Ain Shams University
Faculty of Engineering
Discipline Programs

High Performance Computing

Major Task Report

Computer Engineering and Software Systems (CESS)

Submitted to:

Dr. Karim Emara

Dr. Tamer Mostafa

Eng. Malak Mohamed

Eng. Mostafa Ashraf

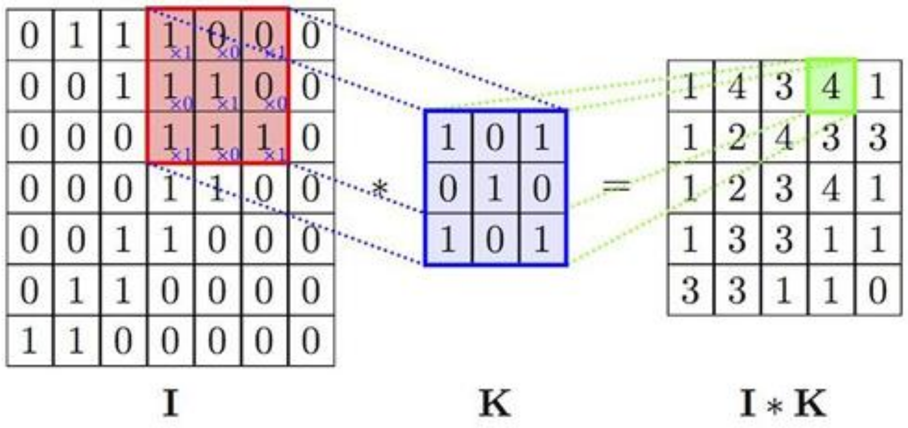
Submitted by Team 6:

Habiba Yasser AbdelHalim	20P3072
Nadine Hisham Hassan	20P9880
Tamer Ihab Mohamed	20P5567
Ziad Mahmoud Gouda	20P2765

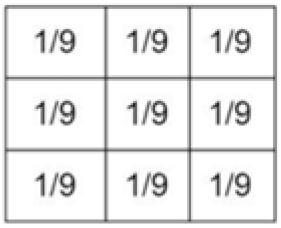
Table of Contents

1.0 Project Description

Parallel low pass filter 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.



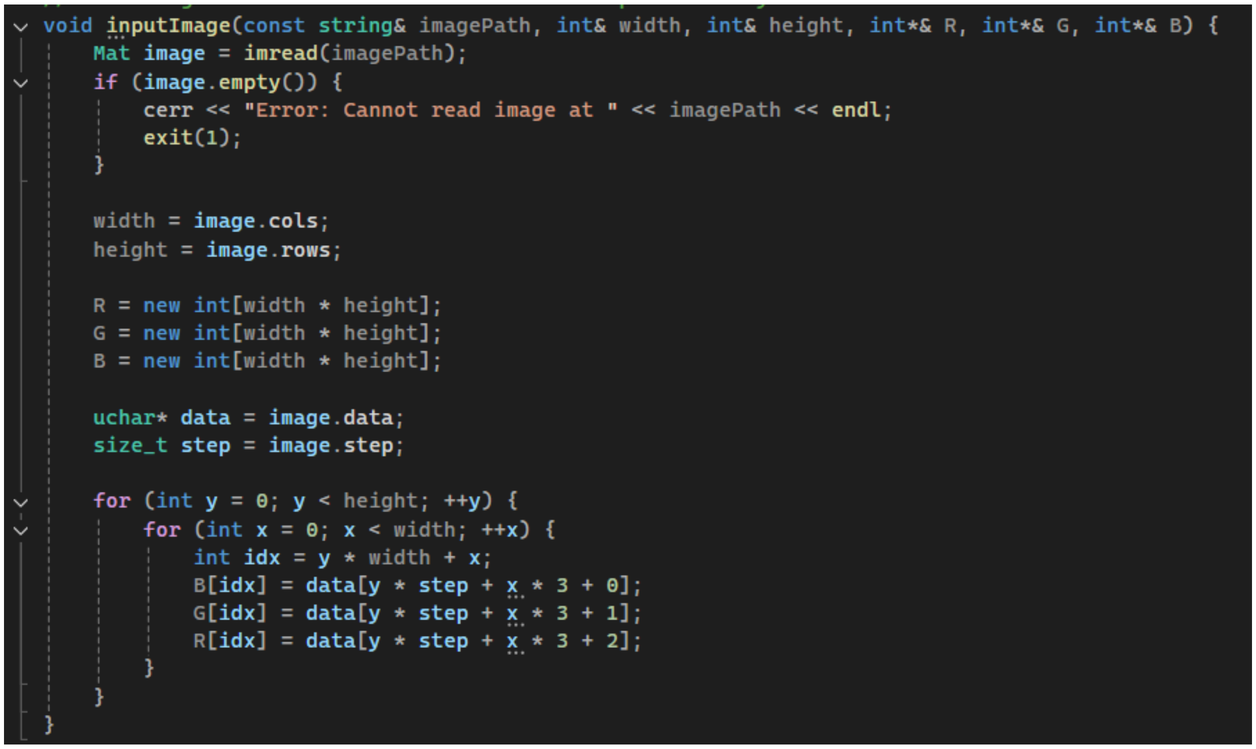
We assume Image is I, the kernel is K. If we applied the filter on the red region at I, the result will be computed by aligning the kernel onto the image part then doing basic multiplication between the aligned elements as: (1*1 + 0*0 + 0*1) + (0*1 + 1*1 + 0*0) + (1*1 + 1*0 + 1*1) = 4. Every kernel-based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

2.0 Sequential

2.1 inputImage

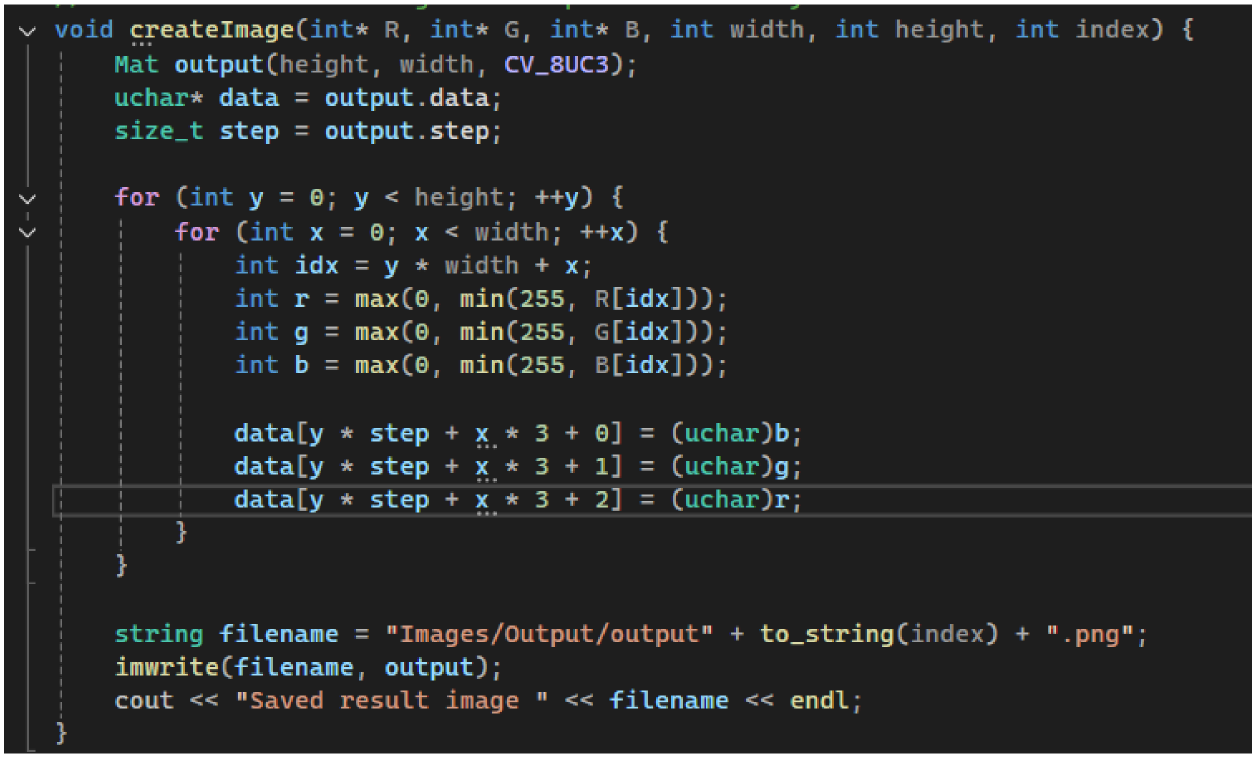


```
void inputImage(const string& imagePath, int& width, int& height, int*& R, int*& G, int*& B) {
    Mat image = imread(imagePath);
    if (image.empty()) {
        cerr << "Error: Cannot read image at " << imagePath << endl;
        exit(1);
    }

    width = image.cols;
    height = image.rows;

    R = new int[width * height];
    G = new int[width * height];
    B = new int[width * height];

    uchar* data = image.data;
    size_t step = image.step;

    for (int y = 0; y < height; ++y) {
        for (int x = 0; x < width; ++x) {
            int idx = y * width + x;
            B[idx] = data[y * step + x * 3 + 0];
            G[idx] = data[y * step + x * 3 + 1];
            R[idx] = data[y * step + x * 3 + 2];
        }
    }
}
```

The inputImage function reads an image from a given file path and extracts its red, green, and blue (RGB) components into separate dynamically allocated arrays. It uses OpenCV's imread to load the image and verifies that it was successfully loaded. The image dimensions are stored in the referenced width and height variables.

