

HPC Project Documentation

Parallel High Pass Filter

Maryam Ahmed Galal Nough – 18P2824

Youssef Sherif Mohamed Youssef – 18P3054

1.0. Implementation

Please note that this implementation allows for different kernel sizes.

1.1. Sequential Implementation

```
16 int* inputImage(int* w, int* h, System::String^ imagePath) //put the size of image in w & h
17 {
18     int* input;
19
20     int OriginalImageWidth, OriginalImageHeight;
21
22     //*****Read Image and save it to local arrayss*****
23     //Read Image and save it to local arrayss
24     System::Drawing::Bitmap BM(imagePath);
25
26     OriginalImageWidth = BM.Width;
27     OriginalImageHeight = BM.Height;
28     *w = BM.Width;
29     *h = BM.Height;
30     int *Red = new int[BM.Height * BM.Width];
31     int *Green = new int[BM.Height * BM.Width];
32     int *Blue = new int[BM.Height * BM.Width];
33     input = new int[BM.Height*BM.Width];
34     for (int i = 0; i < BM.Height; i++)
35     {
36         for (int j = 0; j < BM.Width; j++)
37         {
38             System::Drawing::Color c = BM.GetPixel(j, i);
39             Red[i * BM.Width + j] = c.R;
40             Blue[i * BM.Width + j] = c.B;
41             Green[i * BM.Width + j] = c.G;
42
43             input[i*BM.Width + j] = ((c.R + c.B + c.G) / 3); //gray scale value equals the average of RGB values
44         }
45     }
46     return input;
47 }
```

The function above is implemented to take as input an image, transforming it to a vector of size equal to the product of the image's width and height, and filling the vector with the corresponding bit values.

```
55 void createImage(int* image, int width, int height, int index)
56 {
57     System::Drawing::Bitmap MyNewImage(width, height);
58
59     for (int i = 0; i < MyNewImage.Height; i++)
60     {
61         for (int j = 0; j < MyNewImage.Width; j++)
62         {
63             //i * OriginalImageWidth + j
64             if (image[i*width + j] < 0)
65             {
66                 image[i*width + j] = 0;
67             }
68             if (image[i*width + j] > 255)
69             {
70                 image[i*width + j] = 255;
71             }
72             System::Drawing::Color c = System::Drawing::Color::FromArgb(image[i*MyNewImage.Width + j], image[i*MyNewImage.Width + j], image[i*MyNewImage.Width + j]);
73             MyNewImage.SetPixel(j, i, c);
74         }
75     }
76     MyNewImage.Save("C:\\Users\\user\\source\\repos\\VHPC_ProjectTemplate_OpenMP\\VHPC_ProjectTemplate" + index + ".png");
77     cout << "result Image Saved " << index << endl;
78 }
79 }
```

We also implement a function to create an image upon receiving the result of the algorithm in the form of an array. This function converts the vector to an image and saves it locally.

```

5 int** padImgMatrix(int rows, int cols, int* imgMatrix)
6 {
7     int** paddedImgMatrix = new int* [rows + 2];
8
9     for (int i = 0; i < rows + 2; i++) {
10         paddedImgMatrix[i] = new int[cols + 2];
11     }
12
13     for (int i = 0; i < rows + 2; i++) {
14         for (int j = 0; j < cols + 2; j++) {
15             paddedImgMatrix[i][j] = 0;
16         }
17     }
18
19     for (int i = 0; i < rows; i++) {
20         for (int j = 0; j < cols; j++) {
21             paddedImgMatrix[i + 1][j + 1] = *(imgMatrix + i * cols + j); //imgMatrix[i][j]
22         }
23     }
24     return paddedImgMatrix;
25 }
26

```

This function is used to pad the image matrix to handle the borders of the input matrix by giving it new borders of value 0. We pass to the function 3 different parameters, the height of the image specified as the row, the width of the image specified as the column, and the pointer that corresponds to the matrix holding the pixels of the image. The first thing is to define a dynamic 2D array with size rows + 2 and cols + 2, which correspond to the height and width of the image in addition to 2 extra rows that represent the padding. After that, we start adding the width of the image to our 2D matrix using the first for loop in the function. Then, we initialize the paddedImgMatrix with 0s. Finally, we copy the values from the imgMatrix to their correct positions, which is the center of the paddedImgMatrix, leaving out the first and last row and column, using a nested loop.

```

103
104 int** vectorToMatrix(int* paddedImgVector, int rows, int cols) {
105     int** paddedImgMatrix = new int* [rows];
106     for (int i = 0; i < rows; i++) {
107         paddedImgMatrix[i] = new int[cols];
108         for (int j = 0; j < cols; j++) {
109             paddedImgMatrix[i][j] = paddedImgVector[i * cols + j];
110         }
111     }
112     return paddedImgMatrix;
113 }
114
115 int* matrixToVector(int rows, int cols, int** paddedImgMatrix) {
116     int* paddedImgVector = new int[rows * cols];
117     for (int i = 0; i < rows; i++) {
118         for (int j = 0; j < cols; j++) {
119             paddedImgVector[i * cols + j] = paddedImgMatrix[i][j];
120         }
121     }
122     return paddedImgVector;
123 }
124

