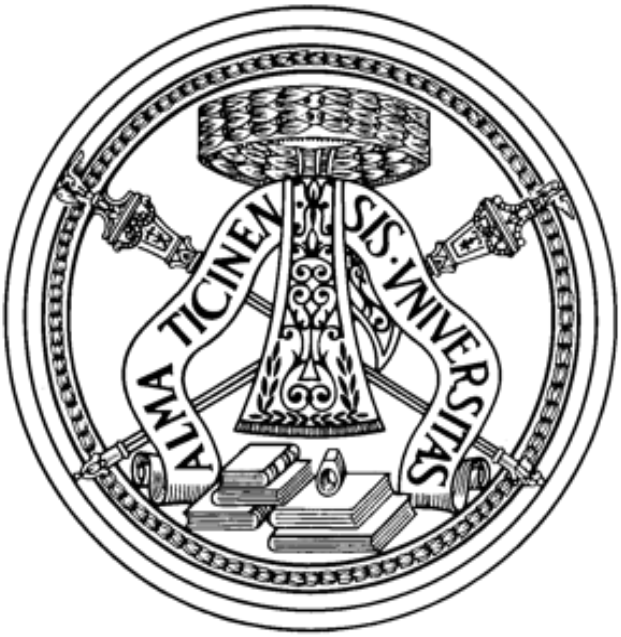
University of Pavia



Serial and parallel implementation of matrix multiplication and inversion

Course of Advanced Computer Architecture

Authors:

Riccardo Crescenti

Riccardo Doveri

INDEX

Introduction 3

Part 1: Analysis of the serial algorithm 3

1.1 Serial matrix multiplication 3

1.2 Serial inversion of a matrix 4

Part 2: A priori study of available parallelism 6

2.1 A priori study in matrix multiplication 6

2.2 A priori study in matrix inversion 7

Part 3: MPI parallel implementation 9

3.1 Product parallelization 9

3.2 Inversion parallelization 11

Part 4: Testing and debugging 13

4.1 Test cases 13

Part 5: Performance and scalability analysis 13

5.0 Theoretical and actual speedup 13

5.1 Local execution analysis 15

5.2 Scalability tests on GCP 16

5.3 Setting up clusters on GCP 18

Introduction

In the scientific research field, two mathematic operations are often used: the computation of the product between matrices and the inversion of a matrix.

The goal of this project is to implement both serial algorithms, parallelize them using open MPI and perform a performance and speed up analysis executed both in a local machine and through scalability tests on clusters built using Google Cloud Platform.

All the work has been carried out jointly, thanks to video calls and in person meetings.
The goals, the structure of the project and the methods used were defined together, with step-by-step reviews.

Part 1: Analysis of the serial algorithm

1.1 Serial matrix multiplication

Let $A \in R^{mxn}$ and $B \in R^{nxp}$, the product $C \in R^{mxp}$ of A and B is defined elementwise as:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}, \forall i \in \{1, 2, \ldots, m\}, \forall j \in \{1, 2, \ldots, p\}$$

The entries of the resulting matrix C are found by the product of the elements of the rows from the first matrix A multiplied by the associated elements of the columns from the second matrix B and summing up the products.
There exists a constraint about the dimensions of matrices in order to compute the product: the number of columns of the first matrix must be equal to the number of rows of the second matrix. The result matrix has always the same number of rows of the first matrix and the same number of columns of the second matrix.

Pseudocode of the algorithm:

Algorithm 1 Matrix Multiplication

INPUT: Matrices A and B
OUTPUT: Matrix C

```
1: for i = 0, 1, 2, ..., m do
2:     for j = 0, 1, 2, ..., p do
3:         sum = 0
4:         for k = 0, 1, 2, ..., n do
5:             sum+ = A_{i,k} · B_{k,j}
6:         end for
7:         C_{ij} = sum
8:     end for
9: end for
```

1.2 Serial inversion of a matrix

Given A ∈ R^{nxn} non-singular matrix (determinant of A is different from zero), the inverse A^{-1} is defined as:

$$AA^{-1} = I$$

There are some ways of computing a matrix inverse: we decided to use LU decomposition over Cramer method since it has less computational cost. Cramer method has $O(n + 1!)$ complexity, instead LU decomposition (i.e. splitting the initial matrix A into two triangular matrices: L lower triangular and U upper triangular) has $O(\frac{8}{3}\, n^3)$ which is a better condition since n may become very large for scalability reasons.

Initially, we thought of carrying out the LU decomposition, calculating the inverse of the single triangular matrices, in this way we could exploit the matrix product program, previously created, by calculating inverse of L and inverse of U and multiplying them. However, this strategy in the parallelization phase was not very effective, as we realized that it was rather complex to manage two matrices and divide them between processes. For this reason, we decided to take a cue from the [2] code and adopt the same steps to obtain the inverse matrix.

The algorithm consists of:

1) LU decomposition with pivoting (this condition was not considered in the previous implementation), A is overwritten

```
LU decomposition

- initialization of array P[n + 1] a {0, 1, 2, ..., n}
  useful later for check determinant and inversion

for each column c of A[n][n] {
        find the row which has the max value on c
        switch rows of A and respective elements of P
        if there was switch {
                P[n]++
        }

        for each row r from A[c + 1] {
                A[r][c] /= A[c][c]
                for k = c + 1 to k = n - 1 {
                        A[r][k] -= A[r][c] * A[c][k];
                }
        }
}
```

Figure 1. Pseudocode LU

2) Determinant check in order to verify through the elements of the diagonal of A that the determinant is different from zero

```
Check determinant

det = A[0][0]
for each element el of A diagonal {
        det = det * el
        if (P[n] - n) is odd {
                det = -det
        }
}

if det = 0 {
        stop the program because A is not invertible
}
```

Figure 2. Pseudocode Check Determinant

3) LUP inversion, the real inversion algorithm

```
Inversion

for each element el[r][c] of Ainv {
        if P[r] == c {
                el = 1
        } else {
                el = 0
        }
}

for each column of Ainv {
        for each row in increasing order {
                for k = r + 1 to k = n - 1 {
                        Ainv[r][c] -= A[r][k] * Ainv[k][c]
                }
        }

        for each row in decreasing order {
                for k = r + 1 to k = n - 1 {
                        Ainv[r][c] -= A[r][k] * Ainv[k][c]
                }
                Ainv[r][c] /= A[r][r]
        }
}
```

Figure 3. Pseudocode Inversion

Part 2: A priori study of available parallelism

Before starting the parallel coding part, we performed a priori study of available parallelism, exploiting the Amdahl's law, that gives the theoretical speedup that a parallel application can achieve.

$$Speedup(N) = \frac{Time_{serial}}{Time_{parallel}(N)} = \frac{(S+P)T_{serial}}{S \cdot T_{serial} + \frac{P \cdot T_{serial}}{N}} = \frac{S+P}{S+\frac{P}{N}} = \frac{1}{S+\frac{P}{N}}$$

Where P is the fraction of code parallelizable, S is the fraction of code not parallelizable.
S + P = 1 and N cores.
In chapter 5.0 will be compared the theoretical speedup with the actual one.

