HIGH PASS FILTER

CSE455: High Performance Computing (HPC)

Team 131

REDA MOHSEN REDA GEBRIL

ID: 18P5141

ZEYAD MAGDY IBRAHIM MOHAMED

ID: 18P5277

NOOR-ELDIN TALAAT EZZAT

ID: 18P3826

OMAR MOHAMED MAHMOUD

ID: 18P3919

1. Abstract

Image processing techniques play a crucial role in various applications, such as computer vision, medical imaging, and digital photography. One of the fundamental operations in image processing is the application of filters to enhance or modify specific features in an image. In this case, the program focuses on applying a high pass filter to a grayscale image.

2. Problem Definition

High pass filter is used to make the image appear sharper. High pass filters amplify noise. It allows high frequency components of the image to pass through and block low frequencies. if there is no change in intensity, nothing happens. But if one pixel is brighter than its neighbors, it gets boosted. We are going to solve this problem in three approaches: in a sequential approach, using OpenMp, and using MPI. Using dynamic filter size (the user can specify any odd filter size he want).

3. Sequential

I. Solution Introduction

This solution applies a high pass filter to a grayscale image. It reads an input image file, converts it to grayscale, and applies the filter to enhance edges and details. The program measures the sequential processing time and saves the resulting image. It provides a basic implementation of image processing techniques for enhancing images. The program utilizes arrays and nested loops to iterate over pixels and convolve them with the filter kernel. Overall, it offers a simple approach to apply a high pass filter to grayscale images.

II. Solution Illustration

Function inputImage:

- Takes the path of an image file, image width and height pointers, and filter size as arguments.
- Reads the image file and converts it to a grayscale image.
- Stores the grayscale image as an array of integers.
- Returns the grayscale image array.

Function highPassFilter:

- Takes an input image array, its width and height, and the desired filter size as arguments.
- It applies a high-pass filter to the input image array to obtain an output image array.
- The high-pass filter used here is a 3x3 filter with -1 in all positions except the center, which contains the sum of all other pixels.

Function createImage:

- Takes an image array, its width, height, and an index as arguments.
- Creates a new bitmap using the provided width and height.

- Iterates over each pixel in the image array and sets the corresponding pixel color in the new bitmap.
- Saves the new bitmap as a file with a specified name based on the index parameter.

main function:

- Initializes variables for image width, image height, and filter size.
- Prompts the user to enter an odd filter size and ensures it is valid.
- Converts the input image path from `std::string` to `System::String^`.
- Calls the `inputImage` function to retrieve grayscale pixel values of the input image.
- Allocates memory for the filtered image array.
- Measures the time taken to process the image using the `clock()` function.
- Calls the `highPassFilter` function to apply the filter to the input image.
- Stops the timer and calculates the total time taken for sequential processing.
- Calls the `createImage` function to save the resulting image.
- Prints the total time taken for sequential processing.
- Deletes the filtered image array and frees the memory allocated for the input image array.

III. Solution Implementation

a) Code

inputImage():

Take in the path of an image file, reads the image, and converts it to a grayscale image stored as an array of integers after applying zero padding and increasing the input/output image size by filtersize/2*2 in width and same in height.

```cpp

// function to take in the path of an image file, reads the image, and converts it to a grayscale image stored as an array of integers.
int* inputImage(int* w, int* h, System::String^ imagePath, int filterSize) //put the size of image in w & h
{
    int* input;
    int OriginalImageWidth, OriginalImageHeight;
    //Read Image and save it to local arrayss
    System::Drawing::Bitmap BM(imagePath); // create a Bitmap object from the image file

    OriginalImageWidth = BM.Width; // set the width of the image
    OriginalImageHeight = BM.Height;  // set the height of the image

    // Calculate padded width and height
    *w = BM.Width + 2 * (filterSize / 2);
    *h = BM.Height + 2 * (filterSize / 2);

    int* Red = new int[*w * *h]; // create an array for the red channel
    int* Green = new int[*w * *h];  // create an array for the green channel
    int* Blue = new int[*w * *h];  // create an array for the blue channel
    input = new int[*w * *h]; // create an array for the grayscale image

    // Calculate padding offset
    int paddingOffset = filterSize / 2;

    for (int i = 0; i < *h; i++) {  // loop through the rows of the image
        for (int j = 0; j < *w; j++) { // loop through the columns of the image
            int originalRow = i - paddingOffset;
            int originalCol = j - paddingOffset;

            // Check if the current pixel is within the original image boundaries
            if (originalRow >= 0 && originalRow < OriginalImageHeight && originalCol >= 0 && originalCol < OriginalImageWidth) {
                System::Drawing::Color c = BM.GetPixel(originalCol, originalRow); // get the color of the pixel at (j,i)

                Red[i * BM.Width + j] = c.R; // store the red channel value
                Blue[i * BM.Width + j] = c.B; // store the blue channel value
                Green[i * BM.Width + j] = c.G; // store the green channel value

                input[i * *w + j] = ((c.R + c.B + c.G) / 3); // gray scale value equals the average of RGB values
            }
            else {
                // Set the values to 0 for the padded pixels
                Red[i * *w + j] = 0;
                Blue[i * *w + j] = 0;
                Green[i * *w + j] = 0;
                input[i * *w + j] = 0;
            }
        }
    }

    return input; // return the grayscale image array
}

```

createImage():

