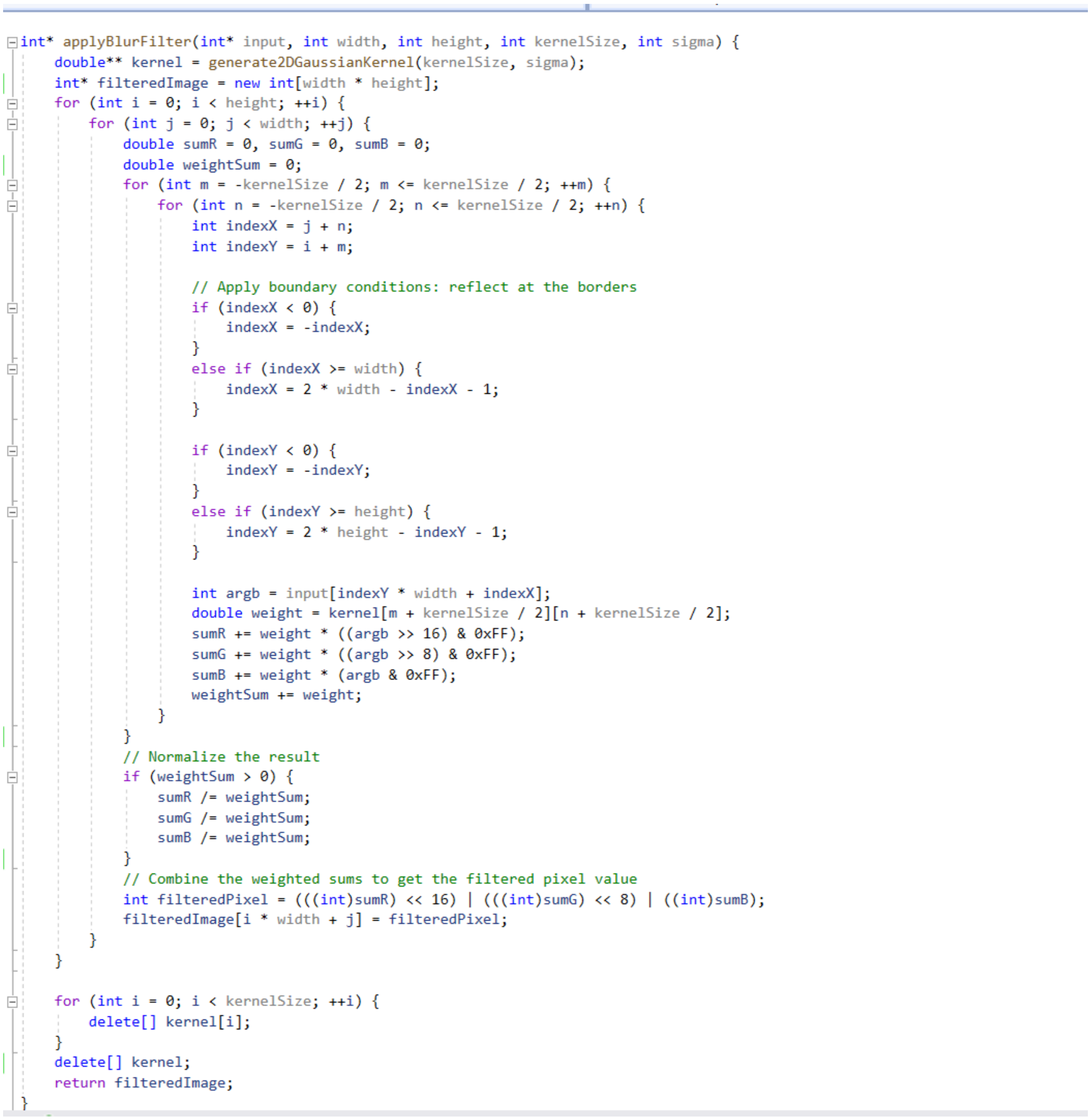


Part 3: Parallel using MPI:



```cpp
void parallelLowPassFilter(int* imageData, int ImageWidth, int ImageHeight, int kernelSize, int index, int rank, int size) {
   // MPI_Bcast(&ImageWidth, 1, MPI_INT, 0, MPI_COMM_WORLD);
    //MPI_Bcast(&ImageHeight, 1, MPI_INT, 0, MPI_COMM_WORLD);
    double start_s, stop_s, TotalTime = 0;
    int rowsPerProcess = ImageHeight / size;
    int startRow = rank * rowsPerProcess;
    int endRow = startRow + rowsPerProcess;

    int* localImageData = new int[ImageWidth * rowsPerProcess];
    start_s = clock();
    MPI_Scatter(imageData + startRow * ImageWidth, ImageWidth * rowsPerProcess, MPI_INT,
        localImageData, ImageWidth * rowsPerProcess, MPI_INT, 0, MPI_COMM_WORLD);

    int sigma = 2;
    int* filteredData = applyBlurFilter(localImageData, ImageWidth, rowsPerProcess, kernelSize, sigma);
    //if (rank == 0) {
        int* gatheredData = new int[ImageWidth * ImageHeight];
    //}
    MPI_Gather(filteredData, ImageWidth * rowsPerProcess, MPI_INT,
        gatheredData + startRow * ImageWidth, ImageWidth * rowsPerProcess, MPI_INT, 0, MPI_COMM_WORLD);
    stop_s = clock();

    if (rank == 0) {
        createImage(gatheredData, ImageWidth, ImageHeight, 1);
        TotalTime += (stop_s - start_s) / double(CLOCKS_PER_SEC) * 1000;
        cout << "time: " << TotalTime << endl;
        delete[] gatheredData;
    }
    delete[] localImageData;
    delete[] filteredData;
    if (rank == 0) {
        delete[] imageData;
    }
}
```

The applyBlurFilter function



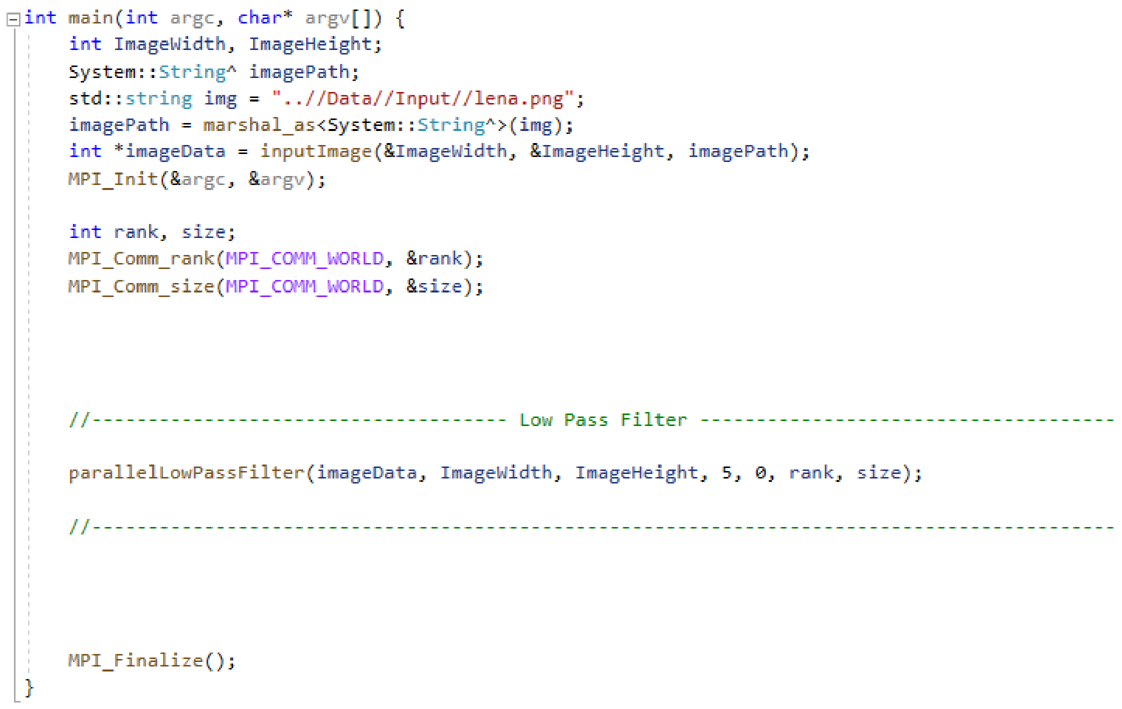
```
int* applyBlurFilter(int* input, int width, int height, int kernelSize, int sigma) {
    double** kernel = generate2DGaussianKernel(kernelSize, sigma);
    int* filteredImage = new int[width * height];
    for (int i = 0; i < height; ++i) {