2.1 A priori study in matrix multiplication

The first step we thought in order to make the product between matrices in parallel was to divide the tasks between processes. Each process is assigned a portion of the matrix A (more precisely, $rows\_assigned = \frac{n\_rows\_of\_A}{n\_processes}$).

If the number of rows is not a multiple of the number of processes, these are computed by the master process. Subsequently, all matrix B is broadcast to each process. In this way, each process can independently calculate a portion of the product.

An example of domain decomposition

In the example below the product between two matrices of the same dimensions 4x4 is made in the case in which there are 2 processes. Matrix A is divided between two processes, the first two rows are assigned to the first process and the last two to the second process. Each process can make the product row by column with all the elements of B.

	A00	A01	A02	A03		B00	B01	B02	B03
A	A10	A11	A12	A13	B	B10	B11	B12	B13
	A20	A21	A22	A23		B20	B21	B22	B23
	A30	A31	A32	A33		B30	B31	B32	B33

	C00	C01	C02	C03	A TUTTI
C	C10	C11	C12	C13	PR. 0
	C20	C21	C22	C23	PR 1
	C30	C31	C32	C33	

Figure 4. Matrix Product Domain Decomposition

2.2 A priori study in matrix inversion

There are several constraints regarding the inversion of matrices, as some operations are necessarily concerning serial (it is needed to wait the previous results to be able to calculate the following ones).

For what concerns the pivoting function: the search for the maximum of each column is divided between processes. Each process receives $\frac{n\_rows\_of\_A}{n\_processes}$ rows and searches for its local maximum, then it forwards the local max_index to the Master process, which finds the global maximum among all the local maxima computed by the various processes and after that it can swap rows.

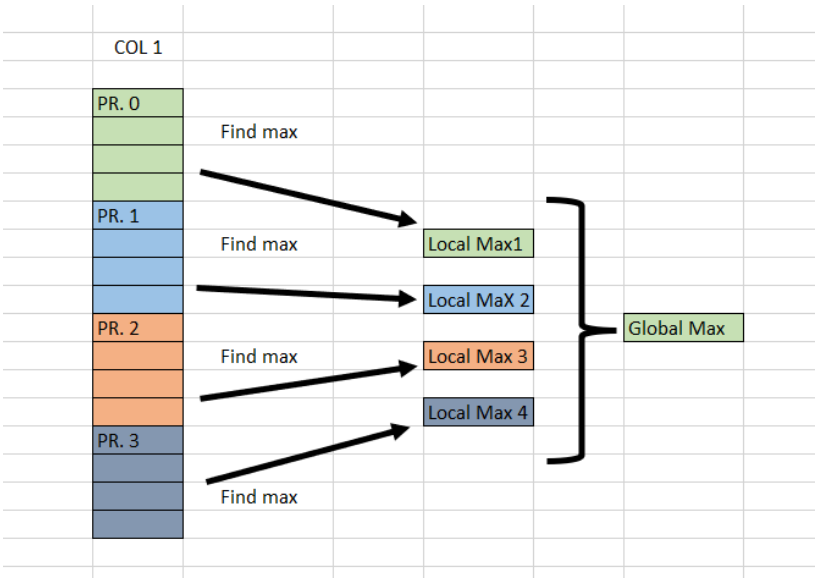


Figure 5. Local Maxima Domain Decomposition

The LU Decomposition is divided as follows: for each column, a main row and a portion of rows per process will be sent. Each process calculates its own portion and sends it back to the master process.

FIRST ITERATION

A	A00	A01	A02	A03
	A10	A11	A12	A13
	A20	A21	A22	A23
	A30	A31	A32	A33

A TUTTI
PR. 0
PR 1

SECOND ITERATION

A'	A'00	A01	A02	A03
	A'10	A11	A12	A13
	A'20	A21	A22	A23
	A'30	A31	A32	A33

Figure 6. LU Domain Decomposition

Therefore, for the check determinant function we decided to adopt a similar strategy used in the search for the maximum function: the elements on the diagonal of A are divided equally among all the processes (obviously managing the rest, in the case in which number_of_rows is not a multiple of number_of_processes) in order to be able to calculate the product subdivided in blocks and then return the partial results to the master process.

The last function that has been parallelized is the actual inversion. In this case, the possible parallelization is limited since the calculations per row must necessarily be performed in serial mode. In this case the previously obtained matrix A' is broadcasted to all processes, and the calculation of the Ainv (the actual inverse of A) is divided between processes by columns, as the calculations are independent. Indeed, each element of A inverse depends on the elements of the same column of A inverse (so, it must be serial computation); each element of A inverse depends also on the elements of the same row of A'.

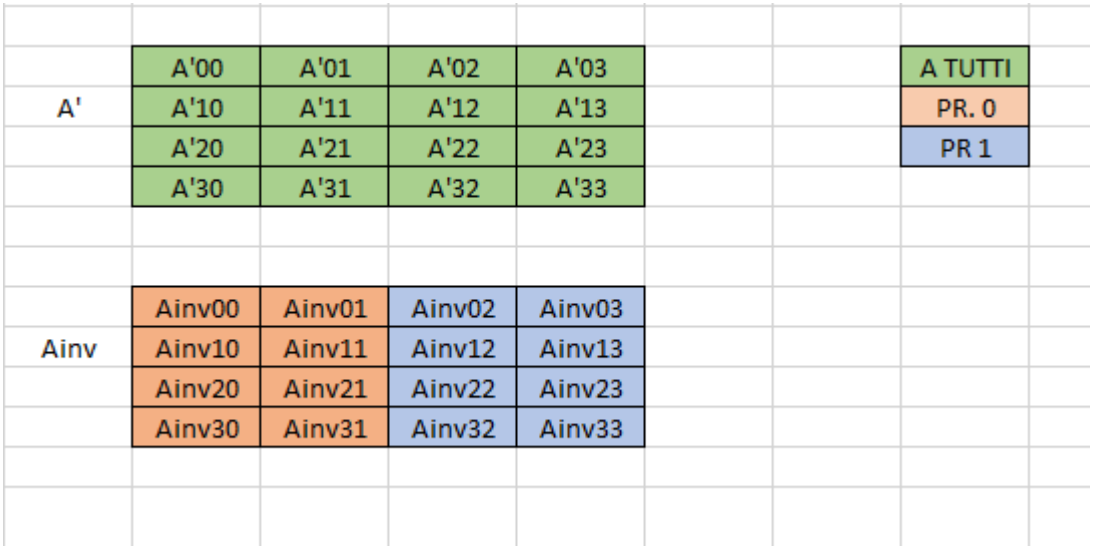


Figure 7. Inversion Domain Decomposition

Part 3: MPI parallel implementation

3.1 Product parallelization

For what concerns the actual implementation of the code, first of all are invoked the main functions in order to initialize the communication environment by all processes: parameters argc and argv are are taken by main method, myRank is the rank of the actual process and P is the size, so the total number of processes.

```
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
MPI_Comm_size(MPI_COMM_WORLD, &P);
```

