University of Illinois at Chicago

Parallel Processing

ECE 566

A.Y. 2017 - 2018

Assignment 2

Cipolletta Antonio 655452970

Gualco Andrea 651262476

Contents

1 Introduction

The main goal of this assignment is to implement a parallel algorithm to find the determinant of a $n \times n$ matrix A using the LU decomposition. A generic square matrix A can be written as $A = LU$ such that L is a lower triangular matrix with all one on the main diagonal and U is an upper triangular matrix as shown in eq. (1).

$$A = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{bmatrix} = \begin{bmatrix} 1 & 0 & \dots & 0 \\ l_{21} & 1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ l_{n1} & l_{n2} & \dots & 1 \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & \dots & u_{1n} \\ 0 & u_{22} & \dots & u_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & u_{nn} \end{bmatrix} \tag{1}$$

In this way the determinant can be computed in an easy way:

$$det(A) = det(L) \cdot det(U) = 1 \cdot \prod_{i=1}^{n} u_{ii} \tag{2}$$

To accomplish this task using EXTREME cluster, it has been necessary to implement the LU decomposition algorithm in a parallel way. To parallelize the LU decomposition two approaches based on data-decomposition have been used:

1. 1-D decomposition. The p[1] processors are organized in a virtual 1-D topology, a ring. At each processor is assigned a group of $\frac{n}{p}$ rows of the matrix A.

2. 2-D decomposition. The p processors are organized in a virtual 2-D mesh topology. Each processor is in charge of a block of dimension $\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$.

For both the approaches: the matrix is generated at the root processor and than distributed over all the other processors. At the end each processor has a group of $\frac{n^2}{p}$ elements of the L and U matrix, therefore it can compute a partial determinant multiplying the diagonal elements of the U submatrix. The determinant is computed multiplying the partial determinant of each processor.
In this assignment, both the 1-D and the 2-D version have been implemented in order to compare the performances of the two algorithms.
In particular two groups of experiments have been done:

1. Given a fixed number $n^2 >> 1$ for the matrix dimension, the number of processor p has been changed in order to see how much different available resources influence the performance. In particular the idea is to observe when the comunication overhead starts to nullify the gain in terms of computation.

2. Given a fixed value of available processors p, change the dimension of the matrix n^2. This is particularly important to compare the two algorithms because it stresses the computation part of the total work.

[1] $p \leq n$.

2 Solution adopted

2.1 1-D approach

2.1.1 Parallelization

The parallelization of the algorithm exploit the dependencies of the elements of the matrix on other previous elements.

$$A = \begin{bmatrix} u_{00} & u_{01} & u_{02} & u_{03} \\ l_{10}u_{00} & l_{10}u_{01} + u_{11} & l_{10}u_{02} + u_{12} & l_{10}u_{03} + u_{13} \\ l_{20}u_{00} & l_{20}u_{01} + l_{21}u_{11} & l_{20}u_{02} + l_{21}u_{12} + u_{22} & l_{20}u_{03} + l_{21}u_{13} + u_{23} \\ l_{30}u_{00} & l_{30}u_{01} + l_{31}u_{11} & l_{30}u_{02} + l_{31}u_{12} + l_{32}u_{22} & l_{30}u_{03} + l_{31}u_{13} + l_{32}u_{23} + u_{33} \end{bmatrix} \quad (3)$$

As we can see in (3) where there is a 4×4 matrix, with the exception of the first row, the elements of other rows depend on values of the previous ones.
In particular the number of dependencies of an element of the A matrix can defined in this way:

- if a_{ij} is in the diagonal or in the upper part of the A matrix the number of dependencies are equal to n/p
- if a_{ij} is in the lower part of the A matrix the number of dependencies are equal to $j+1$ where j is the column index and $j = 0, 1, ..., (n/p - 1)$

So as soon as the first rows completes the computation of its elements and broadcast the result, the other processors can start to resolve all the possible dependencies of the elements. In this way one dependency is solved at each step and it is avoided that each processor will wait to compute all its elements assigned only when there are available all the previous elements.

2.2 Communication protocol

Assuming that at each processor is assigned an `id`. The communication protocol is the folowing:

- For `id - 1` times it receives a row from the processors with smaller `id` and compute some updates to its elements.
- At the end it send its updated row to the processor with greater `id`.

The total algorithm can be seen as a cyle of p iterations when at the start all p processors are active than p-1,p-2...1 and each time a broadcast over the active processors is performed. At the end all the processors are waken up to compute the final result through a reduction operation.

