

**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**

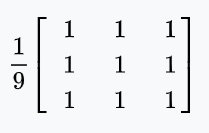
|  |  |  |
| --- | --- | --- |
| **CHAPTER NO.** | **TITLE** | **PAGE NO.** |
| 1 | **Introduction** | 3 |
| 2 | **Proposed Work** | 3 |
| 3 | **Code & Implementation** | 5 |
| 4 | **Results & Discussion** | 27 |
| 5 | **Conclusion** | 33 |
| 6 | **Future work** | 33 |

1. **INTRODUCTION**

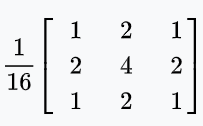
Convolution is the process of adding each element of the image to its local neighbors, weighted by the kernel. This is related to a form of mathematical convolution. The matrix operation being performed—convolution—is not traditional matrix multiplication, despite being similarly denoted by \*.

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.

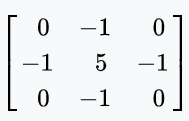
For Box Blur,



For Gaussian Blur,



For Sharpen,



1. **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).

1. 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.

1. Agglomeration
2. 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

1. **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();

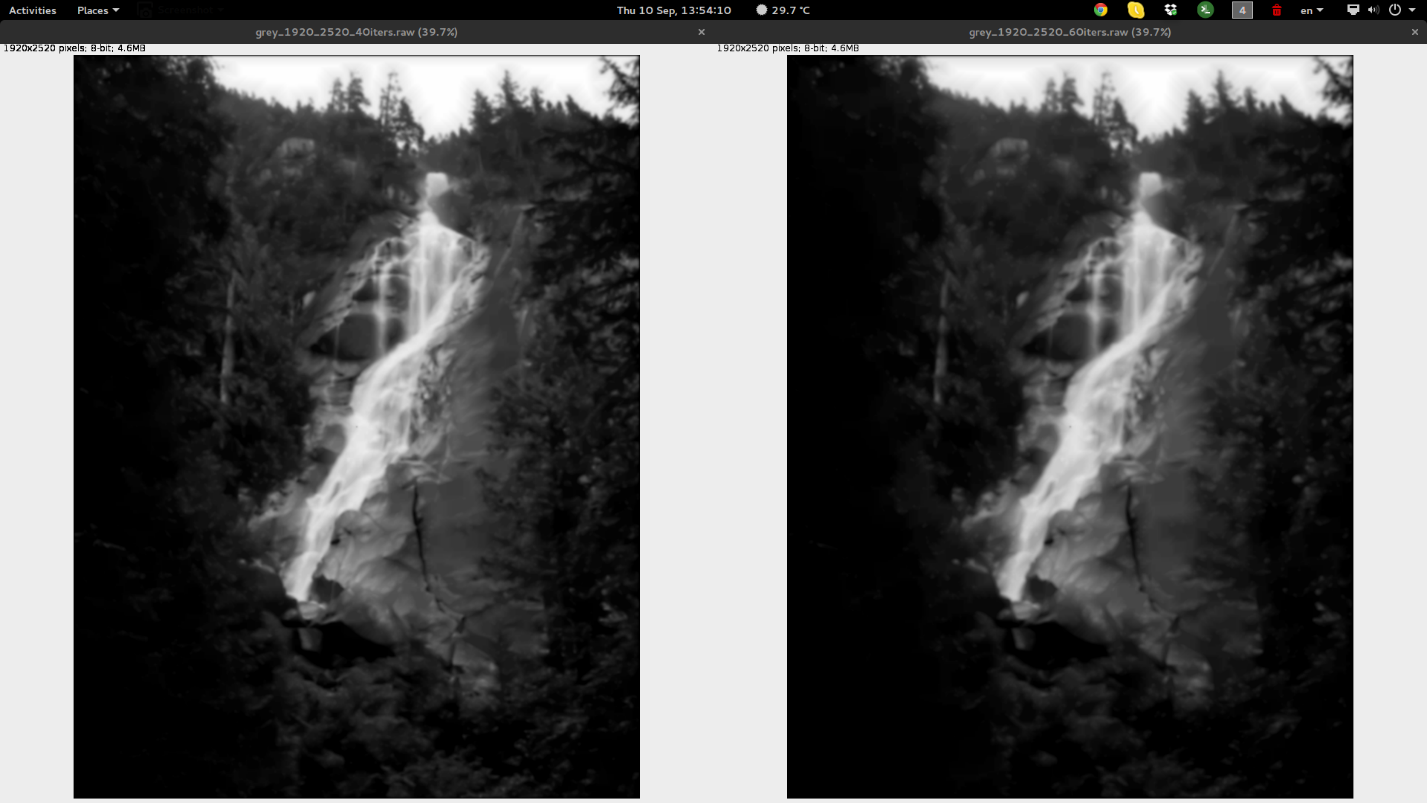
return 0;

