HIGH-PERFORMANCE SIMULATION OF INTERACTING MULTIPARTICLE QUANTUM WALKS WITH APACHE SPARK

André Luiz Figueiredo de Albuquerque

Dissertação de Mestrado apresentada ao Programa de Pós-graduação em Engenharia de Sistemas e Computação, COPPE, da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Mestre em Engenharia de Sistemas e Computação.

Orientadores: Alexandre de Assis Bento Lima
Franklin de Lima Marquezino

Rio de Janeiro
Março de 2018

HIGH-PERFORMANCE SIMULATION OF INTERACTING MULTIPARTICLE QUANTUM WALKS WITH APACHE SPARK

André Luiz Figueiredo de Albuquerque

DISSERTAÇÃO SUBMETIDA AO CORPO DOCENTE DO INSTITUTO ALBERTO LUIZ COIMBRA DE PÓS-GRADUAÇÃO E PESQUISA DE ENGENHARIA (COPPE) DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

RIO DE JANEIRO, RJ – BRASIL

MARÇO DE 2018

Albuquerque, André Luiz Figueiredo de

High-Performance Simulation of Interacting Multiparticle Quantum Walks with Apache Spark/André Luiz Figueiredo de Albuquerque. – Rio de Janeiro: UFRJ/COPPE, 2018.

X, 67 p.: il.; 29, 7cm.

Orientadores: Alexandre de Assis Bento Lima
Franklin de Lima Marquezino

Dissertação (mestrado) – UFRJ/COPPE/Programa de Engenharia de Sistemas e Computação, 2018.

Referências Bibliográficas: p. 65 – 67.

1. quantum walk. 2. simulation. 3. high performance computing. 4. map-reduce. 5. apache spark. I. Lima, Alexandre de Assis Bento *et al.* II. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia de Sistemas e Computação. III. Título.

Acknowledgments

I would like to thank my advisors Alexandre de Assis Bento Lima and Franklin de Lima Marquezino for their presence and patience when helping me to finish this work.

I thank my mother for all the support she gave me during this process.

I also thank the High Performance Computer Center (NACAD) - COPPE/UFRJ, which has supported this research.

Thanks must be given to CAPES for the financial support too.

Resumo da Dissertação apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

SIMULAÇÃO EM ALTA PERFORMANCE DE CAMINHADAS QUÂNTICAS DE MULTIPARTÍCULAS INTERATIVAS COM APACHE SPARK

André Luiz Figueiredo de Albuquerque

Março/2018

Orientadores: Alexandre de Assis Bento Lima
Franklin de Lima Marquezino

Programa: Engenharia de Sistemas e Computação

Embora muitos algoritmos quânticos têm sido desenvolvidos nas últimas décadas com consideráveis ganhos em complexidade quando comparados a seus equivalentes clássicos, a construção de um computador quântico de propósito geral ainda é um desafio tecnológico. Enquanto o *hardware* necessário para rodar algoritmos quânticos ainda não está disponível, pesquisadores dependem de simulações clássicas. Entretanto, as simulações mais interessantes demandam grandes quantidades de recurso computacional devido à quantidade de dados crescer exponencialmente em relação ao tamanho das instâncias e, assim, técnicas de computação de alto desempenho são necessárias. Caminhadas quânticas de multipartículas têm recebido grande atenção recentemente como uma ferramenta para desenvolvimento de algoritmos quânticos e para modelagem de fenômenos físicos. No presente trabalho, nós mostramos que o Apache Spark, um arcabouço para processamento de dados em larga escala, pode ser usado para simular caminhadas quânticas de multipartículas interativas, com tamanhos que são impraticáveis em computadores de propósito geral e com apenas um processador. Nós também disponibilizamos um protótipo para simulações de caminhadas quânticas, adequado para *clusters* de computadores, desenvolvido utilizando o Spark.