2.2.1 Example with a 4×4 matrix and $p = n$

Starting from (4) the first row that doesn't need any computation so it is sent with the broadcast function.

$$A = \left[\begin{array}{cccc} u_{00} & u_{01} & u_{02} & u_{03} \\ \hline l_{10}u_{00} & l_{10}u_{01} + u_{11} & l_{10}u_{02} + u_{12} & l_{10}u_{03} + u_{13} \\ \hline l_{20}u_{00} & l_{20}u_{01} + l_{21}u_{11} & l_{20}u_{02} + l_{21}u_{12} + u_{22} & l_{20}u_{03} + l_{21}u_{13} + u_{23} \\ \hline l_{30}u_{00} & l_{30}u_{01} + l_{31}u_{11} & l_{30}u_{02} + l_{31}u_{12} + l_{32}u_{22} & l_{30}u_{03} + l_{31}u_{13} + l_{32}u_{23} + u_{33} \end{array}\right] \tag{4}$$

As soon as the other rows receive the data they will resolve their dependencies and the first rows will compute its elements.
The result before the broadcast of the second row will be:

$$A = \left[\begin{array}{cccc} u_{00} & u_{01} & u_{02} & u_{03} \\ \hline l_{10} & u_{11} & u_{12} & u_{13} \\ \hline l_{20} & l_{21}u_{11} & l_{21}u_{12} + u_{22} & l_{21}u_{13} + u_{23} \\ \hline l_{30} & l_{31}u_{11} & l_{31}u_{12} + l_{32}u_{22} & l_{31}u_{13} + l_{32}u_{23} + u_{33} \end{array}\right] \tag{5}$$

Now the second row con be broadcast and the third row can be completely computed. The partial result before broadcasting the third row is:

$$A = \left[\begin{array}{cccc} u_{00} & u_{01} & u_{02} & u_{03} \\ \hline l_{10} & u_{11} & u_{12} & u_{13} \\ \hline l_{20} & l_{21} & u_{22} & u_{23} \\ \hline l_{30} & l_{31} & l_{32}u_{22} & l_{32}u_{23} + u_{33} \end{array}\right] \tag{6}$$

The the third row is broadcast and finally the last row can be computed to obtain the final result

$$A = \left[\begin{array}{cccc} u_{00} & u_{01} & u_{02} & u_{03} \\ \hline l_{10} & u_{11} & u_{12} & u_{13} \\ \hline l_{20} & l_{21} & u_{22} & u_{23} \\ \hline l_{30} & l_{31} & l_{32} & u_{33} \end{array}\right] \tag{7}$$

2.2.2 Example with a 4×4 matrix and $p < n$

In order to understand better the algorithm it is presented an example with a 4×4 matrix. Starting from (1) and multiplying the two L and U matrices, the following result is obtained.

$$\begin{bmatrix} l_{00}u_{00} & l_{00}u_{01} & l_{00}u_{02} & l_{00}u_{03} \\ l_{10}u_{00} & l_{10}u_{01} + l_{11}u_{11} & l_{10}u_{02} + l_{11}u_{12} & l_{10}u_{03} + l_{11}u_{13} \\ l_{20}u_{00} & l_{20}u_{01} + l_{21}u_{11} & l_{20}u_{02} + l_{21}u_{12} + l_{22}u_{22} & l_{20}u_{03} + l_{21}u_{13} + l_{22}u_{23} \\ l_{30}u_{00} & l_{30}u_{01} + l_{31}u_{11} & l_{30}u_{02} + l_{31}u_{12} + l_{32}u_{22} & l_{30}u_{03} + l_{31}u_{13} + l_{32}u_{23} + l_{33}u_{33} \end{bmatrix} \tag{8}$$

Supposing to have only two processors, we need to assign the first two rows on processor 0 and the third and fourth to processor 1 (see (9)).

$$\left[\begin{array}{cccc} l_{00}u_{00} & l_{00}u_{01} & l_{00}u_{02} & l_{00}u_{03} \\ l_{10}u_{00} & l_{10}u_{01} + l_{11}u_{11} & l_{10}u_{02} + l_{11}u_{12} & l_{10}u_{03} + l_{11}u_{13} \\ \hline l_{20}u_{00} & l_{20}u_{01} + l_{21}u_{11} & l_{20}u_{02} + l_{21}u_{12} + l_{22}u_{22} & l_{20}u_{03} + l_{21}u_{13} + l_{22}u_{23} \\ l_{30}u_{00} & l_{30}u_{01} + l_{31}u_{11} & l_{30}u_{02} + l_{31}u_{12} + l_{32}u_{22} & l_{30}u_{03} + l_{31}u_{13} + l_{32}u_{23} + l_{33}u_{33} \end{array}\right] \tag{9}$$

