University of Illinois at Chicago

Parallel Processing

ECE 566

A.Y. 2017 - 2018

Assignment 3

CIPOLLETTA Antonio 655452970
GUALCO Andrea 651262476

Contents

1 Introduction

The main goal of this assignment is to implement a parallel algorithm to compute the operation $|X|^k$. In order to do this two main operations are needed:

- Compute k-1 matrix multiplication[1].
- Compute the determinant of the matrix X^k.

For the *matrix multiplication* multiple parallel implementations are possible. It has been chosen to explore two ways based on two different data decompositions:

1. Cannon's algorithm, i.e. 2D data decomposition: a mesh of $\sqrt{p} \times \sqrt{p}$ processor is used. The algorithm is implemented for the general case ($p \leq n$), so the each processor on the mesh receives a block of $\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$ elements of the initial matrix.
2. DNS algorithm, i.e. 3D data decomposition: a 3-D mesh of $\sqrt[3]{p} \times \sqrt[3]{p} \times \sqrt[3]{p}$ that can be seen as a set of $\sqrt[3]{p}$ 2D mesh. Each processor receives a block of dimension $\frac{n}{\sqrt[3]{p}} \times \frac{n}{\sqrt[3]{p}}$.

The determinant computation has been done in serial at the end only for check purposes.
As usual it has been decided to do the following to experiments:

- fix the size of the matrix n and change the number of processors p;
- fix the number of processors p and change the size n of the matrix;

Moreover the exponent k has been changed despite not being useful to evaluate the relation between n, p and the run time.

2 Solution adopted

2.1 Cannon's algorithm

The cannon's algorithm is modification of the normal 2D matrix multiplication algorithm. The idea of this algorithm is to use shift in order to exchange the blocks in order to reduce the amount of memory used.
The steps in order to implement it are showed in the algorithm 1 The algorithm needs to have only 4 sub-matrices for each processor. Using the shift the coordinates in the mesh are changed, the using the send receive and replace the sub-matrices are exchanged between the processor without the need to use buffers. This will lead to a great memory saving.
The time requested for a single shift is $t_s + t_w \frac{n^2}{p}$. Then the total time to make the multiplication is $\frac{n^3}{p}$.
In the end $T_p = \frac{n^3}{p} + 2t_s\sqrt{p} + 2t_w\frac{n^2}{\sqrt{p}}$.
The memory requirement for each processor is O(n^2/p) because the matrix allocated have

[1] Actually is possible to reduce the number of multiplication to $O(log(k))$ but at the cost of increasing the memory space required. It has been chosen to use the k-1 multiplication as trade-off between memory requirements and computational cost.

```
Data: A
Result: A^k
initialize B, D, T;
scatter the matrix A on the mesh;
sub-matrix received in B by each processor;
sub-matrix B copied in D;
for q from 0 to k-2 do
    reset T;
    shift left the rows of sub-matrix D by i where i is the row index;
    reorder the blocks on the D;
    shift up the columns of sub-matrix B by j where j is the column index;
    reorder the blocks on the B;
    for j from 0 to $\sqrt{p} - 1$ do
        matrix multiplication between D and B;
        shift left all the rows of sub-matrix D by 1;
        reorder the blocks on the D;
        shift up all the columns of sub-matrix B by 1;
        reorder the blocks on the B;
    end
    Copy original sub-matrix received with scatter in B;
    save the sub-matrix T in D;
end
gather the result in the source;
```

Algorithm 1: Cannon's algorithm

size $n/\sqrt{p} \times n/\sqrt{p}$. Only in the root node there is the allocation of the source matrix and the gather of the result matrix that have size $n \times n$.
Isoefficiency: $p^{3/2}$
Cost optimality: n^2

2.2 DNS algorithm

The DNS algorithm is based on the following observations: the naive $O(n^3)$ implementation of the matrix multiplication is a concatenation of 3 `for cycle` as showned in 2. The classic implementation can be rearranged as swoned in 3, i.e. now for each value of k there are n^2 multiplication. In total there are n^3 multiplication or $(\sqrt[3]{p})^3$ block multiplication. Each block multiplication is assigned to a processor in the 3D mesh topology. In particular the 3D mesh can be seen as a set of $\sqrt[3]{p}$ 2D mesh where each 2D mesh is in charge of computing part of C_{ij}. At the end summing the partial result computed by each 2D mesh will give the complete result.

```
Data: A,B
Result: $C = A * B$
for i from 0 to n-1 do
    for j from 0 to n-1 do
        C[i][j] = 0;
        for k from 0 to n-1 do
            C[i][j] += A[i][k]*B[k][j];
        end
    end
end
```

