Distributed algorithm for the analysis of properties of complex networks

Filipe Pedro Guerra Magalhães

Instituto Superior Técnico, Lisboa, Portugal,
filipepgmagalhaes@tecnico.ulisboa.pt,

Abstract. This paper proposes a Monte Carlo method to compute metrics of complex networks, such as the centrality of nodes in a network. The proposed method uses random walks over a matrix that represents the network, to calculate functions based on the powers of the matrix. In particular, several interesting metrics can be derived, such as the inverse of the matrix multiplied by a vector, corresponding to Katz centrality, and the trace of the inverse of the matrix. The proposed method aims to be parallel and scalable, in order to deal with large networks, that may exceed the memory limits of individual machines.

Keywords: parallelism, Monte Carlo, matrix inversion, network metrics

1 Introduction

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. It has been important to model disease spreading, predicting how epidemics progress [1] , analyzing the resilience of networks, such as the power grid, to failures both random [2] and maliciously targeted [3]. It is also fundamental in the analysis of social networks, examining the dynamics of personal opinions, collaboration and habits [4].

One fundamental metric in the study of networks is node centrality. Centrality can be interpreted as how close to the rest of the network it is, how influential it is or how important it is in establishing connections between other nodes in the network. To meet these different ideas, there are several formal definitions of centrality [5]. Influence, for example, is important in the study of social networks, representing the relative importance of individuals in the network over their peers, as well as in the world wide web, where modern web search engines are based on this metric. This notion of centrality is made concrete by the Eigenvector Centrality, Katz centrality and Google's PageRank. The proposed method will compute some important metrics in high performance computing settings.

The high performance machines now in the service of academia and industry, the supercomputers, are essentially multiprocessor machines (nodes) interconnected by high speed networks. In order to explore these parallel oriented machines, there are several frameworks. This work will explore the academia standards, OpenMP (Open Multi Processing) [6] for multithreading and MPI (Message Passing Interface) for interprocess communication [7].

2 Networks

Formally, a network is a graph $G =< V, E >$, composed of a set of nodes V, and a set of edges E, each edge an unordered pair of nodes [5]. Graphs can be represented in the form of a square matrix, the adjacency matrix, by labeling each node with a number, so that each position of the matrix represents the edge between the node corresponding to its row number and the node corresponding to its column number [5]. This convention allows us to represent, if necessary, the direction and weight of edges.

There are several strategies to store sparse matrices in memory. The Compressed Sparse Row (CSR) or Yale Format can be used [8], which represents a matrix using three vectors, storing only the non-zero entries. One vector contains the number of non-zero entries above each row (IA). The other two have, for each entry ordered left-to-right and top-to-bottom, the column of the entry (JA) and its value (AA).

In order to generate artificial networks that emulate real networks, several models exist. These are useful, for example, to test algorithms over networks with well defined properties. This work will use small world matrices generated with a variation of the Watts–Strogatz method [9] and Kronecker matrices[10], generated with Graph 500[11].

To characterize a network and its nodes, there is a vast collection of metrics that can be used [5]. Following the notion of centrality as the influence of a node on other nodes in the network, there is Katz centrality [12]. This notion of centrality can be defined in terms of the adjacency matrix B as

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

with α as an adjustable real parameter [13] and $\mathbf{1}$ as a vector with 1 in all entries. $K(\alpha)$ is a vector with each node's centrality. This parameter α, designated attenuation factor, should be smaller than $1/\lambda$, λ being the largest eigenvalue of B [12]. It allows this centrality to be tuned to extract slightly different information. Notably, it is shown in [13] that, as α tends to 0, Katz centrality approaches the value of the degree centrality. Additionally, as α tends to $1/\lambda-$, Katz centrality approaches the value of eigenvector centrality. It should be noted that this vector with 1 in all entries can be modified to have different values, giving weights to the contribution of each node to the centrality.

