SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

WINTER SEMESTER – 2022

Image Convolution using Parallel Programming Techniques

A Report

submitted by

VISHAKHA KUMARESAN

19BCE2678

CSE4001 – Parallel and Distributed Computing – J Component

E1 Slot

Faculty Name – Sivakumar N.

TABLE OF CONTENTS

1. INTRODUCTION

Convolution is the process of adding each element of the image to its local neighbors, weighted by the kernel. This is related to a form of mathematical convolution. The matrix operation being performed—convolution—is not traditional matrix multiplication, despite being similarly denoted by *.

In image processing, a kernel, convolution matrix, or mask is a small matrix used for blurring, sharpening, embossing, edge detection, and more. This is accomplished by doing a convolution between the kernel and an image.

For Box Blur,

$$\frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

For Gaussian Blur,

$$\frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

For Sharpen,

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

2. PROPOSED WORK

The problem of image convergence parallelism requires careful planning and understanding of each component of the element. A direct approach, where we proceed to the implementation stage without the proper design of the parallel algorithm, leads to the

deterministic failure of the project on important issues such as scaling and execution time. Therefore, a systematic approach to our problem is necessary.

In this direction there is the Foster methodology, which defines four stages of design. Namely the stage of partitioning, communication, agglomeration and mapping.

1. Partitioning

   At this stage we try to export ways of parallelizing the execution. A good partition divides into small pieces both the calculations and the data of the problem. So, we divide the image into smaller images depending on the number of processes given. According to the methodology, in the first stages we have to choose more aggressive partitions.

   More specifically, at the start of the program the user selects the number of processes for which the program will be executed. Based on an algorithm (which is described in the divide_rows function, the optimal way in which the table will be distributed between the processes is calculated).

2. Communication

   As already mentioned the processes are distributed in the grid in such a way that they are next to each other in ascending order. This facilitates any process in finding its neighboring processes. A special case are the processes located in the periphery of the grid. In this case, adjacent processes are those that are on the appropriate off-limits opposite side.

   Proceeding now to code issues as an option we have decided to use nonblocking sending and receiving messages thus being able to compile the central parts of the table (inner data) and leaving for later the calculation of peripheral pieces (outer and corner data). This allows a process to be busy with a task while waiting for the adjacent cells in the table to arrive. When this task is completed, if the data it has been waiting for has arrived, it proceeds to perform the required new tasks, otherwise it is still waiting to receive the required data.

   For sending and receiving we decided to do with the help of Datatypes (vector, contiguous) data where possible (up, right, down, left) to avoid various risks and better manage them.

3. Agglomeration

4. Mapping

Algorithm for Convolution

```
for each image row in input image:
    for each pixel in image row:

        set accumulator to zero

        for each kernel row in kernel:
            for each element in kernel row:

                if element position  corresponding* to pixel position then
                    multiply element value  corresponding* to pixel value
                    add result to accumulator
                endif

         set output image pixel to accumulator
```

Flow of code:

In general, our implementation followed the following pattern:

for loop

ISend

IRecv

Inner Data Computations

Wait(RRecv)

Outer Data Computations

Wait(RSend)

end for

3. CODE & IMPLEMENTATION

Image Convolution using MPI

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <stdint.h>
#include "mpi.h"
```

```
typedef enum {RGB, GREY} color_t;

void convolute(uint8_t *, uint8_t *, int, int, int, int, int, int, float**, color_t);
void convolute_grey(uint8_t *, uint8_t *, int, int, int, int, float **);
void convolute_rgb(uint8_t *, uint8_t *, int, int, int, int, float **);
void Usage(int, char **, char **, int *, int *, int *, color_t *);
uint8_t *offset(uint8_t *, int, int, int);
int divide_rows(int, int, int);


int main(int argc, char** argv) {
        int fd, i, j, k, width, height, loops, t, row_div, col_div, rows, cols;
        double timer, remote_time;
        char *image;
        color_t imageType;
        /* MPI world topology */
   int process_id, num_processes;
        /* Find current task id */
   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &num_processes);
   MPI_Comm_rank(MPI_COMM_WORLD, &process_id);
        /* MPI status */
   MPI_Status status;
        /* MPI data types */
   MPI_Datatype grey_col_type;
   MPI_Datatype rgb_col_type;
   MPI_Datatype grey_row_type;
   MPI_Datatype rgb_row_type;
        /* MPI requests */
   MPI_Request send_north_req;
   MPI_Request send_south_req;
   MPI_Request send_west_req;
   MPI_Request send_east_req;
   MPI_Request recv_north_req;
   MPI_Request recv_south_req;
   MPI_Request recv_west_req;
   MPI_Request recv_east_req;

        /* Neighbours */
        int north = -1;
        int south = -1;
   int west = -1;
   int east = -1;

   /* Check arguments */
   if (process_id == 0) {
```

```
                Usage(argc, argv, &image, &width, &height, &loops, &imageType);
                /* Division of data in each process */
                row_div = divide_rows(height, width, num_processes);
                if (row_div <= 0 || height % row_div || num_processes % row_div || width
% (col_div = num_processes / row_div)) {
                        fprintf(stderr, "%s: Cannot divide to processes\n", argv[0]);
                        MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
                        return EXIT_FAILURE;
                }
        }
        if (process_id != 0) {
                image = malloc((strlen(argv[1])+1) * sizeof(char));
                strcpy(image, argv[1]);
        }
        /* Broadcast parameters */
    MPI_Bcast(&width, 1, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Bcast(&height, 1, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Bcast(&loops, 1, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Bcast(&imageType, 1, MPI_INT, 0, MPI_COMM_WORLD);
        MPI_Bcast(&row_div, 1, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Bcast(&col_div, 1, MPI_INT, 0, MPI_COMM_WORLD);

        /* Compute number of rows per process */
        rows = height / row_div;
        cols = width / col_div;

        /* Create column data type for grey & rgb */
        MPI_Type_vector(rows, 1, cols+2, MPI_BYTE, &grey_col_type);
        MPI_Type_commit(&grey_col_type);
        MPI_Type_vector(rows, 3, 3*cols+6, MPI_BYTE, &rgb_col_type);
        MPI_Type_commit(&rgb_col_type);
        /* Create row data type */
        MPI_Type_contiguous(cols, MPI_BYTE, &grey_row_type);
        MPI_Type_commit(&grey_row_type);
        MPI_Type_contiguous(3*cols, MPI_BYTE, &rgb_row_type);
        MPI_Type_commit(&rgb_row_type);

         /* Compute starting row and column */
    int start_row = (process_id / col_div) * rows;
    int start_col = (process_id % col_div) * cols;

        /* Init filters */
        int box_blur[3][3] = {{1, 1, 1}, {1, 1, 1}, {1, 1, 1}};
        int gaussian_blur[3][3] = {{1, 2, 1}, {2, 4, 2}, {1, 2, 1}};
        int edge_detection[3][3] = {{1, 4, 1}, {4, 8, 4}, {1, 4, 1}};
        float **h = malloc(3 * sizeof(float *));
```

```
	for (i = 0 ; i < 3 ; i++)
		h[i] = malloc(3 * sizeof(float));
	for (i = 0 ; i < 3 ; i++) {
		for (j = 0 ; j < 3 ; j++){
			// h[i][j] = box_blur[i][j] / 9.0;
			h[i][j] = gaussian_blur[i][j] / 16.0;
			// h[i][j] = edge_detection[i][j] / 28.0;
		}
	}

	/* Init arrays */
	uint8_t *src = NULL, *dst = NULL, *tmpbuf = NULL, *tmp = NULL;
	MPI_File fh;
	int filesize, bufsize, nbytes;
	if (imageType == GREY) {
		filesize = width * height;
		bufsize = filesize / num_processes;
		nbytes = bufsize / sizeof(uint8_t);
		src = calloc((rows+2) * (cols+2), sizeof(uint8_t));
		dst = calloc((rows+2) * (cols+2), sizeof(uint8_t));
	} else if (imageType == RGB) {
		filesize = width*3 * height;
		bufsize = filesize / num_processes;
		nbytes = bufsize / sizeof(uint8_t);
		src = calloc((rows+2) * (cols*3+6), sizeof(uint8_t));
		dst = calloc((rows+2) * (cols*3+6), sizeof(uint8_t));
	}
	  if (src == NULL || dst == NULL) {
	fprintf(stderr, "%s: Not enough memory\n", argv[0]);
	MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
	return EXIT_FAILURE;
	  }

	  /* Parallel read */
	  MPI_File_open(MPI_COMM_WORLD, image, MPI_MODE_RDONLY,
MPI_INFO_NULL, &fh);
	if (imageType == GREY) {
		for (i = 1 ; i <= rows ; i++) {
			MPI_File_seek(fh, (start_row + i-1) * width + start_col,
MPI_SEEK_SET);
			tmpbuf = offset(src, i, 1, cols+2);
			MPI_File_read(fh, tmpbuf, cols, MPI_BYTE, &status);
		}
	} else if (imageType == RGB) {
		for (i = 1 ; i <= rows ; i++) {
```

