TÉCNICO LISBOA

Distributed algorithm for the analysis of properties of complex networks

Filipe Pedro Guerra Magalhães

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisors: Prof. Juan António Acebron de Torres
Prof. José Carlos Alves Pereira Monteiro

Examination Committee

Chairperson: Prof. Daniel Jorge Viegas Gonçalves
Supervisor: Prof. Juan António Acebron de Torres
Members of the Committee: Prof. Alexandre Paulo Lourenço Francisco

June 2018

Abstract

This thesis proposes a Monte Carlo method to compute metrics of complex networks, such as the centrality of nodes in a network. Concrete metrics of networks can be calculated using functions of the matrices that represent them. The proposed method uses random walks over a matrix 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. To this end, state of the art parallelism frameworks and tools are used to distribute the computation across processes and threads on several machines.

The implementation of the Monte Carlo method is executed in two supercomputers featured among the top twenty fastest machines in the world at the moment. These consist on high speed networks interconnecting nodes with several processors, allowing the evaluation of the implementation's parallelism.

The method is tested regarding precision and performance, and compared against state of the art alternatives. It is shown to be highly scalable, taking advantage of large numbers of computing nodes, and therefore being able to tackle large problem sizes. This method is shown to be a viable, high performance method to compute approximations of metrics in large scale networks.

Keywords

Parallelism; Monte Carlo; Matrix Inversion; Network Metrics.

Resumo

Esta tese propõe um método Monte Carlo para o cálculo de métricas em redes complexas, tal como a centralidade de nós numa rede. Métricas de redes podem ser calculadas com base em funções das matrizes que as representam. O método proposto utiliza caminhos aleatórios em matrizes para calcular funções baseadas nas potências da matriz. Em particular, várias métricas interessantes podem ser derivadas, tal como a inversa da matriz multiplicada por um vetor, que corresponde à centralidade de Katz, ou o traço da inversa da matriz.

O método proposto é paralelo e escalável, de modo a lidar com redes de elevadas dimensões, que podem exceder os limites de memória de máquinas individuais. *Frameworks* e ferramentas do estado da arte são utilizadas para distribuir a computação por múltiplos processos e *threads* em várias máquinas.

A implementação do método Monte Carlo é executada em dois supercomputadores, entre as vinte máquinas mais rápidas do mundo, de momento. Estas máquinas consistem em redes de alta velocidade a ligar nós com múltiplos processadores, permitindo uma avaliação do paralelismo da implementação.

O método é testado quanto à precisão e performance, bem como comparado com as alternativas do estado da arte. Demonstra-se que a implementação é altamente escalável, tirando partido de elevados números de nós para abordar problemas de larga escala. Os resultados analisados mostram que o método proposto é uma possibilidade viável e eficiente para calcular aproximações de métricas de matrizes de grande escala.

Palavras Chave

Paralelismo; Monte Carlo; Inversão de Matrizes; Métricas de Redes.

Contents

List of Figures

List of Tables

List of Algorithms

1

Introduction

Contents

In recent years, the study of networks has received a renewed interest from the scientific community. The study of networks blends with a wide range of areas, with networks emerging from social interactions to biological phenomena, without forgetting the world wide web, financial networks, and the power grid. The research in network science has been important to model disease spreading, predicting how epidemics progress through the human networks, and helping health organizations prepare for them [1]. The resilience of networks, such as the power grid is another relevant topic, where network science contributes with models to predict the resistance to failures both random [2] and maliciously targeted [3]. An application closer to day-to-day is the analysis of social networks, both on-line and off-line, which are intertwined with the dynamics of personal opinions, collaboration and habits [4]. For example, the study of networks shows the strength of peer influence. Even if two individuals do not have direct contact, there is an influence regarding behavior such as eating habits or willingness to donate to charity [5]. The analysis of networks can be very complex, and the metrics to analyze them are many. The concepts involved take entire books [6] to describe, but it is worth highlighting the notions of community, the resilience of a network to failures, network growth and its effects on the topology as well as centrality and proximity of nodes in a network.

One fundamental metric in the study of networks is node centrality. Intuitively, the centrality measures how ”important” a node is to a network. This intuition 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, such as Degree Centrality, Closeness Centrality, Katz Centrality or Eigenvector Centrality, which will be further discussed in Section 2.

One important interpretation of centrality is the one related to the influence of the node in the network. The idea of influence takes into account the nodes with which a node is connected, but also the ones it is indirectly connected to, passing through one or more other nodes before reaching them. The intuition is that the centrality of a node depends on the (sum of) the centrality of its neighbors, the nodes connected to it. In other words, a node connected to very influential nodes should be influential as well. Influence is important, for example, in the study of social networks, representing the relative importance of individuals in the network over their peers. It is also very important in the world wide web, and its immense networks of pages and links, where modern web search engines are based on this metric. This notion of centrality is made concrete by the Eigenvector Centrality, Katz centrality and, notably, Google’s PageRank, used in the famed search engine.

Notably, Katz centrality can be expressed as systems of linear equations, which in turn can be expressed in terms of matrix operations, as further discussed in Section 2. This allows the use of the wide range of tools available to solve systems of linear equations to calculate the centrality of a network.

In order to compute the centrality of the nodes of a network, this work will use a Monte Carlo method.

This method computes powers of matrices to solve the systems of linear equations associated with the network metrics, using random walks on matrices. Additionally, this method is very flexible, and can be generalized to compute several other network metrics, also based on random walks or matrix powers. In particular, this work will also focus the trace of the inverse of the matrix representing the network, as an example.

Monte Carlo methods are a vast class of algorithms that use repeated random sampling to obtain results. Their focus on independent random samples makes them, in general, trivially parallelizable by running different random computations in different processing units. Therefore, they are extremely well suited for the current abundance of multi core technology and clusters of machines.

1.1 Objectives

This work will focus on the development of a Monte Carlo method to calculate network metrics, with special focus in the particular case of network centrality. The method should be highly scalable, in order to handle problems of arbitrarily large size, as the current networks demand. It should also explore parallelism techniques to distribute the processing and storage in shared memory and distributed memory settings.

To validate the developed method, its scalability will be tested by increasing problem sizes and number of machines used in parallel. Furthermore, the method will be compared with state-of-the-art alternatives, in terms of performance and accuracy.

1.2 Document structure

In the following sections the world of networks and their metrics will be examined, as well as the artificial models used to emulate them, in Section 2. This is followed in Section 3 by an exploration on the current state of the art algorithms for systems of linear equations, which can be used to calculate network metrics. Relevant parallelism techniques and frameworks will be explored in Section 4, followed by a discussion of the proposed parallel solution in Section 5 and the respective implementation details necessary to guarantee performance in Section 6. The evaluation of the method regarding precision and performance, compared against the state of the art alternatives, is laid out in Section 7, followed by finishing statements in Section 8.

2

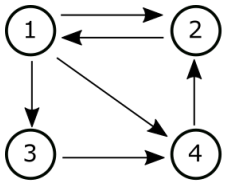
Networks

Contents

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 [6]. Both nodes and edges can be enriched with additional information, such as labels on nodes, or weight and direction on edges.

2.1 Network as a matrix

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 [6]. This convention allows us to represent, if necessary, the direction and weight of edges. The direction is represented by labeling differently the entries with coordinates (i, j) and (j, i), considering i and j as numbers representing distinct nodes, while the weight is represented by placing the weight in the relevant entry of the matrix, instead of a binary indicator of existence of an edge. Figure 2.1 shows the example of a directed network.



0	1	1	1
1	0	0	0
0	0	0	1
0	1	0	0

Figure 2.1: Example of a network and its matrix representation.

2.2 Storing networks in memory

As networks can be represented by their adjacency matrices, the problem of storing a network is the same as the one of storing a matrix in memory.

For a matrix of size N, one simple option is to allocate the $N \times N$ positions in block, and write every value in the matrix, in a dense representation. The use of dense representations, where every entry of the matrix is explicitly stored in memory, is appropriate for storing dense matrices. Sparse matrices, such as the ones representing a common network with few connections per node, have the value 0 on most of the entries. To represent such matrices in a dense fashion is not efficient in terms of used memory. The better option is to store only the values of the non zero entries.

There are several strategies to store sparse matrices in memory. A dictionary mapping $(row, column)$ pairs to values can be used to store the existing elements, while any position missing from the dictionary is considered to have the value 0. Alternatively, the Compressed Sparse Row (CSR) or Yale Format can be used [7]. This format 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).

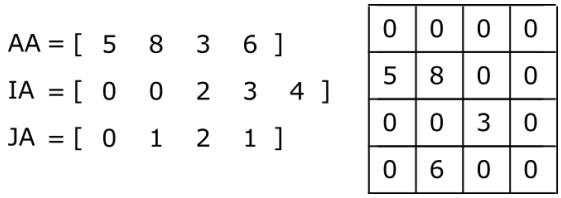


Figure 2.2: Example of a matrix represented in CSR format.

2.3 Network metrics

To characterize a network and its nodes, there is a vast collection of metrics that can be used [6]. The degree of a node is the number of nodes it is connected with by an edge. The degree distribution of a network is the distribution of degrees present in the network. The average path length is the average of all shortest paths between all pairs of nodes. These two metrics, among others, are used to describe the network as a whole.

Other metrics can also be used to study nodes individually, such as the clustering coefficient, which measures, for each node, the fraction of possible connections between its neighbors that exist [6]. The network clustering coefficient is the average of the clustering coefficient of all nodes. Related with the clustering coefficient, the total triangle count in a network is another metric used [8].

Another metric aimed at individual nodes is centrality in its various forms [6]. Degree centrality is the simplest notion, defining centrality as the number of connections of a node in the network. Closeness centrality calculates the centrality of a node considering the average of the minimum distance to all other nodes. Betweeness centrality focuses on how many shortest paths between any two nodes is the node part of.

Following the notion of centrality as the influence of a node on other nodes in the network, there is Katz centrality [9]. This notion of centrality of a node takes into account the number of nodes connected to it, and subsequent nodes connected to these, progressively with less importance. The intuition is that the influence is achieved by reaching nodes, directly or indirectly, and that being connected to influential nodes grants influence to a node. It can be defined, for node i, in terms of the adjacency matrix B as

$$K_i(\alpha) = [(I - \alpha B)^{-1}\mathbf{1}]_i \tag{2.1}$$

with α as an adjustable real parameter [10] and $\mathbf{1}$ as a vector with 1 in all entries. $K(\alpha)$ is a vector with each node's centrality, where the i-th entry is chosen. This parameter α, designated attenuation

factor, should be smaller than $1/\lambda$, λ being the largest eigenvalue of B [9]. It allows this centrality to be tuned to extract slightly different information. Notably, it is shown in [10] 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.