Takes the grayscale image array, its width and height, and an index. It creates a new bitmap image from the grayscale array, ensures the pixel values are within the range of 0-255, sets the pixel colors in the bitmap, and saves the image to a file.

```cpp
void createImage(int* image, int width, int height, int index)
{
    // create a new bitmap with the specified width and height
    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++)
        {
            // ensure that the pixel value is within the range of 0-255
            //i * OriginalImageWidth + j
            if (image[i * width + j] < 0)
            {
                image[i * width + j] = 0;
            }
            if (image[i * width + j] > 255)
            {
                image[i * width + j] = 255;
            }

            // create a new color using the pixel value and set it as the pixel color in the new bitmap
            System::Drawing::Color c = System::Drawing::Color::FromArgb(image[i * MyNewImage.Width + j], image[i * MyNewImage.Width + j], image[i * MyNewImage.Width + j]);
            MyNewImage.SetPixel(j, i, c);
        }
    }

    // save the new bitmap to a file with a specified name (using the index parameter)
    MyNewImage.Save("..//Data//Output//outputRes" + index + ".png");

    // print a message indicating that the image has been saved
    cout << "result Image Saved " << index << endl;
}
```

b) Output

```
D:\Privacy\ASU_Spring23\Sen

Enter the filter size you want (should be odd value): 3

Processing....
result Image Saved 13
Sequential Time: 19 ms
Press any key to continue . . .
```

```
D:\Privacy\ASU_Spring23\Sen

Enter the filter size you want (should be odd value): 5

Processing....
result Image Saved 14
Sequential Time: 47 ms
Press any key to continue . . .
```

```
D:\Privacy\ASU_Spring23\Sen

Enter the filter size you want (should be odd value): 7

Processing....
result Image Saved 15
Sequential Time: 84 ms
Press any key to continue . . .
```