```
                        MPI_File_seek(fh, 3*(start_row + i-1) * width + 3*start_col,
MPI_SEEK_SET);
                        tmpbuf = offset(src, i, 3, cols*3+6);
                        MPI_File_read(fh, tmpbuf, cols*3, MPI_BYTE, &status);
                }
        }
        MPI_File_close(&fh);

        /* Compute neighbours */
   if (start_row != 0)
      north = process_id - col_div;
   if (start_row + rows != height)
      south = process_id + col_div;
   if (start_col != 0)
      west = process_id - 1;
   if (start_col + cols != width)
      east = process_id + 1;

        MPI_Barrier(MPI_COMM_WORLD);

        /* Get time before */
   timer = MPI_Wtime();
        /* Convolute "loops" times */
        for (t = 0 ; t < loops ; t++) {
      /* Send and request borders */
                if (imageType == GREY) {
                        if (north != -1) {
                                MPI_Isend(offset(src, 1, 1, cols+2), 1, grey_row_type,
north, 0, MPI_COMM_WORLD, &send_north_req);
                                MPI_Irecv(offset(src, 0, 1, cols+2), 1, grey_row_type,
north, 0, MPI_COMM_WORLD, &recv_north_req);
                        }
                        if (west != -1) {
                                MPI_Isend(offset(src, 1, 1, cols+2), 1, grey_col_type,
west, 0, MPI_COMM_WORLD, &send_west_req);
                                MPI_Irecv(offset(src, 1, 0, cols+2), 1, grey_col_type,
west, 0, MPI_COMM_WORLD, &recv_west_req);
                        }
                        if (south != -1) {
                                MPI_Isend(offset(src, rows, 1, cols+2), 1, grey_row_type,
south, 0, MPI_COMM_WORLD, &send_south_req);
                                MPI_Irecv(offset(src, rows+1, 1, cols+2), 1,
grey_row_type, south, 0, MPI_COMM_WORLD, &recv_south_req);
                        }
                        if (east != -1) {
```

```
                                MPI_Isend(offset(src, 1, cols, cols+2), 1, grey_col_type,
east, 0, MPI_COMM_WORLD, &send_east_req);
                                MPI_Irecv(offset(src, 1, cols+1, cols+2), 1, grey_col_type,
east, 0, MPI_COMM_WORLD, &recv_east_req);
                        }
                } else if (imageType == RGB) {
                        if (north != -1) {
                                MPI_Isend(offset(src, 1, 3, 3*cols+6), 1, rgb_row_type,
north, 0, MPI_COMM_WORLD, &send_north_req);
                                MPI_Irecv(offset(src, 0, 3, 3*cols+6), 1, rgb_row_type,
north, 0, MPI_COMM_WORLD, &recv_north_req);
                        }
                        if (west != -1) {
                                MPI_Isend(offset(src, 1, 3, 3*cols+6), 1, rgb_col_type,
west, 0, MPI_COMM_WORLD, &send_west_req);
                                MPI_Irecv(offset(src, 1, 0, 3*cols+6), 1, rgb_col_type,
west, 0, MPI_COMM_WORLD, &recv_west_req);
                        }
                        if (south != -1) {
                                MPI_Isend(offset(src, rows, 3, 3*cols+6), 1, rgb_row_type,
south, 0, MPI_COMM_WORLD, &send_south_req);
                                MPI_Irecv(offset(src, rows+1, 3, 3*cols+6), 1,
rgb_row_type, south, 0, MPI_COMM_WORLD, &recv_south_req);
                        }
                        if (east != -1) {
                                MPI_Isend(offset(src, 1, 3*cols, 3*cols+6), 1,
rgb_col_type,  east, 0, MPI_COMM_WORLD, &send_east_req);
                                MPI_Irecv(offset(src, 1, 3*cols+3, 3*cols+6), 1,
rgb_col_type,  east, 0, MPI_COMM_WORLD, &recv_east_req);
                        }
                }

                /* Inner Data Convolute */
                convolute(src, dst, 1, rows, 1, cols, cols, rows, h, imageType);


        /* Request and compute */
                if (north != -1) {
                        MPI_Wait(&recv_north_req, &status);
                        convolute(src, dst, 1, 1, 2, cols-1, cols, rows, h, imageType);
                }
                if (west != -1) {
                        MPI_Wait(&recv_west_req, &status);
                        convolute(src, dst, 2, rows-1, 1, 1, cols, rows, h, imageType);
                }
                if (south != -1) {
```

```
                        MPI_Wait(&recv_south_req, &status);
                        convolute(src, dst, rows, rows, 2, cols-1, cols, rows, h,
imageType);
                }
                if (east != -1) {
                        MPI_Wait(&recv_east_req, &status);
                        convolute(src, dst, 2, rows-1, cols, cols, cols, rows, h, imageType);
                }

                /* Corner data */
                if (north != -1 && west != -1)
                        convolute(src, dst, 1, 1, 1, 1, cols, rows, h, imageType);
                if (west != -1 && south != -1)
                        convolute(src, dst, rows, rows, 1, 1, cols, rows, h, imageType);
                if (south != -1 && east != -1)
                        convolute(src, dst, rows, rows, cols, cols, cols, rows, h,
imageType);
                if (east != -1 && north != -1)
                        convolute(src, dst, 1, 1, cols, cols, cols, rows, h, imageType);

                /* Wait to have sent all borders */
                if (north != -1)
                        MPI_Wait(&send_north_req, &status);
                if (west != -1)
                        MPI_Wait(&send_west_req, &status);
                if (south != -1)
                        MPI_Wait(&send_south_req, &status);
                if (east != -1)
                        MPI_Wait(&send_east_req, &status);

                /* swap arrays */
                tmp = src;
        src = dst;
        dst = tmp;
           }
           /* Get time elapsed */
     timer = MPI_Wtime() - timer;

           /* Parallel write */
           char *outImage = malloc((strlen(image) + 9) * sizeof(char));
           strcpy(outImage, "blur_");
           strcat(outImage, image);
           MPI_File outFile;
           MPI_File_open(MPI_COMM_WORLD, outImage, MPI_MODE_CREATE |
MPI_MODE_WRONLY, MPI_INFO_NULL, &outFile);
           if (imageType == GREY) {
```

```
                for (i = 1 ; i <= rows ; i++) {
                        MPI_File_seek(outFile, (start_row + i-1) * width + start_col,
MPI_SEEK_SET);
                        tmpbuf = offset(src, i, 1, cols+2);
                        MPI_File_write(outFile, tmpbuf, cols, MPI_BYTE,
MPI_STATUS_IGNORE);
                }
        } else if (imageType == RGB) {
                for (i = 1 ; i <= rows ; i++) {
                        MPI_File_seek(outFile, 3*(start_row + i-1) * width + 3*start_col,
MPI_SEEK_SET);
                        tmpbuf = offset(src, i, 3, cols*3+6);
                        MPI_File_write(outFile, tmpbuf, cols*3, MPI_BYTE,
MPI_STATUS_IGNORE);
                }
        }
        MPI_File_close(&outFile);

        /* Get times from other processes and print maximum */
   if (process_id != 0)
      MPI_Send(&timer, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
   else {
      for (i = 1 ; i != num_processes ; ++i) {
         MPI_Recv(&remote_time, 1, MPI_DOUBLE, i, 0, MPI_COMM_WORLD,
&status);
         if (remote_time > timer)
            timer = remote_time;
      }
      printf("%f\n", timer);
   }

   /* De-allocate space */
   free(src);
   free(dst);
   MPI_Type_free(&rgb_col_type);
   MPI_Type_free(&rgb_row_type);
   MPI_Type_free(&grey_col_type);
   MPI_Type_free(&grey_row_type);


        /* Finalize and exit */
   MPI_Finalize();
        return EXIT_SUCCESS;
}
```

```
void convolute(uint8_t *src, uint8_t *dst, int row_from, int row_to, int col_from, int
col_to, int width, int height, float** h, color_t imageType) {
	int i, j;
	if (imageType == GREY) {
		for (i = row_from ; i <= row_to ; i++)
			for (j = col_from ; j <= col_to ; j++)
				convolute_grey(src, dst, i, j, width+2, height, h);
	} else if (imageType == RGB) {
		for (i = row_from ; i <= row_to ; i++)
			for (j = col_from ; j <= col_to ; j++)
				convolute_rgb(src, dst, i, j*3, width*3+6, height, h);
	}
}

void convolute_grey(uint8_t *src, uint8_t *dst, int x, int y, int width, int height, float** h)
{
	int i, j, k, l;
	float val = 0;
	for (i = x-1, k = 0 ; i <= x+1 ; i++, k++)
		for (j = y-1, l = 0 ; j <= y+1 ; j++, l++)
			val += src[width * i + j] * h[k][l];
	dst[width * x + y] = val;
}

void convolute_rgb(uint8_t *src, uint8_t *dst, int x, int y, int width, int height, float** h)
{
	int i, j, k, l;
	float redval = 0, greenval = 0, blueval = 0;
	for (i = x-1, k = 0 ; i <= x+1 ; i++, k++)
		for (j = y-3, l = 0 ; j <= y+3 ; j+=3, l++){
			redval += src[width * i + j]* h[k][l];
			greenval += src[width * i + j+1] * h[k][l];
			blueval += src[width * i + j+2] * h[k][l];
		}
	dst[width * x + y] = redval;
	dst[width * x + y+1] = greenval;
	dst[width * x + y+2] = blueval;
}