There are a great number of other network metrics based on matrix functions. For example, the diagonal of the matrix exponential represents the exponential subgraph centrality [14], the trace of the matrix exponential is known as the Estrada Index [15], 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 [16]. This work will focus Katz centrality and the trace of the inverse of the matrix, but it would require minor adaptations to focus other of the many metrics available.

Evolving networks In order to calculate the inverse of a matrix knowing the inverse of another, we can use the Sherman-Morrison formula[17] or its generalization, the Woodbury formula[18]. Let us consider an original matrix A, of size N by N, and a second matrix 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 $A + UV^t$ are invertible, the new inverse is given by the following:

$$B^{-1} = A^{-1} - A^{-1}U(I_k + VA^{-1}U)^{-1}VA^{-1} \tag{2}$$

Using this formula, we can compute metrics based on the inverse of the matrix, for an evolving network, based in a previously obtained inverse.

3 State-of-the-Art Algorithm for Systems of Linear Algebraic Equations

The problem of solving a system of linear equations, can be written in matrix form, considering a vector x, with the variables, a matrix A with the coefficients of each variable, arranged to represent one equation per row, and a vector b, with the independent term of each equation [19].

$$Ax = b \Rightarrow x = A^{-1}b \tag{3}$$

This allows us to solve the system of linear equations by inverting the matrix and multiplying by the vector b, as long as we are dealing with a square and invertible matrix. To solve Systems of Linear Algebraic Equations (SLAE), there are several classes of methods, with a huge number of variants.

Direct methods Direct methods, such as the Gauss-Jordan elimination [20] or the frontal solver and its more complex parallel implementation, the multifrontal solver [21], provide an exact solution, under the limits of the hardware. However, dense matrix operations are involved, incurring in large storage costs.

Iterative methods Iterative methods are applied iteratively, resulting in incrementally more accurate solutions, based on an initial guess [22], a rough approximation. Some of the simplest iterative methods are Richardson's, Jacobi and Gauss-Seidel methods [19]. The most successful algorithms are based in the use of Krylov subspaces. These perform matrix-vector operations in each iteration. There is a great number of algorithms under this category, from which we can highlight the biconjugate gradient method [19], GMRES [23] and BICGstab [24].

Monte Carlo algorithms Monte Carlo methods are a class of algorithms, that rely on repeated random sampling to approximate solutions to a wide range of problems [25]. To improve the accuracy of the solution, it is in general necessary to increase the number of random samples taken, the greater the number, the better the accuracy [25]. Markov Chain Monte Carlo (MCMC) methods are a subclass of Monte Carlo methods, based on a Markov Chain with a desired distribution as a stationary distribution [26]. These 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 its solutions depend on it. Iterative Monte Carlo algorithms use more traditional iterative algorithms alongside with Monte Carlo components, generating two types of error, stochastic error and systematic error. The Monte Carlo method is here used in each iteration of the iterative method, to improve its results and reduce the number of iterations [27].

4 Proposed Solution

The proposed solution is based on the method described in [28]. Considering a matrix B, 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 is held:

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

In order to approximate B^{-1}, the method under study approximates the sum the of first n powers. The method calculates an approximation R to

$$R = \sum_{k=0}^{n} A^k \tag{6}$$

This is a good approximation for a large enough n, since it converges as n tends to infinity. The method calculates independently each row of each power of the matrix A, using random walks, similarly to Markov Chain Monte Carlo. In order to calculate the approximation of the l-th row of A^k, the k-th power of matrix A, the method builds a vector r with the same length as the row, initially with 0 in all entries, as described in Algorithm 1.