Initially the processor 0 starts to compute its element of the matrix A (see (10)).

$$A_{P0} = \begin{bmatrix} l_{00}u_{00} & l_{00}u_{01} & l_{00}u_{02} & l_{00}u_{03} \\ l_{10}u_{00} & l_{10}u_{01} + l_{11}u_{11} & l_{10}u_{02} + l_{11}u_{12} & l_{10}u_{03} + l_{11}u_{13} \end{bmatrix} \tag{10}$$

Being the first processor its elements doesn't depends on previous block and it doesn't need to receive data to compute them. So the `Compute_Intern` procedure is run on the block. Since the values l_{00}, l_{11}, l_{22} and l_{33} are all equal to 1 the first line will be computed as

$$\begin{aligned} u_{00} &= A[0][0] = A[0][0] \\ u_{01} &= A[0][1] = A[0][1] \\ u_{02} &= A[0][2] = A[0][2] \\ u_{03} &= A[0][3] = A[0][3] \end{aligned} \tag{11}$$

while in the second one, the values are computed in this way:

$$\begin{aligned} l_{10} &= A[1][0] = A[0][1]/A[0][0]; \\ u_{11} &= A[1][1] = A[1][1] - A[1][0] * A[0][1] \\ u_{12} &= A[1][2] = A[1][2] - A[1][0] * A[0][2] \\ u_{13} &= A[1][3] = A[1][3] - A[1][0] * A[0][3] \end{aligned} \tag{12}$$

The block element computations is now done and the final result is

$$A_{P0} = \begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \end{bmatrix} = \begin{bmatrix} u_{00} & u_{01} & u_{02} & u_{03} \\ l_{10} & u_{11} & u_{12} & u_{13} \end{bmatrix} \tag{13}$$

At this point the A_{P0} matrix is broadcast to the others blocks. In order to do the broadcast the matrix has to be flattened to a vector and so the other processors, after receiving it, have to de-flatten it.
After this operation in processor number 1 there will be this configuration

$$\left[\begin{array}{c} B_{P1} \\ \hline A_{P1} \end{array}\right] = \left[\begin{array}{cccc} u_{00} & u_{01} & u_{02} & u_{03} \\ l_{10} & u_{11} & u_{12} & u_{13} \\ \hline l_{20}u_{00} & l_{20}u_{01} + l_{21}u_{11} & l_{20}u_{02} + l_{21}u_{12} + u_{22} & l_{20}u_{03} + l_{21}u_{13} + u_{23} \\ l_{30}u_{00} & l_{30}u_{01} + l_{31}u_{11} & l_{30}u_{02} + l_{31}u_{12} + l_{32}u_{22} & l_{30}u_{03} + l_{31}u_{13} + l_{32}u_{23} + u_{33} \end{array}\right] \tag{14}$$

Now processor 1 can start to compute the elements of A_{P1} that depend on the values of the received B_{P1} through `Compute_Extern` procedure. This procedure start to calculate all the possible elements that depends on the previous block. If the element depends only on previous values final value is immediately computed, otherwise it is computed a partial result

by subtracting only the values that depends on the other block.

$$\begin{aligned}
l_{20} &= A[0][0] = A[0][0]/B[0][0] \\
l_{21} &= A[0][1] = (A[0][1] - A[0][0] * B[0][1])/B[1][1] \\
l_{30} &= A[1][0] = A[1][0]/B[0][0] \\
l_{31} &= A[1][1] = (A[1][1] - A[1][0] * B[0][1])/B[1][1] \\
l_{32,partial} &= A[1][2] = A[1][2] - A[1][0] * B[0][2] - A[1][1] * B[1][2] \\
u_{22} &= A[0][2] = A[0][2] - A[0][0] * B[0][2] - A[0][1] * B[1][2] \\
u_{23} &= A[0][3] = A[0][3] - A[0][0] * B[0][3] - A[0][1] * B[1][3] \\
u_{33,partial} &= A[1][3] = A[1][3] - A[1][0] * B[0][3] - A[1][1] * B[1][3]
\end{aligned} \tag{15}$$