/* Get pointer to internal array position */
uint8_t *offset(uint8_t *array, int i, int j, int width) {
   return &array[width * i + j];
}

void Usage(int argc, char **argv, char **image, int *width, int *height, int *loops,
color_t *imageType) {
```

```
        if (argc == 6 && !strcmp(argv[5], "grey")) {
                *image = malloc((strlen(argv[1])+1) * sizeof(char));
                strcpy(*image, argv[1]);
                *width = atoi(argv[2]);
                *height = atoi(argv[3]);
                *loops = atoi(argv[4]);
                *imageType = GREY;
        } else if (argc == 6 && !strcmp(argv[5], "rgb")) {
                *image = malloc((strlen(argv[1])+1) * sizeof(char));
                strcpy(*image, argv[1]);
                *width = atoi(argv[2]);
                *height = atoi(argv[3]);
                *loops = atoi(argv[4]);
                *imageType = RGB;
        } else {
                fprintf(stderr, "\nError Input!\n%s image_name width height loops
[rgb/grey].\n\n", argv[0]);
                MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
                exit(EXIT_FAILURE);
        }
}

/* Divide rows and columns in a way to minimize perimeter of blocks */
int divide_rows(int rows, int cols, int workers) {
   int per, rows_to, cols_to, best = 0;
   int per_min = rows + cols + 1;
   for (rows_to = 1 ; rows_to <= workers ; ++rows_to) {
      if (workers % rows_to || rows % rows_to) continue;
      cols_to = workers / rows_to;
      if (cols % cols_to) continue;
      per = rows / rows_to + cols / cols_to;
      if (per < per_min) {
         per_min = per;
         best = rows_to;
      }
   }
   return best;
}
```

Image Convolution using OMP

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <stdint.h>
```

```
#include "mpi.h"
#include "omp.h"

typedef enum {RGB, GREY} color_t;

void convolute(uint8_t *, uint8_t *, int, int, int, int, int, int, float**, color_t);
void convolute_grey(uint8_t *, uint8_t *, int, int, int, int, float **);
void convolute_rgb(uint8_t *, uint8_t *, int, int, int, int, float **);
void Usage(int, char **, char **, int *, int *, int *, color_t *);
uint8_t *offset(uint8_t *, int, int, int);



/* Divide rows and columns in a way to minimize perimeter of blocks */
int divide_rows(int rows, int cols, int workers) {
   int per, rows_to, cols_to, best = 0;
   int per_min = rows + cols + 1;
   for (rows_to = 1 ; rows_to <= workers ; ++rows_to) {
      if (workers % rows_to || rows % rows_to) continue;
      cols_to = workers / rows_to;
      if (cols % cols_to) continue;
      per = rows / rows_to + cols / cols_to;
      if (per < per_min) {
         per_min = per;
         best = rows_to;
      }
   }
   return best;
}


int main(int argc, char** argv) {
         int thread_count = 4;
         int fd, i, j, k, width, height, loops, t, row_div, col_div, rows, cols;
         double timer, remote_time;
         char *image;
         color_t imageType;
         /* MPI world topology */
   int process_id, num_processes;
         /* Find current task id */
   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &num_processes);
   MPI_Comm_rank(MPI_COMM_WORLD, &process_id);
         /* MPI status */
   MPI_Status status;
         /* MPI data types */
```

```
    MPI_Datatype grey_col_type;
    MPI_Datatype rgb_col_type;
    MPI_Datatype grey_row_type;
    MPI_Datatype rgb_row_type;
        /* MPI requests */
    MPI_Request send_north_req;
    MPI_Request send_south_req;
    MPI_Request send_west_req;
    MPI_Request send_east_req;
    MPI_Request recv_north_req;
    MPI_Request recv_south_req;
    MPI_Request recv_west_req;
    MPI_Request recv_east_req;

        /* Neighbours */
        int north = -1;
        int south = -1;
    int west = -1;
    int east = -1;

    /* Check arguments */
    if (process_id == 0) {
                Usage(argc, argv, &image, &width, &height, &loops, &imageType);
                /* Division of data in each process */
                row_div = divide_rows(height, width, num_processes);
                if (row_div <= 0 || height % row_div || num_processes % row_div || width
% (col_div = num_processes / row_div)) {
                                fprintf(stderr, "%s: Cannot divide to processes\n", argv[0]);
                                MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
                                return EXIT_FAILURE;
                }
        }
        if (process_id != 0) {
                image = malloc((strlen(argv[1])+1) * sizeof(char));
                strcpy(image, argv[1]);
        }
        /* Broadcast parameters */
    MPI_Bcast(&width, 1, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Bcast(&height, 1, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Bcast(&loops, 1, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Bcast(&imageType, 1, MPI_INT, 0, MPI_COMM_WORLD);
        MPI_Bcast(&row_div, 1, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Bcast(&col_div, 1, MPI_INT, 0, MPI_COMM_WORLD);

        /* Compute number of rows per process */
        rows = height / row_div;
```

```
        cols = width / col_div;

        /* Create column data type for grey & rgb */
        MPI_Type_vector(rows, 1, cols+2, MPI_BYTE, &grey_col_type);
        MPI_Type_commit(&grey_col_type);
        MPI_Type_vector(rows, 3, 3*cols+6, MPI_BYTE, &rgb_col_type);
        MPI_Type_commit(&rgb_col_type);
        /* Create row data type */
        MPI_Type_contiguous(cols, MPI_BYTE, &grey_row_type);
        MPI_Type_commit(&grey_row_type);
        MPI_Type_contiguous(3*cols, MPI_BYTE, &rgb_row_type);
        MPI_Type_commit(&rgb_row_type);

         /* Compute starting row and column */
int start_row = (process_id / col_div) * rows;
int start_col = (process_id % col_div) * cols;

        /* Init filters */
        int box_blur[3][3] = {{1, 1, 1}, {1, 1, 1}, {1, 1, 1}};
        int gaussian_blur[3][3] = {{1, 2, 1}, {2, 4, 2}, {1, 2, 1}};
        int edge_detection[3][3] = {{1, 4, 1}, {4, 8, 4}, {1, 4, 1}};
        float **h = malloc(3 * sizeof(float *));
        for (i = 0 ; i < 3 ; i++)
                h[i] = malloc(3 * sizeof(float));
        for (i = 0 ; i < 3 ; i++) {
                for (j = 0 ; j < 3 ; j++){
                        // h[i][j] = box_blur[i][j] / 9.0;
                        h[i][j] = gaussian_blur[i][j] / 16.0;
                        // h[i][j] = edge_detection[i][j] / 28.0;
                }
        }

        /* Init arrays */
        uint8_t *src = NULL, *dst = NULL, *tmpbuf = NULL, *tmp = NULL;
        MPI_File fh;
        int filesize, bufsize, nbytes;
        if (imageType == GREY) {
                filesize = width * height;
                bufsize = filesize / num_processes;
                nbytes = bufsize / sizeof(uint8_t);
                src = calloc((rows+2) * (cols+2), sizeof(uint8_t));
                dst = calloc((rows+2) * (cols+2), sizeof(uint8_t));
        } else if (imageType == RGB) {
                filesize = width*3 * height;
                bufsize = filesize / num_processes;
                nbytes = bufsize / sizeof(uint8_t);
```

```
                src = calloc((rows+2) * (cols*3+6), sizeof(uint8_t));
                dst = calloc((rows+2) * (cols*3+6), sizeof(uint8_t));
        }
        if (src == NULL || dst == NULL) {
    fprintf(stderr, "%s: Not enough memory\n", argv[0]);
    MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
    return EXIT_FAILURE;
        }

        /* Parallel read */
        MPI_File_open(MPI_COMM_WORLD, image, MPI_MODE_RDONLY,
MPI_INFO_NULL, &fh);
        if (imageType == GREY) {
                for (i = 1 ; i <= rows ; i++) {
                        MPI_File_seek(fh, (start_row + i-1) * width + start_col,
MPI_SEEK_SET);
                        tmpbuf = offset(src, i, 1, cols+2);
                        MPI_File_read(fh, tmpbuf, cols, MPI_BYTE, &status);
                }
        } else if (imageType == RGB) {
                for (i = 1 ; i <= rows ; i++) {
                        MPI_File_seek(fh, 3*(start_row + i-1) * width + 3*start_col,
MPI_SEEK_SET);
                        tmpbuf = offset(src, i, 3, cols*3+6);
                        MPI_File_read(fh, tmpbuf, cols*3, MPI_BYTE, &status);
                }
        }
        MPI_File_close(&fh);

        /* Compute neighbours */
   if (start_row != 0)
      north = process_id - col_div;
   if (start_row + rows != height)
      south = process_id + col_div;
   if (start_col != 0)
      west = process_id - 1;
   if (start_col + cols != width)
      east = process_id + 1;

        /* Get time before */
   timer = MPI_Wtime();
        /* Convolute "loops" times */
        for (t = 0 ; t < loops ; t++) {
    /* Send and request borders */
                if (imageType == GREY) {
                        if (north != -1) {
```

```
                                MPI_Isend(offset(src, 1, 1, cols+2), 1, grey_row_type,
north, 0, MPI_COMM_WORLD, &send_north_req);
                                MPI_Irecv(offset(src, 0, 1, cols+2), 1, grey_row_type,