```

Two other functions are implemented to transform a matrix to a vector and vice versa.

```

125 int main()
126 {
127     int ImageWidth = 4, ImageHeight = 4;
128
129     int start_s, stop_s, TotalTime = 0, kernel = 3;
130
131
132     System::String^ imagePath;
133     std::string img;
134     img = "C:\\Users\\user\\source\\repos\\HPC_ProjectTemplate_OpenMP\\HPC_ProjectTemplate\\lena.png";
135
136     imagePath = marshal_as<System::String^>(img);
137     int* imageData = inputImage(ImageWidth, ImageHeight, imagePath);
138     start_s = clock();
139
140     int** imgMatrix = vectorToMatrix(imageData, ImageHeight, ImageWidth);
141
142     int** paddedImgMatrix = padImgMatrix(ImageHeight, ImageWidth, imgMatrix);
143
144     int paddedImg = ImageHeight + 2;
145     int sum = 0;
146     int kernelMatrix[3][3] = {{0, -1, 0}, {-1, 4, -1}, {0, -1, 0}};
147     int imgBorder = kernel / 2;
148     int borderlessImg = paddedImg - imgBorder;
149     int** newImgMatrix = (int**)malloc(ImageHeight * sizeof(int*));
150
151     for (int i = 0; i < ImageHeight; i++)
152     {
153         newImgMatrix[i] = (int*)malloc(ImageHeight * sizeof(int));
154     }
155

```

In the next part of the code, we declared and initialized all the variables that will be used in our code. The most important variables are the following:

- `sum` = a dummy variable that is used to hold the values of the multiplication that will be explained below.
- `image` = a variable that holds the value of the rows and columns of the image.
- `kernel` = a variable that holds the value of the rows and columns of the kernel square matrix.
- `paddedImg` = a variable for the new value of the rows and columns of the image after padding it.
- `imgBorder` = this variable is a limit that will be used below in the matrix multiplication to neglect the rows and cols added in the padding function from the top of the matrix and start the multiplication from the pixels of the image.
- `borderlessImg` = this variable is another limit that will be used below in the matrix multiplication to neglect the rows and the columns added in the padding function from the bottom of the matrix and end the multiplication at the last pixel of the image.

We also dynamically allocate the `newImgMatrix` which will hold the final result of the algorithm.

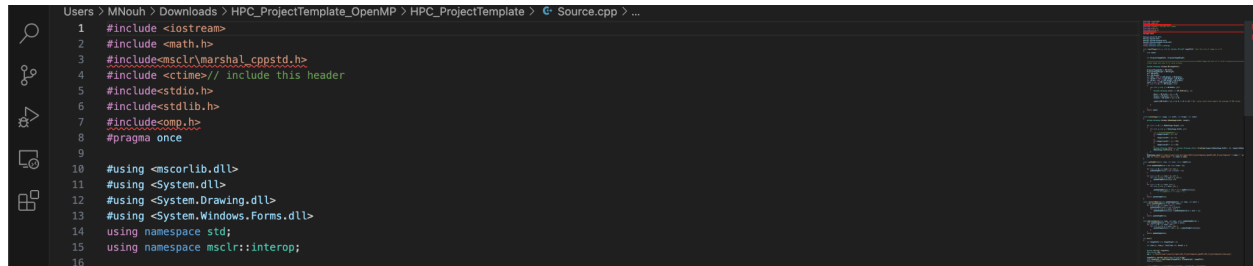
```
161 for (int i = imgBorder; i < borderlessImg; i++)
162 {
163     for (int j = imgBorder; j < borderlessImg; j++)
164     {
165         sum = 0;
166         for (int k = 0; k < kernel; k++)
167         {
168             for (int l = 0; l < kernel; l++)
169             {
170                 sum += kernelMatrix[k][l] * paddedImgMatrix[i - imgBorder + k][j - imgBorder + l];
171             }
172             newImgMatrix[i - imgBorder][j - imgBorder] = sum;
173         }
174     }
175 }
176
177 imageData = matrixToVector(ImageHeight, ImageWidth, newImgMatrix);
178
179 stop_s = clock();
180 TotalTime += (stop_s - start_s) / double(CLOCKS_PER_SEC) * 1000;
181 createImage(imageData, ImageWidth, ImageHeight, 1);
182 cout << "time: " << TotalTime << endl;
183
184 free(imageData);
185 return 0;
186
187 }
```

This part of the code computes the multiplication of the two matrices in the algorithm using 4 for loops. The first for loop is the one controlling the rows in the `imgMatrix`, while the second one loops on the columns of the `imgMatrix`. The third and fourth for loops mirror the first and second with respect to the `kernelMatrix`. We first loop on the rows and columns of the image starting from the `imgBorder` to `borderlessImg`, which represent the first row and column after the border we added using the padding function, ending right before the last row and column, which is also added due to the padding. These two for loops also apply the concept of the sliding window in which we compute the multiplication on each row then move the index by 1. Finally, we compute the value of each cell in the result matrix by multiplying the kernel's current value at position (k, l) by its corresponding position from the `paddedImgMatrix` of value $(i+k, j+l)$, excluding the `imgBorder`, limiting the sliding window to the end of the `paddedImgMatrix`. After computing the sum of the multiplications, looping over all the kernel matrix, we save the value in a new matrix called `newImgMatrix`, of size rows x cols.

The last part of the code prints the result matrix for testing purposes.