Figure 1 Using serial programming, with filter size 3, 5, 7 on a dog.png

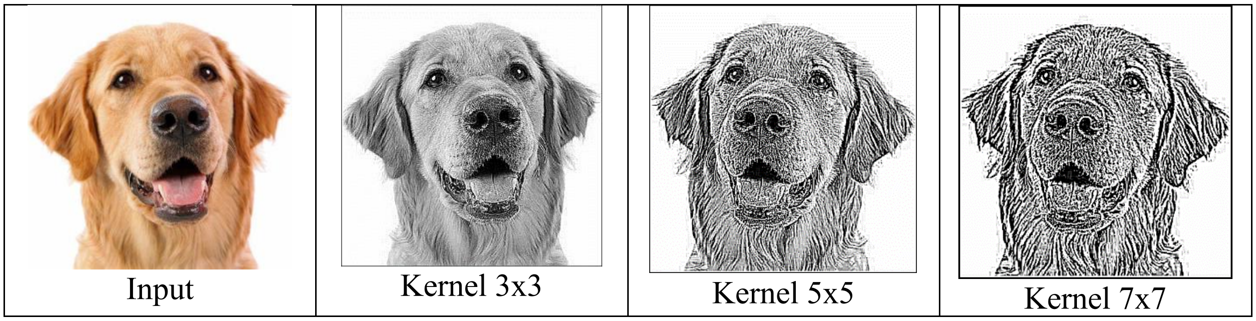

Figure 2 Output of serial programming of a dog.png

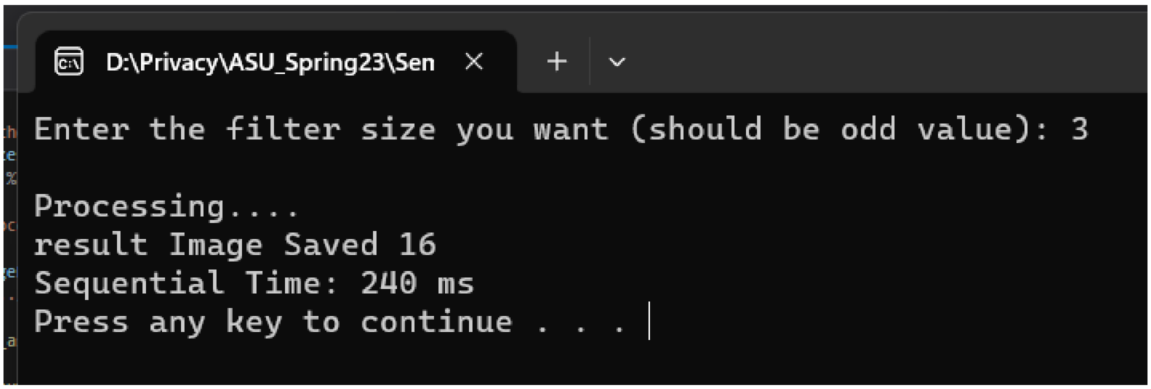

Figure 3 Using serial programming, filter size = 3 , on a moon.png

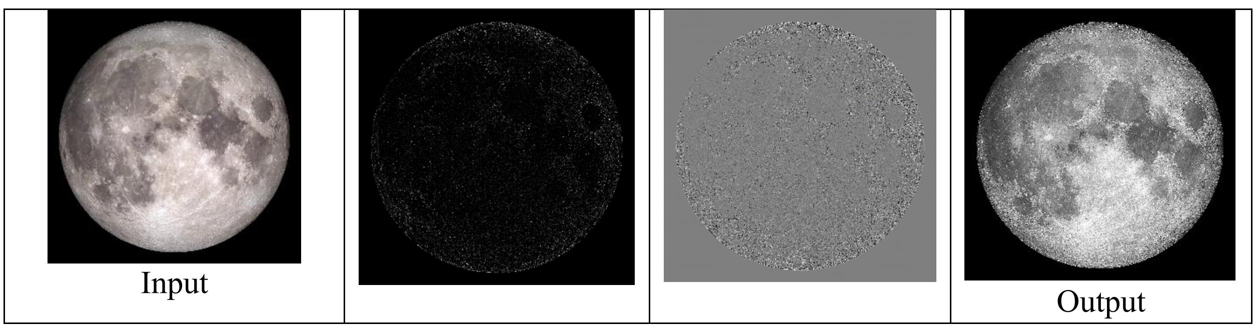

Figure 4 Output of serial programming of a moon.png

4. OpenMp

I. Solution Introduction

Image processing tasks, such as applying filters, can be computationally intensive and time-consuming, especially for large images. To improve the efficiency of these tasks, parallel computing techniques are employed to leverage the power of multi-core processors and process multiple pixels simultaneously. The solution utilizes OpenMP, an API for shared-memory parallel programming, to parallelize the high pass filtering process. By distributing the workload across multiple threads, the code takes advantage of parallel processing capabilities and reduces the overall execution time.

II. Solution Illustration

- The inputImage function is defined, which takes in the path of an image file, reads the image, and converts it to a grayscale image stored as an array of integers. It utilizes the System::Drawing::Bitmap class to load the image, extracts the RGB values of each pixel, and calculates the grayscale value and add zero padding according to filter size.
- The openMp_highPassFilter function is defined, which applies a parallel high pass filter to an input image array to obtain an output image array. It allocates memory for the output image array, sets the filter kernel dynamically based on the filter size, and uses nested loops to apply the filter to each pixel in parallel. The filtered value of each pixel is stored in the corresponding position in the output image array.
- The createImage function is defined, which takes an image array, its width and height, and an index as arguments. It creates a new bitmap image from the image array, ensures the pixel values are within the range of 0-255, sets the pixel colors in the bitmap, and saves the image to a file.
- In the main function, the dimensions of the input image (ImageWidth and ImageHeight) and the desired filter size (FilterSize) are initialized.
- The user is prompted to enter the filter size, and the input is validated to ensure it is an odd value.
- The path of the input image is set, and the inputImage function is called to retrieve the grayscale values of the pixels of the input image, storing them in the imageData array.
- Memory is allocated for the filteredImage array.
- The timer is started (start_s) to measure the time taken for image processing.
- The openMp_highPassFilter function is called to apply the high pass filter to the imageData array using OpenMP, and the resulting filtered image is stored in the filteredImage array.
- The timer is stopped (stop_s), and the total time taken to process the image using OpenMP is calculated and stored in the openmp_TotalTime variable.
- The createImage function is called to save the resulting image.
- The total time taken using OpenMP is displayed.
- Memory is freed, and the program terminates.
- In summary, the code reads an input image, converts it to grayscale, applies a high pass filter using OpenMP for parallel processing, saves the filtered image, and measures the time taken for the image processing.

III. Solution Implementation

a) Code

inputImage():

Take in the path of an image file, reads the image, and converts it to a grayscale image stored as an array of integers after applying zero padding and increasing the input/output image size by filtersize/2*2 in width and same in height.

```
// function to take in the path of an image file, reads the image, and converts it to a grayscale image stored as an array of integers.
int* inputImage(int* w, int* h, System::String^ imagePath, int filterSize) //put the size of image in w & h
{
    int* input;
    int OriginalImageWidth, OriginalImageHeight;
    //Read Image and save it to local arrayss
    System::Drawing::Bitmap BM(imagePath); // create a Bitmap object from the image file

    OriginalImageWidth = BM.Width; // set the width of the image
    OriginalImageHeight = BM.Height;  // set the height of the image

    // Calculate padded width and height
    *w = BM.Width + 2 * (filterSize / 2);
    *h = BM.Height + 2 * (filterSize / 2);

    int* Red = new int[*w * *h]; // create an array for the red channel
    int* Green = new int[*w * *h];  // create an array for the green channel
    int* Blue = new int[*w * *h];  // create an array for the blue channel
    input = new int[*w * *h]; // create an array for the grayscale image

    // Calculate padding offset
    int paddingOffset = filterSize / 2;

    for (int i = 0; i < *h; i++) {  // loop through the rows of the image
        for (int j = 0; j < *w; j++) { // loop through the columns of the image
            int originalRow = i - paddingOffset;
            int originalCol = j - paddingOffset;

            // Check if the current pixel is within the original image boundaries
            if (originalRow >= 0 && originalRow < OriginalImageHeight && originalCol >= 0 && originalCol < OriginalImageWidth) {
                System::Drawing::Color c = BM.GetPixel(originalCol, originalRow); // get the color of the pixel at (j,i)

                Red[i * BM.Width + j] = c.R; // store the red channel value
                Blue[i * BM.Width + j] = c.B; // store the blue channel value
                Green[i * BM.Width + j] = c.G; // store the green channel value

                input[i * *w + j] = ((c.R + c.B + c.G) / 3); // gray scale value equals the average of RGB values
            }
            else {
                // Set the values to 0 for the padded pixels
                Red[i * *w + j] = 0;
                Blue[i * *w + j] = 0;
                Green[i * *w + j] = 0;
                input[i * *w + j] = 0;
            }
        }
    }

    return input; // return the grayscale image array
}
```

createImage():

