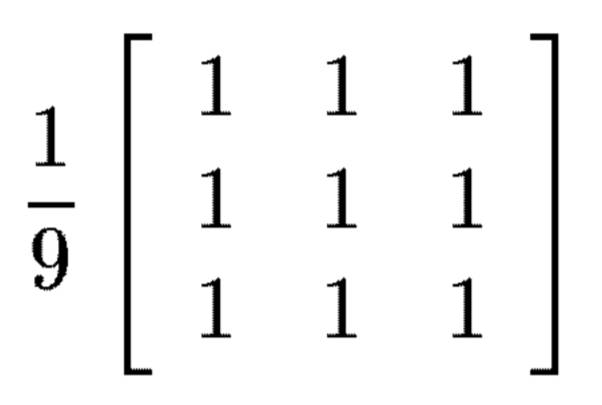
**Image Processing using Parallel programming**

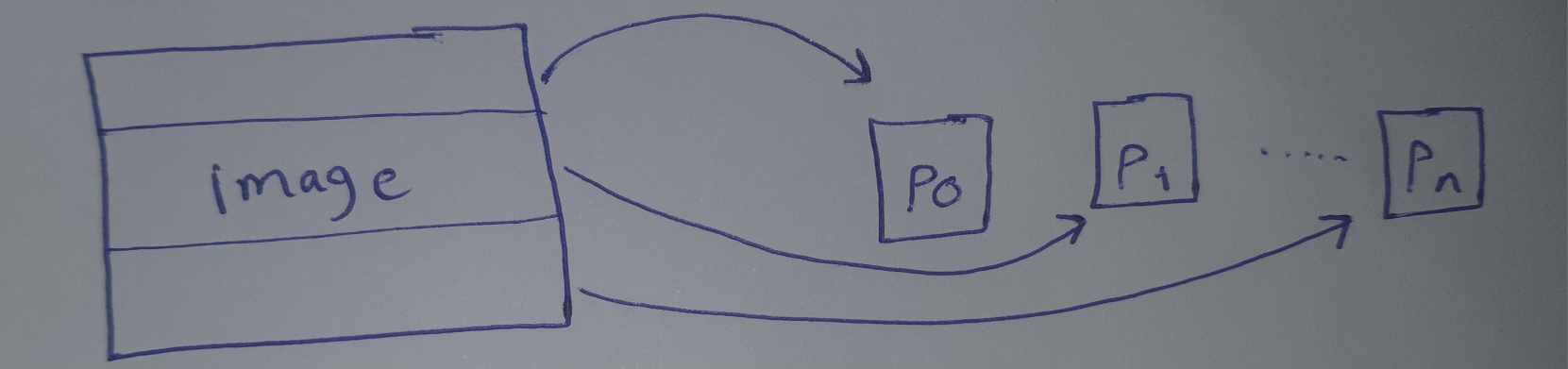
**Introduction:**

Our project will be about using parallel computing to speed up applying an average filter in image processing.

1. we will broadcast the filter array to all processors.



2.Divide the image equally among all processors.



3.Each processor will apply the filter on its own part of the image.

4.All processors will the send their parts of the image after applying the filter to the master.

5.The master processor will assemble all processed parts into one complete image.

**A) Using MPI Library:**

Process 0 initializes MPI then Process 0 partitions the data array among processors by scattering the data array through MPI send operations. Each processor receives its segment of the data array from process 0 by MPI receive and stores it into recv\_data array. After this each processor, including process 0, applies the average filter on its portion of the data array. then each processor transmits its filtered portion of recv\_data array back to process 0 and are stored into data\_filterd array. Once all filtered data is gathered, process 0 showcases the original matrix and the filtered matrix after applying the filter, displaying the before-and-after effects of the filter, as described in the following steps.

* Process 0 divides the matrix among processors by scattering the data array using MPI send
* Each processes receives its portion of the data array from process 0 using MPI receive and stores it into recv\_data array then perform filtering on the received portion of the data array including process 0
* Each processes sends its filtered data to process 0
* Process 0 gathers the filtered data from all processes and stores it into data\_filterd array
* Process 0 displays the data array and the new data\_filterd array after applying filter

**The Code of MPI:**

#include<stdio.h>

#include<mpi.h>

#include<math.h>

#define maxsize\_row 50

#define maxsize\_col 10

/\*double array\_copy[maxsize\_row][maxsize\_col]={{1,2,3},{4,5,6},{7,8,9},{10,11,12},{13,14,15},{16,17,18},{19,20,21},{22,23,24},{1,2,3},{4,5,6},{7,8,9},{10,11,12},{13,14,15},{16,17,18},{19,20,21},{22,23,24},{1,2,3},{4,5,6},{7,8,9},{10,11,12},{13,14,15},{16,17,18},{19,20,21},{22,23,24}};//here the image in array\_copy\*/

double filter\_average[3][3]={{0.11,0.11,0.11},{0.11,0.11,0.11},{0.11,0.11,0.11}};

int c=1;

void fill\_Matrix(double m[maxsize\_row][maxsize\_col])

{

for (int i = 0; i < maxsize\_row; i++)

{

for (int j = 0; j < maxsize\_col; j++)

{

m[i][j] =c;

c++;

}

}

}

void print\_Matrix(double m[maxsize\_row][maxsize\_col])

{

for(int a=0;a<maxsize\_row;a++)

{

//printf("row number %d:",a);

for(int b=0;b<maxsize\_col;b++)

printf("%f ",m[a][b]);

printf("\n");

}

}

//+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

void main(int argc,char \*\*argv)

{

int id,nop;

MPI\_Init(&argc,&argv);

MPI\_Comm\_size(MPI\_COMM\_WORLD,&nop);

MPI\_Comm\_rank(MPI\_COMM\_WORLD,&id);

MPI\_Status status;

int x=(maxsize\_row-2)/nop;

int rest=(maxsize\_row-2) % nop;

double data[maxsize\_row][maxsize\_col];

double data\_filterd[maxsize\_row][maxsize\_col];

double local\_data[x+rest+2][maxsize\_col];

double local\_data\_filterd[x+rest+2][maxsize\_col];

if(id==0){ // the root is p0

fill\_Matrix(data);

int count=1;

//-----------------------------------------------------------------------------------------

// scatter the data to all p exept p0 by send&recv

for(int i=1;i<nop;i++)

{

for(int j=0;j<x;j++)

{

if(j==0)

MPI\_Send(&data[count-1],maxsize\_col,MPI\_DOUBLE,i,1,MPI\_COMM\_WORLD);

MPI\_Send(&data[count],maxsize\_col,MPI\_DOUBLE,i,1,MPI\_COMM\_WORLD);

if(j==x-1)

MPI\_Send(&data[count+1],maxsize\_col,MPI\_DOUBLE,i,1,MPI\_COMM\_WORLD);

count++;

}

}

//---------------------------------------------------------------------------------------------

// scatter data to p0

int local\_count=0;

for(int i=count-1;i<maxsize\_row;i++)

{

for(int j=0;j<maxsize\_col;j++)

local\_data[local\_count][j]=data[i][j];

local\_count++;

}

//-------------------------------------------------------------------------------------------

// start filter on p0 (local data)

for (int i = 0; i < x+2+rest; i++)

{

for (int j = 0; j < maxsize\_col; j++)

{

local\_data\_filterd[i][j] =local\_data[i][j] ;

}

}

for(int a=1;a<x+1+rest;a++)

for(int b=1;b<maxsize\_col-1;b++)

{

local\_data\_filterd[a][b]=(local\_data[a-1][b-1]\*filter\_average[0][0]) + (local\_data[a-1][b]\*filter\_average[0][1]) + (local\_data[a-1][b+1]\*filter\_average[0][2]) + (local\_data[a][b-1]\*filter\_average[1][0]) +(local\_data[a][b]\*filter\_average[1][1]) + (local\_data[a][b+1]\*filter\_average[1][2]) + (local\_data[a+1][b-1]\*filter\_average[2][0]) + (local\_data[a+1][b]\*filter\_average[2][1]) + (local\_data[a+1][b+1]\*filter\_average[2][2]);

}

//-------------------------------------------------------------------------------------------

// gathering the data filterd from all processes by send/recv

int countReturnData=0;

for(int np=1;np<nop;np++)

{

if(np!=1)

for(int r=0;r<x;r++)

{

MPI\_Recv(&data\_filterd[countReturnData],maxsize\_col,MPI\_DOUBLE,np,1,MPI\_COMM\_WORLD,&status);

countReturnData++;

}

else

for(int r=0;r<x+1;r++)

{

MPI\_Recv(&data\_filterd[countReturnData],maxsize\_col,MPI\_DOUBLE,np,1,MPI\_COMM\_WORLD,&status);

countReturnData++;

}

}

//-------------------------------------------------------------------------------------------

// gathering the data filterd from p0 by into data\_filterd

int local\_count2=1;

for(int ii=countReturnData;ii<maxsize\_row;ii++)

{

for(int j=0;j<maxsize\_col;j++)

data\_filterd[ii][j]=local\_data\_filterd[local\_count2][j];

local\_count2++;

}

}

//=============================================================================================

// other p

//=============================================================================================

if(id!=0)

{

int rcount=0;

double recv\_data[x+2][maxsize\_col];

double recv\_data\_filterd[x+2][maxsize\_col];

//---------------------------------------------------------------------------------------------

// recv the scattering data into (recv\_data)

for(int j=0;j<x+2;j++)

{

MPI\_Recv(&recv\_data[rcount],maxsize\_col,MPI\_DOUBLE,0,1,MPI\_COMM\_WORLD,&status);

rcount++;

}

//---------------------------------------------------------------------------------------------

// start filtering on recv\_data

for (int i = 0; i < x+2; i++)

{

for (int j = 0; j < maxsize\_col; j++)

{

recv\_data\_filterd[i][j] =recv\_data[i][j] ;

}

}

for(int a=1;a<x+1;a++)

for(int b=1;b<maxsize\_col-1;b++)

{

recv\_data\_filterd[a][b]=(recv\_data[a-1][b-1]\*filter\_average[0][0]) + (recv\_data[a-1][b]\*filter\_average[0][1]) + (recv\_data[a-1][b+1]\*filter\_average[0][2]) + (recv\_data[a][b-1]\*filter\_average[1][0]) +(recv\_data[a][b]\*filter\_average[1][1]) + (recv\_data[a][b+1]\*filter\_average[1][2]) + (recv\_data[a+1][b-1]\*filter\_average[2][0]) + (recv\_data[a+1][b]\*filter\_average[2][1]) + (recv\_data[a+1][b+1]\*filter\_average[2][2]);

}

//------------------------------------------------------------------------------------------

// send the partion(after filterd) of evry process to p0

int scount=0;

if(id!=1)

{

scount=0;

for(int j=1;j<x+1;j++)

{

MPI\_Send(&recv\_data\_filterd[j],maxsize\_col,MPI\_DOUBLE,0,1,MPI\_COMM\_WORLD);

scount++;

}

}

else

{

scount=0;

for(int j=0;j<x+1;j++)

{

MPI\_Send(&recv\_data\_filterd[j],maxsize\_col,MPI\_DOUBLE,0,1,MPI\_COMM\_WORLD);

scount++;

}

}

/\*printf("\n\nthe process is %d:\n",id);

for(int a=0;a<x+2;a++)