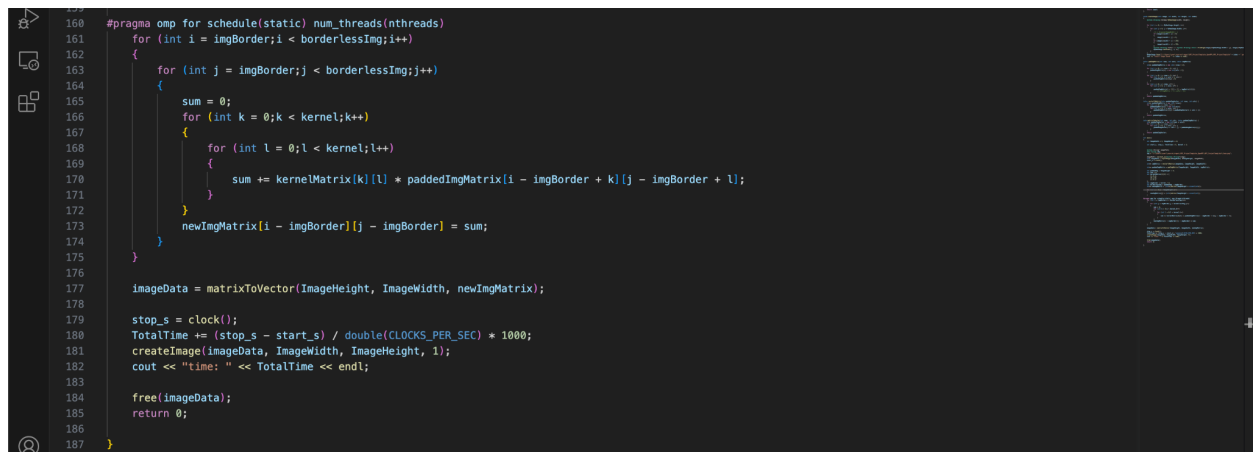
1.2. OpenMP Implementation

This implementation is very similar to the sequential one. A single line is added to allow explicit parallelization using OpenMP via the imported library omp.



```
1 #include <iostream>
2 #include <math.h>
3 #include <msclr\marshal_cppstd.h>
4 #include <ctime> // include this header
5 #include <stdio.h>
6 #include <stdlib.h>
7 #include <omp.h>
8 #pragma once
9
10 #using <mscorlib.dll>
11 #using <System.dll>
12 #using <System.Drawing.dll>
13 #using <System.Windows.Forms.dll>
14 using namespace std;
15 using namespace mscrl::interop;
16
```

All libraries needed for the pr

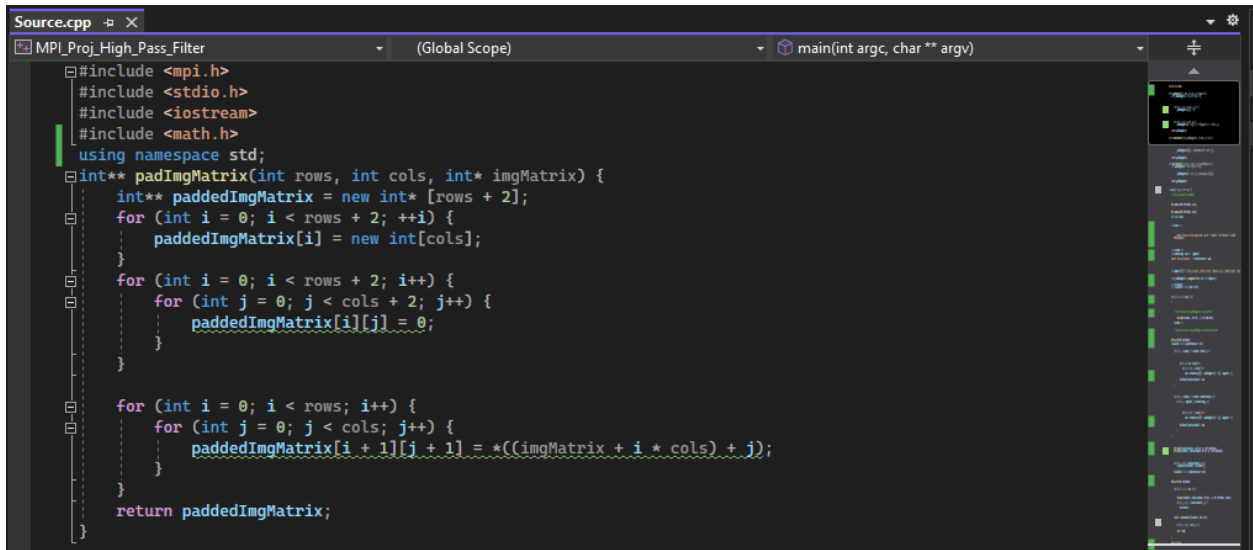


```
168 #pragma omp for schedule(static) num_threads(nthreads)
169 for (int i = imgBorder; i < borderlessImg; i++)
170 {
171     for (int j = imgBorder; j < borderlessImg; j++)
172     {
173         sum = 0;
174         for (int k = 0; k < kernel; k++)
175         {
176             for (int l = 0; l < kernel; l++)
177             {
178                 sum += kernelMatrix[k][l] * paddedImgMatrix[i - imgBorder + k][j - imgBorder + l];
179             }
180             newImgMatrix[i - imgBorder][j - imgBorder] = sum;
181         }
182     }
183
184     imageData = matrixToVector(ImageHeight, ImageWidth, newImgMatrix);
185
186     stop_s = clock();
187     totalTime += (stop_s - start_s) / double(CLOCKS_PER_SEC) * 1000;
188     createImage(imageData, ImageWidth, ImageHeight, 1);
189     cout << "time: " << totalTime << endl;
190
191     free(imageData);
192     return 0;
193 }
```

object are listed above.

