Assignment-2

Aarchie[200004]
Udit Prasad [201055]

Coding methodology:

i) First, for each process, we have initialize matrix of dimensions (n/py) x (n/px) which represents the subdomain of that process using random double values. Say it is “mat”.

ii) Then for each process, we calculate the number of elements it need to send to the process below it (according to the decomposition). Say for the following example

7,00	7,1 0	7,2 0	7,3 0	7,4 1	7,5 1	7,6 1	7,7 1
6,0 0	6,1 0	6,2 0	6,3 0	6,4 1	6,5 1	6,6 1	6,7 1
5,0 2	5,1 2	5,2 2	5,3 2	5,4 3	5,5 3	5,6 3	5,7 3
4,0 2	4,1 2	4,2 2	4,3 2	4,4 3	4,5 3	4,6 3	4,7 3
3,0 4	3,1 4	3,2 4	3,3 4	3,4 5	3,5 5	3,6 5	3,7 5
2,0 4	2,1 4	2,2 4	2,3 4	2,4 5	2,5 5	2,6 5	2,7 5
1,0 6	1,1 6	1,2 6	1,3 6	1,4 7	1,5 7	1,6 7	1,7 7
0,0 6	0,1 6	0,2 6	0,3 6	0,4 7	0,5 7	0,6 7	0,7 7

This is 8 x 8 matrix with px = 4 and py = 2
Each cell contains the index of that cell along with the rank of the process of which it is part.
Rank 0 will send 3 elements to rank 2,
Rank 1 will send 0 elements to rank 3,
Rank 2 will send 4 elements to rank 4,
Rank 3 will send 1 element to rank 5,
Rank 4 will send 4 elements to rank 6,
Rank 5 will send 3 elements to rank 7

Represent last row of each sub-domain as “last_row”

This can be generalised using **(((myrank/px)+1)*(n/py))+1 - ((myrank%px)*(n/px))** —--(I)
Where (((myrank/px)+1)*(n/py))+1 is the total number of elements to be sent from each row to the row below it (here row refers to the processes which share boundaries left to right).

And ((myrank%px)*(n/px)) is the number of elements sent by ranks present in that row before that rank i.e., part of the subdomain which lies left to it.

If this expression becomes greater than length of last_row then expression taken value equal to last_row.(because a process can only send elements that it own)

And as we have stored the subdomain in a matrix, the start address will be simply **mat[(n/py)-1]**

Similarly, we can calculate the number of elements one has to receive from the upper process.

iii) Now, we will do MPI_sends and MPI_receives using the parameters we calculated.

iv) We updated the elements of same subdomain using a temporary matrix, which will store the value for (t+1)th time step and then we will update the mat, and we will update the first row of every mat (which uses elements from other subdomain) using the received buffer.

OBSERVATION:

All the observations of small data size are made using prutor and that of bigger data size is made using csews directly. Number of iterations are 20.

i) Total number of MPI_Sends and receives

It is less in 2-d domain decomposition as compared to 1-d domain one as all the process which have their last row of matrix in their subdomain does not send any data, similarly some other ranks which are in upper triangular matrix need not do any communication.

ii) Communication/ Computation ratio:

- **1d domain decomposition** - All process sends to the process below it and receives from the process adove it except the boundary cases. Also, it sends only that much data which is required.
  for p processes with n x n matrix, each rank will send (myrank+1)*(n/p)+1 number of elements
  And computation will be of the order (n^2)/p.
  Ratio = (myrank+1)*(n/(px*py))+1 / ((n^2)/p)
- **2d domain decomposition -**All process sends to the process below it only if the below process is part of lower triangular matrix . Similarly a process receives only if it is part of

lower triangular matrix.
Also, it sends only that much data which is required.
For ex- In the figure shown on above page, rank 1 does not send any data and other process send and receive as per requirement.

Number of Computation will be same as 1d.
Ratio = (((myrank/px)+1)*(n/py))+1 - ((myrank%px)*(n/px)) / ((n^2)/p)

Hence, 2d one has less communication/computation ratio.

iii) Load Balancing

In 1-d case, every process does some computation and communication but the ranks present in lower part of matrix has higher load as compared to upper ones.

But in 2-d case, on top of load imbalance as in 1-d case, there are some processes which sits idle . for eg- those in upper triangular matrix.