north, 0, MPI_COMM_WORLD, &recv_north_req);
                        }
                        if (west != -1) {
                                MPI_Isend(offset(src, 1, 1, cols+2), 1, grey_col_type,
west, 0, MPI_COMM_WORLD, &send_west_req);
                                MPI_Irecv(offset(src, 1, 0, cols+2), 1, grey_col_type,
west, 0, MPI_COMM_WORLD, &recv_west_req);
                        }
                        if (south != -1) {
                                MPI_Isend(offset(src, rows, 1, cols+2), 1, grey_row_type,
south, 0, MPI_COMM_WORLD, &send_south_req);
                                MPI_Irecv(offset(src, rows+1, 1, cols+2), 1,
grey_row_type, south, 0, MPI_COMM_WORLD, &recv_south_req);
                        }
                        if (east != -1) {
                                MPI_Isend(offset(src, 1, cols, cols+2), 1, grey_col_type,
east, 0, MPI_COMM_WORLD, &send_east_req);
                                MPI_Irecv(offset(src, 1, cols+1, cols+2), 1, grey_col_type,
east, 0, MPI_COMM_WORLD, &recv_east_req);
                        }
                } else if (imageType == RGB) {
                        if (north != -1) {
                                MPI_Isend(offset(src, 1, 3, 3*cols+6), 1, rgb_row_type,
north, 0, MPI_COMM_WORLD, &send_north_req);
                                MPI_Irecv(offset(src, 0, 3, 3*cols+6), 1, rgb_row_type,
north, 0, MPI_COMM_WORLD, &recv_north_req);
                        }
                        if (west != -1) {
                                MPI_Isend(offset(src, 1, 3, 3*cols+6), 1, rgb_col_type,
west, 0, MPI_COMM_WORLD, &send_west_req);
                                MPI_Irecv(offset(src, 1, 0, 3*cols+6), 1, rgb_col_type,
west, 0, MPI_COMM_WORLD, &recv_west_req);
                        }
                        if (south != -1) {
                                MPI_Isend(offset(src, rows, 3, 3*cols+6), 1, rgb_row_type,
south, 0, MPI_COMM_WORLD, &send_south_req);
                                MPI_Irecv(offset(src, rows+1, 3, 3*cols+6), 1,
rgb_row_type, south, 0, MPI_COMM_WORLD, &recv_south_req);
                        }
                        if (east != -1) {
                                MPI_Isend(offset(src, 1, 3*cols, 3*cols+6), 1,
rgb_col_type,  east, 0, MPI_COMM_WORLD, &send_east_req);
```

```
                                MPI_Irecv(offset(src, 1, 3*cols+3, 3*cols+6), 1,
rgb_col_type,  east, 0, MPI_COMM_WORLD, &recv_east_req);
                        }
                }

                /* Inner Data Convolute */
                convolute(src, dst, 1, rows, 1, cols, cols, rows, h, imageType);

        /* Request and compute */
                if (north != -1) {
                        MPI_Wait(&recv_north_req, &status);

                        convolute(src, dst, 1, 1, 2, cols-1, cols, rows, h, imageType);
                }
                if (west != -1) {
                        MPI_Wait(&recv_west_req, &status);
                        convolute(src, dst, 2, rows-1, 1, 1, cols, rows, h, imageType);
                }
                if (south != -1) {
                        MPI_Wait(&recv_south_req, &status);
                        convolute(src, dst, rows, rows, 2, cols-1, cols, rows, h,
imageType);
                }
                if (east != -1) {
                        MPI_Wait(&recv_east_req, &status);
                        convolute(src, dst, 2, rows-1, cols, cols, cols, rows, h, imageType);
                }

                /* Corner data */
                if (north != -1 && west != -1)
                        convolute(src, dst, 1, 1, 1, 1, cols, rows, h, imageType);
                if (west != -1 && south != -1)
                        convolute(src, dst, rows, rows, 1, 1, cols, rows, h, imageType);
                if (south != -1 && east != -1)
                        convolute(src, dst, rows, rows, cols, cols, cols, rows, h,
imageType);
                if (east != -1 && north != -1)
                        convolute(src, dst, 1, 1, cols, cols, cols, rows, h, imageType);

                /* Wait to have sent all borders */
                if (north != -1)
                        MPI_Wait(&send_north_req, &status);
                if (west != -1)
                        MPI_Wait(&send_west_req, &status);
                if (south != -1)
                        MPI_Wait(&send_south_req, &status);
```

```
                if (east != -1)
                        MPI_Wait(&send_east_req, &status);

                /* swap arrays */
                tmp = src;
            src = dst;
            dst = tmp;
        }
        /* Get time elapsed */
    timer = MPI_Wtime() - timer;

        /* Parallel write */
        char *outImage = malloc((strlen(image) + 9) * sizeof(char));
        strcpy(outImage, "blur_");
        strcat(outImage, image);
        MPI_File outFile;
        MPI_File_open(MPI_COMM_WORLD, outImage, MPI_MODE_CREATE |
MPI_MODE_WRONLY, MPI_INFO_NULL, &outFile);
        if (imageType == GREY) {
                for (i = 1 ; i <= rows ; i++) {
                        MPI_File_seek(outFile, (start_row + i-1) * width + start_col,
MPI_SEEK_SET);
                        tmpbuf = offset(src, i, 1, cols+2);
                        MPI_File_write(outFile, tmpbuf, cols, MPI_BYTE,
MPI_STATUS_IGNORE);
                }
        } else if (imageType == RGB) {
                for (i = 1 ; i <= rows ; i++) {
                        MPI_File_seek(outFile, 3*(start_row + i-1) * width + 3*start_col,
MPI_SEEK_SET);
                        tmpbuf = offset(src, i, 3, cols*3+6);
                        MPI_File_write(outFile, tmpbuf, cols*3, MPI_BYTE,
MPI_STATUS_IGNORE);
                }
        }
        MPI_File_close(&outFile);

        /* Get times from other processes and print maximum */
    if (process_id != 0)
        MPI_Send(&timer, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
    else {
        for (i = 1 ; i != num_processes ; ++i) {
            MPI_Recv(&remote_time, 1, MPI_DOUBLE, i, 0, MPI_COMM_WORLD,
&status);
        if (remote_time > timer)
            timer = remote_time;
```

```
        }
        printf("%f\n", timer);
    }

    /* De-allocate space */
    free(src);
    free(dst);
    MPI_Type_free(&rgb_col_type);
    MPI_Type_free(&rgb_row_type);
    MPI_Type_free(&grey_col_type);
    MPI_Type_free(&grey_row_type);

        /* Finalize and exit */
    MPI_Finalize();
        return EXIT_SUCCESS;
}

void convolute(uint8_t *src, uint8_t *dst, int row_from, int row_to, int col_from, int
col_to, int width, int height, float** h, color_t imageType) {
        int i, j;
        if (imageType == GREY) {
#pragma omp parallel for shared(src, dst) schedule(static) collapse(3)
                for (i = row_from ; i <= row_to ; i++)
                        for (j = col_from ; j <= col_to ; j++)
                                convolute_grey(src, dst, i, j, width+2, height, h);
        } else if (imageType == RGB) {
#pragma omp parallel for shared(src, dst) schedule(static) collapse(3)
                for (i = row_from ; i <= row_to ; i++)
                        for (j = col_from ; j <= col_to ; j++)
                                convolute_rgb(src, dst, i, j*3, width*3+6, height, h);
        }
}

void convolute_grey(uint8_t *src, uint8_t *dst, int x, int y, int width, int height, float** h)
{
        int i, j, k, l;
        float val = 0;
        for (i = x-1, k = 0 ; i <= x+1 ; i++, k++)
                for (j = y-1, l = 0 ; j <= y+1 ; j++, l++)
                        val += src[width * i + j] * h[k][l];
        dst[width * x + y] = val;
}

void convolute_rgb(uint8_t *src, uint8_t *dst, int x, int y, int width, int height, float** h)
{
        int i, j, k, l;
```

```
		float redval = 0, greenval = 0, blueval = 0;
		for (i = x-1, k = 0 ; i <= x+1 ; i++, k++)
			for (j = y-3, l = 0 ; j <= y+3 ; j+=3, l++){
				redval += src[width * i + j]* h[k][l];
				greenval += src[width * i + j+1] * h[k][l];
				blueval += src[width * i + j+2] * h[k][l];
			}
		dst[width * x + y] = redval;
		dst[width * x + y+1] = greenval;
		dst[width * x + y+2] = blueval;
}

/* Get pointer to internal array position */
uint8_t *offset(uint8_t *array, int i, int j, int width) {
   return &array[width * i + j];
}

void Usage(int argc, char **argv, char **image, int *width, int *height, int *loops,
color_t *imageType) {
		if (argc == 6 && !strcmp(argv[5], "grey")) {
			*image = malloc((strlen(argv[1])+1) * sizeof(char));
			strcpy(*image, argv[1]);
			*width = atoi(argv[2]);
			*height = atoi(argv[3]);
			*loops = atoi(argv[4]);
			*imageType = GREY;
		} else if (argc == 6 && !strcmp(argv[5], "rgb")) {
			*image = malloc((strlen(argv[1])+1) * sizeof(char));
			strcpy(*image, argv[1]);
			*width = atoi(argv[2]);
			*height = atoi(argv[3]);
			*loops = atoi(argv[4]);
			*imageType = RGB;
		} else {
			MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
			fprintf(stderr, "Error Input!\n%s image_name width height loops
[rgb/grey].\n", argv[0]);
			exit(EXIT_FAILURE);
		}
}
```

