Project

Jun Zhang, junzh@kth.se

Stockholm

DD2356
Methods in High-Performance Computing
Kungliga Tekniska Högskolan

Step1

Gaussian Elimination

There are two kinds of solution methods for linear system, direct solution methods and iterative solution method. Gaussian elimination methods is one of the direct solution methods for solving systems of linear equations of the form Ax=b.

Gaussian Elimination Pseudocode:

```
//forward elimination
For k in 0..n-1 do
    Find pivot row r such that |$a_{rk}$| is maximum for k <= r <= n
    If( k != r )
        Exchange row k and row r of matrix A
    For each row i, i in (k+1)..(n-1) do
        Calculate elimination factor $l_i = a_{ik}/a_{kk}$
        For each element j in this row, j in (k+1)..(n-1)
            $a_{ij}=a_{ij} – l_i * a_{kj}$
        $b_i = b_i – l_i * b_k$
//backward substitution
for k in n-1 .. 0 do
    calculate $x_k$
```

The result of the forward elimination phase will be a upper triangular matrix (Figure 1) so that the in the second phase, each x_k can be calculated according to $x_k = \frac{1}{a_{kk}} \left(b_k - \sum_{j=k+1}^{n} a_{kj}x_j\right)$

$$A^{(k)} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1,k-1} & a_{1k} & \cdots & a_{1n} \\ 0 & a_{22}^{(2)} & \cdots & a_{2,k-1}^{(2)} & a_{2k}^{(2)} & \cdots & a_{2n}^{(2)} \\ \vdots & \ddots & \ddots & \vdots & \vdots & & \vdots \\ \vdots & & \ddots & a_{k-1,k-1}^{(k-1)} & a_{k-1,k}^{(k-1)} & \cdots & a_{k-1,n}^{(k-1)} \\ \vdots & & & 0 & a_{kk}^{(k)} & \cdots & a_{kn}^{(k)} \\ \vdots & & & \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & \cdots & 0 & a_{nk}^{(k)} & \cdots & a_{nn}^{(k)} \end{bmatrix}.$$

Figure 1

Step2

For the row-cyclic data distribution, I followed the algorithm described in the book, section 8.1.2. Each row i handles the rows which with row number mod N, the number of processors, equals to i.

For example, if N=8, which means A is a 8x8 matrix, and N is 4, then rank 0 handles row 0 and 4, rank 1 for row 1 and 5, rank 2 for row 2 and 6, and rank 3 for row 3 and7.
In the forward elimination phase, each processor calculates its own pivot row r, and shares with all the others by MPI_Allreduce so that all processors know the same r. If exchanging the rows are needed, the program will exchange it in 2 different ways depends on whether row k and r belongs to the same process. If yes, then this process exchanges this row locally, otherwise, there will be a Send-Recv communication, the processor sends row r to the processor handling row k. Now, the processor for row k have the pivot row's value and broadcast it so that all processors could do the elimination.
In the backward phase, each process has to broadcast the xi after calculating so that other processors could use this xi to calculate x_{i-1}

Functions for step 2-6

The files in library folder are for all programs
helper: contains the functions for reading, writing matrixes and handling the optional options in command line.

```
double *input_matrix(const char* fname, int *n);
bool output_matrix(const char* fname, const double* x, int n);
bool handle_opt(int argc, char *argv[], char **fin, char **fout, int *p1, int *p2);
```

mmio: it's the support files for matrix from http://math.nist.gov/MatrixMarket/

Functions for step 2

```
int max_col_loc(double *a, int k, int p,int me, int loc_n);
void exchange_row(double *a, double *b, int r, int k);
void copy_row(double  *a, double *b, int k, double *buf);
void copy_exchange_row(double *a, double *b, int r, double *buf, int k);
void copy_back_row(double *a, double *b, double *buf, int k);
double *gauss_cyclic(double *a, double *b, int p, int me);
```

Step3

For the row-cyclic data distribution, I followed the algorithm described in the book, section 8.1.3. In my implementation, the block size b1 and b2 are set to 1 by default, and p1 p2 are the numbers of processors along the columns and rows respectively, following the definition in the book.
In this algorithm, each processor belongs to a row group and a column group, decided by its rank, the number of p2 and p1, and the number of b1 and b2.
In the forward phase, the processors in column group Co(k) determines the r and get the biggest pivot among this group with MPI_Reduce. The group leader then broadcast the pivot to all processors.