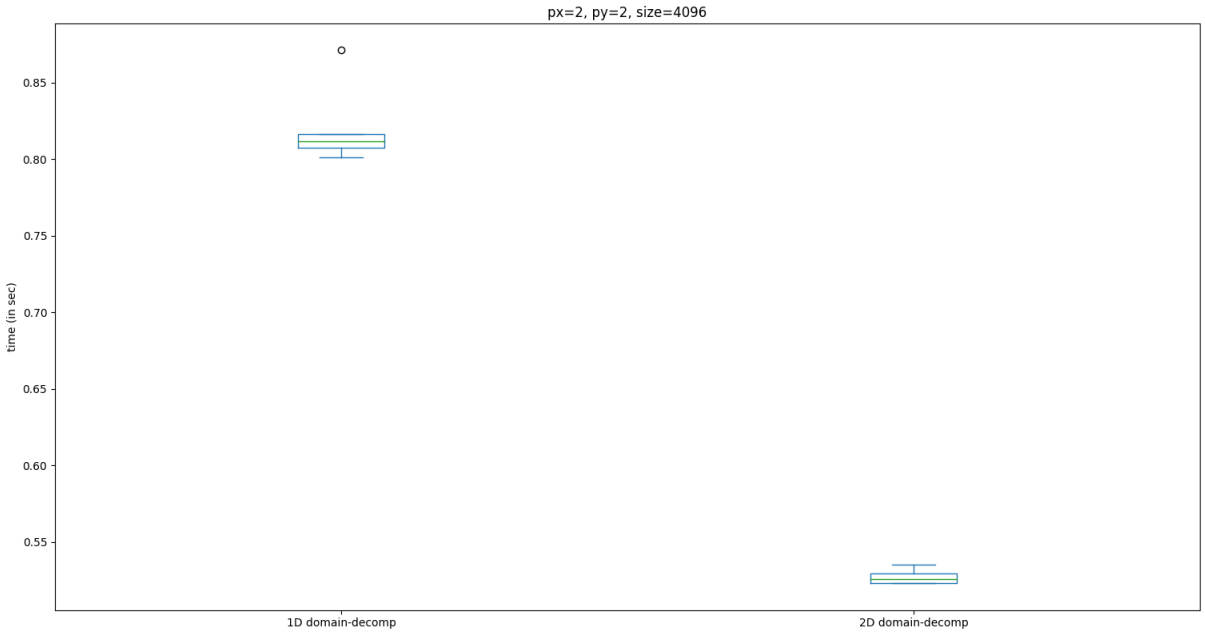
iv) Time

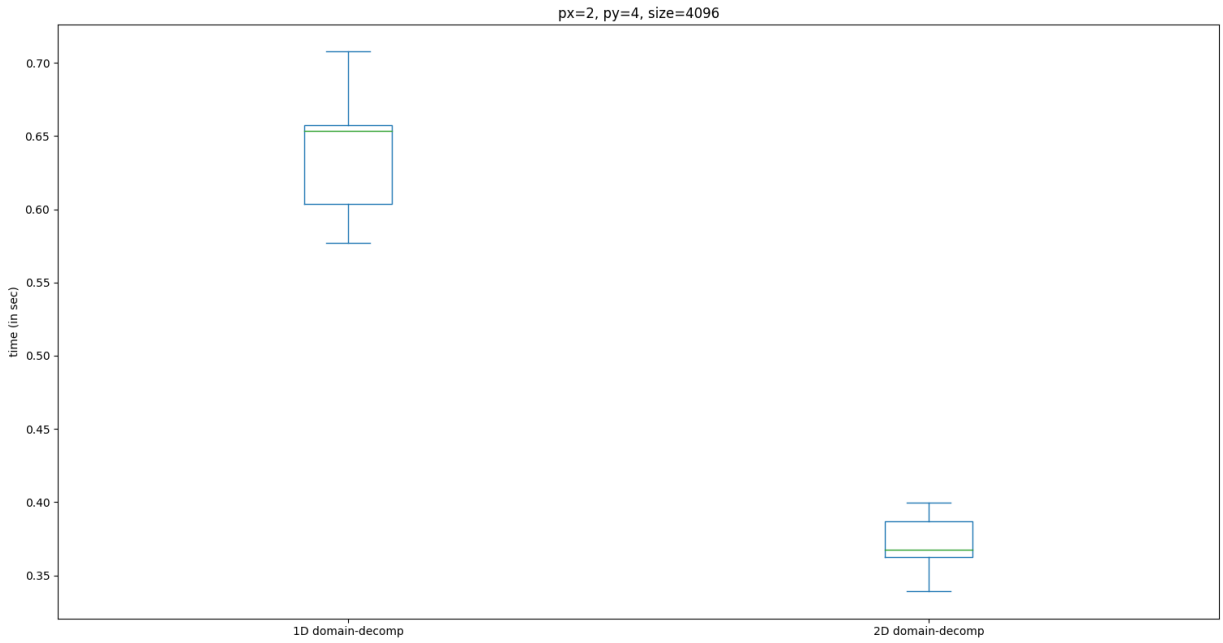
Time taken can be seen in 3 aspects: Domain decomposition, number of processes and size of matrix.