Master process is in charge of reading both matrices. Actually, this part may be divided between two processes (reading one matrix each), but for simplicity we decided to keep it that way, in fact it would not have been a scalable solution on a larger number of processes. *r1,c1,r2,c2* are respectively the number of rows and columns of matrix 1 and rows and columns of matrix 2

```
if (myRank == 0){
    //Reading matrices from files
    matrix1 = readMatrixFromFile(argv[1], &r1, &c1);
    if(matrix1 == NULL) {
        printf("File %s is not correct\n", argv[1]);
        return;
    }

    matrix2 = readMatrixFromFile(argv[2], &r2, &c2);
    if(matrix2 == NULL) {
        printf("File %s is not correct\n", argv[2]);
        return;
    }
```

The dimensions and variables *q* and *r* ($q = \frac{n\_rows\_of\_matrix\_1}{n\_processes}$, $r = r1 \,\%\, P$) are broadcast from the master process so all the processes can allocate the right dimensions.

```
//broadcasting dimensioni
MPI_Bcast(&r1, 1, MPI_FLOAT, 0, MPI_COMM_WORLD);
MPI_Bcast(&c1, 1, MPI_FLOAT, 0, MPI_COMM_WORLD);
MPI_Bcast(&r2, 1, MPI_FLOAT, 0, MPI_COMM_WORLD);
MPI_Bcast(&c2, 1, MPI_FLOAT, 0, MPI_COMM_WORLD);
MPI_Bcast(&q, 1, MPI_FLOAT, 0, MPI_COMM_WORLD);
MPI_Bcast(&r, 1, MPI_FLOAT, 0, MPI_COMM_WORLD);
```

Are then called some MPI functions:
Broadcast: The master process sends to all processes *matrix 2*, that *has r2 * c2* float elements
Scatter: Matrix 1 is scattered among all processes by the master in portion of q * c1 float elements. The computation of the product for each portion (with the correct dimensions) is stored in the *localResult* variable. Finally, with the Gather function, all the results calculated by the individual processes are returned back to the master process.

```
MPI_Bcast(&matrix2[0][0], r2 * c2, MPI_FLOAT, 0, MPI_COMM_WORLD);

MPI_Scatter(matrix1[0], q * c1, MPI_FLOAT, &recvMatrix1[0][0], q * c1, MPI_FLOAT, 0, MPI_COMM_WORLD);

localResult = matrixProduct(recvMatrix1, q, c1, matrix2, r2, c2);

MPI_Gather(localResult[0], q * c2, MPI_FLOAT, &resultMatrix[0][0], q * c2, MPI_FLOAT, 0, MPI_COMM_WORLD);
```

The master process will then take care of calculating the remaining portions (if the number of processes is not a multiple of the rows of matrix1). In conclusion the previously allocated variables are deallocated and the MPI_Finalize() function is called in order to finalize the communication environment.

3.2 Inversion parallelization

As before, MPI functions are invoked:
MPI_Init(&argc, &argv); MPI_Comm_rank(MPI_COMM_WORLD, &myRank) and MPI_Comm_size(MPI_COMM_WORLD, &size).

First of all, the master process has the task of reading and allocating matrix A. In this case it is necessary to broadcast only one parameter n, which is the number of rows and columns of matrix A, which must necessarily be a square matrix in order to be inverted.

```
MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

At this point, as explained in the previous chapter in the decomposition of the domain, for each column the portions of matrix A are sent to all processes (except the master).
A for loop is then executed for all p processes (except the master) where the correct matrix portions are sent by the master process through the Standard Send MPI_Send function.

```
for (int p = 1; p < size; p++){
    for (int i = q * p + r + col; i < q * p + r + col + q; i++){
        retVal = MPI_Send(&A[i][col], 1, MPI_FLOAT, p, 555, MPI_COMM_WORLD);
    }
}
```

Then MPI_Recv function is called, it allows each process to receive the corresponding portion of the matrix in the vector colArray.

```
for (int i = 0; i < q; i++){
    retVal = MPI_Recv(&colArray[i], 1, MPI_FLOAT, 0, 555, MPI_COMM_WORLD, &status);
}
```

After calculating all the local maximums, the MPI_Gather function is called, which allows the master to receive the index of the local maximum value calculated by each process

```
MPI_Gather(&imax, 1, MPI_INT, &indexVector[0], 1, MPI_INT, 0, MPI_COMM_WORLD);
```

The master process then knows the pivot index and can exchange the rows. The LU parallel decomposition starts with the MPI function Bcast that allows to broadcast the main row.

```
MPI_Bcast(&mainRow, n, MPI_FLOAT, 0, MPI_COMM_WORLD);
```

Are then sent portions of matrix A to all processes with MPI_Send and MPI_Recv functions with the corresponding elements. After having carried out all the calculations, the results are received back to the master.