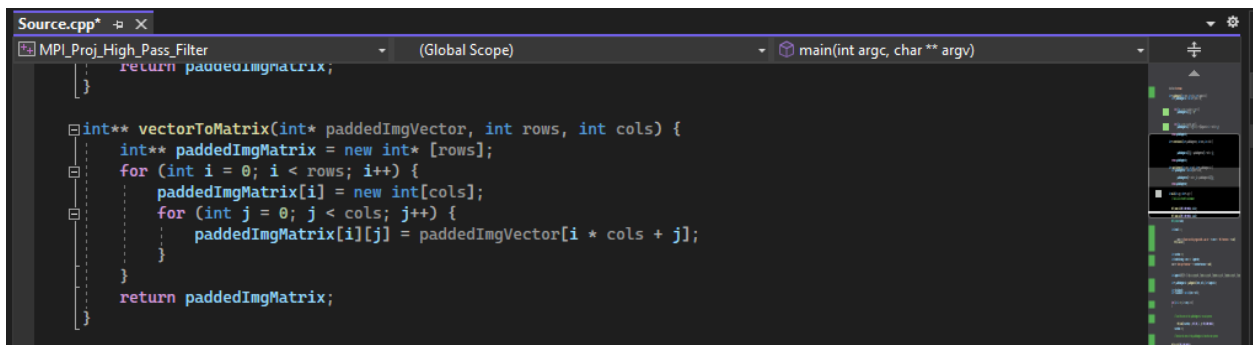
The first line in the code above signals for the parallelization of the consecutive nested for loops through the keywords #pragma omp for. The schedule is set to static, as the computations are equal in balance, without specifying the chunk size, such that the chunk size is approximately equal in size, and the number of threads is passed as well.

1.3. MPI Implementation



```
Source.cpp [X]
MPI_Proj_High_Pass_Filter (Global Scope) main(int argc, char ** argv)
#include <mpi.h>
#include <stdio.h>
#include <iostream>
#include <math.h>
using namespace std;
int** padImgMatrix(int rows, int cols, int* imgMatrix) {
    int** paddedImgMatrix = new int* [rows + 2];
    for (int i = 0; i < rows + 2; ++i) {
        paddedImgMatrix[i] = new int[cols];
    }
    for (int i = 0; i < rows + 2; i++) {
        for (int j = 0; j < cols + 2; j++) {
            paddedImgMatrix[i][j] = 0;
        }
    }
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            paddedImgMatrix[i + 1][j + 1] = *((imgMatrix + i * cols) + j);
        }
    }
    return paddedImgMatrix;
}
```

In the first part of the code, we defined all the libraries that will be used like mpi, stdio, iostream, and math. The first function is used to pad the image matrix. We pass to the function 3 different parameters, the height of the image specified as the row, the width of the image specified as the column, and the pointer that corresponds to the matrix holding the pixels of the image. The first thing is to define a dynamic 2D array with size rows + 2, which corresponds to the height of the image in addition to 2 extra rows that present the padding. After that, we start adding the width of the image to our 2D matrix using the first for loop in the function. Then, we initialize the paddedImgMatrix with 0s. Finally, we copy the values from the imgMatrix to their correct positions in paddedImgMatrix using a nested loop.



```
Source.cpp [X]
MPI_Proj_High_Pass_Filter (Global Scope) main(int argc, char ** argv)
return paddedImgMatrix;
int** vectorToMatrix(int* paddedImgVector, int rows, int cols) {
    int** paddedImgMatrix = new int* [rows];
    for (int i = 0; i < rows; i++) {
        paddedImgMatrix[i] = new int[cols];
        for (int j = 0; j < cols; j++) {
            paddedImgMatrix[i][j] = paddedImgVector[i * cols + j];
        }
    }
    return paddedImgMatrix;
}
```

In the second function, we convert a single array to a 2D array. First, we pass in 3 different parameters just like the previous function. Then we define a dynamic 2D array called paddedImgMatrix just as explained in the previous function. Then for each position in the 2D array, we copy the value of the position $i * \text{cols} + j$. The equation $i * \text{cols} + j$ limits that each row in the paddedImgMatrix, holds only a certain number of values that corresponds to the width of the image (cols).

```

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

    int start_s, stop_s, TotalTime = 0, kernel = 3;

    System::String^ imagePath;
    std::string img;
    img = "C:\\Users\\user\\source\\repos\\HPC_ProjectTemplate_MPI\\HPC_ProjectTemplate\\lena.png";

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

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

    MPI_Status status;

    if (size > ImageHeight) {
        if (rank == 0)
            cout << "Please run this program with a max of " << ImageHeight << " MPI Processes." << endl;
        MPI_Finalize();
        exit(1);
    }
}