Then, the function iterates over each pixel and extracts the color channels (assuming the image is in BGR format, as OpenCV uses by default) by accessing the pixel data directly. These channel values are stored in the respective R, G, and B arrays, which are allocated based on the image resolution.

2.2 createImage

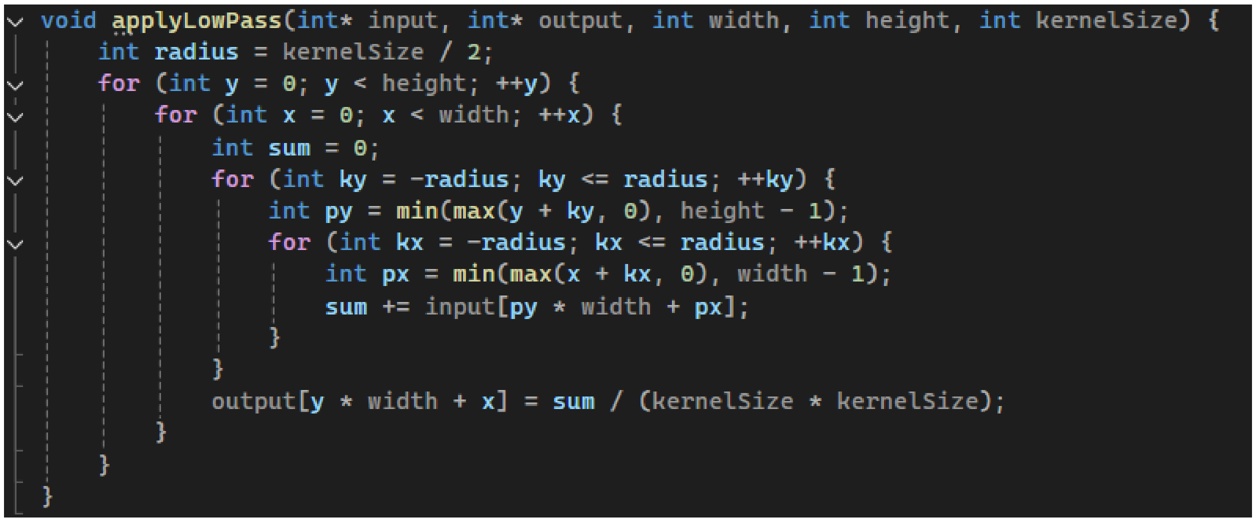


```
void createImage(int* R, int* G, int* B, int width, int height, int index) {
    Mat output(height, width, CV_8UC3);
    uchar* data = output.data;
    size_t step = output.step;

    for (int y = 0; y < height; ++y) {
        for (int x = 0; x < width; ++x) {
            int idx = y * width + x;
            int r = max(0, min(255, R[idx]));
            int g = max(0, min(255, G[idx]));
            int b = max(0, min(255, B[idx]));

            data[y * step + x * 3 + 0] = (uchar)b;
            data[y * step + x * 3 + 1] = (uchar)g;
            data[y * step + x * 3 + 2] = (uchar)r;
        }
    }

    string filename = "Images/Output/output" + to_string(index) + ".png";
    imwrite(filename, output);
    cout << "Saved result image " << filename << endl;
}
```

The createImage function reconstructs and saves an image from the separated RGB channel arrays after processing. It creates an empty Mat of the appropriate size and type, then populates each pixel's color values from the input R, G, and B arrays.

It also clamps each color value to the valid 8-bit range (0–255) using min and max. After assembling the image, it writes the result to a file using imwrite, saving it under the Images/Output/ directory with a sequential file name based on the given index. A confirmation message is printed to indicate the saved file.

2.3 applyLowPass

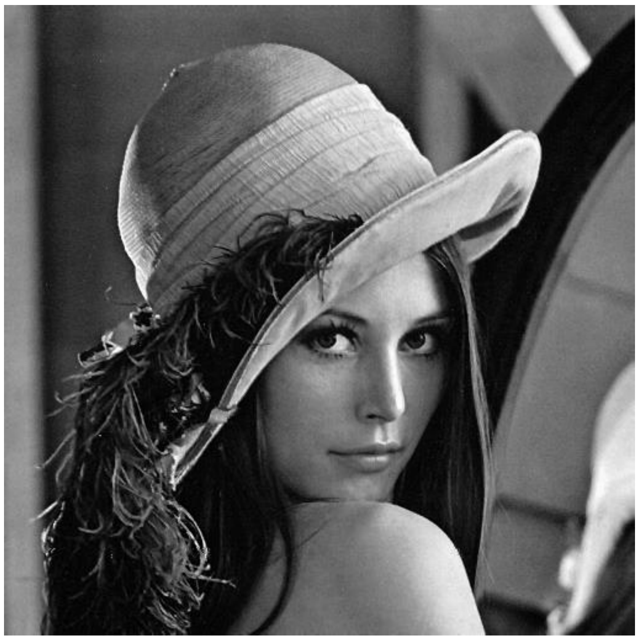


```
void applyLowPass(int* input, int* output, int width, int height, int kernelSize) {
    int radius = kernelSize / 2;
    for (int y = 0; y < height; ++y) {
        for (int x = 0; x < width; ++x) {
            int sum = 0;
            for (int ky = -radius; ky <= radius; ++ky) {
                int py = min(max(y + ky, 0), height - 1);
                for (int kx = -radius; kx <= radius; ++kx) {
                    int px = min(max(x + kx, 0), width - 1);
                    sum += input[py * width + px];
                }
            }
            output[y * width + x] = sum / (kernelSize * kernelSize);
        }
    }
}
```

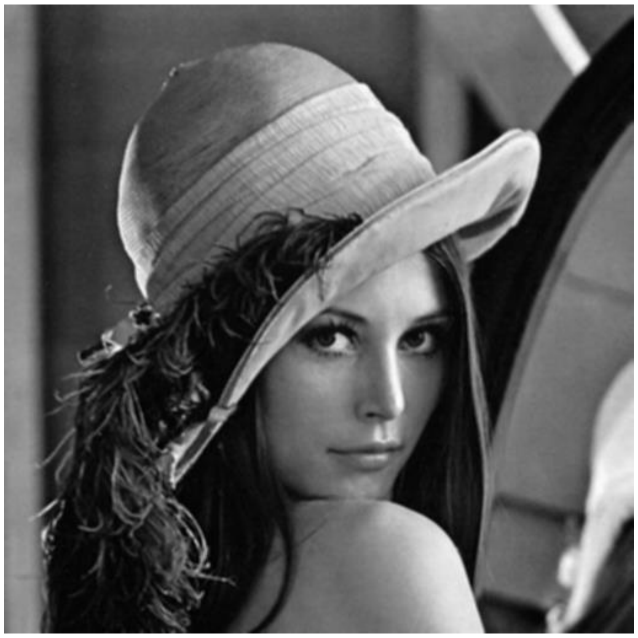
The applyLowPass function applies a basic low-pass (blurring) filter to a single color channel. It uses a square averaging kernel of a specified kernelSize, centered at each pixel. For each pixel, the function computes the average value of its neighboring pixels within the kernel window. To handle edge pixels, the function uses border replication by clamping the indices so that out-of-bounds coordinates are replaced with the closest valid ones. This prevents access violations and maintains image size. The result is stored in the output array.

