A Distributed Monte Carlo Based Linear Algebra Solver Applied to the Analysis of Large Complex Networks[⋆]

Filipe Magalhães[a], José Monteiro[a,∗], Juan A. Acebrón[b,a], José R. Herrero[c]

[a]*INESC-ID, Instituto Superior Técnico, Universidade de Lisboa, Portugal.*
[b]*Dept. Information Science and Technology, ISCTE-University Institute of Lisbon, Portugal.*
[c]*Dept. d'Arquitectura de Computadors, Universitat Politècnica de Catalunya, Spain.*

Abstract

Methods based on Monte Carlo for solving linear systems have some interesting properties which make them, in many instances, preferable to classic methods. Namely, these statistical methods allow the computation of individual entries of the output, hence being able to handle problems where the size of the resulting matrix would be too large. In this paper, we propose a distributed linear algebra solver based on Monte Carlo. The proposed method is based on an algorithm that uses random walks over the system's matrix to calculate powers of this matrix, which can then be used to compute a given matrix function. Distributing the matrix over several nodes enables the handling of even larger problem instances, however it entails a communication penalty as walks may need to jump between computational nodes. We have studied different buffering strategies and provide a solution that minimizes this overhead and maximizes performance. We used our method to compute metrics of complex networks, such as node centrality and resolvent Estrada index. We present results that demonstrate the excellent scalability of our distributed implementation on very large networks, effectively providing a solution to previously unreachable problem instances.

Keywords: Matrix inverse, Monte Carlo, Distributed computation, Network metrics

[⋆]The work has been performed under the Project HPC-EUROPA3 (INFRAIA-2016-1-730897), with the support of the EC Research Innovation Action under the H2020 Programme; in particular, the authors gratefully acknowledge the support of the Computer Architecture Department at Universitat Politècnica de Catalunya (UPC) and the computer resources and technical support provided by Barcelona Supercomputing Center (BSC). This work was also supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) with reference UID/CEC/50021/2020, by the Spanish Ministry of Science and Technology through PID2019-107255GB project and by the Generalitat de Catalunya (contract 2017-SGR-1414).

[∗]Corresponding author.

Email addresses: `filipepgmagalhaes@tecnico.ulisboa.pt` (Filipe Magalhães), `jcm@inesc-id.pt` (José Monteiro), `juan.acebron@iscte-iul.pt` (Juan A. Acebrón), `josepr@ac.upc.edu` (José R. Herrero)

1. Introduction

The high-performance systems now available are essentially multiprocessor machines interconnected by high-speed networks. Very large applications can be accommodated by conveniently distributing the data over a large set of machines. In order to fully exploit the computational power these machines provide, the synchronization points and amount of data interchange between the different machines need to be carefully considered and minimized. This issue is particularly relevant for numerical methods that operate on large vectors and matrices, as is the case of the problems addressed in this paper. Probably the most studied class of methods in the theory, and used in practice, are those based on the Krylov subspace method. The idea behind the method consists in projecting the given (typically large) matrix onto a Krylov subspace (*e.g.*, see [1, 2]). However, such classical methods tend to scale very poorly due to intrinsic data-dependencies [3, 4], and in practice lead to a very low efficiency of utilization of processor power.

The approach proposed in this paper enables the computation of functions over extremely large matrices supported on the following two main features: the implementation splits the matrices across different machines hence combining their memory capacity; and the method allows calculating individual elements of the resulting matrix, avoiding the computation of the entire matrix. While the input matrix is typically sparse, the resulting matrix will, in general, be dense. For this reason, the usage of classic methods becomes unfeasible since they compute this entire matrix, which will not be possible to represent in memory for large problems. Using our method, we can compute separately subsets of the resulting matrix, or even simply compute the particular positions of the matrix that we are interested in.

The method is based on deriving different matrix functions from sums of powers of the matrix, and was originally proposed in [5]. A matrix to the power k is computed using random walks of length k over the matrix. These type of Monte Carlo (MC) methods have the important feature that they are trivially parallelizable, achieving a high degree of efficiency in a parallel system, since each walk is independent of one another. However, the splitting of the matrix across machines, central to the scalability of the method, implies that the walks may have to be continued on another machine. We present a highly optimized implementation that minimizes the impact of handing over one walk to another computational node and demonstrate the scalability of our application up to 768 cores.

Essentially the main goal of these MC methods is to generate a discrete Markov chain whose underlying random paths evolve through the different indices of the matrix. The method can be understood formally as a procedure consisting of an MC sampling of the Neumann series of the inverse of the matrix. The convergence of the method was rigorously established in [6]. More recently, the method was improved by adding an acceleration scheme based on a Richardson iteration (see for instance [7], and [8] just to cite a few references). In [9] a parallel code was implemented for solving linear algebra problems. However, the examples considered were small, avoiding the necessity to distribute the matrix, but limiting its applicability.

One example of an area for which this method can be a significant contribution is in the analysis of complex networks. In recent years, the study of networks has received a renewed interest from the scientific community. It blends with a wide range of areas, with networks emerging for example from social interactions, biological phenomena, the world wide web, financial networks, and the power grid. Network analysis has played an important role in modeling disease spreading, predicting how epidemics progress [10], analyzing the resilience of networks, such as the power grid, to failures both random [11] and maliciously targeted [12]. It is also fundamental in the analysis of social networks, examining the dynamics of personal opinions, collaboration, and habits [13]. A network can be represented by its, typically sparse, adjacency matrix and most metrics used in these analyses can be determined by operations on this matrix.

One prominent metric in the study of networks is node centrality. Centrality can be interpreted as how close to the rest of the nodes in the network one node is, how influential it is, or how important it is in establishing connections between any other pair of nodes. To meet these different perspectives, several formal definitions of node centrality have been proposed before [14]. Centrality as a measure of influence over nodes is important, for example, in the study of social networks or in the classification of pages by search engines over the world wide web, indicating the relative importance of individuals in the network over their peers. Eigenvector Centrality, Katz centrality, and Google's PageRank are concrete instantiations of this type of node centrality [14].

We have developed a highly scalable method to compute the solution of linear systems on distributed-memory systems that is able to handle extremely large problem instances. To demonstrate the relevance of our method, we present results on the computation of node centrality metrics and resolvent Estrada index [15] of complex networks. There are a number of models to generate artificial networks that emulate real networks, useful to test algorithms over networks with well-defined properties. We use small-world matrices generated with a variation of the Watts-Strogatz method [16]; and Kronecker matrices [17] generated using a Graph500 [18] generator.

In this paper, we first make a review of background material. We start in Section 2 with an overview of basic concepts both on linear algebra and some relevant methods, and on networks and the metrics we consider for our analysis. Then Section 3 describes the original algorithm that we base our approach on and presents a discussion about some of the adaptations that we performed for our proposed solution.

The second part of the paper presents the actual contribution and the results we obtained. Section 4 describes the Distributed Monte Carlo (DMC) novel parallel implementation that we developed which performs the operations on matrices fully distributed across the different computational nodes, discussing the tuning effort that significantly reduced the communication overheads leading to the observed scalability of the solution. Experimental evaluations are presented in Section 5 and Section 6 gives some conclusions and discusses future work.

2. Background

In this section, we give a brief overview of the basic background material on linear systems of equations, matrices and networks that is relevant to the work described in the following sections.

2.1. Systems of Linear Algebraic Equations

The problem of solving a System of Linear Algebraic Equations (SLAE) can be written in matrix form, using a vector $\mathbf{x}$ with the variables, a matrix A with the coefficients and a vector $\mathbf{b}$ with the independent terms [19].

$$A\mathbf{x} = \mathbf{b} \quad \Leftrightarrow \quad \mathbf{x} = A^{-1}\mathbf{b} \tag{1}$$

Hence the system of linear equations can be solved by simply computing $A^{-1}\mathbf{b}$ as long as we are dealing with a square and invertible matrix. In practice, more efficient approaches are used, and there are several classes of these methods.