        for (int j = 0; j < width; ++j) {
            double sumR = 0, sumG = 0, sumB = 0;
            double weightSum = 0;
            for (int m = -kernelSize / 2; m <= kernelSize / 2; ++m) {
                for (int n = -kernelSize / 2; n <= kernelSize / 2; ++n) {
                    int indexX = j + n;
                    int indexY = i + m;

                    // Apply boundary conditions: reflect at the borders
                    if (indexX < 0) {
                        indexX = -indexX;
                    }
                    else if (indexX >= width) {
                        indexX = 2 * width - indexX - 1;
                    }

                    if (indexY < 0) {
                        indexY = -indexY;
                    }
                    else if (indexY >= height) {
                        indexY = 2 * height - indexY - 1;
                    }

                    int argb = input[indexY * width + indexX];
                    double weight = kernel[m + kernelSize / 2][n + kernelSize / 2];
                    sumR += weight * ((argb >> 16) & 0xFF);
                    sumG += weight * ((argb >> 8) & 0xFF);
                    sumB += weight * (argb & 0xFF);
                    weightSum += weight;
                }
            }
            // Normalize the result
            if (weightSum > 0) {
                sumR /= weightSum;
                sumG /= weightSum;
                sumB /= weightSum;
            }
            // Combine the weighted sums to get the filtered pixel value
            int filteredPixel = (((int)sumR) << 16) | (((int)sumG) << 8) | ((int)sumB);
            filteredImage[i * width + j] = filteredPixel;
        }
    }

    for (int i = 0; i < kernelSize; ++i) {
        delete[] kernel[i];
    }
    delete[] kernel;
    return filteredImage;
}
```