}

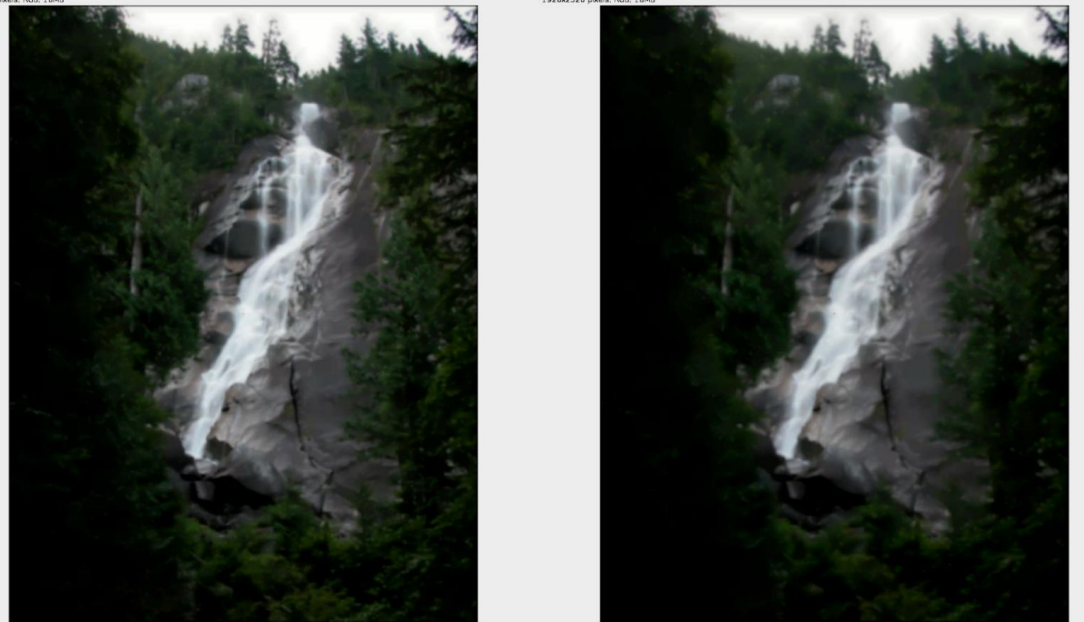
1. **RESULTS & DISCUSSION**

MPI

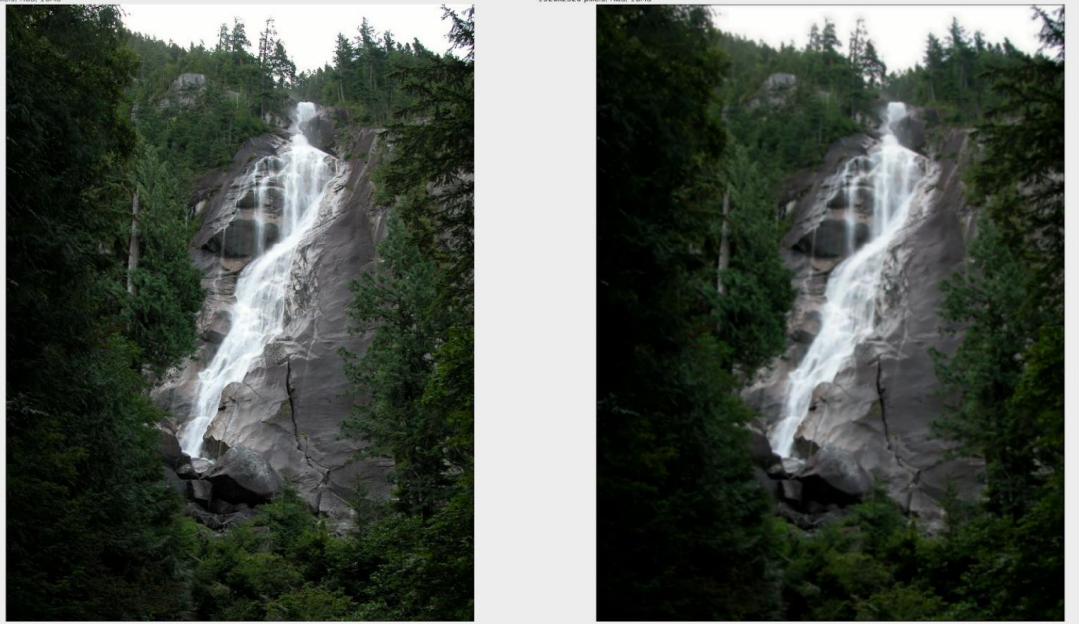
On running the MPI