After the `Compute_Extern` shown in (15) the (18) will become

$$\left[\begin{array}{c} B_{P1} \\ \hline A_{P1} \end{array}\right] = \left[\begin{array}{cccc} b_{00} & b_{01} & b_{02} & b_{03} \\ b_{10} & b_{11} & b_{12} & b_{13} \\ \hline a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \end{array}\right] = \left[\begin{array}{cccc} u_{00} & u_{01} & u_{02} & u_{03} \\ l_{10} & u_{11} & u_{12} & u_{13} \\ \hline l_{20} & l_{21} & u_{22} & u_{23} \\ l_{30} & l_{31} & l_{32}u_{22} & l_{32}u_{23} + u_{33} \end{array}\right] \tag{16}$$

After that the dependency on the elements of the previous block is solved, the `Compute_Intern` is run on the block and at the end the final value are retrieved

$$\begin{aligned}
l_{32} &= A[1][2] = A[1][2]/A[0][2] \\
u_{33} &= A[1][3] = A[1][3] - A[1][2] * A[0][3]
\end{aligned} \tag{17}$$

In this way the (18) will become

$$\left[\begin{array}{c} B_{P1} \\ \hline A_{P1} \end{array}\right] = \left[\begin{array}{cccc} b_{00} & b_{01} & b_{02} & b_{03} \\ b_{10} & b_{11} & b_{12} & b_{13} \\ \hline a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \end{array}\right] = \left[\begin{array}{cccc} u_{00} & u_{01} & u_{02} & u_{03} \\ l_{10} & u_{11} & u_{12} & u_{13} \\ \hline l_{20} & l_{21} & u_{22} & u_{23} \\ l_{30} & l_{31} & l_{32} & u_{33} \end{array}\right] \tag{18}$$

Being P1 the last processor, there is no need to do a broadcast of the block. So the algorithm terminates.
The elements obtained are stored in the A sub-matrix of each processor, so the final result will be

$$\left[\begin{array}{c} A_{P0} \\ \hline A_{P1} \end{array}\right] = \left[\begin{array}{cccc} a_{00,p0} & a_{01,p0} & a_{02,p0} & a_{03,p0} \\ a_{10,p0} & a_{11,p0} & a_{12,p0} & a_{13,p0} \\ \hline a_{00,p1} & a_{01,p1} & a_{02,p1} & a_{03,p1} \\ a_{10,p1} & a_{11,p1} & a_{12,p1} & a_{13,p1} \end{array}\right] = \left[\begin{array}{cccc} u_{00} & u_{01} & u_{02} & u_{03} \\ l_{20} & u_{11} & u_{12} & u_{13} \\ \hline l_{20} & l_{21} & u_{22} & u_{23} \\ l_{30} & l_{31} & l_{32} & u_{33} \end{array}\right] \tag{19}$$

At the point, using a all-to-one reduction operation on the elements of each sub-matrix A, the determinant is computed.

2.3 2-D approach

The same idea applied to the 1-D approach has been extended to the 2-D approach, with the only difference that in this case instead of broadcasting a row, it will be broadcasted a

block.
The communication scheme is a little bit more complex than the case of 1-D partitioning but it is still very regular. Below are reported all the steps where the white processors are the active in the step, the black lines indicate first communication, blue lines indicate second communication and at the end the red ones. Again the parallelization is done exploiting the dependencies of the elements of the previous ones.
There is only the constraint to use a number of p such that the $\sqrt{p}$ will be an integer value and also n is a multiple of $\sqrt{p}$. This is just a simplification in order to make the code lighter but the approach is exactly the same also in the general case.

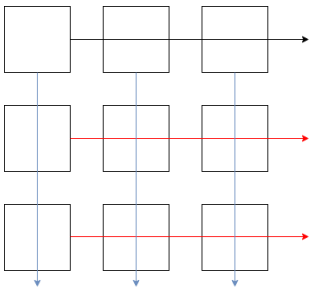

Figure 1: 2-D partitioning. Example p = 9. Step 1.

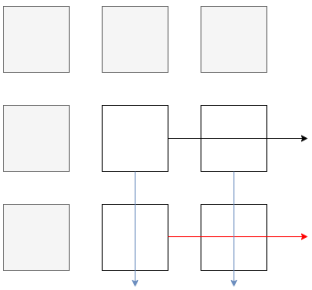

Figure 2: 2-D partitioning. Example p = 9. Step 2.