Direct methods, such as the Gauss-Jordan elimination [20] or the multi-frontal solver [21], provide an exact solution (within the limits of the hardware). However, dense matrix operations are involved, incurring in large memory and computation costs.

Iterative methods are applied repetitively, resulting in incrementally more accurate solutions, based on an initial guess [19]. Some of the simplest iterative methods are Richardson's, Jacobi and Gauss-Seidel methods. The most successful algorithms are based on the use of Krylov subspaces, which perform matrix-vector operations in each iteration. There is a large number of algorithms under this category, from which the most commonly used are the (bi)conjugate gradient method [22], GMRES [23] and BICGstab [24].

Monte Carlo methods rely on repeated random sampling to obtain solutions to a wide range of problems [25]. An arbitrary level of accuracy of the solution can be set, appropriate for the problem at hand, through the number of random samples computed. Markov Chain Monte Carlo (MCMC) methods are based on a Markov Chain with a desired distribution as a stationary distribution [26]. They rely on random walks through a matrix.

Monte Carlo methods for SLAE are essentially split between direct and iterative methods. Direct methods rely solely on the stochastic component, and the error of their solutions depends on it. Iterative MC algorithms use more traditional iterative algorithms alongside MC components, generating both stochastic error and systematic error [7].

2.2. Matrix Representation

Typically, most large, complex matrices of interest are sparse, *i.e.*, most of the entries are zero. This is the case of real-world networks where nodes are most times connected to a very small fraction of all the other nodes, meaning that the corresponding adjacency matrix has few non-zero entries per row. There are many data structures that allow an efficient representation of sparse matrices in memory. The Compressed Sparse Row (CSR) is probably the most popular sparse storage scheme [19]. It uses three vectors to store exclusively the non-zero entries of the matrix.

2.3. Network Metrics

Formally, a network is a graph $G =< V, E >$, composed of a set of nodes V, and a set of edges E, being each edge an ordered pair of nodes [14]. Graphs can be represented in the form of a square matrix, called the adjacency matrix, by associating each node with an index, so that each entry in the matrix represents the edge between the node corresponding to its row and the node corresponding to its column [14]. This convention allows us to represent, if necessary, the direction and weight of edges.

To characterize a network and its nodes, there is a vast collection of metrics that can be used [14]. Katz centrality [27] follows the notion of centrality as a measure of the influence of a node on other nodes in the network. Katz's notion of centrality, later refined in [28], assigns a measure of the importance of a node based on the importance level not only of its adjacent nodes in the network but also on their successive neighbors, albeit attenuated by a factor α^d being d is the distance to the node. Given the adjacency matrix A of the network, Katz centrality can be determined by

$$K(\alpha) = (I - \alpha A)^{-1}\mathbf{1} \tag{2}$$

with α the attenuation scalar parameter [29] and $\mathbf{1}$ a vector with all ones. $K(\alpha)$ is a vector representing the centrality value of each node. Parameter α, named the attenuation factor, should be smaller than $1/\lambda$, with λ being the largest eigenvalue of A [27]. Through the tuning of α, this centrality metric can be used to distill different information. Notably, it is shown in [29] that, as α tends to 0, Katz centrality approaches the value of the degree centrality; and as α tends to $1/\lambda$, Katz centrality approaches the value of eigenvector centrality. Moreover, vector $\mathbf{1}$ can be modified to have different values, allowing different weights in the contribution of each node to the centrality. Note that the costly computation of eigenvalues can be avoided by leveraging Gershgorin's Theorem [30]. Thus, one can estimate α from information about the node in the network with the highest degree, something which can be computed at a much lower cost.

There are other types of metrics for networks. The exponential of the adjacency matrix can be used to obtain other useful network measures related to the centrality, communicability, or betweenness of nodes in a network. However, computing the exponential is computationally costly. An alternative was introduced in [15]. For any matrix $A \in \mathbb{R}^{N\times N}$, the function $g(s) = (A - sI)^{-1}$ over $s \in \mathbb{C}$ is called the matrix *resolvent*. The resolvent is related to the exponential through the Laplace transform and can be used to compute similar metrics. The resolvent measure can be regarded as lying between the degree and exponential measures.

An important tool used to get a global characterization of a network is the Estrada index. Originally it referred to the trace of the matrix exponential, used to compute the sum over all nodes of the subgraph centrality based on the exponential, and has since become known as the Estrada index [31]. However, it is also used in a wider sense to refer to the sum of some metrics computed for all the subgraphs computed as the trace of a matrix. For instance, the resolvent-based Estrada index, *i.e.* the trace of the resolvent, can be seen as a global measure of clustering.

There exist other network metrics which can be computed on matrix functions. For example, the diagonal of the matrix exponential represents the exponential subgraph centrality [32]; and the trace of the third power of the adjacency matrix yields the number of triangles in a network, which is a useful metric in the analysis of networks, but difficult to calculate for large graphs [33].

This work will focus on the Katz centrality and the resolvent Estrada index (which we will refer to as the trace of the inverse of the matrix), but the method would require only minor adaptations to focus on other of the many existing metrics.

2.4. Evolving Networks

It is possible to calculate the inverse of a matrix knowing the inverse of another using the Sherman-Morrison formula [34] or its generalization, the Woodbury formula [35]. Let us consider a matrix A, of size N by N, and a second matrix B that results from small modifications to A. As long as we can describe the difference between the two matrices as UV^T, where U and V are matrices of size N by K, and both the original matrix A and the new matrix $B = A + UV^T$ are invertible, the inverse of the new matrix B is given by the following expression:

$$B^{-1} = A^{-1} - A^{-1}U(I_k + V^T A^{-1} U)^{-1} V^T A^{-1} \tag{3}$$

This formula allows the computation of metrics based on the inverse of the matrix for an evolving network, based on a previously obtained inverse.

3. Description of the Method

We describe the method of [5] and present some adaptations made for our implementation. Let us consider the matrices $B = \{b_{ij}\}$, $A = \{a_{ij}\} \in \mathbb{R}^{N\times N}$ and let $A = I - B$, where I is the identity matrix. Then

$$B^{-1} = (I - A)^{-1} = \sum_{k=0}^{\infty} A^k \tag{4}$$

as long as, considering $\lambda_r(A)$ as the r-th eigenvalue of A, the following holds:

$$\max_r |\lambda_r(A)| < 1. \tag{5}$$

In order to approximate B^{-1}, the method under study approximates the sum of the first n powers of A using random walks, similarly to Markov Chain Monte Carlo. The method calculates $\widehat{B}$ as

$$\widehat{B} = \sum_{k=0}^{n} A^k. \tag{6}$$

$\widehat{B}$ is a good approximation as long as n is large enough, since the summation converges to B^{-1} as n tends to infinity. In [5, 8] it has been shown that each power k of the matrix A in Eq. (6) can be represented probabilistically as the estimator of a suitable multiplicative random variable η. Such a multiplicative random variable is defined as $\eta = \prod_{l=1}^{k} v_{X_l}$. Here $\mathbf{v} \in \mathbb{R}^N$ is a vector with components $v_j = \sum_j a_{ij}$, and $X_0, X_1, \ldots$ is a discrete-time Markov chain. This Markov chain is defined on the state space $S = \{1, 2, \cdots, N\}$ and the one-step transition probability matrix $P = \{p_{ij}\}_{i,j=1}^{N}$ is given by $p_{ij} = a_{ij}/\sum_j a_{ij}$, provided $a_{ij} > 0$. Then, the power k of the matrix A can be computed probabilistically as $(A^k)_{ij} = \mathbb{E}[\eta\, \delta_{j\, X_k}]$ with $X_0 = i$. Note, however, that to implement the method in practice, we need to choose a finite sample of given size p, replacing in such a way the expected value by the arithmetic mean. Accordingly, this entails a statistical error that will be discussed in Sec.5.

The method allows computing each row of each power of matrix A independently, as described in Algorithm 1. Naturally, we can compute only the subset of required rows for the problem at hand.

Algorithm 1 Monte Carlo Matrix Inversion [5]

Input: B: matrix to invert; n: length of series calculated; p: number of plays.
Output: $\widehat{B}$: approximation of B^{-1}

```
1: function InvertMatrix($B$, $n$, $p$)
2:     $dim \leftarrow$ size of $B$                               ▷ Dimension of the matrix
3:     $I \leftarrow$ IdentityMatrix($dim$)
4:     $A \leftarrow I - B$
5:     for $i \leftarrow 0$ to $dim - 1$ do                      ▷ Compute each row independently
6:         $\widehat{B}_{i*} \leftarrow [0, \ldots, 0]$
7:         for $k \leftarrow 2$ to $n$ do                        ▷ Compute the different powers of $A$
8:             $\widehat{B}_{i*} \leftarrow \widehat{B}_{i*} +$ CalculateRow($A, i, k, p$)     ▷ Algorithm 2
       return $\widehat{B}$
```

In order to calculate row i of A^k, the method computes a vector $\widehat{B}_{i*}$ with the same length as the row, initially with 0 in all entries. A neat picture of this probabilistic representation can be described as follows. First, let us choose a finite number of p trials for the Markov chain simulations, which hereafter we call "plays". Those "plays"start from this row i, and during k steps evolve by random paths through the different indices of the matrix, which corresponds to a different state of the Markov chain. At each step one entry (A_{ij}) of the current row i is chosen, and jumps to the row with index j, as described in Algorithm 2. For the column selection, the next row is chosen randomly with probabilities for each entry in the row weighted proportionally to their value (obtained via RandomWeightedChoice()).

A play starts with a value of 1 and, at each step, is multiplied by the sum of the entries in the current row ($\sum_j A_{ij}$ obtained via GetRowWeight())[1].

[1]In the implementation, for efficiency, this can be retrieved from a precomputed vector.

Algorithm 2 Monte Carlo Row Power Computation [5]

Input: A: input Matrix; i: row index; k: length of play used to approximate the k^{th} power of A; p: number of plays.

Output: r: approximation of row i of k^{th} power of A

```
1:  function CalculateRow($A$, $i$, $k$, $p$)
2:      $dim \leftarrow$ size of $A$                                  ▷ Dimension of the matrix
3:      $r[0 : dim - 1] \leftarrow [0, \ldots, 0]$
4:      for $m \leftarrow 1$ to $p$ do                        ▷ Use $p$ plays in the approximation
5:          $value \leftarrow 1$
6:          $currentRow \leftarrow i$
7:          for $l \leftarrow 0$ to $k - 1$ do                                  ▷ Length of play
8:              $value \leftarrow value \times$ GetRowWeight($A$, $currentRow$)
9:              $selectedColumn \leftarrow$ RandomWeightedChoice($A$, $currentRow$)
10:             $currentRow \leftarrow selectedColumn$
11:         $r[currentRow] \leftarrow r[currentRow] + value$
12:     for $j \leftarrow 0$ to $dim - 1$ do
13:         $r[j] \leftarrow r[j]/p$
        return $r$
```

When a play reaches the end of its k steps, its value is added to the vector r at the index corresponding to the row where the play finished. In the end, the values in r must be normalized by the number of plays, yielding an approximation of the requested row i of A^k.

In order to calculate A^{k+1}, plays of length $k+1$ are used. These, by definition, contain plays of length k. Therefore, instead of computing p different plays for each power k, we can reduce significantly the total number of steps simply by saving the results obtained in each step l of the play and use them for computing the power $l+1$, *i.e.*, use pn steps instead of the $pn(n+1)/2$ steps computed by the algorithms presented above. This change comes at the cost of making the approximation of each power dependent on the approximation of the lower powers. This could lead to larger errors, but experiments showed only a small increase in the error, and a very beneficial reduction in execution time. In Section 4, we use this optimization when processing plays in parallel in Algorithm 5.

When the goal is to calculate the product of a vector by the inverse of the matrix ($B^{-1}v$), which is the case both in solving systems of linear equations and when calculating the Katz centrality of a network, the algorithm can be modified to store only vectors rather than the full matrix. In the calculation of the centrality, since the vector to be multiplied contains 1 in every entry, we only need to sum all entries of the resulting matrix row when calculating one row of the inverse, therefore saving memory space.

Additionally, the trace of the resulting matrix can be easily computed, by summing only the entries in the diagonal. This method can also be used to approach the problem of recalculating the inverse after changes to the original matrix, using the Woodbury formula, as described in Section 2.4. To apply this formula to the calculation of the Katz centrality, where the inverse is multiplied by a vector, we need to store the result of the inverse multiplied by several vectors, namely each of the columns of the matrix that describes the changes to the matrix. These can be computed in a single execution. In order to multiply the inverse of the matrix by an arbitrary vector, we just need to multiply the result stored at each step (line 11 in Algorithm 2) by the vector entry corresponding to the current row. In order to compute several vectors at the same time, we need to store results for each of the vectors, multiplying each by the appropriate vector's entry. This corresponds to executing line 11 for each of the vectors, storing the result in separate output vectors.

The underlying method bases itself on the calculation of individual rows of powers of a matrix. This provides flexibility to calculate other matrix functions, such as the exponential

$$e^A = \sum_{k=0}^{\infty} \frac{1}{k!} A^k$$

by simply changing the weights of each matrix power. Hence, we could easily modify the method for the computation of other network metrics and different definitions of centrality [29].

4. Parallel Implementation

This section presents our fundamental contribution towards the deployment of a highly scalable parallel method for the computation of matrix functions. A number of matrix functions can be computed through the calculation of individual rows of the sums of the powers of a matrix. We demonstrate the impact of our work using the calculation of the inverse of a matrix times a vector as an example of the application of the method. Since the method uses a Monte Carlo approach, where the calculation of each row is totally independent of the others, it is therefore trivially parallelizable by computing them in different processing units. In principle, this requires that all machines have access to the entire matrix. However, when a matrix is too large for the memory of a single machine, we need to distribute the matrix across machines, where each machine keeps a part of the matrix in its memory. Also, partial results need to be eventually added with a reduction operation in order to obtain the global result. Algorithm 3 shows the skeleton of the distribution of data and work to the processes.

This distribution implies that plays that start in one machine may eventually have to be transferred over to another, introducing a potentially significant communication overhead. This process is illustrated in Figure 1. Our implementation uses a row-wise matrix distribution and an owner-computes rule. Each machine starts computing the plays associated with the rows it holds in memory. When a play reaches a row assigned to another machine, the machine which was computing the play sends a message with the information of the *outgoing play* to be continued in the destination machine (line 8 in Algorithm 5 and line 6 in Algorithm 4). Therefore, a play is represented by a data structure which

Algorithm 3 Pseudo-code of a Parallel Distributed Monte Carlo $B^{-1}v$

Input: B: matrix to invert; v: vector to multiply; p: number of plays; n: length of series calculated; [α: scalar parameter used when computing Katz centrality (1 otherwise)].

Output: $\widehat{x}$: approximation of $B^{-1}v$

```
1: function DISTRIB_MC_INVMAT_X_VEC($B, v, n, p$ [, $\alpha$])
2:     $NP \leftarrow$ Number of processes
3:     $Pid \leftarrow$ Process Id
4:     $dim \leftarrow$ size of $B$                                  ▷ Dimension of the matrix
5:     $(rini, nrows) \leftarrow$ DEFINESUBDOMAIN($dim$, $NP$, $Pid$)
6:     $A_s \leftarrow$ GETSUBMATRIX($B$, $dim$, $rini$, $nrows$, $\alpha$) ▷ $A_s \leftarrow I - \alpha B$ (in subdomain)
7:     $xlocal \leftarrow$ MC_SUBMROWSPOWER_X_VEC($A_s, v, dim, nrows, rini, n, p$) ▷ Alg 4
8:     $x \leftarrow$ apply a reduce operation over $xlocal$         ▷ Accumulate partial results
9:     return $x$                                                  ▷ Final result
```

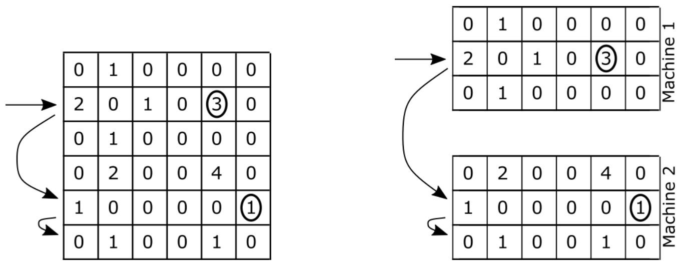

Figure 1: Illustration of the need for communicating plays between machines when the rows of the matrix are distributed, (a) representing the basic algorithm when the entire matrix is available locally, and (b) representing the situation where a play needs to migrate to another machine because the destination row is only available on that machine.

contains: the starting row being simulated; the current row, which is updated though the simulation; a value, which is initialized to 1 and then updated by multiplying the current row weight; and the number of steps left, where the initial number of steps is initialized to the total number of powers of the matrix that one wants to compute.

For simplicity, Algorithms 3 through 5 roughly describe the skeleton of an unoptimized distributed memory parallelization with a single thread per process. However, the sequel of this section presents some of the improvements that have been implemented. Our implementation uses MPI to distribute work and data across machines, and OpenMP to exploit multiple cores that share memory within each machine.

Memory management is of paramount importance. In particular, dynamic memory allocation is a source of overhead, and, as such, is totally avoided, aside from the initial allocations. The efficient use of the memory hierarchy is important to maintain performance. The nature of the random walks, leading

Algorithm 4 Distributed MC computation of n^{th} power of rows in A_s times v

Input: A_s: submatrix to use to calculate its initial n powers; v: vector to multiply; dim: problem dimension; $nrows$: number of rows in the subdomain; $rini$: initial row in the subdomain; n: length of series calculated; p: number of plays.

Output: $\hat{x}$: partial result contributing to the approximation of $B^{-1}v$

```
1: function MC_SubmRowsPower_x_Vec($A_s, v, dim, nrows, rini, n, p$)
2:     $x[0 : dim - 1] \leftarrow [0, \ldots, 0]$
3:     while pendingWork( ) do                    ▷ ∃ unprocessed plays (see Section 4.3)
4:         $play \leftarrow NULL$
5:         if incomingMessage( ) then                                  ▷ ∃ incoming play
6:             $play \leftarrow$ receivePlay()                ▷ Receive play from other process
7:         else if ungeneratedPlays( ) then                    ▷ $p$ plays × $nrows$ in total
8:             $play \leftarrow$ generatePlay()                  ▷ Generate one play at a time
9:         if $play$ is defined then
10:            $x \leftarrow$ processPlay($A_s, play, x, v, nrows, rini$)              ▷ Algorithm 5
11:    for $i \leftarrow 0$ to $dim - 1$ do
12:        $x[i] \leftarrow x[i]/p$
13:    return $x$                              ▷ Local contribution computed in this process
```

to random accesses on the matrix and a lack of temporal locality in accesses, makes it difficult to take advantage of the cache memories. Nonetheless, spatial locality can be leveraged by the cache by keeping all the relevant information about each row of the matrix adjacent in memory.

4.1. Row Distribution

By default, the rows of the input matrix are equally distributed among all machines. However, some matrices may be unbalanced in the sense that some rows may have significantly more non-zero entries than others, leading to an uneven level of computation cost across machines. This is the case when the network presents hubs, where many random walks will pass. To allow a

Algorithm 5 Process a play for row $i = startRow$ while falls in this subdomain

Input: A_s, v, $nrows$ and $rini$ as in Algorithm 4; pl: current play; x: partial result calculated until now.

Output: $\hat{x}$: partial result contributing to the approximation of $B_{i*}^{-1}v$

```
1: function processPlay($A_s$, $pl$, $x$, $v$, $nrows$, $rini$)
2:     while $pl.stepsLeft > 0$ and ownRow($pl.currentRow$, $rini$, $nrows$) do
3:         $pl.stepsLeft \leftarrow pl.stepsLeft - 1$
4:         $pl.value \leftarrow pl.value \times$ GetRowWeight($A_s, pl.currentRow$)
5:         $pl.currentRow \leftarrow$ RandomWeightedChoice($A_s, pl.currentRow$)
6:         $x[pl.startRow] \leftarrow x[pl.startRow] + pl.value \times v[pl.currentRow]$
7:     if $pl.stepsLeft > 0$ then                    ▷ Unfinished play leaves this subdomain
8:         sendToRowOwner($pl$)
9:     return $x$
```

better load distribution, a custom row distribution is accepted by the program. This custom distribution was used in experiments over matrices with hubs, such as networks generated using the scalable Graph500 Kronecker generator, where the number of connections, instead of the number of nodes, was equally divided overall processes. This is a simplistic approach to estimate the expected workload of each process but proved successful in keeping a good workload balancing. In addition, if some rows have a large number of non-zero entries, such rows can be replicated in all the domains in order to reduce jumps from domain to domain.

4.2. Communication

To hide as much as possible the communication overhead, the sending and receiving of messages is performed using the asynchronous versions of MPI calls, so that computation and communication can be overlapped. In our approach, while a node is processing the walks currently within its set of rows, in the background it may be receiving new walks to process. The performance advantage of both sending and receiving messages asynchronously was verified experimentally.

Furthermore, and to reduce the number of messages sent, which could be overwhelming otherwise, outgoing plays are aggregated in a dedicated buffer for each destination, and sent in a single message when the buffer is full. The size of this buffer is a parameter of the implementation, determining the size of the message, and therefore, how often communication occurs. This parameter impacts heavily the performance of the method, the larger the buffer the lower the communication overhead since we have a lower number of messages, however the later the receiving process starts working on the plays coming from another process. Moreover, the characteristics of the interconnect network play an important role in determining the optimal size of the buffer. Hence, a good practical way to arrive at the optimal size of the buffer for a given machine is to perform a few experiments beforehand with different sizes. For the machine we used in this work (MareNostrum 4), we experimentally determined that a buffer that accommodates 1024 plays gives the best results. All results in Section 5 were obtained using this value.

On the receiving end, we manage a set of buffers for incoming messages, filled using asynchronous receive calls. The size of these buffers has less impact on performance, as long as they are big enough not to fill up and delay the computation. Although MPI has the option to internally manage a buffer when sending messages, using the “Bsend” call, we found its behavior and performance worse than our own implementation.

4.3. Terminating the Computation

In order to determine when the computation has finished we use a token-ring algorithm. At the start of the execution, a token with the value 0 is assigned to the first process. When a process becomes idle (*i.e.*, all local plays have been generated and no incoming plays), if it is holding the token it adds the number of finished plays the process has registered locally to the value of the token, and sends it over to the next process. When the token reaches the first process again, we need to check if the token's value is equal to the total amount of plays. In that case, the computation is finished, and all processes get a termination message. Otherwise, this means that there were plays that could not be completed in the current subdomain and had to be sent to another node, restarting their computation in a process where the token had already gone by. In this case, the value of the token is reset to 0, and this procedure is repeated.

This strategy only uses the idle time of the processes to check for the end of the computation, placing no special burden on any of the processes, and is flexible to different strategies of where plays should be considered as terminated.

4.4. Threads

It is desirable to use threads at each computational node, in order to take full advantage of shared-memory computation in multi-core machines. Using threads instead of several processes on the same machine permits storing a larger chunk of the matrix in the same process, and have several processors sharing it, reducing the amount of communication needed.

4.4.1. Thread Organization

When using several threads in the same process, we have the choice to make them homogeneous, *i.e.*, all performing the same tasks and communication, or to specialize some threads in communication and others in processing.

An implementation with one thread dedicated to receiving messages and placing them on input buffers for each of the other threads was compared with a homogeneous implementation where each thread issues the MPI calls necessary to fill its input buffers. The heterogeneous version proved to be faster. Using Barcelona Supercomputing Center (BSC)'s performance visualization and analysis tool Paraver [36], it was observed that the cache misses caused by MPI calls were concentrated in the communication thread, reducing them in the others.