Takes the grayscale image array, its width and height, and an index. It creates a new bitmap image from the grayscale array, ensures the pixel values are within the range of 0-255, sets the pixel colors in the bitmap, and saves the image to a file.

```
void createImage(int* image, int width, int height, int index)
{
    // create a new bitmap with the specified width and height
    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++)
        {
            // ensure that the pixel value is within the range of 0-255
            //i * OriginalImageWidth + j
            if (image[i * width + j] < 0)
            {
                image[i * width + j] = 0;
            }
            if (image[i * width + j] > 255)
            {
                image[i * width + j] = 255;
            }

            // create a new color using the pixel value and set it as the pixel color in the new bitmap
            System::Drawing::Color c = System::Drawing::Color::FromArgb(image[i * MyNewImage.Width + j], image[i * MyNewImage.Width + j], image[i * MyNewImage.Width + j]);
            MyNewImage.SetPixel(j, i, c);
        }
    }

    // save the new bitmap to a file with a specified name (using the index parameter)
    MyNewImage.Save("..//Data//Output//outputRes" + index + ".png");

    // print a message indicating that the image has been saved
    cout << "result Image Saved " << index << endl;
}
```

openMp_highPassFilter():

Applies a parallel high pass filter to an input image array to obtain an output image array. It allocates memory for the output image array, sets the filter kernel dynamically based on the filter size, and uses nested loops to apply the filter to each pixel in parallel. The filtered value of each pixel is stored in the corresponding position in the output image array.

```
// This function takes an input image array, its width and height, and the desired filter size as arguments
// and applies a parallel high pass filter to the input image array to obtain an output image array
int* openMp_highPassFilter(int* input, int width, int height, int filterSize) {

    // Serial Section
    // Allocate memory dynamically for output image array to ensure enough memory
    int* output = new int[width * height];
    // Set the filter kernel dynamically based on the filter size
    std::vector<int> filter(filterSize * filterSize);
    int filterOffset = filterSize / 2;

    // Parallel Section
#pragma omp parallel for collapse(2) schedule(dynamic) shared(filterOffset, filter)
    for (int j = 0; j < filterSize; j++) {
        for (int i = 0; i < filterSize; i++) {
            if (i == filterOffset && j == filterOffset) {
                // Set the center pixel of the filter kernel to the sum of all other pixels
                filter[j * filterSize + i] = (filterSize * filterSize - 1);
            }
            else {
                // Set all other pixels of the filter kernel to -1
                filter[j * filterSize + i] = -1;
            }
        }
    }

    // Apply the high pass filter to each pixel in the input image array reduction(+:sum)
    int sum = 0;
    // Parallel Section
#pragma omp parallel for collapse(2) schedule(dynamic) shared(input,output,filter,filterOffset) reduction(+:sum)

    for (int y = filterOffset; y < height - filterOffset; y++) {
        for (int x = filterOffset; x < width - filterOffset; x++) {
            sum = 0;
            for (int j = -filterOffset; j <= filterOffset; j++) {
                for (int i = -filterOffset; i <= filterOffset; i++) {
                    // The filtered value of each pixel is stored in the corresponding position in the output image array
                    sum += input[(y + j) * width + (x + i)] * filter[(j + filterOffset) * filterSize + (i + filterOffset)];
                }
            }
            // Set the output pixel value
            //output[y * width + x] = sum;
            output[y * width + x] = 128 - sum;      //  Scale the output pixel for display purpose (visualization)
            //output[y * width + x] = sum + input[y * width + x];       // Add the filtered image array to the input image array to get final sharp image
        }
    }

    // Return the filtered image array
    return output;
}
```

Main():

Inialize height, width, and filter size used. And uses the above functions to read an input image, applies a high pass filter using OpenMP for parallel processing, and saves the filtered image. It measures the time taken for image processing and displays it.

```cpp
int main()
{
    int ImageWidth = 4, ImageHeight = 4, FilterSize = 3; // Initialize the dimensions of the input image and the filter size(odd value).

    do {
        cout << "Enter the filter size you want (should be odd value): ";
        cin >> FilterSize;
    } while (FilterSize % 2 != 1 || FilterSize < 1);

    cout << endl << "Processing...." << endl;

    System::String^ imagePath; // Declare the path of the input image as a System::String^ variable
    std::string img = "..//Data//Input//moon.png"; // Initialize the path of the input image as a std::string variable

    imagePath = marshal_as<System::String^>(img); // Convert the std::string variable to a System::String^ variable using the marshal_as function

    int* imageData = inputImage(&ImageWidth, &ImageHeight, imagePath, FilterSize); // Call the inputImage function to retrieve the grayscale values of the pixels of the input image and store them in the integer array imageData

    int* filteredImage = new int[ImageWidth * ImageHeight];

    int start_s, stop_s, openmp_TotalTime = 0; // Declare variables to measure the time taken to process the image

    start_s = clock(); // Start the timer
    // Perform any image processing tasks or calculations
    // Apply the high pass filter using openmp
    filteredImage = openMp_highPassFilter(imageData, ImageWidth, ImageHeight, FilterSize);

    stop_s = clock(); // Stop the timer
    openmp_TotalTime += (stop_s - start_s) / double(CLOCKS_PER_SEC) * 1000; // Calculate the total time taken to process the image

    createImage(filteredImage, ImageWidth, ImageHeight, 12); // Call the createImage function to save the resulting image

    cout << "Time using OpenMp: " << openmp_TotalTime << " ms" << endl; // Print the total time taken using openmp/parallel

    delete[] filteredImage;

    free(imageData); // Free the memory allocated for the imageData array

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

b) Output

```
D:\Privacy\ASU_Spring23\Sen

Enter the filter size you want (should be odd value): 3

Processing....
result Image Saved 7
Time using OpenMp: 55 ms
Press any key to continue . . .
```

```
D:\Privacy\ASU_Spring23\Sen

Enter the filter size you want (should be odd value): 5

Processing....
result Image Saved 8
Time using OpenMp: 55 ms
Press any key to continue . . .
```