```
if (col < n - size){
    if (myRank != 0){
        for (int i = 0; i < q; i++){
            retVal = MPI_Send(&matrixPortion[i][0], n, MPI_FLOAT, 0, 555, MPI_COMM_WORLD);
        }
    }
    if (myRank == 0){
        for (int p = 1; p < size; p++) {
            for (int i = 0; i < q; i++){
                retVal = MPI_Recv(&A[col + 1 + r + p * q + i][0], n, MPI_FLOAT, p, 555, MPI_COMM_WORLD, &status);
            }
        }
    }
}
```

Following the same sending and receiving principles, the determinant is checked (likewise the function that calculates the local maxima) and the actual inversion is calculated. P and A' decomposed are then sent to each process. Subsequently, the inverse calculation of A is performed by each process column by column and the data is sent to the master.
In conclusion, the MPI_finalize function is invoked.

Version 2 and 3 of inversion parallelization

After implementing the first version of the matrix inversion parallelization code described above, we calculated the execution times of each parallel function of the program. We realized that the inversion function from A' to Ainv showed improvements visible immediately (as in the a priori study, indeed the matrix is divided into blocks of independent columns between the processes).

On the other hand, the functions for calculating the maximum and determining check, for the cases tested, were not particularly efficient. In our opinion this is due to the fact that the execution time for these operations is so short that it is not necessary to parallelize (for the matrix dimensions tested).

The LU decomposition function contains many dependencies and therefore the parallel function, for the tests carried out, it is similar to the serial one. For this reason, we decided to create a v2 version of the program in which we have replaced the portion of code in parallel with the serial one. In the v3 version we performed the LU decomposition, maximum search, and determinant check functions in serial, keeping only the true inversion function in parallel.

The results in chapter 5 showed that version 1 and version 2 are quite the same for the test cases performed, while version 3 is better.

We think this is due to over-parallelization for relatively large matrices (our tests are limited to cases of 3000x3000). Based on our studies in the case of even larger matrices we think that this difference will not be so deep and the v1 will overcome v3.

Part 4: Testing and debugging

4.1 Test cases

Initially, we carried out the testing and debugging phases on a local virtual machine (CENTOS 8), then we moved to Google Cloud Platform to verify the speedup that can be reached with a greater number of cores.
