A Parallel Algorithm based on Monte Carlo for Computing the Inverse and other Functions of a Large Sparse Matrix

Patrícia Isabel Duarte Santos
patricia.d.santos@tecnico.ulisboa.pt
Instituto Superior Técnico

***Abstract*—Nowadays, matrix inversion plays an important role in several areas, for instance, when we analyze specific characteristics of a complex network such as *node centrality* and *communicability*. In order to avoid the explicit computation of the inverse matrix, or other matrix functions, which is costly, there are several high computational methods to solve linear systems of algebraic equations that obtain the inverse matrix and other matrix functions. However, these methods, whether direct or iterative, have a high computational cost when the size of the matrix increases. In this context, we present an algorithm based on Monte Carlo methods as an alternative to obtain the inverse matrix and other functions of a large-scale sparse matrix. The main advantage of this algorithm is the possibility of obtaining the matrix function for only one row of the result matrix, avoiding the instantiation of the entire result matrix. Additionally, this solution is parallelized using OpenMP. Among the developed parallelized versions, a scalable version was developed, for the tested matrices, which uses the directive *omp declare reduction*.**

***Keywords*—Monte Carlo methods, OpenMP, parallel algorithm, matrix functions, complex networks**

I. Introduction

A. Motivation

Matrix inversion is an important matrix operation that is widely used in several areas such as financial calculation, electrical simulation, cryptography and complex networks.

One area of application of this work is in complex networks. These can be represented by a graph (*e.g.,* the Internet, social networks, transport networks, neural networks, etc.), and a graph is usually represented by a matrix. In complex networks there are many features that can be studied, such as the node importance in a given network, *node centrality*, and the *communicability* between a pair of nodes that measures how well two nodes can exchange information with each other. These metrics are important when we want to the study of the topology of a complex network.

There are several algorithms over matrices that allow us to extract important features of these systems. However, there are some properties which require the use of the inverse matrix, or other matrix functions, which is impractical to calculate for large matrices. Existing methods, whether direct or iterative, have a costly approach in terms of computational effort and memory needed for such problems. Therefore, Monte Carlo methods represent a viable alternative approach to this problem since they can be easily parallelized in order to obtain a good performance.

B. Objectives

The main goal of this work, considering what was stated in the previous section, is to develop a parallel algorithm based on Monte Carlo for computing the inverse and other matrix functions of large sparse matrices in an efficient way, *i.e.,* with a good performance.

With this in mind, our objectives are:

- To implement an algorithm proposed by J. Von Neumann and S. M Ulam [1] that makes it possible to obtain the inverse matrix and other matrix functions based on Monte Carlo methods;
- To develop and implement a modified algorithm based on the item above that has its foundation on the Monte Carlo methods;
- To demonstrate that this new approach improves the performance of matrix inversion when compared to existing algorithms;
- To implement a parallel version of the new algorithm using OpenMP.

C. Contributions

The main contributions of our work include:

- The implementation of a modified algorithm based on the Monte Carlo methods to obtain the inverse matrix and other matrix functions;
- The parallelization of the modified algorithm when we want to obtain the matrix function over the entire matrix, using OpenMP; Two versions of the parallelization of the algorithm when we want to obtain the matrix function for only one row of the matrix: one using *omp atomic*, and another one using *omp declare reduction*;
- A scalable parallelized version of the algorithm, using *omp declare reduction*, for the tested matrices.

All the implementations stated above were successfully executed, with special attention to the version that calculates the matrix function for a single row of the matrix, using *omp declare reduction*, which is scalable and capable of reducing the computational effort compared with other existing methods, at least the synthetic matrices tested. This is due to the fact that instead of requiring the calculation of the matrix function over the entire matrix it calculates the matrix function for only one row of the matrix. It has a direct application, for example, when a study of the topology of a complex network is required, being able to effectively retrieve the node importance of a node in a given network, *node centrality*, and the *communicability* between a pair of nodes.

II. Background and Related Work

A. Application Areas

Nowadays, there are many areas where efficient matrix functions, such as the matrix inversion, are required. For example, in image reconstruction applied to computed tomography [2] and astrophysics [3], and in bioinformatics to solve the problem of protein structure prediction [4]. This work will mainly focus on complex networks, but it can easily be applied to other application areas.

A Complex Network [5] is a graph (network) with very large dimension. So, a Complex Network is a graph with non-trivial topological features that represents a model of a real system. These real systems can be, for example: the Internet and the World Wide Web; Biological systems; Chemical systems; and Neural networks.

A graph $G = (V, E)$ is composed by a set of nodes (vertices) V and edges (links) E represented by unordered pairs of vertices. Every network is naturally associated with a graph $G = (V, E)$ where V is the set of nodes in the network and E is the collection of connections between nodes, that is $E = \{(i, j)|$ there is an edge between node i and node j in $G\}$.

One of the hardest and most important tasks in the study of the topology of such complex networks is to determine the node importance in a given network, and this concept may change from application to application. This measure is normally referred to as *node centrality* [5]. Regarding the *node centrality* and the use of matrix functions, Kylmko *et al.* [5] show that the matrix resolvent plays an important role. The resolvent of an $n \times n$ matrix A is defined as:

$$R(\alpha) = (I - \alpha A)^{-1} \tag{1}$$

where I is the identity matrix and $\alpha \in \mathbb{C}$ excluding the eigenvalues of A (that satisfy $det(I - \alpha A) = 0$), and $0 < \alpha < \frac{1}{\lambda_1}$, where λ_1 is the maximum eigenvalue of A. The entries of the matrix resolvent count the number of walks in the network, penalizing longer walks. This can be seen by considering the power series expansion of $(I-\alpha A)^{-1}$:

$$(I-\alpha A)^{-1} = I + \alpha A + \alpha^2 A^2 + \cdots + \alpha^k A^k + \cdots = \sum_{k=0}^{\infty} \alpha^k A^k \quad (2)$$

Here, $[(I-\alpha A)^{-1}]_{ij}$, counts the total number of walks from node i to node j, weighting walks of length k by α^k. The bounds on α ($0 < \alpha < \frac{1}{\lambda_1}$) ensure that the matrix $I - \alpha A$ is invertible and the power series in (2) converges to its inverse.

Another property that is important when we are studying a complex network is the *communicability* between a pair of nodes i and j. This measures how well two nodes can exchange information with each other. According to Kylmko *et al.* [5], this can be obtained using the matrix exponential function [6] of a matrix A defined by the following infinite series:

$$e^A = I + A + \frac{A^2}{2!} + \frac{A^3}{3!} + \cdots = \sum_{k=0}^{\infty} \frac{A^k}{k!} \quad (3)$$

with I being the identity matrix and with the convention that $A^0 = I$. In other words, the entries of the matrix, $[e^A]_{ij}$, counts the total number of walks from node i to node j, penalizing longer walks by scaling walks of length k by the factor $\frac{1}{k!}$.

As a result, the development and implementation of efficient matrix functions is an area of great interest since complex networks are becoming more and more relevant.

B. The Monte Carlo Methods Applied to Matrix Inversion

The result of the application of these statistical sampling methods depends on how an infinite sum of finite sums is done. An example of such methods is random walk, a Markov Chain Monte Carlo algorithm, which consists in the series of random samples that represents a random walk through the possible configurations. This fact leads to a variety of Monte Carlo estimators.