Algorithm 2: Naive ijk algorithm

```
Data: A,B
Result: $C = A * B$
for k from 0 to n-1 do
    for i from 0 to n-1 do
        for j from 0 to n-1 do
            C[i][j] += A[i][k]*B[k][j];
        end
    end
end
```

Algorithm 3: Naive ijk algorithm with swapped cycle

2.2.1 Communication pattern

The communication pattern of the DNS algorithm is composed of 4 or 3 phases depending on the distribution of the operand matrices:

- The first phase, needed in case the operands are stored inside only one processor, the root processor, consists of scattering the different block of the matrices A and B to the 2D submesh with coordinate k equal to 0.
  Communication cost $= 2 * t_s * (\sqrt[3]{p} - 1) + t_w * (\frac{n}{p^{1/3}})^2 * (p^{3/2} - 1)$.
- In the second phase the coloumns of the matrix A are distributed on the different planes:
  coloumn 1 is sent to the 2D submesh with cordinate k equal to 1,
  coloumn 2 is sent to the 2D submesh with cordinate k equal to 2, an so on.
  The rows of the matrix B are distributed similarly to the coloumns of the A matrix:
  row 1 is sent to the 2D submesh with cordinate k equal to 1,
  row 2 is sent to the 2D submesh with cordinate k equal to 2, an so on.
  Communication cost: $t_s + t_w * (\frac{n}{p^{1/3}})^2$
- In the third phase on each 2D submesh the row of B is broadcasted among the coloumns of the mesh, instead the coloumn of A is broadcasted among the rows of the mesh.
  Communication cost: $t_s * log(p^{1/3}) + t_w * (\frac{n}{p^{1/3}})^2$

- The fourth phase is started only after the processor ends the computation of a matrix multiplication on its own blocks. It is a reduce operation on the dimension k. In this way the result matrix can be accumulated at the 2D mesh of coordinate k equal to 0 as the original block distribution of matrices A and B after the first phase.
  Communication cost: $t_s * log(p^{1/3}) + t_w * (\frac{n}{p^{1/3}})^2$

The computation on a single block has complexity of $\Theta(\frac{n^3}{p})$.
In the end $T_p = \frac{n^3}{p} + t_s log(p) + t_w \frac{n^2}{p^{2/3}} log(p)$.
The memory requirement is $O(\frac{n^2}{p^{2/3}})$. Only in the root node there is the allocation of the source matrix and the gather of the result matrix that have size $n \times n$.
Isoefficiency: $plog^3(p)$
Cost optimality: $(\frac{n}{log(n)})^3$
An example of the second and the third communication phases is shown for a $2 \times 2 \times 2$ mesh in fig. 1 and fig. 2.

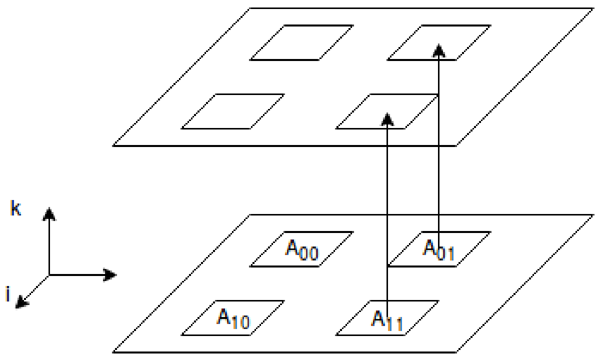

Figure 1: Phase 2 DNS communication in case of a $2 \times 2 \times 2$ mesh.

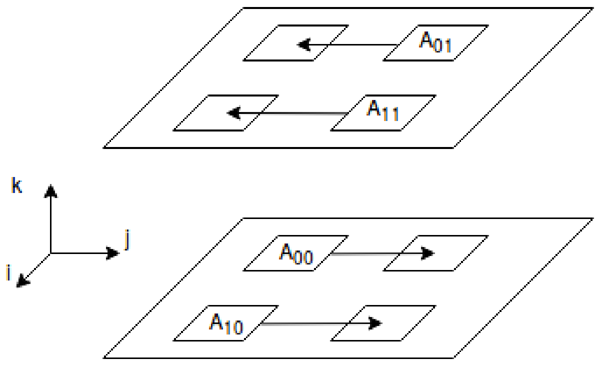

Figure 2: Phase 3 DNS communication in case of a $2 \times 2 \times 2$ mesh.