To obtain a vector with the Katz centrality of all nodes in a network, it is therefore necessary to solve

$$K(\alpha) = (I - \alpha B)^{-1}\mathbf{1} \tag{2.2}$$

Alternatively, this equation can be expressed as a system of linear equations, and as such, the tools available to solve that problem can be used to address this one [10]. The networks under study can vary greatly in size, therefore, to analyze them, there is a need for tools able to invert rather large matrices or, equivalently, solve the linear algebraic system.

It should be noted that, while Katz centrality is defined as a matrix multiplied by a vector with 1 in all entries, modifying this vector to have different values corresponds to giving weights to the contribution of each node to the centrality.

Another centrality metric to consider is the resolvent subgraph centrality [10] which, for node i is given by

$$RC_i(\alpha) = [(I - \alpha B)^{-1}]ii \tag{2.3}$$

This expression represents the number of closed walks from node i, weighted according to the length of the walk. Counting closed walks measures the clustering of the network, As such, the sum of all resolvent subgraph centralities in a network can be seen as a global measure of clustering. This is represented by the trace of the inverse of a matrix

$$RC(\alpha) = tr[(I - \alpha B)^{-1}] \tag{2.4}$$

where tr is the trace of a matrix. The trace of a square matrix is defined as the sum of all entries in the main diagonal [8].

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 [11], and the trace of the matrix exponential is known as the Estrada Index [12], and is yet another measure of the centrality of complex networks. Aditionally, 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 [8].

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 relevant metrics available.

PageRank [13] is another very popular centrality metric. It is mainly used in the context of the world

wide web to rank pages in search engines. Intuitively, PageRank considers random walks through the web, where in each page, the user picks any link at random or leaves to a random page. PageRank aims to give the distribution of users over all known pages. It is similar to Katz centrality in that it can also be expressed as an expression with matrices:

$$\frac{(1-c)}{n}\mathbf{1}^T \sum_{k=0}^{\infty} c^k P^k \tag{2.5}$$

where $1-c$ is the probability that a user leaves to a random page, P is a probability matrix representing every link for each page,

This metric is very popular and therefore there is a great amount of research, alternative variations of the idea and implementations. It can be computed using Monte Carlo methods, which can be distributed [14]. These methods can work with evolving networks, improving over previous results as new information arrives. Since it is a well explored topic, it will not be focused in this work.

2.4 Network models

While networks can be found virtually anywhere and in any topic, to capture their information and structure is complex and sometimes unfeasible. It is therefore important to be able to generate artificial networks with desired properties, such as a specific number of nodes, or clustering coefficient.

One of the simplest examples is to obtain a matrix, and therefore a network, from a discrete Poisson equation [15]. This symmetric matrix is composed of a full diagonal and four other diagonals parallel to it. In the network context, this is a very simple network, where every node has a path to every other node. It does not, however, offer much flexibility to obtain desired properties in the generated network.

One simple idea is to obtain a random network, with a desired number of nodes and edges. This can be obtained with the Erdős–Rényi model, which generates a random network from the collection of all possible graphs with the requested number of nodes and edges [16]. This model does not, however, feature some characteristics of social networks and other real networks, such as a high clustering and the small world effect. The intuition of a small world effect is that most nodes of a graph can reach most other nodes in a small number of steps. The Watts Strogatz model was developed, as a simple way to generate networks with these properties [17].

The Watts Strogatz model creates a ring lattice of nodes, each connected to a constant number of nodes, just before and just after the node itself, in the ring. Then, with a given probability, which is a parameter of the model, each edge can be rewired to connect to a different node [17]. This way, the original ring lattice provides a higher clustering to the network than a fully random network would, while still showing a small world effect thanks to the random rewires.

One other property found in some real world networks is that the degree distribution follows a power

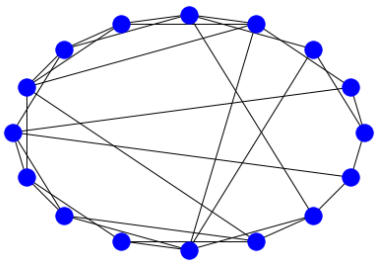


Figure 2.3: Example of a network generated with the Watts Strogatz model.

law. These networks, called scale-free networks, are not generated by the previous models. To generate them, the Barabási–Albert model follows a method based on the growth of a network, with preferential attachment, the idea that a node with more connections is more likely to be connected to [18]. So, the method starts with a number of nodes in a ring, connected to each other, and adds the rest of the desired nodes, one by one. As each node is added, a constant number of edges to it is created. These edges connect to one of the existing nodes chosen at random, with probability proportionally to its degree. Therefore, a node with double the edges of another one, is twice as likely to be connected to the new node. Based in these ideas of growth and preferential attachment, there are a vast number of similar models, appropriate to model different networks [6].

2.4.1 Asymmetric Small World

Matlab was initially used to generate small world matrices, using the Watts–Strogatz method. However, Matlab was unsuitable for the generation of larger matrices, due to the time taken and excessive memory requirements. As such, a custom version of the method was developed, in order to easily generate matrices of virtually any size.

The goal was to keep as many properties from the original model as possible, while making the generation of every row independent from the others, allowing for the parallelization of the generation, and removing the requirement to keep information about other rows in memory.

The method can be extended to any average degree, but it was used to generate networks with an average of 4 connections per node, the same average degree of the Watts–Strogatz networks used. Just like Watts–Strogatz, every node has four connections by default, two to the two nodes preceding it, and the other two to the two nodes after it, in a ring network.

All connections in this network can be asymmetric, changes to one direction of the connection do not affect the other. The two connections to previous nodes each have a probability p to be removed, simulating the behavior in the Watts–Strogatz model, where these connections could be rewired with probability p, and end up connecting to another node. In this adapted model, the connection can disappear or remain independently of the connection departing from the previous node. In the same spirit, the

two connections leading to the following nodes can be rewired to lead to any other node in the network. Each connection forward is rewired with probability p, to any node, with uniform probability.

Additionally, a number of new connections, to any destination, chosen with uniform probability, can be added. The number of new connections is taken from a binomial distribution with probability of success $1/numberOfNodes$ and $numberOfNodes \times 2 \times p$ experiments.

Following this method, the average degree is 4, and the small world effect is provided by the connections with random destinations. The symmetry of the network is lost, but each node or row of matrix can be generated independently, which allows parallelization of the generation, and frees us from maintaining any memory demanding state during the generation.

2.4.2 Kronecker graphs

Kronecker graphs [19] try to emulate real networks, matching their statistical networks. They have been shown to be realistic asymmetric networks. These networks are generated using on a small base matrix, to which the Kronecker product is applied several times. Large instances can be efficiently generated in parallel, for example by the freely available implementation of Graph 500 (Section 4.4).

The available implementation of Graph 500 can be configured to dump the generated network into a chosen file. In the version used, an analysis of the source code shows that the dump corresponds to an unordered sequence of pairs origin-destination, representing the generated connections. These connections can appear repeated several times. Some nodes can also be left isolated, with no inbound or outbound connections.

In order to convert the output of graph 500 into a usable matrix, all isolated nodes were pruned from the matrix. Additionally, the connections were sorted, in order to be useful as input, and repeated connections were removed, leaving only one of each. To sort such a large amount of connections, a bin sorting strategy was used to reduce the amount of elements in each sort operation to a reasonable amount.

The resulting matrices feature hubs, in a scale-free fashion, as well as a large number of poorly connected nodes. These include nodes with only inbound or only outbound, connections.

2.5 Evolving Networks

When we consider networks as entities that change over time rather than static information, we are dealing with evolving networks. Most real networks are evolving networks, such as social networks, as connections are added and destroyed, and nodes are inserted and removed from the network.

If we consider two versions of the same network at different instants of time, in general, we will see some differences, but also great similarity. In order to exploit this similarity, it is helpful to use knowledge

about an earlier version of the network in order to compute metrics about another version.

In order to calculate the inverse of a matrix knowing the inverse of another, the Sherman-Morrison formula [20] can be used. 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 vectors, and the original matrix A is invertible as well as the new matrix $A + uv^t$, the new inverse is given by the following formula:

$$B^{-1} = A^{-1} - A^{-1}u(1 + vA^{-1}u)^{-1}vA^{-1} \tag{2.6}$$

This is generalized by the Woodbury formula [21], which states that:

$$B^{-1} = A^{-1} - A^{-1}U(I_k + VA^{-1}U)^{-1}VA^{-1} \tag{2.7}$$

where U and V are matrices with dimensions n rows by k columns, where k is arbitrary. Using matrices rather than vectors allows us to represent more complex modifications, but implies the inversion of a matrix of size k. As such, k should be as small as possible.

When calculating the product of the inverse of the matrix by a vector of ones, this formula can be written as:

$$B^{-1}\mathbf{1} = A^{-1}\mathbf{1} - (A^{-1}U)(I_k + V(A^{-1}U))^{-1}V(A^{-1}\mathbf{1}) \tag{2.8}$$

As denoted, if we know the value of the original inverse multiplied by the right matrices, we can use this formula to calculate the new inverse, using only operations with matrices of size n by k.

3

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

Contents

Systems of linear algebraic equations are prevalent in an extremely wide area of applications, and their importance could hardly be brought to question. They are used in communications, digital signal processing, as well as physical problems involving partial differential equations [22] among others.

The problem of solving a system of linear equations, with as many variables as 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 [23].

$$Ax = b \Rightarrow x = A^{-1}b \tag{3.1}$$

Such representation allows us to solve the system of linear equations by inverting the matrix and multiplying the result 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. Systems with different characteristics are more efficiently solved by different solvers, there is no single best method for every problem.

3.1 Direct methods

Direct methods aim at finding an exact solution to the SLAE problem. The Gauss-Jordan elimination is a simple example of this class of algorithms [24].

A more practical variant of this method is materialized in the frontal solver, and the more complex parallel implementation, the multifrontal solver. These, applied to sparse linear systems, start with an analysis step on the full matrix, followed by the assembly of dense sub-matrices (called frontal matrices), where additional steps are performed [25]. These methods provide an exact solution, under the limits of the hardware used. However, dense matrix operations are involved, incurring in large storage costs. The software package MUMPS is a free implementation of the multifrontal method, using a parallel approach [26].

3.2 Iterative methods

Iterative methods offer an approximation of the desired result. They are applied iteratively, resulting in incrementally more accurate solutions, based on an initial guess [15]. The initial guess can be generic, such as a vector with 1 in all entries, or one that approximates the desired result. Since it conditions the number of iterations necessary to achieve the desired precision, several methods are used to offer a good initial approximation of the solution. Preconditioners too are used to ensure the convergence of

the result or speed up the process. This allows the iterative method to run faster, as it will need fewer iterations, with good initial conditions.