```
D:\Privacy\ASU_Spring23\Sen

Enter the filter size you want (should be odd value): 7

Processing....
result Image Saved 9
Time using OpenMp: 64 ms
Press any key to continue . . .
```

Figure 5 Using openmp, filter size = 3,5,7, on a dog.png

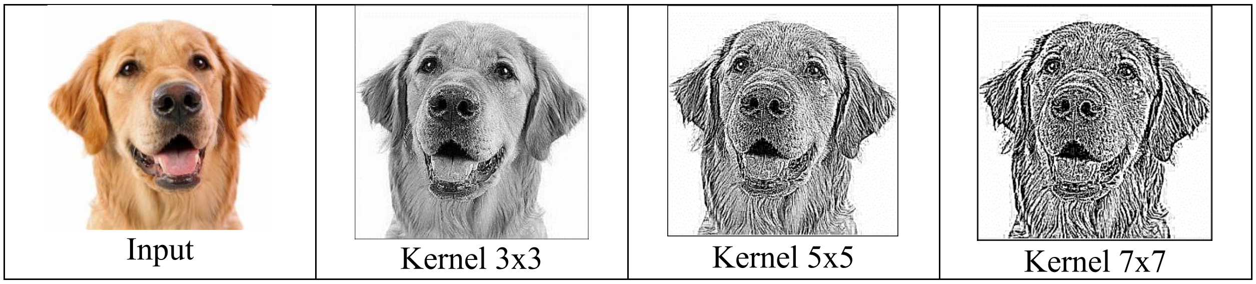

Figure 6 Output of parallel programming using openmp, on a dog.png

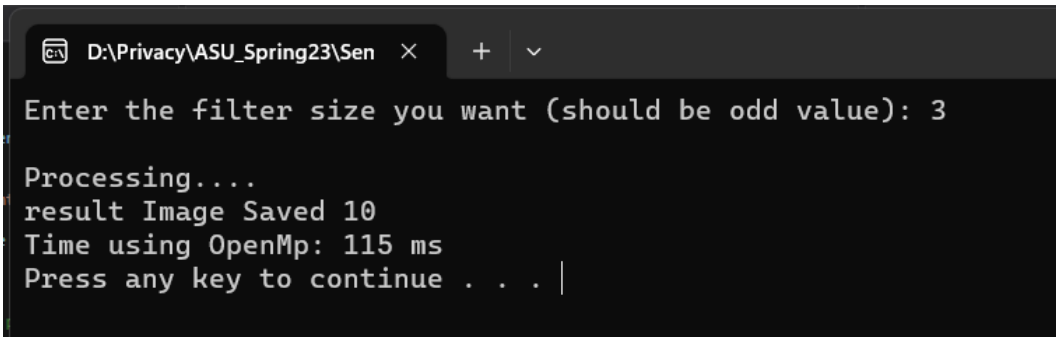

Figure 7 Using parallel programming, openmp filter size = 3, on a moon.png

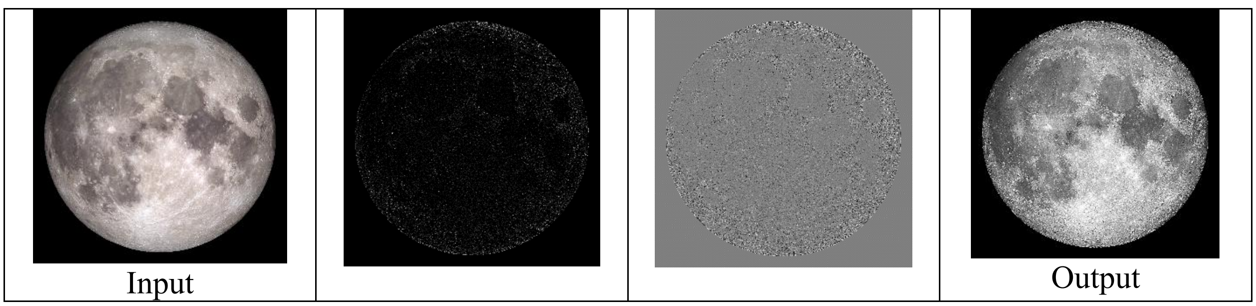

Figure 8 Output of openmp, on a moon.png

5. MPI

I. Solution Introduction

Parallel image processing using the Message Passing Interface (MPI). The code takes an input image, applies a high pass filter to the image in parallel using multiple processes, and saves the resulting filtered image.

II. Solution Illustration

- The `inputImage` function is defined to read an image file, convert it to grayscale, and store the grayscale values in an integer array. It uses the System.Drawing.Bitmap class to read the image and extract the RGB values of each pixel. The function also handles padding to accommodate the filter size.
- The `createImage` function is defined to create a new bitmap image from the filtered grayscale array. It iterates over each pixel, ensures the pixel values are within the range of 0-255, and sets the pixel colors in the bitmap. Finally, it saves the image to a file.

- In the `main` function, MPI is initialized, and the number of processes and the rank of the current process are obtained.
- The code prompts the user to enter the filter size, and the value is broadcasted to all other processes using `MPI_Bcast`. It performs input validation to ensure the filter size is an odd value.
- The path of the input image is set, and the `inputImage` function is called to retrieve the grayscale values of the image pixels.
- Memory is allocated for the `filteredImage` array, which will store the filtered output image.
- The timer is started to measure the time taken for image processing.
- The input image array is scattered to all processes using `MPI_Scatter`. Each process receives a portion of the input image to process.
- The filter kernel is dynamically created based on the filter size. It is an odd-sized square matrix with the center pixel having a special value and all other pixels set to -1.
- Border padding is added to the local input array of each process to account for filter calculations at the image edges.
- Each process applies the high pass filter to its portion of the input image by iterating over the pixels and performing the filter calculations. The filtered values are stored in the local output array.
- The filtered output image arrays from all processes are gathered using `MPI_Gather`, resulting in the `filteredImage` array containing the complete filtered image.
- If the current process is the master process (rank 0), the timer is stopped, and the total time taken for image processing is calculated. The `createImage` function is called to save the resulting filtered image to a file. The total time is displayed.
- Memory is freed, MPI is finalized, and the program terminates.

In summary, the solution leverages MPI to distribute the image processing workload among multiple processes, applies a high pass filter to the image in parallel, and produces a filtered image as the output. The code demonstrates the use of MPI collective operations such as `MPI_Scatter` and `MPI_Gather` to distribute and gather data across processes.

III. Solution Implementation

a) Code

inputImage():

Take in the path of an image file, reads the image, and converts it to a grayscale image stored as an array of integers after applying zero padding and increasing the input/output image size by filtersize/2*2 in width and same in height.