How to parallelize the program using MPI?

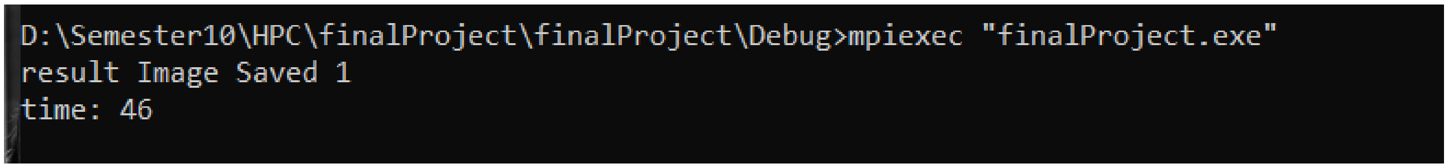
- **MPI Initialization and Setup**: Prior to calling this function, MPI should be initiated using MPI_Init, and MPI_Comm_rank and MPI_Comm_size should be used to determine the rank and size of the MPI communicator and they are defined in main. The current process's rank is the rank parameter; the total number of processes is the size parameter.
- **Memory Allocation**: To save the area of the image that each process will operate on, the function allots memory for a local image data buffer (localImageData). Additionally, RAM is allotted for a filtered data buffer (filteredData), which will hold the blur filter's output.
- **Scattering**: A piece of the picture data is sent to each MPI process via the MPI_Scatter function. A part of the image data, beginning at startRow and ending at endRow, is sent to each process.
- **Blur Filter**: To apply a blur filter to the section of the image data kept in localImageData, call the applyBlurFilter function. The blur effect's strength is adjusted by the sigma parameter.
- **Collecting Filtered Data**: The MPI_Gather function is used to gather the filtered data back to the root process (rank == 0) following the application of the blur filter. The gatheredData buffer contains the collected data.
- **Producing the Output Image**: The createImage method is used to make an output image from the collected data if the running process is the root process (rank == 0). It is presumed that this function creates an output picture file by using the input data, image width, image height, and output file format.
- **Deallocation of Memory**: If the running process is the root process, then the function deallocates memory for the filtered data buffer (filteredData), the local image data buffer (localImageData), and the image data buffer (imageData).
- **Finalization**: MPI needs to be finalized using MPI_Finalize after using this method in main.