Algorithm 1 Monte Carlo Matrix Inversion

```
1: function InvertMatrix($B$, $n$, $p$)
2:     $I \leftarrow$ IdentityMatrix(length of $B$)
3:     $A \leftarrow I - B$
4:     for $i \leftarrow 0$ to length of $A - 1$ do
5:         $R[i] \leftarrow$ [0, . . . , 0]
6:         for $k \leftarrow 0$ to $n$ do
7:             $R[i] \leftarrow R[i] +$ CalculateRow($A, k, i, p$)
    return $R$
```

A number of "plays" p start from this row i, each with k steps. A "play" is no more than a Markov Chain with an associated numeric value, which, at each step, chooses one entry ($A_{i,j}$) of its current row i, and moves to the row with index j, as described in Algorithm 2. The choice of entry is made at random, with probabilities for each entry of the row weighted proportionally to their absolute value.

Algorithm 2 Monte Carlo Power Row Simulator

```
1: function CalculateRow($A$, $k$, $i$, $p$)                    ▷ Calculate row $i$ of the $k$-th power of $A$
2:     r[0, length of A - 1] ← [0, ..., 0]
3:     for $x \leftarrow 1$ to $p$ do                                ▷ Use $p$ plays in the approximation
4:         value ← 1
5:         currentRow ← $i$
6:         for $y \leftarrow 0$ to $k-1$ do
7:             value ← value × GetRowWeight($A$, currentRow)
8:             selectedColumn ← RandomWeightedChoice($A, i$)
9:             currentRow ← selectedColumn
10:        r[currentRow] ← r[currentRow] + value
11:    for $j \leftarrow 0$ to length of $r$ do
12:        r[j] ← r[j]/p
       return $r$
```

A play starts with the value of 1 and, at each step, multiplies it by the sum of the values of the entries in the current row ($\sum_j A_{i,j}$). When a play reaches the end of its k steps, its value is added to the vector r, at the index of the row it is currently at (r_i). In the end, the values in r must be divided by the number of plays, yielding an approximation of the desired i-th row of A^k. If a row of the input matrix has no non-zero entries, and therefore no outbound connections, any random walk reaching it will terminate there. To speed up computation, the sum of all entries in each row is initially calculated and stored in a vector. The matrix representation in memory is similar to CSR, with this added information.

In order to calculate A^{k+1}, plays of length $k+1$ are used. These, by definition, contain plays of length k, which can be used in their own right to calculate A^k. This way we can use a significantly smaller total number of steps, pn steps instead of $p(n^2+n)/2$. However, this change comes at the cost of making the approximation of each power dependent on the approximation of the lower powers. This can lead to larger errors, but experiments showed only a small increase in the error, and a very beneficial reduction in execution time.

Computation of the centrality When the objective is simply to calculate the product of a vector by the inverse of the matrix ($B^{-1}v$), a vector c, which is the case both in 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 need only to sum all entries of the resulting vector, when calculating one row of the inverse, saving memory space.

4.1 Parallel implementation

The goal of this work lies in developing a scalable parallel method. The calculation of each row is totally independent from the others, and therefore trivially parallelizable by computing them in different processing units. However, it assumes access to the entire matrix. Should a matrix be too large for the memory of a single machine, we need to distribute the computation to several machines, where each machine keeps a part of the matrix in memory. This form of parallelization requires plays to start in one machine, and be transferred over to another if necessary. Each machine should start computing the plays belonging to the rows held in memory, and, when they reach a row in the memory of another machine, send a message with the information of the play to be continued there, as described in Algorithm 3.

Communication should be performed in parallel with computation, in order to fully use the available resources. Messages between machines may have a number of plays aggregated in them,