2.4 Test 1 (3x3 Kernel)

Input:

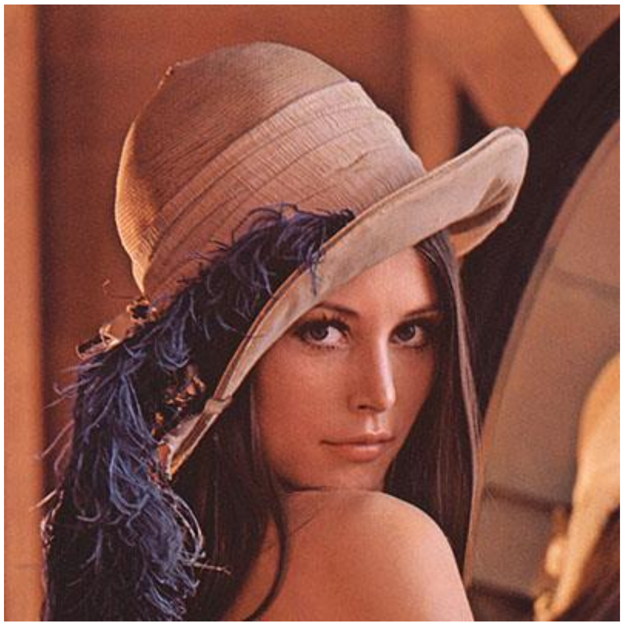


Output:

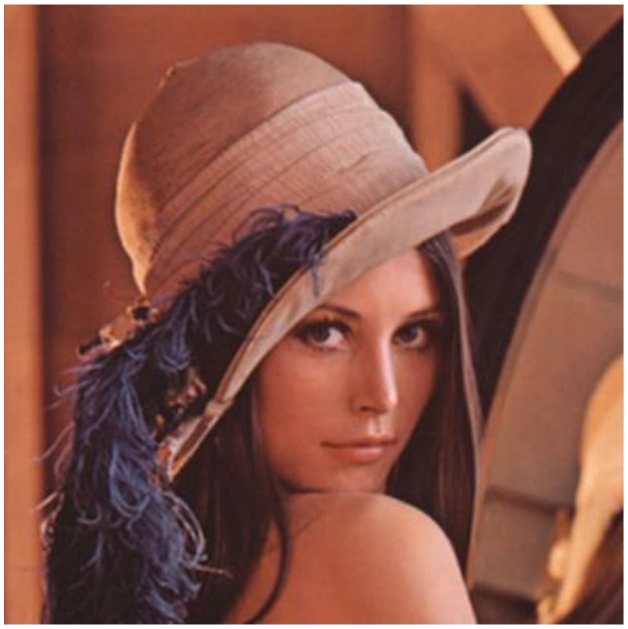


2.5 Test 2 (3x3 Kernel)

Input:



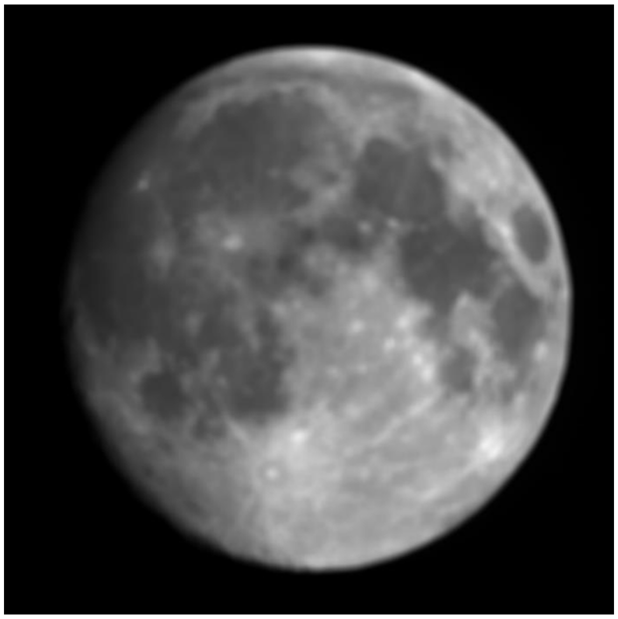
Output:



2.6 Test 3 (11x11 Kernel)

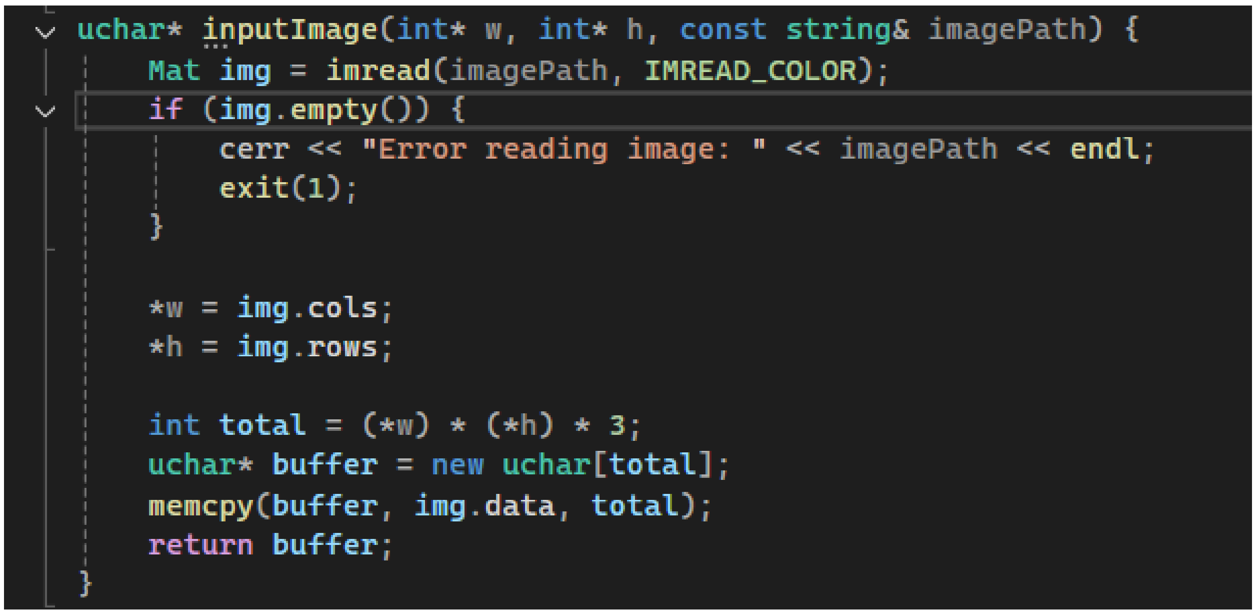
Input:

Output:



3.0 OpenMP

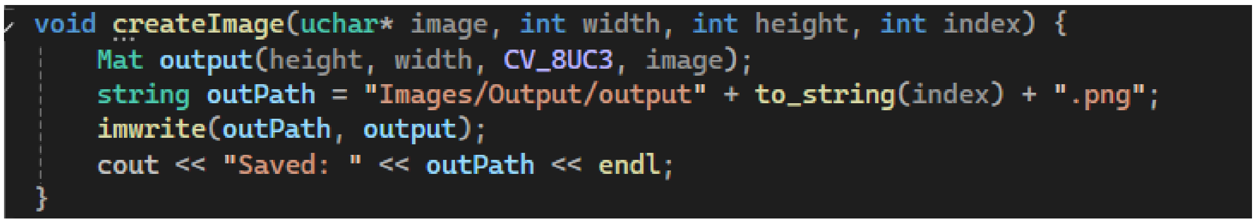
3.1 inputImage



```
uchar* inputImage(int* w, int* h, const string& imagePath) {
    Mat img = imread(imagePath, IMREAD_COLOR);
    if (img.empty()) {
        cerr << "Error reading image: " << imagePath << endl;
        exit(1);
    }

    *w = img.cols;
    *h = img.rows;

    int total = (*w) * (*h) * 3;
    uchar* buffer = new uchar[total];
    memcpy(buffer, img.data, total);
    return buffer;
}
```

The inputImage function reads an image from the given file path using OpenCV's imread function in color mode. It retrieves the image's width and height and returns a dynamically allocated raw BGR (Blue-Green-Red) buffer containing all pixel data. This buffer is a 1D array of uchar (unsigned char) that holds all color channels sequentially. If the image cannot be read, the function prints an error message and exits the program. The caller receives the image dimensions via pointer parameters (w and h), and the buffer is returned for further processing.