The algorithm implemented in this thesis is based on a classic paper that describes a Monte Carlo method of inverting a class of matrices, devised by J. Von Neumann and S. M Ulam [1]. This method can be used to invert a class of n-th order matrices, but it is capable of obtaining a single element of the inverse matrix without determining the rest of the matrix. To better understand how this method works we present a concrete example and all the necessary steps involved.

Firstly, there are some restrictions that, if satisfied, guarantee that the method produces a correct solution. Let us consider as an example the $n \times n$ matrix A and B in Fig. 1. The restrictions are:

- Let B be a matrix of order n whose inverse is desired, and let $A = I - B$, where I is the identity matrix.
- For any matrix M, let $\lambda_r(M)$ denote the r-th eigenvalue of M, and let m_{ij} denote the element of M in the i-th row and j-th column. The method requires that

$$\max_r |1 - \lambda_r(B)| = \max_r |\lambda_r(A)| < 1. \quad (4)$$

When (4) holds, it is known that

$$(B^{-1})_{ij} = ([I-A]^{-1})_{ij} = \sum_{k=0}^{\infty} (A^k)_{ij}. \quad (5)$$

- All elements of matrix A ($1 \leq i,j \leq n$) have to be positive, $a_{ij} \geq 0$, let us define $p_{ij} \geq 0$ and v_{ij} the corresponding "value factors", that satisfy the following:

$$p_{ij} v_{ij} = a_{ij}; \quad (6)$$

$$\sum_{j=1}^{n} p_{ij} < 1. \quad (7)$$

In the example considered, we can see that all this is verified in Fig. 2 and Fig. 3, except the sum of the second row of matrix A that is not inferior to 1, *i.e.,* $a_{21} + a_{22} + a_{23} = 0.4 + 0.6 + 0.2 = 1.2 \geq 1$ (see Fig. 1). In order to guarantee that the sum of the second row is inferior to 1, we divide all the elements of the second row by the total sum of that row plus some normalization constant (let us assume 0.8) so the value will be 2 and therefore the second row of V will be filled with 2 (Fig. 2).

- In order to define a probability matrix given by p_{ij}, an extra column in the initial matrix A should be added. This corresponds to the "stop probabilities" and are defined by the relations (see Fig. 3):

$$p_i = 1 - \sum_{j=1}^{n} p_{ij} \quad (8)$$

Secondly, once all the restrictions are met, the method proceeds in the same way to calculate each element of the inverse matrix. So, we are only going to explain how it works to calculate one element of the inverse matrix, that is the element $(B^{-1})_{11}$. As stated in [1], the Monte Carlo method to compute $(B^{-1})_{ij}$ is to play a solitaire game whose expected payment is $(B^{-1})_{ij}$, and according to a result by Kolmogoroff [7] on the strong law of numbers, if one plays such a game repeatedly, the average payment for N successive plays will converge to $(B^{-1})_{ij}$ as $N \rightarrow \infty$, for almost all sequences of plays. Taking all this into account, to calculate one element of the inverse matrix we will need N plays, with N sufficiently large for an accurate solution. Each play has its own gain, *i.e.,* its contribution to the final result, and the gain of one play is given by

$$GainOfPlay = v_{i_0 i_1} \times v_{i_1 i_2} \times \cdots \times v_{i_{k-1} j} \quad (9)$$

considering a route $i = i_0 \rightarrow i_1 \rightarrow i_2 \rightarrow \cdots \rightarrow i_{k-1} \rightarrow j$.

Finally, assuming N plays, the total gain from all the plays is given by the following expression

$$TotalGain = \frac{\sum_{k=1}^{N} (GainOfPlay)_k}{N \times p_j} \quad (10)$$

which coincides with the expectation value in the limit $N \rightarrow \infty$, being therefore $(B^{-1})_{ij}$.

To calculate $(B^{-1})_{11}$, one play of the game is explained with an example in the following steps, and knowing that the initial gain is equal to 1:

1) Since the position we want to calculate is in the first row, the algorithm starts in the first row of matrix A (see Fig. 4). Then, it is necessary to generate a random number uniformly between 0 and 1. Once we have the random number, let us consider 0.28, we need to know to which drawn position of matrix A it corresponds. To see what position we have drawn, we have to start with the value of the first position of the current row, a_{11} and compare it with the random number. The search only stops when the random number is inferior to the value. In this case $0.28 > 0.2$, so we have to continue accumulating the values of the visited positions in the current row. Now, we are in position a_{12} and we see that $0.28 < a_{11} + a_{12} = 0.2 + 0.2 = 0.4$, so the position a_{12} has been drawn (see Fig. 5) and we have to jump to the second row and execute the same operation. Finally, the gain of this random play is the initial gain multiplied by the value of the matrix with "value factors" correspondent with the position of a_{12}, which in this case is 1, as we can see in Fig. 2.

2) In the second random play, we are in the second row and a new random number is generated, let us assume 0.1, which corresponds to the drawn position a_{21} (see Fig. 6). Doing the same reasoning we have to jump to the first row. The gain at this point is equal to multiplying the existent value of gain by the value of the matrix with "value factors" correspondent with the position of a_{21}, which in this case is 2, as we can see in Fig. 2.
3) In the third random play, we are in the first row and generating a new random number, let us assume 0.6 which corresponds to the "stop probability" (see Fig. 7). The drawing of the "stop probability" has two particular properties considering the gain of the play, that follow:
   - If the "stop probability" is drawn in the first random play, the gain is 1;
   - In the remaining random plays, the "stop probability" gain is 0 (if $i \neq j$) or p_j^{-1} (if $i = j$), *i.e.,* the inverse of the "stop probability" value from the row in which the position we want to calculate is.

$$\begin{array}{ccc} B & A & B^{-1} = (I-A)^{-1} \\ \begin{bmatrix} 0.8 & -0.2 & -0.1 \\ -0.4 & 0.4 & -0.2 \\ 0 & -0.1 & 0.7 \end{bmatrix} & \begin{bmatrix} 0.2 & 0.2 & 0.1 \\ 0.4 & 0.6 & 0.2 \\ 0 & 0.1 & 0.3 \end{bmatrix} \xrightarrow[\text{results}]{\text{theoretical}} & \begin{bmatrix} 1.7568 & 1.0135 & 0.5405 \\ 1.8919 & 3.7838 & 1.3514 \\ 0.2703 & 0.5405 & 1.6216 \end{bmatrix} \end{array}$$

Fig. 1. Example of a matrix $B = I - A$ and A, and the theoretical result $B^{-1} = (I - A)^{-1}$ of the application of this Monte Carlo method.

$$V \qquad \begin{bmatrix} 1.0 & 1.0 & 1.0 \\ 2.0 & 2.0 & 2.0 \\ 1.0 & 1.0 & 1.0 \end{bmatrix}$$

Fig. 2. Matrix with "value factors" v_{ij} for the given example.

$$A \qquad \begin{bmatrix} 0.2 & 0.2 & 0.1 & \mathbf{0.5} \\ 0.2 & 0.3 & 0.1 & \mathbf{0.4} \\ 0 & 0.1 & 0.3 & \mathbf{0.6} \end{bmatrix}$$

Fig. 3. Example of "stop probabilities" calculation (bold column).