3 Formulation

The matrix power A of dimension $n \times n$ has been implemented in the Extreme cluster using a 2-D approach of the Cannon algorithm and the 3D approach of the DNS algorithm. For this reason a mesh 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 the communication pattern of the Cannon's algorithm, other functions have been used. In particular, `MPI_Cart_Shift` is used to find the rank of the processors for a shift of a number of units on a dimension of the grid. After the rank is known it is possible to use the `MPI_Sendrecv_replace` to exchange data between the processors using only one buffer. This MPI function is used specifically to efficiently implement shifts that what is needed in Cannon's communication pattern. Other function called are the `MPI_Scatterv` that is used to deliver the blocks of the matrix A to all processors. At the end the `MPI_Gather` is used to receive all the blocks of the result matrix at the root processor.
The second approach (the 3D one) uses instead:

- `MPI_Scatter` and `MPI_Gather` to distribute the blocks of the matrix on a plane and to collect the blocks to build the complete result matrix at the root processor.
- `MPI_Send` and `MPI_Recv` for the one-to-one communication to distribute rows and coloumns to the right k-plane.
- `MPI_Broadcast` and `MPI_Reduce` for the one-to-all and all-to-one communication on subgrid of the 3D virtual topology. In order to construct virtual sub-topology starting from the main 3D topology and to translate the rank from one virtual topology to the others it has been used `MPI_Cart_sub` and `MPI_Group_translate_ranks` library functions.

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;

- k that is the exponent.

Many experiments can be done in this case. All the combination of the n, p and k has been investigated.
The ranges of the values for the Cannon's algorithm experiments are:

- $n = 240, 480, 960, 1920$
- $p = 4, 9, 16, 25, 36, 64, 100$
- $k = 2, 3, 4$

The choice has been influenced by the fact that the number of processors has to have an integer square root. Moreover the $n/\sqrt{p}$ has to be an integer value.
The ranges of the values for the DNS algorithm experiments are:

- $n = 240, 480, 960, 1920$
- $p = 8, 27, 64, 125$
- $k = 2, 3, 4$

The choice has been influenced by the fact that the number of processors has to have an integer cube root. Moreover the $n/p^{1/3}$ has to be an integer value.
So in order to compare the two algorithm it is necessary to choose the similar values for all n and p.

5 Input methods

The program take as input from command line the following elements:

- the size n of the matrix;
- the exponent k;
- a value to choose between input method 1 and input method 2;
  - input method 1 if value equal to 0;
    then specify the probability to have 1 and -1 (the probability is a percentage probability, so it has to be an integer);
  - input method 2 if value not equal to 0;
    specify other four values that is the basic sequence to build the matrix

6 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. matrix multiplication. This part has been executed 20 times with a time measurement for each iteration.
At the end the average time and the deviation $\sqrt{\frac{\sum_{i=1}^{20}(time[i]-average\ time)^2}{19}}$ has been computed and used to evaluate the simulation.

7 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.
NOTE: There are some missing points in the graphs of the DNS. During the submission of the job using few processor the program encountered a problem of segmentation fault using a big matrix (ie. 1920). This is probably due to the impossibility to allocate such a huge sub-matrices and therefore the program terminates it execution. However the code doesn't work only in that specific cases, in fact using a reasonable number of processors for the topogy the program executes normally.

7.1 Analysis

In this case, there was one variable more to explore. All the possible combinations has been explored. As in the previous assignments, the same two experiments has been conducted.

7.1.1 Fixed n of the matrix - Cannon

Fixing the number of elements of the array and changing the number of processors, it is possible to see that the behavior is always the same and doesn't depend on the exponent k. The time is related to the exponent in this way: a higher means a high execution time. The graphs in the figures 3, 4 and 5.
It is possible to notice that increasing the number of processor the execution time is decreased (big matrices continue to have higher execution time that smaller matrices). This is due to the fact that with few processors the block in which the matrix is divided are huge and therefore the communications exchange many data and therefore the code is slowed down. Moreover a bigger matrix need of more communications and this is another factor for which the time is higher.