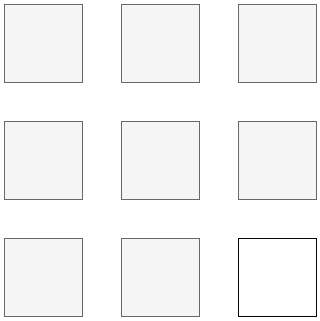

Figure 3: 2-D partitioning. Example p = 9. Step 3.

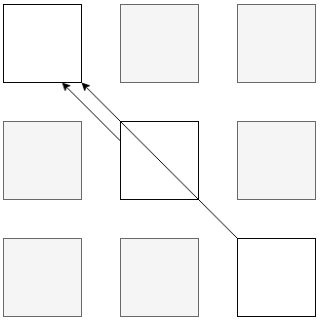

Figure 4: 2-D partitioning. Example p = 9. Final step.

2.4 Complexity

2.4.1 1D partitioning

In 1D partitioning case, at each step it is done an operation which complexity is $O\left(\frac{n^3}{p^2}\right)$ (dominated by the `Compute_Extern`). Since there are p steps the final complexity is:

$$p \sum_{k=0}^{n/p-1} \frac{n}{p} \cdot (n-k-1) = \frac{n^3}{p^2} \cdot p = \frac{n^3}{p} \tag{20}$$

2.4.2 2D partitioning

In 2D partitioning case, at each step it is done an operation which complexity is $\mathrm{O}\left(\frac{n^3}{p^{3/2}}\right)$ (dominate by the `compute_up_left`). There are $\sqrt{p}$ steps so the final complexity is

$$\frac{n^3}{p^{3/2}} \cdot \sqrt{p} = \frac{n^3}{p} \tag{21}$$

3 Formulation

The determinant of the matrix A of dimension $n \times n$ has been implemented in the Extreme cluster using a 1-D approach. For this reason a ring topology has been used.
First,the MPI library has been initialized using the function `MPI_Init`. After that, the topology has been declared using the function `MPI_Cart_create`.
Other functions used are:

- `MPI_Comm_size` in order to retrieve the number of processor used (p);
- `MPI_Comm_rank` in order to save the rank of each processor in the topology;
- `MPI_Cart_coords` in order to save the coordinates of each processor.

Before the `return`, the function `MPI_Finalize` in order to close the MPI environment.
In order to implement in a parallel way the LU decomposition, other functions have been used. In particular in the first approach, `MPI_Bcast` is used to send the block computed to the other processors.
In order to implement the first approach of the LU decomposition, other functions have been used. In particular in the first approach, `MPI_Bcast` is used to send the block computed to the other processors.
The second approach (the 2D one) uses instead the `MPI_Send` in order to send the block to a specific target and the `MPI_Recv` to receive a specific block from the correct source.
In both the approaches `MPI_Reduce` is used to compute the value of the determinant on the root processor.

4 Parameter ranges

The input parameters of the programs are:

- n that is the dimension of the square matrix A
- p that is the number of physical processor used for the topology;

First, after having fixed the number of physical processors p, different values n of the matrix dimension has been used to test the communication and computation overhead. Then it has been decided the different number of processor used, given a fixed $n >> 1$. In 1 there are the n and p used in the experiments.

Table 1: Different n and p for experiments on the two approaches

(a) Values of p and n for testing the 1-D approach

p	n
2	60
3	80
4	106
6	141
9	188
13	250
19	333
42	
63	

(b) Values of p and n for testing the 2-D approach

p	n
4	42
9	84
16	126
36	168
49	210
64	255
	294
	336

5 Timing measurements

In order to have a quantitative measurement of the performance for each simulation a time measurement has been done. Since the cluster is a shared machine and due to the high sources of variability the following approach has been adopted.
In each source code the main part has been identified, i.e. the LU decomposition. This part has been executed 20 times with a time measurement for each iteration.
At the end the average time and the deviation $\sqrt{\frac{\sum_{i=1}^{20}(time[i]averagetime)^2}{19}}$ has been computed and used to evaluate the simulation.

6 Results and analysis

NOTE: There are some points outside the trend curve in all the graphs. Since the CLUSTER is a shared machine is not easy to perform good measurement of time even if an average between multiple running is performed, for this reason the *outsiders* are not considered for the comments.