To exchange row k and r, there are two strategies. The first one is for k and r belong to the same row group Ro(k), when Ro(k)=Ro(r). In this case, each processor in this Ro(k) just exchanges the part of row k and r belongs to it. On the other hand, when they belongs to different row groups, each processor in Ro(k) sends its part of row to the processor in Ro(r) for the same part of rows along the column.
After the exchanging, each processor in Ro(r) has part of the pivot row and broadcasts it among its column group so that each processor know part of the pivot row that's necessary for them.
The calculating of the eliminate factor is also different from step as it's need to be broadcasted among each row groups.
In the backward substitution phase, the sum $\left(\sum_{j=k+1}^{n} a_{kj}x_j\right)$ is also needed to be gathered by MPI_Allreduce, and the x_i will be broadcasted to all processors as well.

Functions for step 3

```
int max_col_loc(double *a, int k, int p1, int p2,int me);
copy_exchange_row(double *a, double *b, int r, double *buf, int k);
void copy_back_row(double *a, double *b, double *buf, int k);
bool isInCoK(int k, int p1, int p2, int t);
bool isInRor(int r, int p1, int p2, int t);
int *Co(int k, int p1, int p2);
int *Ro(int r, int p1, int p2);
int *Cop(int q, int p1, int p2);
int *Rop(int q, int p1, int p2);
bool isInRop(int me, int q, int p1);
bool isInCop(int me, int q, int p1);
int grp_leader(int k, int p1, int p2 );
void exchange_row_loc(double *a, double *b, int r, int k, int p1, int p2, int me);
void copy_row_loc(double *a, double *b, int k, double *buf, int me, int p1);
int compute_partner(int *ro, int me, int p1);
int compute_size(int k);
void exchange_row_buf(double *a, double *b, int r, double *buf, int k, int me, int psz);
double *compute_elim_fact_loc(double *a, double *b, int k, double *buf, int p1, int p2, int me);
void compute_local_entries(double *a, double *b, int k, double *elim_buf, double *buf, int me,
int p1, int p2);
double * gauss_double_cyclic(double *a, double *b, int p1, int p2, int p, int me);
```

As the different definition of p1, p2, the gauss_double_cyclic function will be called by

```
double *x=gauss_double_cyclic(A, b, p2, p1, p, myrank);
```

Step4

Results from step 2

In step 2, the time complexity will be $O(N^2/p)$ regardless of the communication time, whilst the cost for communication will be O(N/p) for all processors. (p is the number of processors, and N is the column and row number for A).

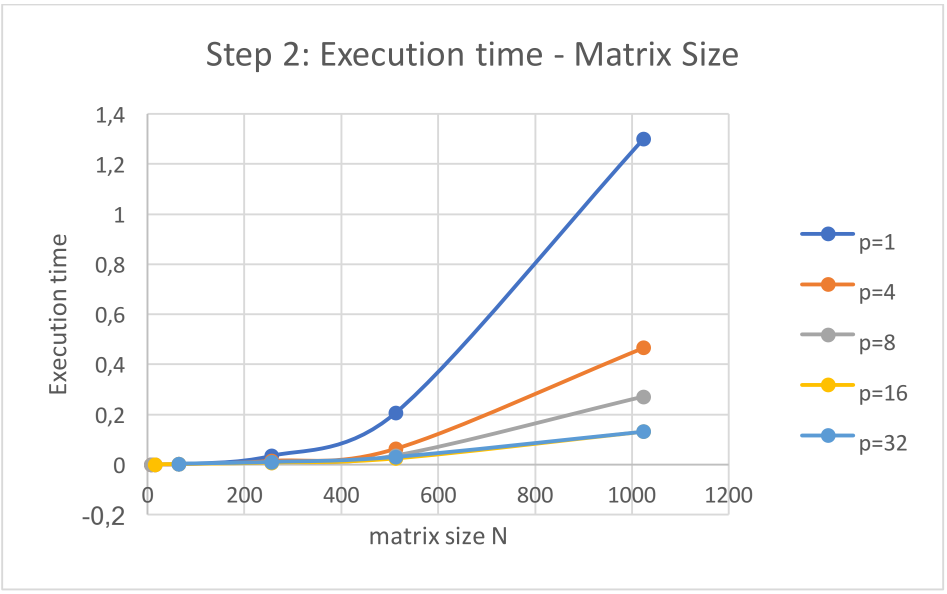

Figure 2

As shown in Figure 2, the executing time increases exponentially when p=1 with N increase, which means it goes as the traditional Gaussian elimination algorithm with time complexity O(n^3). And the execution time decreases with the p increasing.

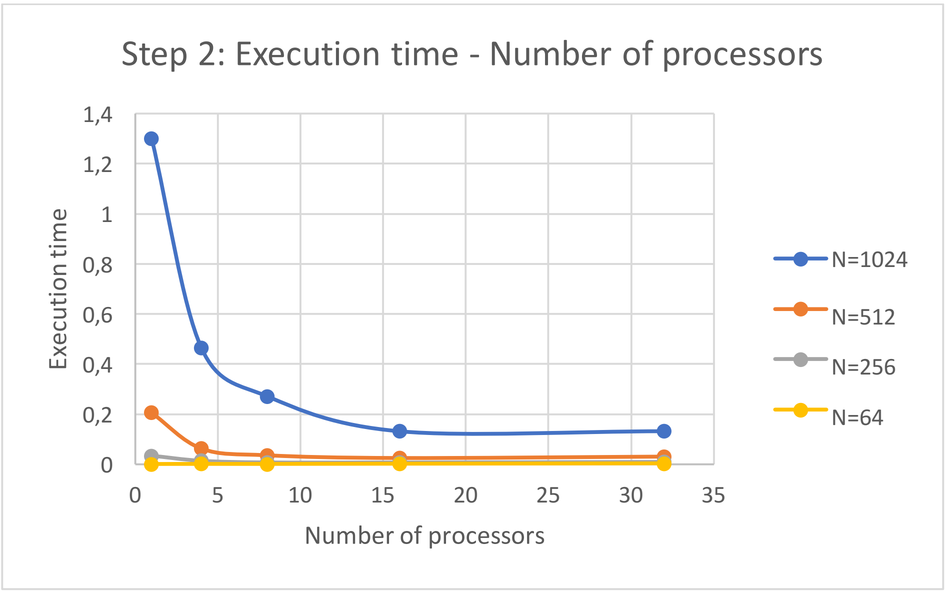

Figure 3

In Figure 3, the execution time decreases with the increase of processors, and the speedup = $T/T_{n=1}$ is calculated and plotted in Figure 4. When the matrix is larger than 64x64, the speedup will be larger than 1 which means it's faster to run parallel with this row-cyclic Gaussian elimination. The speedup rates will increase with the number of processors increasing when that's small (less than 16 nodes in this case), then have a slight drop.

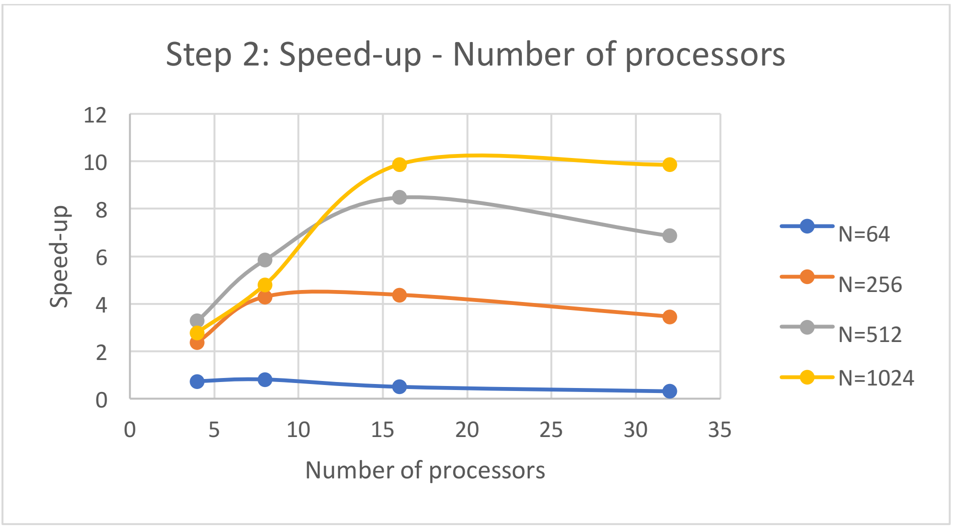

Figure 4

However, the efficiency decreases greatly for all matrixes, but a larger number of N will leads to a better efficiency compared to that of other matrix sizes, as shown in Figure 5.