Some of the simplest iterative methods are Richardson's, Jacobi and Gauss-Seidel methods [23]. 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, several improving on previous work. From these we can highlight the biconjugate gradient method [23], GMRES [27] and BICGstab [28]. In some of the methods, such as the Jacobi and Gauss-Seidel methods, convergence to a solution is guaranteed only in problems that respect some restrictions, depending on the method, such as the matrix being diagonally dominant. To circumvent this problem, preconditioning techniques may be used, to transform the problem into the space of solvable problems. The PETSc suite implements several of these methods, along with direct solvers and a number of preconditioners [29].

While precision is important in the iterative methods, the purpose of the initial guess is merely of approximating the solution to speed up the method, and should be calculated as fast as possible. The preconditioner too, should be very efficient to compute, more so than the "main" method itself.

3.3 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 [30]. They are commonly applied for optimization purposes or numerical simulation, and are most effective when the complexity of calculating an exact solution is overwhelming. 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 [30].

The Central Limit Theorem establishes that the sum of independent random variables tends towards a normal distribution. Additionally, it is shown [31] that the error of the estimate is of the order of $1/\sqrt{N}$, where N is the sample size. This gives us a concrete bound with which to tune Monte Carlo algorithms, with respect to the number of samples needed to reach a desired level of accuracy.

One great advantage of Monte Carlo algorithms is that, being based on the repetition of random sampling, they can be trivially parallelizable, taking full advantage of several processing units or machines to calculate different samples.

3.3.1 Markov Chain Monte Carlo methods

Markov Chain Monte Carlo (MCMC) methods are a subclass of Monte Carlo methods, with the purpose to sample from and approximate a probability distribution. These are based on a Markov Chain with the desired distribution as a stationary distribution [32]. A large portion of these algorithms rely on random walks through the Markov chain, consisting of a series of steps through the entries of the

matrix. The length of these walks (number of steps) is another variable that changes the accuracy of the approximation. Using more steps will improve the approximation. This must be balanced with the sample size, to obtain the desired accuracy in the shortest amount of time.

3.3.2 Monte Carlo methods for Systems of Linear Algebraic Equations

Monte Carlo methods, given their tempting characteristics, are not new to this problem. These can be used as preconditioners for iterative methods, in order to speed them up [22]. Monte Carlo methods fit this task well, as they are efficient, but have an error associated with their result, and the preconditioner should be a fast way to calculate an approximation of the solution. Monte Carlo methods are also used to calculate the final solution [33].

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. As such, they also tend to have a slow convergence rate, since the methods follow the behavior dictated by the central theorem $(1/\sqrt{N})$. Examples of these are described in [34], in the form of the Forward Method and the Adjoint Method and in [35].

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 [34].

In [34] some examples are presented, such as Sequential Monte Carlo, where each iteration of Richardson's method (iterative method) is followed by a Monte Carlo iteration to correct the result, and as such, speedup the convergence. In a similar vein, [34] presents the Monte Carlo Synthetic Acceleration, which adds to the previous an iterative step, in order to filter out some error generated by the Monte Carlo method.

Monte Carlo methods in general depend on performing the average of a large number of samples, and the error reduces as we increase the number of samples, according to the Central Limit Theorem. There are, however, limits to this reduction of error, as the sum of so many numbers will incur in large rounding errors. There are methods for reducing the variance, as described in [22], that allow us to reduce the amount of samples needed, avoiding this problem and speeding up the computation.

4

Parallelism

Contents

With the rise of multiprocessors, computer clusters and high performance supercomputers, efficient computation has steered away from sequential programming. To achieve high performance it is fundamental to fully use several processors working together in the same machine, and several machines in collaboration, enjoying the increased computational power and memory space provided by joining efforts [30].

The high performance machines now in the service of academia and industry, the supercomputers, are essentially multiprocessor machines (nodes) interconnected by high speed networks. Nodes can be organized in islands, with higher interconnection, with links between islands suffering from slower connections. These machines can also be equipped with graphic processors, providing massively parallel processing under some restrictions.

In order to explore these parallel oriented machines, there are several frameworks supporting thread management and inter process communication. This work will explore the academia standards, OpenMP (Open Multi Processing) [36] for multithreading and MPI (Message Passing Interface) for interprocess communication [37].

4.1 OpenMP

This framework allows for easy parallelization of serial code, providing an easy to use interface for thread parallelism, based on annotations [30]. The ease of use is implemented with no loss of performance or functionality, as OpenMP provides the usual tools for parallelization, such as critical zones, and atomic operations. For example, it is simple to launch several threads running different iterations of a for loop simply by annotating the for instruction. This annotation is also configurable, in order to define how the iterations are divided among threads, giving the programmer simple tools to adapt thread behavior to the needs of the project.

While thread management is largely simplified by the available commands, the task of making sure there is no unsafe concurrent access to variables, nor a deadlock or other parallel programming pitfall, is left to the programmer. As with all parallel computation frameworks, it is in the interest of performance to avoid synchronization between threads as much as possible, in order to prevent idle time for any of the threads and fully use the processing resources available.

4.2 MPI

MPI is the *de facto* standard for high performance multi processing, abstracting the details of communication to a high level interface [30]. As such, the creation of communication channels and management of its details such as IP addresses, ports and communication protocol are handled by the framework,

providing to the user simple numeric identifiers for processes, abstracting completely the machine where they are running. Aside from end to end communication, this specification offers built-in routines for distributing, reducing and aggregating data over multiple processes. These routines are implemented for an efficient execution, overlapping communication where possible, providing the user with out of the box optimized methods which would be tedious and error prone to reimplement otherwise.

This is the standard in the research community due to ease of use and high performance. This specification has several implementations, namely the Open MPI, MPICH and Intel MPI, which causes some variance in behavior between different installations, as not all functionalities are totally defined by the standard [30].

4.3 Graphical Processing Units

While not in the focus of this work, Graphical Processing Units (GPU) and their uses cannot be overlooked. These have expanded from their original graphic purpose into general use, thanks to their massively parallel architecture. This comes at the cost of some restrictions, since efficient execution in GPUs is limited in ways that CPU threads are not, making their programming less intuitive and more complex. Nonetheless, these restrictions have not been a barrier to the use of GPUs in a wide range of applications in the scientific field, such as the training of neural networks [38] or alignment of DNA sequences [39].

GPUs have some variation in design, but in general focus on vector operations, with a large number of threads performing the same operations over different data, in a Single Instruction Multiple Data (SIMD) paradigm [40]. This is specially useful in vector or matrix operations, in the scientific field. It is also convenient for the implementation of Monte Carlo algorithms, since the computation of different samples follows the pattern of applying the same operations to different data.

Nowadays, GPUs have thousands of concurrent threads, performing some tasks much faster than if using a traditional CPU [38]. Their use in academia has been supported by the emergence of abstractions over the complex GPU architecture and graphic oriented notions, such as CUDA [40] by NVIDIA or OpenCl [41]. CUDA works with C, C++ and Fortran, as well as other languages, geared towards parallel computing notions rather than graphics programming notions, as the prior APIs such as Direct3D and OpenGL, which required more specialized knowledge.

4.4 Supercomputers

Supercomputers, which provide high performance computing, are essentially large clusters of machines, each possessing multi core CPUs and sometimes GPUs. These are interconnected by high

speed networks, specially designed for the purpose. The Top500 project [42] lists and details the most powerful supercomputers in the world, including information about the processing speed in PetaFLOPS (10^{15} Floating point operations per second).

As the processing power escalated, power consumption and heat production rose as well. These motivated the construction of large cooling facilities to ensure proper performance of the machines, exaggerating even more the energy consumption. This escalation led to an alternative focus on power efficiency, supported by the Green500 [43], which lists the top supercomputing facilities in relation to energy-efficient performance, with the goal to promote sustainable supercomputing.

As an example, Marconi is the Italian leading high performance computing infrastructure, divided in three sections [44]. The architecture used is Intel OmniPath Cluster, a high speed network designed for supercomputers. The second section is placed in 14th place in the top500 list, as of November 2017, composed by 3600 nodes, each with 68 cores of Intel Xeon Phi CPUs, listed with a peak performance of 11 PetaFLOPS. The current work has been granted access to run on the first section of Marconi, which features 1512 nodes with 36 cores of Intel Xeon E5-2697 v4 (Broadwell) CPUs, listed 72th in the top500 list. The third section is still in a preliminary phase. Development and benchmark for the current project were also executed in Mare Nostrum, at the Barcelona Supercomputing Centre, currently sitting in 16th place in the top500 list.

Graph 500 [45] is another benchmark ranking super computers based on their performance. It was developed as an alternative to Top500, recognizing that the current benchmarks are not good indicators of the performance of supercomputers in data intensive applications, and aiming to develop a set of alternative large-scale benchmarks. It executes operations over networks represented by Kronecker graphs.

4.5 Tools

Extrae and Paraver are tools developed at the Barcelona Supercomputing Centre, in order to analyze parallel applications. Extrae measures and registers events during the execution of the program, through sampling and instrumentation of the code. Among the events logged, we can count MPI calls and messages, OpenMP actions, and hardware counter values. Paraver allows for a visual representation of the metrics and activities logged. There are several possibilities of graphics to show, including, for example, a timeline of the program activity in each process and thread, including the communications, with lines representing every message between two processes. It is also possible to summarize the data in useful statistics, for example on how much time is spent inside each MPI Call, or the amount of cache misses inside and outside MPI calls. The volume of data stored by Extrae only allows for short executions with a limited number of processes and threads, which can be a problem when attempting

to diagnose behaviors observed in large scale executions. Nonetheless, the ability to visualize the execution flow and message pattern is invaluable to properly understand the behavior and efficiency of the program.

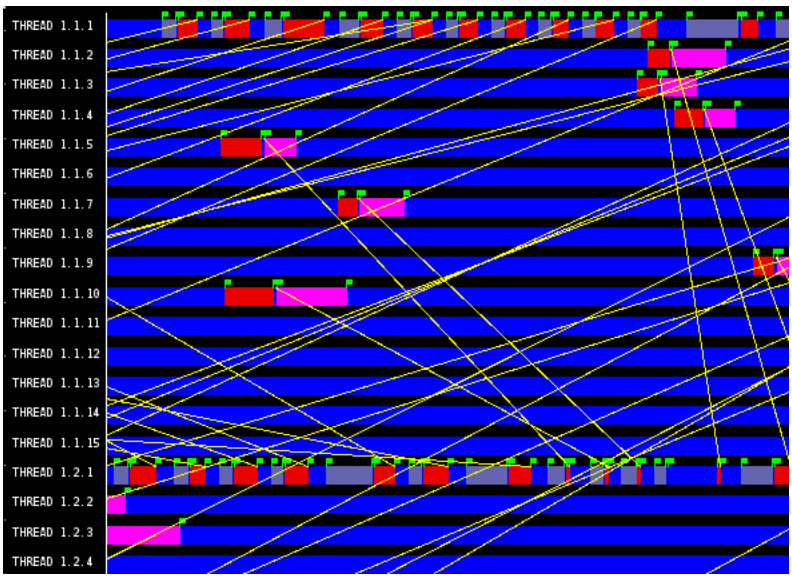


Figure 4.1: Example of the representation of multithreaded processes in Paraver. Each row represents one thread and its actions.

5

Proposed Solution

Contents

In this section, the proposed method to invert a matrix will be laid out, as well as the modifications necessary to compute the centrality of a network, followed by the discussion of the challenges of parallelizing the method. Additionally, details of implementation regarding optimization of the shared memory are discussed and finally, how the method can be adapted to meet different challenges, such as evolving networks or computing the trace of a matrix.

5.1 The method

The proposed solution is based on the method described in [46]. Considering a matrix B, let $A = I - B$, where I is the identity matrix. This method is based in the fact that

$$B^{-1} = (I - A)^{-1} = \sum_{k=0}^{\infty} A^k \tag{5.1}$$

when the following condition is held:

$$\max_r |\lambda_r(A)| < 1 \tag{5.2}$$

considering $\lambda_r(A)$ as the r-th eigenvalue of A.

In order to approximate B^{-1}, the method under study approximates the $\sum_{k=0}^{\infty} A^k$ calculation. In turn, to approximate the sum to infinity, we can only sum the first n powers. The method calculates an approximation R to

$$R = \sum_{k=0}^{n} A^k \tag{5.3}$$

This is a good approximation for a large enough n, since this expression converges as n tends to infinity. To calculate Equation 5.3, the method calculates independently each row of each power of the matrix A, using a Monte Carlo strategy analogous to the Markov Chain Monte Carlo notion of chains (described in Algorithm 1). 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.

Algorithm 1 Monte Carlo Matrix Inversion

```
1: function InvertMatrix(B, n, p)
2:     I ← IdentityMatrix(length of B)
3:     A ← I − B
4:     for i ← 0 to length of A − 1 do
5:         R[i] ← [0, ..., 0]
6:         for k ← 0 to n do
7:             R[i] ← 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, the one with the index of the column of the chosen entry, described in Algorithm 2 and ilustrated in Figure 5.1. The choice of entry is made at random, with probabilities for each entry of the row weighted proportionally to their absolute value.

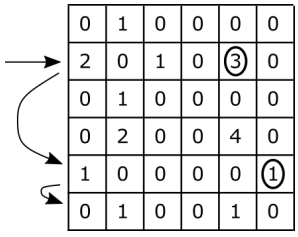


Figure 5.1: Example of a possible play in a matrix, with the selected entries highlighted

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 ← 1 to p do                      ▷ Use p plays in the approximation
4:          value ← 1
5:          currentRow ← i
6:          for y ← 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 ← 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 its value by the sum of the values of the entries in the current row ($\sum_j A_{i,j}$, described in Algorithm 3). When a play reaches the end of its k steps, its value is added to the vector r, in the position with the index of the row it is currently at (r_i). After running all plays, the values in r must be divided by the number of plays made, 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.

The value of n and the number of plays p used to approximate each power determine both the accuracy of the result and the computational effort required to calculate it.

Algorithm 3 Row Weight Computation

```
1: function GetRowWeight(A, i)                    ▷ Calculate the weight of row i of A
2:     rowWeight ← 0
3:     for j ← 0 to length of A[i] − 1 do
4:         rowWeight ← A[i][j] + rowWeight          ▷ Constant values
       return rowWeight                            ▷ This operation can be done once and cached
```

5.2 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 without sacrificing precision.

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 strategy allows the method to use a significantly smaller number of steps, Cn steps instead of $C(n^2+n)/2$. 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 initial experiments showed only a small increase in the error, and a very beneficial reduction in execution time. Algorithm 4 describes the algorithm in practice.

Algorithm 4 Optimized Monte Carlo Centrality Calculator

```
1: function ComputeCentrality(B, n, p)
2:     I ← IdentityMatrix(length of B)
3:     A ← I − B
4:     c[0, length of A − 1] ← [0, . . . , 0]          ▷ Array of numbers
5:     for i ← 0 to length of A − 1 do
6:         r[i] ← ComputeRow(A, n, i, p)
       return r
7: function ComputeRow(A, n, i, p)                ▷ Compute row i of the sum of n potencies
8:     rowCentrality ← p
9:     for x ← 0 to p do                           ▷ Use p plays in the approximation
10:        value ← 1
11:        currentRow ← i
12:        for k ← 0 to n − 1 do
13:            value ← value × GetRowWeight(A, currentRow)
14:            currentRow ← RandomWeightedChoice(A, currentRow)
15:            rowCentrality ← rowCentrality + value
       return rowCentrality / p
```

5.3 Parallel implementation

The goal of this work lies in developing a scalable parallel method to calculate the centrality. In order to parallelize the described method (Algorithm 4), there are two main strategies.

The calculation of each row is totally independent from the others, and therefore trivially parallelizable by running them in different processing units. This can be done in a shared or distributed memory setting, but the calculation of one play assumes access to the entire matrix. This strategy involves simply parallelizing the iterations of the cycles in Algorithm 4 line 5 or line 9.

Should a matrix be too large for the memory of a single machine, we need to be able to distribute this computation to several machines without a shared memory, where each machine keeps a part of the matrix in its memory. This form of parallelization is more complex, as it 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 it reaches 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 5 and illustrated in Figure 5.2.

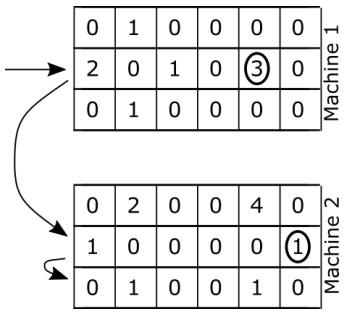


Figure 5.2: Example of a possible path in a matrix distributed over two machines. The highlighted positions are the selections the algorithm performs in each row.

This communication adds a layer of complexity and overhead to the initial method. Communication should be performed, as much as possible, in parallel with computation, in order to fully use the available resources. Messages between machines may have a number of plays aggregated in them, and the receiving end must have a buffer to collect plays to process. These buffering techniques help improve performance, but must be tuned to the execution in machines with different characteristics.

A parallel implementation exploring the use of several machines does not invalidate the use of shared memory parallelism, in the form of thread parallelism in each machine. This can be implemented using uniform threads, all receiving, generating and computing plays in parallel, over the same part of the matrix stored in memory. It is also possible to implement heterogeneous threads, where some should handle communication tasks such as receiving or sending messages and managing associated buffers, while others exclusively process plays. There is no clearly optimal option, practical implementations and

Algorithm 5 Parallel Monte Carlo Centrality Simulator

```
1: function WORKER(A)
2:     r[0, length of A] ← [0, ..., 0]                          ▷ Array of numbers
3:     while working do
4:         if INCOMINGMESSAGE( ) then
5:             play ← RECEIVEPLAY()
6:         else if UNGENERATEDPLAYS( ) then
7:             play ← GENERATEPLAY()
8:         if play is defined then
9:             PROCESSPLAY(A, play, r)
10:    for i ← 0 to length of r − 1 do
11:        r[i] ← 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 ← pl.stepsLeft − 1
16:        pl.value ← pl.value × GETROWWEIGHT(A, pl.currentRow)
17:        pl.currentRow ← RANDOMWEIGHTEDCHOICE(A, pl.currentRow)
18:        r[pl.startRow] ← r[pl.startRow] + pl.value
19:    if pl.stepsLeft > 0 then
20:        SEND(pl)
```

tests will be needed to choose.

As parallel programming is used in Monte Carlo methods, it is important to have uncorrelated pseudo-random number sequences in each execution thread, in order to preserve the accuracy of the method. To this end, several techniques can be used, such as centralizing the generation of random numbers. Centralizing is, however, a poor choice regarding performance, therefore it is preferable to use decentralized methods, such as independent seeds for each pseudo-random sequence [30].

5.4 Storing the matrix in memory

In the proposed method, it is necessary to perform two operations over rows of the matrix, choose an entry randomly taking in account the values, and getting the sum of all entries. To speed up computation, the sum of all entries in each row can be initially calculated and stored in a vector. The matrix representation in memory will be similar to CSR, with this added information. To choose an entry randomly, with probabilities proportional to the absolute value of the entry, the interval between 0 and 1 is proportionally divided between all entries. This normalization attributes to each entry an interval, with a size proportional to the original value. The union of all intervals has a size of 1. Then, a random number between 0 and 1 is generated, and the entry whose interval the number belongs to is selected. This behavior is illustrated in Figure 5.3

In practice, the limits of each interval are calculated at the start of the computation and used every

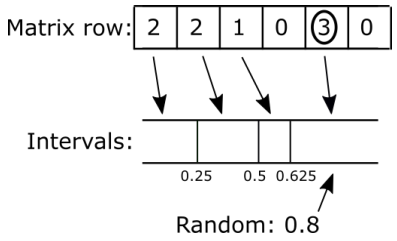


Figure 5.3: Example of the entry choice in a row and the respective intervals used.

time it is necessary to randomly select an entry. In order to select the interval where the random number falls into, it is necessary to perform a search in this vector. To perform this search, there are a few options from which to choose from. The simple linear search, going through each element of the vector in order, is of the simplest implementation. A binary search, starting at the middle of the vector, and recursively searching one half or the other based on the observed value, also works over a simple vector, but, in average, requires a smaller number of steps to find the right entry.

The binary search just described can be performed in a binary search tree, where in each node, starting at the root, the left or right child is explored depending on the value found, until a leaf (entry) is reached [47]. In memory, the tree is a set of nodes with pointers to the value and left and right child nodes. Since the probabilities of choosing each interval are known and fixed, an optimal binary search tree can be used, with its branching organized in order to reduce the average number of steps to reach the desired entry to the minimum. This strategy requires extra computation at the start, but reduces the effort of choice to the minimum.

A practical comparison of these different strategies has been made. It is in the interest of this project to focus on sparse matrices, with few entries per row, therefore the vectors used while testing contained a few dozens of entries at maximum. For vectors with this small number of entries, the trees presented a disadvantage in reading time. Although the steps to perform a binary search over a vector and over a binary search tree are exactly the same, the difference in speed can be attributed to the extra space taken up by pointers in the tree data structure. These can be harmful to the cache locality principle and slow down the access. The binary search outperformed slightly the simple linear search, and is, therefore, the chosen option in the implementations.

5.5 Alternative applications

The method description so far focuses on the calculation of the Katz centrality. However, as described, the proposed method bases itself on the calculation of individual rows of powers of a matrix. This provides flexibility to the base method, which can be used 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 [10].

5.5.1 Trace of a matrix

The process to calculate the inverse matrix and the product of the inverse matrix by a vector have been described in previous sections. In the proposed method, the trace of a resulting matrix can be easily computed, by accounting and summing only the entries in the diagonal. Since this is a Monte Carlo method, which performs approximations for every entry, the sum of elements should help in achieving a greater precision in comparison with the calculation of individual elements, as errors will cancel out.

5.5.2 Evolving networks

Regarding the calculation of the inverse of a matrix, this method can also be used to approach the problem of recalculating the inverse after small changes to the original matrix or network, which is specially important for evolving networks. The intuition here is that, should we remove or add an entry to a matrix, we are effectively adding or removing paths followed by the plays used.

In each row i, the value of each entry represents the impact of the row with column's index, j, in the values of row i. Therefore, should we remove an entry from the original matrix, the new inverse can be calculated by removing from the original inverse the effect of that entry. This can be achieved by running the plays starting at that entry, to calculate its results, and then subtracting the obtained result at every row of the inverse, weighted according to the value in the column j. Likewise, to add an entry, it is necessary to calculate the effect of plays starting at said entry, and add it to all rows of the inverse, weighted accordingly to the value of the plays reaching the row of the entry (the value on the j-th column of each row).

Considering that any change to a matrix can be decomposed in terms of removing and adding individual entries, this method should allow for the calculation of a new inverse matrix after arbitrary changes to the original matrix, taking advantage of the notion of plays and the ability of the Monte Carlo method to calculate individual rows.

This intuition is equivalently expressed by the Sherman-Morrison formula, as described in Section 2.5. In order to extended the idea to arbitrary modifications, we can use instead the Woodbury formula, which depends on knowing the result of the inverse multiplied by a matrix. However, this equivalent to knowing the result of the inverse multiplied by several vectors, each of the columns of the matrix.

When computing Katz centrality, we compute the inverse of a matrix multiplied by a vector (v) of ones, as shown in Section 2.3.

$$K(\alpha) = (I - \alpha B)^{-1} v \tag{5.4}$$

However, with the Monte Carlo method, we can use any other vector easily, and we can compute several vectors at the same time. In order to use an arbitrary vector, we just need to multiply the result stored at each step by the vector entry corresponding to the current row, as illustrated in Algorithm 6.

Algorithm 6 Parallel Monte Carlo Simulator with custom vector

```
1: function PROCESSPLAY(A, pl, r, vector)
2:     while pl.stepsLeft > 0 and ISINMEMORY(pl.currentRow) do
3:         pl.stepsLeft ← pl.stepsLeft − 1
4:         pl.value ← pl.value × GETROWWEIGHT(A, pl.currentRow)
5:         pl.currentRow ← RANDOMWEIGHTEDCHOICE(A, pl.currentRow)
6:         r[pl.startRow] ← r[pl.startRow] + pl.value × vector[pl.currentRow]
7:     if pl.stepsLeft > 0 then
8:         SEND(pl)
```

In order to compute several vectors at the same time, we need to, in each step, store results for each of the vectors, multiplying each by the appropriate vector's entry. In Algorithm 6 this corresponds to executing line 6 for each of the vectors, storing the result in separate output vectors. We can compute an arbitrary number of products of the inverse by vectors.

6

Implementation details

Contents

As a general guideline, the implementation should avoid or minimize high overhead tasks. Reducing the amount of calls to MPI as much as possible is one such case, due to the efficiency cost of messages, as well as the overhead of interacting with the library.

Memory management is, as well, of paramount importance. In particular, memory allocation is a source of overhead, and, as such, is totally avoided in the final implementation, aside from the initial allocations.

The efficient use of the cache is important to maintain performance. The nature of the random walks, leading to random accesses on the matrix and a lack of temporal locality in accesses, makes it difficult to take advantage of the cache. Nonetheless, spacial locality can be leveraged by the cache, by keeping all the relevant information about each row of the matrix adjacent in memory.

6.1 Distributing rows over processes

By default, the rows of the input matrix are equally distributed among all processes. However, the network structure might lead to heavier computation cost in some nodes, such as hubs, where many random walks will go. In order to allow a more balanced distribution of nodes over processes, a file defining a custom distribution is accepted by the program.

This custom distribution was used in the experiments over matrices with hubs, such as networks generated by Graph 500. The distribution split the amount of connections equally over all processes, rather than the amount of nodes. This is a simplistic approach to the expected workload of each process, but proved successful in keeping a good workload balancing.

6.2 Communication

In order to use the available resources as efficiently as possible, the asynchronous versions of MPI calls were used both in sending and receiving messages, so that computation and communication could be overlapped. The performance advantage of asynchronous functions was verified experimentaly. Both sending and receiving messages asynchronously proved advantageous in the total execution time, and analysis with Paraver.

To reduce the amount of messages sent, which would be overwhelming otherwise, each message contains a number of plays with the same destination process. Outgoing plays are aggregated in a buffer according to their destination, and sent in a 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. Messages with hundreds of plays give the best performance, but the optimal size of the buffer must be experimentaly

determined for the machine and network where the method is executed.

On the receiving end, MPI has the option to internally manage a buffer to receive messages, using the ”Bsend” call. However, there is some lack of clarity on the behavior when the buffer fills and reduced control over the execution, compared to a programmer controlled buffer. As such, the final implementation features manually managed buffers for incoming messages, filled using asynchronous receive calls. The size of this buffer is less impactful, given that it is big enough, in order not to fill and delay the computation.

6.3 Terminating the computation

In order to determine when the computation is finished, each process counts the finished plays it registers. A token with value 0 is given to the first process, at the start of the execution. When a process becomes idle, if it has the token, it will add its number of finished plays to the value of the token, and send it over to the next process. When the token reaches the first process again, it will check if the token's value is equal to the total amount of plays. If it is so, the computation is finished, and all processes get a termination message. Otherwise, the value of the token is reset to 0, and the behavior is repeated.

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

6.4 Concurrency

Due to the use of threads in the implementation, concurrency over memory positions is another problem to consider. All accesses to the matrix stored in memory are ”reads”, it is convenient that all threads read from the same matrix stored in memory, to better use the available RAM. However, when writing in memory, it is very important to avoid writes in shared memory positions, that need to be synchronized. It is also fundamental to avoid ”false sharing”, where two threads write in the same cache block, ”stealing” the block from each other. These considerations are important when dealing with writing the final results of random walks, but also in any other memory structure shared between threads.

6.4.1 Thread configuration

It is desirable to use several threads, in order to fully utilize multi-core machines. Using threads instead of several processes in the same machine allows us to store a bigger chunk of the matrix in the same process, and have several processors sharing it, reducing the amount of communication needed.

When using several threads in the same process, we have the the choice to make them homogeneous, 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 Paraver, it was observed that the cache misses caused by accessing MPI functions were concentrated in the communication thread, reducing them in the others.

The use of more communication threads, each serving a subset of the working threads was experimented with, however, it didn't show any advantage. The analysis using Paraver showed that the calls to MPI where taking longer in each thread, making the performance similar to the version with only one communication thread, hinting at synchronization happening inside the MPI library.

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 its own buffer. This requires some communication between the communication thread and each of the worker threads, namely, two index variables, representing the portion of the input buffer that is filled with messages to process.

6.4.2 Workload distribution

The input buffer of each thread is treated as a circular buffer. Each worker thread acts as as a consumer, advancing one index through the existing messages, while the communication thread acts as a producer, advancing the other index while it fills the buffer. Thanks to this configuration, where each thread writes in one index and the other reads, it is not necessary to synchronize the access to these shared variables. The decision to keep separate message input buffers for each thread allows us to maintain this lack of synchronization, as, for each buffer, there is only one consumer. This reduces the performance hurdles to a minimum, limiting them to the cache misses that must occur when one thread modifies the value of the index the other reads.

Each of these structures with shared variables is separately allocated in the heap, since, if they were in the stack of one of the threads, it could lead to cache invalidation of other variables in the stack. It is also important to keep the variables of each thread separated so that they do not cause cache invalidations to each other.

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.

6.4.3 Sending messages

In order to keep threads as independent as possible, each has its own output buffers, where plays going to other processes are aggregated into messages sent by each thread in separate. Synchronization outside the MPI calls is completely avoided here, however, as 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 messages are sent in a small number and with as many plays as possible.

6.4.4 Storing the results

As each play finishes, its value is added to a vactor 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 (Figure 6.1). additionally 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.

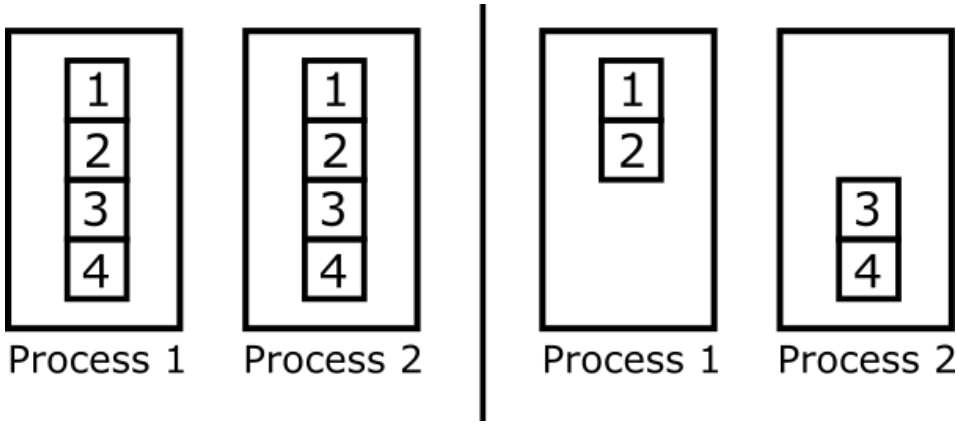


Figure 6.1: Result vector replicated in every process (left), or distributed among processes (right).

With only the ”owned” rows being 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 step and communication in most plays. It is desirable to avoid this extra step by keeping a full sized vector of results in every process. However, as matrix size increases, keeping the full vector in memory in every process is an unacceptable strain on the available memory. As such, the final implementation sees each process storing only the results for its owned rows.

With one vector where all threads write, every write must be synchronized. Since these are, each, updates over a single memory position, the ”omp atomic” instruction 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 a greater performance, and is the choice taken in the final implementation.

In order to further reduce the access to the result vector, the data structure of the play has a field where the partial result it will contribute is accumulated. The partial result is only added to the result vector at the end of the play.

7

Evaluation

Contents

The resulting implementations must be evaluated under two main characteristics, precision and performance. The relevant computation steps, in the Monte Carlo method, to calculate both the trace and the centrality vector are the same, therefore they are not analyzed in separate regarding performance, only regarding precision.

The main topologies used in this analysis will be custom small world matrices (Section 2.4.1) and Graph 500 matrices (Section 2.4.2). Additionally, there are two real world networks, two Wikipedia cross-reference networks, at two points in time, available to us. As alternatives to compare against, a direct method and an iterative method will be considered. MUMPS provides an implementation of the multifrontal method (direct), while PETSc implements, among others, the iterative GMRES method. These will be used as the representatives of the state of the art alternatives, due to their use and existence of maintained academic implementations.

Both measures will need to be tested against increasing problem sizes, matrices of increasing dimensions and different matrix topologies. It is one of the objectives of this work to ensure the scalability of the algorithm, maintaining a good performance as the problem size and the number of processes used increases. As such, the scalability when tackling larger problem sizes will also be measured, and compared with alternatives. Ideally, the method will retain its efficiency, and spend resources proportionally to the problem size, with reduced overhead.

7.1 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 5, and evaluate the behavior of the method, as these parameters are adjusted.

7.1.1 Sources of error

The implementation has two sources of error, the number of powers of the matrix calculated, controlled by the parameter 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 obeys the bounds described in section 3.3. Monte Carlo error can be controlled predictably, knowing that the error of the estimate is in the order of $1/\sqrt{p}$.

However, the number of powers computed, which is related to the length of the random walk, is a source of error as well. The desired result is the sum of all powers, up to infinity, therefore, the error grows when the number of powers computed is smaller. Since the power series converges (per the requirements laid out in Section 5), powers of higher order should have a small impact on the error. The precise impact, however, depends on the convergence of the particular matrix chosen, leaving the significance of the value of n problem dependent.

7.1.2 Error analysis

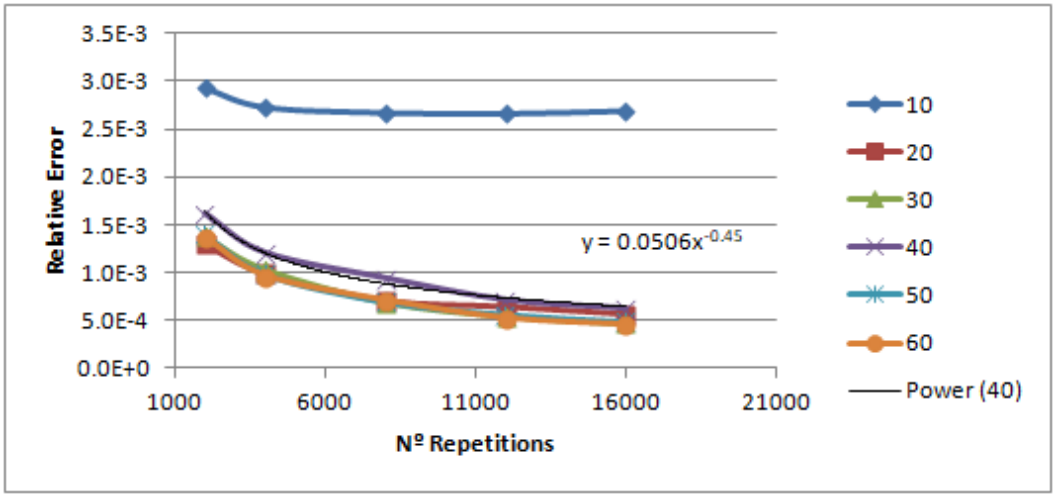


Figure 7.1: Relative error of the centrality of the small world matrix with 4096 nodes.

We can observe a similar downwards pattern for all play lengths (Figure 7.1). The results with 10 steps are less precise while all others fall in the same values. This suggests that, for this matrix, the higher powers, beyond 20, have a small impact, while powers up to 20 have more impact in the result.

Using a logarithmic scale on both axis (Figure 7.2), we can observe the downwards slope in error as it would be expected of the Monte Carlo approximation. It is not perfect, presenting nonetheless a good approximation to the expected slope of 0.5. That 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 5.2.

When the length of the plays is increased, the error reduces (Figure 7.3), 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 observed in Figure 7.4. As indicated by the expression of the trend-line, the error decreases with the exponential of the length of plays.

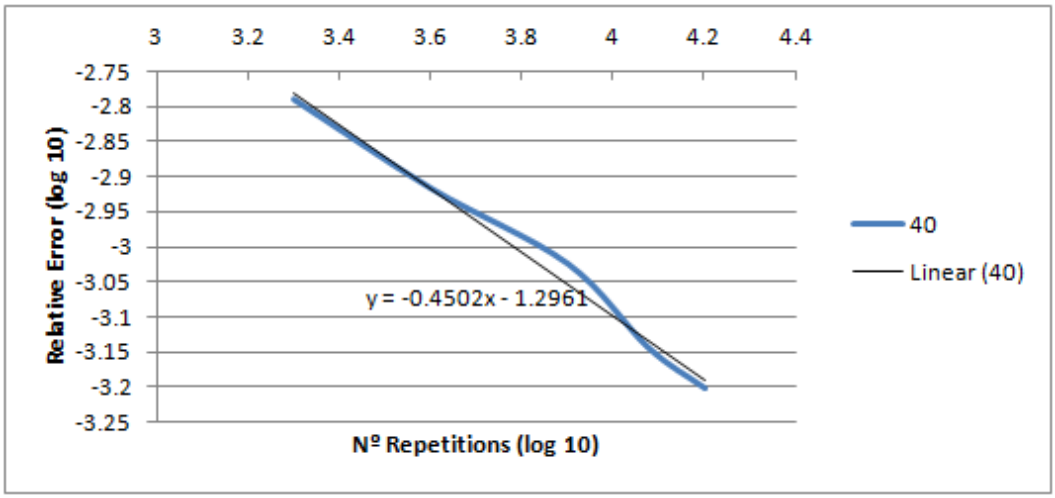


Figure 7.2: Relative error of the centrality of the small world matrix with 4096 nodes, using a path with 40 steps.

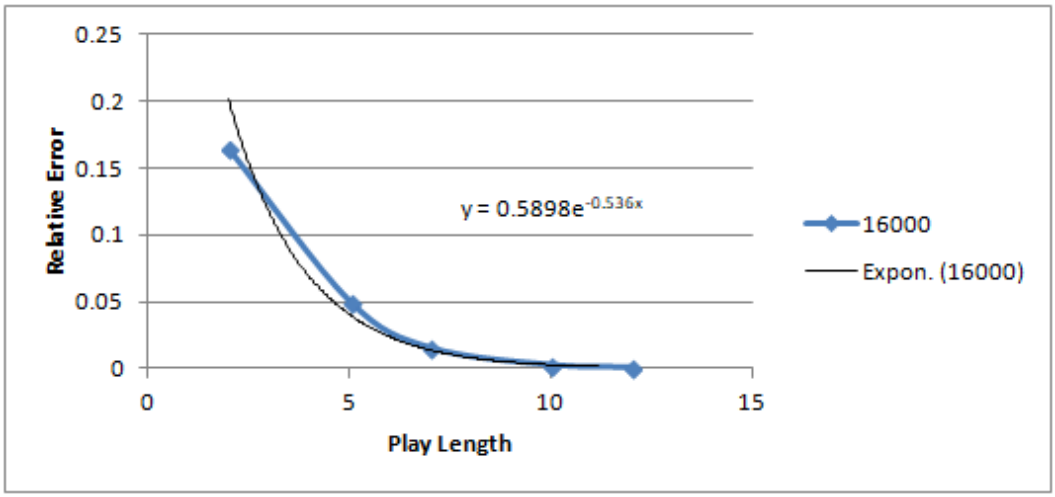


Figure 7.3: Relative error of the centrality of the small world matrix with 4096 nodes, using 16000 repetitions.

Results in the Graph 500 matrix of size 4819 also present a downwards pattern, although less pronounced (Figure 7.5). The relative error is also much smaller. Since these matrix 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 (Figure 7.6) 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.

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 7.7). 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

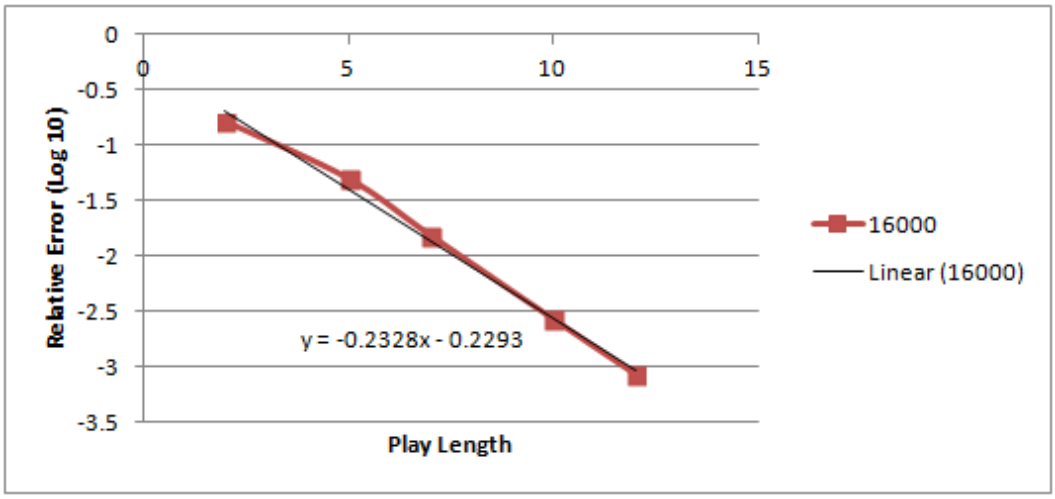


Figure 7.4: Logarithm base 10 of the relative error of the centrality of the small world matrix with 4096 nodes, using 16000 repetitions.

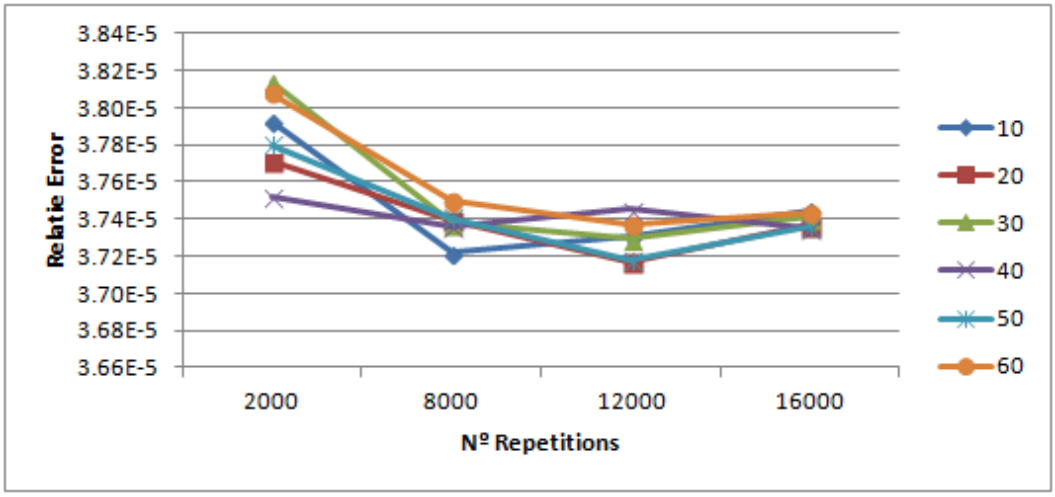


Figure 7.5: Relative error of the centrality vector of the graph 500 matrix with 4819 nodes.

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 (Tables 7.1 and 7.2).

7.2 Performance

The method will be analyzed in comparison with state of the art alternatives, regarding execution time, while using several different number of processors. Furthermore, the algorithm will be analyzed by itself, with respect to scalability, testing an increasing number of processing units.

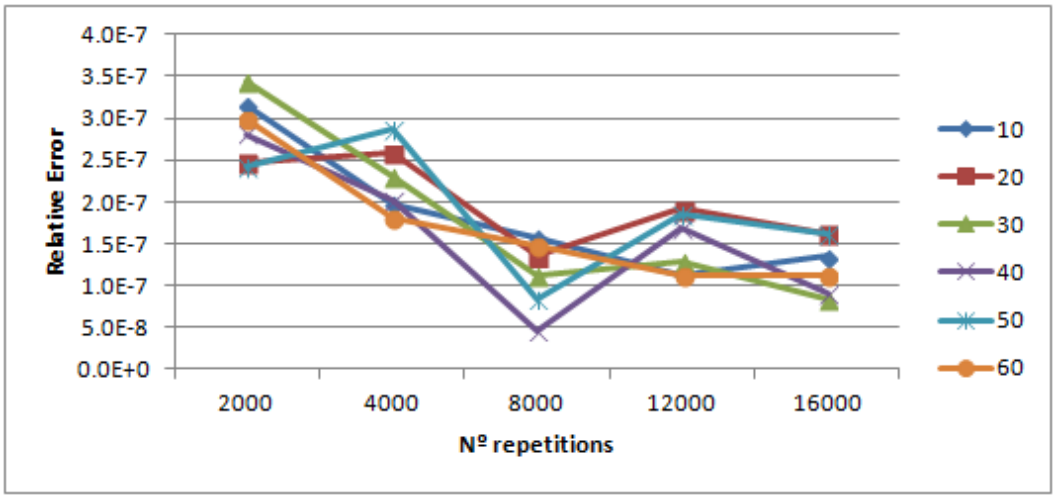


Figure 7.6: Relative error of the trace of the inverse of the graph 500 matrix with 4819 nodes.

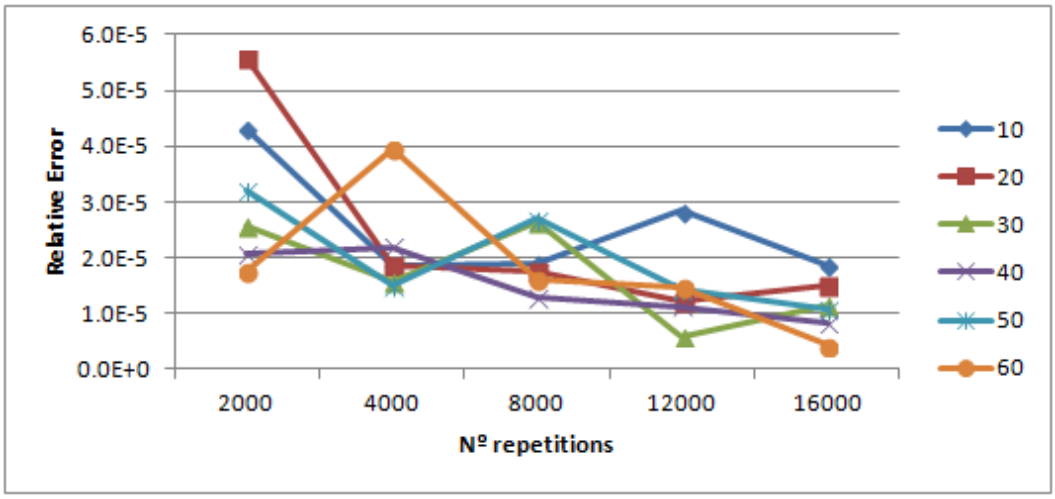


Figure 7.7: Relative error of the trace of the inverse of the small world matrix with 4096 nodes.

7.2.1 Performance results

When scaling the problem size over the same number of processes (Figure 7.8), 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 but the number of processes increases, the execution time drops as expected (Figure 7.9). The efficiency is not perfect here, as it can be observed by the trendline, expressed as a power function, where the exponent doesn't reach 0.5, which would be ideal. This is also shown in Figure 7.10, where the ideal speedup is represented by a straight line, but we can observe that the growth of the speedup does not hold for larger number of processes. This is to be expected as adding more processes increases the communication, which slows down the computation.

Table 7.1: PETSc trace relative errors for small world matrices

size	error
1024	0.0032%
2048	0.0026%
4096	0.0028%
8192	0.0029%
16384	0.0019%

Table 7.2: PETSc trace relative errors for graph 500 matrices

size	error
1341	0.0022%
2553	0.0016%
4819	0.0008%
9183	0.0004%
17490	0.0002%

A very large number of processes is not a good way to speedup the computation. However, it should not be necessary, 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. Splitting the repetitions between several independent groups of processes and aggregating the results in the end allows for a perfect scalability, for large enough problems.

A weak scaling analysis of the algorithm (Figure 7.11) 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. The Wikipedia cross-reference networks presented similar execution times and patterns to the model networks.

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

Table 7.3: MUMPS execution time (s) over several small world matrices (Mare Nostrum)

Nº processes	64k	128k	256k
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

MUMPS encounters problems with scaling using several processes, showing barely any scaling (Table 7.3). Scaling by increasing the problem size is equally problematic, as the scaling is far from

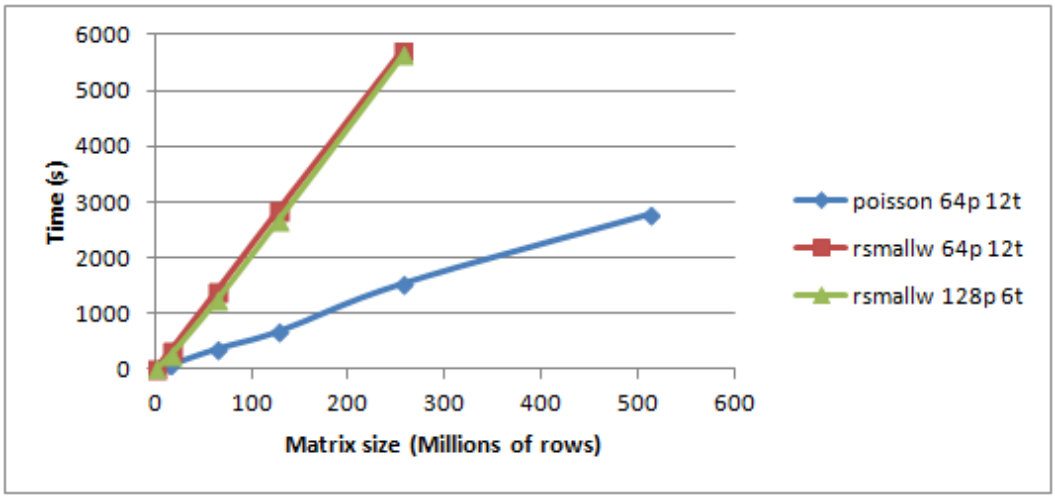


Figure 7.8: Execution time of the implementation while increasing problem size, with different number of processes and threads over small world and Poisson matrices (Mare Nostrum). The type of matrix is stated first, followed by the number of processes used and then the number of threads per process.

linear. Furthermore, storing the matrix in memory quickly becomes a problem for MUMPS, making it impossible to explore large matrices.

Table 7.4: 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 in 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 7.4) were taken from a single process running the implementation, for 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, and divide to get the parallelized 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 takes 244.26 seconds when executing with 128 processes, and therefore 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 advantage, as it explores the fact that it calculate all entries in the result matrix in its execution, unlike PETSc. Furthermore,

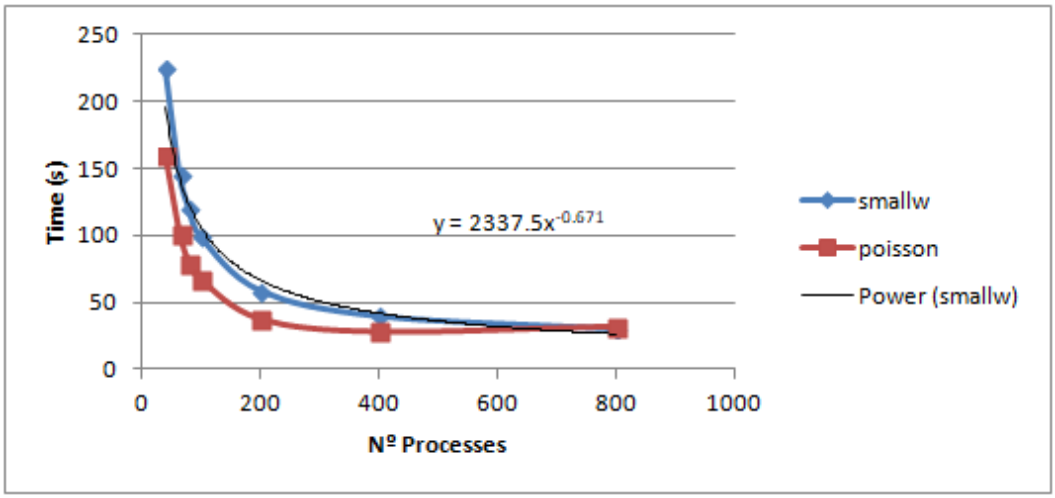


Figure 7.9: Execution time of the implementation over matrices of size 16M rows and 12 threads, while increasing the number of processes over small world and Poisson matrices (Mare Nostrum).

the PETSc implementation used encountered some problems with very large input matrices, while the Monte Carlo method distributes large matrices with ease.

Table 7.5: 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 5.5.2, 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 7.5. 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 2.5), 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.

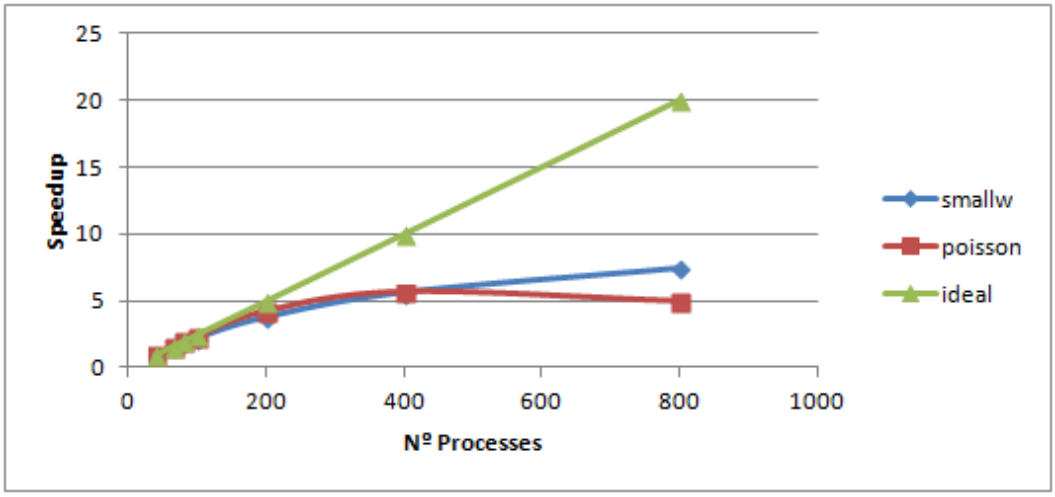


Figure 7.10: Speedup of the implementation, compared to the execution with 40 processes, over matrices of size 16M and 12 threads, while increasing the number of processes over small world and Poisson matrices (Mare Nostrum).

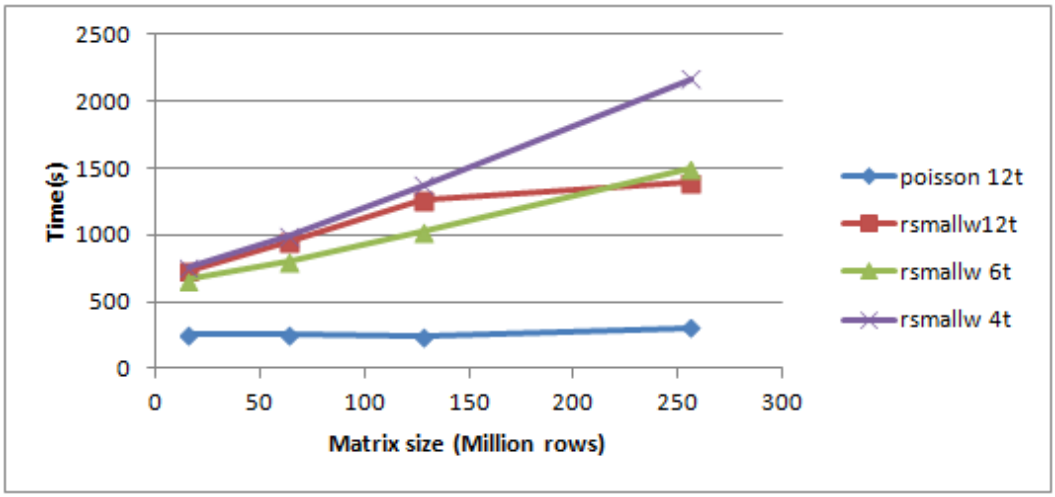


Figure 7.11: Execution time of the implementation with different number of processes and matrix size scaling at the same rate (Mare Nostrum).

8

Conclusion

This work proposed a Monte Carlo method to compute metrics of complex networks. This method uses random walks over a matrix representing the network to obtain an approximation of the powers of the matrix. From this information, approximations to 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.

This Monte Carlo method and its current implementation can also deal with very large matrices, and surpass individual machine's memory limitations thanks to an efficient distributed computation, avoiding the hurdles found in the alternative methods. To evaluate both performance and precision, the method was tested over different types of networks, with small world properties, simulating real complex networks. The implementation was executed in the supercomputers Mare Nostrum and Marconi. Both feature high speed networks interconnecting nodes with two NUMA cores, with several processors each, making it interesting to explore process parallelism and thread parallelism.