```cpp

// function to take in the path of an image file, reads the image, and converts it to a grayscale image stored as an array of integers.
int* inputImage(int* w, int* h, System::String^ imagePath, int filterSize) //put the size of image in w & h
{
    int* input;
    int OriginalImageWidth, OriginalImageHeight;
    //Read Image and save it to local arrayss
    System::Drawing::Bitmap BM(imagePath); // create a Bitmap object from the image file

    OriginalImageWidth = BM.Width; // set the width of the image
    OriginalImageHeight = BM.Height;  // set the height of the image

    // Calculate padded width and height
    *w = BM.Width + 2 * (filterSize / 2);
    *h = BM.Height + 2 * (filterSize / 2);

    int* Red = new int[*w * *h]; // create an array for the red channel
    int* Green = new int[*w * *h];  // create an array for the green channel
    int* Blue = new int[*w * *h];  // create an array for the blue channel
    input = new int[*w * *h]; // create an array for the grayscale image

    // Calculate padding offset
    int paddingOffset = filterSize / 2;

    for (int i = 0; i < *h; i++) {  // loop through the rows of the image
        for (int j = 0; j < *w; j++) { // loop through the columns of the image
            int originalRow = i - paddingOffset;
            int originalCol = j - paddingOffset;

            // Check if the current pixel is within the original image boundaries
            if (originalRow >= 0 && originalRow < OriginalImageHeight && originalCol >= 0 && originalCol < OriginalImageWidth) {
                System::Drawing::Color c = BM.GetPixel(originalCol, originalRow); // get the color of the pixel at (j,i)

                Red[i * BM.Width + j] = c.R; // store the red channel value
                Blue[i * BM.Width + j] = c.B; // store the blue channel value
                Green[i * BM.Width + j] = c.G; // store the green channel value

                input[i * *w + j] = ((c.R + c.B + c.G) / 3); // gray scale value equals the average of RGB values
            }
            else {
                // Set the values to 0 for the padded pixels
                Red[i * *w + j] = 0;
                Blue[i * *w + j] = 0;
                Green[i * *w + j] = 0;
                input[i * *w + j] = 0;
            }
        }
    }

    return input; // return the grayscale image array
}

```

createImage():

Takes the grayscale image array, its width and height, and an index. It creates a new bitmap image from the grayscale array, ensures the pixel values are within the range of 0-255, sets the pixel colors in the bitmap, and saves the image to a file.

```cpp

void createImage(int* image, int width, int height, int index)
{
    // create a new bitmap with the specified width and height
    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++)
        {
            // ensure that the pixel value is within the range of 0-255
            //i * OriginalImageWidth + j
            if (image[i * width + j] < 0)
            {
                image[i * width + j] = 0;
            }
            if (image[i * width + j] > 255)
            {
                image[i * width + j] = 255;
            }

            // create a new color using the pixel value and set it as the pixel color in the new bitmap
            System::Drawing::Color c = System::Drawing::Color::FromArgb(image[i * MyNewImage.Width + j], image[i * MyNewImage.Width + j], image[i * MyNewImage.Width + j]);
            MyNewImage.SetPixel(j, i, c);
        }
    }

    // save the new bitmap to a file with a specified name (using the index parameter)
    MyNewImage.Save("..//Data//Output//outputRes" + index + ".png");

    // print a message indicating that the image has been saved
    cout << "result Image Saved " << index << endl;
}

```

Main():

Sets up the MPI environment, performs parallel image processing using the high pass filter, saves the resulting image, and measures the time taken for the processing.

```cpp
int main()
{
    int width = 4, height = 4, filterSize = 3; // Initialize the dimensions of the input image and the filter size(odd value).

    MPI_Init(NULL, NULL);  // Initialize MPI
    int size, rank;
    MPI_Comm_size(MPI_COMM_WORLD, &size);  // Get the number of processes
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);  // Get the rank of the current process

    do {
        if (rank == 0) {
            cout << "Enter the filter size you want (should be an odd value): ";
            cin >> filterSize;
            cout << endl << "Processing...." << endl;
        }

        // Broadcast the filter size to all other processes
        MPI_Bcast(&filterSize, 1, MPI_INT, 0, MPI_COMM_WORLD);

        // Check if the filter size is valid
        if (filterSize % 2 != 1 || filterSize < 1) {
            if (rank == 0) {
                cout << "Invalid filter size. Please enter an odd value." << endl;
            }
        }
    } while (filterSize % 2 != 1 || filterSize < 1);