Abstract of Dissertation presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

HIGH-PERFORMANCE SIMULATION OF INTERACTING MULTIPARTICLE QUANTUM WALKS WITH APACHE SPARK

André Luiz Figueiredo de Albuquerque

March/2018

Advisors: Alexandre de Assis Bento Lima
Franklin de Lima Marquezino

Department: Systems Engineering and Computer Science

Although many quantum algorithms have been developed in the last few decades with considerable speedup when compared to their best classical counterparts, the task of building a general purpose quantum computer is still a technological challenge. While the hardware necessary to run quantum algorithms is not available, researchers rely on classical simulations. However, the most interesting simulations are very demanding of computational resources due to the amount of data growing exponentially with the instance sizes and, thus, high performance computing techniques are necessary. Multiparticle quantum walks have been receiving a great deal of attention recently as a tool for designing quantum algorithms and for modeling physical phenomena. In the present work, we show that Apache Spark, a framework for large-scale data processing, can be used to simulate quantum walks with multiple interacting particles, with instance sizes that are impractical on single-processor, general-purpose computers. We also provide a prototype for quantum walks simulations, suitable to computer clusters, being developed atop of Spark.

Contents

List of Figures **ix**

1 Introduction **1**

1.1 Motivation 1

1.2 Organization 3

2 Fundamental Concepts **4**

2.1 A Short Introduction to Quantum Mechanics 4

2.2 Quantum Walks 6

2.2.1 Single-Particle Quantum Walks 6

2.2.2 Mesh Percolation 11

2.2.3 Multiparticle Quantum Walk 13

2.3 High-Performance Computing with Apache Spark 14

2.3.1 The Apache Spark Framework 15

2.3.2 Considerations when Using Spark 18

2.4 Related Work 20

3 Simulating Quantum Walks **23**

3.1 Our Quantum Walk Simulator 23

3.1.1 Operators and States 25

3.1.2 Coins 27

3.1.3 Meshes 29

3.1.4 Discrete Time Quantum Walk 35

3.1.5 Probability Distribution Functions 43

3.2 How to Use 47

4 Experiments **51**

4.1 Methodology and Execution Environment 51

4.2 Results 53

5 Final Remarks **62**

5.1 Conclusions 62

5.2 Future Work 63

Bibliography **65**

List of Figures

2.1 Probability distribution of a one-dimensional quantum walk with Hadamard coin after 100 steps. 8
2.2 Example of diagonal and natural meshes. 10
2.3 Probability distribution of a two-dimensional quantum walk with Hadamard coin after 100 steps. 10
2.4 Possible cases of percolations in one-dimensional mesh. Adapted from [1]. 11
2.5 Example of quantum walks on a line with $p = 0.01$ (left) and $p = 0.1$ (right). 10 simulations were run to get their mean probability distribution functions. 12
2.6 Spark's driver-workers architecture. Adapted from [2]. 16
2.7 Examples of narrow (top diagrams) and wide (bottom diagrams) dependencies. Adapted from [3]. 17
2.8 Example of execution of stages. Adapted from [3]. 18
2.9 Example of possible RDD dependencies in a join transformation. Adapted from [4]. 19

3.1 Example of edges numbering with an one-dimensional mesh of size 5. 33
3.2 Example of edges numbering with two-dimensional meshes (diagonal and natural) of size 5x5. 35
3.3 Comparison of the number of nonzero elements of evolution operators when simulating quantum walks with two particles over a lattice. . . 38

4.1 Execution times for 10 steps quantum walks on one worker node with different number of partitions. 54
4.2 Execution times for 15 steps quantum walks on one worker node with different number of partitions. 54
4.3 Execution times for 20 steps quantum walks on one worker node with different number of partitions. 54
4.4 Execution times for 10 steps quantum walks on two worker nodes with different number of partitions. 55

4.5 Execution times for 15 steps quantum walks on two worker nodes with different number of partitions. 55
4.6 Execution times for 20 steps quantum walks on two worker nodes with different number of partitions. 56
4.7 Execution times for 10 steps quantum walks on three worker nodes with different number of partitions. 56
4.8 Execution times for 15 steps quantum walks on three worker nodes with different number of partitions. 57
4.9 Execution times for 20 steps quantum walks on three worker nodes with different number of partitions. 57
4.10 Memory footprint for the scalability experiments. 58
4.11 Scalability of Spark when doubling the number of cores and the problem size. 59
4.12 Speedup produced by Spark for a 30 steps quantum walk, while doubling the number of cores. 59
4.13 Efficiency of Spark for a 30 steps quantum walk, while doubling the number of cores. 60
4.14 Memory footprint of large quantum walks processed by Spark when possessing few computational resources. 60
4.15 Execution times of large quantum walks processed by Spark when possessing few computational resources. 61

Chapter 1

Introduction

Quantum computing is a computational model based on the properties and principles of quantum mechanics, which in turn is a physical theory that describes systems at the atomic and subatomic scales. Richard Feynman is considered as the pioneer of quantum computing, since in 1982 he first observed that simulating quantum systems with classical computers would be highly inefficient, and suggested that a quantum computer would be able to perform such simulations efficiently [5]. Since then, many quantum algorithms have been invented, offering time complexity gain in relation to their classical counterparts — for example, Shor's algorithm for integer factorization and discrete logarithm performing exponentially faster, Grover's algorithm for unstructured search with its quadratic gain, and Harrow, Hassidim and Lloyd's algorithm for solving linear systems of equations which, under certain conditions, can also perform exponentially faster, all of them in [6]. One of the most successful methods for designing quantum algorithms is the quantum walk [7], which is the quantum counterpart of the random walk, and is currently a topic of interest and research in the quantum computing field. Quantum walks can be studied on different models, and the present work focus on the coined discrete-time model (DTQW).

1.1 Motivation

Although several quantum algorithms have already been developed (and keep emerging), the hardware for a universal quantum computer is still a great technological challenge. This is due to the fact that the inner components of quantum computers consist of subatomic particles which must be controlled with great precision while still being kept well isolated from the environment. If the particles interact with the environment, a phenomenon known as quantum decoherence occurs and they lose the quantum properties. Some companies have already successfully built prototypes of quantum computers. The most famous of those prototypes is, perhaps, the one from canadian company D-Wave Systems, which created a lineage of quantum computers — although not universal — starting with the 128-qubit[1] D-Wave One in 2011[2]. Currently, Google and IBM are battling for the quantum supremacy, with the former planning to bring a 49-qubit general-purpose quantum computer this year, and the latter planning a 50-qubit computer to the next few years[3].

While a reasonably-sized universal quantum computer is not available, researchers run numerical simulations of quantum algorithms using classical computers which are, in turn, quite inefficient for these tasks. Therefore, the idea is to run those simulations using high-performance computing (HPC) environments, like computer clusters, allowing such data to be generated and processed at a reduced time in a parallel/distributed way.

The focus of the present work is the simulation of interacting multiparticle DTQW with HPC using Apache Spark[4]. Spark is a parallel, cluster-oriented framework for data processing, being mainly used for Big Data applications. As they evolve, some quantum walk simulations acquire characteristics of a Big Data application due to the exponential growth suffered by their data structures. Thus, employing a framework like Apache Spark seems to be a good approach, making it possible to execute larger simulations than when using single-processor, general-purpose computers. On a previous work, Apache Hadoop[5] has been used to simulate DTQW on a low-cost computer cluster [8]. However, Spark delivers better performance than Hadoop in several cases due to its in-memory characteristic avoiding the persistence to disk of every step of MapReduce and providing methods that better exploits the use of the main memory.

In order to evaluate our approach, we performed experiments on a HPC cluster using a prototype of a DTQW simulator built atop of Spark that we developed for this work, which aimed to offer an extensive set of features, be extensible and easy to use. Results show that the prototype delivers good performance for simulating DTQW, scaling-up with no issues and provides a speedup reasonably close to the linear for a majority of the selected nodes.

[1]A qubit, or quantum bit, is the minimum unity of information in quantum computing; analogous to concept of bit from classical computing.

[2]R. C. Johnson, "Is D-Wave a Quantum Computer?", EETimes, available at `http://ubm.io/2wYNHKD`, last access in Sep. 12, 2017.