The testing phase began with a check of the validity of the algorithms in serial mode for small matrices with fixed dimensions by checking the correctness of the results with an online matrix calculator [3] both for product and inversion, and the intermediate decomposition step.
Then we tested the serial code using very large matrices dimensions (using memory allocation functions). After that, the implementation of the parallel code was carried out with the variation of 2 - 4 cores for small and large dimensions of matrices (managing the cases in which the dimensions of matrices were not multiple of the number of cores).
In conclusion we carried out the scalability tests (discussed in the next chapter) on Google Cloud Platform.
The createMatrix.c file allowed us to create user-defined size matrices from the command line with random elements. The matrices created used for the tests have the following dimensions: 50x50, 250x250, 500x500, 1000x1000, 2000x2000, 3000x3000.
For simplicity in the tests, we used only square matrices, despite the algorithm for calculating the product between matrices can work with matrices of different dimensions (respecting the constraints of chapter 1).

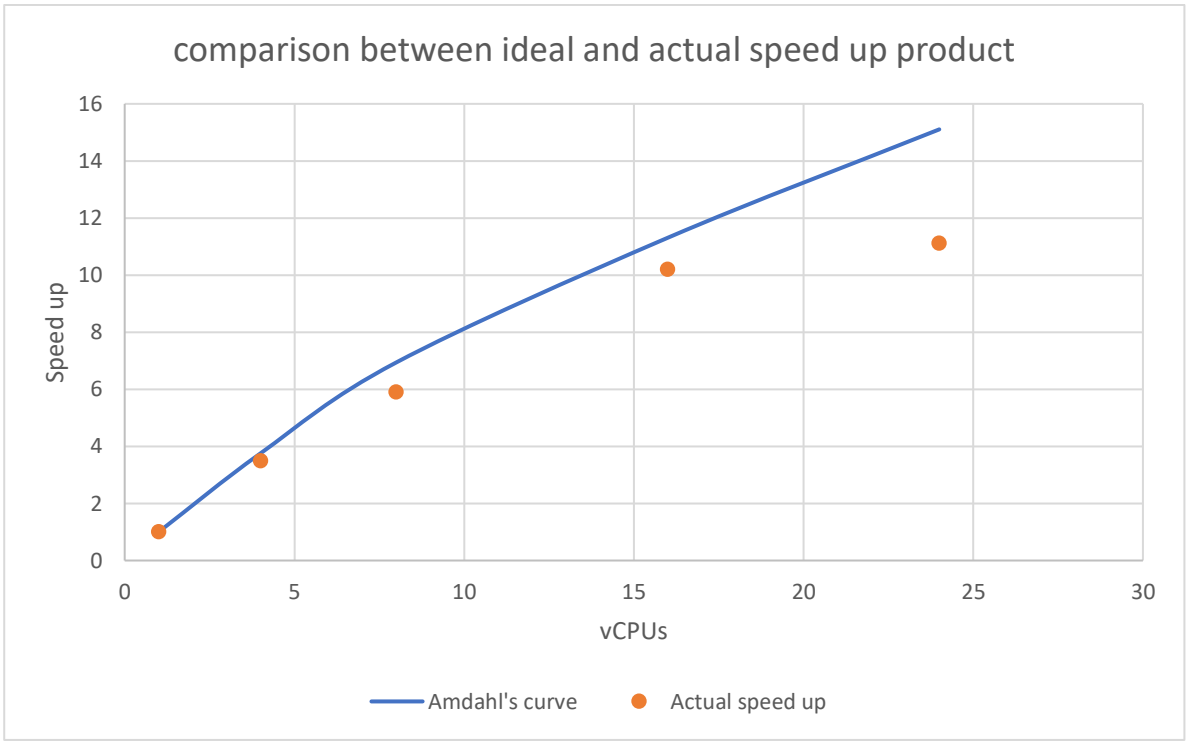
Part 5: Performance and scalability analysis

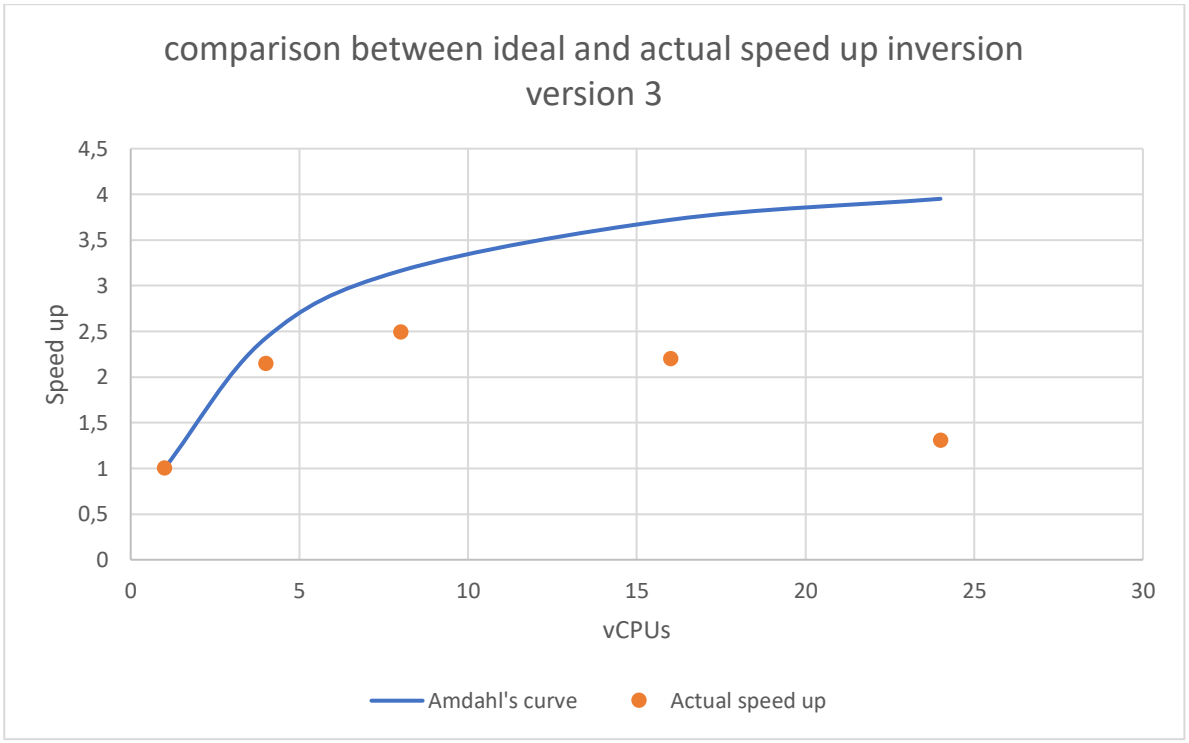
5.0 Theoretical and actual speedup

The first two plots represent a comparison between ideal and actual speed up of the product and the inversion of matrices. We are considering fixed size matrices of dimension 3000x3000.

As we can see from the plots the speedup is clearly effective for a limited number of cores, this fact is due to the matrix dimensions. Indeed, a big overhead is introduced when running the algorithm small matrices with a huge number of cores. In particular, for the inversion algorithm it is clearly visible that using an increased number of threads does not produce any significant improvement. Our tests in a 3000x3000 matrix inversion with 24 number of cores shows that the workload of initialization of the cores introduces a noticeable overhead.

Keeping at 24 the number of cores, and enlarging enough the matrices dimensions will enhances the speed up, reaching good results in the parallel performance evaluation.

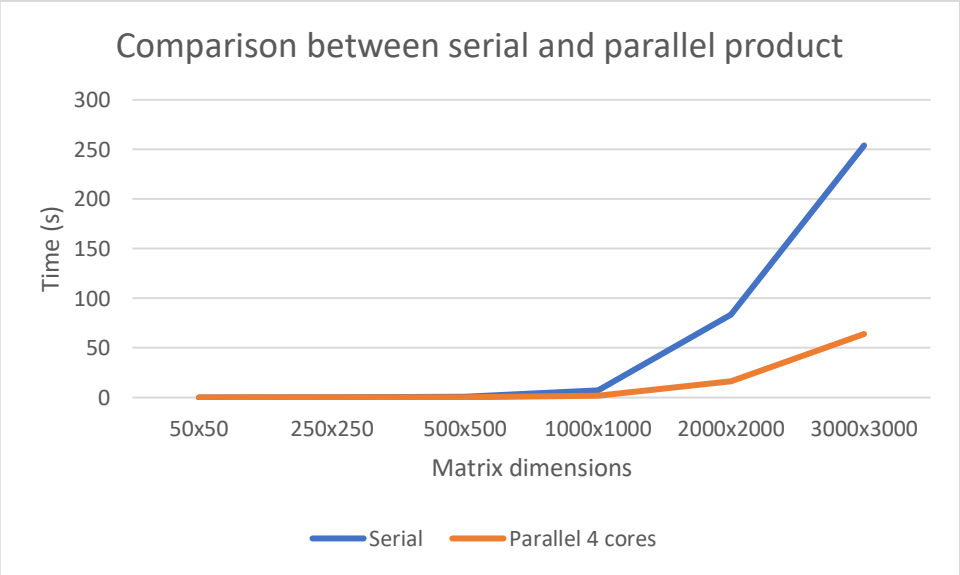

comparison between ideal and actual speed up product
Speed up
16
14
12
10
8
6
4
2
0
0
5
10
15
20
25
30
vCPUs
Amdahl's curve
Actual speed up

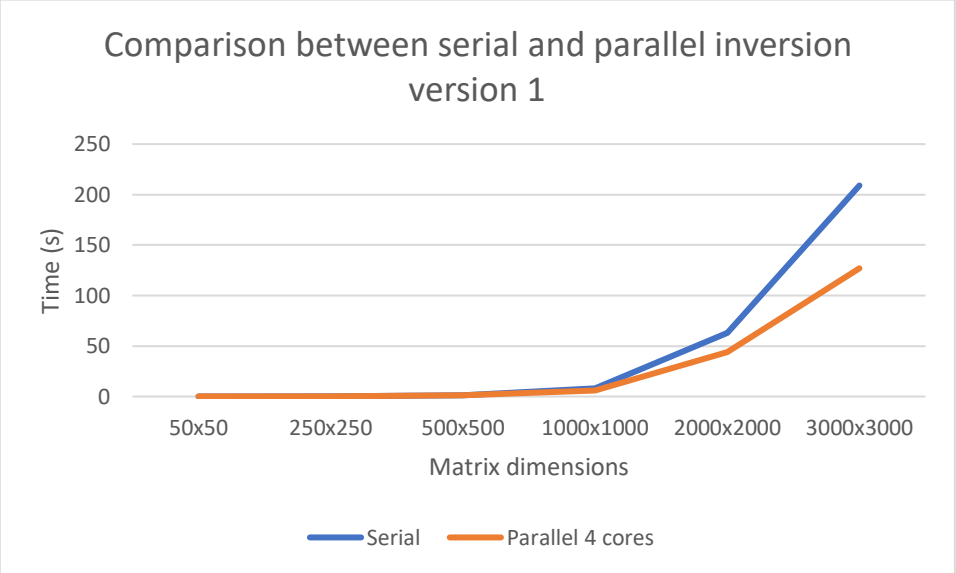