Thus, in this example, we see that the "stop probability" is not drawn in the first random play, but it is situated in the same row as the position we want to calculate the inverse matrix value, so the gain of this play is $GainOfPlay = v_{12} \times v_{21} = 1 \times 2$. To obtain an accurate result N plays are needed, with N sufficiently large, and the $TotalGain$ is given by Equation 10.

Although the method explained in the previous paragraphs is expected to rapidly converge, it can be inefficient due to having many plays where the gain is 0. Our solution will take this in consideration in order to reduce waste.

There are other Monte Carlo algorithms that aim to enhance the performance of solving linear algebra problems [8], [9], [10]. These algorithms are similar to the one explained above in this section and it is shown that, when some parallelization techniques are applied, the obtained results have a great potential. One of these methods [9] is used as a pre-conditioner, as a consequence of the costly approach of direct and iterative methods, and it has been proved that the Monte Carlo methods present better results than the former classic methods. Consequently, our solution will exploit these parallelization techniques, explained in the next subsections, to improve our method.

III. Algorithm Implementation

A. General Approach

The algorithm we propose is based on the algorithm presented in Section II-B. For this reason all the assumptions are the same, except that our algorithm does not have the extra column corresponding to the "stop probabilities" and the matrix with "value factors" v_{ij} is in this case a vector v_i, where all values are the same for the same row.

random number = 0.28

$$A \qquad \begin{bmatrix} 0.2 & \mathbf{0.2} & 0.1 & 0.5 \\ 0.2 & 0.3 & 0.1 & 0.4 \\ 0 & 0.1 & 0.3 & 0.6 \end{bmatrix}$$

Fig. 4. First random play of the method.

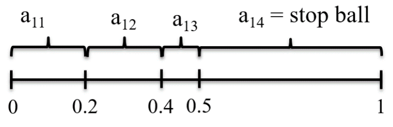

Fig. 5. Situating all elements of the first row given its probabilities.

random number = 0.1

$$A \qquad \begin{bmatrix} 0.2 & 0.2 & 0.1 & 0.5 \\ \mathbf{0.2} & 0.3 & 0.1 & 0.4 \\ 0 & 0.1 & 0.3 & 0.6 \end{bmatrix}$$

Fig. 6. Second random play of the method.

random number = 0.6

$$A \qquad \begin{bmatrix} 0.2 & 0.2 & 0.1 & \mathbf{0.5} \\ 0.2 & 0.3 & 0.1 & 0.4 \\ 0 & 0.1 & 0.3 & 0.6 \end{bmatrix}$$

Fig. 7. Third random play of the method.

This new approach aims to reuse every single play, *i.e.,* the gain of each play is never zero, and it is also possible to control the number of plays. It can be used as well to compute more general functions of a matrix.

Coming back to the example of Section II-B, the algorithm starts by assuring that the sum of all the elements of each row is equal to 1. So, if the sum of the row is different from 1, each element of one row is divided by the sum of all elements of that row, and the vector v_i will contain the values, "value factors", used to normalized the rows of the matrix. This process is illustrated in Fig. 1, Fig. 8 and Fig. 9.

Then, once we have the matrix written in the required form, the algorithm can be applied. The algorithm, as we can see in Fig. 10, has four main loops. The first loop defines the row that is being computed. The second loop defines the number of iterations, *i.e.,* random jumps inside the probability matrix, and this relates to the power of the matrix in the corresponding series expansion. Then, for each number of iterations, N plays, *i.e.,* the sample size of the Monte Carlo method, are executed for a given row. Finally, the remaining loop generates this random play with the number of random jumps given by the number of iterations. In order to better understand the algorithms' behavior, two examples will be given:

1) In the case where we have one iteration, one possible play for that is the example of Fig. 11. That follows the same reasoning as the algorithm presented in Section II-B, except for the matrix element where the gain is stored, *i.e.,* in which position of the inverse matrix the gain is accumulated. This depends on

$$\begin{array}{cc} A & A \\ \begin{bmatrix} 0.2 & 0.2 & 0.1 \\ 0.4 & 0.6 & 0.2 \\ 0 & 0.1 & 0.3 \end{bmatrix} \xrightarrow[\text{normalization}]{} & \begin{bmatrix} 0.4 & 0.4 & 0.2 \\ 0.33 & 0.5 & 0.17 \\ 0 & 0.25 & 0.75 \end{bmatrix} \end{array}$$

Fig. 8. Initial matrix A and respective normalization.

$$V \qquad \begin{bmatrix} 0.5 \\ 1.2 \\ 0.4 \end{bmatrix}$$

Fig. 9. Vector with "value factors" v_i for the given example.

```
for(i = 0 ; i < rowSize; i++)
{
    for(q = 0; q < NUM_ITERATIONS; q++)
    {
        for(k = 0; k < NUM_PLAYS; k++)
        {
            currentRow = i;
            vP = 1;
            for(p = 0; p < q; p++)
            {
```

Fig. 10. Code excerpt in C with the main loops of the proposed algorithm.

random number = 0.6

$$A$$

$$\begin{bmatrix} 0.4 & \mathbf{0.4} & 0.2 \\ 0.33 & 0.5 & 0.17 \\ 0 & 0.25 & 0.75 \end{bmatrix}$$

Fig. 11. Example of one play with one iteration.

the column where the last iteration stops and what is the row where it starts (first loop). The gain is accumulated in a position corresponding to the row in which it started and the column in which it finished. Let us assume that it started in row 3 and ended in column 1, the element to which the gain is added would be $(B^{-1})_{31}$. In this particular instance, it stops in the second column while it started in the first row, so the gain will be added in the element $(B^{-1})_{12}$.

2) When we have two iterations, one possible play for that is the example of Fig. 12 for the first iteration, and Fig. 13 for the second iteration. In this case, it stops in the third column and it started in the first row, so the gain will count for the position $(B^{-1})_{13}$ of the inverse matrix.

Finally, after the algorithm computes all the plays for each number of iterations, if we want to obtain the inverse matrix for only one row of the matrix, we must retrieve the total gain for each position. This process consists in the sum of all the gains for each number of iterations divided by the N plays, as we can see in Fig. 14.

The proposed algorithm was implemented in C, since it is a good programming language to manipulate the memory usage, and it provides language constructs that efficiently map machine instructions as well. One other reason is the fact that it is compatible/adaptable with OpenMP. Therefore, we used OpenMP since it is the simpler and easier way to transform a serial program into a parallel program.

B. Implementation of the Different Matrix Functions

The algorithm we propose, depending on how we aggregate the output results, is capable of obtaining different matrix functions as a result. In this thesis, we are interested in obtaining the inverse matrix

random number = 0.7

$$A$$

$$\begin{bmatrix} 0.4 & \mathbf{0.4} & 0.2 \\ 0.33 & 0.5 & 0.17 \\ 0 & 0.25 & 0.75 \end{bmatrix}$$

Fig. 12. Example of the first iteration of one play with two iterations.

random number = 0.85

$$A$$

$$\begin{bmatrix} 0.4 & 0.4 & 0.2 \\ 0.33 & 0.5 & \mathbf{0.17} \\ 0 & 0.25 & 0.75 \end{bmatrix}$$

Fig. 13. Example of the second iteration of one play with two iterations.

```
for(j = 0 ; j < columnSize; j++)
{
    for (q = 0 ; q < NUM_ITERATIONS; q ++)
    {
        inverse[j] += aux[q][j];
    }
}
for(j = 0 ; j < columnSize; j++)
{
    inverse[j] = inverse[j]/(NUM_PLAYS);
}
```

Fig. 14. Code excerpt in C with the necessary operations to obtain the inverse matrix of one single row.

```
for(j = 0 ; j < columnSize; j++)
{
    for (q = 0 ; q < NUM_ITERATIONS; q ++)
    {
        exponential[j] += aux[q][j]/factorial(q);
    }
}
for(j = 0 ; j < columnSize; j++)
{
    exponential[j] = exponential[j]/(NUM_PLAYS);
}
```

Fig. 15. Code excerpt in C with the necessary operations to obtain the matrix exponential of one single row.

and the matrix exponential, since these functions give us important complex networks metrics: *node centrality* and *node communicability*, respectively (see Section II-A). In Fig. 14, we can see how we obtain the inverse matrix of one single row, according to Equation 2. And in Fig. 15 we can observe how we obtain the matrix exponential taking into account Equation 3. If we iterate this process for a number of times equivalent to the number of lines (1^{st} dimension of the matrix), we get the results for the full matrix.