{

printf("the row %d are: ",a);

for(int b=0;b<maxsize\_col;b++)

printf("%f ,",recv\_data[a][b]);

printf("\n"); }

for(int a=0;a<x+2;a++)

{

printf("the row filterd %d are: ",a);

for(int b=0;b<maxsize\_col;b++)

printf("%f ,",recv\_data\_filterd[a][b]);

printf("\n"); }\*/

}

MPI\_Barrier(MPI\_COMM\_WORLD);

//=============================================================================================

// show image befor and after the fitering

//=============================================================================================

if(id==0)

{

printf("\n\nthe original image (old image):\n");

print\_Matrix(data);

printf("\n\nthe filterd image by average filter (new image):\n");

print\_Matrix(data\_filterd);

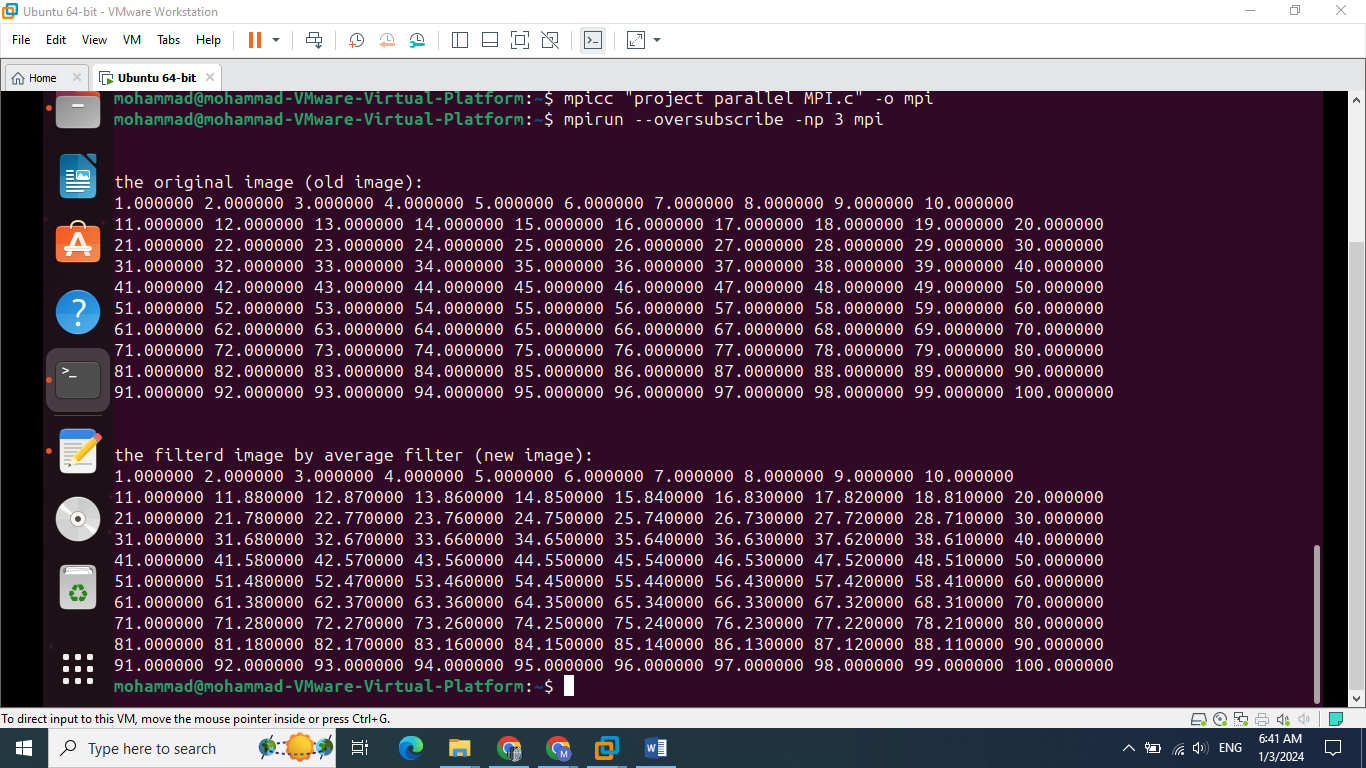
}

MPI\_Finalize();

}

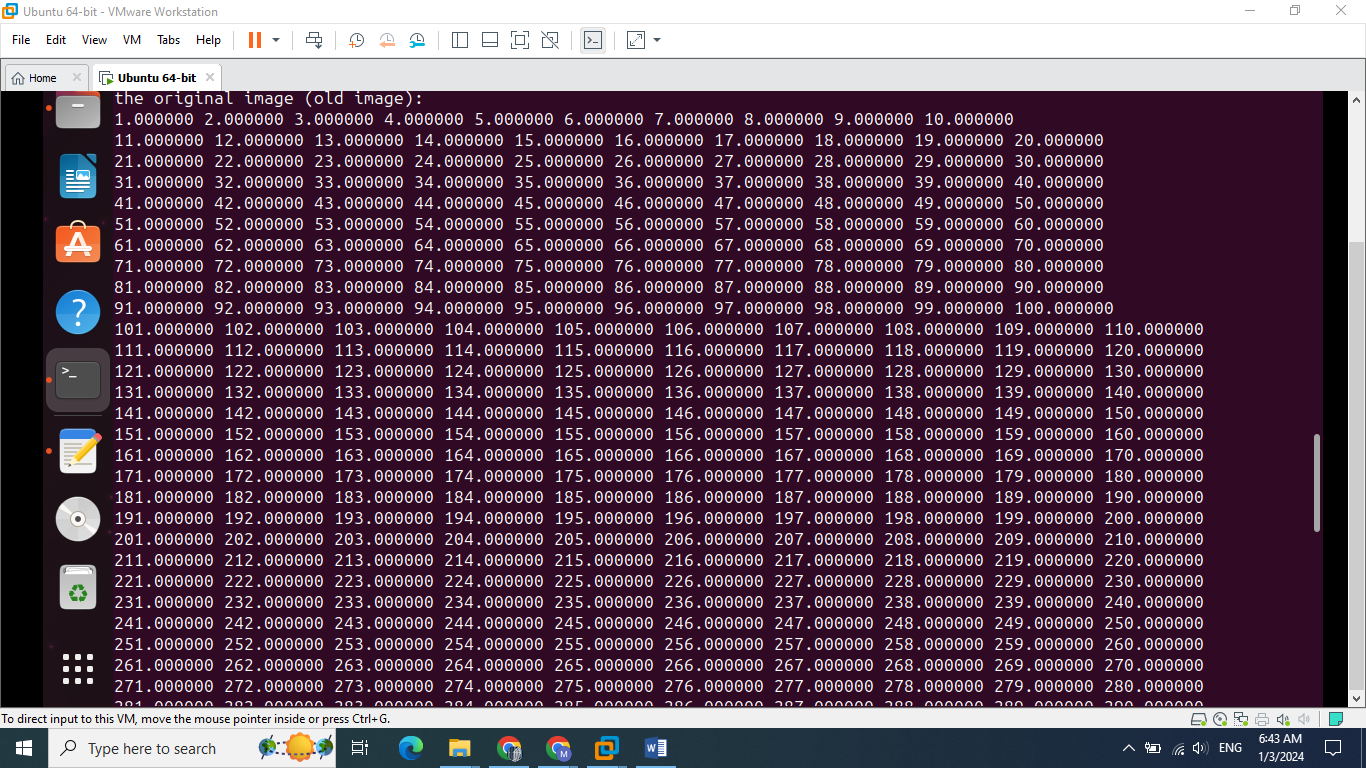
**Test Cases for MPI:**

1. Small matrix(10x10):

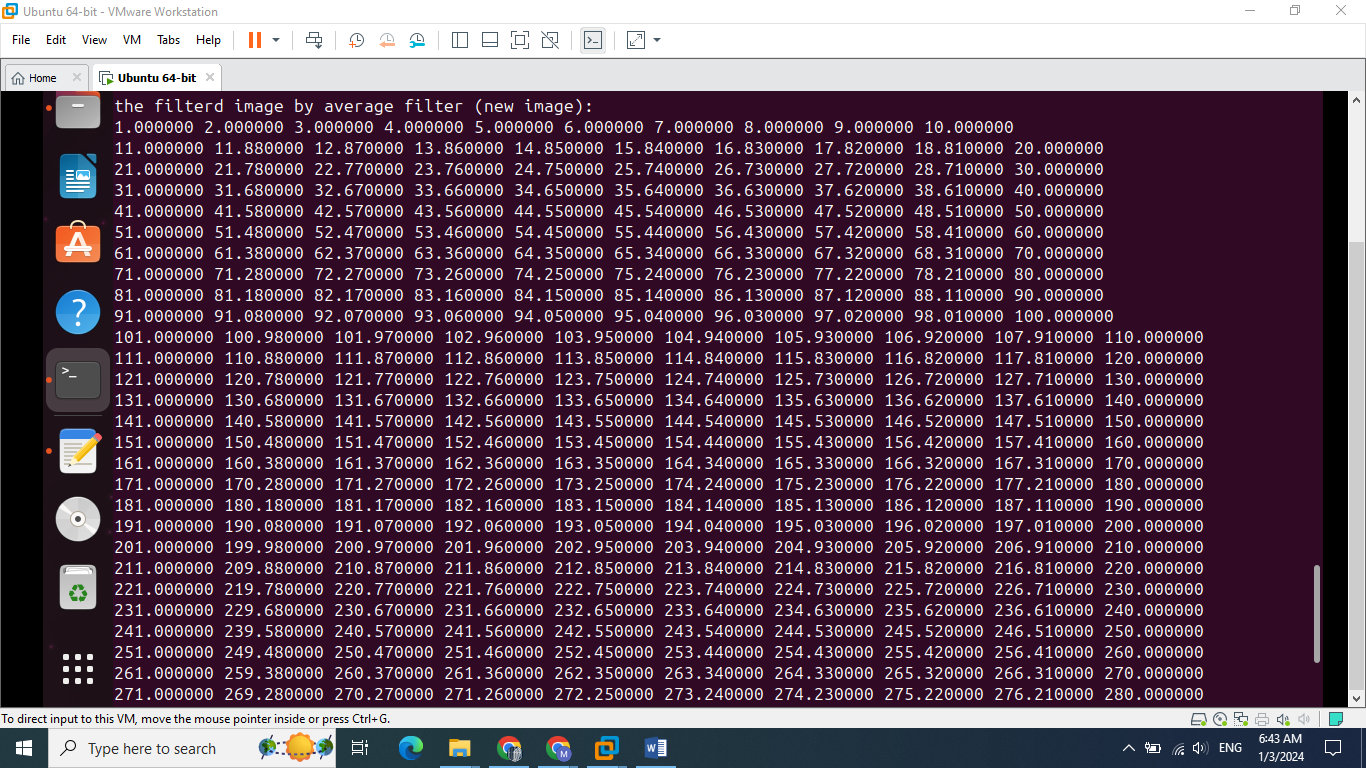


1. Medium matrix(50x10):

Input image:

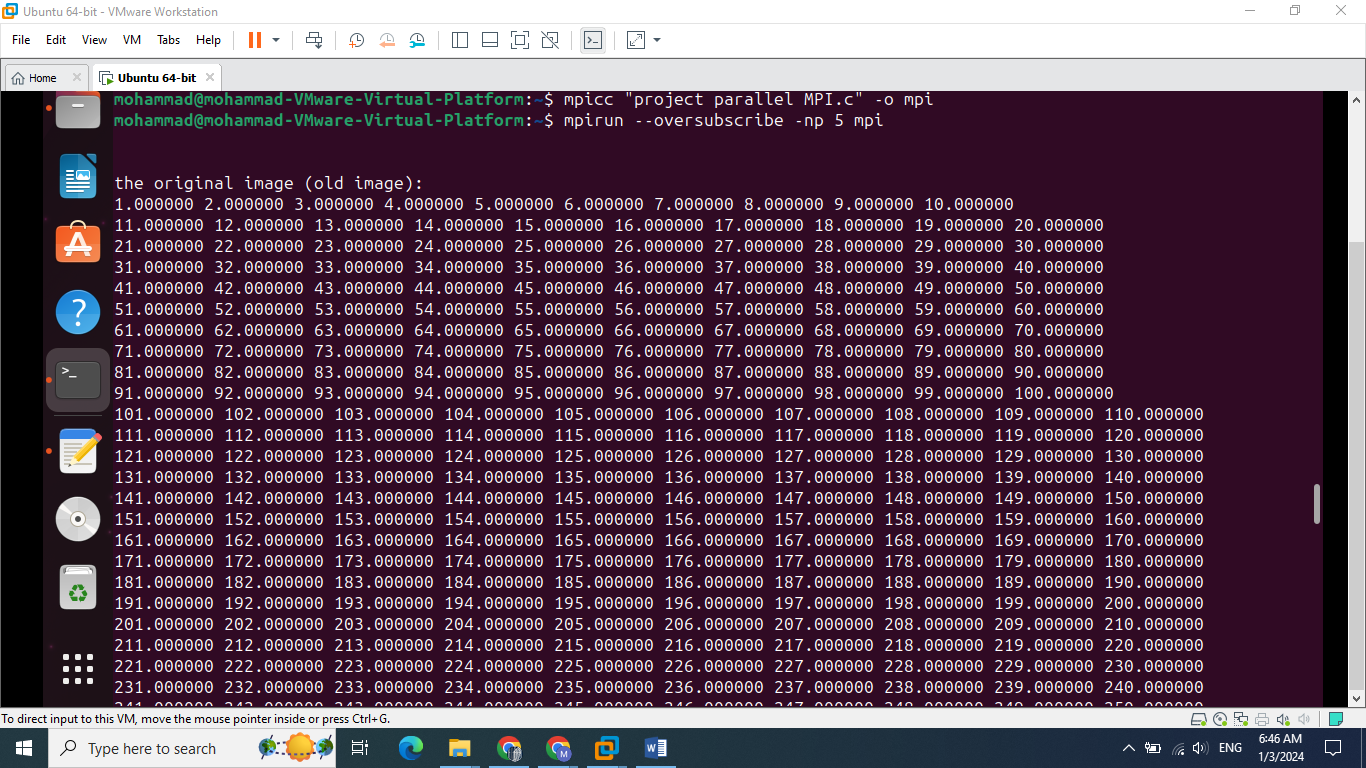


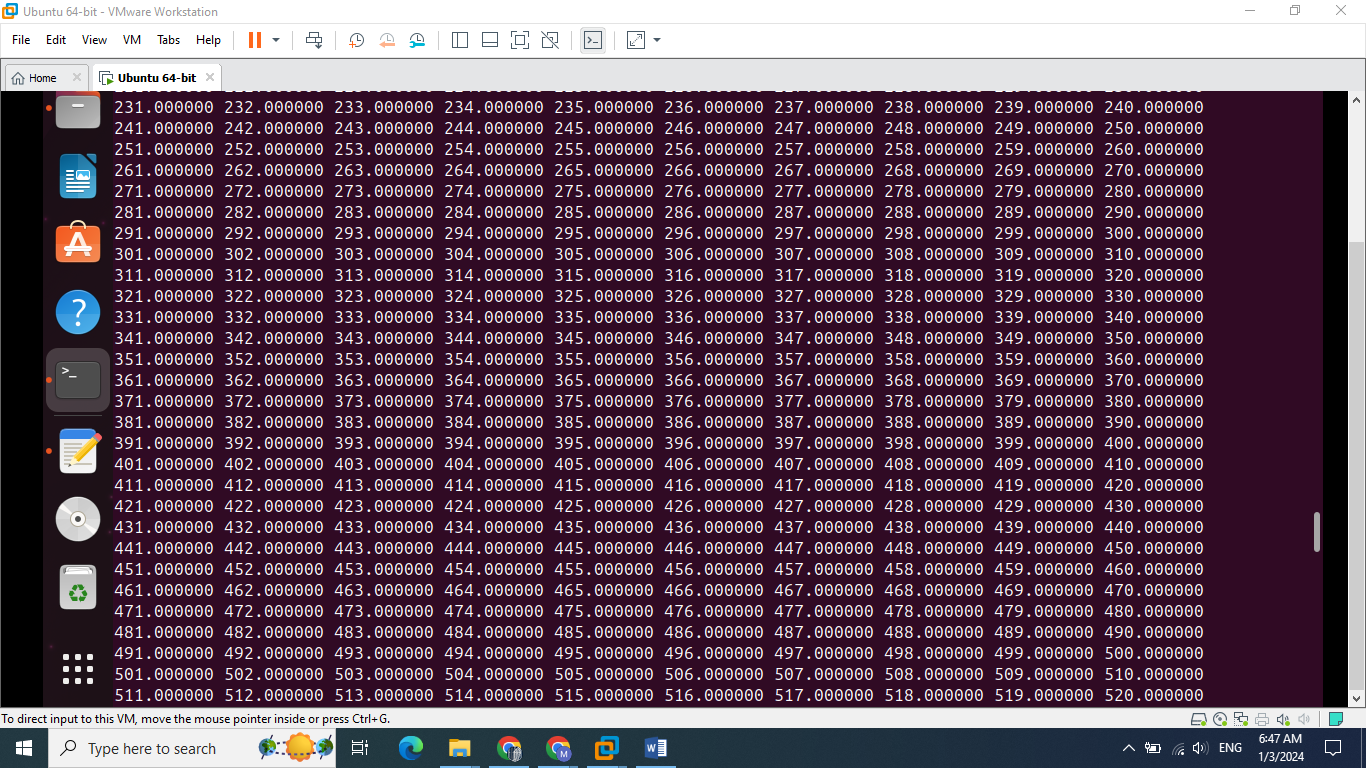
output image:

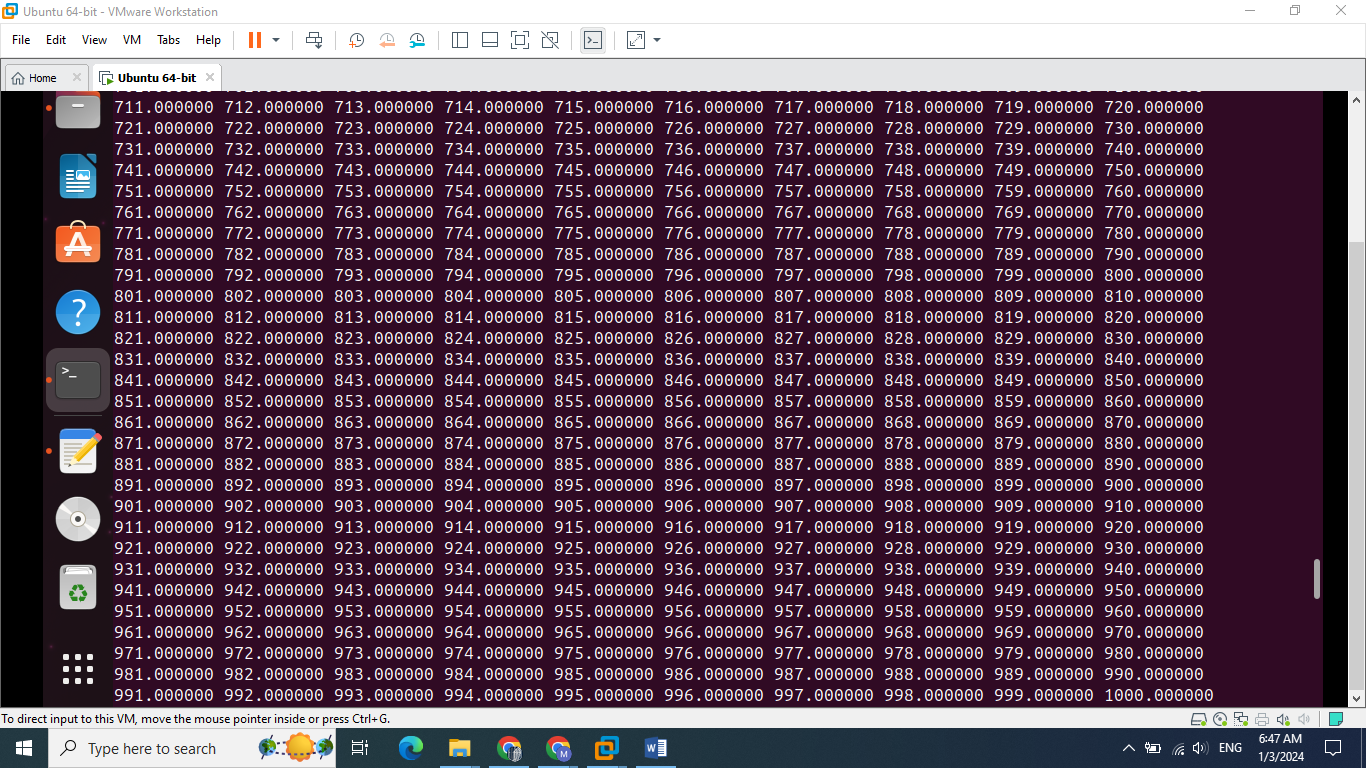


1. Big Matrix(100x10):

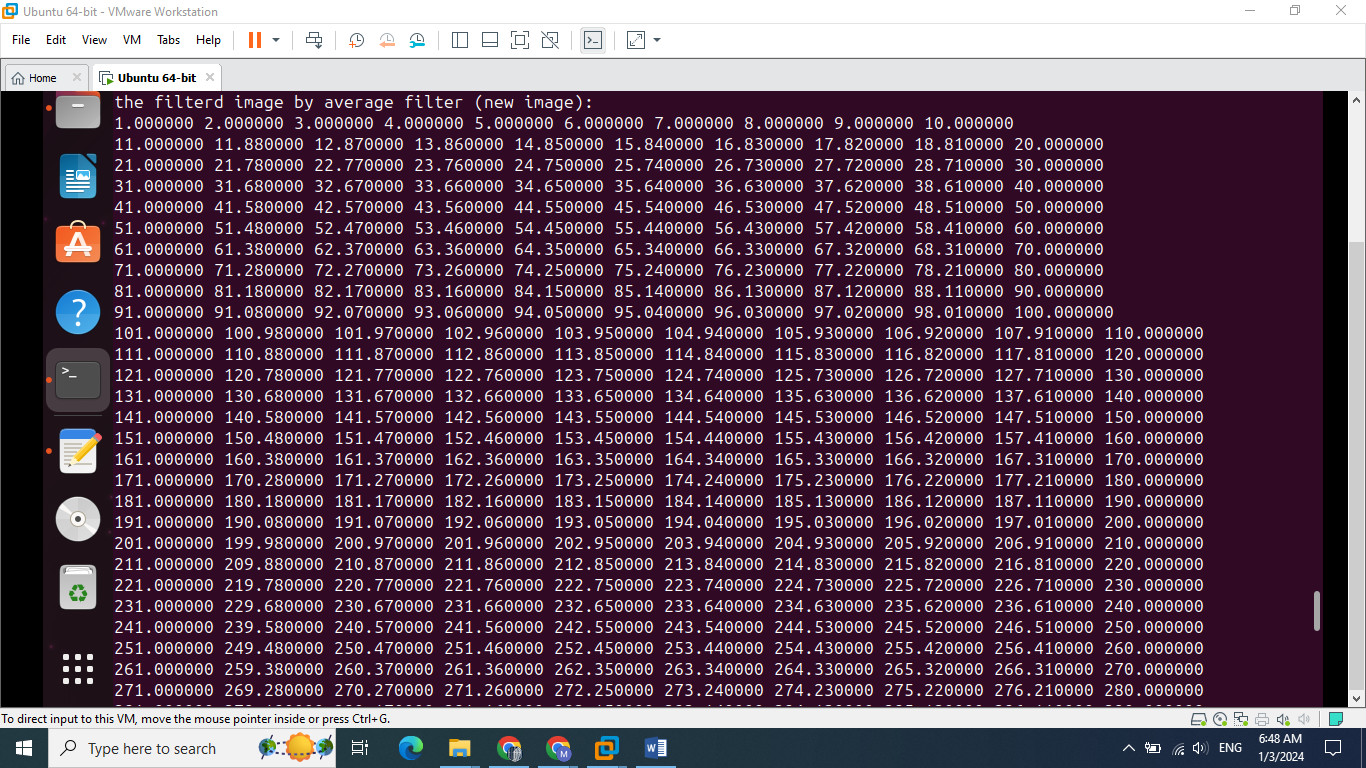
Input image:

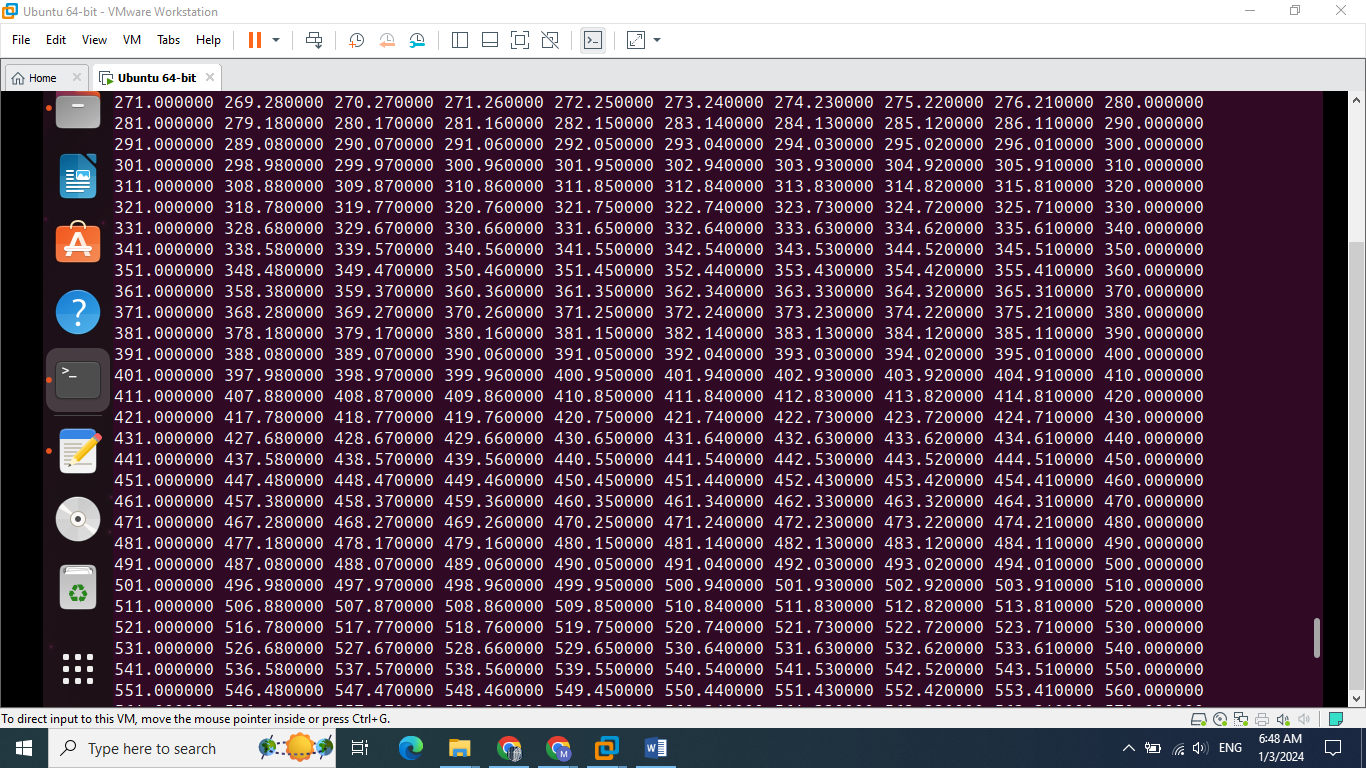


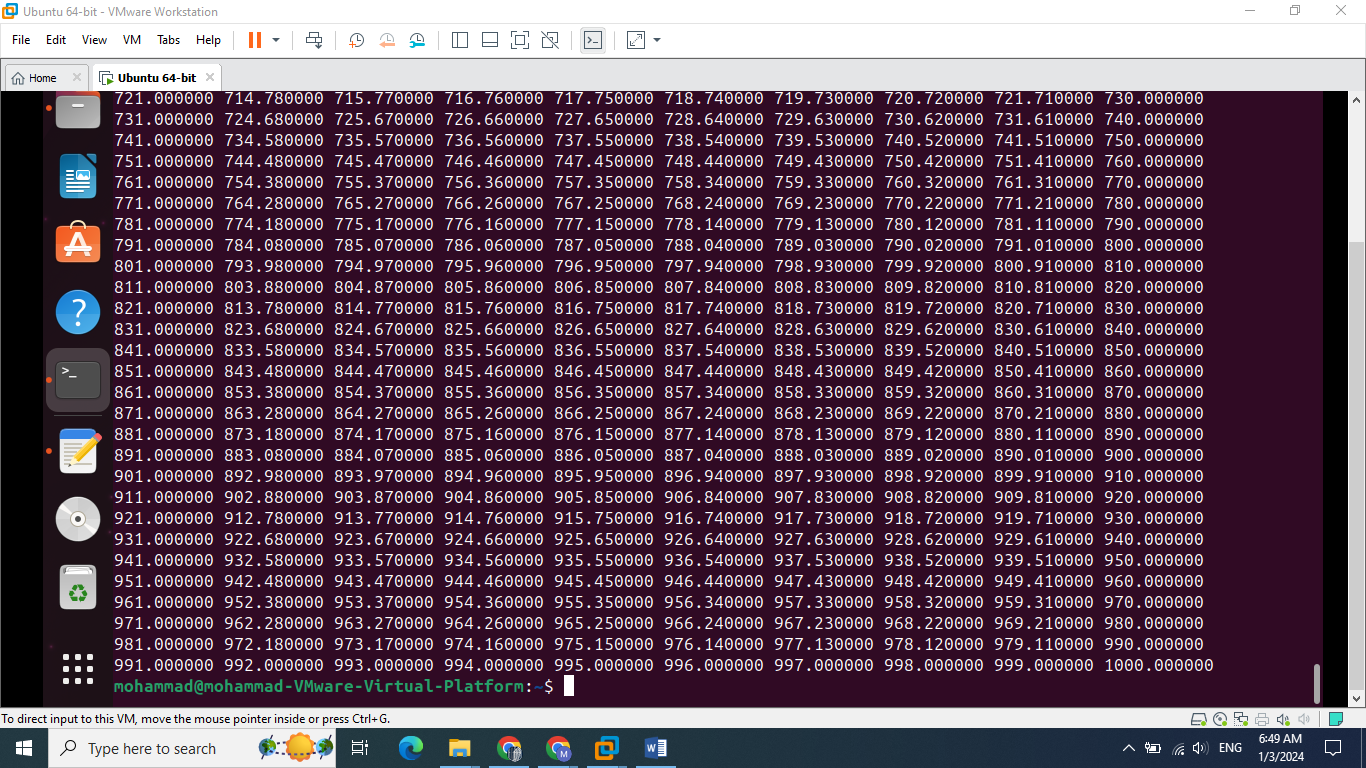




Output image:







**B) Using pthread library:**

This code employs Pthread to parallelize the application of a filter on an image matrix. It initializes matrices for the original image data and the filtered result, along with a mutex for synchronization. The slave function defines the task distribution among threads, enabling each to handle a specific segment of the image. Threads simultaneously compute the filter on their allocated portions, guarded by a mutex to ensure exclusive access to the shared matrix. The main function creates multiple threads, assigns them sections of the matrix to process, waits for their completion, and then prints both the original and filtered images.

**The Code of pthread:**

#include <pthread.h>

#include <stdio.h>

#define num\_threads 6

#define maxsize\_row 50

#define maxsize\_col 10

double filter\_average[3][3]={{0.11,0.11,0.11},{0.11,0.11,0.11},{0.11,0.11,0.11}};

pthread\_mutex\_t mutex1;

double data[maxsize\_row][maxsize\_col];

double data\_filterd[maxsize\_row][maxsize\_col];

void fill\_Matrix(double m[maxsize\_row][maxsize\_col])

{

int c=1;

for (int i = 0; i < maxsize\_row; i++)

{

for (int j = 0; j < maxsize\_col; j++)

{

m[i][j] =c;

c++;

}

}

}

void print\_Matrix(double m[maxsize\_row][maxsize\_col])

{

for(int a=0;a<maxsize\_row;a++)

{

//printf("row number %d:",a);

for(int b=0;b<maxsize\_col;b++)

printf("%f ",m[a][b]);

printf("\n");

}

}

void \*slave(void \*tn)

{

long threadN=(long )tn;

int x=(maxsize\_row-2)/num\_threads;

int rest=(maxsize\_row-2) % num\_threads;

int start=rest+1;

int low=0,high=0;

if(threadN==0)

{

low=1;

high=start+x;

}

else

{

low=threadN\*x+start;

high=low+x;

}

for(int a=low;a<high;a++)

for(int b=1;b<maxsize\_col-1;b++)

{

pthread\_mutex\_lock(&mutex1);

data\_filterd[a][b]=(data[a-1][b-1]\*filter\_average[0][0]) + (data[a-1][b]\*filter\_average[0][1]) + (data[a-1][b+1]\*filter\_average[0][2]) + (data[a][b-1]\*filter\_average[1][0]) +(data[a][b]\*filter\_average[1][1]) + (data[a][b+1]\*filter\_average[1][2]) + (data[a+1][b-1]\*filter\_average[2][0]) + (data[a+1][b]\*filter\_average[2][1]) + (data[a+1][b+1]\*filter\_average[2][2]);

pthread\_mutex\_unlock(&mutex1);