```

In this part of the code, we just initialize MPI as mentioned in the slides and then we specify the width and height of the matrix of pixels, cols and rows respectively. Then we created a condition to handle a special case we faced in the implementation in which the number of processors is greater than the height of the image.

```

int sum = 0;
int startRow = 0;
int imgBorder = kernel / 2;
int borderlessImg = ImageHeight + 2 - imgBorder;
int rowsPerProcessor = ImageHeight / size;
cout << "Rows per Processor: " << rowsPerProcessor << endl;
int sizeofLocalResult;
int resultIndex = 0;
int rowsLeft = ImageHeight - rowsPerProcessor * (size - 1);
int kernelMatrix[3][3] = {{0, -1, 0},{-1, 4, -1},{0, -1, 0}};

int** imgMatrix = vectorToMatrix(imageData, ImageHeight, ImageWidth);
int** paddedImgMatrix = padImgMatrix(ImageHeight, ImageWidth, imgMatrix);

int* localResult;
int* finalResult = new int[ImageHeight * ImageWidth];
int* rcvdResult;
int** result = new int* [ImageHeight];

for (int i = 0; i < ImageHeight; i++) {
    result[i] = new int[ImageWidth];
}

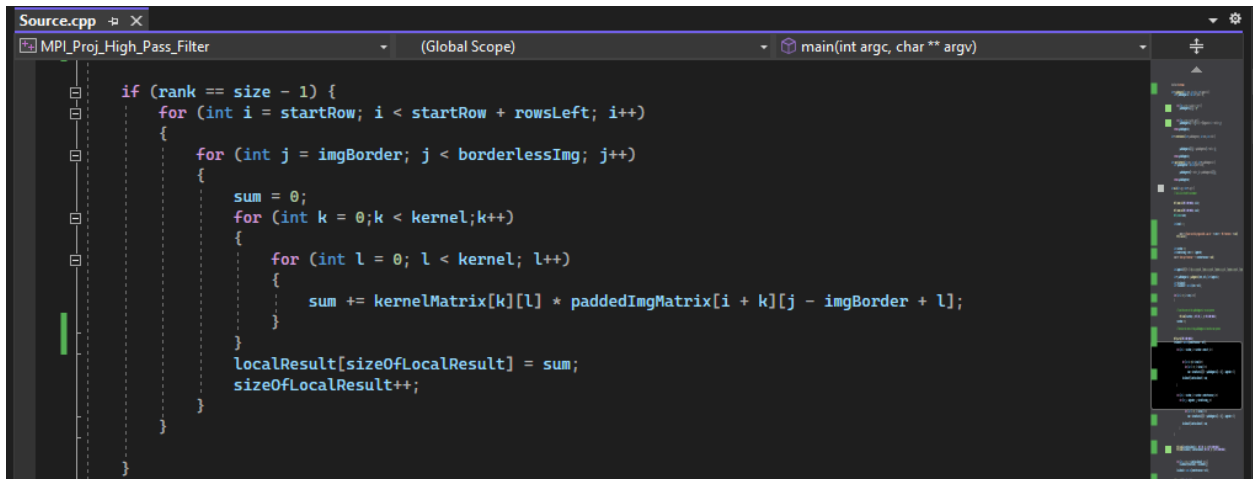
```

In the second part of the code, we declared and initialized all the variables that will be used in our code. The most important variables are sum, imgBorder, borderlessImg, rowsPerProcessor, sizeofLocalResult, and rowsLeft.

- sum = a dummy variable that is used to hold the values of the multiplication that will be explained below.
- imgBorder = this variable is a limit that will be used below in the matrix multiplication to neglect the rows and cols added in the padding function from the top of the matrix and start the multiplication from the pixels of the image.

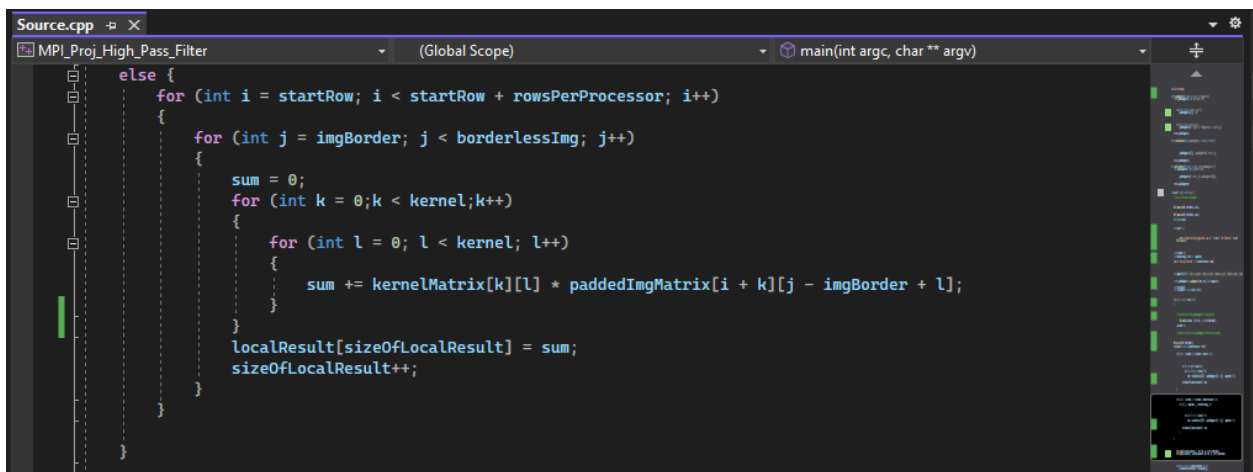
- `borderlessImg` = this variable is another limit that will be used below in the matrix multiplication to neglect the rows and the cols added in the padding function from the bottom of the matrix and end the multiplication at the last pixel of the image.
- `rowsPerProcessor` = divide the height of the image on the number of processors to specify the number of rows that should be calculated by each processor.
- `sizeofLocalResult` = a variable that will be used to specify the size of the array that will hold the result of the matrix multiplication calculated by each process.
- `rowsLeft` = a variable that specifies the number of rows left from the matrix of pixels that needs to be calculated. This variable is used to handle the special case when the number of rows is indivisible by the total number of processors. `rowsLeft` is calculated by subtracting the number of rows from `rowsPerProcessor * size - 1`. The `rowsPerProcessor` is multiplied by `size - 1` since we want to know how many rows were calculated by all the processors so we can give the remaining rows to the last processor.

In the third section of the main function, we check in a condition if the rank of the processor is 0 in order to start sending to each processor the index at which it will start applying the matrix multiplication. It is important to note that we need to make `startRow = 0` to prevent other processors from having the same starting index. If the rank is anything other than 0, then the process will receive the `startRow`.



```
Source.cpp x
MPI_Proj_High_Pass_Filter (Global Scope) main(int argc, char ** argv)

if (rank == size - 1) {
    for (int i = startRow; i < startRow + rowsLeft; i++)
    {
        for (int j = imgBorder; j < borderlessImg; j++)
        {
            sum = 0;
            for (int k = 0; k < kernel; k++)
            {
                for (int l = 0; l < kernel; l++)
                {
                    sum += kernelMatrix[k][l] * paddedImgMatrix[i + k][j - imgBorder + l];
                }
            }
            localResult[sizeOfLocalResult] = sum;
            sizeOfLocalResult++;
        }
    }
}
```



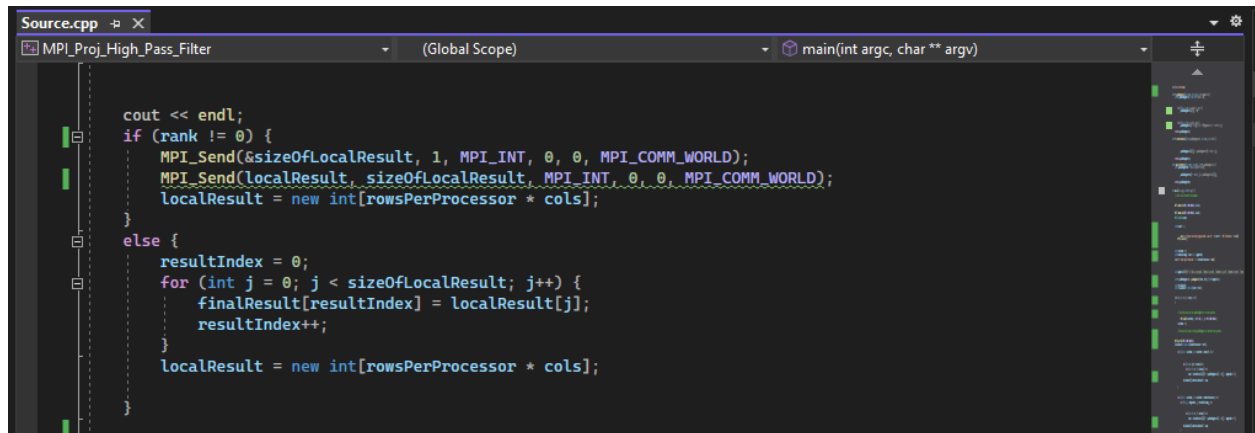
```
Source.cpp x
MPI_Proj_High_Pass_Filter (Global Scope) main(int argc, char ** argv)

else {
    for (int i = startRow; i < startRow + rowsPerProcessor; i++)
    {
        for (int j = imgBorder; j < borderlessImg; j++)
        {
            sum = 0;
            for (int k = 0; k < kernel; k++)
            {
                for (int l = 0; l < kernel; l++)
                {
                    sum += kernelMatrix[k][l] * paddedImgMatrix[i + k][j - imgBorder + l];
                }
            }
            localResult[sizeOfLocalResult] = sum;
            sizeOfLocalResult++;
        }
    }
}
```

This is the most important section in the main function because it applies the matrix multiplication which is the main functionality of the low pass filter. First of all, we check if this processor is the last processor, and if it is we loop on the number of the rows specified for each processor which is indicated by the startRow variable and the limit is startRow + rowsLeft since it is the last processor we need to start with the startRow and end at the end of the matrix which will be specified by the rowsLeft + the startRow. The second for loop starts from the first row after the border we added using the padding function and ends before the last row, which is also added due to the padding. This for loop also applies the concept of the sliding window in which we compute the multiplication on each row then move the index i by 1. The third for loop is for the height of the kernel and the final for loop is responsible for the width of the kernel. Finally, we apply the multiplication by multiplying the kernel by its corresponding position from the paddedImgMatrix.

Second of all, if the processor is not the last processor, we loop on the number of the rows specified for each processor which is indicated by the startRow variable and the limit is startRow + rowsPerProcessor since this is not the last processor we need to start with the startRows and end at the end of the rowsPerProcessor. The rest of the for loops are explained as above.