C. Matrix Format Representation

The matrices that are going to be studied in this thesis are sparse, so instead of storing the full matrix, $n \times n$, it is desirable to find a solution that uses less memory and at the same time does not compromise the performance of the algorithm.

There is a great variety of formats to store sparse matrices, such as the Coordinate Storage format, the Compressed Sparse Row (CSR) format, the Compressed Diagonal Storage (CDS) format, and the Modified Sparse Row (MSR) format [11], [12], [13]. Since this algorithm processes row by row, a format where each row can be easily accessed, knowing there it starts and ends, is needed. After analyzing the existing formats, we decided to use the Compressed Sparse Row (CSR) format, since this format is the most efficient when we are dealing with row-oriented algorithms. Additionally, the CDS and MSR formats are not suitable in this case, since they store the nonzero elements per subdiagonals in consecutive locations. The CSR format is going to be explained in detail in the following paragraph.

The CSR format is a row-oriented operations format that only stores the nonzero elements of a matrix. This format requires 3 vectors:

- One vector that stores the values of the nonzero elements - *val* with length nnz (nonzero elements);
- One vector that stores the column indexes of the elements in the *val* vector - *jindx* with length nnz;
- One vector that stores the locations in the *val* vector that start a row - *ptr* with length $n + 1$.

Regarding the storage, with this format instead of storing n^2 elements, we only need to store $2nnz + n + 1$ locations.

D. Algorithm Parallelization using OpenMP

The algorithm we propose is a Monte Carlo method and, as we stated before, these methods generally make it easy to implement a parallel version. Therefore, we parallelized our algorithm using a shared memory system, OpenMP framework, since it is the simpler and easier way to achieve our goal, *i.e.,* to mold a serial program into a parallel program.

To achieve this parallelization, we developed two approaches: one where we calculate the matrix function over the entire matrix, and another where we calculate the matrix function for only one row of the matrix. We felt the need to use these two approaches due to the fact that when we are studying some features of a complex network, we are only interested in having the matrix function of a single row, instead of having the matrix function over the entire matrix.

In the posterior subsections, we are going to explain in detail how these two approaches were implemented and how we overcame some obstacles found in this process.

1) Calculating the Matrix Function Over the Entire Matrix: When we want to obtain the matrix function over the entire matrix, to do the parallelization we have three immediate choices, corresponding to the three initial loops in Fig. 16. The best choice is the first loop (per rows, *rowSize*), since the second loop (*NUM_ITERATIONS*) and the third loop (*NUM_PLAYS*) will have some cycles that are smaller than others, *i.e.,* the workload will not be balanced among threads. Doing this reasoning we can see in Fig. 16, that we parallelized the first loop, and made all the variables used in the algorithm private for each thread to assure that the algorithm works correctly in parallel. Except the *aux* vector, that is the only variable that it is not private, since it is accessed independently in each thread (this is assure because we parallelized the number of rows, so each thread accesses a different row, *i.e.,* a different position of the *aux* vector). It is also visible that we use the CSR format, as stated in Section III-C, when we sweep the rows and want to know the value or column of a given element of a row, using the three vectors (*val*, *jindx*, and *ptr*).

With regards to the random number generator, we used the function displayed in Fig. 17, that receives a *seed*, composed by the number of the current thread (*omp_get_thread_num()*) plus the value returned by the C function *clock()* (Fig. 16). This *seed* guarantees some randomness when we are executing this algorithm in parallel.

2) Calculating the Matrix Function for Only One Row of the Matrix: As discussed in Section II-A, among others features in a complex network, two important features that this thesis focuses on are *node centrality* and *communicability*. To collect them, we have already seen that we need the matrix function for only one row of the matrix. For that reason, we adapted our previous parallel version in order to obtain the matrix function for one single row of the matrix with less computational effort than having to calculate the matrix function over the entire matrix.

In the process, we noticed that this task would not be as easily as we had thought, because when we want the matrix function for one single row of the matrix, the first loop in Fig. 16 “disappears” and we have to choose another one. We parallelized the *NUM_PLAYS* loop, since it is, in theory, the factor that most contributes to the convergence of the algorithm, so this loop is the largest. If we had parallelized the *NUM_ITERATIONS* loop it would be unbalanced, because some threads would have more work than others, and, in theory, the algorithm requires less iterations than random plays. The chosen parallelization leads to a problem where the *aux* vector needs exclusive access, because the vector will be accessed at the same time by different threads, compromising the final results of the algorithm. As a solution we propose two approaches explained in the following paragraphs. The first one using the *omp atomic* and the second one the *omp declare reduction*.

Firstly, we started by implementing the simplest way of solving this problem with a version that uses the *omp atomic* as shown in Fig. 18. This possible solution enforces exclusive access to *aux*, and ensures that the computation towards *aux* is executed atomically. However, as we show in Section IV, it is a solution that is not scalable, because threads will be waiting for each other in order to access the *aux* vector. For that reason, we came up with another version explained in the following paragraph.

```
#pragma omp parallel private(i, q, k, p, j, currentRow, vP,
randomNum, totalRowValue, myseed)
{
    myseed = omp_get_thread_num() + clock();
    #pragma omp for
    for(i = 0 ; i < rowSize; i++)
    {
        for(q = 0; q < NUM_ITERATIONS; q++)
        {
            for(k = 0; k < NUM_PLAYS; k++)
            {
                currentRow = i;
                vP = 1;
                for(p = 0; p < q; p++)
                {
                    randomNum=randomNumFunc(&myseed);
                    totalRowValue = 0;
                    for(j = ptr[currentRow];
                    j < ptr[currentRow+1]; j++)
                    {
                        totalRowValue += val[j];
                        if(randomNum<totalRowValue)
                        break;
                    }
                    vP = vP * v[currentRow];
                    currentRow = jindx[j];
                }
                aux[q][i][currentRow] += vP;
            }
        }
    }
}
```

Fig. 16. Code excerpt in C with the parallel algorithm when calculating the matrix function over the entire matrix.

```
TYPE randomNumFunc(unsigned int *seed)
{
    return ((TYPE)rand_r(seed) / RAND_MAX);
}
```

Fig. 17. Code excerpt in C with the function that generates a random number between 0 and 1.