}

}

int main()

{

fill\_Matrix(data);

fill\_Matrix(data\_filterd);

pthread\_mutex\_init(&mutex1, NULL);

pthread\_t thread[num\_threads];

for (long i = 0; i < num\_threads; i++)

pthread\_create(&thread[i], NULL, slave, (void \*) i);

for (long i = 0; i < num\_threads; i++)

pthread\_join(thread[i], NULL);

printf("\n\nthe original image (old image):\n");

print\_Matrix(data);

printf("\n\nthe filterd image by average filter (new image):\n");

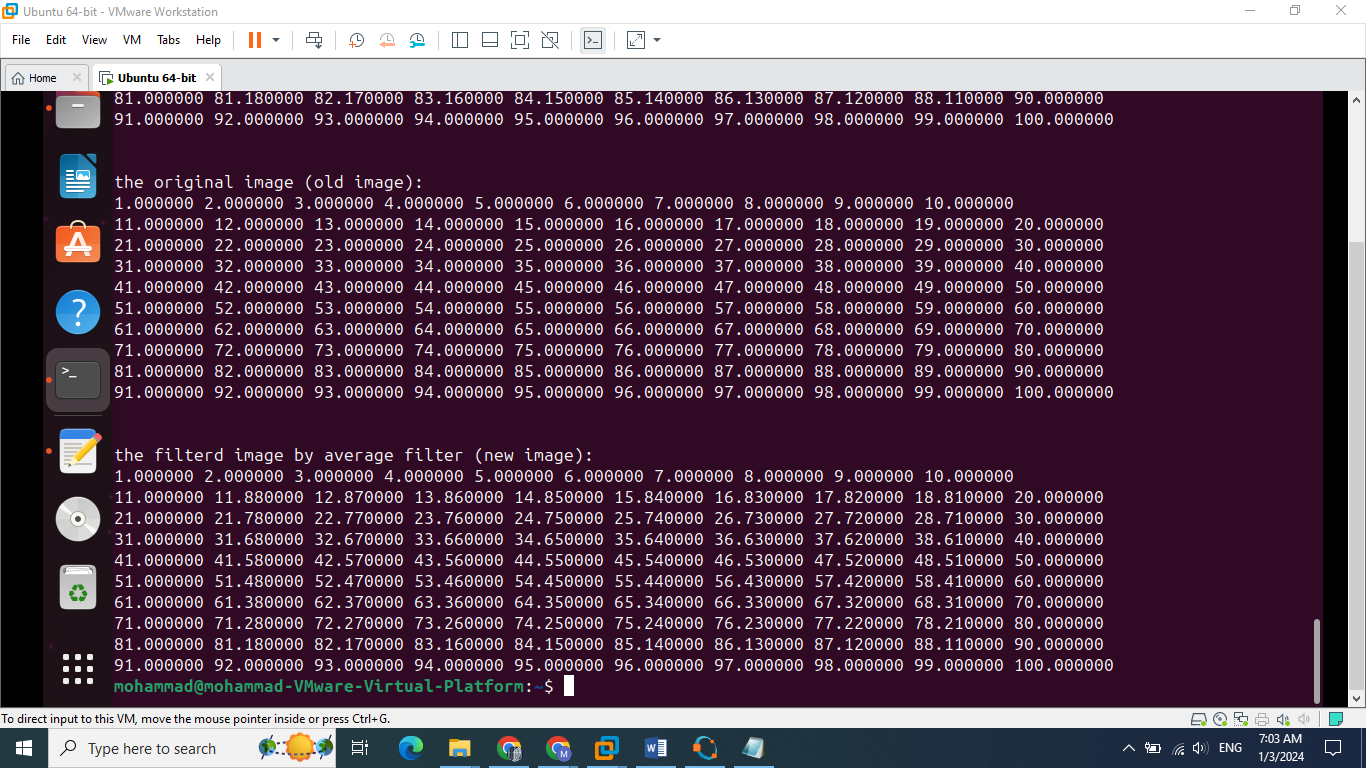
print\_Matrix(data\_filterd);

return 0;

}

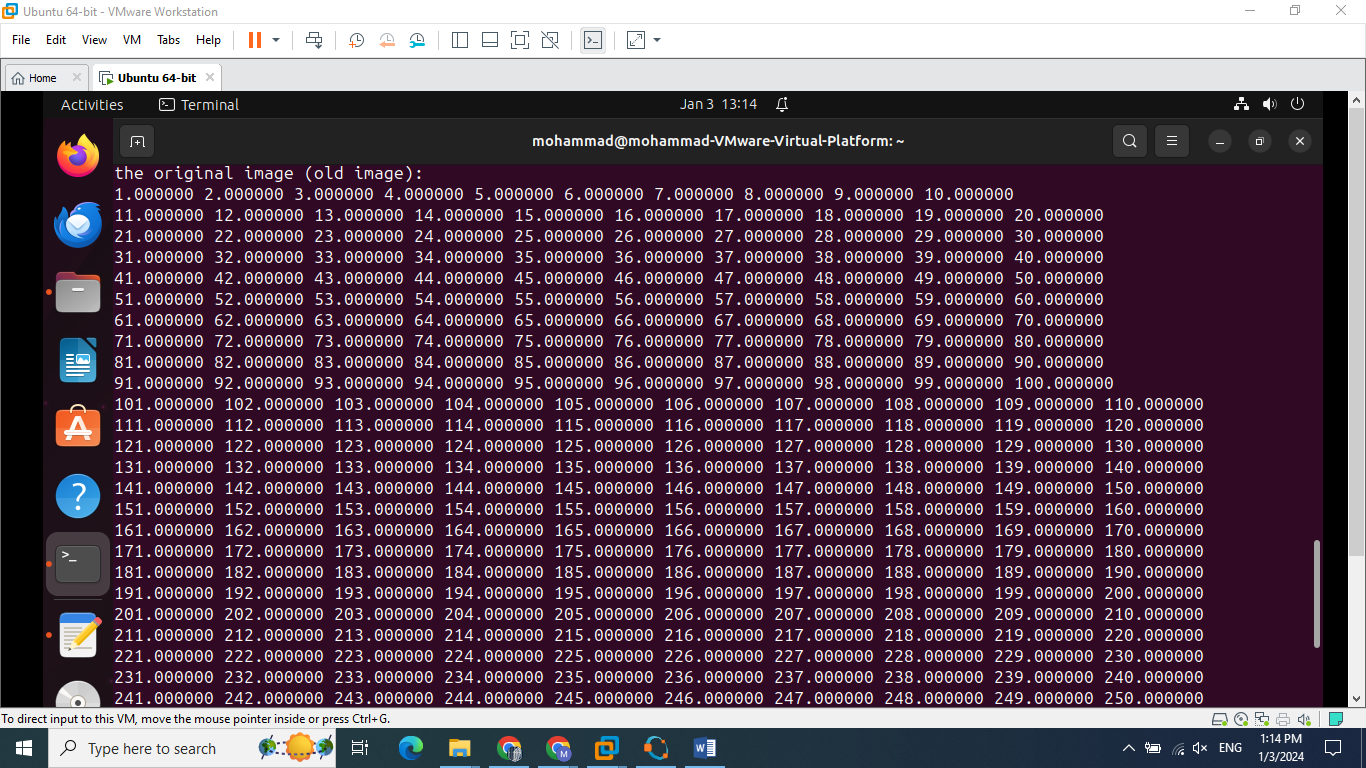
**Test Cases for Pthread:**

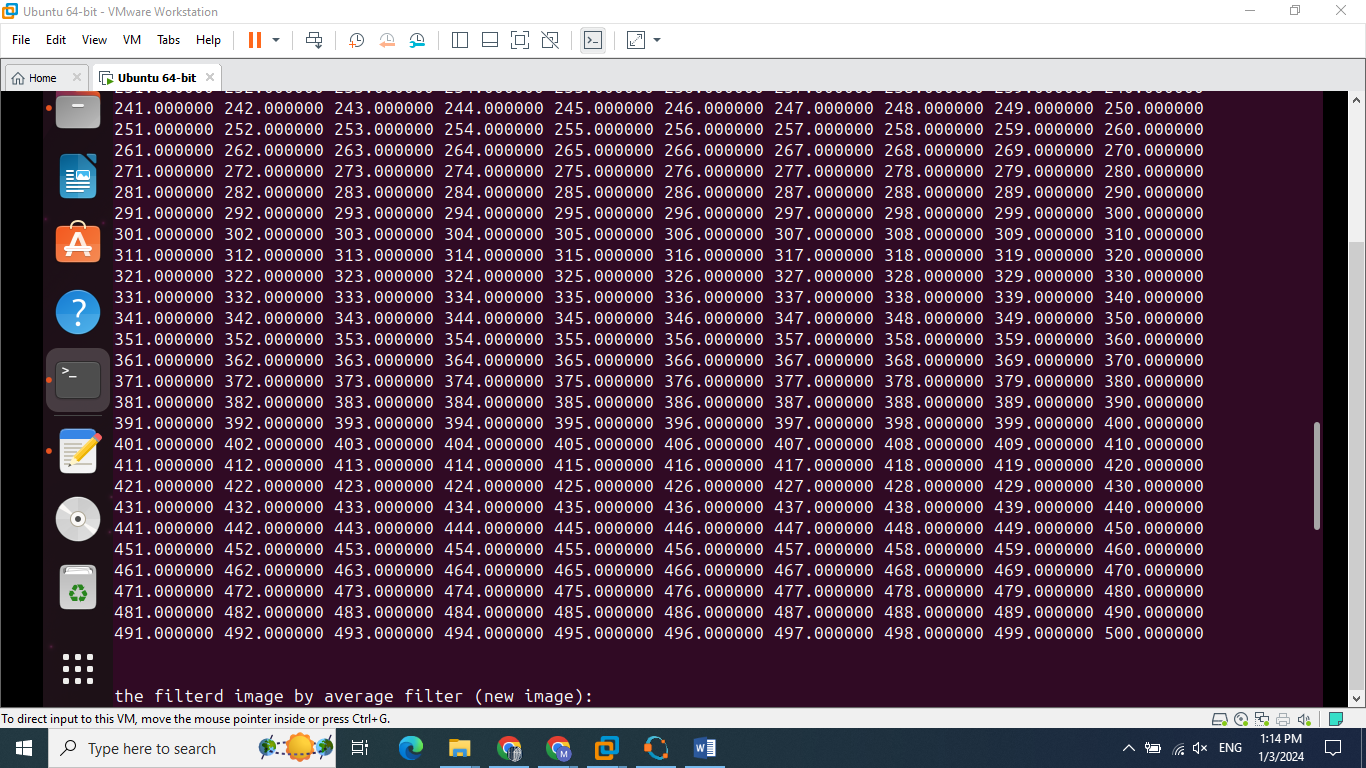
1) Small matrix(10x10):