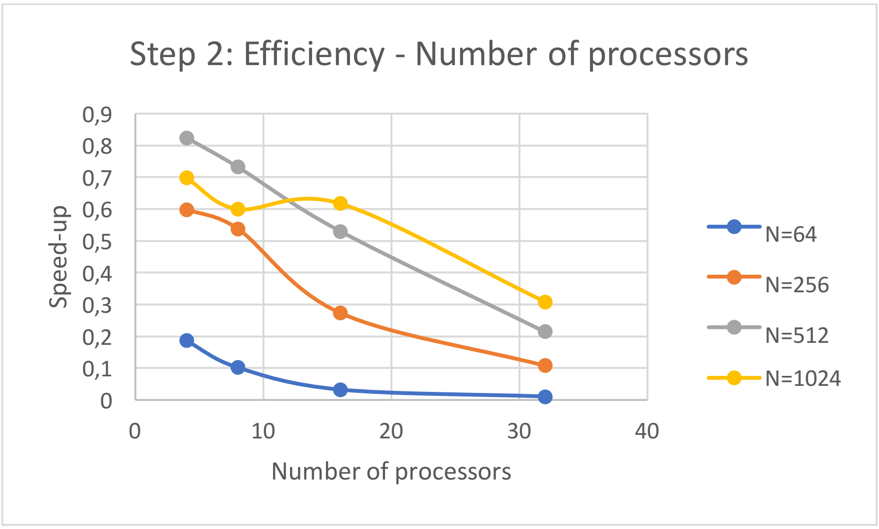

Figure 5

Results from step 3

The results in Figure 6, 7, 8, 9 are tested with processors distributions as 1x1, 2x2, 4x2, 4x4, 8x4 for p=1, 4, 8, 16, 32, and the effects of different distribution will be discussed later.

As shown in Figure 6, the executing time increases exponentially when p=1 with N increase, which means it goes as the traditional Gaussian elimination algorithm with time complexity O(n^3). And the execution time decreases with the p increasing but with a slight increase, in Figure 7. The time complexity is O($N^2/\sqrt{p1 * p2}$), whilst there is much more communication needed.

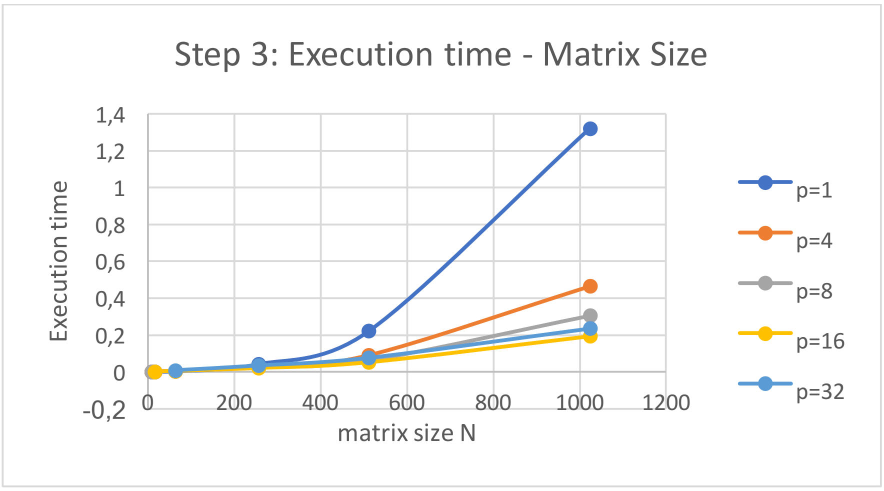

Figure 6

The Figure 8 shows the speedup for the cases with different size of matrix for different number of processors. The speedup rate is larger than 1 for large N cases (larger than 64 in my test). It goes up then down with the increase of the processors' number.