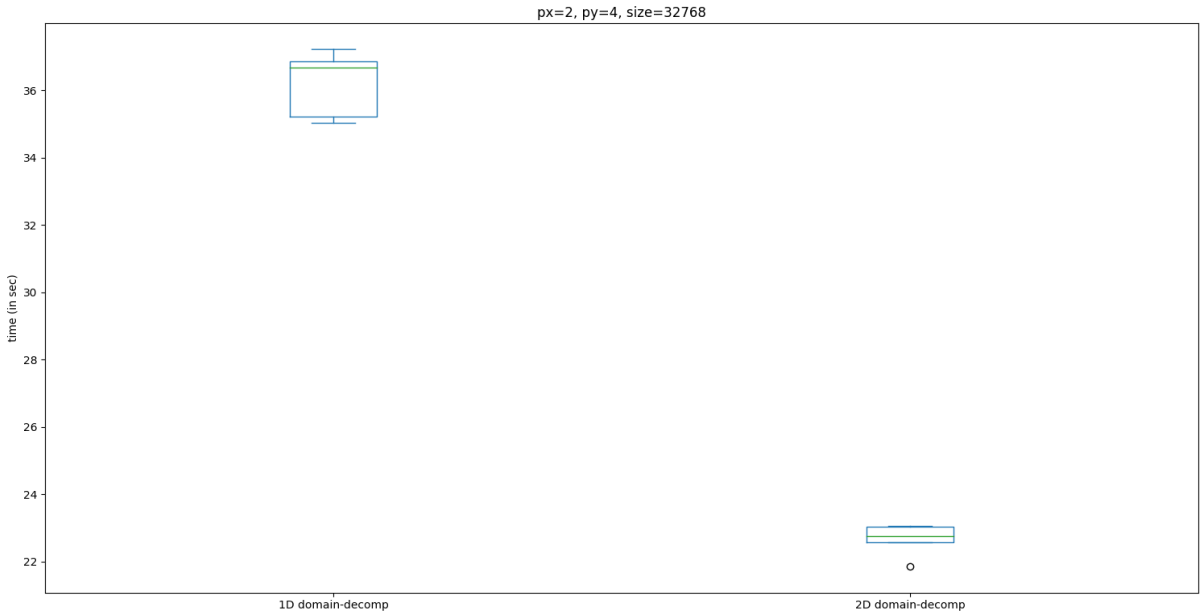
- Number of processes:
  Time decreases as we increase the number of processes,
  It is because now each process needs to compute and communicate less and all the processes are doing so in parallel . Hence time decreases by increasing number of processes.
  This shows that code is scalable.
- Size of matrix:
  It is obvious that by increasing the size of the matrix, we will increase the work load on each process and hence time will increase.
- Domain decomposition
  Time taken is larger in 1-d domain as compared to 2-d case.
  -> This is also supported by the observation of communication/ computation ratio.
  -> Also, in 1d case, processes send and receive large number of elements which increases copy buffer time as compared to 2-d one.