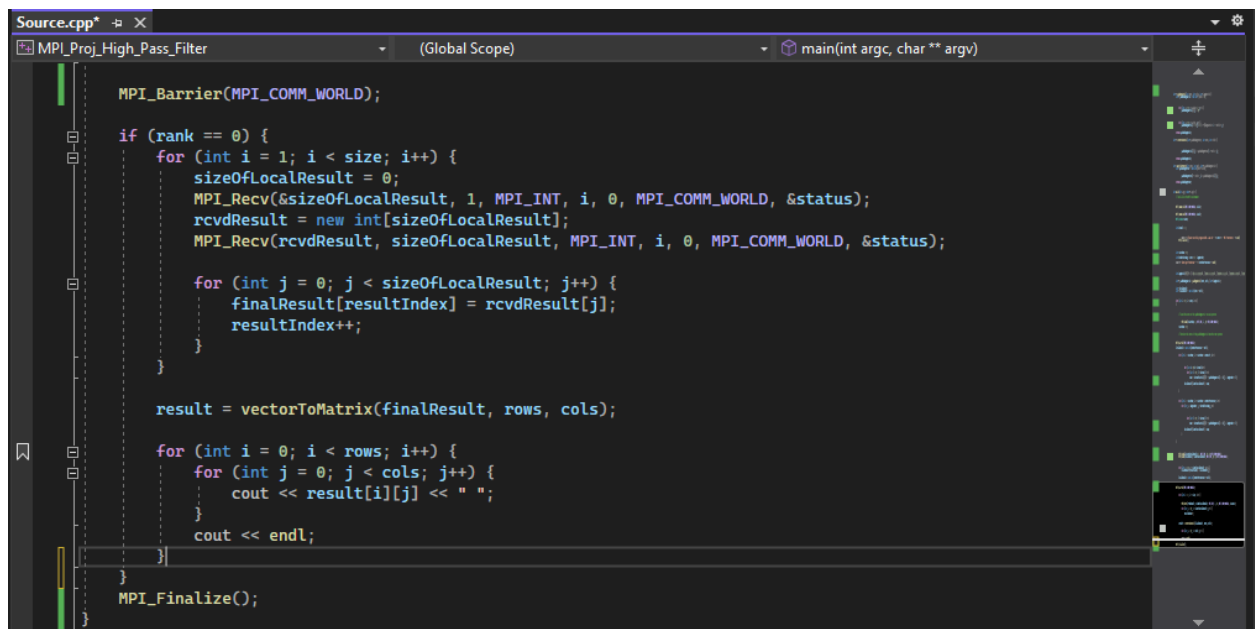
Finally, we put the values of the multiplication in the localResult array and we increment the sizeofLocalResult.



```
Source.cpp
MPI_Proj_High_Pass_Filter (Global Scope) main(int argc, char ** argv)

cout << endl;
if (rank != 0) {
    MPI_Send(&sizeofLocalResult, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
    MPI_Send(localResult, sizeofLocalResult, MPI_INT, 0, 0, MPI_COMM_WORLD);
    localResult = new int[rowsPerProcessor * cols];
}
else {
    resultIndex = 0;
    for (int j = 0; j < sizeofLocalResult; j++) {
        finalResult[resultIndex] = localResult[j];
        resultIndex++;
    }
    localResult = new int[rowsPerProcessor * cols];
}
```

In the fourth section of the main function, we check if the rank of the processor is not equal to zero and if it is we use MPI_Send to send the localResult array of each processor to the first processor. If the rank is equal to 0, we just place the values of its localResult inside the finalResult array.



```
Source.cpp
MPI_Proj_High_Pass_Filter (Global Scope) main(int argc, char ** argv)

MPI_Barrier(MPI_COMM_WORLD);

if (rank == 0) {
    for (int i = 1; i < size; i++) {
        sizeofLocalResult = 0;
        MPI_Recv(&sizeofLocalResult, 1, MPI_INT, i, 0, MPI_COMM_WORLD, &status);
        rcvdResult = new int[sizeofLocalResult];
        MPI_Recv(rcvdResult, sizeofLocalResult, MPI_INT, i, 0, MPI_COMM_WORLD, &status);

        for (int j = 0; j < sizeofLocalResult; j++) {
            finalResult[resultIndex] = rcvdResult[j];
            resultIndex++;
        }
    }

    result = vectorToMatrix(finalResult, rows, cols);

    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            cout << result[i][j] << " ";
        }
        cout << endl;
    }
}

MPI_Finalize();
```

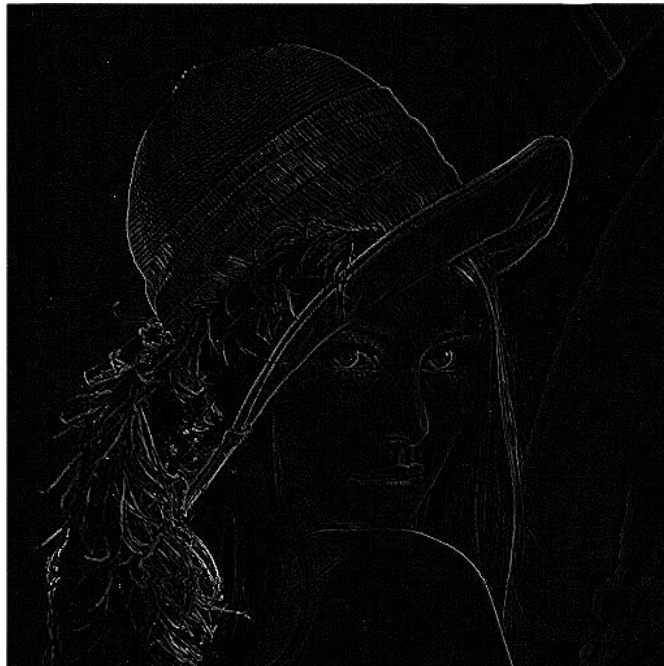
Finally, we check if rank is equal to 0, we use MPI_Recv to receive the localResult of each processor and also the size of the localResult array of each processor. Then we place the values of the received values from each processor in the finalResult array. Then we use the vectorToMatrix function to convert the finalResult array to a matrix. Then print the result matrix and use MPI_Finalize to clean all states related to MPI.

2.0. Input and Output

Input:



Output:

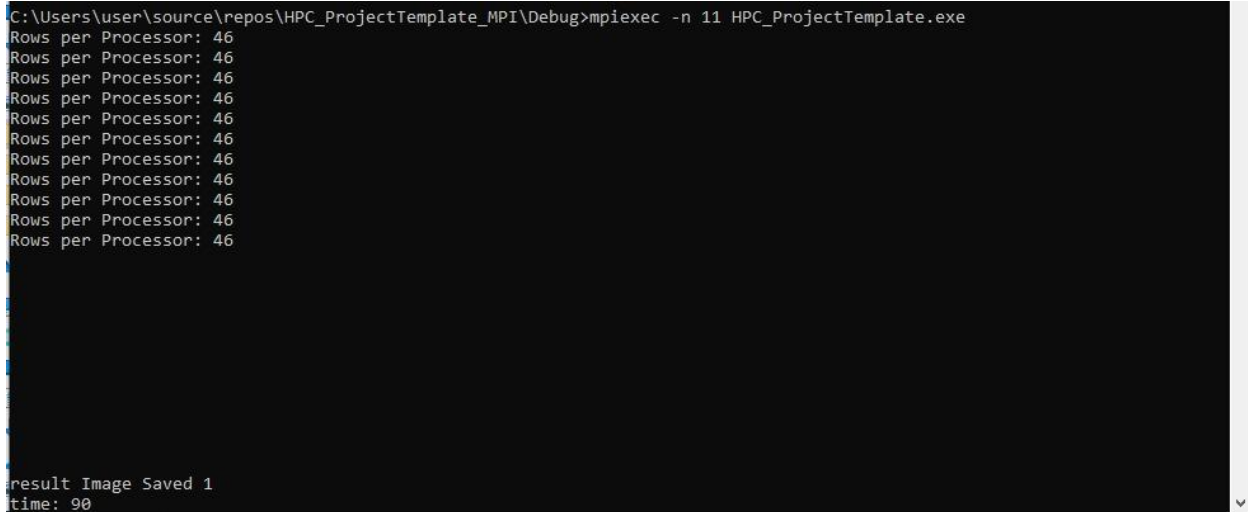


3.0. Time Comparison

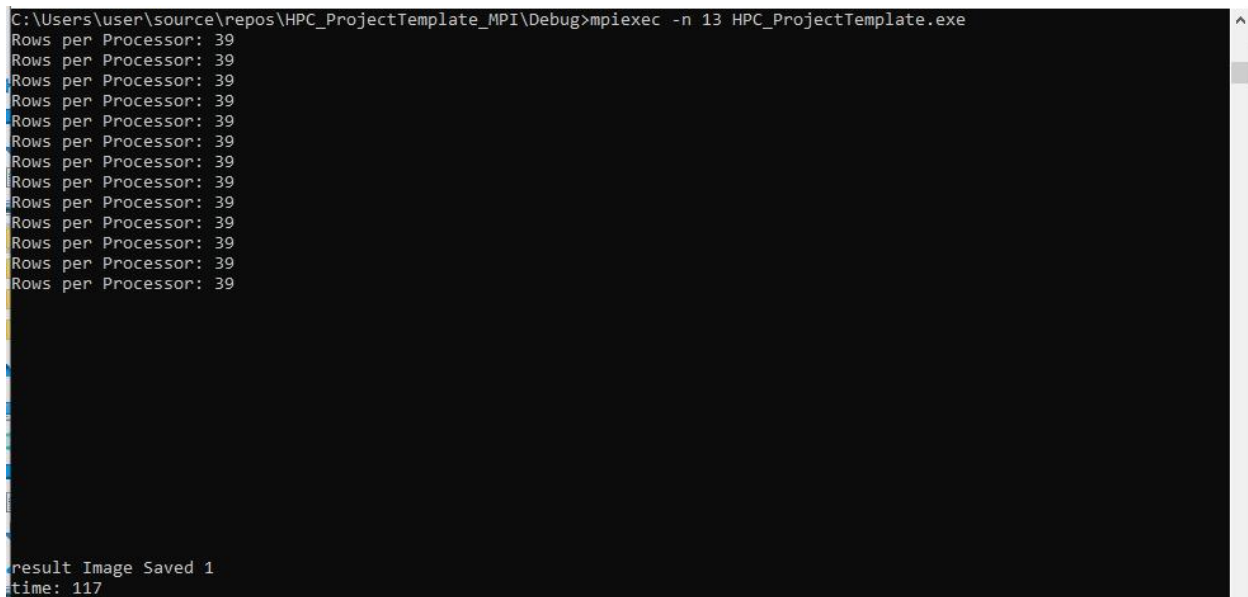


```
C:\Users\user\source\repos\HPC_ProjectTemplate_OpenMP\Debug\HPC_ProjectTemplate.exe
result Image Saved 1
time: 18
```

OpenMP execution time is 18 milliseconds. Meanwhile, MPI execution depends on the number of processes used:



```
C:\Users\user\source\repos\HPC_ProjectTemplate_MPI\Debug>mpiexec -n 11 HPC_ProjectTemplate.exe
Rows per Processor: 46
Rows per Processor: 46
Rows per Processor: 46
Rows per Processor: 46
Rows per Processor: 46
Rows per Processor: 46
Rows per Processor: 46
Rows per Processor: 46
Rows per Processor: 46
Rows per Processor: 46
Rows per Processor: 46
Rows per Processor: 46
result Image Saved 1
time: 90
```



```
C:\Users\user\source\repos\HPC_ProjectTemplate_MPI\Debug>mpiexec -n 13 HPC_ProjectTemplate.exe
Rows per Processor: 39
Rows per Processor: 39
Rows per Processor: 39
Rows per Processor: 39
Rows per Processor: 39
Rows per Processor: 39
Rows per Processor: 39
Rows per Processor: 39
Rows per Processor: 39
Rows per Processor: 39
Rows per Processor: 39
Rows per Processor: 39
Rows per Processor: 39
result Image Saved 1
time: 117
```

Conclusion:

The execution time with MPI is worse than OpenMP, which is very reasonable. The setup we are running on is a single computer, with shared memory, hence, it is more suitable to use OpenMP, decreasing communication delay and leading to a better performance and minimizing the execution time. The setup we use here does not require MPI since we always use a single computer or node, making its simulation result in a worse performance. Needless to mention that this won't be the case with a different setup that includes multiple computers.