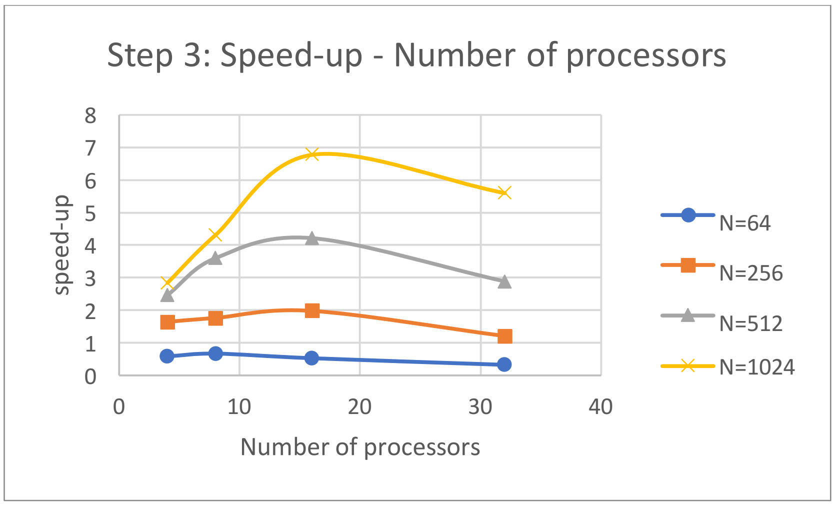

Figure 7

The Efficiency decreases continually with the increasing of processors' number and the efficiency for a larger matrix will be higher than that of a smaller matrix for same a same number of processors, as in Figure 9.

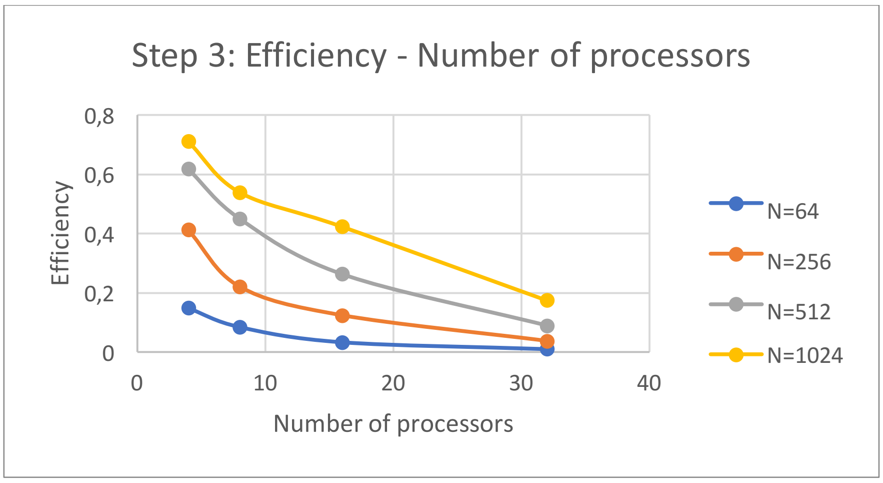

Figure 8

Figure 10 plots the execution time for different processor's distributions for 32 processors, namely 2x16, 4x8, 8x4, 16x2, for different size of matrixes. The results of 8x4 and 16x2 are so close that it's not quite clear in the plot. It shows the 2x16 takes longest time whilst a higher number of p1 value (along rows) results in a better value.