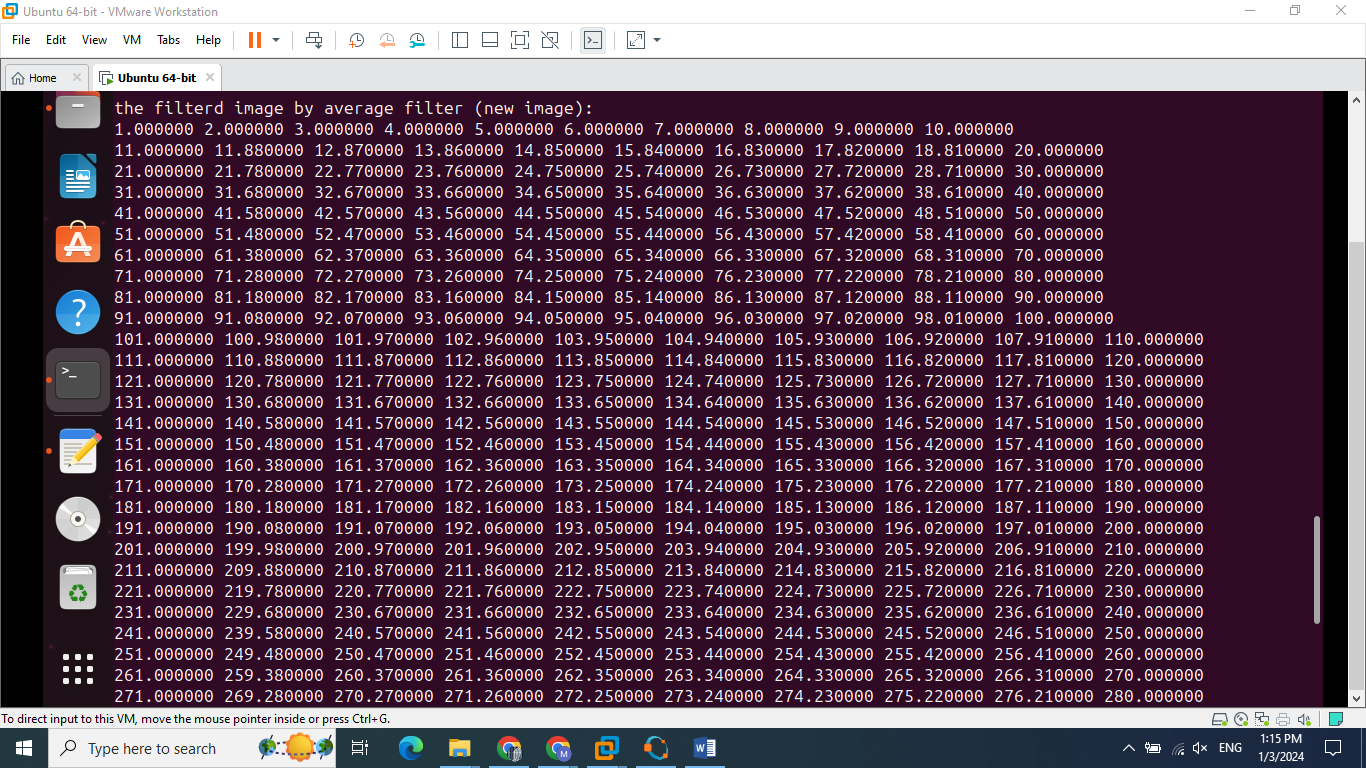
2)Medium matrix(50x10):

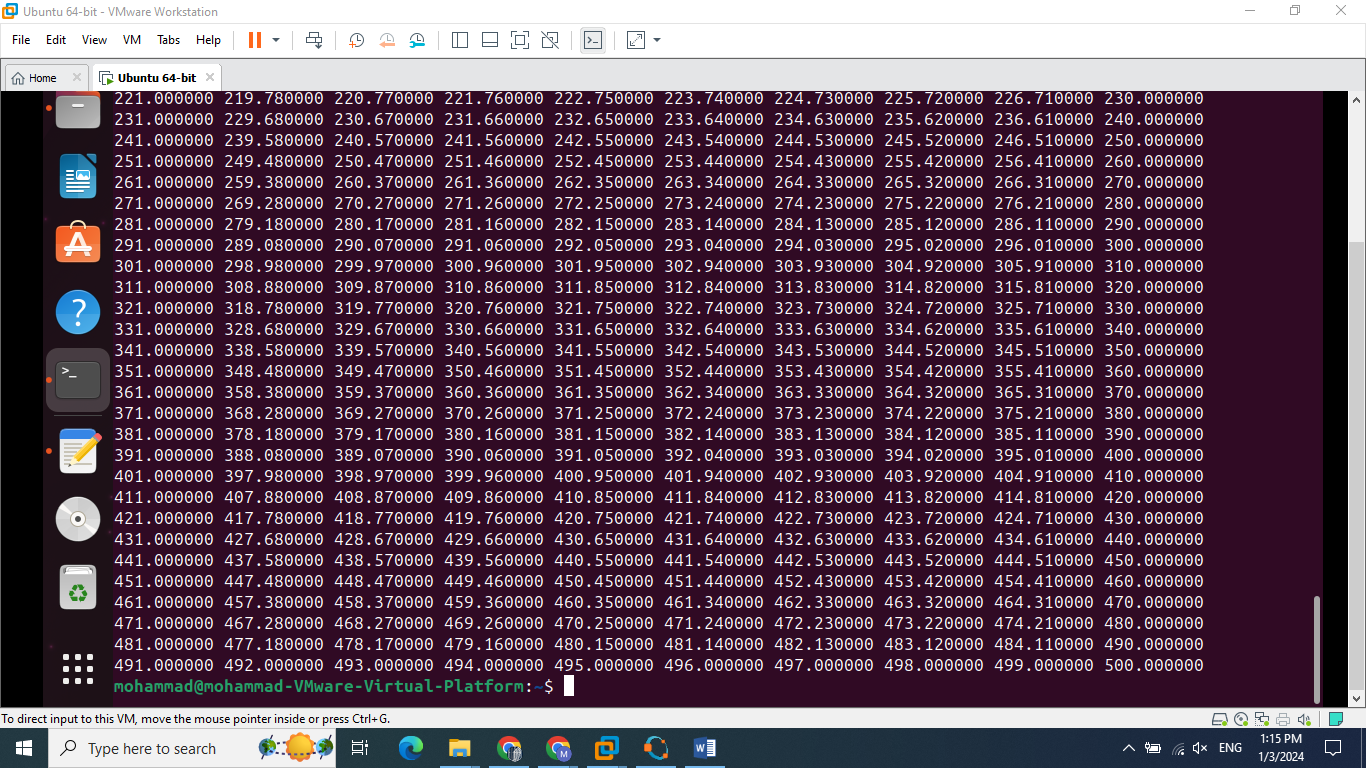
Input image:





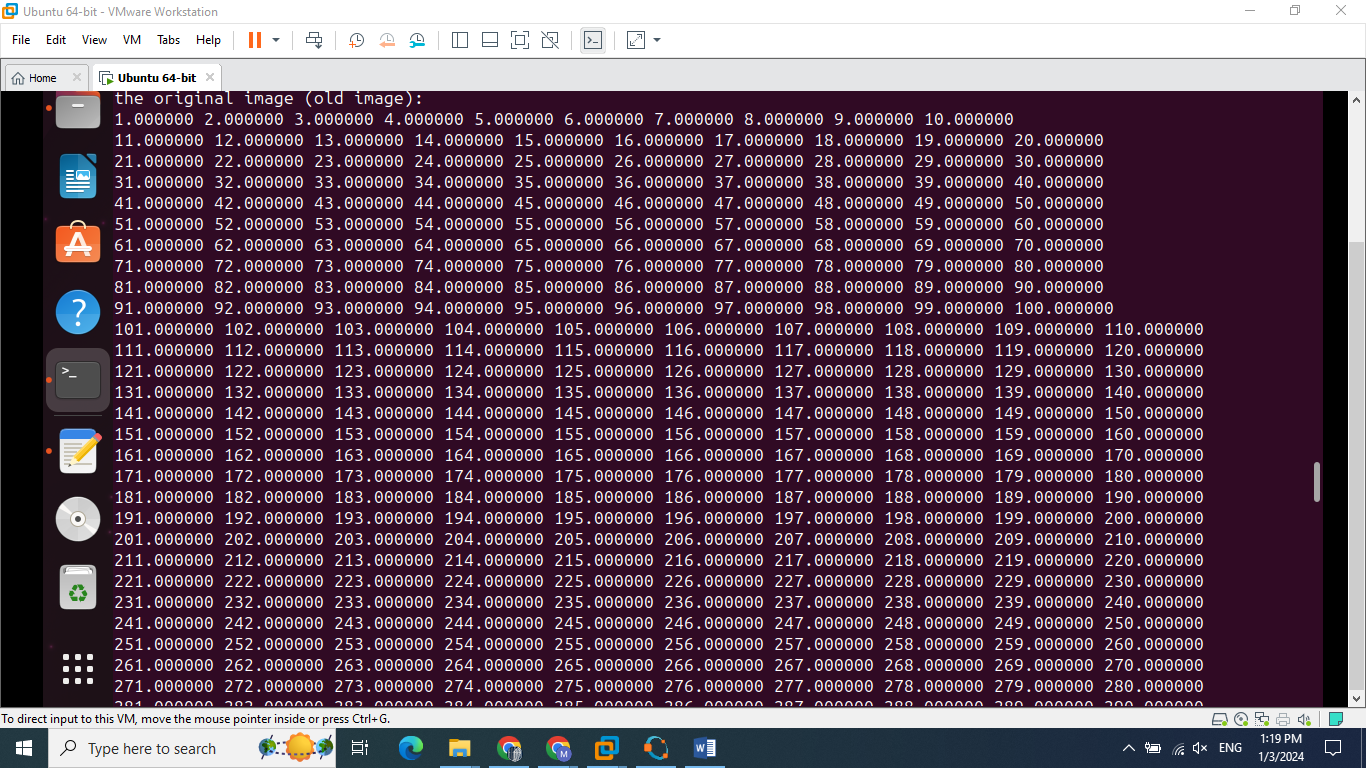
Output image:

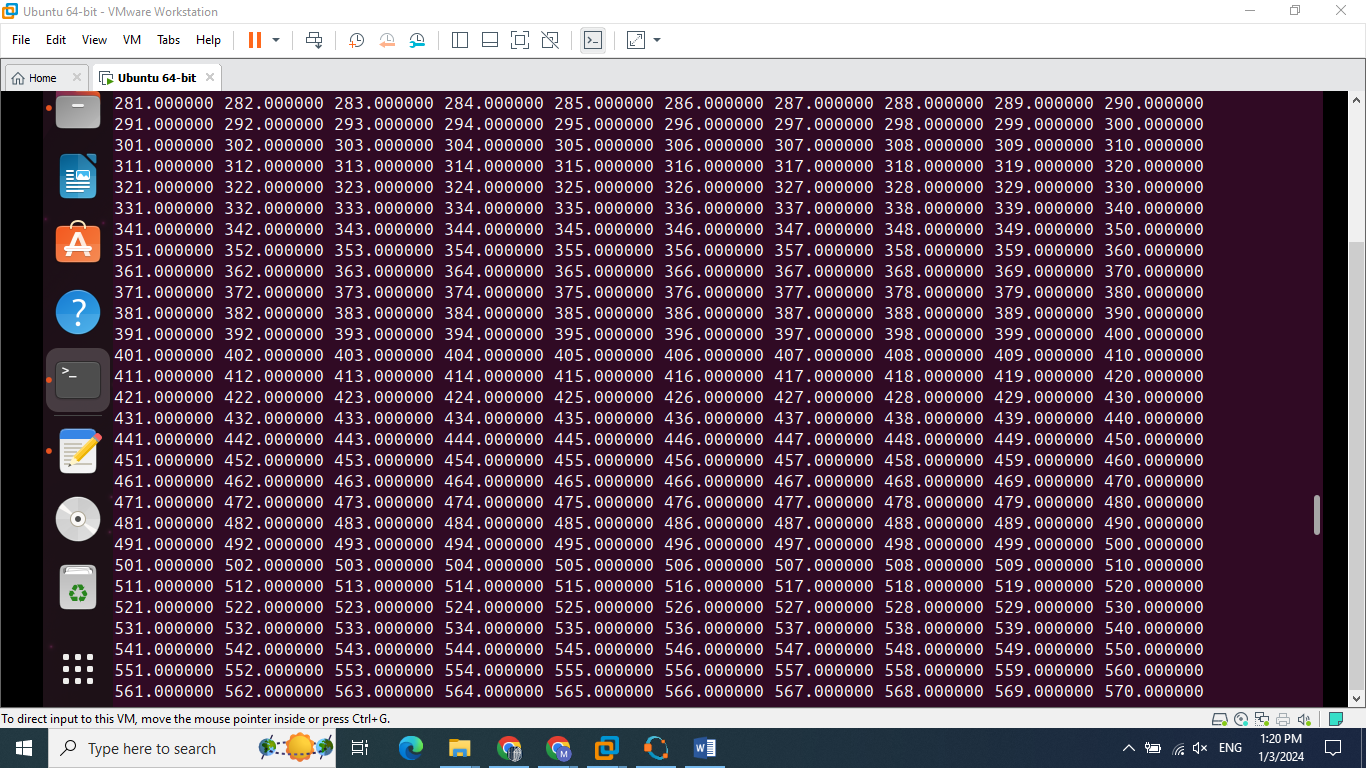


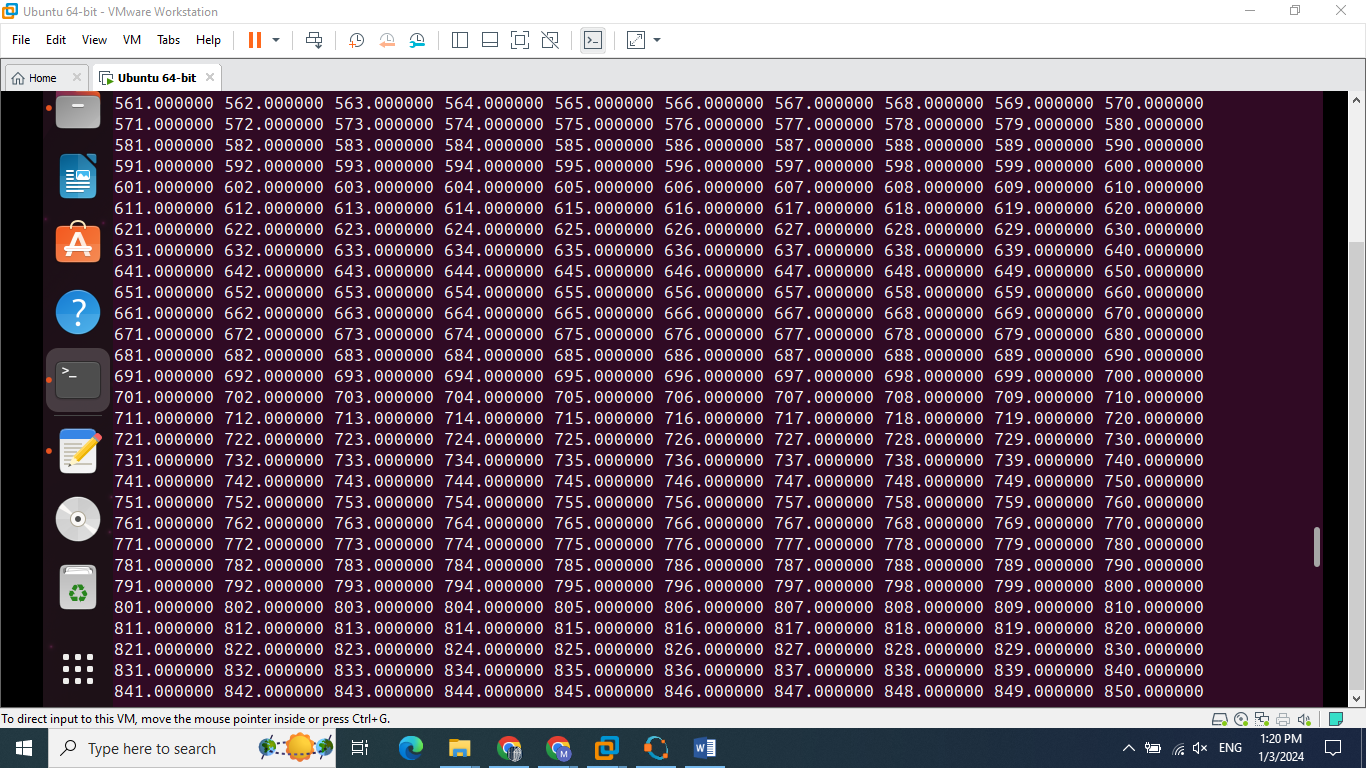


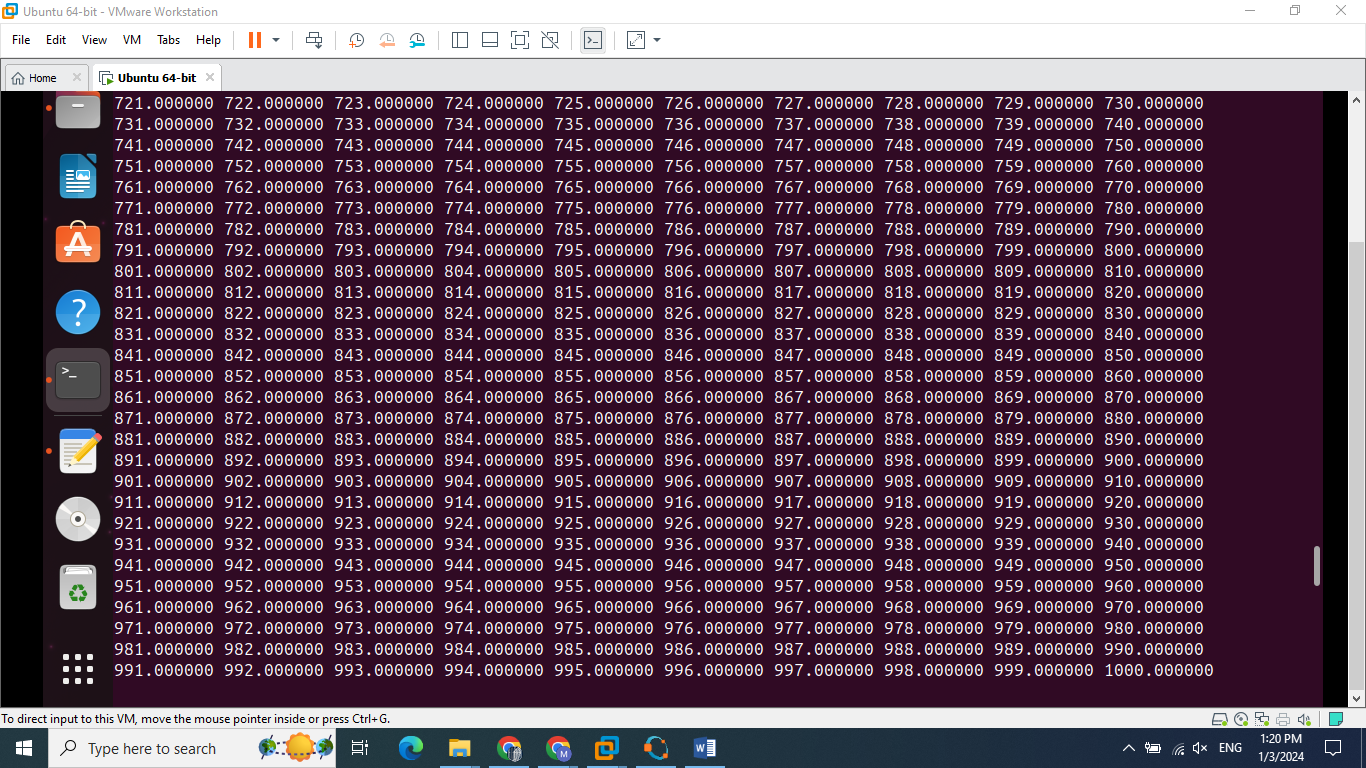
3)Big matrix(100x10):

Input image:

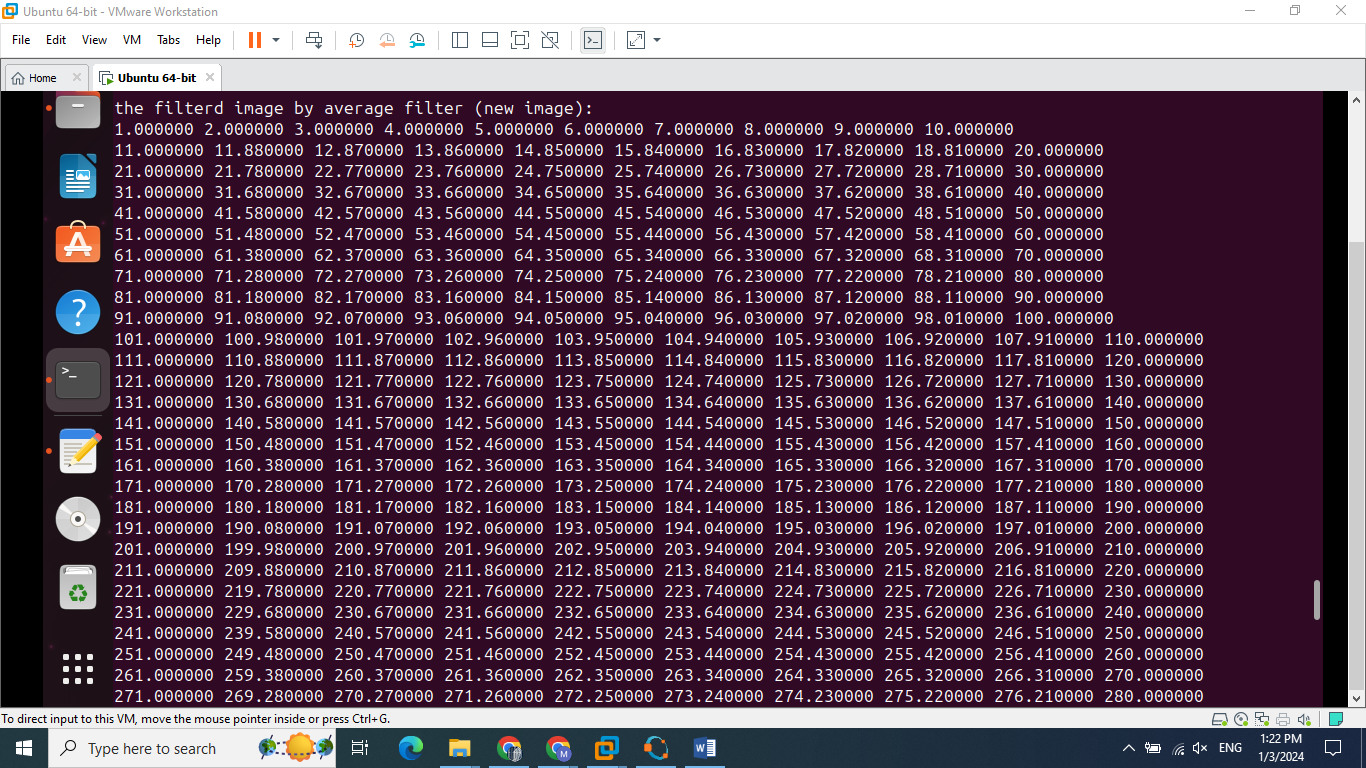


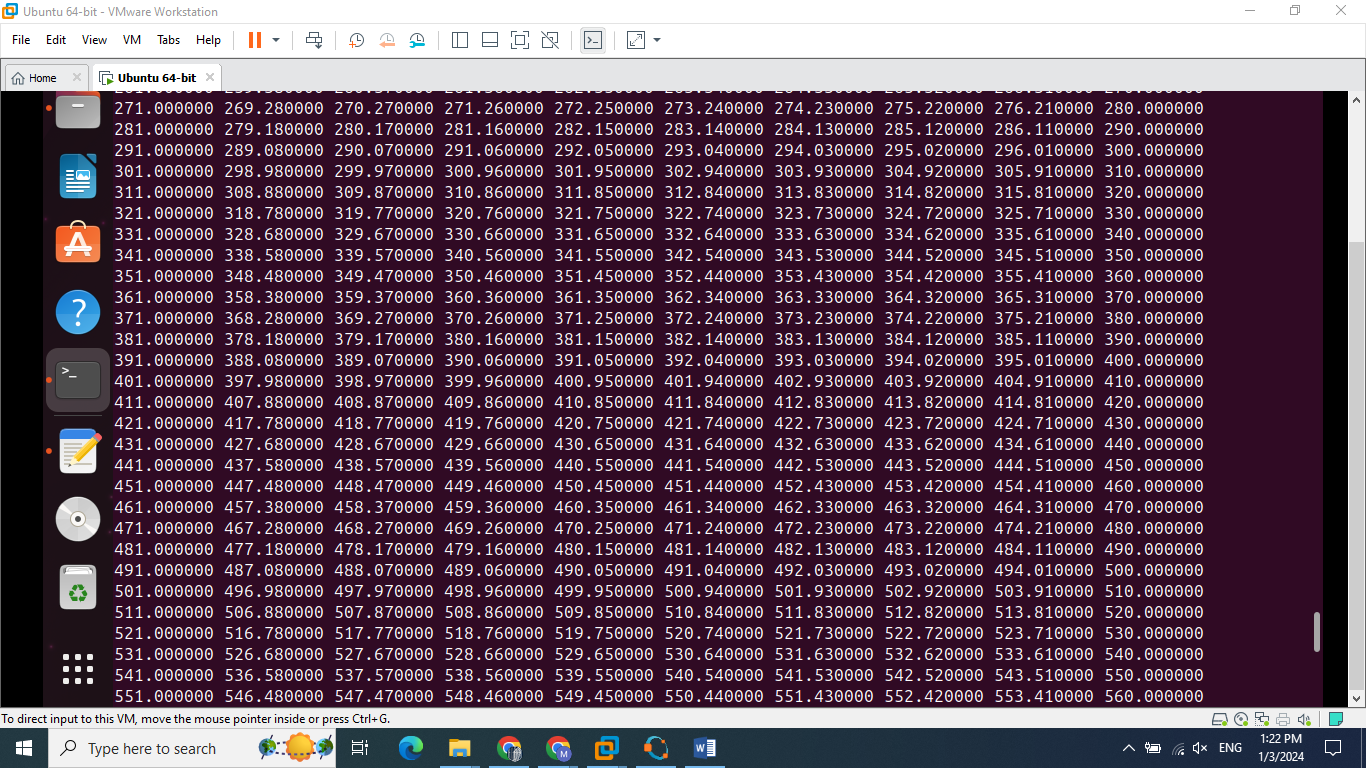


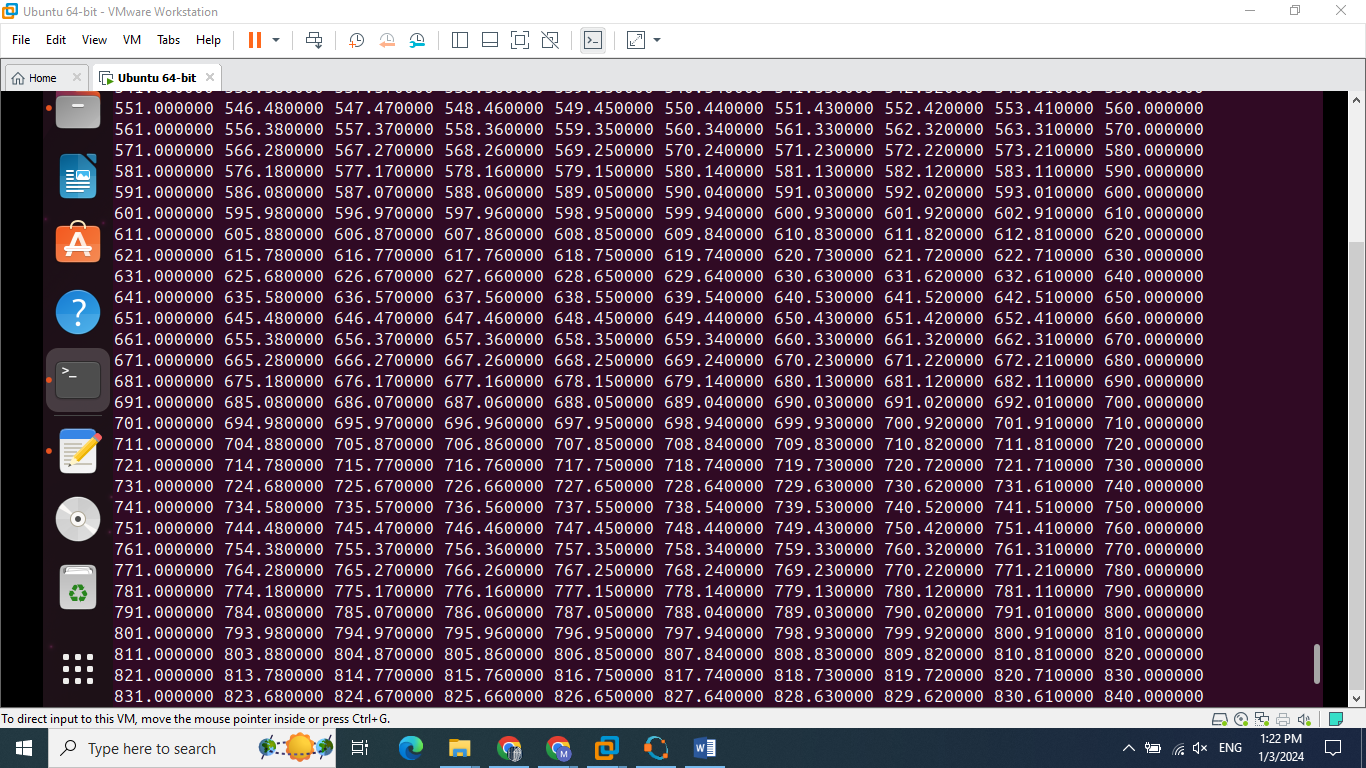


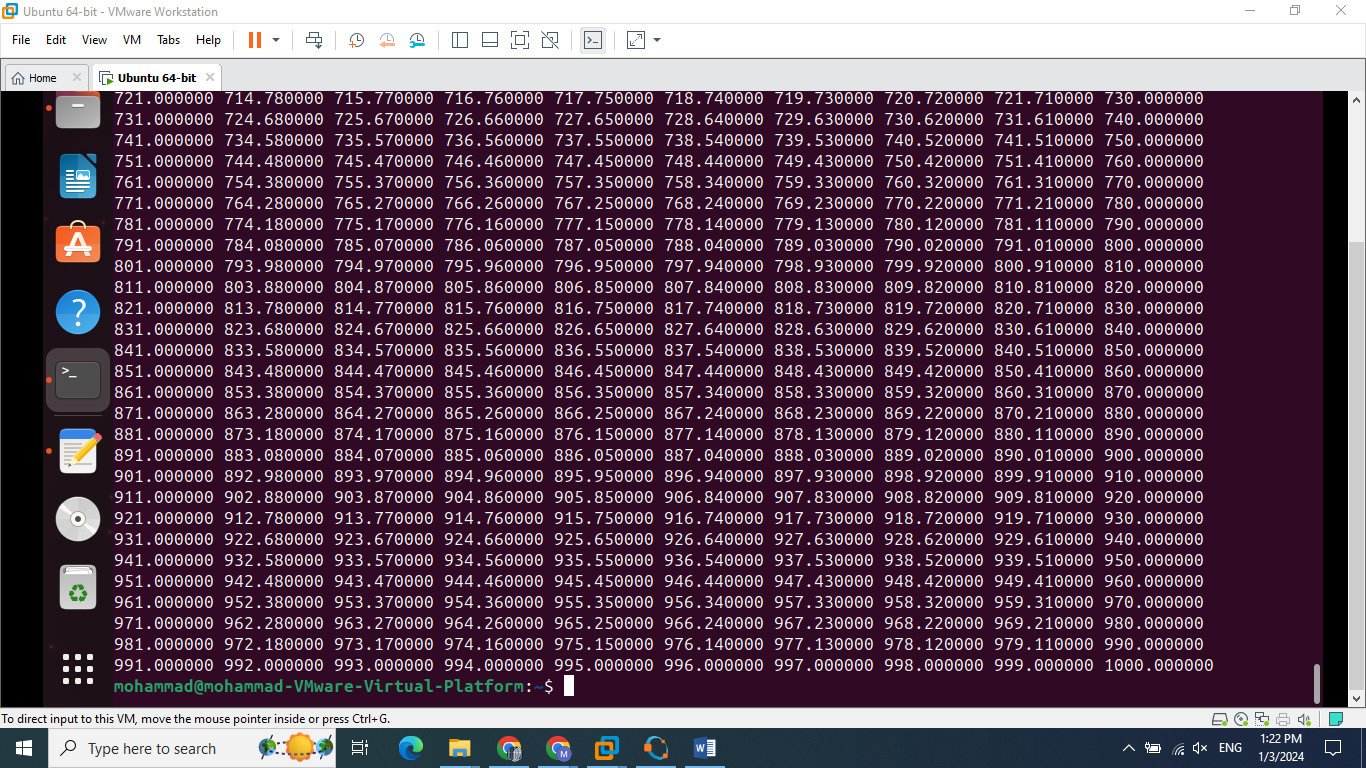


Output image:









**C)Using the filter on real image:**

1)We will use octave to save pixel values into text file and name it “input.txt”.

2)The program will read pixel values from “input.txt” and apply average filter on the image

3)The program will save the filtered image in text file named “output.txt”.

4)Octave is used to read “output.txt” values and display it as an image.

**Sample:**

Input Image:



Output Image:

