1. Introduction

In this assignment, we implemented a parallelized version of the box blur convolution operator applied to a grayscale image. The goal was to leverage MPI (Message Passing Interface) to distribute the computational workload across multiple processors, thereby reducing the total runtime.

2. Problem Statement

The task is to apply a box blur to an image using parallel processing techniques. A box blur is a simple image processing operation where each pixel's value is replaced by the average of the pixel values in its neighborhood. The convolution operator needs to be parallelized to handle large images efficiently.

3. Implementation Details

3.1. Initial Setup

The initial code provided includes functions to read an image into a matrix (`imageToMat`) and to save a matrix back to an image (`matToImage`). The main program initializes MPI and distributes the workload across available processors.

3.2. MPI Initialization and Image Reading

```
int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);

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

    int *matrix = NULL, *temp = NULL;
    char *name = "image.jpg";
    int *dims = (int*) malloc(2 * sizeof(int));

    if (rank == 0) {
        // Read image only in the root process
        matrix = imageToMat(name, dims);
    }

    // Broadcast the image dimensions to all processes
    MPI_Bcast(dims, 2, MPI_INT, 0, MPI_COMM_WORLD);
```

The root process reads the image and broadcasts its dimensions to all other processes.

3.3. Workload Distribution

```
// Allocate memory for the local and temp matrices
int local_height = height / size + (rank < height % size ? 1 : 0);
int *local_matrix = (int*) malloc(local_height * width * sizeof(int));
int *local_temp = (int*) malloc(local_height * width * sizeof(int));

// Scatter the matrix rows to all processes
int *sendcounts = (int*) malloc(size * sizeof(int));
int *displs = (int*) malloc(size * sizeof(int));
int sum = 0;

for (int i = 0; i < size; i++) {
    sendcounts[i] = (height / size + (i < height % size ? 1 : 0)) * width;
    displs[i] = sum;
    sum += sendcounts[i];
}

MPI_Scatterv(matrix, sendcounts, displs, MPI_INT, local_matrix, local_height * width, MPI_INT, 0, MPI_COMM_WORLD);
```

The image matrix is divided into rows, and these rows are distributed among the processors. Each processor receives a contiguous block of rows.

3.4. Convolution Operation

```
// Perform the convolution on the local matrix
for (int i = 0; i < local_height; i++) {
    for (int j = 0; j < width; j++) {
        int index = i * width + j;
        int sum = 0;
        int counter = 0;
        for (int u = -k; u <= k; u++) {
            for (int v = -k; v <= k; v++) {
                int global_i = global_i_offset + i - u;
                int global_j = j - v;

                if (global_i < 0 || global_i >= height || global_j < 0 || global_j >= width) {
                    continue;
                }
                // Convert global indices to local indices for the current process
                int local_i = global_i - global_i_offset;
                if (local_i >= 0 && local_i < local_height) {
                    int cindex = local_i * width + global_j;
                    sum += local_matrix[cindex];
                    counter++;
                }
            }
        }
        local_temp[index] = sum / counter;
    }
}
```

Each processor performs the box blur convolution on its assigned submatrix.

3.5. Gathering Results

```
// Gather the results back to the root process
if (rank == 0) {
    temp = (int*) malloc(height * width * sizeof(int));
}

MPI_Gatherv(local_temp, local_height * width, MPI_INT, temp, sendcounts, displs, MPI_INT, 0, MPI_COMM_WORLD);

if (rank == 0) {
    // Save the image in the root process
    matToImage("processedImage.jpg", temp, dims);
    free(matrix);
    free(temp);
}

free(dims);
free(local_matrix);
free(local_temp);
free(sendcounts);
free(displs);

MPI_Finalize();

return 0;
```

The processed submatrices are gathered back to the root processor.