comparison between ideal and actual speed up inversion version 3
Speed up
4,5
4
3,5
3
2,5
2
1,5
1
0,5
0
0
5
10
15
20
25
30
vCPUs
Amdahl's curve
Actual speed up

5.1 Local execution analysis

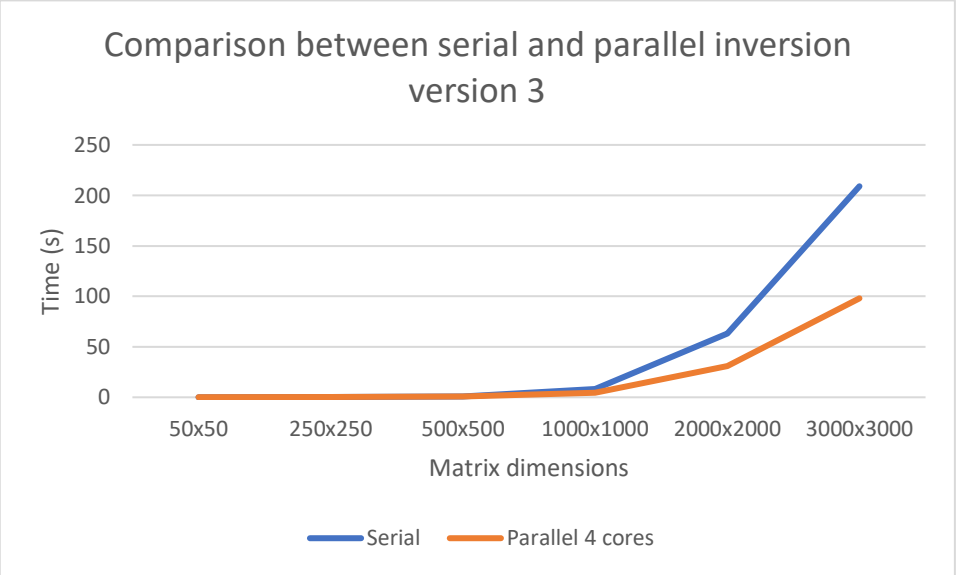
All the tests were realized within the same local machine, an Intel i7-9750H CPU @2.60GHz. However, the project has been carried out on CENTOS 8 using a virtual machine with 4 cores.

The first plot is a graphical comparison between serial and parallel product using 4 cores, the second one is the comparison between serial and parallel inversion using the same parameters as before. As we expected, the serial codes are faster than the parallel one for small matrices dimensions, because the amount of operation is low with respect to the workload of splitting tasks among all threads. For matrices of bigger dimensions, the parallel has good performance improvements.





The third plot represents the comparison between serial and parallel matrix inversion with the version 3, that is the one in which only the inversion function is parallelized. The data confirms the visual representation of the graph. For this set of tests carried out (at a limited dimensions 3000x3000) version 3 is better than version 1. Version 2 data are comparable with version 1.

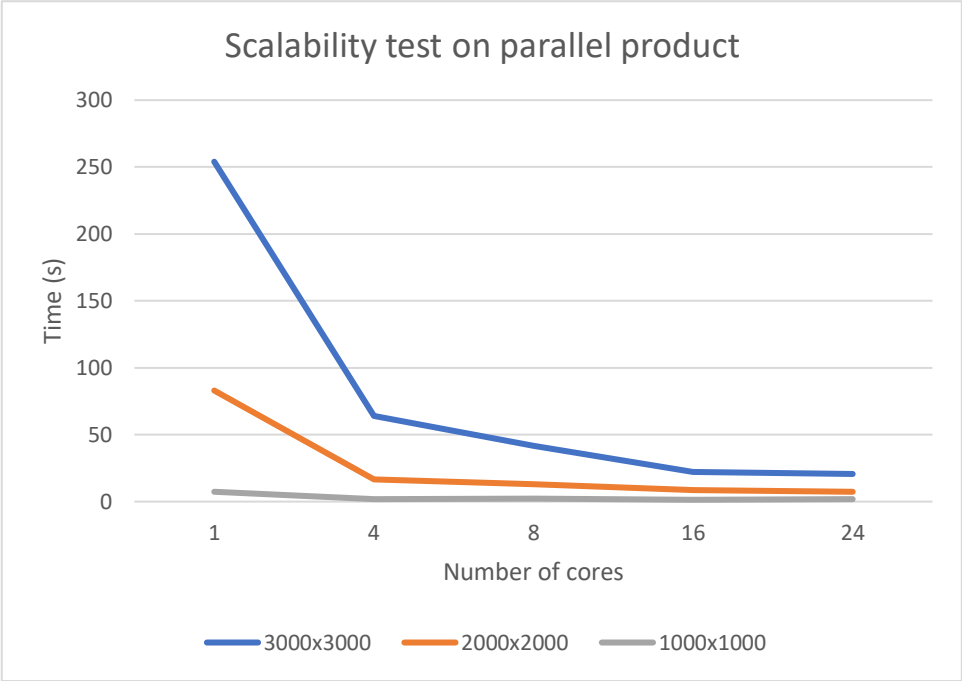


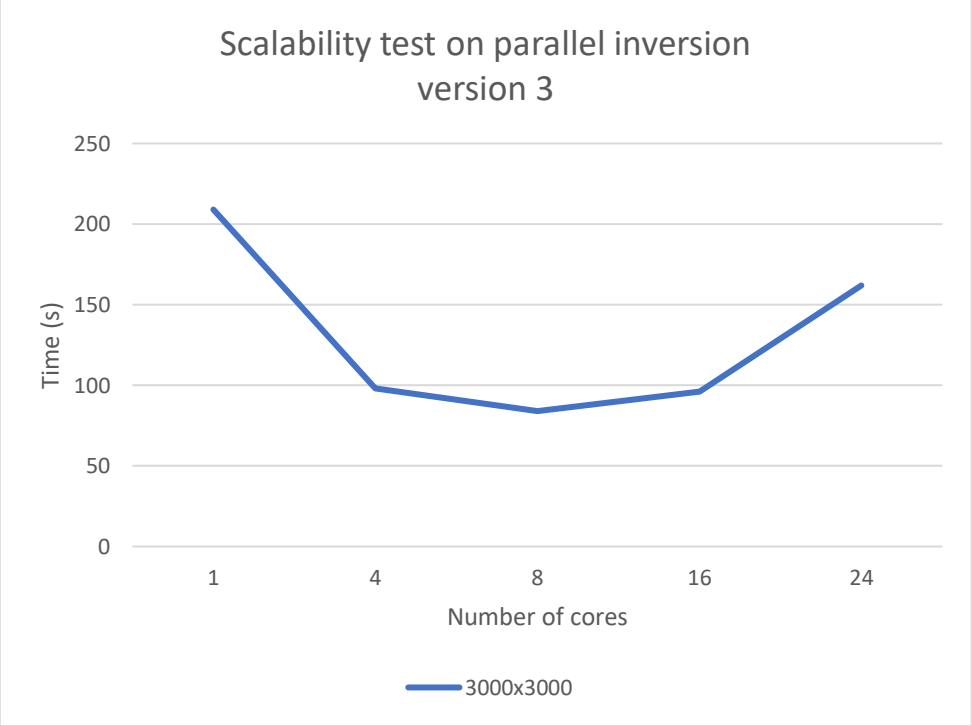
5.2 Scalability tests on GCP

The next step was therefore to carry out scalability tests on the google cloud platform, carrying out tests on all matrix sizes, depending on the number of cores. In this case we used a virtual machine instance with a maximum number of cores equal to 24, 96 GB of memory and 50 GB of disk.

We then carried out scalability tests at varying serial and 4, 8, 16, 24 cores for all versions: matrix product, inversion v1, v2, v3.

As can be seen from the plots, scalability is particularly effective in the parallel program of the product, by increasing the number of cores, a significant improvement in performance is obtained for all the dimensions of the tested matrices. As for the scalability test in v3 inversion, there is an increase in execution time when the number of processes is 16 and 24. This is due, as explained above, to an overhead for arrays of this size. Being able to perform tests on much larger matrices would result in an improvement in performance for 16 and 24 cores.





5.3 Setting up clusters on GCP

The project is then set up on the Google Cloud Platform building clusters, whose data will be presented in this chapter:

The objective of the analysis is to evaluate previously created programs in clusters, that is: an arbitrary number of computers connected by a network. Two types of clusters have been built: fat clusters and light clusters.

In addition, a further analysis has been carried out concerning the regions where the computers are located. Infra-regional (therefore, within the same region, in our case US central 1a) and inter-regional (i.e., between multiple regions: US central, Oregon, South Carolina, North Virginia) clusters will be discussed.

The constraints are the following ones: for infra-regional the maximum total available cores are 24, while distributing the load over multiple regions, the limit is raised to 32.
A further constraint is due to the maximum number of machines that can be used simultaneously in the same region: Google Cloud Platform grants a maximum of 8 IPs available at the same moment.

The clusters were configured through machine image creations, which allowed in a very fast way to copy the main node, on which open MPI and all the C programs and test matrices were present.

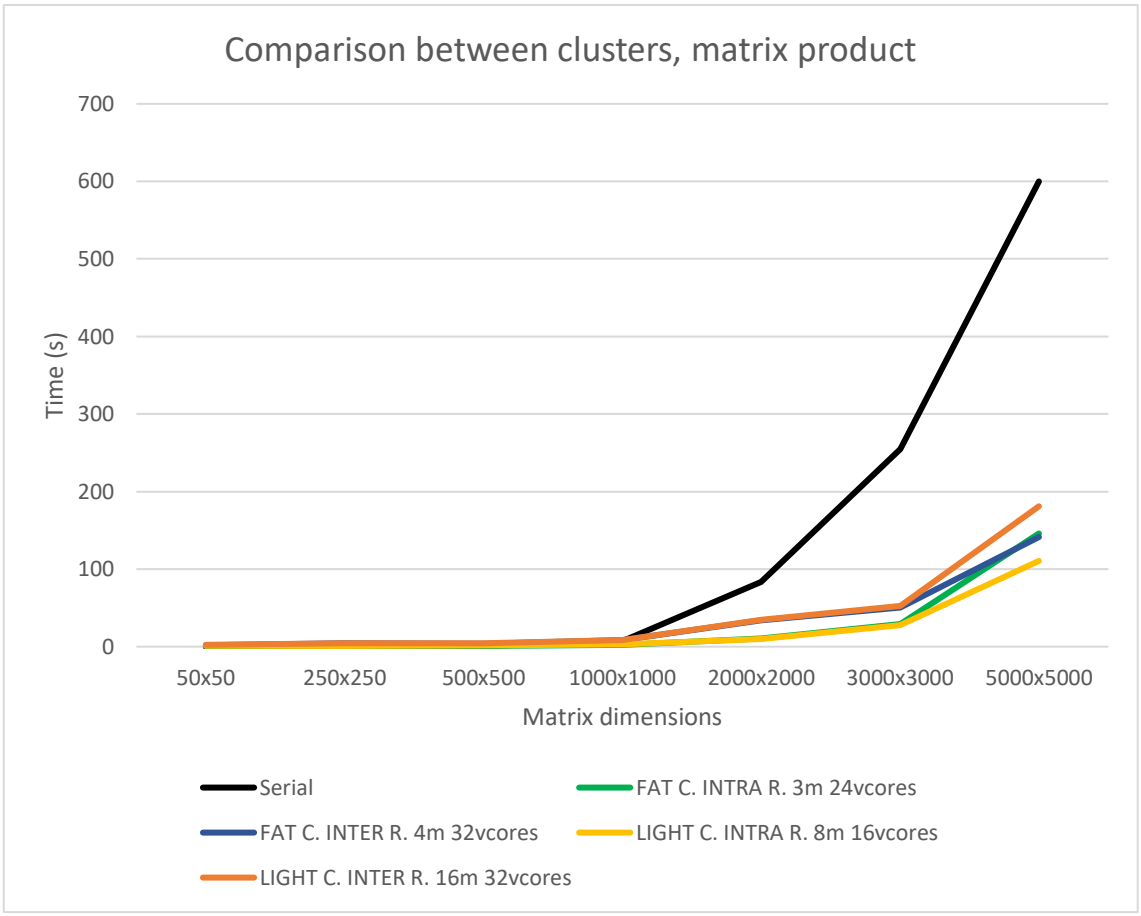
Second generation CPUs were selected, whose CPU platform is selected based on availability.
The 8-core machines for fat clusters were configured with 36GB of RAM and 50GB of disk.
The 2-core machines for light clusters were configured with 8GB of RAM and 50GB of disk.

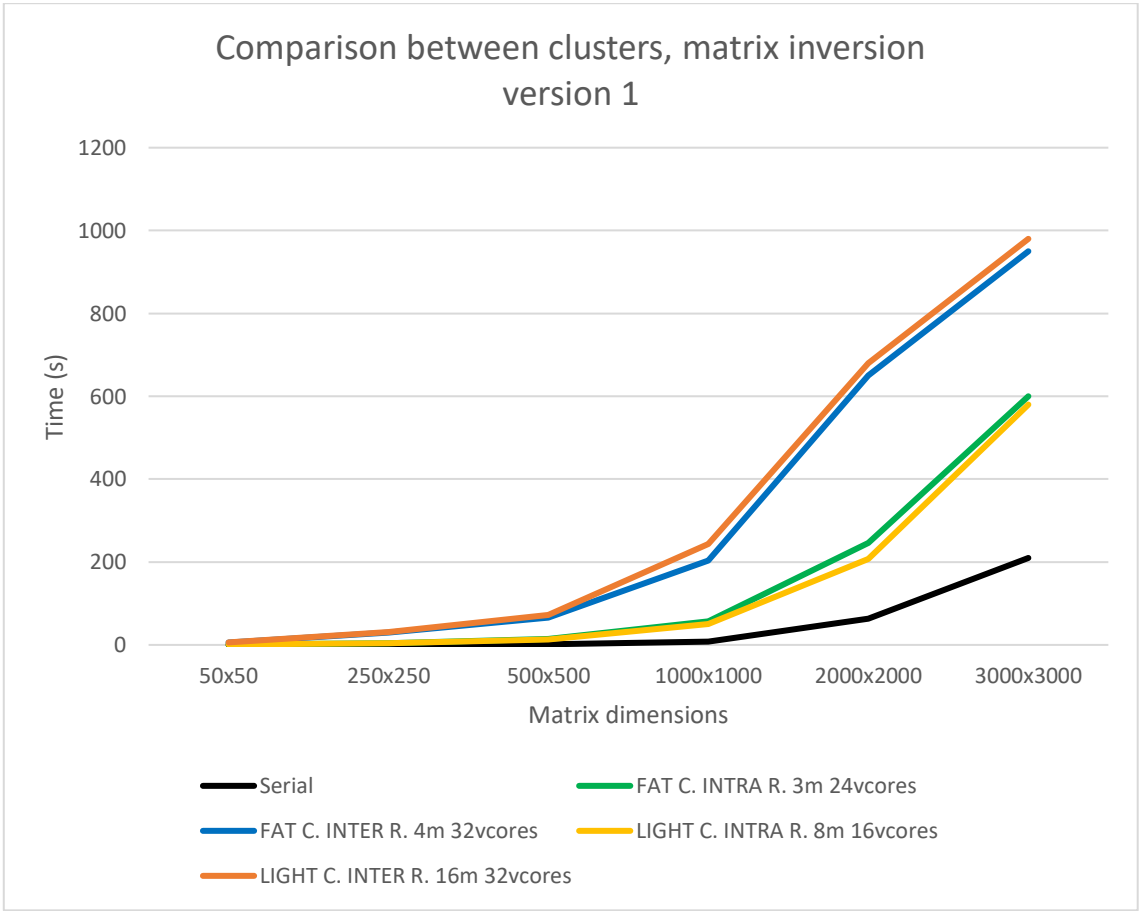
Fat cluster and light cluster analysis

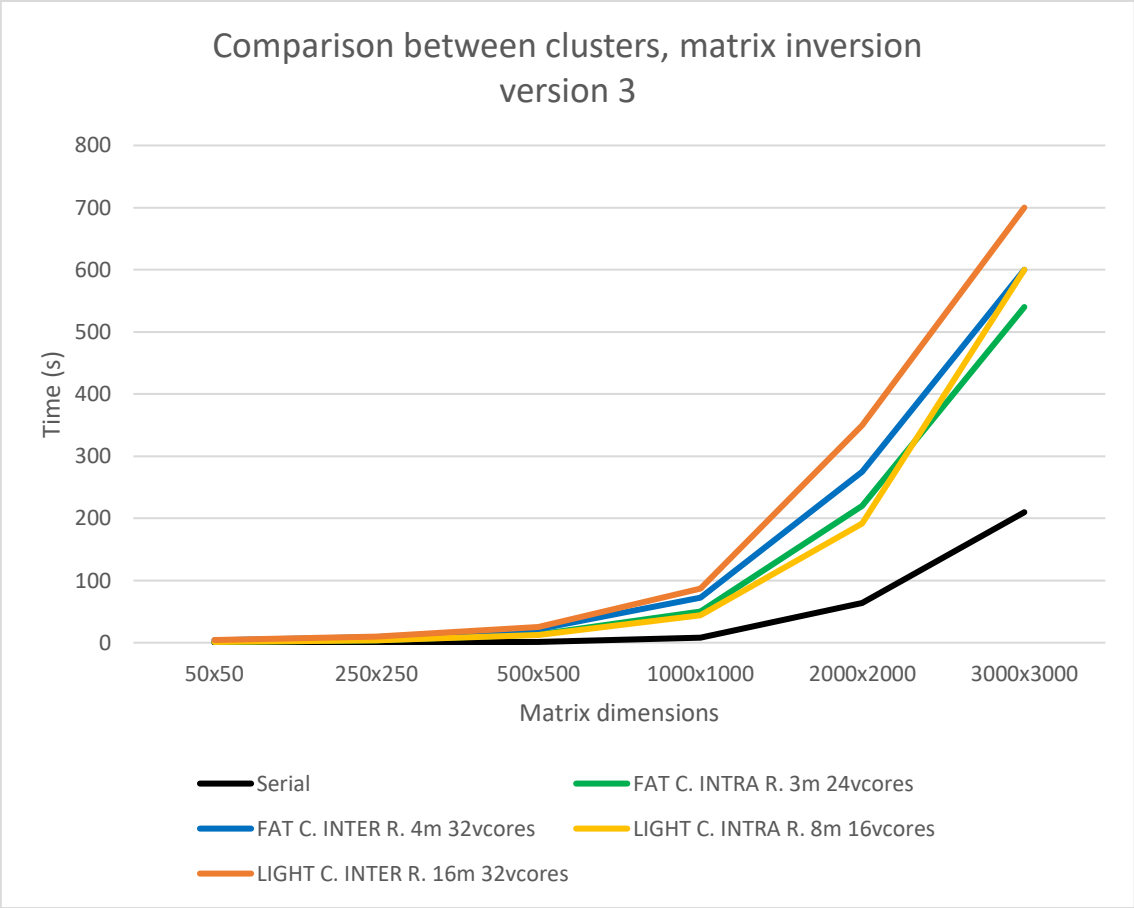
In the following plots both types of clusters will be analysed, located both in the same region and in different regions.

Fat cluster:
- 3 machines with 8 core each, means 24 total infra-regional (US central)
- 4 machines with 8 cores each, equal to 32 inter-regional (1 machine per region: US central, Oregon, South Carolina, North Virginia)

Light cluster:
- 8 machines with two cores each means 16 total infra-regional cores (US central), it was not possible to choose a configuration with a higher number of cores as GCP limits the maximum number of addresses per region to 8.
- 16 machines with two cores each equals to 32 inter-regional cores (4 machines per region, the same as before).


Comparison between clusters, matrix product
700
600
500
400
300
200
100
0
Time (s)
50x50
250x250
500x500
1000x1000
2000x2000
3000x3000
5000x5000
Matrix dimensions
Serial
FAT C. INTRA R. 3m 24vcores
FAT C. INTER R. 4m 32vcores
LIGHT C. INTRA R. 8m 16vcores
LIGHT C. INTER R. 16m 32vcores


Comparison between clusters, matrix inversion version 1
1200
1000
800
600
400
200
0
Time (s)
50x50
250x250
500x500
1000x1000
2000x2000
3000x3000
Matrix dimensions
Serial
FAT C. INTRA R. 3m 24vcores