[3]M. Feldman, "Google and IBM Battle for Quantum Supremacy", Top500.org, available at `https://bitly.com/2f9YUA8`, last access in Sep. 12, 2017.

[4]`http://spark.apache.org/`

[5]`http://hadoop.apache.org/`

1.2 Organization

This text is organized as follows. In Chap. 2, we provide an introductory review and a formal definition of DTQW, along with the basic concepts of quantum mechanics. Also, we discuss the use of Apache Spark as an HPC framework, reviewing its architecture and some components. In the end, we provide some informations about other quantum walk simulators. In Chap. 3, we present the application developed atop of Spark for this work, giving in-depth details of its implementation and providing some usage instructions. The set of experiments to evaluate our approach is described in Chap. 4, which also contains its corresponding results. Finally, in Chap. 5, we present our conclusion about the obtained results and the usage of Spark to simulate DTQW and discuss future improvements to the ou developed software.

Chapter 2

Fundamental Concepts

This chapter starts with a brief review of the fundamental ideas from quantum mechanics, which are necessary to understand quantum computing and quantum walks. The following section contains a formal description of one-dimensional and two-dimensional quantum walks, detailing the construction of their components and presenting a previously-developed technique to simulate interactions between particles. Next, we present the Apache Spark, a large-scale data processing framework, providing some details of its core components and some important considerations when using it. The last section contains some related works, where we provide a summary of some already developed quantum walks simulators.

2.1 A Short Introduction to Quantum Mechanics

The mathematical framework of quantum mechanics can be summarized in only four postulates: the first describes the state of the system; the second describes the dynamics of the process; the third describes the composition of many systems; and, finally, the fourth describes the measurements of the system. The mathematical formalism basically requires linear algebra, and is usually written in *Dirac notation*, where vectors, known as *kets*, are denoted by:

$$|\Psi\rangle = \begin{pmatrix} \psi_i \\ \vdots \\ \psi_n \end{pmatrix}, \tag{2.1}$$

and their duals, known as *bras*, are denoted by

$$\langle\Psi| = |\Psi\rangle^\dagger = (\psi_i^* \cdots \psi_n^*), \tag{2.2}$$

with $\psi_i \in \mathbb{C}, \forall i$. The postulates of quantum mechanics can be stated as follows.

State-space postulate The state of a quantum physical system isolated from the environment is described by an unitary vector in a Hilbert space $\mathcal{H}$. A qubit is described as a unit vector in a two-dimensional Hilbert space,

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle, \tag{2.3}$$

where $\alpha, \beta \in \mathbb{C}$ are called *amplitudes*, and $\{|0\rangle, |1\rangle\}$ is known as the computational basis. Notice that the amplitudes must satisfy $|\alpha|^2 + |\beta|^2 = 1$. The matrix representation of the computational basis states are usually given by column matrices

$$|0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \text{ and } |1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix}. \tag{2.4}$$

Evolution postulate The evolution of a closed quantum system is described by an unitary operator. If a quantum system is at a state $|\psi_1\rangle$ at time t_1, the state of the system at time t_2 will be $|\psi_2\rangle$ due to a transformation applied by an unitary operator U,

$$|\psi_2\rangle = U|\psi_1\rangle \tag{2.5}$$

Since U is an unitary operator, the norm of the state is preserved — that is, if state $|\psi_1\rangle$ is a unit vector, the result $|\psi_2\rangle$ of the transformation will also be an unit vector. According to the postulates of quantum mechanics, we expect quantum algorithms to be described as chains of unitary transformations applied in a given state vector, as we have in

$$|\psi_n\rangle = U_n \dots U_1|\psi_1\rangle \tag{2.6}$$

Composition postulate The Hilbert space of a composite system is the tensor product of the associated state spaces. Therefore, if the state of subsystem A is associated to Hilbert space $\mathcal{H}_A$, of dimension $\dim(\mathcal{H}_A) = d_A$, and the state of subsystem B is associated to Hilbert space $\mathcal{H}_B$, $\dim(\mathcal{H}_B) = d_B$, then a quantum state in the composed system will be represented by a unit vector in the Hilbert space $\mathcal{H}_A \otimes \mathcal{H}_B$, $\dim(\mathcal{H}_A \otimes \mathcal{H}_A) = d_A d_B$. When we are restricted to the matrix representations of state vectors and linear transformations, then we refer to the tensor product as the Kronecker product. If A is an $m \times n$ matrix, and B is a $p \times q$ matrix, then the Kronecker product $A \otimes B$ is the $mp \times nq$ block matrix obtained by multiplying each entry of A by the entire B matrix, as in