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. When computing the product of the inverse by a vector, the Monte Carlo implementation is very slow compared to PETSc, when considering a similar error.

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.

Bibliography

[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, no. 1, p. 45, Sep 2009. [Online]. Available: https://doi.org/10.1186/1741-7015-7-45

[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, no. 5, 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. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevLett.95.098104

[5] F. L. Pinheiro, M. D. Santos, F. C. Santos, and J. M. Pacheco, "Origin of peer influence in social networks," *Phys. Rev. Lett.*, vol. 112, p. 098702, Mar 2014. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevLett.112.098702

[6] M. E. J. Newman, *Networks : an introduction*. Oxford New York: Oxford University Press, 2010.

[7] 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*, ser. SPAA '09. New York, NY, USA: ACM, 2009, pp. 233–244. [Online]. Available: http://doi.acm.org/10.1145/1583991.1584053

[8] 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.

[9] L. Katz, “A new status index derived from sociometric analysis,” *Psychometrika*, vol. 18, no. 1, pp. 39–43, 1953.

[10] C. Klymko, “Centrality and Communicability Measures in Complex Networks : Analysis and Algorithms,” Ph.D. dissertation, Emory University, 2013.

[11] E. Estrada and J. A. Rodríguez-Velázquez, “Subgraph centrality in complex networks,” *Physical Review E*, vol. 71, no. 5, May 2005. [Online]. Available: http://dx.doi.org/10.1103/PhysRevE.71.056103

[12] 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. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0024379507002844

[13] L. Page, S. Brin, R. Motwani, and T. Winograd, “The pagerank citation ranking: Bringing order to the web,” 11 1998.

[14] K. Avrachenkov, N. Litvak, D. Nemirovsky, and N. Osipova, “Monte carlo methods in pagerank computation: When one iteration is sufficient,” *SIAM Journal on Numerical Analysis*, vol. 45, no. 2, p. 890–904, Jan 2007. [Online]. Available: http://dx.doi.org/10.1137/050643799

[15] G. H. Golub and C. F. Van Loan, *Matrix computations*. Johns Hopkins Univ. Press, 2007.

[16] P. Erdős and A. Rényi, “On Random Graphs I,” *Publicationes Mathematicae (Debrecen)*, vol. 6, pp. 290–297, 1959 1959.

[17] D. J. Watts and S. H. Strogatz, “Collective dynamics of “small-world” networks,” *Nature*, vol. 393, no. 6684, p. 440–442, Jun 1998. [Online]. Available: http://dx.doi.org/10.1038/30918

[18] R. Albert and A.-L. Barabási, “Statistical mechanics of complex networks,” *Reviews of Modern Physics*, vol. 74, no. 1, p. 47–97, Jan 2002. [Online]. Available: http://dx.doi.org/10.1103/RevModPhys.74.47

[19] 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. [Online]. Available: http://dl.acm.org/citation.cfm?id=1756006.1756039

[20] 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, no. 1, p. 124–127, Mar 1950. [Online]. Available: http://dx.doi.org/10.1214/aoms/1177729893

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

[22] J. Straßburg and V. Alexandrov, "A Monte Carlo Approach to Sparse Approximate Inverse Matrix Computations," *Procedia Computer Science*, vol. 18, pp. 2307–2316, jan 2013. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S1877050913005450

[23] Y. Saad, *Iterative methods for sparse linear systems*. Philadelphia: SIAM, 2003.

[24] K. Atkinson, *An introduction to numerical analysis*. New York: Wiley, 1989.

[25] 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. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S004578259900242X

[26] 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. [Online]. Available: https://doi.org/10.1137/S0895479899358194

[27] 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. [Online]. Available: https://doi.org/10.1137/0907058

[28] 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. [Online]. Available: https://doi.org/10.1137/0913035

[29] 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. [Online]. Available: http://www.mcs.anl.gov/petsc

[30] M. Quinn, *Parallel programming in C with MPI and openMP*. Dubuque, Iowa: McGraw-Hill, 2004.

[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. [Online]. Available: http://www.jstor.org/stable/1990053?origin=crossref

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

[33] I. T. Dimov and V. N. Alexandrov, "A new highly convergent Monte Carlo method," *Mathematics and Computers in Simulation*, vol. 47, no. I, pp. 165–181, 1998.

[34] 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.

[35] T. Wu and D. F. Gleich, “Multi-way Monte Carlo Method for Linear Systems,” 2016. [Online]. Available: http://arxiv.org/abs/1608.04361

[36] 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.

[37] L. Clarke, I. Glendinning, and R. Hempel, *The MPI Message Passing Interface Standard*. Birkhäuser Basel, 1994, p. 213–218. [Online]. Available: http://dx.doi.org/10.1007/978-3-0348-8534-8_21

[38] R. Raina, A. Madhavan, and A. Y. Ng, “Large-scale deep unsupervised learning using graphics processors,” in *Proceedings of the 26th Annual International Conference on Machine Learning*, ser. ICML ’09. New York, NY, USA: ACM, 2009, pp. 873–880. [Online]. Available: http://doi.acm.org/10.1145/1553374.1553486

[39] S. A. Manavski and G. Valle, “CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment,” *BMC Bioinformatics*, vol. 9, no. 2, p. S10, Mar 2008. [Online]. Available: https://doi.org/10.1186/1471-2105-9-S2-S10

[40] J. Nickolls, I. Buck, M. Garland, and K. Skadron, “Scalable parallel programming with cuda,” *Queue*, vol. 6, no. 2, pp. 40–53, Mar. 2008. [Online]. Available: http://doi.acm.org/10.1145/1365490.1365500

[41] “Opencl,” https://www.khronos.org/opencl/, accessed: 2018-06-19.

[42] “Top500,” https://www.top500.org, accessed: 2017-12-20.

[43] “Green500,” http://www.top500.org/green500/, accessed: 2017-12-20.

[44] “Marconi User Guide,” http://wiki.u-gov.it/confluence/display/SCAIUS/UG3.1%3A+MARCONI+UserGuide/, accessed: 2017-12-20.

[45] “Top500,” https://graph500.org, accessed: 2018-06-09.

[46] 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. [Online]. Available: http://www.cs.fsu.edu/{~}mascagni/Forsythe{_}Leibler{_}MTAC{_}1950.pdf

[47] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, *Introduction to algorithms*. Cambridge, Mass: MIT Press, 2009.