4. Results

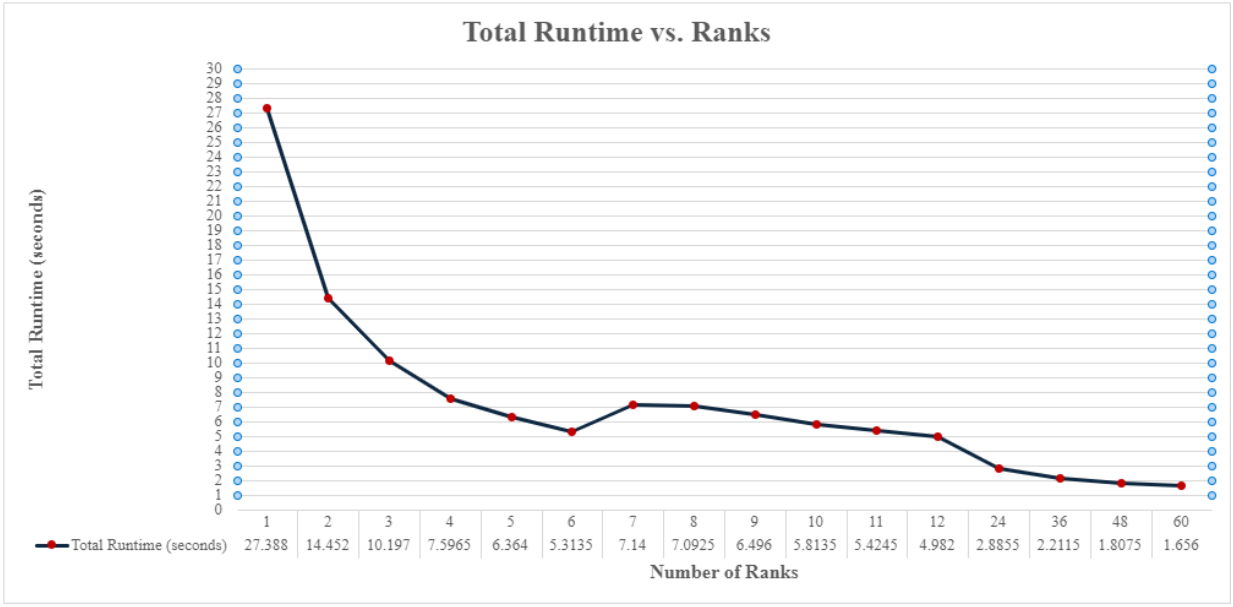
Original Image (3840 x 2160px):

Box Blurred Image (kernel = 10):

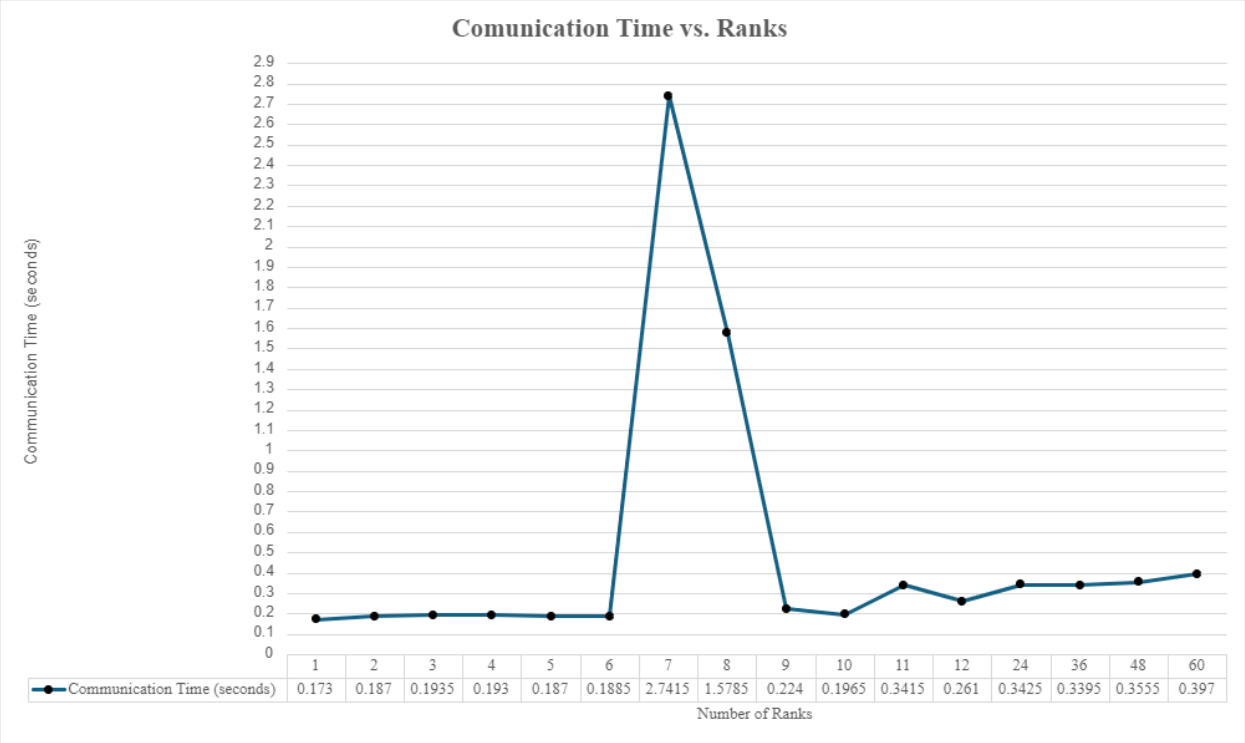
Graphs:

The program was tested with varying numbers of processors. The total runtime and communication time for each run were recorded and are presented in the graphs below.

Total Runtime vs. Number of Ranks:



Communication Time vs. Number of Ranks:



	1	2	3	4	5	6	7	8	9	10	11	12	24	36	48	60
Communication Time (seconds)	0.173	0.187	0.1935	0.193	0.187	0.1885	2.7415	1.5785	0.224	0.1965	0.3415	0.261	0.3425	0.3395	0.3555	0.397