We explored the usage of more communication threads, each serving a subset of the working threads. However, such an approach did not provide any improvement. The analysis using Paraver showed that calls to MPI were taking longer in each thread, making the performance similar to the version with only one communication thread, hinting that synchronization happening inside the MPI library may be the source of contention. The chosen implementation, therefore, features one communication thread per process, responsible for issuing MPI calls to fill the input buffer of each thread, and each of the working threads consuming from their own private buffer.

4.4.2. Workload Distribution among Threads

Each process must generate plays for the rows it owns. In the multi-threaded version, each thread generates an equal subset of the plays, when it has no incoming messages to process. Incoming messages are distributed in a round-robin fashion among the worker threads.

The input buffer of each thread is treated as a circular buffer. Each worker thread acts as a consumer, advancing an index through the existing messages, while the communication thread acts as a producer, advancing another index as it fills the buffer. Thanks to this configuration it is not necessary to synchronize using atomic access to these variables.

4.4.3. Sending Messages

In order to keep threads as independent as possible, each of them has its own set of output buffers, where plays going to other processes are aggregated into messages to be sent by each thread separately. Synchronization outside the MPI calls is completely avoided here. When threads go idle, with no more plays to process, the plays waiting to be sent are aggregated in buffers shared by all threads. This synchronized aggregation of plays only happens when threads are idle, minimizing the impact on performance while ensuring a reduced number of messages and with as many plays as possible.

4.4.4. Storing the Results

As each play finishes, its value is added to a vector of results. This vector can be divided or replicated. In particular, each process can have partial results for all rows or just for the rows owned by the process. Also, each thread can have its own private partial result vector, or threads in the same process can share the memory space used to store results.

If only the owned rows are stored in each process, at the end of a random walk it is necessary to send the result to the process where it started, resulting in an extra communication step for most plays. It is desirable to avoid this extra step by keeping a full-sized vector of results in every process. However, as the matrix size increases, keeping the full vector in memory in every process is an unacceptable strain on the available memory. As such, our final implementation has each process store only the results for the rows it owns, therefore, differing from the replicated results vector used in the skeleton presented in Algorithm 3.

Moreover, in a multithreaded implementation, we need to consider whether to replicate or not such results vector within each process. With one vector where all threads within a process write, every write must be synchronized. Since each of these updates accesses a single memory position, the “omp atomic” directive can be used instead of more elaborated, and slow, synchronization mechanisms. Nonetheless, unless memory space is very limited, it is preferable to have independent memory space for each thread and avoid synchronization altogether, save for a reduce operation at the end of the computation. This option allows for greater performance and was the choice made for the final implementation.

5. Numerical Errors and Parallel Performance of the Algorithm

The designed algorithm[2] was evaluated both in terms of accuracy and parallel performance in the analysis of complex networks, namely when computing two different network metrics: the centrality and the resolvent Estrada index of different networks. Since the required computational steps of the MC method are essentially the same for both metrics, we analyzed separately only the accuracy for these two metrics. Concerning the examples simulated so far, they consist of the adjacency matrices corresponding to two different complex networks, namely a small-world network and a Kronecker graph. In addition, the so-called Poisson matrix resulting from the numerical discretization of the 2D Laplacian operator through a 5-point finite difference stencil is considered. Although the underlying adjacency matrix of such problem cannot be regarded as complex (quite the opposite, it is typically used as the prototype of a perfectly structured homogeneous lattice), it is used in the following with the purpose of merely assessing the efficiency of our implementation for dealing with highly heterogeneous networks. The results are compared with those obtained with a homogeneous network characterized by an extremely regular intercommunication pattern among the different processors involved in the parallel computation. For the case of the small-world network, a custom algorithm was used to generate the matrices. For the Kronecker graph, the algorithms we used to generate the corresponding adjacency matrices belong to the well-known Graph500 benchmarks suite. Both networks are specifically generated to be directed, and therefore the corresponding adjacency matrices are asymmetric.

Direct solvers can produce accurate solutions. However, they suffer from a lack of scalability when dealing with large sparse matrices. This is due to the fill-ins incurred by direct methods that cause the matrix to become dense, with the consequent increase in memory and computation time requirements. Hence, iterative solvers are the methods of choice when solving large linear systems of equations if the matrix is sparse unless the coefficient matrix is ill-conditioned [19]. For instance, the GMRES method implemented in PETSc is very efficient in calculating the product of a vector by the inverse of the matrix ($B^{-1}v$) used when computing the vector of centralities of a network. This is because only a few iterations are needed to achieve a solution with sufficiently good accuracy. MC methods cannot be competitive in this case since they have a slow convergence rate to the solution of the numerical method due to their stochastic nature, requiring many repetitions in order to reach some desired accuracy level. However, in view of the Woodbury formula presented in Section 2.4, the MC algorithm can be used to compute the action of the inverse matrix over several different vectors during a single execution. This is a competitive advantage which can make MC very efficient at computing the resolvent Estrada index, as well as other metrics that require calculating the

[2]The current version of the code is available at `https://github.com/filipepgm/MatrixInversion`.

trace of a matrix.

In the sequel of this section, we first check the quality of the solutions found by DMC against those provided by direct methods of reference. Next, we show the scalability of DMC. Finally, we compare the performance of DMC against an iterative method when computing the trace of the inverse of a matrix or the action of the inverse matrix over several vectors. We do so by comparing DMC results with those obtained by other well-established methods: on the one hand, we use direct methods based on the LU factorization in Matlab [37] and a multifrontal method in MUMPS [38]; on the other hand, we use the GMRES implementation in PETSc [39] as the iterative solver.

The simulations on a distributed memory architecture were carried out on a subset of the MareNostrum 4 Supercomputer at the Barcelona Supercomputing Center (BSC). In our experiments we used up to 16 computational nodes, where each node has two Intel Xeon Platinum chips, each with 24 processors, amounting to a total of 768 cores. Nodes are interconnected by an InfiniBand network. We used Intel's icc, MPI and OpenMP in version 2018.1; MUMPS version 5.1.2; and PETSc version 3.8. The OS used was a Linux kernel release 4.4.114. All computations were performed using single-precision floating point operations except for the direct solver used as a reference in order to check the accuracy of the solution calculated by DMC. In those cases computations were done in double-precision and the order of the relative error was 10^{-15}. On the other hand, PETSc has a configurable tolerance that can be initially set up by the user and determines in practice the accuracy of the numerical solution. Therefore, it can be conveniently adjusted to obtain a solution characterized by an error similar to that of the solution obtained using the MC method. Since the target relative error of the numerical simulations using the MC method was kept fixed to 10^{-5}, PETSc was conveniently adjusted to obtain the corresponding solution within the same relative error range.

Due to the random nature of any of the MC algorithms, it is worth observing that the obtained results can vary from simulation to simulation. To mitigate such variability, all the results presented in this section correspond to an average solution obtained repeating the simulations with 6 different initial random seeds of the pseudo-random generator.

5.1. Numerical Errors and Solution Accuracy

Our numerical method has two sources of error, we have to face the error due to the truncation of the infinite power series (Eq. 4) by a sum of a finite number of powers of the matrix, and the error due to using a given finite number p of plays. The second error is the well-known Monte Carlo statistical error, and is known to be of order of $O(p^{-1/2})$ [40]. More precisely, for a large value of p, this error turns out to be close to a random Gaussian variable with standard deviation proportional to $p^{-1/2}$, being approximately $\frac{\sigma\nu}{p^{1/2}}$, where σ denotes the square root of the variance and ν is a standard normal (*i.e.*, $N(0,1)$) random variable. To ensure the convergence to the expected value of the corresponding estimator used in the MC simulations, it is important to guarantee the finiteness of the variance and, consequently, the first and second moment of the multiplicative random variable η defined in Sec. 3. Since $\eta = \prod_{l=1}^{k} v_{X_l}$, it suffices to ensure that every component v_j of the vector $\mathbf{v}$ is less than one. This indeed holds true in view of the definition of the Katz centrality in Eq. (2). In fact, from the definition of the centrality, the adjacency matrix has to be rescaled multiplying it by the scalar parameter α, which turns out to be smaller than $1/\sum_j a_{ij}$ by the Gershgorin theorem, and hence it holds that $v_j < 1$. Moreover, following the same reasoning, it can be seen that the variance turns out to be independent of the size of the matrix, and consequently it is not needed to increase further the sample size when dealing with larger matrices.