On running OMP

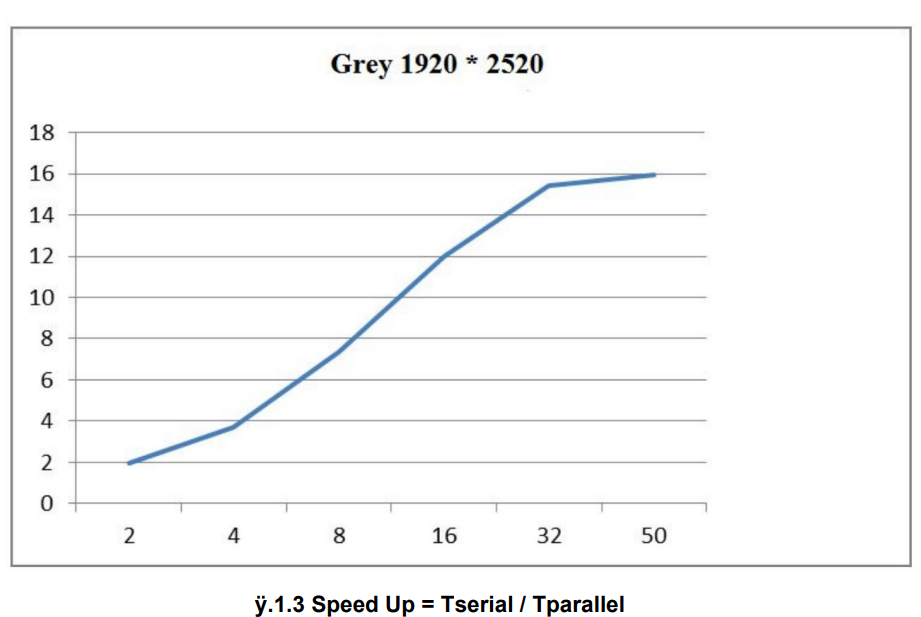


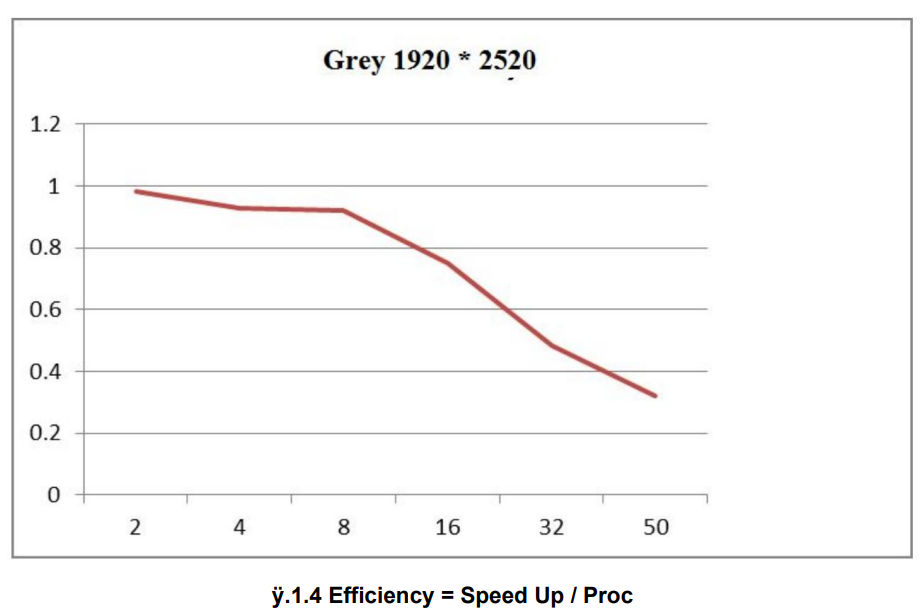
On running it serially