Test Cases:

Testing with kernel size=5 and sigma =2:



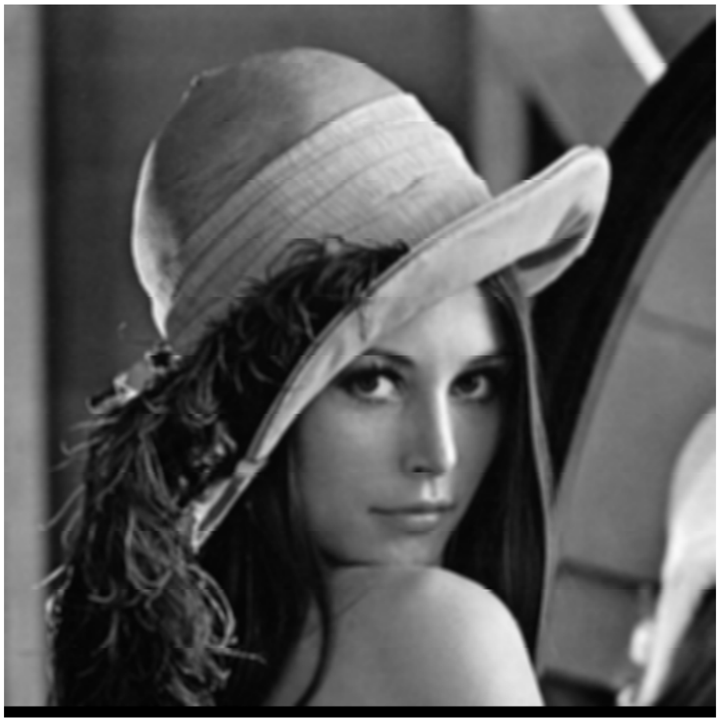
```
int main(int argc, char* argv[]) {
    int ImageWidth, ImageHeight;
    System::String^ imagePath;
    std::string img = "..//Data//Input//lena.png";
    imagePath = marshal_as<System::String^>(img);
    int *imageData = inputImage(&ImageWidth, &ImageHeight, imagePath);
    MPI_Init(&argc, &argv);

    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);



    //------------------------------------- Low Pass Filter -------------------------------------

    parallelLowPassFilter(imageData, ImageWidth, ImageHeight, 5, 0, rank, size);

    //---------------------------------------------------------------------------------------------



    MPI_Finalize();
}
```



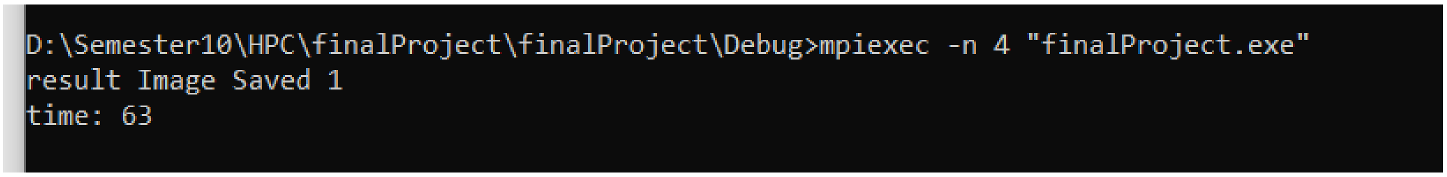
```
D:\Semester10\HPC\finalProject\finalProject\Debug>mpiexec "finalProject.exe"
result Image Saved 1
time: 46
```

Input:

Output:



When I tried to execute using only 4 processors the time increased and the output was almost the same:



```
D:\Semester10\HPC\finalProject\finalProject\Debug>mpiexec -n 4 "finalProject.exe"
result Image Saved 1
time: 63
```

Output:

Conclusion:

In this research, we investigated parallel low pass image filtering implementations using MPI, OpenMP, and sequential techniques.

Remarkably, in terms of processing time, the sequential approach fared better than the MPI and OpenMP implementations. It took 46 milliseconds for the sequential implementation, 21 milliseconds for the OpenMP implementation, and 46 milliseconds for the MPI implementation.

After utilizing OpenMP to parallelize the filtering process and take use of shared memory parallelism, we compared it to the sequential approach as a baseline.

We further optimized for distributed memory systems by dividing the filtering process among several nodes using MPI.

The project offered insightful information on parallel programming methods and how to use them for image processing applications. Subsequent research endeavors may concentrate on refining the parallel implementations to utilize the parallel computing capacities of contemporary hardware more effectively.

Kernel	Sigma	Sequential	OpenMP	MPI
5	2	46 ms	21 ms	46 ms
3	1.5	18 ms	16 ms	27 ms

GitHub link

https://github.com/Shaimaa-moh/LowPassFilter