Concerning the first error, it is in practice related to the length of the random walk. Therefore, it decreases when a large number of powers are computed, or equivalently, long random walks inside the matrix are generated. Since the power series is known to converge (provided the requirements laid out in Section 3 are verified), computing matrix powers of higher order, which entails a high computational cost, can be avoided. Quantifying precisely the error incurred when ignoring such matrix powers requires specifically knowing the convergence rate to the solution of the given matrix, which in general is not available. Naturally, the length, as well as the number, of plays is directly related to the computational time.

In the following, and for the specific case of the Katz centrality, the accuracy of the vector solution computed with the proposed MC method was assessed by comparing the results with those obtained using Matlab's direct method `linsolve`, which corresponds to an LU factorization with partial pivoting. This method is based on more precise methods than MC, and therefore we can assume its results to be an accurate approximation of the, generally unavailable, theoretical solution. The relative error ε_r of K^{MC} was computed using the L^2-norm as usual, that is

$$\varepsilon_r = \frac{\|K^{MC} - K^{Matlab}\|_2}{\|K^{Matlab}\|_2}, \tag{7}$$

where $\|x\|_2 = (\sum_i x_i^2)^{1/2}$.

These experiments were performed on a small-world network with 4,096 nodes generated using Matlab with default parameters for the number of nearest-neighbors to link and the probability of adding a shortcut (Matlab uses 1 and 0.1 for these values, respectively). We performed the same experiments with Poisson matrices and observed similar results.

Figure 2 shows the relative error as a function of the number of plays for a small-world network adjacency matrix, and this is done for several lengths of the random walks, *i.e.*, maximum power of the matrix. Note that the error decreases when the number of plays increases, and this occurs independently of the length of the plays. Moreover, the slope of the curves can be readily computed, and it turns out to be approximately $-1/2$, hence in good agreement with the theoretical considerations explained above.

For the case of a small-world network adjacency matrix, Figure 3 shows that when the length of the plays increases, the error decreases accordingly. The

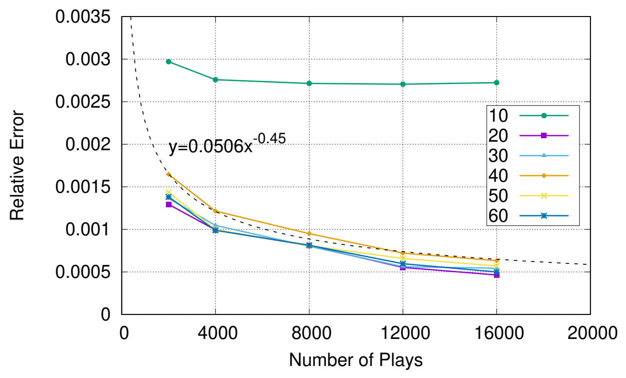

Figure 2: Relative error of the centrality of a small-world network with 4,096 nodes as a function of the number of plays. Each curve corresponds to a different length of the random walk.

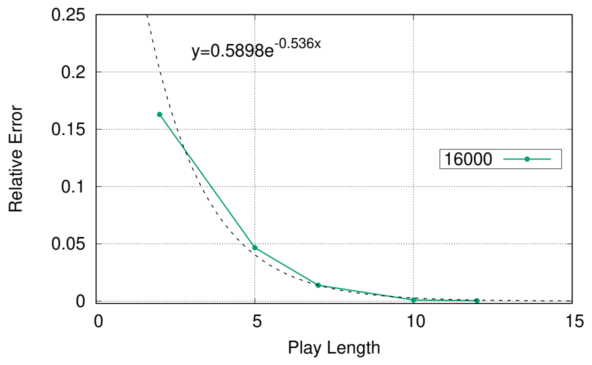

Figure 3: Relative error of the centrality for a small-world network adjacency matrix with 4,096 nodes as a function of the length of the play. The number of plays was kept fixed at 16,000 plays.

specific decreasing rate depends on the matrix, as it was pointed out above. Moreover, it is worth noting that the overall error seems to reduce exponentially fast with the length of the plays, as can be seen in the figure after a suitable regression of the numerical results. Obviously, this trend can only be sustained until the statistical error due to the finite number of plays becomes smaller than the error related to the finite length of the plays.

The relative error for the case of the trace of the inverse matrix of a small-world network is presented in Figure 4. The results obtained follow the same behavior as the results for the vector centralities (Figure 2). Consequently, similar conclusions can be drawn. However, since now the relative errors are much smaller, they are strongly affected by the inherent randomness of any MC simulations, being in practice advisable to increase further the number of plays.

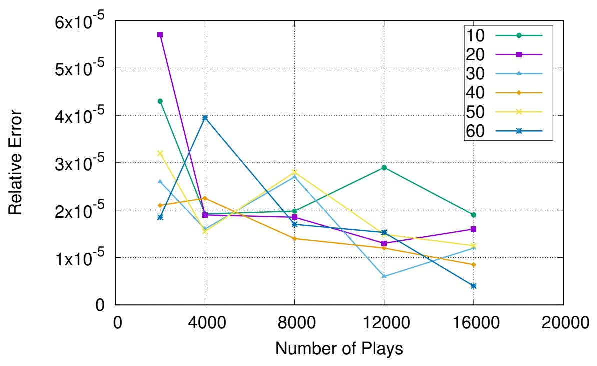

Figure 4: Relative error of the trace of the inverse matrix for a small-world network adjacency matrix with 4,096 nodes as a function of the number of plays. Each curve corresponds to a different length of the random walk.

Table 1: MUMPS execution time (in seconds) for small-world matrices with different sizes.

Processes	Matrix size		
	64,000	128,000	256,000
1	44.94		
2	77.81		
4	39.42	Out of memory	
8	29.49	184.13	
16	21.58	119.89	Out of memory
32	21.06	97.50	511.08
64	27.61	111.36	497.40
128	38.75	160.75	684.34

5.2. Parallel Performance

In Figure 5 the execution time of the distributed Monte Carlo method is plotted as a function of the problem size for two different matrices and varying the number of processes and threads.

Note that the time increases linearly with the size of the matrix, and this occurs independently of the number of process/threads and matrix type. However, it is worth observing that the results are especially better for the case of the Poisson matrix. This is due to the small number of links across the nodes of the network, as well as, because of the inherent homogeneity of its associated network reduces in practice the intercommunication overhead of the parallel code.

To put in perspective the deterministic methods used for comparison, we show in Table 1 that MUMPS suffers from significant scalability problems when increasing either the number of processes or the problem size. For easier comparison with Figure 5, we present in Figure 6 a graph with the bottom three rows of Table 1. It is important to observe that the shape of the curves is very

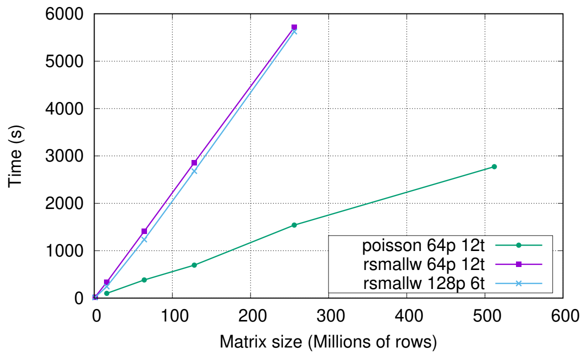

Figure 5: Execution time of DMC as a function of the problem size, for different number of processes and threads, operating on two different matrices: a small-world matrix and a Poisson matrix. In the legend, the type of matrix is stated first, followed by the number of processes used and then the number of threads per process. The values used for n and p were respectively 40 and 4,000.