Algorithm 3 Parallel Monte Carlo Centrality Simulator

```
 1: function WORKER($A$)
 2:     $r[0, \text{length of} A] \leftarrow [0, \ldots, 0]$                ▷ Array of numbers
 3:     while working do
 4:         if INCOMINGMESSAGE( ) then
 5:             $play \leftarrow$ RECEIVEPLAY()
 6:         else if UNGENERATEDPLAYS( ) then
 7:             $play \leftarrow$ GENERATEPLAY()
 8:         if $play$ is defined then
 9:             PROCESSPLAY($A, play, r$)
10:     for $i \leftarrow 0$ to length of $r - 1$ do
11:         $r[i] \leftarrow r[i]/p$
12:     return REDUCE($r$)                ▷ Sum the results gathered by all machines
13: function PROCESSPLAY($A$, $pl$, $r$)
14:     while $pl.stepsLeft > 0$ and ISINMEMORY($pl.currentRow$) do
15:         $pl.stepsLeft \leftarrow pl.stepsLeft - 1$
16:         $pl.value \leftarrow pl.value \times$ GETROWWEIGHT($A, pl.currentRow$)
17:         $pl.currentRow \leftarrow$ RANDOMWEIGHTEDCHOICE($A, pl.currentRow$)
18:         $r[pl.startRow] \leftarrow r[pl.startRow] + pl.value$
19:     if $pl.stepsLeft > 0$ then
20:         SEND($pl$)
```

and the receiving end must have a buffer to collect plays to process. These techniques help improve performance, but must be tuned to the execution in machines with different characteristics.

In each machine, heterogeneous threads are used, where one in each process handles communication tasks such as receiving messages and managing associated buffers, while the others exclusively process plays.

4.2 Alternative applications

The proposed 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, with very small modifications. This can be useful for computing other network metrics and different definitions of centrality [13].

The trace of a 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. 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, 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 18 in Algorithm 3) 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 18 for each of the vectors, storing the result in separate output vectors. This way, we can, in a single execution, compute all the vectors necessary to apply the Woodbury formula, or the centrality according to different node weights.

5 Evaluation

The implementation is evaluated under precision and performance. The relevant computation steps in the Monte Carlo method to calculate the trace and the centrality vector are the same, therefore they are not analyzed in separate regarding performance, only regarding precision. The topologies used will be custom small world matrices and Graph500 matrices. As alternatives to compare against, a direct method, MUMPS [29] with the multifrontal method, and an iterative method, GMRES implemented by PETSc [30], will be considered.

5.1 Precision

Precision will be measured by comparing the results obtained with the result of a method capable of reaching the maximum precision allowed by the machine, which is Matlab's direct method. When calculating the vector of centralities, the error of the resulting vector in relation to the reference vector will be measured based on the relative differences between each corresponding entry. The presented values are the average of all the rows from this vector of errors, averaged over several (6) repetitions. To achieve the best accuracy, it is necessary to adjust the number of plays p and length of the plays n, as described in Section 4, and evaluate the behavior of the method, as these parameters are adjusted.

The implementation has two sources of error, the number of powers of the matrix calculated n, and the precision with which each is calculated, controlled by parameter p. The precision of each individual power is the direct result of sampling, and therefore is an error that can be controlled predictably[31], knowing that the error of the estimate is in the order of $1/\sqrt{p}$. The number of powers computed, related to the length of the random walk, is a source of error as well. This error grows when the number of powers computed is smaller. Since the power series converges (per the requirements laid out in Section 4), powers of higher order should have a small impact on the error. The precise impact depends on the convergence of the particular matrix chosen.

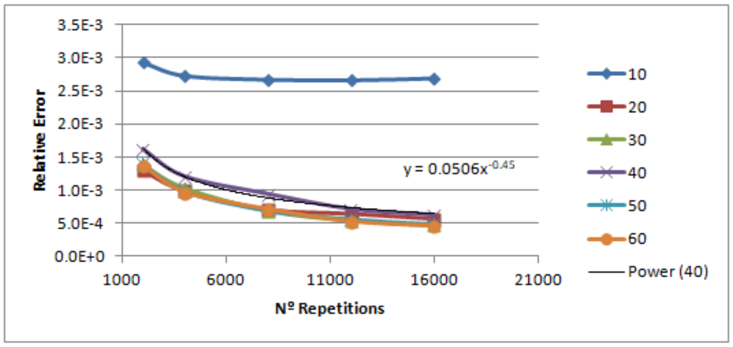

Fig. 1. Relative error of the centrality of the small world matrix with 4096 nodes.