$$A \otimes B = \begin{pmatrix} a_{00}B & a_{01}B & \cdots & a_{0,n-1}B \\ a_{10}B & a_{11}B & \cdots & a_{1,n-1}B \\ \vdots & \vdots & \ddots & \vdots \\ a_{m-1,0}B & a_{n-1,1}B & \cdots & a_{m-1,n-1}B \end{pmatrix}. \tag{2.7}$$

The tensor product of two vectors can be denoted by $|\psi\rangle \otimes |\varphi\rangle$, or $|\psi\rangle|\varphi\rangle$, or even $|\psi, \varphi\rangle$ for short. When the state of a composed system cannot be factorized into the tensor product of the states of its constituent subsystems, we say it is *entangled*. An example of one such state is $\frac{1}{\sqrt{2}}(|0,0\rangle + |1,1\rangle)$, which cannot be factorized into the tensor product of two qubits.

Measurements A projective measurement is described by a Hermitian operator $\mathcal{O}$ in the state space of the measured system, also known as *observable*. The possible outcomes of the measurement are the eigenvalues of $\mathcal{O}$. The observable has diagonal representation given by

$$\mathcal{O} = \sum_{\lambda} \lambda P_{\lambda}, \tag{2.8}$$

where P_{λ} is the projector onto the eigenspace associated with the eigenvalue λ. If the state before the measurement is $|\psi\rangle$, then the probability of obtaining outcome λ is given by

$$p_{\lambda} = \langle\psi|P_{\lambda}|\psi\rangle, \tag{2.9}$$

and in that case the state of the system after the measurement is irreversibly collapsed to

$$|\psi'\rangle = \frac{1}{\sqrt{p_{\lambda}}} P_{\lambda}|\psi\rangle. \tag{2.10}$$

When clear from context, we may refer to the outcome of the measurement as the collapsed state $|\psi'\rangle$ instead of the eigenvalue λ.

Although the above exposition is sufficient for the scope of this text, the reader interested in a more detailed explanation of quantum mechanics may refer to Venegas-Andraca [7] or Portugal [9], which are textbooks focused on quantum computing and quantum walks.

2.2 Quantum Walks

2.2.1 Single-Particle Quantum Walks

Quantum walk is a generalization of the random walk by replacing the classical walker by a quantum particle. The simplest discrete time random walk example is over a line, where the walker moves at each step according to the result of a (possibly biased) coin flip: if the outcome is heads, with probability p, the particle moves to the right; if the outcome is tails, with probability $1 - p$, the particle moves to the left. Due to the random nature of this process, one cannot know the previous position of the walker given the current state. However, the model raises several questions, such as calculating the probability of finding the particle at position n after t steps, or the spread of the particle over time (described by the variance of the position). The dynamics of the quantum walk is similar to the classical setup.

For a quantum walk on a line, the position of the particle is represented by a unit vector in Hilbert space $\mathcal{H}_p$, spanned by the basis $\mathcal{B}_p = \{|x\rangle : x \in \mathbb{Z}\}$. The discrete time model requires an extra vector space $\mathcal{H}_c$, spanned by $\mathcal{B}_c = \{|0\rangle, |1\rangle\}$, which corresponds to the possible coin outcomes. Thus, the DTQW on the line takes place on Hilbert space $\mathcal{H}_c \otimes \mathcal{H}_p$.

The general state of the particle at an arbitrary time t can be described as

$$|\psi(t)\rangle = \sum_{i=0}^{1} \sum_{x=-\infty}^{\infty} \psi_{i,x}(t)|i\rangle|x\rangle, \tag{2.11}$$

where $\psi_{i,x}(t) \in \mathbb{C}$ and $\sum_i \sum_x |\psi_{i,x}(t)|^2 = 1$.