Image Convolution done serially

```
#include<iostream>
#include<opencv2/imgproc/imgproc.hpp>
#include<opencv2/highgui/highgui.hpp>
```

```
using namespace std;
using namespace cv;

int reflect(int M, int x)
{
    if(x < 0)
    {
        return -x - 1;
    }
    if(x >= M)
    {
        return 2*M - x - 1;
    }
  return x;
}

int circular(int M, int x)
{
    if (x<0)
        return x+M;
    if(x >= M)
        return x-M;
  return x;
}


void noBorderProcessing(Mat src, Mat dst, float Kernel[][3])
{

    float sum;
    for(int y = 1; y < src.rows - 1; y++){
        for(int x = 1; x < src.cols - 1; x++){
            sum = 0.0;
            for(int k = -1; k <= 1;k++){
                for(int j = -1; j <=1; j++){
                    sum = sum + Kernel[j+1][k+1]*src.at<uchar>(y - j, x - k);
                }
            }
            dst.at<uchar>(y,x) = sum;
        }
    }
}

void refletedIndexing(Mat src, Mat dst, float Kernel[][3])
{
```

```
    float sum, x1, y1;
    for(int y = 0; y < src.rows; y++){
        for(int x = 0; x < src.cols; x++){
            sum = 0.0;
            for(int k = -1;k <= 1; k++){
                for(int j = -1;j <= 1; j++ ){
                    x1 = reflect(src.cols, x - j);
                    y1 = reflect(src.rows, y - k);
                    sum = sum + Kernel[j+1][k+1]*src.at<uchar>(y1,x1);
                }
            }
            dst.at<uchar>(y,x) = sum;
        }
    }
}

void circularIndexing(Mat src, Mat dst, float Kernel[][3])
{
    float sum, x1, y1;
    for(int y = 0; y < src.rows; y++){
        for(int x = 0; x < src.cols; x++){
            sum = 0.0;
            for(int k = -1;k <= 1; k++){
                for(int j = -1;j <= 1; j++ ){
                    x1 = circular(src.cols, x - j);
                    y1 = circular(src.rows, y - k);
                    sum = sum + Kernel[j+1][k+1]*src.at<uchar>(y1,x1);
                }
            }
            dst.at<uchar>(y,x) = sum;
        }
    }
}

int main()
{

     Mat src, dst;


     /// Load an image
     src = imread("salt.jpg", CV_LOAD_IMAGE_GRAYSCALE);

     if( !src.data )
     { return -1; }
```

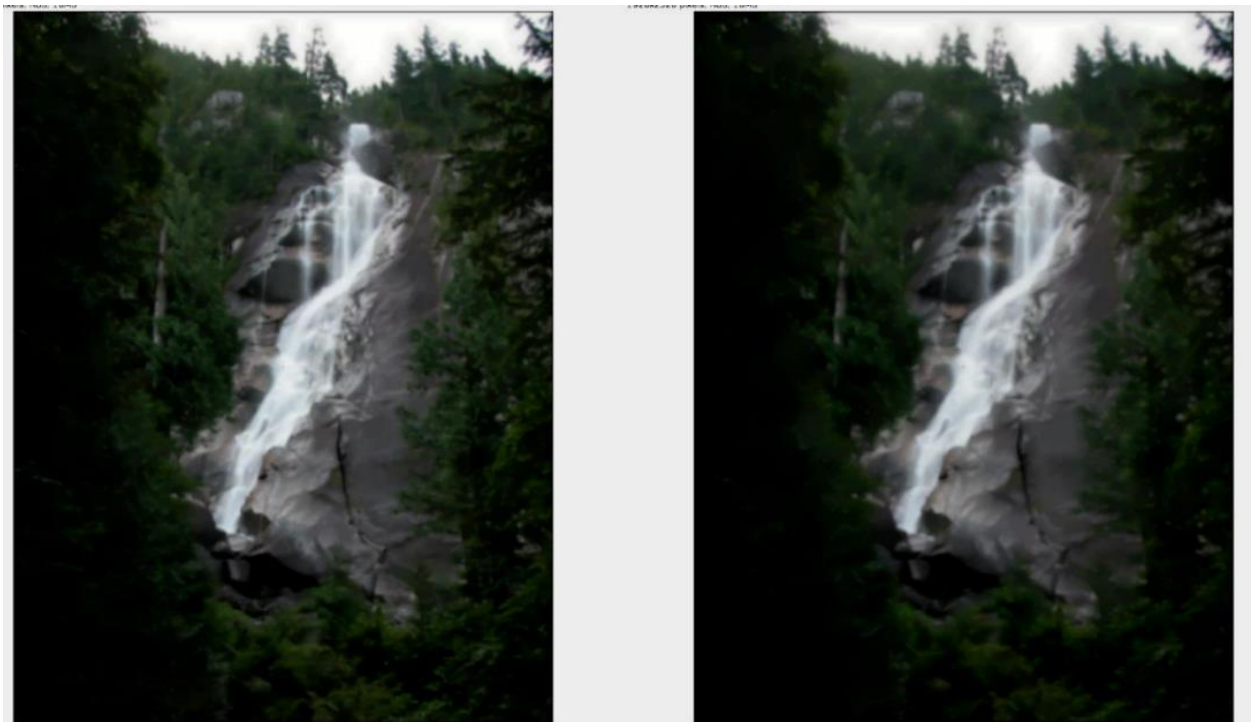
```
    float Kernel[3][3] = {
                         {1/9.0, 1/9.0, 1/9.0},
                         {1/9.0, 1/9.0, 1/9.0},
                         {1/9.0, 1/9.0, 1/9.0}
                        };

     dst = src.clone();
     for(int y = 0; y < src.rows; y++)
        for(int x = 0; x < src.cols; x++)
           dst.at<uchar>(y,x) = 0.0;

     circularIndexing(src, dst, Kernel);



     namedWindow("final");
     imshow("final", dst);

     namedWindow("initial");
     imshow("initial", src);

    waitKey();


  return 0;
}
```