6.1 Execution time vs data size

As explained in the introduction, the first experiment has been done fixing the number of available processors and changing the data size in particular the dimension n of the initial square matrix. Fig.5 shows the result of the experiment *constant resources* performed on the 1-D partitioning. It is possible to observe that for all the value of `p` the trend is more or less equal: initially the execution time is more or less constant because for low values of n the major contribution is given by the communication. From a certain point the curve starts growing because increasing the data size the computation complexity starts to be the main factor, and the cause could not be the increasing of the message size since the growth is more than linear. An important point that emerges from the graph is that the value of n that divides the moments is clearly dependent on the number of processors: greater the number of processors, greater the speed-up in the computation part.

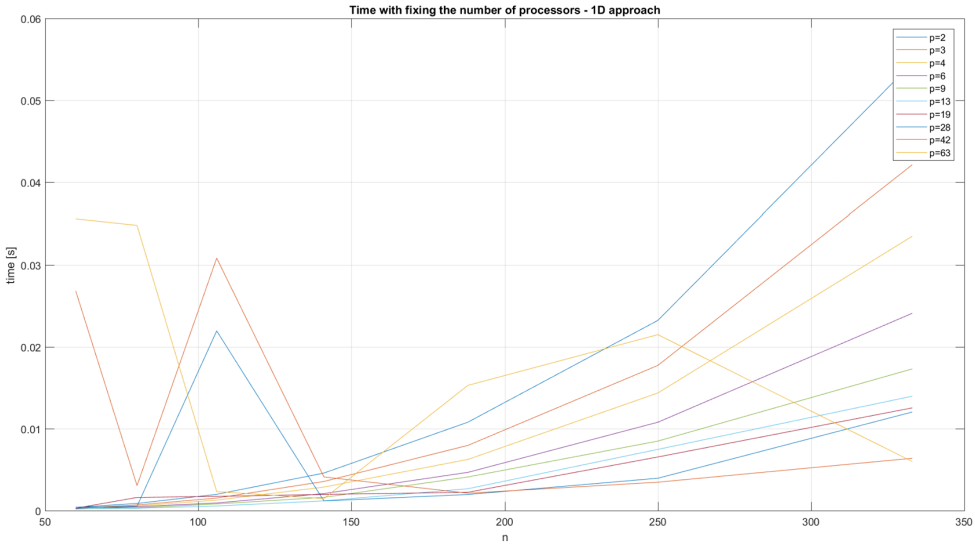

Figure 5: 1-D partitioning. Execution time as function of data size.

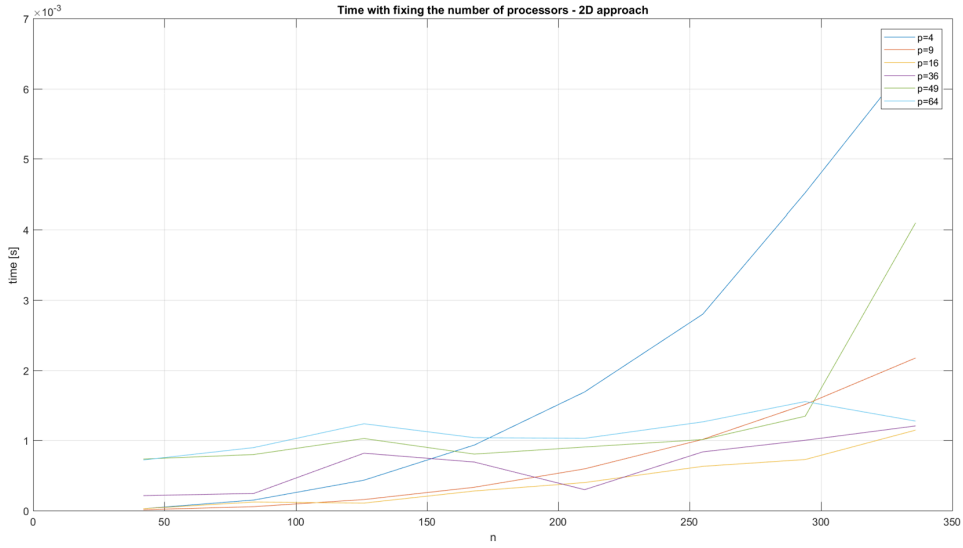

Figure 6: 2-D partitioning. Execution time as function of data size.