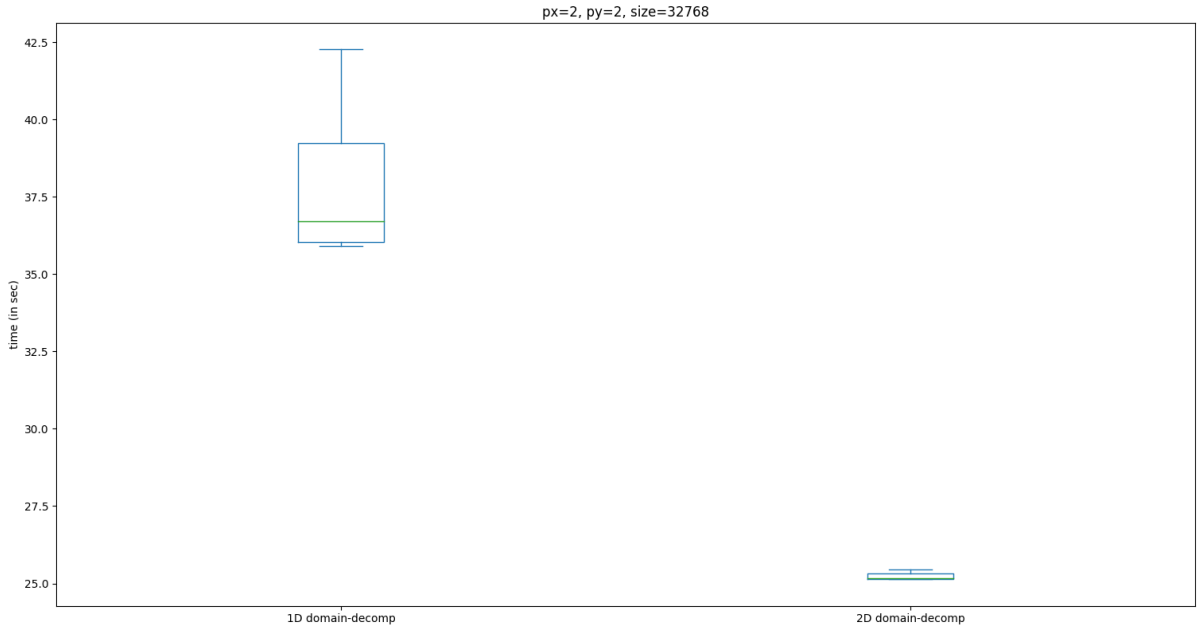
PLOTS:

We have shown all the boxplots in same graph, but as it is very unclear , we have also shown some graphs to emphasize the difference in 1-d domain decomposition and 2-d domain decomposition.

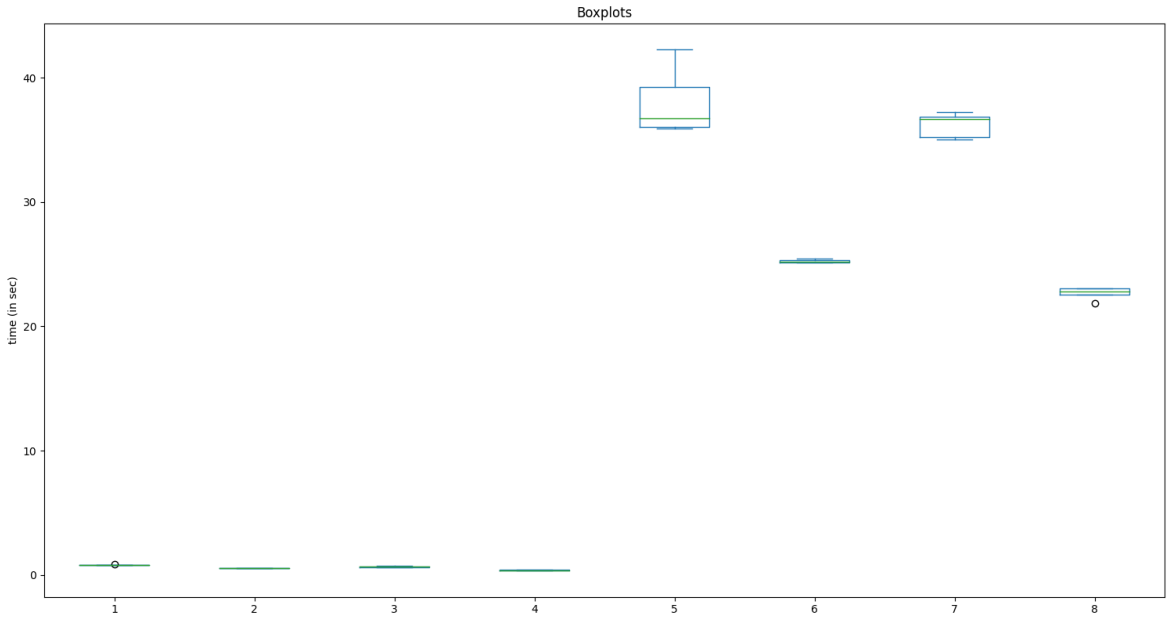





px=2, py=4, size=32768
time (in sec)
36
34
32
30
28
26
24
22
1D domain-decomp
2D domain-decomp