We can observe a similar downwards pattern for all play lengths (Figure 1) when Increasing the repetitions. We can observe the downwards slope in error as it would be expected of the Monte Carlo approximation, presenting a good approximation to the expected value of 0.5, in the power of the trendline. The approximation error can be attributed to the fact we are only computing a finite number of powers and to the error caused by aggregating the computation of several powers in the same plays, as described in Section 4.

When the length of the plays is increased, the error reduces (Figure 2), as expected, until the error caused by the number of repetitions is higher than the error caused by the length of the plays. The rate at which the error decreases depends on the type of matrix. It is worth noting that, as long as the number of repetitions allows, the error can be reduced exponentially just by increasing the play length, as indicated by the expression of the trend-line.

Results in the Graph 500 matrix of size 4819 also present a downwards pattern, although less pronounced. The relative error is also much smaller, in the order of 100 times smaller. Since these matrices have a much higher normalization factor, the first few powers will be a lot more impactful then the others, and increasing the length of the plays will not help. For the same reason, the results when calculating the trace of the inverse for the Graph 500 matrix have a small relative error, most of the value is given by the first powers. Increasing the length of the random walks doesn't show a clear boost in precision.

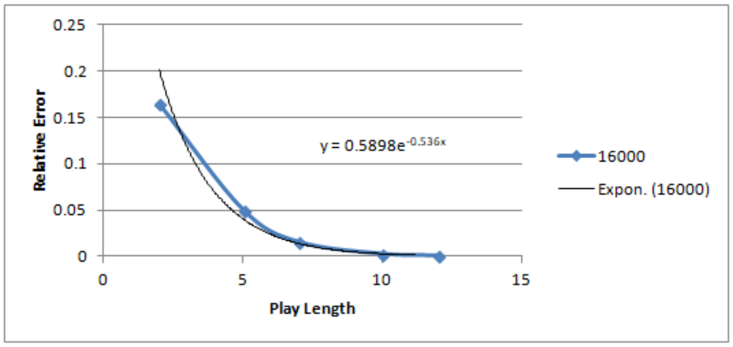

Fig. 2. Relative error of the centrality of the small world matrix with 4096 nodes, using 16000 repetitions.

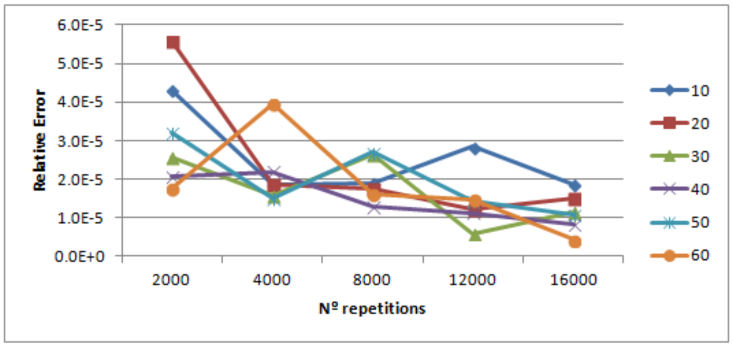

Fig. 3. Relative error of the trace of the inverse of the small world matrix with 4096 nodes.

In the small world matrices, when calculating the trace, the behavior is similar to when calculating the vector of centralities, but the downwards pattern is less clear (Figure 3). The precision is greater, and the pattern is influenced by the noise of the random calculations.

MUMPS, being a direct method, computes the result vector with the maximum precision allowed by the machine.

PETSc has a configurable error threshold. Its results are comparable to the precision achieved with Monte Carlo, in the less precise error threshold.

5.2 Performance

When scaling the problem size over the same number of processes (Figure 4), the execution time scales linearly, as expected. Poisson matrices are particularly forgiving, as there is a small number of connections between rows of the matrix in different processes, reducing communication.