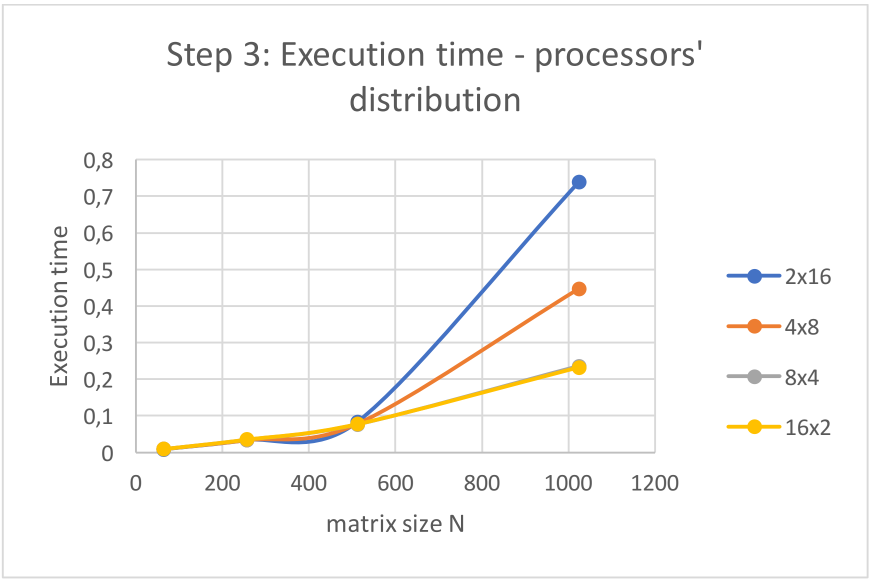

Figure 9

Step5

Results from step 5

The execution time as shown in Figure 10 decreases, increases, and then decreases again with the increase of threads as 1, 4, 8, 16, 24.

Figure 11 presents the speed up with the increase of the number of threads, and there's a valley as in Figure 10. However, except that lowest points, the speed up for N larger than 64 (larger or equal to 256) is larger than 1, and the larger of the matrix size, the higher speed up ratio.

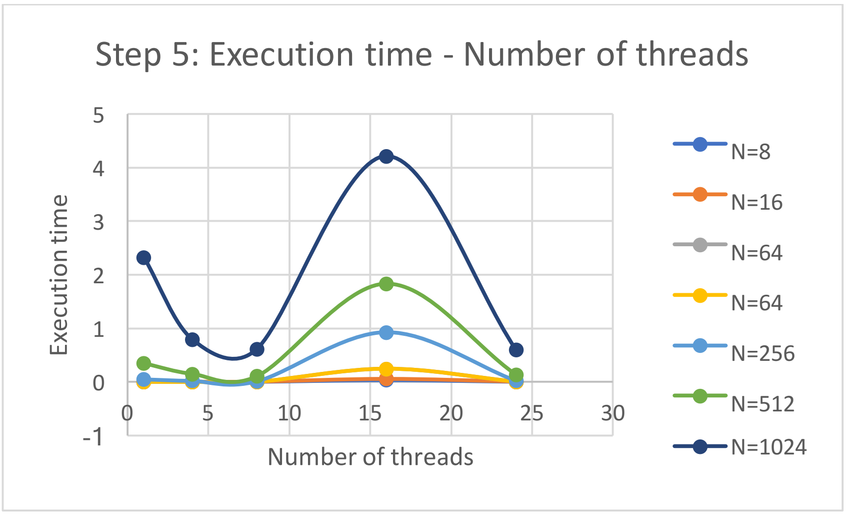

Figure 10

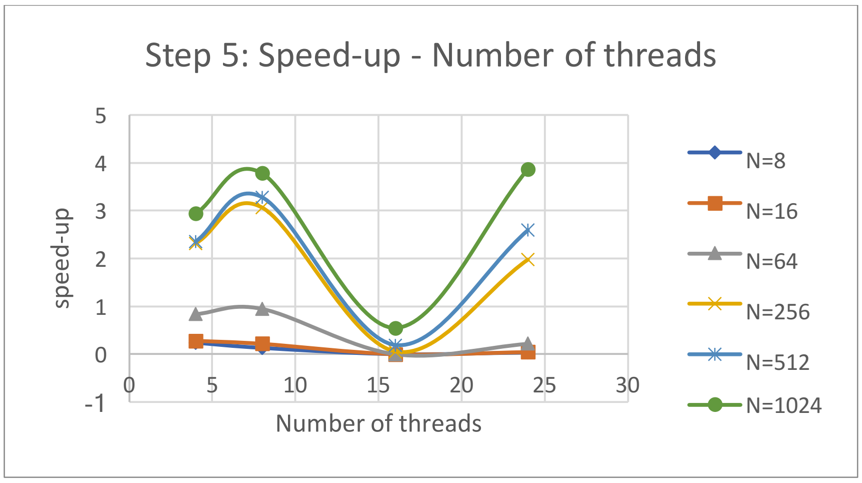

Figure 11

Step6

The time complexity will be $O(N^2/p)$ regardless of the communication time. The executions time regarding to the matrix size and the number of processors are in Figure 12 and Figure 13.