The evolution process of a DTQW is defined by an unitary transformation U applied to the state of the system after each step. This transformation is built by the composition of two unitary operators, namely the coin and the shift operators. The coin operator is an unitary operator that acts on the coin subspace $\mathcal{H}_c$, and may be represented as a (2x2) complex matrix. The Hadamard operator, defined as

$$H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}, \tag{2.12}$$

is a typical coin operator for the one-dimensional DTQW.

The shift operator is a unitary operator that acts on the position of the particle conditioned to the state of the coin. For example,

$$S = \sum_{i=0}^{1} \sum_{x=-\infty}^{\infty} |i\rangle\langle i| \otimes |x + (-1)^i\rangle\langle x| \tag{2.13}$$

is a valid shift operator. Notice that the position of the particle is increased when the coin state is $|0\rangle$, and is decreased when the coin state is $|1\rangle$.

Therefore, the unitary operator for the evolution of the quantum walk is

$$U = S(C \otimes I_p), \tag{2.14}$$

where I_p is the identity operator over the position subspace, and C is the chosen coin operator. The state of the system after t steps can be obtained by applying the unitary transformation t times after some initial state, achieving

$$|\psi(t)\rangle = U^t|\psi(0)\rangle. \tag{2.15}$$

If the position of the particle is measured after t steps, the probability of finding it at position x is given by

$$p_{x,t} = \sum_{i=0}^{1} |\psi_{i,x}(t)|^2. \tag{2.16}$$

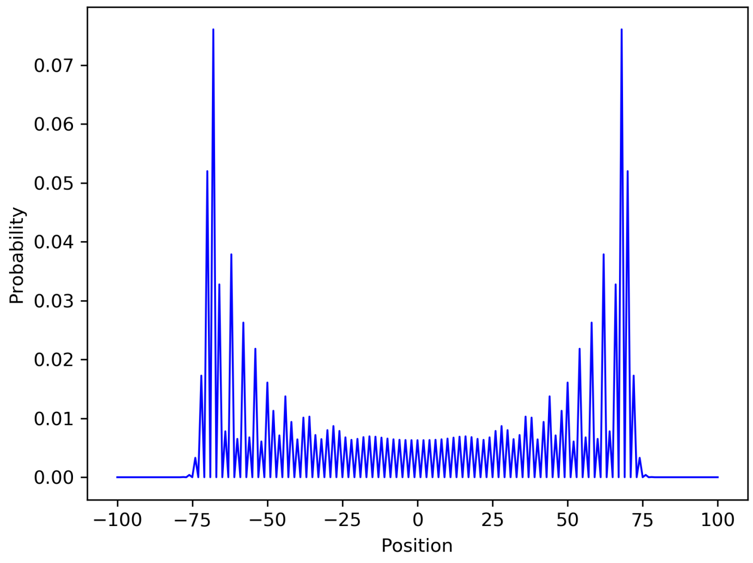

Figure 2.1: Probability distribution of a one-dimensional quantum walk with Hadamard coin after 100 steps.

On Fig. 2.1, we see the probability distribution of a quantum walk using Hadamard coin after 100 steps with initial state

$$|\psi\rangle = \frac{1}{\sqrt{2}}(|0\rangle + i|1\rangle) \otimes |0\rangle. \tag{2.17}$$

Quantum walks on two-dimensional lattices are an extension of the previous example. The position subspace is now spanned by the basis $\mathcal{B}_p = \{|x, y\rangle : x, y \in \mathbb{Z}\}$, and the coin subspace is spanned by $\mathcal{B}_c = \{|i, j\rangle : i, j \in \{0, 1\}\}$. The state of the particle at an arbitrary time t is then described as

$$|\psi(t)\rangle = \sum_{i,j=0}^{1} \sum_{x,y=-\infty}^{\infty} \psi_{i,j;x,y}(t)|i, j\rangle|x, y\rangle, \tag{2.18}$$

where $\psi_{i,j;x,y}(t) \in \mathbb{C}$ and $\sum_{i,j} \sum_{x,y} |\psi_{i,j;x,y}(t)|^2 = 1$.