px=2, py=2, size=32768
time (in sec)
42.5
40.0
37.5
35.0
32.5
30.0
27.5
25.0
1D domain-decomp
2D domain-decomp

Below is the plot which shows all the 8 boxplots in same graph



Legend:

1 - px=2, py=2, size=4096, 1d-domain
2 - px=2, py=2, size=4096, 2d-domain
3 - px=2, py=4, size=4096, 1d-domain
4 - px=2, py=4, size=4096, 2d-domain
5 - px=2, py=2, size=4096, 1d-domain
6 - px=2, py=2, size=4096, 2d-domain
7 - px=2, py=4, size=4096, 1d-domain
8 - px=2, py=4, size=4096, 2d-domain

CODE

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "mpi.h"

int main(int argc, char *argv[])
{
	int myrank, size;
	MPI_Status status;
	double sTime, eTime;

	// This is for 2-d decomposition

	int process_rows = atoi(argv[2]);		//Number of processes in y axis
	int process_cols = atoi(argv[1]);		//Number of processes in x axis

	char *eptr;
	long long n = strtoll(argv[3], &eptr, 10);

	long long int rows = n / process_rows;	// Number of rows in a sub-domain
	long long int cols = n / process_cols;	// Number of cols in a sub-domain

	double matrix[rows][cols];			// Each rank owns its subdomain
	double buffer[cols + 2];			// for receiving

	MPI_Init(&argc, &argv);

	MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
	MPI_Comm_size(MPI_COMM_WORLD, &size);

	// initializing matrix
	srand((unsigned int)time(NULL) + myrank);
	for (long long int i = 0; i < rows; i++)
	{
	for (long long int j = 0; j < cols; j++)
	{
	int x = rand();
	int y = rand();
	if (y == 0)
	y = 1;
	matrix[i][j] = (double)x / (double)y;
	}
	}

	long long int count = ((((myrank / process_cols) + 1) * rows) + 1); //  total number of elements to be sent from each row to the row below it (here row refers to the processes which share boundaries left to right)
	count -= (myrank % process_cols) * cols;			// decrease the number of elements sent by ranks present in that row to the left of that rank i.e., part of the subdomain which lies left to it.
	if (count > 0)
	{
	if (count >= (n / process_cols))
	{
	count = n / process_cols;				// can send only elements of its subdomain
	}
	}
	else
	count = 0;
```

```
if ((myrank / process_cols) == process_rows - 1)                    // count = 0 for last row processes
count = 0;

int recv_rank = myrank - (process_cols);                            // rank present just above it
long long int recv_count = ((((recv_rank / process_cols) + 1) * (n / process_rows)) + 1); // Similar logic as count
recv_count -= (recv_rank % process_cols) * (n / process_cols);
if (recv_count > 0)
{
if (recv_count >= (n / process_cols))
{
recv_count = n / process_cols;
}
}
else
recv_count = 0;
// recv_count = 0 for first row processes
if (recv_rank < 0)
recv_count = 0;

sTime = MPI_Wtime();

for (int i = 0; i < 20; i++)                                        // 20 iterations
{
if (count)
MPI_Send(matrix[rows - 1], count, MPI_DOUBLE, myrank + process_cols, 1, MPI_COMM_WORLD);

if (recv_count)
MPI_Recv(buffer, recv_count, MPI_DOUBLE, recv_rank, 1, MPI_COMM_WORLD, &status);

// update matrix
long long int comp = (myrank / process_cols) * rows - (myrank % process_cols) * cols;   //to compute the number of
elements which need to be updated (lower triangular)