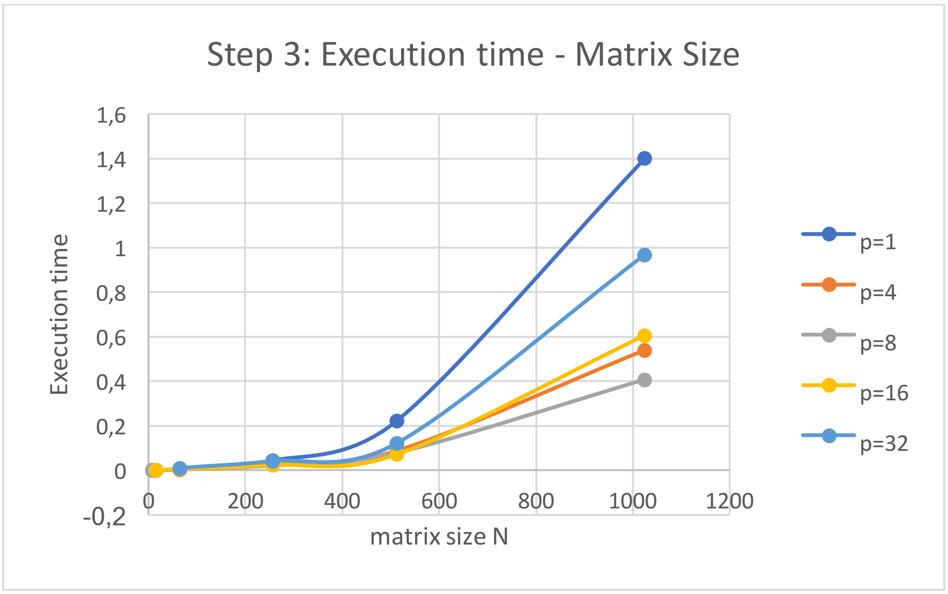

Figure 12

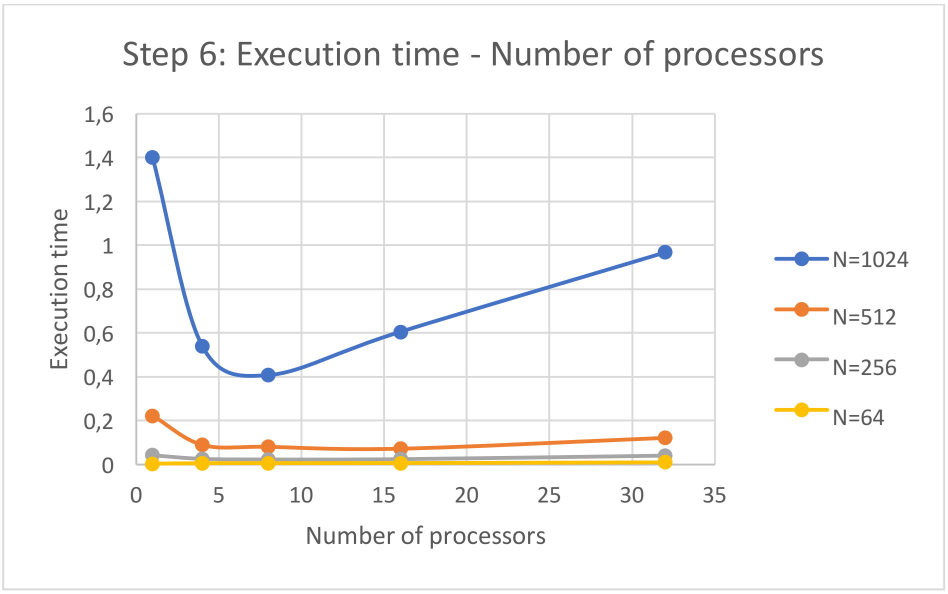

Figure 13