Likewise, when the problem size is kept constant and the number of processes increases, the execution time drops. The efficiency is not perfect here, as it can be observed in Figure 5, where the growth of the speedup does not hold for larger numbers of processes. This is to be expected as adding more processes increases the communication, which slows down the computation. However, as the objective is to distribute the matrix in order to store it in memory, even with very large matrices, it won't be necessary to use that many processes. After the matrix fits in memory of a certain number of processes, the computation can just be replicated in an embarrassingly parallel fashion, allowing for a perfect scalability, for large enough problems. A weak scaling analysis of the algorithm shows a decreasing efficiency as the number of processes increases, since communication increases as well. When the communication is negligible, such as the Poisson matrix, the efficiency scales much better.

Graph 500 matrices show similar patterns. While hubs could be a problem for work load balancing, an asymmetric distribution of nodes and the presence of nodes with no outbound connections compensate for the additional difficulty.

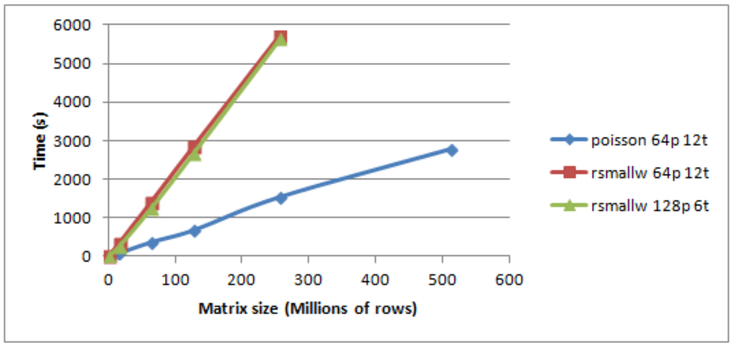

Fig. 4. Execution time of the implementation while increasing problem size, with different number of processes and threads over small world and Poisson matrices (Mare Nostrum). The type of matrix is stated first, followed by the number of processes used and then the number of threads per process.

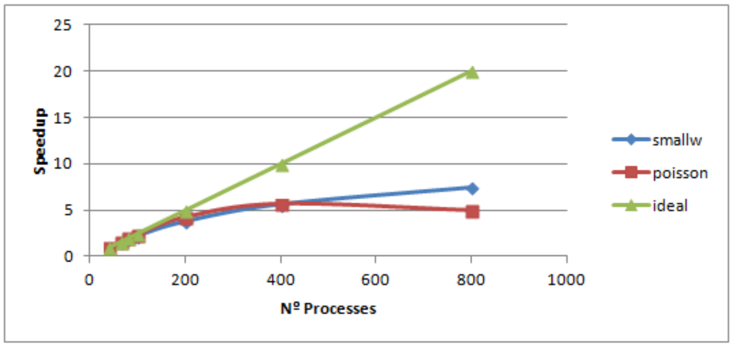

Fig. 5. Speedup of the implementation, compared to the execution with 40 processes, over matrices of size 16M and 12 threads, while increasing the number of processes over small world and Poisson matrices (Mare Nostrum).

MUMPS encounters problems with scaling using several processes.Increasing the problem size is equally problematic, as the scaling is far from linear. Furthermore, storing the matrix in memory quickly becomes a problem for MUMPS, making it impossible to explore large matrices.

Table 1. PETSc execution time while computing the trace of the inverse. Only the first 512 rows are computed, total serial time is an estimation.

size	time (s)	estimated full serial time (s)
1 M	15	30,720
4 M	91	745,472
16 M	337	11,042,816
64 M	1777	232,914,944

PETSc's GMRES implementation is extremely fast calculating the vector of centralities, as it needs very few iterations in order to calculate it with an acceptable precision. However, in order to calculate the trace of the inverse, it needs to calculate every row of the inverse. The execution for each row is completely independent, and as such, can be perfectly parallelized. The times shown (Table 1) were taken from a single process computing the first 512 rows of each matrix. Since each row should take approximately the same time to compute, and can be perfectly parallelized, we can multiply the obtained time in order to get the total processing time. For an error margin comparable to the Monte Carlo method, the time taken is much larger. For example, over the small world network with 16 million nodes, PETSc is estimated to take $11,042,816$ seconds, using a single process. Monte Carlo is estimated to take $31,269$ seconds, when using a single process.