FAT C. INTER R. 4m 32vcores
LIGHT C. INTRA R. 8m 16vcores
LIGHT C. INTER R. 16m 32vcores



After carefully analysing the data presented in the previous plots, we can draw some conclusions. We can see a difference between inter-regional and intra-regional clusters. The physical position of the machines in fact influences the results.

Physical distance between the machines even if it is initially large, it remains fixed, so at the beginning the inter-regional cluster has worse performance, then it becomes better because it has 32 cores instead of 24. In the product it is also noted for 5000x5000 matrices (fat cluster), while by inversion it would be necessary to try larger dimensions.

Also, int the matrix product in light cluster we are over-parallelizing with 32 cores (we assume that in the long run it will be seen, but for very large matrices).

Matrix product

Carrying out a deeper and more detailed analysis, the best solution is:
- For matrices whose size is less than 2000x2000, intra-regional fat cluster (excluding the 50x50 serial)
- For matrices with dimensions greater than 3000x3000, intra-regional light cluster is better (which at the beginning has worse performance, but for large dimensions it increases)
- For even larger matrices we assume that the inter-regional fat cluster will be the best because it has 32 cores.

The worst solution, for the tested matrices would seem the inter-regional light cluster (but it would be necessary to see very large matrices)

Matrix inversion

As regards the inversion, we carry out a similar analysis. Version v1 and version v3 are compared. As expected, the v1 version is worse in terms of performance, it is more noticeable in the inter-regional both at and light cluster. In general, v3 is better than v1.

Therefore, looking for the best solution we can summarize that:
For 1000x1000 matrices, the 3 light cluster intra-regional version is the best (it must be considered that there are 8 machines with 16 cores, so we are not over-parallelizing excessively, as happens in other cases).
The worst solution, on the other hand, is the inter-regional 1 light cluster version.
In general, inter-regional clusters, for the tests carried out it are the worst solution.

In conclusion, we implemented serial matrix and inversion product algorithms, parallelized and performed a performance and speed up analysis obtaining results that reflect the values studied a priori quite well.

References

[1] Slide and notions of Numerical Methods course, professor Sangalli
[2] https://en.wikipedia.org/wiki/LU_decomposition
[3] https://matrixcalc.org/it/