// update in same subdomain using temporary matrix
double temp[rows][cols];
for (long long int i = 1; i < rows; i++)
{
long long int colsToUpdate = comp + i + 1;
if (colsToUpdate >= cols)
colsToUpdate = cols;
for (long long int j = 0; j < colsToUpdate; j++)
{
temp[i][j] = matrix[i][j] - matrix[i - 1][j];
}
}
// now copy the temp matrix into matrix.
for (long long int i = 1; i < rows; i++)
{
long long int colsToUpdate = comp + i + 1;
if (colsToUpdate >= cols)
colsToUpdate = cols;
for (long long int j = 0; j < colsToUpdate; j++)
{
matrix[i][j] = temp[i][j];
}
}

// update using other subdomain
for (long long int j = 0; j < recv_count; j++)
{
matrix[0][j] -= buffer[j];
```

```
//  printf("[%d]--(%lld)%lf ",myrank,j,buffer[j]);
}
}

eTime = MPI_Wtime();

double mtime2d = eTime - sTime;
double maxTime2d;
MPI_Reduce(&mtime2d, &maxTime2d, 1, MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);

if (!myrank)
printf(" 2dtime for totalProcesses = %d for n= %lld = %lf\n", process_cols* process_rows, n, maxTime2d);

// Now we will calculate for 1-d domain decomposition

process_rows *= process_cols;                          // as it is row-wise decomposition
process_cols = 1;

rows = n / process_rows;
cols = n / process_cols;

double matrix1[rows][cols];
double buffer1[cols+2];

// initializing matrix1
srand((unsigned int)time(NULL) + myrank);
for (long long int i = 0; i < rows; i++)
{
for (long long int j = 0; j < cols; j++)
{
int x = rand();
int y = rand();
if(y==0)y=1;
matrix1[i][j] = (double)x / (double)y;
}
}

count = ((((myrank / process_cols) + 1) * (n / process_rows)) + 1);
count -= (myrank % process_cols) * (n / process_cols);
if (count > 0)
{
if (count >= (n / process_cols))
{
count = n / process_cols;
}
}
else
count = 0;

if ((myrank / process_cols) == process_rows - 1)
count = 0;                                          // count = 0 for last row processes

recv_rank = myrank - (process_cols);
recv_count = ((((recv_rank / process_cols) + 1) * (n / process_rows)) + 1);
recv_count -= (recv_rank % process_cols) * (n / process_cols);
if (recv_count > 0)
{
if (recv_count >= (n / process_cols))
{
recv_count = n / process_cols;
}
```

```
        }
        else
        recv_count = 0;
        // recv_count = 0 for first row processes
        if (recv_rank < 0)
        recv_count = 0;


        sTime = MPI_Wtime();

        for (int i = 0; i < 20; i++)
        {
        if (count)
        MPI_Send(matrix1[rows - 1], count, MPI_DOUBLE, myrank + process_cols, 1, MPI_COMM_WORLD);

        if (recv_count)
        MPI_Recv(buffer1, recv_count, MPI_DOUBLE, recv_rank, 1, MPI_COMM_WORLD, &status);

        // update matrix1
        long long int comp = (myrank / process_cols) * rows - (myrank % process_cols) * cols;
        double temp[rows][cols];
        // update in same subdomain
        for (long long int i = 1; i < rows; i++)
        {
        long long int colsToUpdate = comp + i + 1;
        if (colsToUpdate >= cols)
        colsToUpdate = cols;
        for (long long int j = 0; j < colsToUpdate; j++)
        {
        temp[i][j] = matrix1[i][j] - matrix1[i - 1][j];
        }
        }
        for (long long int i = 1; i < rows; i++)
        {
        long long int colsToUpdate = comp + i + 1;
        if (colsToUpdate >= cols)
        colsToUpdate = cols;
        for (long long int j = 0; j < colsToUpdate; j++)
        {
        matrix1[i][j] = temp[i][j];
        }
        }
        // update using other subdomain
        for (long long int j = 0; j < recv_count; j++)
        {
        matrix1[0][j] -= buffer1[j];
        //  printf("[%d]--(%lld)%lf ",myrank,j,buffer1[j]);
        }
        }

        eTime = MPI_Wtime();

        double mtime1d = eTime - sTime;
        double maxTime1d;
        MPI_Reduce(&mtime1d, &maxTime1d, 1, MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);
        if (!myrank)
        printf(" 1dtime for totalProcess = %d for n= %lld = %lf\n", process_cols *process_rows, n, maxTime1d);

        MPI_Finalize();
        return 0;
}
```