Fig.6 shows the result in case of the 2-D partitioning. The number of processors and the values for the data size used in both case are comparable. Similar consideration as in the previous case. What can be seen is that the curve trend is very similar but the time axis is scaled of a factor 10. From the experiment made by us the 2-D partitioning seems to work much better than 1-D. This is reasonable due to the fact that the greater work is done by the processors in charge of the lower part of the initial matrix. In case of the 2-D partitioning the

lower part is spread across multiple processors and therefore there is a better parallelization of the work compared to the 1-D partitioning where all the last part of the matrix is assigned to one processor.

6.2 Execution time vs available resources

In this case the value of n is fixed and the number of processors changes. As in the previous case similar considerations can be done for the 1-D partitioning and the 2-D partitioning. Still the time axis is scaled proving that the 2-D partitioning is a better solution. In general for small values of n the execution time depends only on the communication overhead and so the curve is increasing for the number of processors. For higher values of n instead the shape of the curve is more or less convex since increasing the available resource has a good impact on the computation complexity until the communication overhead starts to be significant and so the execution time starts increasing again. It can be seen that for large n the gain of increasing the number of processors cause a decrease of the execution time more than linear instead when the minimum is passed the increasing is pretty linear.

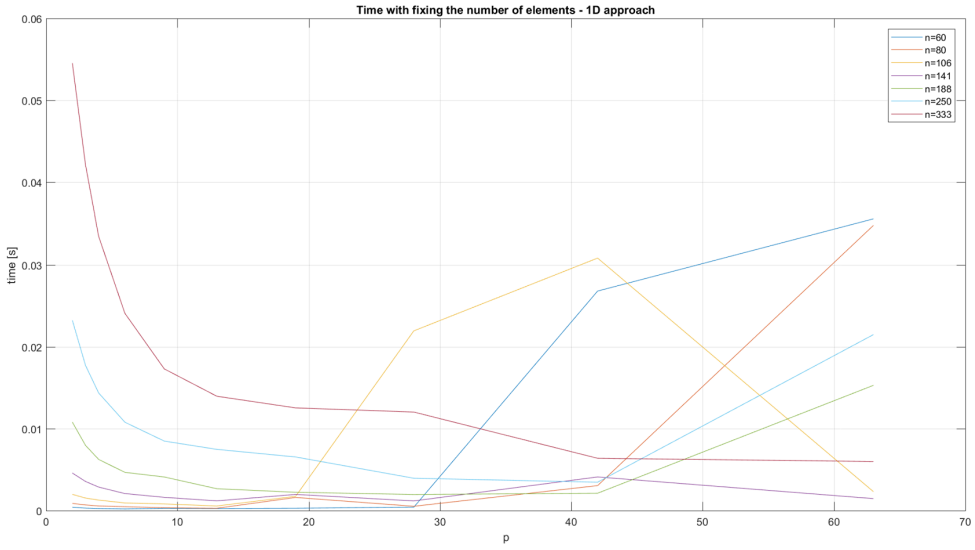

Figure 7: 1-D partitioning. Execution time as function of #processors.

6.3 Timig results

In this subsection there are the tabular results of timing and deviations of the two approaches with the values of n and p of the 1.
The results are present in table 2, 3, 4 and 5.

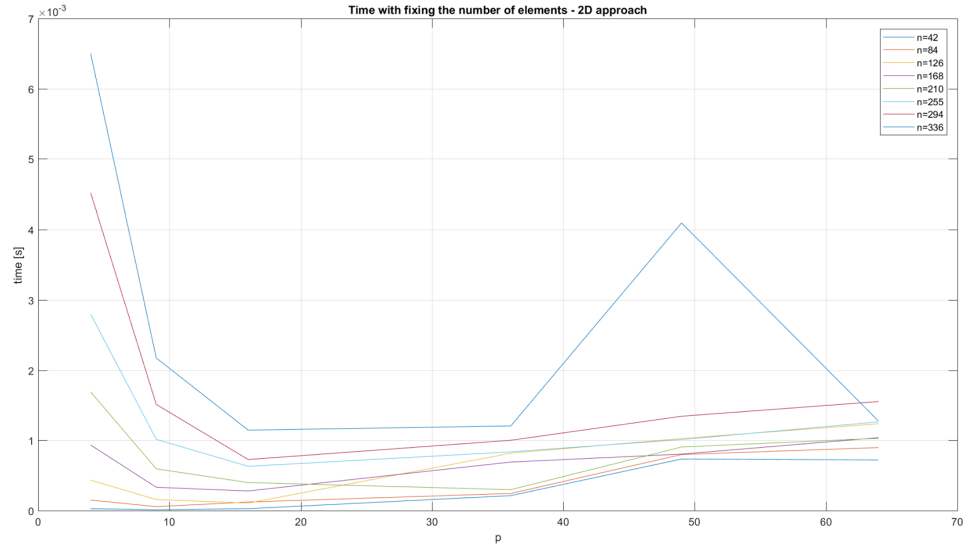

Figure 8: 2-D partitioning. Execution time as function of #processors.