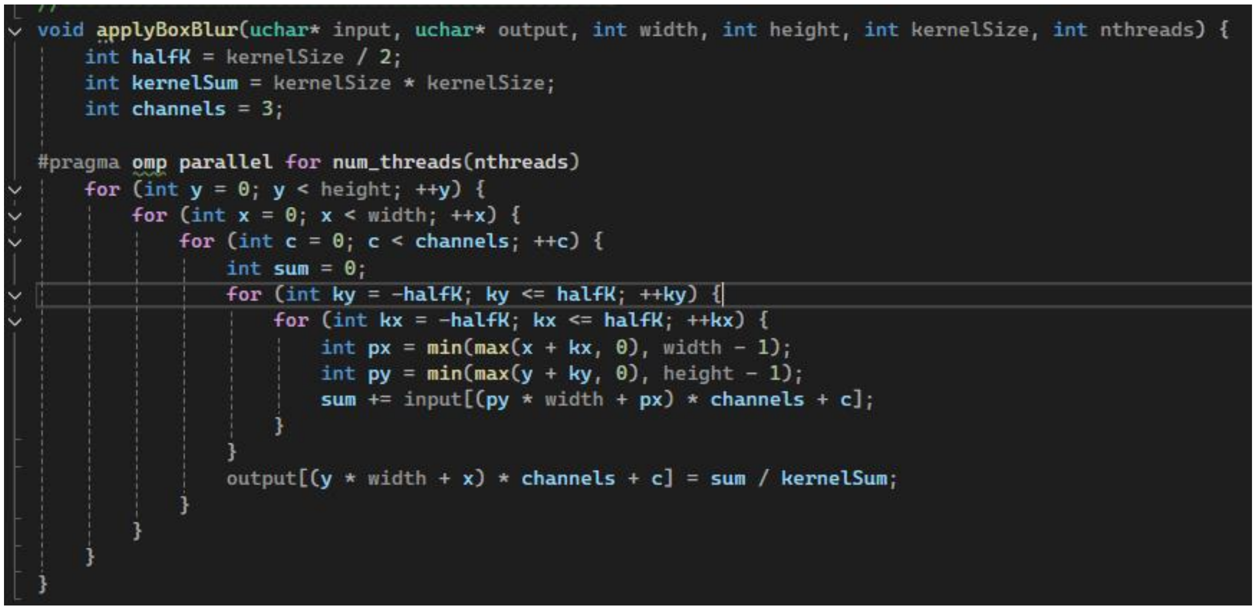
3.2 createImage



```
void createImage(uchar* image, int width, int height, int index) {
    Mat output(height, width, CV_8UC3, image);
    string outPath = "Images/Output/output" + to_string(index) + ".png";
    imwrite(outPath, output);
    cout << "Saved: " << outPath << endl;
}
```

This function saves an image from raw pixel data to a file. It takes four parameters: the raw image data (image), the image dimensions (width and height), and an index (index) to generate a unique output filename. The function creates a Mat object from the raw data, specifying the image format as CV_8UC3 (8-bit unsigned, 3-channel BGR). The resulting image is saved to a file in the Images/Output directory with a filename formatted as outputX.png, where X is the provided index.

3.3 applyBoxBlur



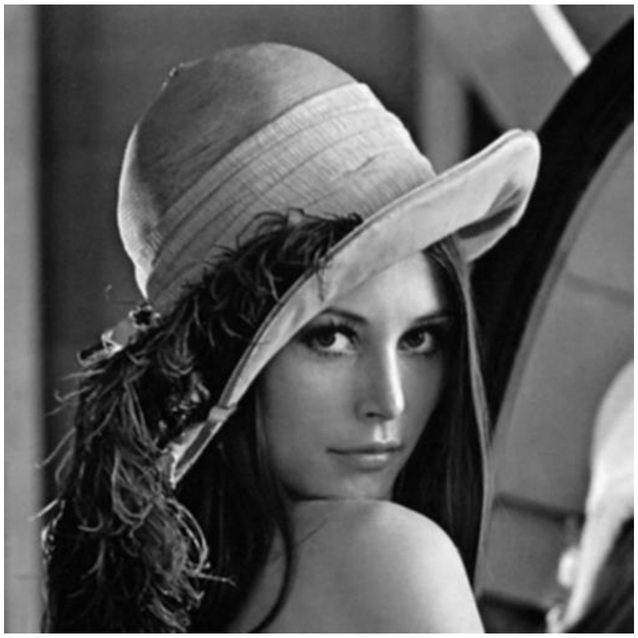
```
void applyBoxBlur(uchar* input, uchar* output, int width, int height, int kernelSize, int nthreads) {
    int halfK = kernelSize / 2;
    int kernelSum = kernelSize * kernelSize;
    int channels = 3;

#pragma omp parallel for num_threads(nthreads)
    for (int y = 0; y < height; ++y) {
        for (int x = 0; x < width; ++x) {
            for (int c = 0; c < channels; ++c) {
                int sum = 0;
                for (int ky = -halfK; ky <= halfK; ++ky) {
                    for (int kx = -halfK; kx <= halfK; ++kx) {
                        int px = min(max(x + kx, 0), width - 1);
                        int py = min(max(y + ky, 0), height - 1);
                        sum += input[(py * width + px) * channels + c];
                    }
                }
                output[(y * width + x) * channels + c] = sum / kernelSum;
            }
        }
    }
}
```

The applyBoxBlur function performs a box blur (a simple form of low-pass filtering) on a raw BGR image buffer using OpenMP to parallelize the computation. It operates by iterating over every pixel and averaging the surrounding pixel values within a square kernel of a specified size (kernelSize). This is done separately for each of the three color channels. Border pixels are handled using clamping, which replicates edge values rather than excluding them. The #pragma omp parallel for directive enables the loop to be executed in parallel across multiple threads to speed up the filtering process.

3.4 Test 1 (4 Threads and 3x3 Kernel)

Input:

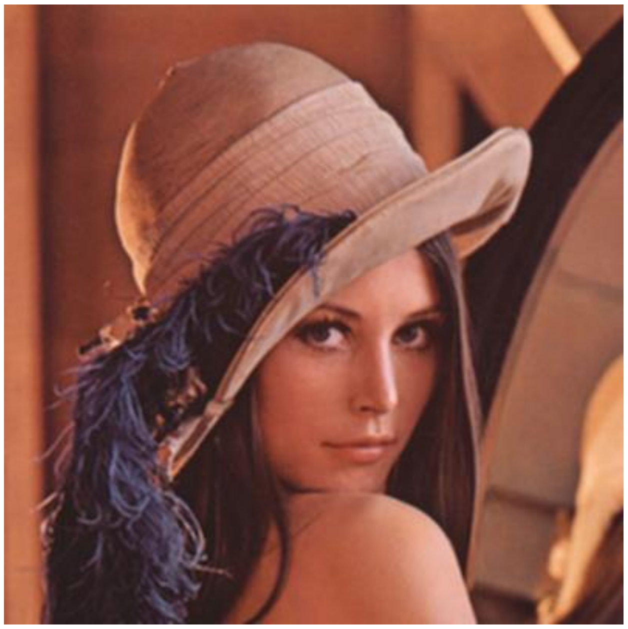
Output:



3.5 Test 2 (8 Threads and 3x3 Kernel)

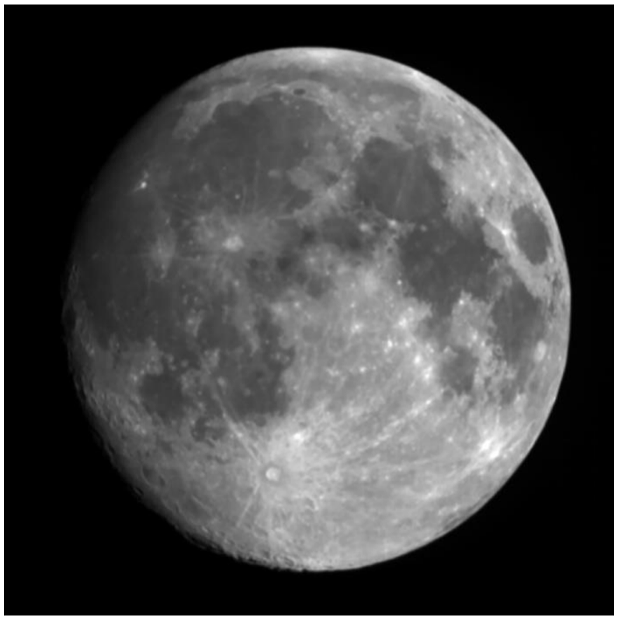
Input:

Output:

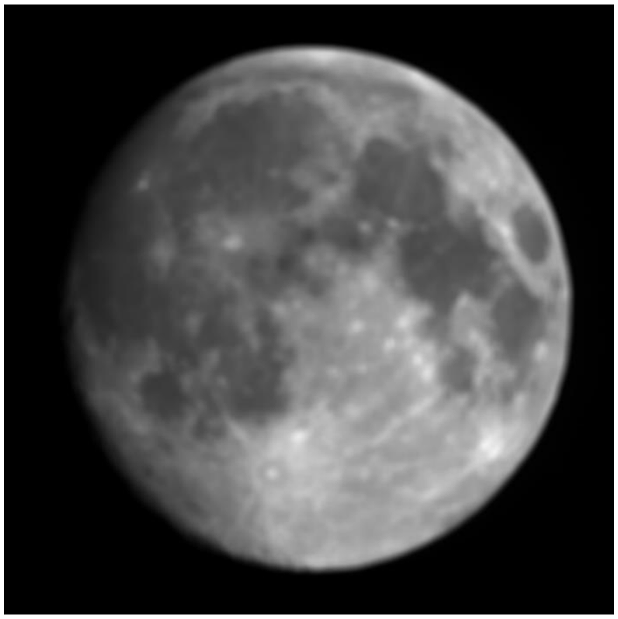


3.6 Test 3 (8 Threads and 11x11 Kernel)

Input:

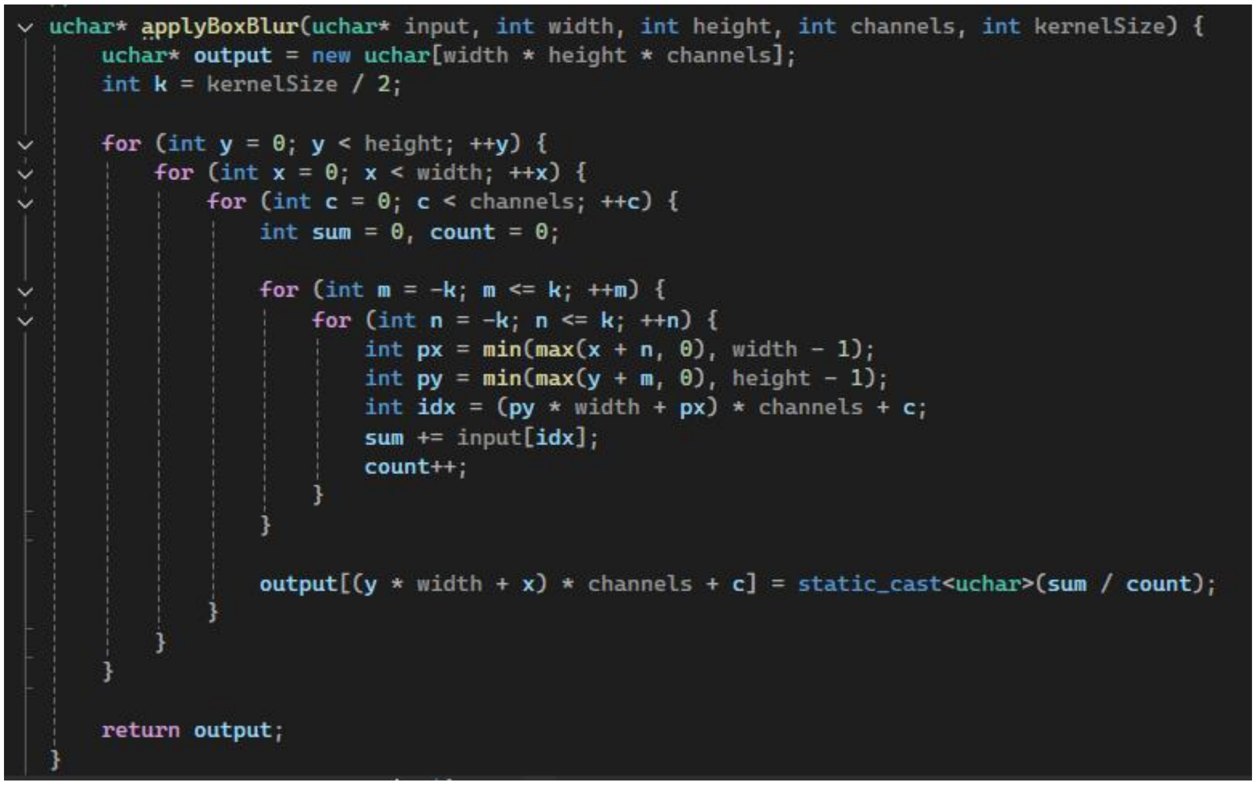


Output:



4.0 MPI

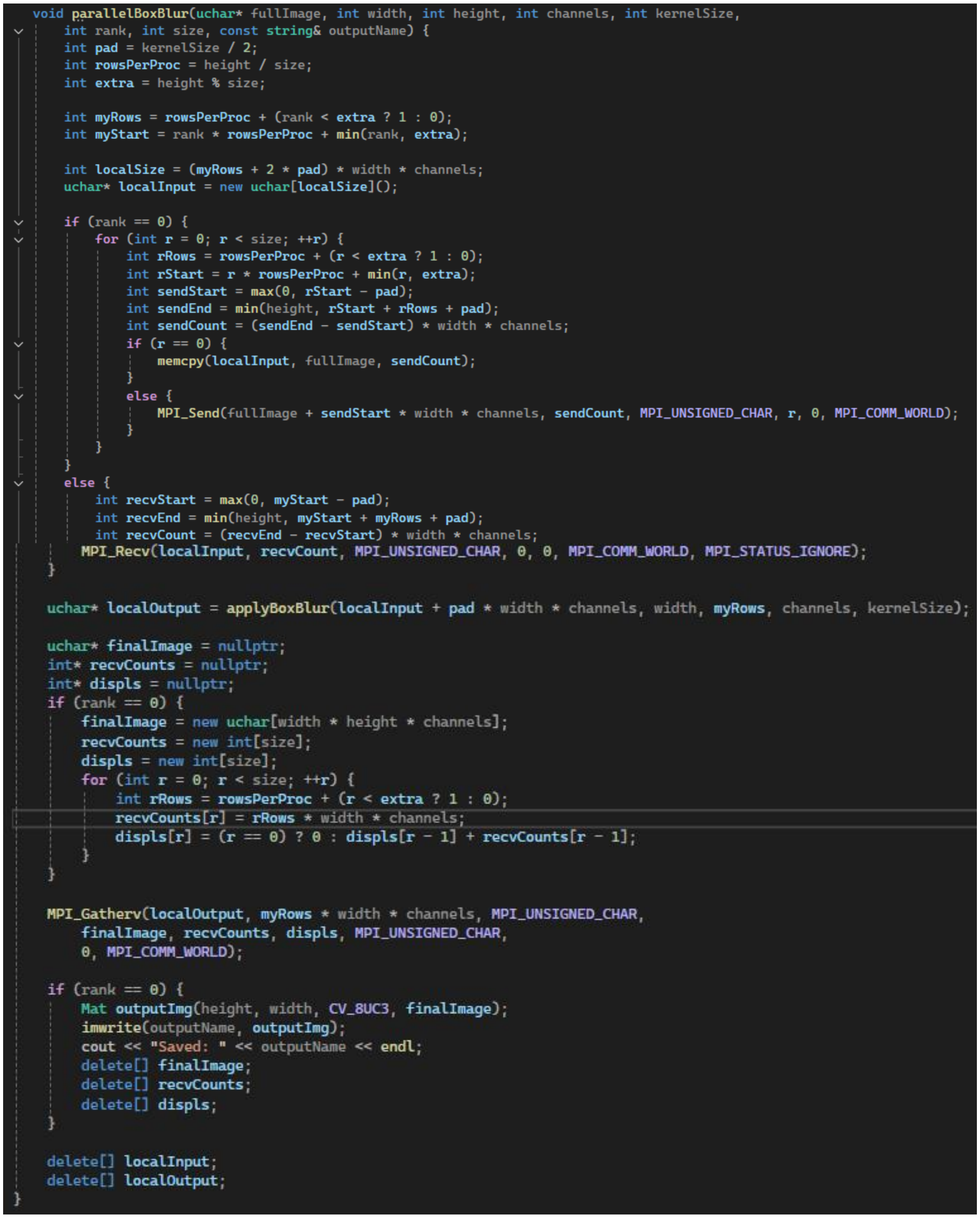
4.1 applyBoxBlur



```cpp
uchar* applyBoxBlur(uchar* input, int width, int height, int channels, int kernelSize) {
    uchar* output = new uchar[width * height * channels];
    int k = kernelSize / 2;

    for (int y = 0; y < height; ++y) {
        for (int x = 0; x < width; ++x) {
            for (int c = 0; c < channels; ++c) {
                int sum = 0, count = 0;

                for (int m = -k; m <= k; ++m) {
                    for (int n = -k; n <= k; ++n) {
                        int px = min(max(x + n, 0), width - 1);
                        int py = min(max(y + m, 0), height - 1);
                        int idx = (py * width + px) * channels + c;
                        sum += input[idx];
                        count++;
                    }
                }

                output[(y * width + x) * channels + c] = static_cast<uchar>(sum / count);
            }
        }
    }

    return output;
}
```

The applyBoxBlur function performs a sequential box blur (a basic low-pass filter) on an image represented as a raw uchar buffer. For each pixel, it averages the surrounding pixel values within a square window defined by kernelSize. The blur is applied individually to each color channel (B, G, R). Edge pixels are handled via clamping, ensuring the kernel doesn't read outside the image bounds. The result is stored in a newly allocated buffer, which the function returns to the caller. This function operates on a local section of the image and serves as the core filtering algorithm in the program.