Another way to solve the problem stated in the second paragraph and the scalability problem found in the first solution, is using the *omp declare reduction*, which is a recent instruction that only works with recent compilers (Fig. 19) and allows the redefinition of the reduction function applied. This instruction makes a private copy to all threads with the partial results, and at the end of the parallelization it executes the operation stated in the *combiner*, *i.e.,* the expression that specifies how partial results are combined into a single value. In this case the results will be all combined into the *aux* vector (Fig. 20). Finally, according to the results in Section IV, this last solution is scalable.

IV. Results

A. Instances

In order to test our algorithm we considered to have two different kinds of matrices:

- Generated test cases with different characteristics that emulate complex networks over which we have full control (in the following sections we call synthetic networks);
- Real instances that represent a complex network.

All the tests were executed in a machine with the following properties: Intel(R) Xeon(R) CPU E5-2620 v2 @ 2.10 GHz, that has 2 physical processors, each one with 6 physical and 12 virtual cores. In total 12 physical and 24 virtual cores; 32 GB RAM ; gcc version 6.2.1; and OpenMP version 4.5.

```
#pragma omp parallel private(q, k, p, j, currentRow, vP,
randomNum, totalRowValue, myseed)
{
    myseed = omp_get_thread_num() + clock();

    for(q = 0; q < NUM_ITERATIONS; q++)
    {
        #pragma omp for
        for(k = 0; k < NUM_PLAYS; k++)
        {
            currentRow = i;
            vP = 1;
            for(p = 0; p < q; p++)
            {
                randomNum = randomNumFunc(&myseed);
                totalRowValue = 0;
                for(j = ptr[currentRow];
                j < ptr[currentRow+1]; j++)
                {
                    totalRowValue += val[j];
                    if(randomNum < totalRowValue)
                    break;
                }
                vP = vP * v[currentRow];
                currentRow = jindx[j];
            }
            #pragma omp atomic
            aux[q][currentRow] += vP;
        }
    }
}
```

Fig. 18. Code excerpt in C with the parallel algorithm when calculating the matrix function for only one row of the matrix, using *omp atomic*.

```
#pragma omp parallel private(q, k, p, j, currentRow, vP,
randomNum, totalRowValue, myseed)
reduction(mIterxlengthMAdd: aux)
{
    myseed = omp_get_thread_num() + clock();

    for(q = 0; q < NUM_ITERATIONS; q++)
    {
        #pragma omp for
        for(k = 0; k < NUM_PLAYS; k++)
        {
            currentRow = i;
            vP = 1;
            for(p = 0; p < q; p++)
            {
                randomNum = randomNumFunc(&myseed);
                totalRowValue = 0;
                for(j = ptr[currentRow];
                j < ptr[currentRow+1]; j++)
                {
                    totalRowValue += val[j];
                    if(randomNum < totalRowValue)
                    break;
                }
                vP = vP * v[currentRow];
                currentRow = jindx[j];
            }
            aux[q][currentRow] += vP;
        }
    }
}
```

Fig. 19. Code excerpt in C with the parallel algorithm when calculating the matrix function for only one row of the matrix, using *omp declare reduction*.

```
void add_mIterxlengthM(TYPE** x,TYPE** y)
{
    int l,k;
    #pragma omp parallel for private(l)
    for(k = 0; k < NUM_ITERATIONS; k++)
    {
        for (l = 0; l < columnSize; l++)
        {
            x[k][l] += y[k][l];
        }
    }
}

#pragma omp declare reduction(mIterxlengthMAdd: TYPE** :
add_mIterxlengthM(omp_out, omp_in))
initializer ( omp_priv = init_priv())
```

Fig. 20. Code excerpt in C with *omp delcare reduction* declaration and *combiner.*

```
A = gallery('poisson',n);
A = full(A);
B = 4 * eye(n^2);
A = A - B;
A = A./-4;
```

Fig. 21. Code excerpt in Matlab with the transformation needed for the algorithm convergence.

1) Matlab Matrix Gallery Package: The synthetic examples we used to test our algorithm were generated by the test matrices gallery in Matlab [14]. More specifically, *poisson* that is a function, which returns a block tridiagonal (sparse) matrix of order n^2 resulting from discretizing differential equations with a 5-point operator on an $n - by - n$ mesh. This type of matrices were chosen for its simplicity. To ensure the convergence of our algorithm we had to transform this kind of matrix, *i.e.,* we used a pre-conditioner based in the Jacobi iterative method (see Fig. 21), to met the restrictions, stated in Section II-B, which guarantee that the method procudes a correct solution. The proof of the Theorema 1 in Page 184 [15], shows that if our transformed matrix has the maximum eigenvalue less than to 1, the algorithm should converge. And the Gershgorin's Theorem proves that our transformed matrix always has absolute eigenvalues less than to 1 [15].

2) CONTEST toolbox in Matlab: Other synthetic examples used to test our algorithm were produced using the CONTEST toolbox in Matlab [16], [17]. The graphs tested were of two types: preferential attachment (Barabsi-Albert) model and small world (Watts-Strogatz) model. In the CONTEST toolbox, these graphs and the correspondent adjacency matrices can be built using the functions *pref* and *smallw*, respectively.

The first type is the preferential attachment model and was designed to produce networks with scale-free degree distributions as well as the small world property [18]. In the CONTEST toolbox, preferential attachment networks are constructed using the command $pref(n, d)$, where n is the number of nodes in the network and $d \geq 1$ is the number of edges each new node is given when it is first introduced to the network. As for the number of edges, d, it is set to its default value $d = 2$ throughout our experiments.

The second type is the small-world networks, and the model to generate these matrices was developed as a way to impose a high clustering coefficient onto classical random graphs [19]. In the CONTEST toolbox, the input is $smallw(n, d, p)$ where n is the number of nodes in the network, which are arranged in a ring and connected to their d nearest neighbors on the ring. Then each node is considered independently and, with probability p, a link is added between the node and one of the others nodes in the network, chosen uniformly at random. In our experiments, different n values were used. As for the number of edges and probability it remained fixed ($d = 1$ and $p = 0.2$).

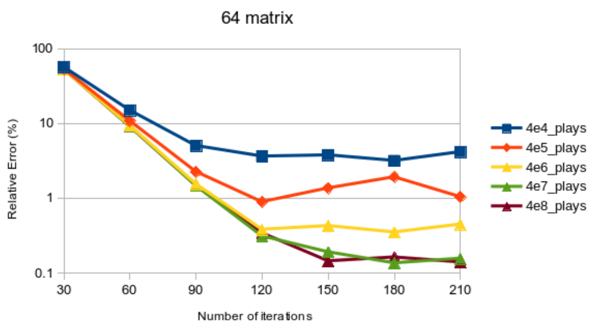

Fig. 22. *inverse matrix function* - Relative Error (%) for row 33 of 64×64 matrix.

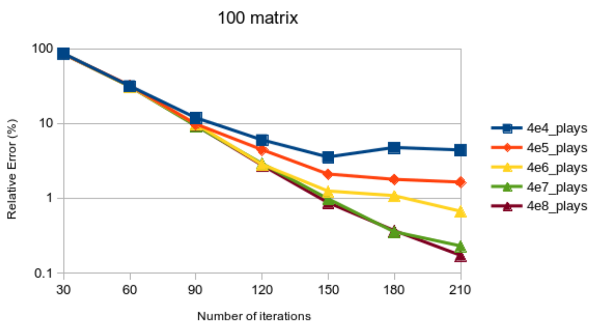

Fig. 23. *inverse matrix function* - Relative Error (%) for row 51 of 100×100 matrix.