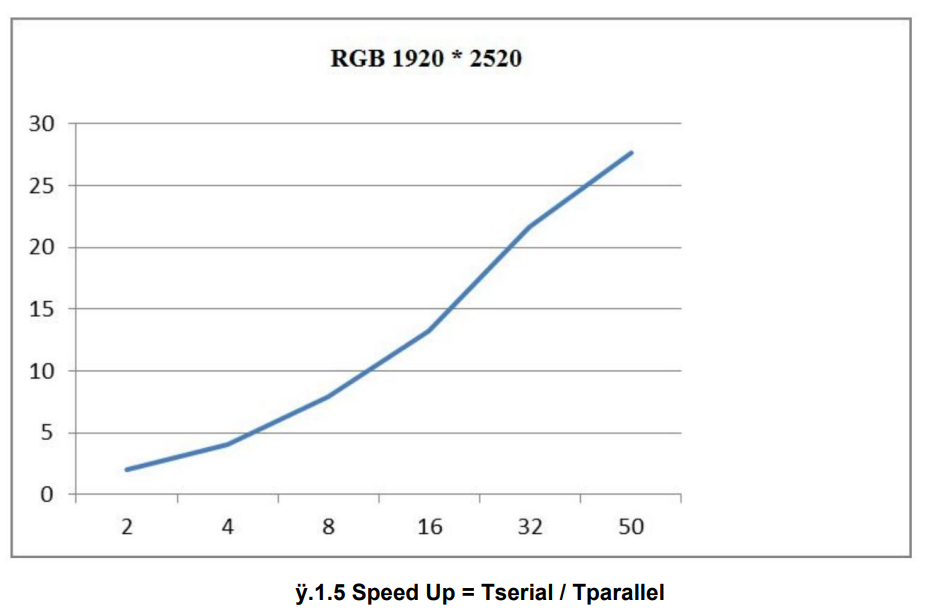


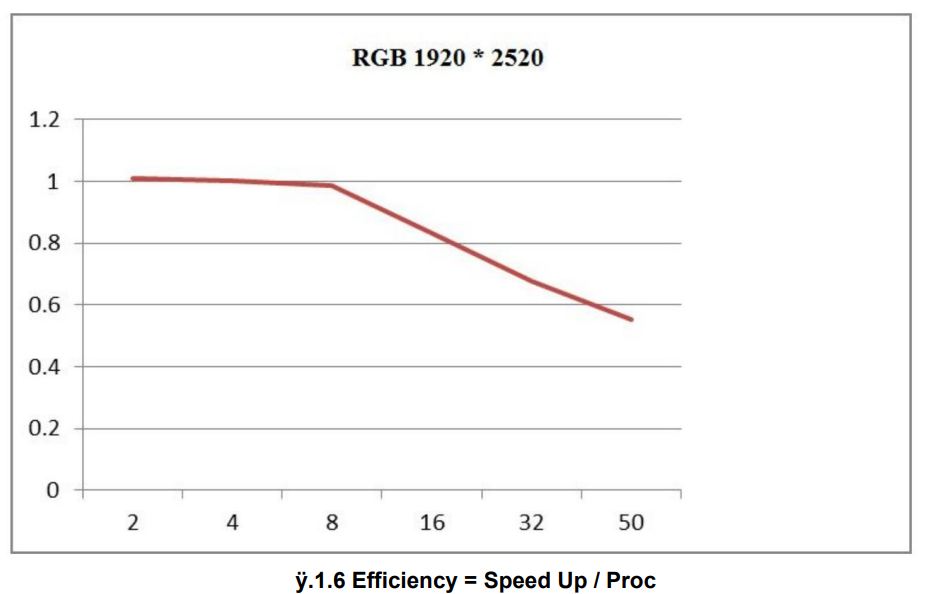
Some comparative charts between MPI and OMP were created to evaluate.