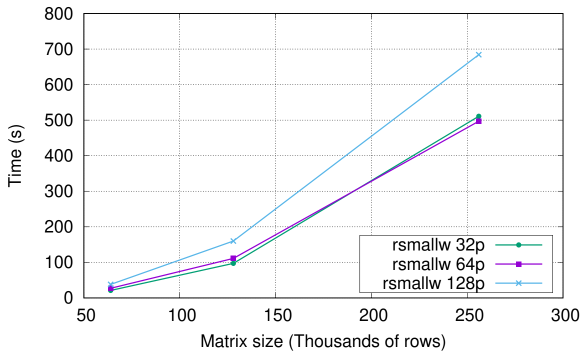

Figure 6: Execution time of MUMPS on small-world matrices as a function of the problem size and for different number of processes.

different, with MUMPS execution time departing clearly from the desired ideal linear behavior. Furthermore, storing the matrix in memory quickly becomes an unaffordable task for MUMPS, becoming in practice impossible to operate on large matrices. These are the reasons why the former plot (DMC) presents results operating on matrices with millions of rows and the latter (MUMPS) on matrices with only thousands of rows.

For the case of Kronecker matrices generated with the Graph500 generator, we have conducted weak scalability tests for DMC (see Table 2). Note that the presence of hubs in these networks might adversely affect the results since these can break the initially established computational load balancing. However, the asymmetric redistribution of nodes, as well as the presence of nodes with no outbound connections, mitigates this issue.

In summary, computing the product of the inverse of a matrix times a vector

Table 2: Execution time of DMC on Kronecker matrices from Graph500 with the number of processes and matrix size scaling with similar rate (weak scalability).

Size	N. Processes	Time DMC (s)
0.85 M	8	85
1.6 M	16	82
11 M	107	101
41 M	400	127

the DMC implementation is highly scalable and produces the correct results up to the desired accuracy. Since the iterative methods are extremely fast at solving such kind of problems they are in general the method of choice. However, resolvent-based metrics require the calculation of a parameter (α) such that the centrality scores obtained from the Katz centrality vector $(I - \alpha A)^{-1}\mathbf{1}$ and the action of the exponential of the adjacency matrix $e^{A}\mathbf{1}$ are similar. Such parameter tends to lead to linear systems that are potentially ill-conditioned. Although ill-conditioning has essentially no effect on the suitability of the computed centralities for ranking, iterative methods are not likely to be successful [41]. However, we observed when running our simulations with the DMC method that no preconditioning of the adjacency matrices was required to guarantee convergence, and in such a case, the highly scalable DMC can provide a very convenient alternative.

The same method can be used to compute the trace of the inverse matrix. In fact the trace can be computed as follows

$$\mathrm{Tr}(A^{-1}) = \sum_{i=1}^{N} e_i^T x_i, \tag{8}$$

where N is the size of the matrix, and $e_1, e_2, \ldots, e_N$ are the vectors of the canonical orthonormal basis, being e_i the vector whose i^{th} component is equal to 1 and the rest are equal to 0. Here x_i is the solution of the linear algebra problem $Ax_i = e_i$, which can be obtained iteratively by means of the GMRES method. Note that the solution of the problem $Ax_i = e_i$ corresponds to obtaining the i^{th} column of the inverse matrix. However, in order to compute the trace of the inverse matrix, it is required to evaluate every term in Eq. (8), which in practice requires to compute each column of the inverse matrix. Although the simulation of each term is completely independent of each other, and as such, can be perfectly parallelized, the overall procedure turns out to be still computationally costly. Nevertheless, we can easily estimate theoretically the total processing time, since computing each column should take approximately the same time independently of the column. For this purpose, we obtain the time T_b to compute a batch of b columns of the inverse matrix running PETSc sequentially with a single process, and then estimate the full theoretical serial time simply multiplying the size of the matrix by T_b/b. For this specific case, the corresponding parallel time can be readily theoretically estimated simply

Table 3: Elapsed computational time of PETSc and of DMC when computing the trace of the inverse of small-world matrices (M indicates 2^{20}), and running both codes in parallel with 768 cores.

Size	Time PETSc(s)	Time DMC (s)
1 M	40	14
4 M	970	68
16 M	14,378	244
64 M	303,274	1237

Table 4: Execution time of DMC on 768 cores when computing the action of the inverse matrix over several result vectors using a small-world matrix with a size of 16 million rows.

Number of vectors	Time (s)
1	262
2	318
4	476
8	674

dividing the total serial time by the number of cores. To compare with the performance of the DMC method, we have run in parallel with 768 cores and setting the same absolute and relative error as the GMRES method, which is of order 10^{-6} and 10^{-5}, respectively. The results are shown in Table 3 for the specific case of the inverse of a small-world matrix with different sizes.

Therefore, for computing the trace of the inverse matrix the MC method clearly outperforms the deterministic algorithm, with striking differences in performance. Furthermore, the PETSc implementation used exhibits major flaws when dealing with very large matrices, while our approach is able to effectively tune the MC method for dealing with distributed matrices.

As it was mentioned in Section 2.4, our MC algorithm can be used as well to compute the action of the inverse matrix over several different vectors, and this can be done during a single execution. The efficiency of the method for dealing with this problem is shown in Table 4 through the parallel execution times obtained using 768 cores when computing the solution for several input vectors. Moreover, this approach can be used to compute weighted centralities according to different sets of weights in a single execution. This can be useful in order to apply successfully the Woodbury formula (see Section 2.4). Note that the computational cost increases weakly with the number of vectors used, being therefore much more convenient to use this algorithm for computing several vectors in a single run than computing them independently in multiple runs.

Furthermore, the proposed approach can be readily modified to allow the computation of other matrix functions. In practice, this would allow the algorithm to compute several functions of the input matrix during a single execution. For instance, still in the context of the analysis of complex networks [42] considers the calculation of the exponential matrix, which characterizes another metric of paramount importance, the overall communicability of the network.

6. Conclusion

This paper proposes a distributed implementation of a Monte Carlo method that computes the solution of linear systems of equations. The method requires generating suitable Markov chains that evolve jumping randomly through the indices of an input matrix, and averaging numerical approximations of the different powers of the matrix. By distributing the input matrix through computational nodes, our method extends the reach of the size of the problems that can be handled. However, this creates a potentially large communication overhead as the Markov chains used in Monte Carlo may now need to migrate between computational nodes frequently. We have effectively solved this problem both by minimizing the number of messages exchanged through appropriately sized buffers, and by overlapping these message exchanges with computation.

We have applied our method in practice to compute several network metrics, such as the Katz centrality and the trace of the inverse of a matrix. The algorithm has been implemented in an efficient parallel code capable of handling very large matrices, of sizes far beyond the available memory of a single machine. The algorithm was tested simulating several examples consisting of different complex networks, and run on a high performance supercomputer equipped with a high speed network interconnecting nodes with several cores each.

Furthermore, we have shown that when computing the trace of the inverse matrix, within about the same error, the Monte Carlo method can be much faster than the alternative deterministic methods. In addition, the method can be readily adapted to compute other matrix functions, provided that such functions can be expanded through a convergent power series, as it happens for instance for the exponential matrix. Finally, it is worth pointing out that although typically the convergence to the solution of any Monte Carlo method is slow (being the approximated solution obtained with low accuracy within a reasonable computational cost), when necessary the accuracy can be easily improved by resorting to parallel computation since the method is based on independent calculations. For a fast low-accuracy estimation of the solution, the method stands out as an efficient alternative to the deterministic methods, especially as it has been shown for dealing with large-scale problems.

References

[1] N. Higham, A. H. Al-Mohy, Functions of matrices: Theory and Computation, SIAM (2008).

[2] A. H. Al-Mohy, N. Higham, Computing the Action of the Matrix Exponential with an Application to Exponential Integrators, SIAM J. Sci. Comput. 3 (2011) 488–511.

[3] J. Demmel, M. Hoemmen, M. Mohiyuddin, K. Yelick, Minimizing communication in sparse matrix solvers, Proceedings of the ACM/IEEE Supercomputing SC09 Conference (2009).