3) The University of Florida Sparse Matrix Collection: The real instance used to test our algorithm is part of a wide collection of sparse matrices from the University of Florida [20]. We chose to test our algorithm with the *minnesota* matrix, with size $2,642$, from the Gleich group, since it will help our algorithm to quickly converge since it is almost diagonal. To ensure that our algorithm works, *i.e.,* that this sparse matrix is invertible, we verified if all rows and columns have at least one nonzero element. If not, we added 1 in the ij position of that row e/or column in order to guarantee that the matrix is non singular.

B. Inverse Matrix Function Metrics

In this thesis, we compare our results for the inverse matrix function with the Matlab results for the inverse matrix function, and to do so we use the following metric [15]:

$$RelativeError = \left| \frac{x - x^*}{x} \right| \tag{11}$$

where x is the expected result, and x^* is an approximation of expected result .

In this results, we always consider the worst possible case , *i.e.,* the maximum Relative Error. When the algorithm calculates the matrix function for one row, we calculate the Relative Error for each position of that row. Then we choose the maximum value obtained.

To test the inverse matrix function, we used the transformed *poisson* matrices stated in Section IV-A1. We used two matrices: 64×64 matrix ($n = 8$) and 100×100 matrix ($n = 10$). The size of these matrices is relatively small due to the fact that when we increase the matrix size, the convergence decays, *i.e.,* the eigenvalues are increasingly closed to 1. The tests were done with the version using *omp declare reduction* stated in Section III-D2, since it is the fastest and most efficient version, as we will describe in detail on the following section(s).

Focusing on the 64×64 matrix results, we test the inverse matrix function in one row(random selection, no particular reason), in this case row 33. We also ran these row with different number of iterations and random plays. We can see that, as we expected, a minimum number of iterations is needed to achieve lower relative error values when we increase the number of random plays. It is visible that after 120 iterations the relative error stays almost unaltered, and then the factor that contributes to obtaining smaller relative error values is the number of random plays (see Fig. 22).

Regarding the 100×100 matrix, we also tested the inverse matrix function with one row: row 51, and test it with different number of iterations and random plays. The convergence of the algorithm is also visible, but in this case since the matrix is bigger than the previous one, it is necessary more iterations and random plays to obtain the same results. As a consequence, the results stay almost unaltered only after 180 iterations, demonstrating that after having a fixed number of iterations, the factor that most influences the relative error values is the number of random plays. (see Fig. 23).

Comparing the results where we achieved the lower relative error of both matrices, with $4e8plays$ and different number of iterations,

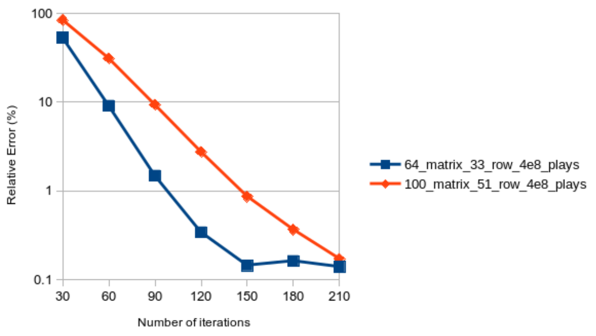

Fig. 24. *inverse matrix function* - Relative Error (%) for row 33 of 64×64 matrix and row 51 of 100×100 matrix.

we can observe the slow convergence of the algorithm and that it decays when we increase the matrix size. Although, it is shown that is works and converges to the expected result (see Fig. 24).

C. Complex Networks Metrics

As we stated in Section II-A, there are two important complex network metrics: *node centrality* and *communicability*. In this thesis, we compare our results for these two complex network metrics with the Matlab results for the same matrix function, and to do so we use the metric stated in Eq. 11, *i.e.,* the Relative Error.

1) Node Centrality: The two synthetic types used to test the *node centrality* were preferential attachment model, *pref*, and small world model, *smallw*, referenced in Section IV-A2.

The *pref* matrices used have $n = 100$ and 1000, and $d = 2$. The tests involved $4e^7$ and $4e^8$ plays, each with $40, 50, 60, 70, 80$ and 90 iterations. We observed that for this type of synthetic matrices, *pref*, the algorithm converges quicker for the smaller matrix, 100×100 matrix, than for the 1000×1000 matrix. The relative error obtained was always inferior to 1%, having some cases close to 0%, demonstrating that our algorithm works for this type of matrices. (see Fig. 25 that shows a specific tested case).

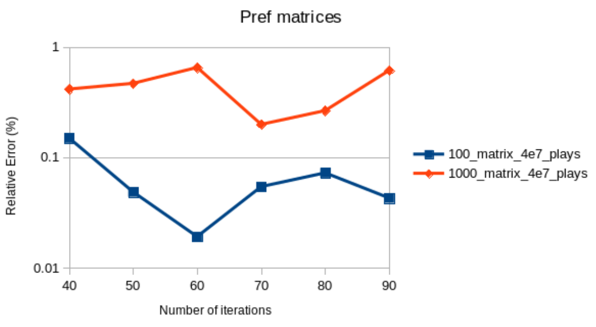

Fig. 25. *node centrality* - Relative Error (%) for row 71 of 100×100 and 1000×1000 pref matrices.

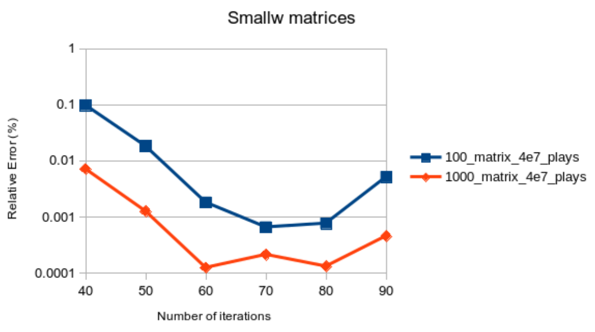

Fig. 26. *node centrality* - Relative Error (%) for row 71 of 100×100 and 1000×1000 smallw matrices.

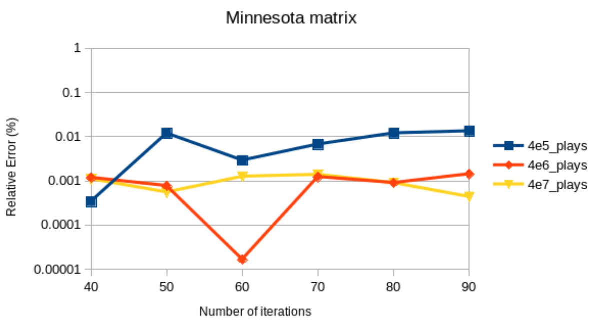

Fig. 27. *node centrality* - Relative Error (%) for row 71 of 2642×2642 minnesota matrix.

The *smallw* matrices used have $n = 100$ and 1000, $d = 1$ and $p = 0.2$. The number of random plays and iterations were the same executed for the *smallw* matrices. We observe that the convergence of the algorithm in this case increases when n is larger having the same N random plays and iterations, *i.e.,* the relative error reaches lower values quicker in the 1000×1000 matrix than in the 100×100 matrix (Fig. 26). It is also possible to observe that the 1000×1000 matrix reaches the lowest relative value with 60 iterations, and the 100×100 matrix needs more iterations, 70. These results support the thought that for this type of matrices the convergence increases with the matrix size.