Table 2: Timing results 1D approach

		p									
		2	**3**	**4**	**6**	**9**	**13**	**19**	**28**	**42**	**63**
n	**60**	0.0004	0.0003	0.0003	0.0002	0.0003	0.0003	0.0003	0.0004	0.0268	0.0356
	80	0.0009	0.0007	0.0006	0.0005	0.0004	0.0003	0.0016	0.0005	0.0031	0.0348
	106	0.0020	0.0015	0.0013	0.0009	0.0008	0.0006	0.0017	0.0219	0.0308	0.0023
	141	0.0046	0.0036	0.0029	0.0021	0.0016	0.0012	0.0020	0.0012	0.0041	0.0015
	188	0.0108	0.0080	0.0063	0.0047	0.0041	0.0027	0.0022	0.0020	0.0021	0.0153
	250	0.0232	0.0177	0.0144	0.0108	0.0085	0.0075	0.0066	0.0040	0.0035	0.0215
	333	0.0545	0.0421	0.0335	0.0241	0.0173	0.0140	0.0125	0.0120	0.0064	0.0060

7 Lessons

From this assignment, we learned how serial algorithms can be parallelized in order to achieve better results.
We also get used to handle 2-D matrices and how to use indexes to complete the task.
Moreover we used other functions of MPI library that we did not used during the last assignment.
Last but not least, we had the chance to focus on an algorithm that during the lesson has only been presented in its serial version, so we had to understand and think a way to parallelize it. This means thinking a lot about the implementation and this carry a deep understanding of the algorithm

Table 3: Deviation results 1D approach

		p									
		2	**3**	**4**	**6**	**9**	**13**	**19**	**28**	**42**	**63**
n	**60**	0	0	0	0	0	0	0	0.0000	0.0034	0.0094
	80	0	0	0	0	0	0	0.0018	0.0000	0.0020	0.0047
	106	0	0	0	0	0	0	0.0001	0.0009	0.0073	0.0015
	141	0	0	0	0	0	0	0.0012	0.0000	0.0033	0.0001
	188	0.0001	0	0	0	0	0	0.0000	0.0000	0.0001	0.0069
	250	0	0	0	0	0	0	0	0.0002	0.0001	0.0001
	333	0.0002	0.0004	0.0003	0.0001	0.0001	0.0001	0.0011	0.0001	0.0002	0.0001

Table 4: Timing results 2D approach

		p					
		4	**9**	**16**	**36**	**49**	**64**
n	**42**	2.8e-05	1.3e-05	2.8e-05	0.0002	0.0007	0.0007
	84	15.1e-5	5.7e-05	0.0001	0.0002	0.0008	0.0009
	126	0.0004	0.0002	0.0001	0.0008	0.0010	0.0012
	168	0.0009	0.0003	0.0003	0.0007	0.0008	0.0010
	210	0.0017	0.0006	0.0004	0.0003	0.0009	0.0010
	255	0.0028	0.0010	0.0006	0.0008	0.0010	0.0013
	294	0.0045	0.0015	0.0007	0.0010	0.0013	0.0016
	336	0.0065	0.0022	0.0011	0.0012	0.0041	0.0013

Table 5: Deviation results 2D approach

		p					
		4	**9**	**16**	**36**	**49**	**64**
n	**42**	0	0	0	4.4744e-05	0.0003	0.0003
	84	0	0	3.164e-05	4.4744e-05	0.0004	0.0004
	126	0	0	0	0.0003	0.0005	0.0006
	168	0	0	3.164e-05	0.0003	0.0003	0.0004
	210	0	0	0	4.4744-05	0.0003	0.0004
	255	0	0	3.164e-05	0.0003	0.0004	0.0005
	294	0	0	0	0.0003	0.0005	0.0006
	336	4.474e-05	0	3.164e-05	0.0004	0.0009	0.0004