    System::String^ imagePath; // Declare the path of the input image as a System::String^ variable
    std::string img = "..//Data//Input//dog.png"; // Initialize the path of the input image as a std::string variable

    imagePath = marshal_as<System::String^>(img); // Convert the std::string variable to a System::String^ variable using the marshal_as function

    int* imageData = inputImage(&width, &height, imagePath, filterSize); // Call the inputImage function to retrieve the grayscale values of the pixels of the input image and store them in the integer array imageData

    int* filteredImage = new int[width * height];

    int start_s, stop_s, mpi_TotalTime = 0; // Declare variables to measure the time taken to process the image

    start_s = clock(); // Start the timer
    // Perform any image processing tasks or calculations
    // Apply the high pass filter using mpi
```

```cpp
    start_s = clock(); // Start the timer
    // Perform any image processing tasks or calculations
    // Apply the high pass filter using mpi
    int local_height = height / size; // Calculate the local height for each process
    int local_input_size = (local_height + 2) * width;
    int local_output_size = local_height * width;
    int* local_input = new int[local_input_size];
    int* local_output = new int[local_output_size];

    // Scatter the input image array to all processes
    MPI_Scatter(imageData, local_height * width, MPI_INT, local_input + width, local_height * width, MPI_INT, 0, MPI_COMM_WORLD);

    // Set the filter kernel dynamically based on the filter size
    std::vector<int> filter(filterSize * filterSize);
    int filterOffset = filterSize / 2;
    for (int j = 0; j < filterSize; j++) {
        for (int i = 0; i < filterSize; i++) {
            if (i == filterOffset && j == filterOffset) {
                // Set the center pixel of the filter kernel to the sum of all other pixels
                filter[j * filterSize + i] = (filterSize * filterSize - 1);
            }
            else {
                // Set all other pixels of the filter kernel to -1
                filter[j * filterSize + i] = -1;
            }
        }
    }

    // Add border padding to local_input
    if (rank == 0) {
        // Copy the top border pixels to the extra top row in the local_input array
        memcpy(local_input, imageData, width * sizeof(int));
        // Copy the bottom border pixels to the extra bottom row in the local_input array
        memcpy(local_input + (local_height + 1) * width, imageData + (height - 1) * width, width * sizeof(int));
    }

    // Copy the top border pixels to the extra top row in the local_input array of each process
    if (rank > 0) {
        memcpy(local_input, local_input + width, width * sizeof(int));
    }

    // Copy the bottom border pixels to the extra bottom row in the local_input array of each process
    if (rank < size - 1) {
        memcpy(local_input + (local_height + 1) * width, local_input + local_height * width, width * sizeof(int));
    }

    // Apply the high pass filter to each pixel in the input image array
    for (int y = 1; y <= local_height; y++) {
        for (int x = filterOffset; x < width - filterOffset; x++) {
            int sum = 0;
            for (int j = -filterOffset; j <= filterOffset; j++) {
                for (int i = -filterOffset; i <= filterOffset; i++) {
                    // The filtered value of each pixel is stored in the corresponding position in the output image array
                    sum += local_input[((y + j) * width) + (x + i)] * filter[(j + filterOffset) * filterSize + (i + filterOffset)];
                }
            }
            // Set the output pixel value
            local_output[(y - 1) * width + x] = sum + local_input[y * width + x];
```

```cpp
            }
            // Set the output pixel value
            local_output[(y - 1) * width + x] = sum + local_input[y * width + x];
        }
    }

    // Gather the filtered output image arrays from all processes
    MPI_Gather(local_output, local_output_size, MPI_INT, filteredImage, local_output_size, MPI_INT, 0, MPI_COMM_WORLD);

    if (rank == 0) {
        stop_s = clock(); // Stop the timer
        mpi_TotalTime += (stop_s - start_s) / double(CLOCKS_PER_SEC) * 1000; // Calculate the total time taken to process the image
        createImage(filteredImage, width, height, 2); // Call the createImage function to save the resulting image
        cout << "Time using MPI: " << mpi_TotalTime << " ms" << endl; // Print the total time taken using openmp/parallel
        delete[] filteredImage;
        MPI_Finalize();  // Finalize MPI
    }
    else {
        MPI_Finalize();  // Finalize MPI
    }

    delete[] imageData; // Free the memory allocated for the input image array

    return 0;
}
```

a) Output

```
mpi.cpp    Developer PowerShell    Source.cpp
+ Developer PowerShell
PS D:\Privacy\ASU_Spring23\Semester10\HPC\Project\HPC_ProjectTemplate\debug> mpiexec -n 8 "HPC_ProjectTemplate.exe"
Enter the filter size you want (should be an odd value): 3

Processing....
result Image Saved 0
Time using MPI: 32 ms
PS D:\Privacy\ASU_Spring23\Semester10\HPC\Project\HPC_ProjectTemplate\debug> mpiexec -n 8 "HPC_ProjectTemplate.exe"
Enter the filter size you want (should be an odd value): 5

Processing....
result Image Saved 1
Time using MPI: 22 ms
PS D:\Privacy\ASU_Spring23\Semester10\HPC\Project\HPC_ProjectTemplate\debug> mpiexec -n 8 "HPC_ProjectTemplate.exe"
Enter the filter size you want (should be an odd value): 7