7.1.2 Fixed p of the topology - Cannon

The same considerations are applied in case of fixing the number of processor of the topology. Again increasing the size of the matrix leads to a general increase of the run time due to much

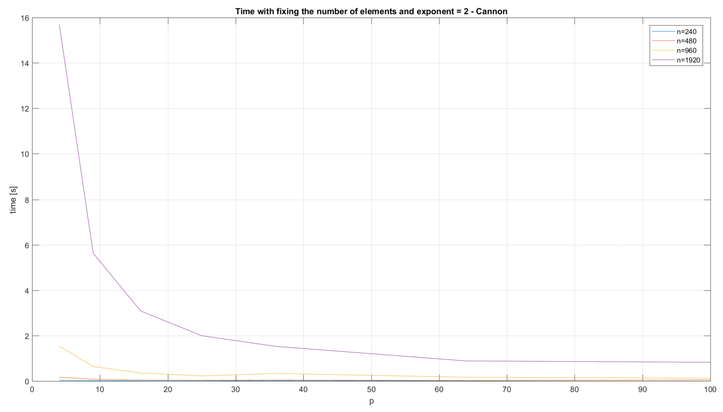

Figure 3: Fixed element n and $k = 2$ - Cannon

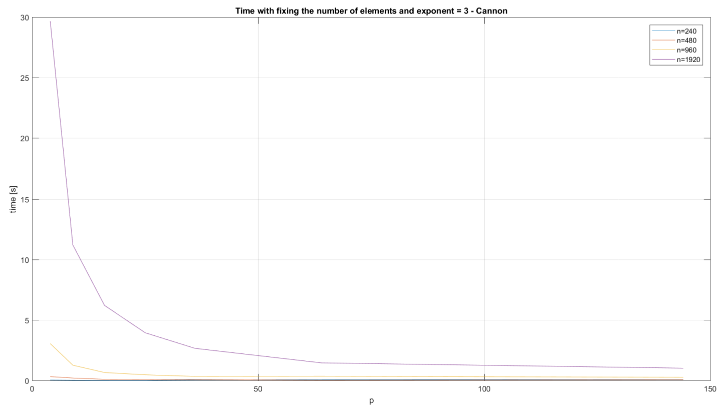

Figure 4: Fixed element n and $k = 3$ - Cannon

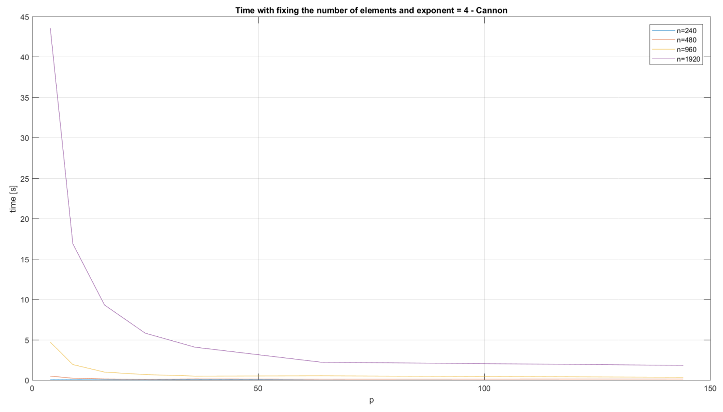

Figure 5: Fixed element n and $k = 4$ - Cannon

more communications to perform. In particular using less processor inside the topology the slope is higher to the respect in the cases where many processors are used (ie 4 vs 100, 4 has

an higher slope that the 100).
The relative graphs are showed in figures 6, 7 and 8.