When computing the trace of the inverse, the Monte Carlo method shows a clear performance advantage. Furthermore, the PETSc implementation used encountered some problems with very large input matrices, while the Monte Carlo method distributes large matrices with ease.

Table 2. Execution time when computing several result vectors.

N vectors	Time (s)
1	261.77
2	317.79
4	476.46
8	674.48

As laid out in Section 4.2, the Monte Carlo method can be used to compute the result of the inverse multiplied by several different vectors, at the same time, in a single execution. The Monte Carlo method is efficient at calculating several results at the same time, as shown in Table 2. This can be used to compute weighted centralities according to different sets of weights, in a single execution.

If the objective is to use the Woodbury formula (Section 2), and we need more than a few vectors to describe the differences between matrices, it is faster to calculate the inverse of the new matrix from scratch. This is true for a large enough modification, where the time taken to calculate several vectors will surpass the cost of running the algorithm twice. The use of this technique is still useful for modifications described by a small number of vectors.

The efficient computation of several results at the same time is a very useful property of the Monte Carlo method not present in the alternatives. It is shown here that it is efficient to compute the inverse multiplied by several different vectors, but, perhaps more importantly, the techniques used can be modified to allow the computation of other matrix functions. This would allow the Monte Carlo method to, with a single execution, compute several different functions of the input metric, such as the trace of the inverse and several centrality measures based both in the inverse of the matrix or in the exponential.

6 Conclusion

This work proposed a Monte Carlo method to compute metrics of complex networks, based on an approximation of the powers of a matrix. From this information, approximations to several interesting metrics can be derived, such as Katz centrality and the trace of the inverse of the matrix. This implementation can deal with very large matrices, and surpass individual machine's memory limitations. It was tested over different types of networks, with small world properties, simulating real complex networks, and executed in supercomputers featuring high speed networks interconnecting nodes with several processors each.

When computing the trace of the inverse, Monte Carlo can be much faster than alternative methods, for a comparable error. It is also adaptable to other functions that can be obtained by computing the powers of a matrix, such as the exponential. Additionally, the Monte Carlo computation can be replicated any number of times, increasing the precision in a predictable way and scaling perfectly. The existing implementation of this method is therefore suitable for very large problems, where an estimation of the result is desired.

References

1. 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, and 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*, vol. 7, p. 45, Sep 2009.