Processing....
result Image Saved 2
Time using MPI: 51 ms
PS D:\Privacy\ASU_Spring23\Semester10\HPC\Project\HPC_ProjectTemplate\debug>
```

Figure 9 Using MPI, with filter size 3,5,7 on dog.png

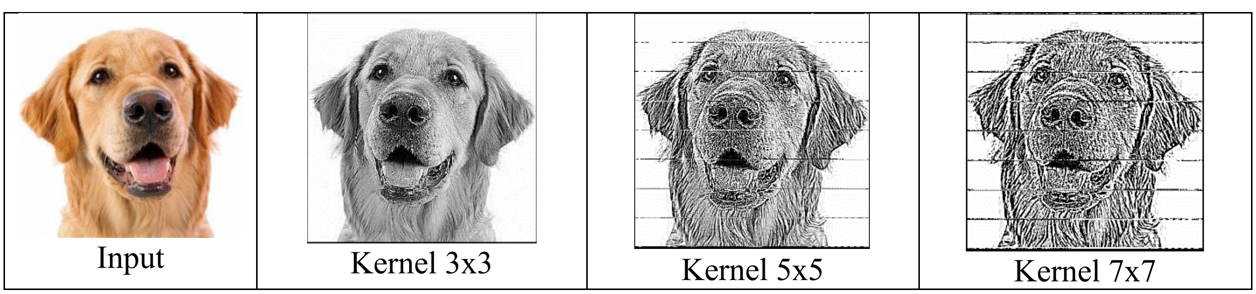

Figure 10 Output of MPI, on a dog.png

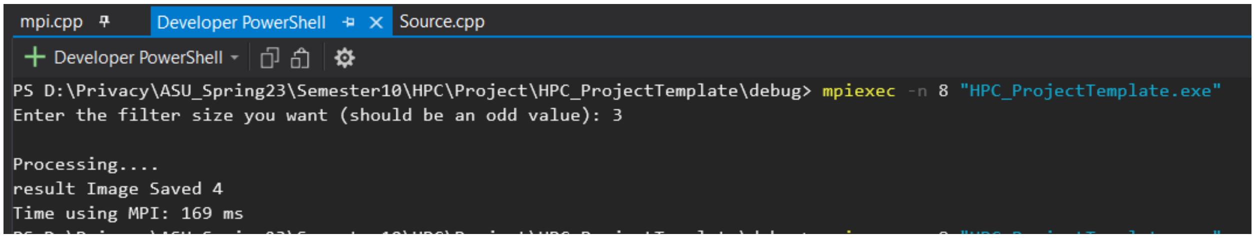

Figure 11 Using MPI, with filter size 3 on moon.png but with different pixel values for visualization and image 4 is the real output

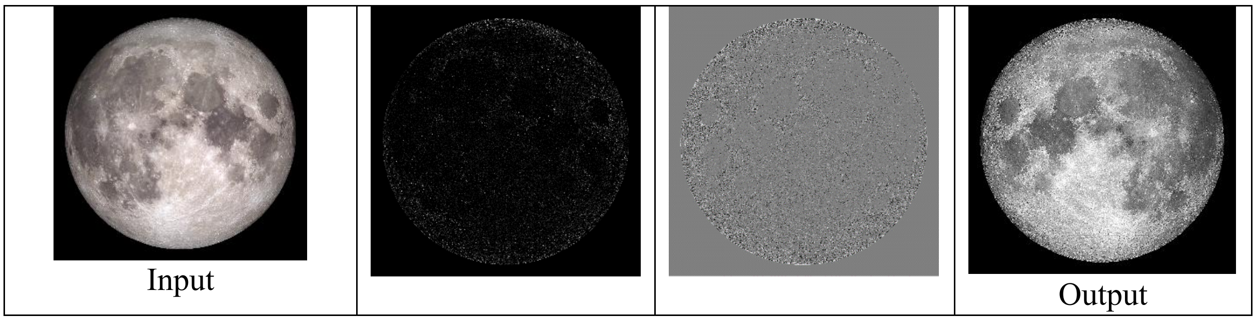

Figure 12 Output of MPI, on a moon.png

Comparing Between the Three Approaches

In this section, there is a comparison between the three approaches we used in the report. Using the same number of processors (8), same filter size, same image, same resolution, and same method for calculating time.

	Dog.png		Moon.png	
	Filter Size	**Time in Milliseconds**	**Filter Size**	**Time in Milliseconds**
Sequential	3	19	3	240
	5	47		
	7	84		
OpenMp	3	55	3	115
	5	55		
	7	64		
MPI	3	32	3	169
	5	22		
	7	51		

For the Moon Image

The input was 1960x1960 and the output was 1962x1962 because of the zero padding, so we can apply convolution between filter mask and the input image after it converted to an array and grayscaled. It is obvious that in the sequential approach it is the slowest among others as it run on 1 processor(main thread) only. While in the openmp approach, it was much faster than the other approaches as it uses parallel programming and the loop iterating each pixel is divided among the 8 processors used in this approach. Mpi, was faster than the sequential but slower than the openmp in this run.

For the Dog Image

The input was 512x512 and the output was 514x514 because of the zero padding, so we can apply convolution between filter mask and the input image without losing the border pixels after it converted to an array and grayscale. Using filter size equal to 7, it is obvious that in the sequential approach it is the slowest among others as it run on 1 processor(main thread) only. While in the mpi approach, it was much faster than the other approaches as it uses parallel programming and the loop iterating each pixel is divided among the 8 processors used in this approach. openmp, was faster than the sequential but slower than the openmp in this run. Note, in small filter sizes and small images the parallel programming didn't seem to be large effective in the problem.

Conclusion

Overall, the comparison highlights the advantages of parallel programming, specifically with OpenMP and MPI, in optimizing image convolution tasks. Parallel approaches offer substantial speed improvements over sequential execution, with the choice between OpenMP and MPI depending on the specific requirements of the task, such as the image size, filter size, and available hardware resources.

Link

reda-mohsen/High_Pass_Filter: HPC, Parallel programming. (github.com)