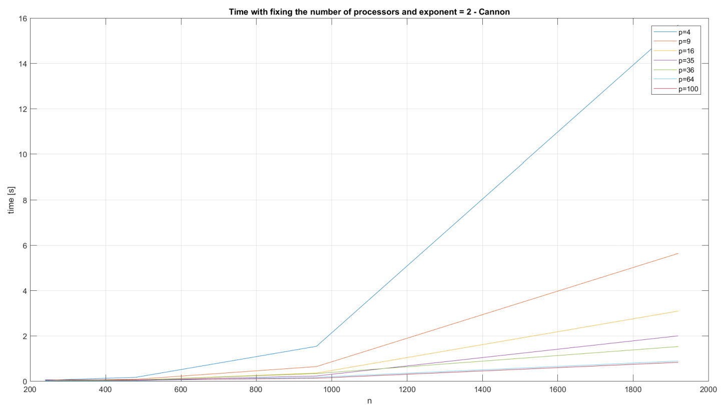

Figure 6: Fixed element p and $k = 2$ - Cannon

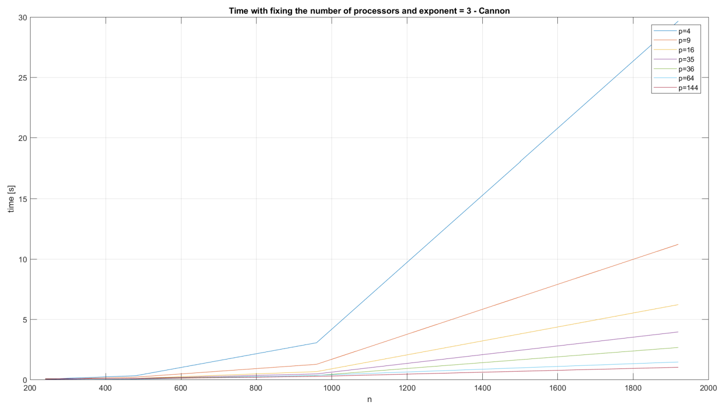

Figure 7: Fixed element p and $k = 3$ - Cannon

7.1.3 Fixed n of the matrix - DNS

Fixing the number of elements of the array and changing the number of processors, it is possible to see that the behavior is always the same and doesn't depend on the exponent k. The only parameter influenced by the exponent k is the run time that increases with a higher k regardless the matrix size and the number of processors used.
As in the Cannon's algorithm, the trend when the size of the matrix is fixed is that the run time is decreased by incrementing the number of processor used. The reason are again that in case of few processors there are communications containing huge amount of data that slow down the communication.
The related graphs are shown in figures 9, 10 and 11.

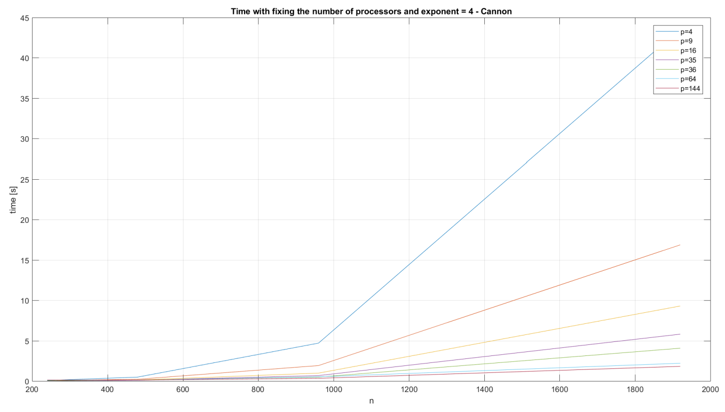

Figure 8: Fixed element p and $k = 4$ - Cannon

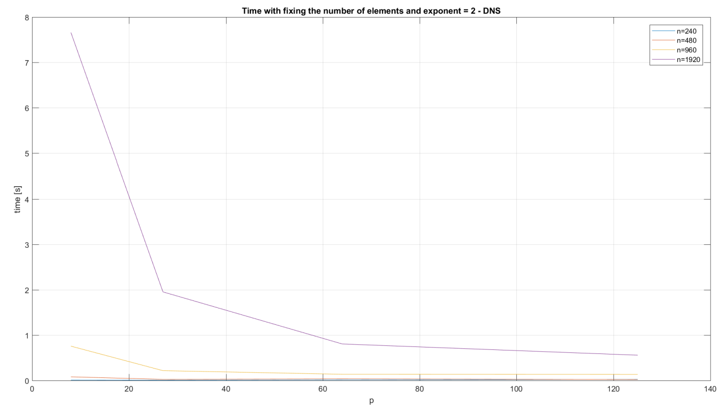

Figure 9: Fixed element n and $k = 2$ - DNS

7.1.4 Fixed p of the topology - DNS

The same considerations made in the Cannon, when p has been fixed, are applied in case of fixing the number of processor of the topology.
The relative graphs are shown in figures 12, 13 and 14.

7.1.5 Comparison between the algorithms

The two algorithms behaves in the same way. Both, increasing the size of the matrix n, increase their run time and, increasing the number of processors p they decrease their run time.
However the run time of the DNS algorithm are better with the respect to the Cannon's.
On the contrary the memory management in the Cannon is better that in the DNS.