We conclude that our algorithm retrieves the expected results when we want to calculate the *node centrality* for both type of synthetic matrices, achieving relative error inferior to 1%, in some cases close to 0%. In addition to that, the convergence of the *pref* matrices degrades with the matrix size, whereas the convergence of the *smallw* improves with the matrix size.

Furthermore, we test the *node centrality* with the real instance stated in Section IV-A3, the *minnesota* matrix. We tested with $4e^5, 4e^6$ and $4e^8$ plays, each with $40, 50, 60, 70, 80$ and 90 iterations. We conclude that for this specific matrix, the relative error values obtained were close to 0%, as we expected. Additionally, comparing the results with the results obtained for the *pref* and *smallw* matrices, we can see that with less number of random plays and iterations, we achieved even lower relative error values.

2) Node Communicability: To test the *node communicability*, the two synthetic types used were the same as the one used to test the *node centrality*, preferential attachment model, *pref*, and small world model, *smallw*, referenced in Section IV-A2.

The *pref* and *smallw* matrices used have the same parameters as the matrices when the tested the *node centrality*. We observe that, despite some variations, the relative values stay almost the same, meaning that our algorithm quickly converges to the optimal solution. These results lead to a conclusion that with less number of random plays it would retrieve low relative errors, demonstrating that our algorithm, for these type of matrices, converges quicker to obtain the *node communicability*,*i.e.,* the exponential of a matrix, than to obtain the *node centrality*, *i.e.,* the inverse of a matrix (see results

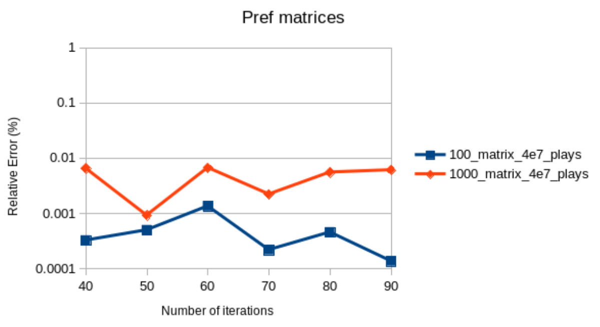

Fig. 28. *node communicability* - Relative Error (%) for row 71 of 100×100 and 1000×1000 pref matrix.

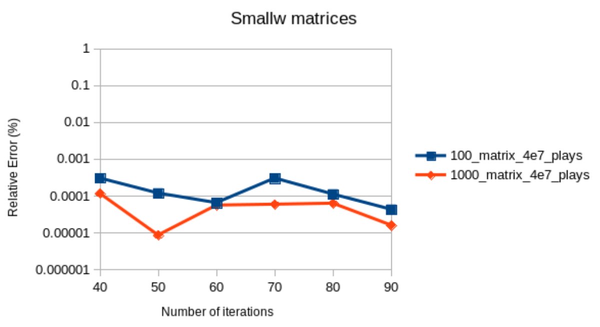

Fig. 29. *node communicability* - Relative Error (%) for row 71 of 100×100 and 1000×1000 smallw matrix.

in Section IV-C1). Finally, as in the previous section, we can see that for the *pref* matrices, the 100×100 smallw matrix converges quicker than the 100×100 pref matrix (see Fig. 28), and the *smallw* matrices, the 1000×1000 smallw matrix converges quicker than the 100×100 smallw matrix (see Fig. 29).

Finally, testing again our algorithm with the real instance in Section IV-A3, the *minnesota* matrix, but this time to test the *node communicability*. The tests were executed with $4e^5, 4e^6$ and $4e^7$ plays, each with $40, 50, 60, 70, 80$ and 90 iterations. As for the *node centrality*, this results were close to 0%, and in some cases with relative error values inferior to the ones obtained for the *node centrality*. This reinforces even more the idea that the exponential matrix converges quicker than the inverse matrix, as we expected (see Fig. 27 and Fig. 30). In conclusion, for both complex metrics, *node centrality* and *node communicability*, our algorithm returns the expected result for the two synthetic matrices tested and real instance. The best results were achieved with the real instance, even thought it was the largest matrix tested, demonstrating the great potential of this algorithm.

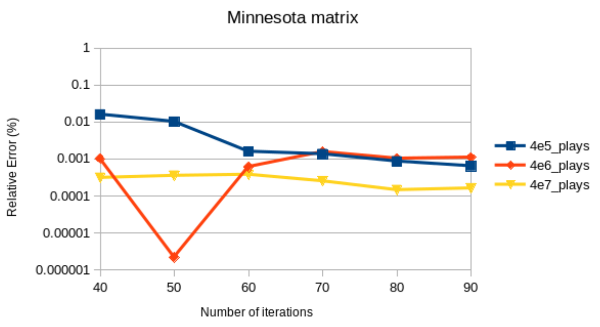

Fig. 30. *node communicability* - Relative Error (%) for row 71 of 2642×2642 minnesota matrix.

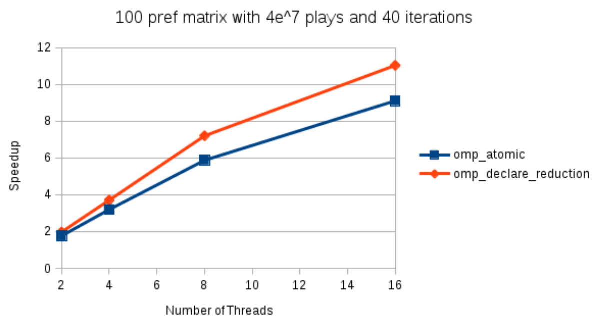

Fig. 31. *omp atomic* and *omp declare reduction* and version - Speedup relative to the number of threads for row 71 of 100 × 100 pref matrix.

D. Computational Metrics

In this section we are going to present the results regarding the scalability of our algorithm. Considering metrics such as the Amdahl's Law, which can help us understand the global impact of local optimization, we observe that our algorithm, in theory, is perfectly scalable, because there is no parallel overhead since it runs in a shared memory system [21]. Taking this into account, we consider the efficiency metric, a measure of the fraction of time for which a processor is employed. Ideally, we would want to obtain 100% efficiency.

The efficiency metric will be evaluated for the versions when we calculate the matrix function for only one row of the matrix, *i.e.,* the version using *omp atomic* and the version using *open declare reduction* (Section III-D2).

Considering the two synthetic matrices referred in Section IV-A1, *pref* and *smallw*, we started by executing the tests using the *omp atomic* version. We execute the tests with $2, 4, 8,$ and 16 threads for $4e^7$ plays and $40, 50, 60, 70, 80$ and 90 iterations. The matrices used were the 100×100 and 1000×1000 *pref* matrices, and 100×100 and 1000×1000 *smallw* matrices. We observed that the efficiency is rapidly decreasing when the number of the threads increases, demonstrating that this version is not scalable, for these synthetic matrices. Comparing the results obtained for both matrices, we can observe that the *smallw* matrices have worse results than the *pref* matrices, achieving efficiency values of 60%. Regarding the results when 16 threads were used, it was expected that they were even worst since the machine where the tests were executed only has 12 physical cores. Taking into account this obtained results, another version was developed where this does not happen. The solution is the *omp declare reduction* version, as we are going to show in the following paragraph.