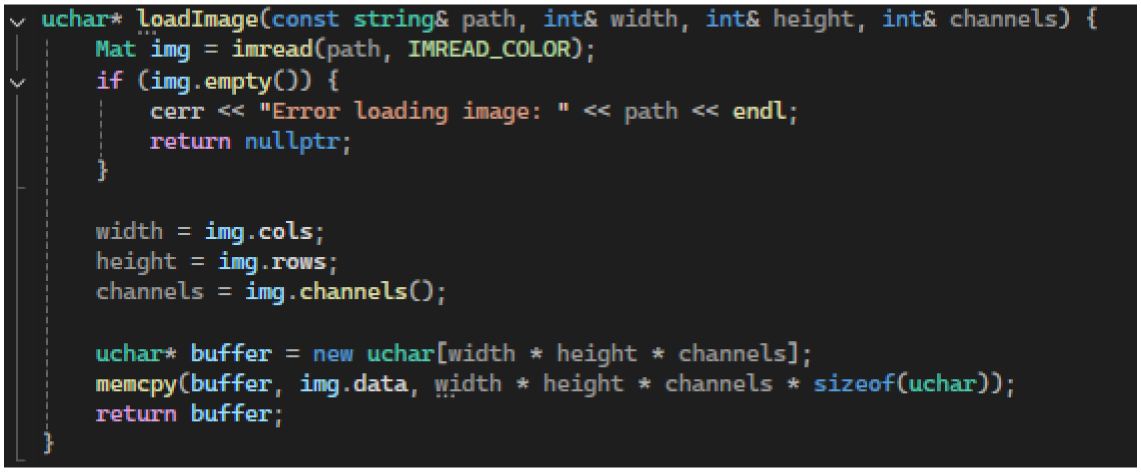
4.2 parallelBoxBlur



```cpp
void parallelBoxBlur(uchar* fullImage, int width, int height, int channels, int kernelSize,
    int rank, int size, const string& outputName) {
    int pad = kernelSize / 2;
    int rowsPerProc = height / size;
    int extra = height % size;

    int myRows = rowsPerProc + (rank < extra ? 1 : 0);
    int myStart = rank * rowsPerProc + min(rank, extra);

    int localSize = (myRows + 2 * pad) * width * channels;
    uchar* localInput = new uchar[localSize]();

    if (rank == 0) {
        for (int r = 0; r < size; ++r) {
            int rRows = rowsPerProc + (r < extra ? 1 : 0);
            int rStart = r * rowsPerProc + min(r, extra);
            int sendStart = max(0, rStart - pad);
            int sendEnd = min(height, rStart + rRows + pad);
            int sendCount = (sendEnd - sendStart) * width * channels;
            if (r == 0) {
                memcpy(localInput, fullImage, sendCount);
            }
            else {
                MPI_Send(fullImage + sendStart * width * channels, sendCount, MPI_UNSIGNED_CHAR, r, 0, MPI_COMM_WORLD);
            }
        }
    }
    else {
        int recvStart = max(0, myStart - pad);
        int recvEnd = min(height, myStart + myRows + pad);
        int recvCount = (recvEnd - recvStart) * width * channels;
        MPI_Recv(localInput, recvCount, MPI_UNSIGNED_CHAR, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }

    uchar* localOutput = applyBoxBlur(localInput + pad * width * channels, width, myRows, channels, kernelSize);

    uchar* finalImage = nullptr;
    int* recvCounts = nullptr;
    int* displs = nullptr;
    if (rank == 0) {
        finalImage = new uchar[width * height * channels];
        recvCounts = new int[size];
        displs = new int[size];
        for (int r = 0; r < size; ++r) {
            int rRows = rowsPerProc + (r < extra ? 1 : 0);
            recvCounts[r] = rRows * width * channels;
            displs[r] = (r == 0) ? 0 : displs[r - 1] + recvCounts[r - 1];
        }
    }

    MPI_Gatherv(localOutput, myRows * width * channels, MPI_UNSIGNED_CHAR,
        finalImage, recvCounts, displs, MPI_UNSIGNED_CHAR,
        0, MPI_COMM_WORLD);

    if (rank == 0) {
        Mat outputImg(height, width, CV_8UC3, finalImage);
        imwrite(outputName, outputImg);
        cout << "Saved: " << outputName << endl;
        delete[] finalImage;
        delete[] recvCounts;
        delete[] displs;
    }

    delete[] localInput;
    delete[] localOutput;
}
```

The parallelBoxBlur function distributes the box blur operation across multiple MPI processes to speed up processing of large images. Each process is assigned a slice of the image (a set of rows), along with ghost rows for padding, to avoid boundary issues when blurring. The root process (rank 0) loads and distributes the relevant image segments to each process using MPI_Send. Non-root processes receive their padded slices using MPI_Recv. Each process then applies the box blur locally using applyBoxBlur. The filtered slices are gathered back at the root process using MPI_Gatherv, accounting for varying row counts among processes. Finally, the root process saves the fully reconstructed image using OpenCV’s imwrite. Memory used for buffers is explicitly deallocated after use.

4.3 loadImage



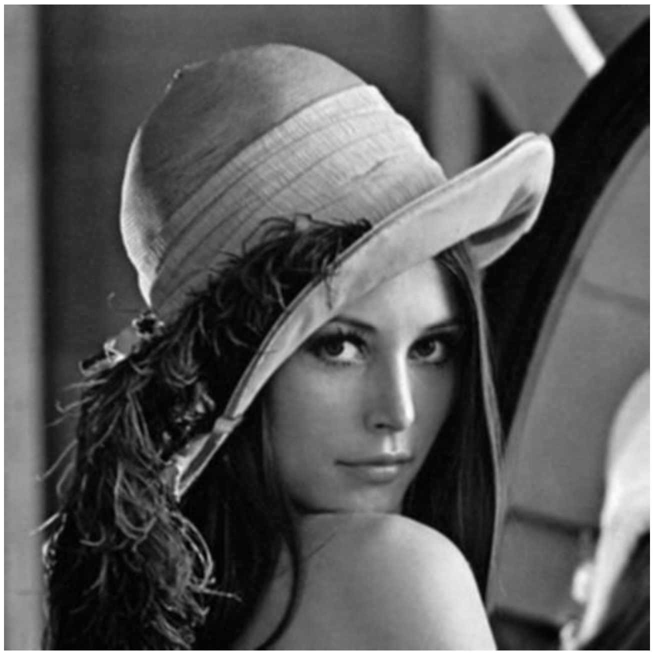
```
uchar* loadImage(const string& path, int& width, int& height, int& channels) {
    Mat img = imread(path, IMREAD_COLOR);
    if (img.empty()) {
        cerr << "Error loading image: " << path << endl;
        return nullptr;
    }

    width = img.cols;
    height = img.rows;
    channels = img.channels();

    uchar* buffer = new uchar[width * height * channels];
    memcpy(buffer, img.data, width * height * channels * sizeof(uchar));
    return buffer;
}
```

The loadImage function reads a color image from the provided file path using OpenCV and returns a raw uchar buffer representing its pixel data in BGR format. It also returns the image's width, height, and number of channels via reference parameters. If the image fails to load, it prints an error message and returns nullptr. This function is only called by the root process (rank 0) in the program, as only one process needs to load the full image before distribution.

4.4 Test 1 (2 Processors and 3x3 Kernel)

Input:

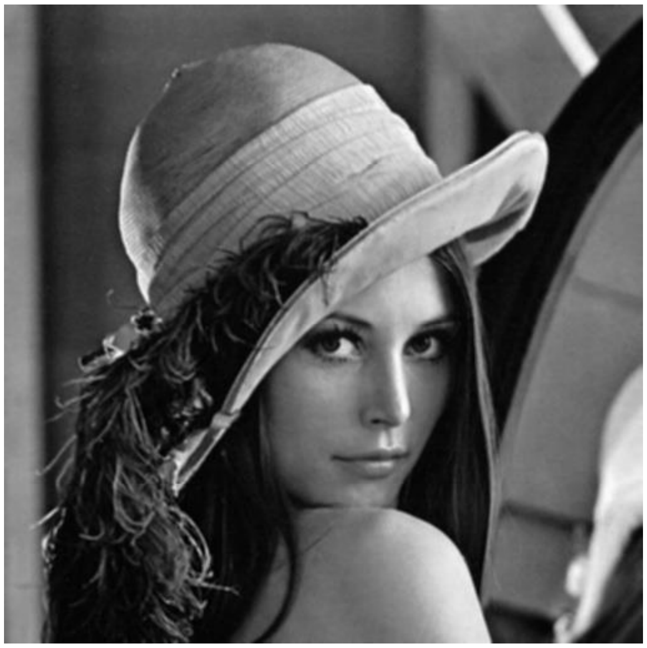
Output:



4.5 Test 2 (4 Processors and 3x3 Kernel)

Input:

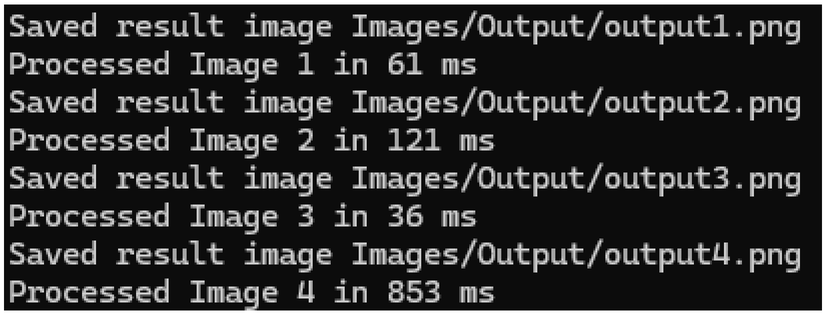
Output:



5.0 Conclusion

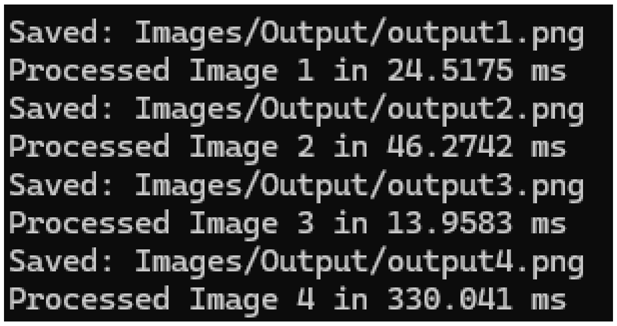
5.1 3x3 Kernel

Sequential:



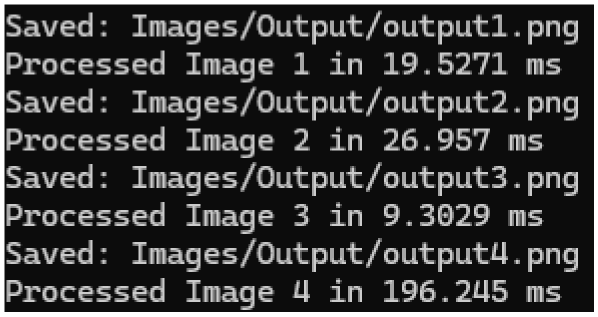
```
Saved result image Images/Output/output1.png
Processed Image 1 in 61 ms
Saved result image Images/Output/output2.png
Processed Image 2 in 121 ms
Saved result image Images/Output/output3.png
Processed Image 3 in 36 ms
Saved result image Images/Output/output4.png
Processed Image 4 in 853 ms
```

OpenMP 4 Threads :



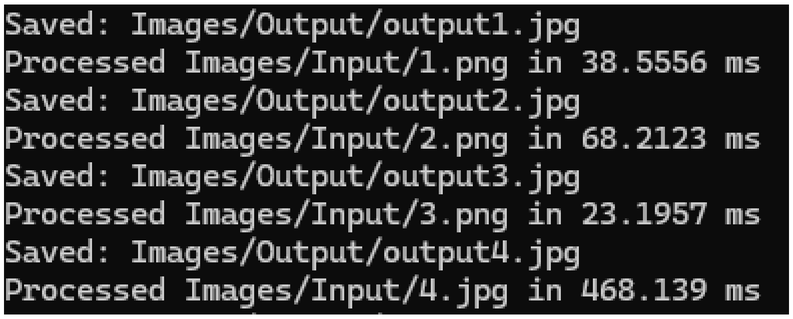
```
Saved: Images/Output/output1.png
Processed Image 1 in 24.5175 ms
Saved: Images/Output/output2.png
Processed Image 2 in 46.2742 ms
Saved: Images/Output/output3.png
Processed Image 3 in 13.9583 ms
Saved: Images/Output/output4.png
Processed Image 4 in 330.041 ms
```

OpenMP 8 Threads :



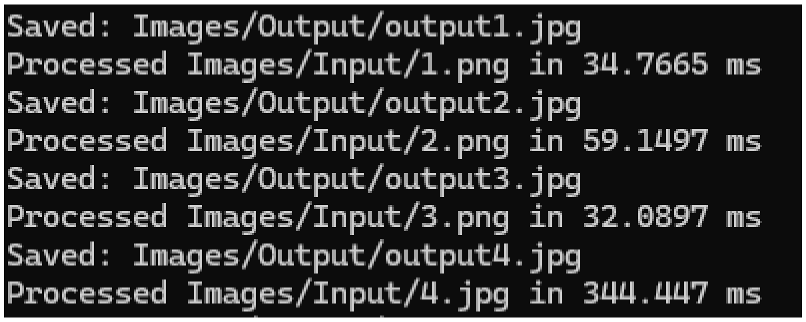
```
Saved: Images/Output/output1.png
Processed Image 1 in 19.5271 ms
Saved: Images/Output/output2.png
Processed Image 2 in 26.957 ms
Saved: Images/Output/output3.png
Processed Image 3 in 9.3029 ms
Saved: Images/Output/output4.png
Processed Image 4 in 196.245 ms
```

MPI 4 Threads :



```
Saved: Images/Output/output1.jpg
Processed Images/Input/1.png in 38.5556 ms
Saved: Images/Output/output2.jpg
Processed Images/Input/2.png in 68.2123 ms
Saved: Images/Output/output3.jpg
Processed Images/Input/3.png in 23.1957 ms
Saved: Images/Output/output4.jpg
Processed Images/Input/4.jpg in 468.139 ms
```

MPI 8 Threads:



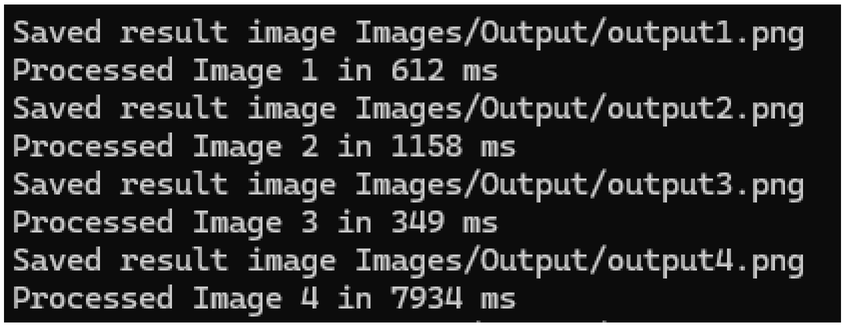
```
Saved: Images/Output/output1.jpg
Processed Images/Input/1.png in 34.7665 ms
Saved: Images/Output/output2.jpg
Processed Images/Input/2.png in 59.1497 ms
Saved: Images/Output/output3.jpg
Processed Images/Input/3.png in 32.0897 ms
Saved: Images/Output/output4.jpg
Processed Images/Input/4.jpg in 344.447 ms
```

Overall, OpenMP implementation outperformed both sequential and MPI approaches, with both 4 and 8 threads.

It is notable in both OpenMP and MPI that when the thread number increases, timing decreases.

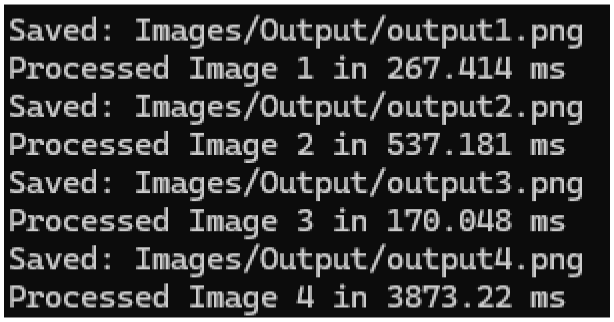
5.2 11x11 Kernel

Sequential:



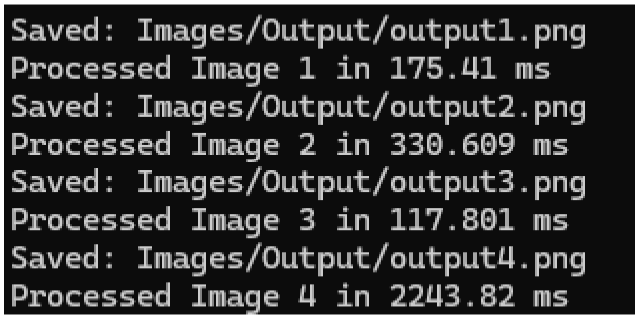
```
Saved result image Images/Output/output1.png
Processed Image 1 in 612 ms
Saved result image Images/Output/output2.png
Processed Image 2 in 1158 ms
Saved result image Images/Output/output3.png
Processed Image 3 in 349 ms
Saved result image Images/Output/output4.png
Processed Image 4 in 7934 ms
```

OpenMP 4 Threads:



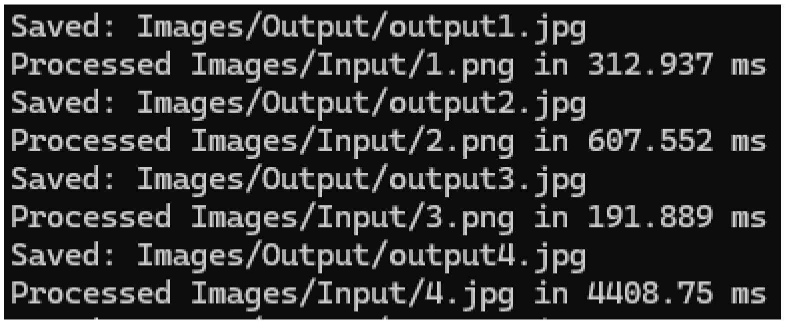
```
Saved: Images/Output/output1.png
Processed Image 1 in 267.414 ms
Saved: Images/Output/output2.png
Processed Image 2 in 537.181 ms
Saved: Images/Output/output3.png
Processed Image 3 in 170.048 ms
Saved: Images/Output/output4.png
Processed Image 4 in 3873.22 ms
```