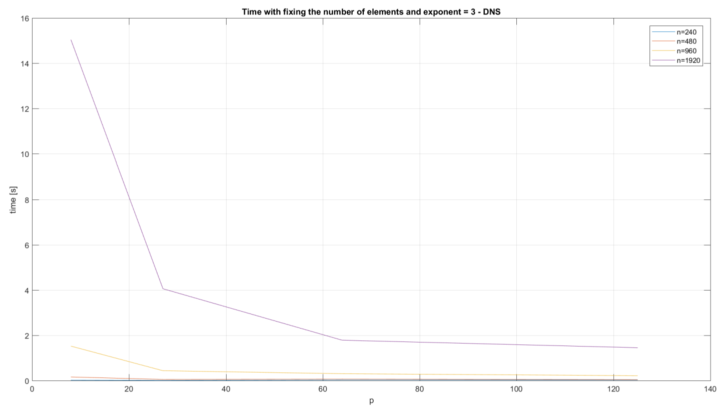

Figure 10: Fixed element n and $k = 3$ - DNS

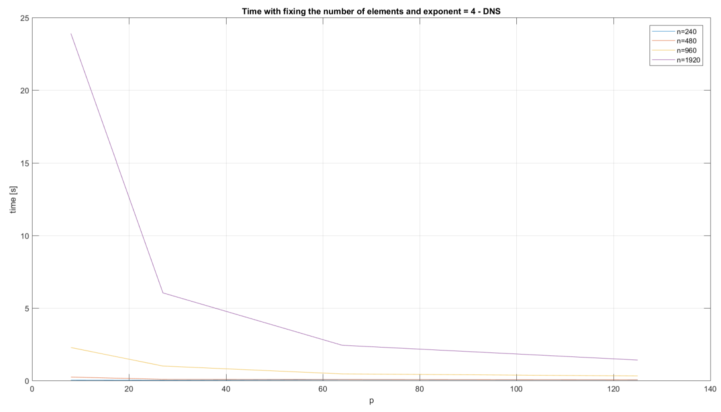

Figure 11: Fixed element n and $k = 4$ - DNS

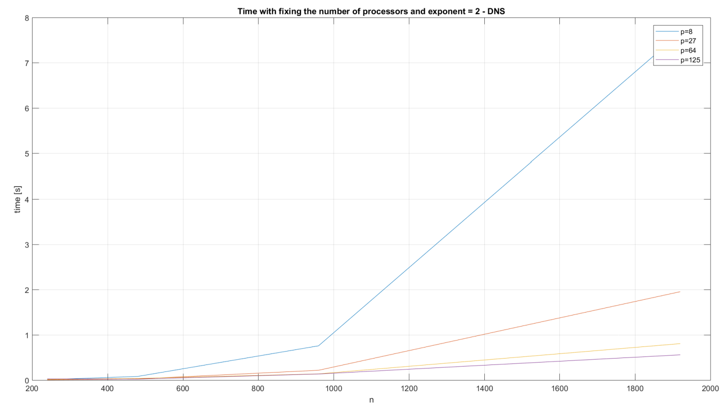

Figure 12: Fixed element p and $k = 2$ - DNS

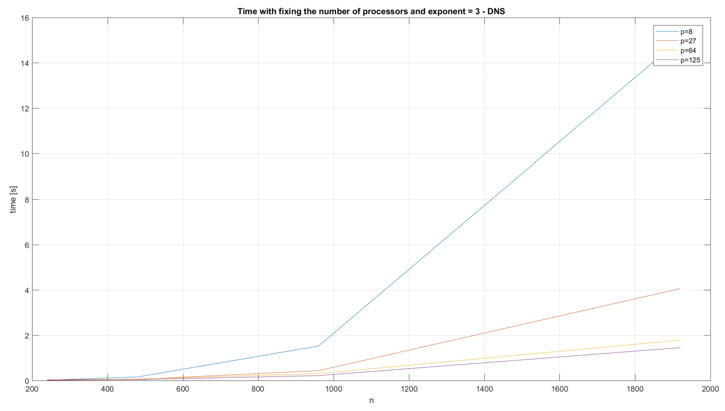

Figure 13: Fixed element p and $k = 3$ - DNS

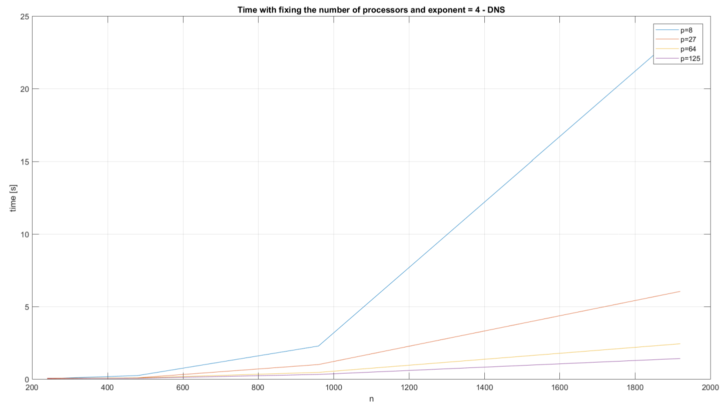

Figure 14: Fixed element p and $k = 4$ - DNS

7.2 Numerical results

The numerical results obtained from the job submission are reported in the following tables.
NOTE: These are the file obtained by the script and for this reason in order to obtain a correct value of the deviation it is necessary to do $\sqrt{Dev/19}$ since 20 iteration on the matrix multiplication code are done.