[4] J. Demmel, L. Grigori, M. Hoemmen, J. Langou, Communication-optimal parallel and sequential QR and LU factorizations, SIAM J. Sci. Comput. 34 (2012) A206–A239.

[5] G. E. Forsythe, R. A. Liebler, Matrix Inversion by a Monte Carlo Method, Mathematical Tables and Other Aids to Computation 4 (31) (1950) 127–129.

[6] H. Ji, M. Mascagni, Y. Li, Analysis of Markov Chain Monte Carlo Linear Solvers Using Ulam–von Neumann Algorithm, SIAM Journal on Numerical Analysis 51 (2013) 2107–2122.

[7] M. Benzi, T. M. Evans, S. P. Hamilton, M. Lupo Pasini, S. R. Slattery, Analysis of monte carlo accelerated iterative methods for sparse linear systems, Numerical Linear Algebra with Applications 24 (3) (2017) e2088. `doi:10.1002/nla.2088`.

[8] I. Dimov, S. Maire, J. Sellier, A new Walk on Equations Monte Carlo method for solving systems of linear algebraic equations, Applied Mathematical Modelling 39 (2015) 4494–4510.

[9] I. T. Dimov, V. Alexandrov, A. Karaivanova, Parallel resolvent Monte Carlo algorithms for linear algebra problems, Mathematics and Computers in Simulation 39 (2015) 25–35.

[10] D. Balcan, H. Hu, B. Goncalves, P. Bajardi, C. Poletto, J. J. Ramasco, D. Paolotti, N. Perra, M. Tizzoni, W. Van den Broeck, V. Colizza, A. Vespignani, Seasonal transmission potential and activity peaks of the new influenza A(H1N1): a Monte Carlo likelihood analysis based on human mobility, BMC Medicine 7 (1) (2009) 45. `doi:10.1186/1741-7015-7-45`.

[11] M. Panteli, C. Pickering, S. Wilkinson, R. Dawson, P. Mancarella, Power System Resilience to Extreme Weather: Fragility Modeling, Probabilistic Impact Assessment, and Adaptation Measures, IEEE Transactions on Power Systems 32 (5) (2017) 3747–3757. `doi:10.1109/TPWRS.2016.2641463`.

[12] C. M Schneider, A. Moreira, J. S Andrade, S. Havlin, H. J Herrmann, Mitigation of malicious attacks on networks, Proceedings of the National Academy of Sciences 108 (10) (2011) 3838–41.

[13] F. C. Santos, J. M. Pacheco, Scale-Free Networks Provide a Unifying Framework for the Emergence of Cooperation, Phys. Rev. Lett. 95 (2005) 098104. `doi:10.1103/PhysRevLett.95.098104`.

[14] M. Newman, Networks : an introduction, Oxford University Press, Oxford New York, 2010.

[15] E. Estrada, D. J. Higham, Network properties revealed through matrix functions, SIAM Rev 52 (4) (2010) 696–714.

[16] D. J. Watts, S. H. Strogatz, Collective dynamics of 'small-world' networks, Nature 393 (6684) (1998) 440–442. doi:10.1038/30918.

[17] J. Leskovec, D. Chakrabarti, J. Kleinberg, C. Faloutsos, Z. Ghahramani, Kronecker Graphs: an Approach to Modeling Networks, J. Mach. Learn. Res. 11 (2010) 985–1042.

[18] Graph500, https://graph500.org, accessed: 2021-05-31.

[19] Y. Saad, Iterative methods for sparse linear systems, SIAM, Philadelphia, 2003.

[20] K. Atkinson, An introduction to numerical analysis, Wiley, New York, 1989.

[21] P. Amestoy, I. Duff, J.-Y. L'Excellent, Multifrontal parallel distributed symmetric and unsymmetric solvers, Computer Methods in Applied Mechanics and Engineering 184 (2) (2000) 501–520. doi:https://doi.org/10.1016/S0045-7825(99)00242-X.

[22] R. Fletcher, Conjugate Gradient methods for indefinite systems, in: G. A. Watson (Ed.), Numerical Analysis, Springer Berlin Heidelberg, Berlin, Heidelberg, 1976, pp. 73–89.

[23] Y. Saad, M. H. Schultz, GMRES: A Generalized Minimal Residual Algorithm for Solving Nonsymmetric Linear Systems, SIAM Journal on Scientific and Statistical Computing 7 (3) (1986) 856–869. doi:10.1137/0907058.

[24] H. A. van der Vorst, Bi-CGSTAB: A Fast and Smoothly Converging Variant of Bi-CG for the Solution of Nonsymmetric Linear Systems, SIAM Journal on Scientific and Statistical Computing 13 (2) (1992) 631–644. doi:10.1137/0913035.

[25] J. S. Rosenthal, Parallel computing and Monte Carlo algorithms, Far East Journal of Theoretical Statistics 4 (1999) 207–236.

[26] W. R. Gilks, Markov chain Monte Carlo in practice, Chapman & Hall, London, 1996.

[27] L. Katz, A new status index derived from sociometric analysis, Psychometrika 18 (1) (1953) 39–43. doi:10.1007/BF02289026.

[28] C. H. Hubbell, An Input-Output Approach to Clique Identification, Sociometry 28 (4) (1965) 377–399.

[29] C. Klymko, Centrality and Communicability Measures in Complex Networks : Analysis and Algorithms, Ph.D. thesis, Emory University (2013).

[30] S. A. Gershgorin, Uber die abgrenzung der eigenwerte einer matrix, Izv. Akad. Nauk. SSSR Ser. Mat (6) (1931) 749–754.

[31] J. A. de la Peña, I. Gutman, J. Rada, Estimating the Estrada index, Linear Algebra and its Applications 427 (1) (2007) 70 – 76. doi:https://doi.org/10.1016/j.laa.2007.06.020.

[32] E. Estrada, J. A. Rodríguez-Velázquez, Subgraph centrality in complex networks, Physical Review E 71 (5) (May 2005). doi:10.1103/physreve.71.056103.

[33] H. Avron, Counting triangles in large graphs using randomized matrix trace estimation, Workshop on Large-scale Data Mining: Theory and Applications 10 (2010) 10,9.

[34] J. Sherman, W. J. Morrison, Adjustment of an Inverse Matrix Corresponding to a Change in One Element of a Given Matrix, The Annals of Mathematical Statistics 21 (1) (1950) 124–127. doi:10.1214/aoms/1177729893.

[35] M. A. Woodbury, Inverting modified matrices, Statistical Research Group, Memo. Rep. no. 42, Princeton University, Princeton, N. J., 1950.

[36] Paraver, https://tools.bsc.es/paraver, accessed: 2021-05-31.

[37] MATLAB, (R2020a), The MathWorks Inc., Natick, Massachusetts, 2020.

[38] P. R. Amestoy, I. S. Duff, J.-Y. L'Excellent, J. Koster, A Fully Asynchronous Multifrontal Solver Using Distributed Dynamic Scheduling, SIAM Journal on Matrix Analysis and Applications 23 (1) (2001) 15–41. doi:10.1137/S0895479899358194.

[39] S. Balay, S. Abhyankar, M. F. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, D. A. May, L. C. McInnes, K. Rupp, B. F. Smith, S. Zampini, H. Zhang, H. Zhang, PETSc Web page, http://www.mcs.anl.gov/petsc (2019).

[40] A. C. Berry, The Accuracy of the Gaussian Approximation to the Sum of Independent Variates, Transactions of the American Mathematical Society 49 (1) (1941) 122. doi:10.2307/1990053.

[41] M. Aprahamian, D. J. Higham, N. J. Higham, Matching exponential-based and resolvent-based centrality measures, J. Complex Networks 4 (2) (2016) 157–176.

[42] J. A. Acebrón, J. R. Herrero, J. Monteiro, A highly parallel algorithm for computing the action of a matrix exponential on a vector based on a multilevel monte carlo method, Computers & Mathematics with Applications 79 (12) (2020) 3495–3515. doi:https://doi.org/10.1016/j.camwa.2020.02.013.