The *omp declare reduction version* efficiency tests were executed for the same matrices as the *omp atomic* version with the same parameters. Since the machine where the tests were executed only has 12 physical cores, it is expected that the tests with 16 threads have low efficiency. But for the other number of threads, $2, 4,$ and 8, the results were good, always having an efficiency between 80% and 100%, *i.e.,* the efficiency stays almost "constant" when the number of threads increases, proving that this version is scalable, for this synthetic matrices.

Comparing the speedup taking into account the number of threads for one specific case, for both versions, we reach the same conclusion as before, that the *omp atomic* version is not scalable, whereas the *omp declare reduction* is. The desired speedup for x number of threads is x. For example, when we have 8 threads, the desirable value is 8. So, in Fig. 31 in the *omp atomic* we have a speedup of 6, unlike the *omp declare reduction* version that has a speedup close to 8.

V. Conclusions

This thesis was motivated by the fact that there are several areas, such as atmospheric calculation, financial calculation, electrical simulation, cryptography and complex networks, where matrix functions like the matrix inversion is an important matrix operation. Despite the fact that there are several methods, whether direct or iterative, these are costly approaches taking into account the computational effort needed for such problems. So, with this work we aimed to implement an efficient parallel algorithm based on Monte Carlo methods to obtain the inverse and other matrix functions. This work is mainly focused in complex networks, but it can be easily be applied to other application areas. The two properties studied, in complex networks, were the *node centrality* and *communicability*.

We implemented one version of the algorithm that calculates the matrix function over the entire matrix and two versions that calculate the matrix function for only one row of the matrix. This is the main advantage of our work, being able to obtain the matrix function for only one row of the matrix, instead of calculating the matrix function over the entire matrix. Since there are applications where only a single row of the matrix function is needed, as it happens for instance in complex networks, this approach can be extremely useful.

A. Main Contributions

The major achievement of the present work was the implementation of a parallel scalable algorithm based on Monte Carlo methods using OpenMP to obtain the inverse matrix and other matrix functions, for the synthetic matrices tested. This algorithm is capable of calculating the matrix function for a single row of a matrix instead of calculating the matrix function over the entire matrix. Firstly, we implemented a version using the *omp atomic*, but we concluded that it was not scalable, for the synthetic matrices tested. Therefore, we solved this scalability problem with the implementation of another version which uses the *omp declare reduction*. We also implemented a version where we calculate the matrix function over the entire matrix, but since we mainly focus on two complex network metrics, *node centrality* and *communicability*, the previous solution is the most efficient for those metrics. According to the results in Section IV, the solution is capable of retrieving results with a relative error inferior to 1, and it can easily be adapted to any problem, since it converges to the optimal solution where the relative error is close to 0, depending on the number of random plays and iterations executed by the algorithm.

B. Future Work

Despite the favorable results described in Section IV, there are some aspects that can be improved as the fact that our solution has the limitation of running in a shared memory system, limiting the tests with larger matrices.

A future possible improvement for this work is to implement a parallel version using MPI (Message Passing Interface), in order to test the algorithm behavior when it runs in different computers with even larger matrices, using real complex network examples. Although, this solution has its limitations, because after some point the communications between the computers, overhead, will penalize the algorithms' efficiency. As the size of the matrix increases, it would be more distributed among the computers. Therefore, the maximum overhead will occur when the matrix is so large that all computers, for a specific algorithm step, will need to communicate with each other to obtain a specific row of the distributed matrix, to compute the algorithm.

Finally, another possible enhancement to this work is to parallelize the algorithm in order to run on GPUs (Graphic Processor Units), since they offer a high computational power for problems like the Monte Carlo Methods, that are easily parallelized to run in such environment.

References

[1] George E. Forsythe and Richard a. Leibler. Matrix inversion by a Monte Carlo method. *Mathematical Tables and Other Aids to Computation*, 4(31):127–127, 1950.

[2] Richard J Warp, Devon J Godfrey, and James T Dobbins III. Applications of matrix inversion tomosynthesis, 2000.

[3] Jingen Xiang, Shuang Nan Zhang, and Yangsen Yao. Probing the Spatial Distribution of the Interstellar Dust Medium by High Angular Resolution X-Ray Halos of Point Sources. *The Astrophysical Journal*, 628(2):769–779, 2005.

[4] Debora S. Marks, Lucy J. Colwell, Robert Sheridan, Thomas A. Hopf, Andrea Pagnani, Riccardo Zecchina, and Chris Sander. Protein 3D Structure Computed from Evolutionary Sequence Variation. *PLoS ONE*, 6(12):e28766, 2011.

[5] Christine Klymko. *Centrality and Communicability Measures in Complex Networks: Analysis and Algorithms*. PhD thesis, 2013.

[6] Michael T. Heath. *Scientific Computing: An Introductory Survey*. 1997.

[7] A N Kolmogorov. Foundations of the theory of probability. 1956.

[8] I. Dimov, V. Alexandrov, and a. Karaivanova. Parallel resolvent Monte Carlo algorithms for linear algebra problems. *Mathematics and Computers in Simulation*, 55(1-3):25–35, 2001.

[9] J. Straßburg and V. N. Alexandrov. A monte carlo approach to sparse approximate inverse matrix computations. *Procedia Computer Science*, 18:2307–2316, 2013.

[10] Janko Straßburg and Vassil N. Alexandrov. Facilitating analysis of Monte Carlo dense matrix inversion algorithm scaling behaviour through simulation. *Journal of Computational Science*, 4(6):473–479, 2013.

[11] Aiyoub Farzaneh, Hossein Kheiri, and Mehdi Abbaspour Shahmersi. AN EFFICIENT STORAGE FORMAT FOR LARGE SPARSE. *Communications Series A1 Mathematics & Statistics*, 58(2):1–10, 2009.

[12] Mathematical Sciences, Survey Of, Sparse Matrix, Storage Formats, and Control Systems. Sparse matrix storage formats and acceleration of iterative solution of linear algebraic systems with dense matrices. *Journal of Mathematical Sciences*, 191(1):10–18, 2013.

[13] Daniel Langr and Pavel Tvrdik. Evaluation Criteria for Sparse Matrix Storage Formats. *IEEE Transactions on Parallel and Distributed Systems*, 27(August):1–1, 2015.

[14] Test matrices gallery in matlab. http://www.mathworks.com/help/matlab/ref/gallery.html. Accessed: 2015-09-20.

[15] D Kincaid and W Cheney. Numerical Analysis: Mathematics of Scientific Computing, 2002.

[16] Alan Taylor and Desmond J. Higham. CONTEST: A Controllable Test Matrix Toolbox for MATLAB. *ACM Trans. Math. Softw.*, 35(4):1–17, 2009.

[17] A. Taylor and D.J.Higham. Contest: Toolbox files and documentation. http://www.maths.strath.ac.uk/research/groups/numerical_analysis/contest. Accessed: 2015-09-19.

[18] Albert-Laszlo Barabási and Reka Albert. Emergence of scaling in random networks. *Science*, 286(October):509–512, 1999.

[19] Duncan J. Watts and Steven H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393(June):440–442, 1998.

[20] The university of florida sparse matrix collection. https://www.cise.ufl.edu/research/sparse/matrices/Gleich/index.html. Accessed: 2015-10-09.

[21] Michael J. Quinn. *Parallel Programming in C with MPI and OpenMP*. 2003.