7.2.1 Cannon's results

mpitest_N_P_K	Time [s]	Dev
mpitest_240_4_2.out	0.035264	0.003091
mpitest_240_4_3.out	0.043412	0.000003
mpitest_240_4_4.out	0.064186	0.000001
mpitest_240_9_2.out	0.016846	0.001927
mpitest_240_9_3.out	0.020440	0.000001
mpitest_240_9_4.out	0.033420	0.002541
mpitest_240_16_2.out	0.006359	0.000001
mpitest_240_16_3.out	0.012439	0.000008
mpitest_240_16_4.out	0.021054	0.000312
mpitest_240_25_2.out	0.005319	0.000056
mpitest_240_25_3.out	0.010395	0.000017
mpitest_240_25_4.out	0.013452	0.000092
mpitest_240_36_2.out	0.017222	0.009556
mpitest_240_36_3.out	0.051898	0.001759
mpitest_240_36_4.out	0.033372	0.007235
mpitest_240_64_2.out	0.010979	0.002210
mpitest_240_64_3.out	0.073157	0.002995
mpitest_240_64_4.out	0.104671	0.046698
mpitest_240_100_2.out	0.053376	0.003032
mpitest_240_100_3.out	0.088368	0.024849
mpitest_240_100_4.out	0.123992	0.031567
mpitest_480_4_2.out	0.165542	0.000002
mpitest_480_4_3.out	0.322443	0.000512
mpitest_480_4_4.out	0.488316	0.000150
mpitest_480_9_2.out	0.077863	0.000022
mpitest_480_9_3.out	0.206964	0.048367
mpitest_480_9_4.out	0.219597	0.000379
mpitest_480_16_2.out	0.045308	0.000032
mpitest_480_16_3.out	0.102006	0.037949
mpitest_480_16_4.out	0.130908	0.002335
mpitest_480_25_2.out	0.034609	0.000721
mpitest_480_25_3.out	0.094936	0.014542
mpitest_480_25_4.out	0.093656	0.006295
mpitest_480_36_2.out	0.044984	0.004360
mpitest_480_36_3.out	0.079352	0.006932
mpitest_480_36_4.out	0.132989	0.002011
mpitest_480_64_2.out	0.019970	0.003153
mpitest_480_64_3.out	0.033685	0.002328
mpitest_480_64_4.out	0.114984	0.008991
mpitest_480_100_2.out	0.051196	0.003271
mpitest_480_100_3.out	0.073596	0.017533
mpitest_480_100_4.out	0.126159	0.006535

mpitest_960_4_2.out	1.534417	0.000008
mpitest_960_4_3.out	3.052898	0.000110
mpitest_960_4_4.out	4.697386	1.384323
mpitest_960_9_2.out	0.640153	0.000009
mpitest_960_9_3.out	1.266948	0.000017
mpitest_960_9_4.out	1.915514	0.016963
mpitest_960_16_2.out	0.353228	0.042796
mpitest_960_16_3.out	0.662118	0.007180
mpitest_960_16_4.out	0.979441	0.000026
mpitest_960_25_2.out	0.225786	0.000135
mpitest_960_25_3.out	0.474418	0.263612
mpitest_960_25_4.out	0.674996	0.083236
mpitest_960_36_2.out	0.333181	0.057813
mpitest_960_36_3.out	0.342094	0.058342
mpitest_960_36_4.out	0.486322	0.058984
mpitest_960_64_2.out	0.167976	0.143311
mpitest_960_64_3.out	0.355685	0.042503
mpitest_960_64_4.out	0.521319	0.105608
mpitest_960_100_2.out	0.125733	0.087764
mpitest_960_100_3.out	0.266310	0.051084
mpitest_960_100_4.out	0.336470	0.008500
mpitest_1920_4_2.out	15.68818	54.26426
mpitest_1920_4_3.out	29.65520	95.61545
mpitest_1920_4_4.out	43.56785	100.4394
mpitest_1920_9_2.out	5.631329	0.191915
mpitest_1920_9_3.out	11.19899	0.585656
mpitest_1920_9_4.out	16.87371	0.971369
mpitest_1920_16_2.out	3.088439	0.003932
mpitest_1920_16_3.out	6.208406	0.293258
mpitest_1920_16_4.out	9.295770	0.505702
mpitest_1920_25_2.out	1.995205	0.003247
mpitest_1920_25_3.out	3.947567	0.006235
mpitest_1920_25_4.out	5.806819	0.014887
mpitest_1920_36_2.out	1.523369	1.698962
mpitest_1920_36_3.out	2.660507	1.211071
mpitest_1920_36_4.out	4.067033	1.659251
mpitest_1920_64_2.out	0.884558	0.631537
mpitest_1920_64_3.out	1.459154	0.302588
mpitest_1920_64_4.out	2.202596	2.013148
mpitest_1920_100_2.out	0.827091	0.081019
mpitest_1920_100_3.out	1.022634	0.364914
mpitest_1920_100_4.out	1.816794	2.334156

7.2.2 DNS results

mpitest_N_P_K	Time [s]	Dev
mpitest_240_8_2.out	0.010719	0.000000
mpitest_240_8_3.out	0.020893	0.000001
mpitest_240_8_4.out	0.030892	0.000003
mpitest_240_27_2.out	0.007095	0.000763
mpitest_240_27_3.out	0.007282	0.000024

```
mpitest_240_27_4.out     | 0.020356 | 0.004539
mpitest_240_64_2.out     | 0.015774 | 0.000676
mpitest_240_64_3.out     | 0.033845 | 0.001239
mpitest_240_64_4.out     | 0.063019 | 0.005414
mpitest_240_125_2.out    | 0.026743 | 0.001893
mpitest_240_125_3.out    | 0.032867 | 0.002580
mpitest_240_125_4.out    | 0.048797 | 0.007532
mpitest_480_8_2.out      | 0.081474 | 0.000002
mpitest_480_8_3.out      | 0.161533 | 0.000171
mpitest_480_8_4.out      | 0.241420 | 0.000002
mpitest_480_27_2.out     | 0.026170 | 0.000007
mpitest_480_27_3.out     | 0.052964 | 0.000539
mpitest_480_27_4.out     | 0.086503 | 0.038391
mpitest_480_64_2.out     | 0.037061 | 0.000759
mpitest_480_64_3.out     | 0.067470 | 0.008263
mpitest_480_64_4.out     | 0.078311 | 0.001509
mpitest_480_125_2.out    | 0.022375 | 0.001213
mpitest_480_125_3.out    | 0.044520 | 0.000532
mpitest_480_125_4.out    | 0.056412 | 0.008323
mpitest_960_8_2.out      | 0.757378 | 0.000700
mpitest_960_8_3.out      | 1.524255 | 0.008671
mpitest_960_8_4.out      | 2.278346 | 0.018271
mpitest_960_27_2.out     | 0.217461 | 0.000164
mpitest_960_27_3.out     | 0.437838 | 0.000289
mpitest_960_27_4.out     | 1.001570 | 0.208252
mpitest_960_64_2.out     | 0.138192 | 0.115765
mpitest_960_64_3.out     | 0.305799 | 0.052919
mpitest_960_64_4.out     | 0.460665 | 0.093883
mpitest_960_125_2.out    | 0.135037 | 0.051661
mpitest_960_125_3.out    | 0.218627 | 0.056946
mpitest_960_125_4.out    | 0.319283 | 0.107732
mpitest_1920_8_2.out     | 7.657850 | 0.069503
mpitest_1920_8_3.out     | 15.03950 | 1.586409
mpitest_1920_8_4.out     | 23.89235 | 25.45206
mpitest_1920_27_2.out    | 1.952571 | 0.165708
mpitest_1920_27_3.out    | 4.057384 | 0.698249
mpitest_1920_27_4.out    | 6.042900 | 11.48114
mpitest_1920_64_2.out    | 0.805840 | 0.002754
mpitest_1920_64_3.out    | 1.784847 | 3.543462
mpitest_1920_64_4.out    | 2.433244 | 1.408748
mpitest_1920_125_2.out   | 0.558334 | 0.220011
mpitest_1920_125_3.out   | 1.453421 | 0.325340
mpitest_1920_125_4.out   | 1.416178 | 2.523127
```

8 Lessons

The lesson learned is very similar to the previous assignment because the topic is again on algorithm related to matrices.
This time the assignment was about the implementation of matrix multiplication algorithm. Since there are many possibilities to reach the same result, the choice has been to focus on the two most clever algorithms that are Cannon's and DNS.

The implementation, as happened in the previous assignments, helped a lot to understand how the algorithms work.
Moreover the assignment has also been useful to learn more functions call of MPI and their behavior.