MPI

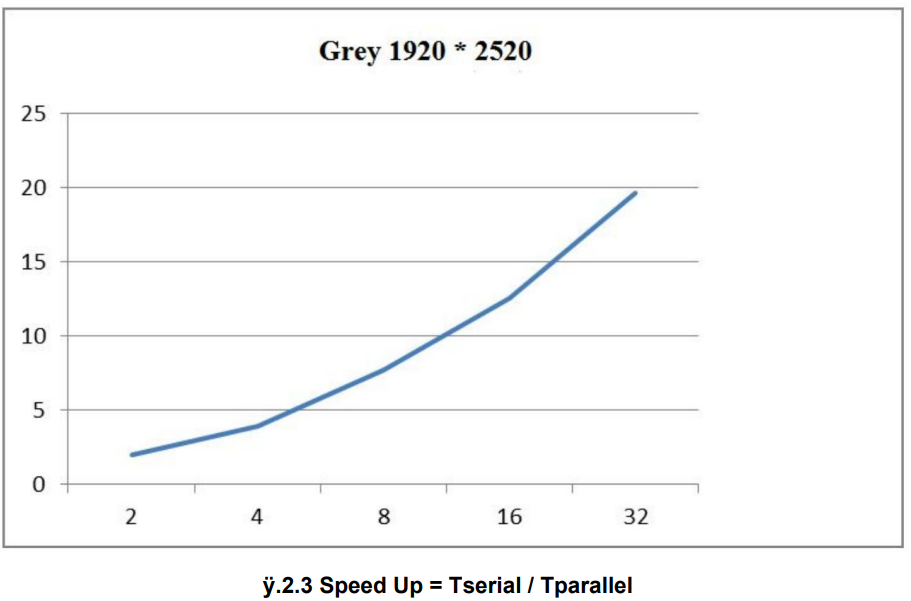


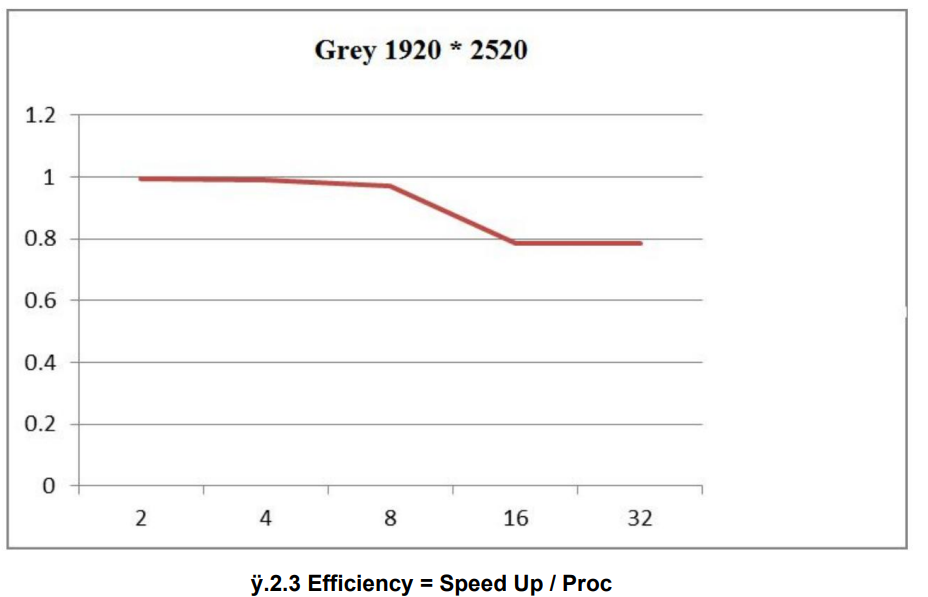


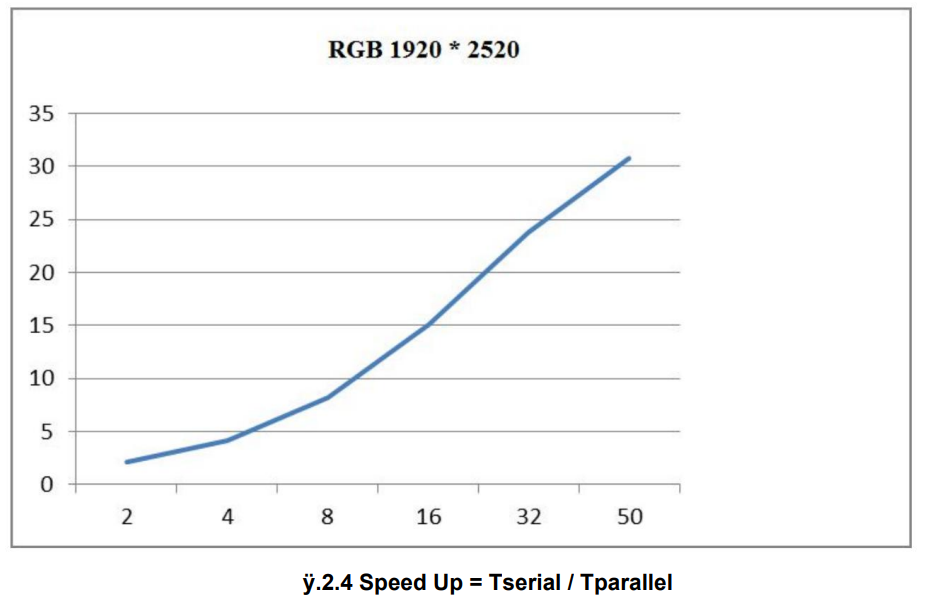


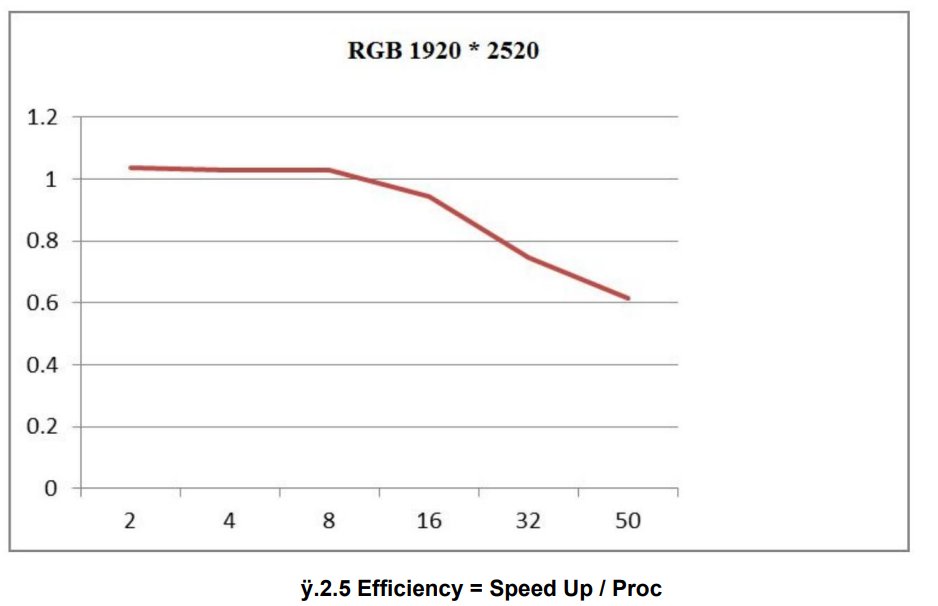


OMP









1. **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.

1. **FUTURE WORK**

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