2. M. Panteli, C. Pickering, S. Wilkinson, R. Dawson, and P. Mancarella, "Power system resilience to extreme weather: Fragility modeling, probabilistic impact assessment, and adaptation measures," *IEEE Transactions on Power Systems*, vol. 32, pp. 3747–3757, Sept 2017.
3. C. M Schneider, A. Moreira, J. S Andrade, S. Havlin, and H. J Herrmann, "Mitigation of malicious attacks on networks," vol. 108, pp. 3838–41, 02 2011.
4. F. C. Santos and J. M. Pacheco, "Scale-free networks provide a unifying framework for the emergence of cooperation," *Phys. Rev. Lett.*, vol. 95, p. 098104, Aug 2005.
5. M. E. J. Newman, *Networks : an introduction.* Oxford New York: Oxford University Press, 2010.
6. L. Dagum and R. Menon, "OpenMP: an industry standard API for shared-memory programming," *Computational Science & Engineering, IEEE*, vol. 5, no. 1, pp. 46–55, 1998.
7. L. Clarke, I. Glendinning, and R. Hempel, *The MPI Message Passing Interface Standard*, p. 213–218. Birkhäuser Basel, 1994.
8. A. Buluç, J. T. Fineman, M. Frigo, J. R. Gilbert, and C. E. Leiserson, "Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks," in *Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '09, (New York, NY, USA), pp. 233–244, ACM, 2009.
9. D. J. Watts and S. H. Strogatz, "Collective dynamics of "small-world" networks," *Nature*, vol. 393, p. 440–442, Jun 1998.
10. J. Leskovec, D. Chakrabarti, J. Kleinberg, C. Faloutsos, and Z. Ghahramani, "Kronecker graphs: An approach to modeling networks," *J. Mach. Learn. Res.*, vol. 11, pp. 985–1042, Mar. 2010.
11. "Top500." https://graph500.org. Accessed: 2018-06-09.
12. L. Katz, "A new status index derived from sociometric analysis," *Psychometrika*, vol. 18, no. 1, pp. 39–43, 1953.
13. C. Klymko, *Centrality and Communicability Measures in Complex Networks : Analysis and Algorithms.* PhD thesis, Emory University, 2013.
14. E. Estrada and J. A. Rodríguez-Velázquez, "Subgraph centrality in complex networks," *Physical Review E*, vol. 71, May 2005.
15. J. A. de la Peña, I. Gutman, and J. Rada, "Estimating the estrada index," *Linear Algebra and its Applications*, vol. 427, no. 1, pp. 70 – 76, 2007.
16. H. Avron, "Counting triangles in large graphs using randomized matrix trace estimation," *Workshop on Large-scale Data Mining: Theory and Applications*, vol. 10, 08 2010.
17. J. Sherman and W. J. Morrison, "Adjustment of an inverse matrix corresponding to a change in one element of a given matrix," *The Annals of Mathematical Statistics*, vol. 21, p. 124–127, Mar 1950.
18. M. A. Woodbury, *Inverting modified matrices.* Statistical Research Group, Memo. Rep. no. 42, Princeton University, Princeton, N. J., 1950.
19. Y. Saad, *Iterative methods for sparse linear systems.* Philadelphia: SIAM, 2003.
20. K. Atkinson, *An introduction to numerical analysis.* New York: Wiley, 1989.
21. P. Amestoy, I. Duff, and J.-Y. L'Excellent, "Multifrontal parallel distributed symmetric and unsymmetric solvers," *Computer Methods in Applied Mechanics and Engineering*, vol. 184, no. 2, pp. 501 – 520, 2000.
22. G. H. Golub and C. F. Van Loan, *Matrix computations.* Johns Hopkins Univ. Press, 2007.
23. Y. Saad and M. H. Schultz, "Gmres: A generalized minimal residual algorithm for solving nonsymmetric linear systems," *SIAM Journal on Scientific and Statistical Computing*, vol. 7, no. 3, pp. 856–869, 1986.
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*, vol. 13, no. 2, pp. 631–644, 1992.
25. M. Quinn, *Parallel programming in C with MPI and openMP.* Dubuque, Iowa: McGraw-Hill, 2004.
26. W. R. Gilks, *Markov chain Monte Carlo in practice.* London: Chapman & Hall, 1996.
27. M. Benzi, T. M. Evans, S. P. Hamilton, M. L. Pasini, and S. R. Slattery, "Analysis of Monte Carlo accelerated iterative methods for sparse linear systems," pp. 1–30.
28. G. E. Forsythe and R. A. Liebler, "Matrix Inversion by a Monte Carlo Method," *Mathematical Tables and Other Aids to Computation*, vol. 4, no. 31, pp. 127–129, 1950.
29. P. R. Amestoy, I. S. Duff, J.-Y. L'Excellent, and J. Koster, "A fully asynchronous multifrontal solver using distributed dynamic scheduling," *SIAM Journal on Matrix Analysis and Applications*, vol. 23, no. 1, pp. 15–41, 2001.
30. 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, and H. Zhang, "PETSc Web page." http://www.mcs.anl.gov/petsc, 2017.
31. A. C. Berry, "The Accuracy of the Gaussian Approximation to the Sum of Independent Variates," *Transactions of the American Mathematical Society*, vol. 49, no. 1, p. 122, 1941.