OpenMP 8 Threads:



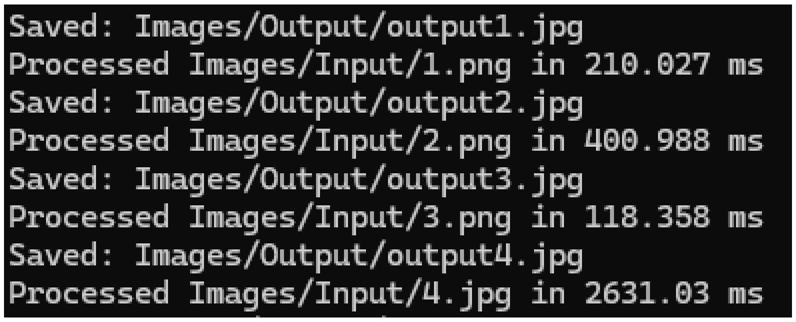
```
Saved: Images/Output/output1.png
Processed Image 1 in 175.41 ms
Saved: Images/Output/output2.png
Processed Image 2 in 330.609 ms
Saved: Images/Output/output3.png
Processed Image 3 in 117.801 ms
Saved: Images/Output/output4.png
Processed Image 4 in 2243.82 ms
```

MPI 4 Threads:



```
Saved: Images/Output/output1.jpg
Processed Images/Input/1.png in 312.937 ms
Saved: Images/Output/output2.jpg
Processed Images/Input/2.png in 607.552 ms
Saved: Images/Output/output3.jpg
Processed Images/Input/3.png in 191.889 ms
Saved: Images/Output/output4.jpg
Processed Images/Input/4.jpg in 4408.75 ms
```

MPI 8 Threads:

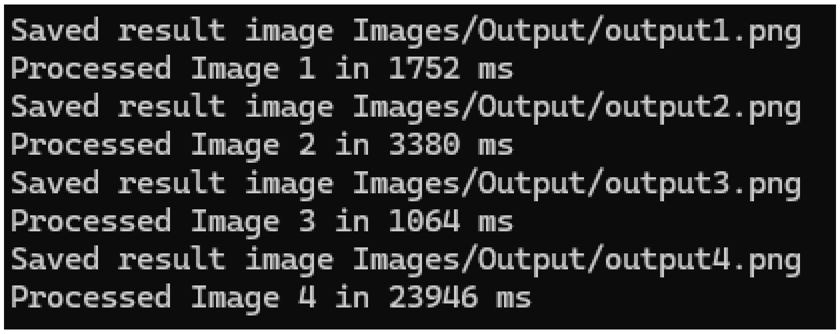

```
Saved: Images/Output/output1.jpg
Processed Images/Input/1.png in 210.027 ms
Saved: Images/Output/output2.jpg
Processed Images/Input/2.png in 400.988 ms
Saved: Images/Output/output3.jpg
Processed Images/Input/3.png in 118.358 ms
Saved: Images/Output/output4.jpg
Processed Images/Input/4.jpg in 2631.03 ms
```

Once again, OpenMP implementation outperformed both sequential and MPI approaches, with both 4 and 8 threads.

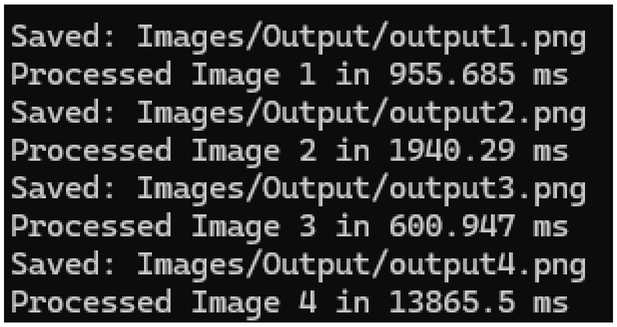
It is notable in both OpenMP and MPI that when the thread number increases, timing decreases.

5.3 21x21 Kernel

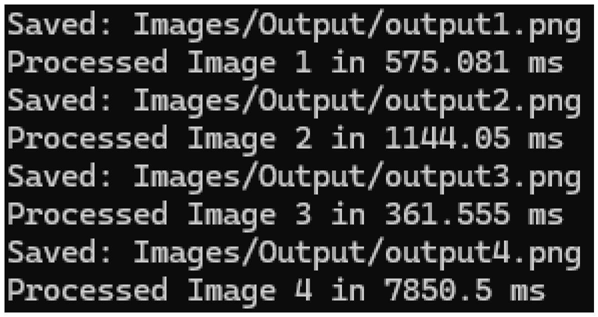
Sequential:


```
Saved result image Images/Output/output1.png
Processed Image 1 in 1752 ms
Saved result image Images/Output/output2.png
Processed Image 2 in 3380 ms
Saved result image Images/Output/output3.png
Processed Image 3 in 1064 ms
Saved result image Images/Output/output4.png
Processed Image 4 in 23946 ms
```

OpenMP 4 Threads :

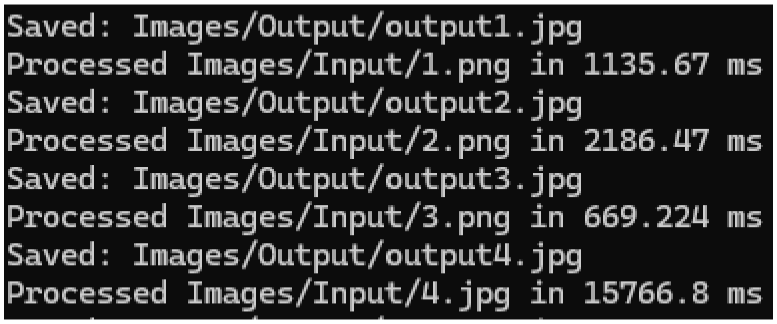

```
Saved: Images/Output/output1.png
Processed Image 1 in 955.685 ms
Saved: Images/Output/output2.png
Processed Image 2 in 1940.29 ms
Saved: Images/Output/output3.png
Processed Image 3 in 600.947 ms
Saved: Images/Output/output4.png
Processed Image 4 in 13865.5 ms
```

OpenMP 8 Threads :



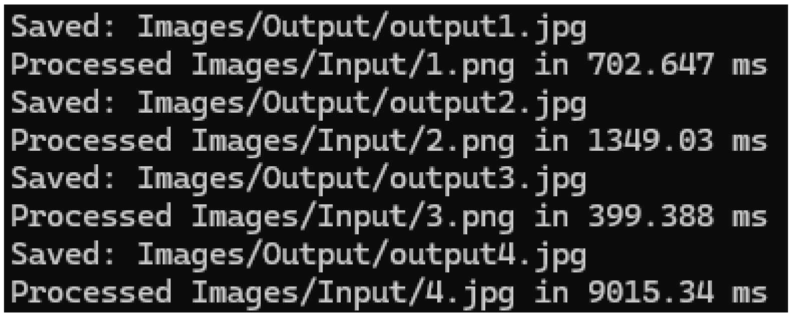
```
Saved: Images/Output/output1.png
Processed Image 1 in 575.081 ms
Saved: Images/Output/output2.png
Processed Image 2 in 1144.05 ms
Saved: Images/Output/output3.png
Processed Image 3 in 361.555 ms
Saved: Images/Output/output4.png
Processed Image 4 in 7850.5 ms
```

MPI 4 Threads :



```
Saved: Images/Output/output1.jpg
Processed Images/Input/1.png in 1135.67 ms
Saved: Images/Output/output2.jpg
Processed Images/Input/2.png in 2186.47 ms
Saved: Images/Output/output3.jpg
Processed Images/Input/3.png in 669.224 ms
Saved: Images/Output/output4.jpg
Processed Images/Input/4.jpg in 15766.8 ms
```

MPI 8 Threads:



```
Saved: Images/Output/output1.jpg
Processed Images/Input/1.png in 702.647 ms
Saved: Images/Output/output2.jpg
Processed Images/Input/2.png in 1349.03 ms
Saved: Images/Output/output3.jpg
Processed Images/Input/3.png in 399.388 ms
Saved: Images/Output/output4.jpg
Processed Images/Input/4.jpg in 9015.34 ms
```

Once again, OpenMP implementation outperformed both sequential and MPI approaches, with both 4 and 8 threads.

It is notable in both OpenMP and MPI that when the thread number increases, timing decreases.

6.0 GitHub Link

https://github.com/habibayasserr/LowPassFilter