4. RESULTS & DISCUSSION

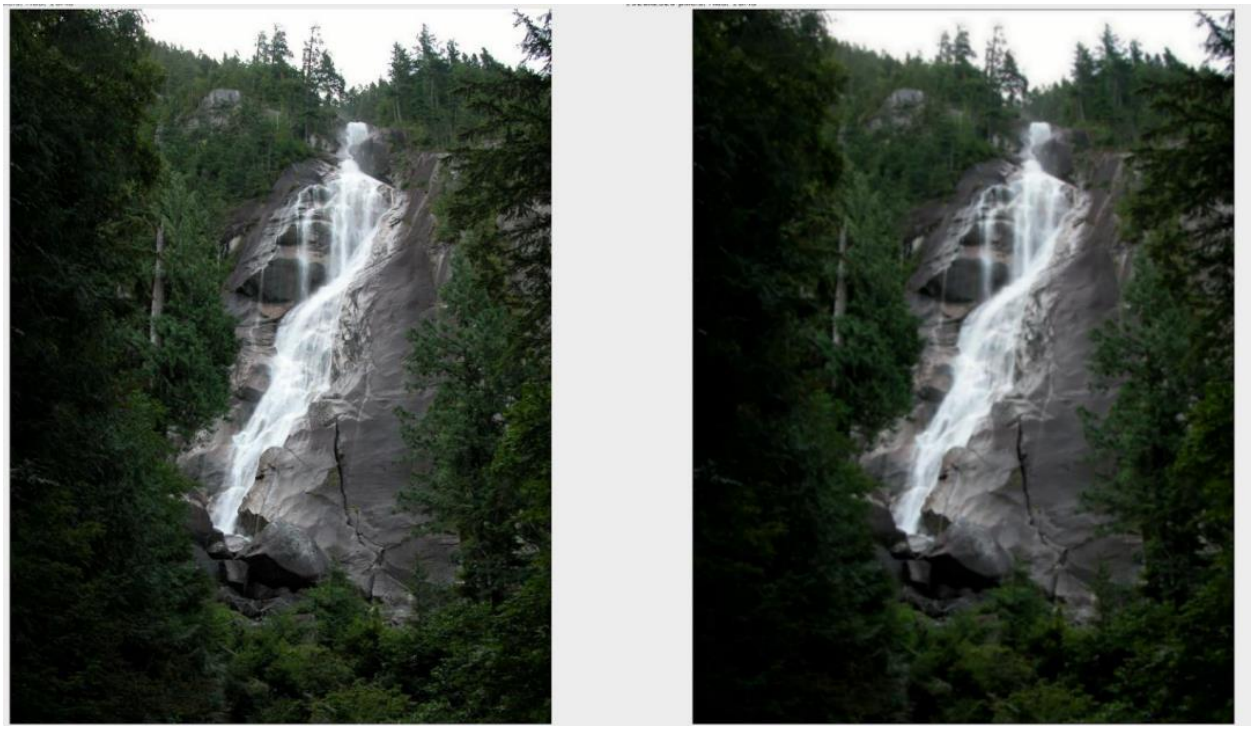
MPI

On running the MPI

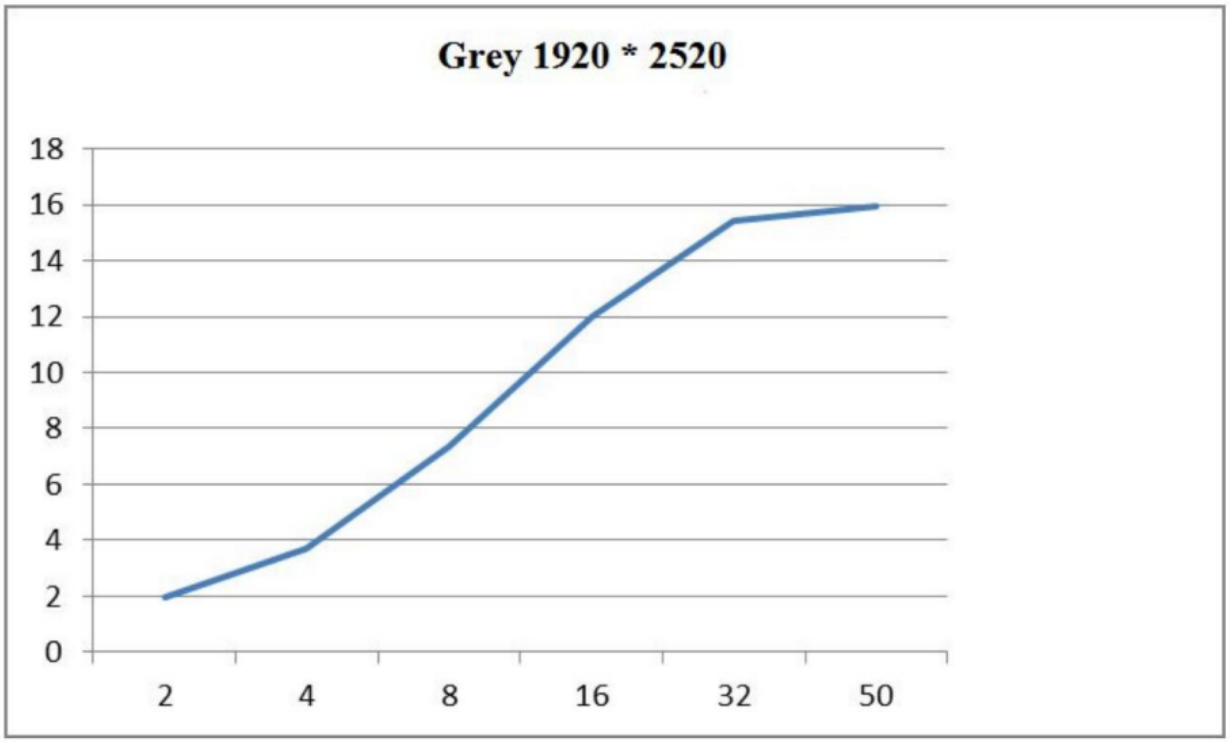
On running OMP



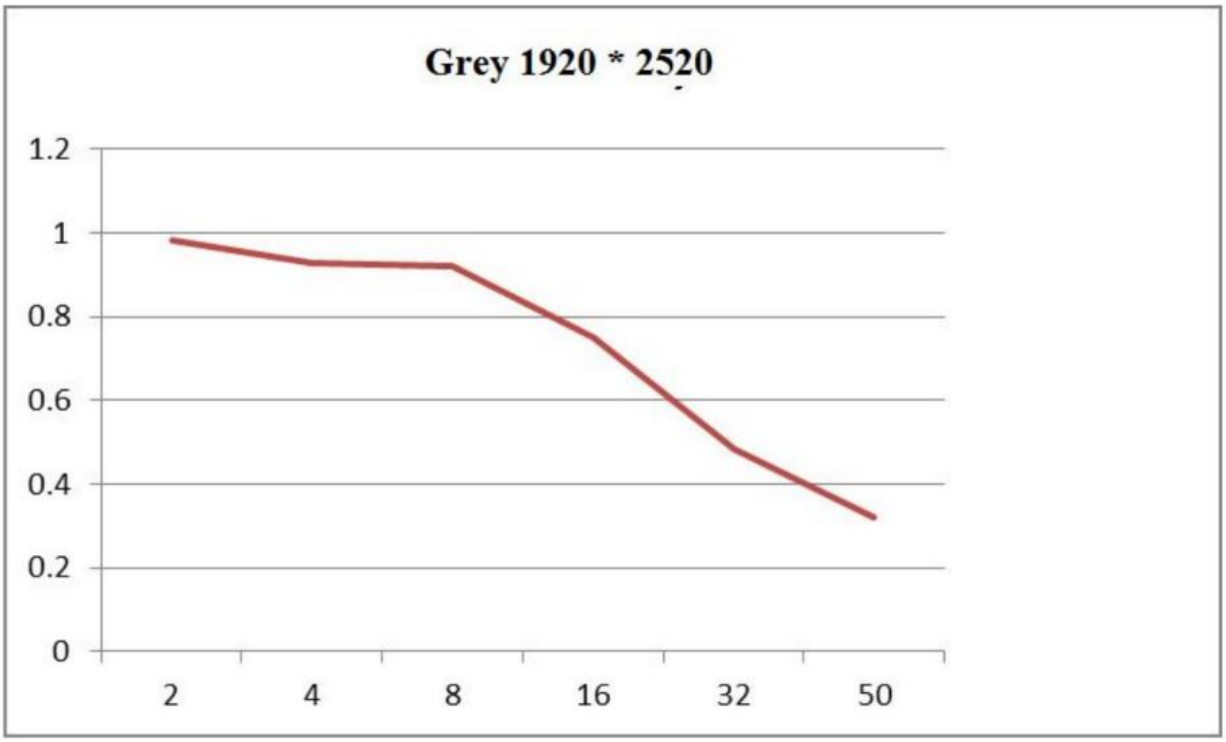
On running it serially



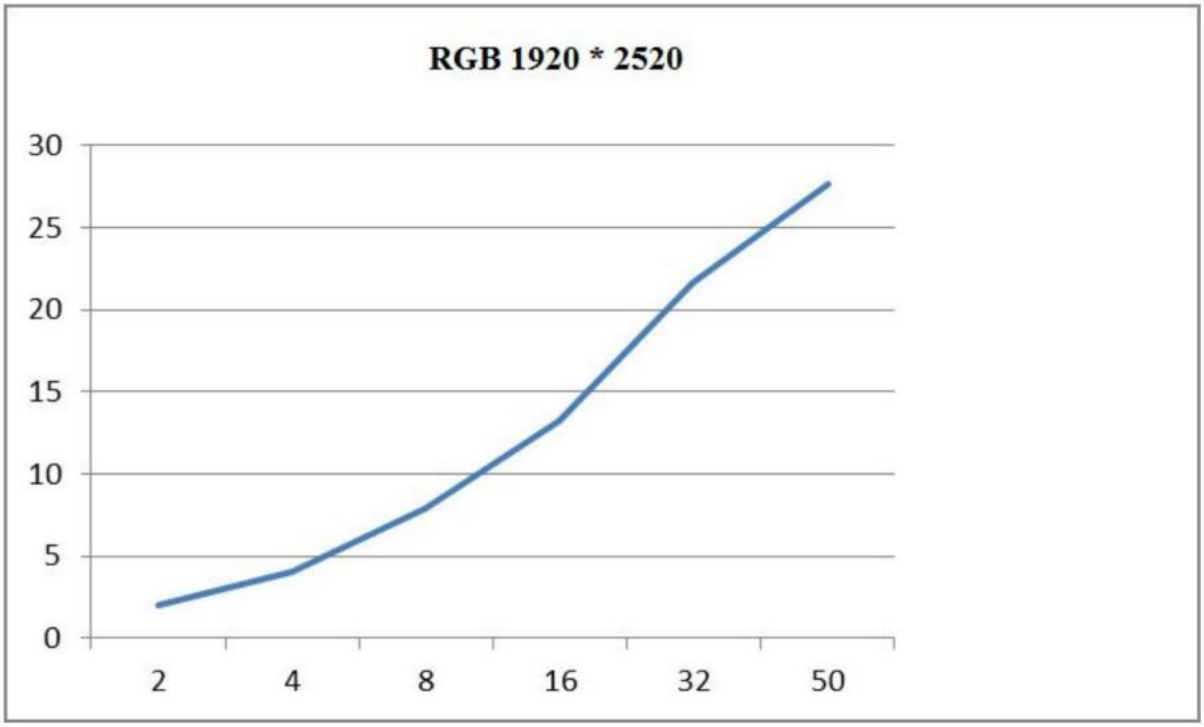
Some comparative charts between MPI and OMP were created to evaluate.
MPI



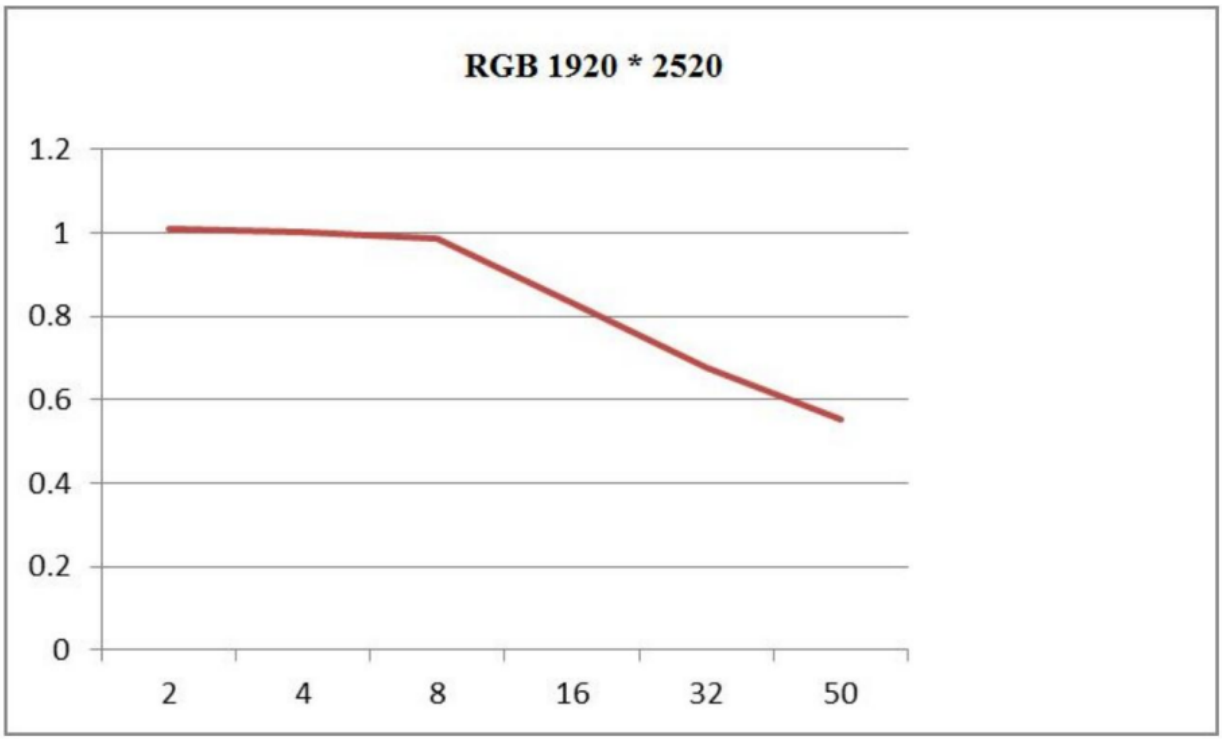
ÿ.1.3 Speed Up = Tserial / Tparallel



ÿ.1.4 Efficiency = Speed Up / Proc

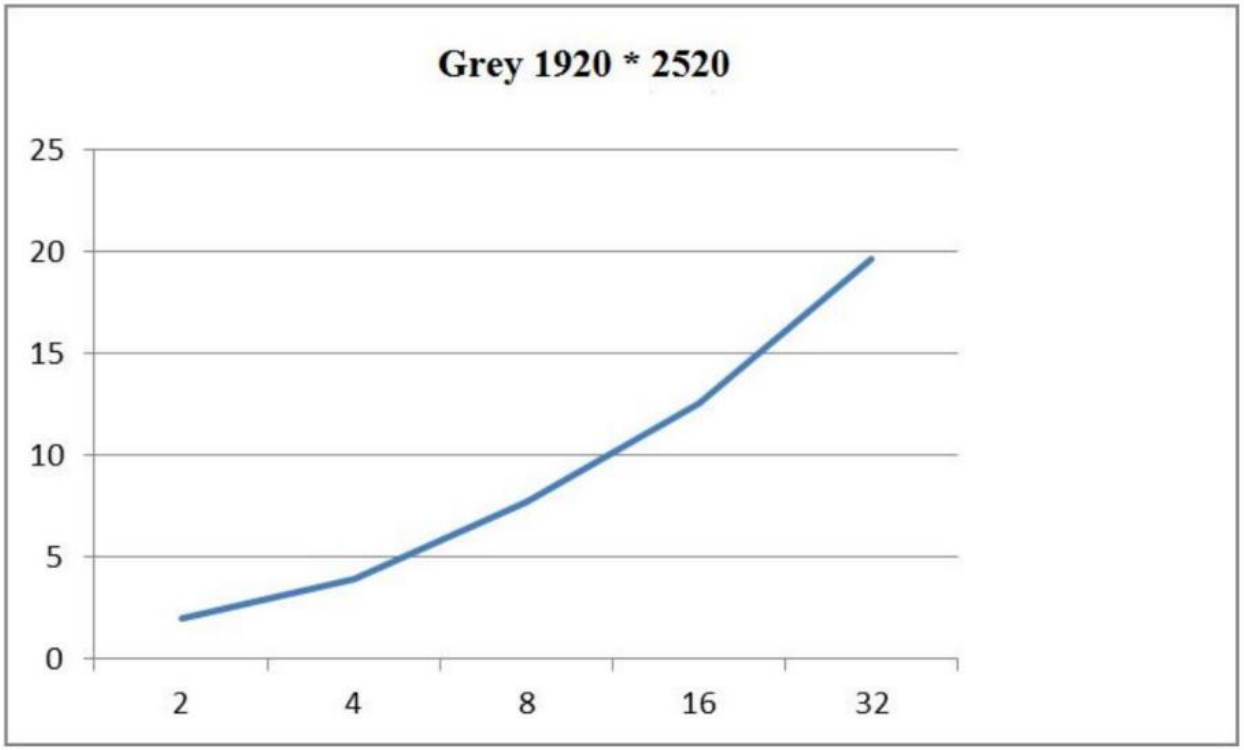


ÿ.1.5 Speed Up = Tserial / Tparallel

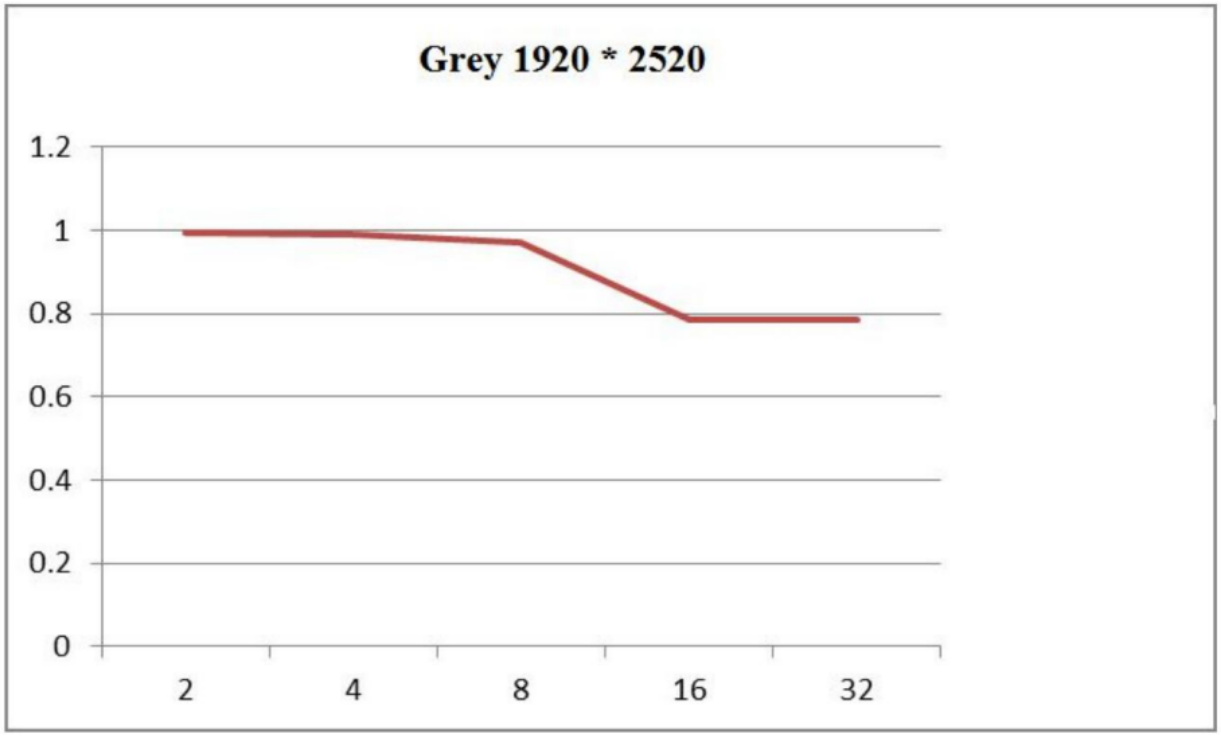


ÿ.1.6 Efficiency = Speed Up / Proc

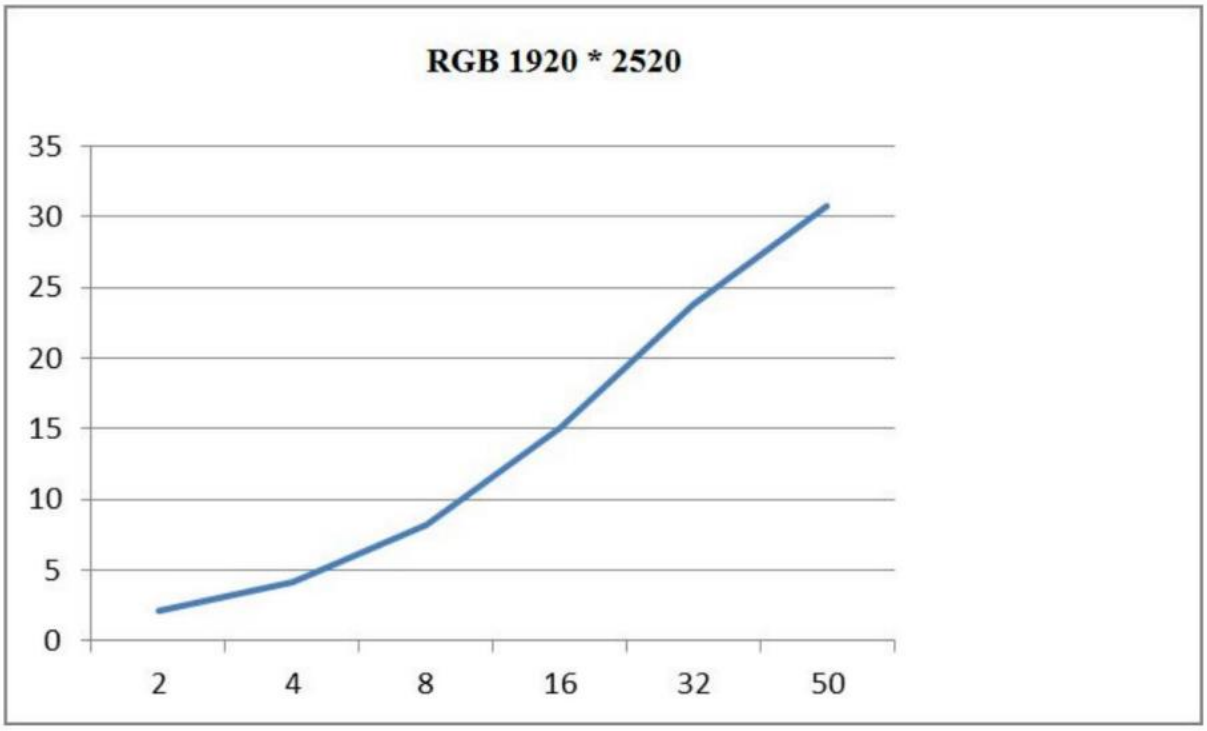
OMP



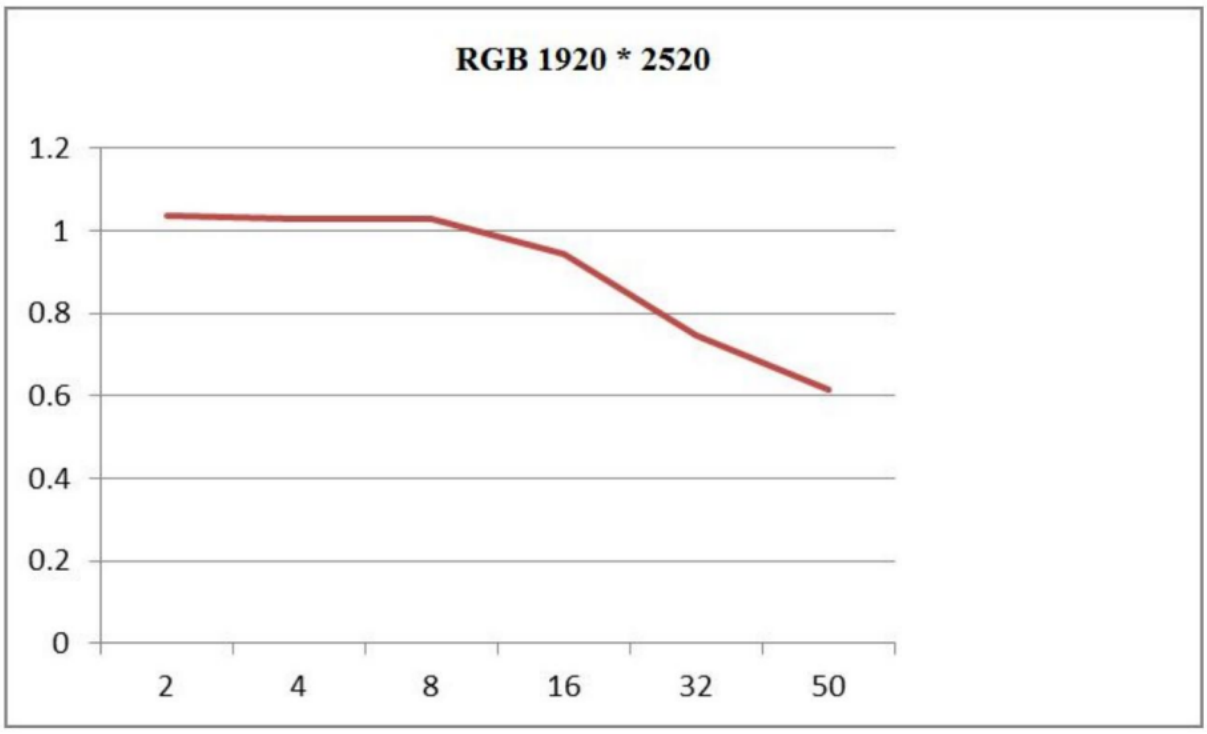
ÿ.2.3 Speed Up = Tserial / Tparallel



ÿ.2.3 Efficiency = Speed Up / Proc



ÿ.2.4 Speed Up = Tserial / Tparallel



ÿ.2.5 Efficiency = Speed Up / Proc

5. CONCLUSION

The image convolution works faster using parallel programming because the work is done to each section of the picture separately.
The efficiency and speed up for MPI and OMP is calculated to compare the overall performance of the algorithm.

6. FUTURE WORK

The efficiency of the code could be improved by refactoring the code more.