The coin operator is an unitary operator that acts over the coin subspace, and may now be represented as a (4x4) matrix. The two-dimensional Hadamard coin, for example, can be achieved by the tensor product $H \otimes H$, resulting in

$$H_4 = \frac{1}{2}\begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{pmatrix}. \tag{2.19}$$

The Grover coin is another typical coin operator for DTQW on lattices of dimension two or greater and can be defined for two-dimensional lattices as $C = 2|s_c\rangle\langle s_c| - I_c$, where $|s_c\rangle = \frac{1}{2}\sum_{0\leq i,j\leq 1}|i,j\rangle$ is the uniform superposition over all coin states, and I_c is the identity operator over the coin subspace. The Grover coin is, therefore, defined as

$$G_4 = \frac{1}{2}\begin{pmatrix} -1 & 1 & 1 & 1 \\ 1 & -1 & 1 & 1 \\ 1 & 1 & -1 & 1 \\ 1 & 1 & 1 & -1 \end{pmatrix}. \tag{2.20}$$

There is a third commonly used coin for two-dimensional lattices: the Fourier coin, which is described as follows:

$$F_4 = \frac{1}{2}\begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{pmatrix}. \tag{2.21}$$

For two-dimensional lattices, the shift operator may be defined as

$$S_a = \sum_{i,j=0}^{1}\sum_{x,y=-\infty}^{\infty}|i,j\rangle\langle i,j| \otimes |x+(-1)^i, y+(-1)^j\rangle\langle x,y|. \tag{2.22}$$

From this equation, one can notice that the particle moves only on diagonal lines, illustrated on Fig. 2.2(a). When the coin evaluates $|0,0\rangle$ or $|1,1\rangle$, the particle moves through the main diagonal of the mesh, while with values $|0,1\rangle$ or $|1,0\rangle$ the particle moves through the other diagonal.

We can also define another shift operator so that the particle's movement can coincide with the mathematical grid — Fig. 2.2(b) —, applying a rotation of $\frac{\pi}{4}$:

$$S_b = \sum_{i,j=0}^{1}\sum_{x,y=-\infty}^{\infty}|i,j\rangle\langle i,j| \otimes |x+(-1)^i(1-\delta_{i,j}), y+(-1)^i\delta_{i,j}\rangle\langle x,y|, \tag{2.23}$$

where $\delta_{i,j}$ is the Kronecker delta, being evaluated to 1 if both i and j are equal, and

0, otherwise.

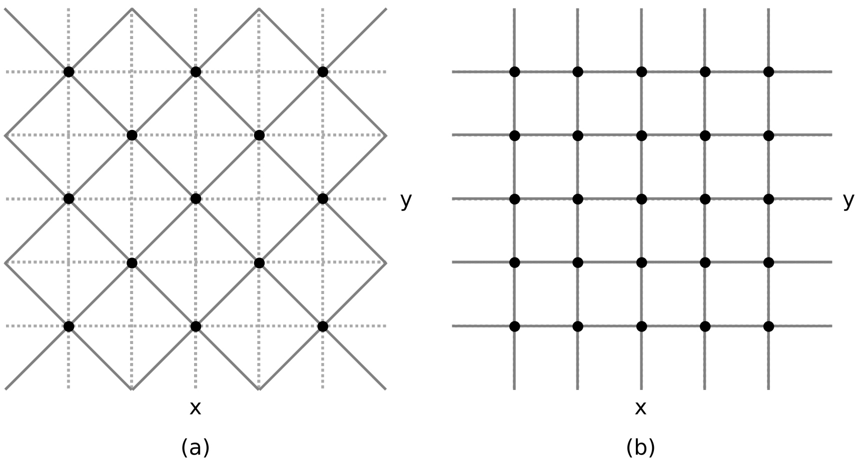

Figure 2.2: Example of diagonal and natural meshes.

The unitary operator for the evolution of the system and the process of measurement are defined analogously as they were for the one-dimensional quantum walk.

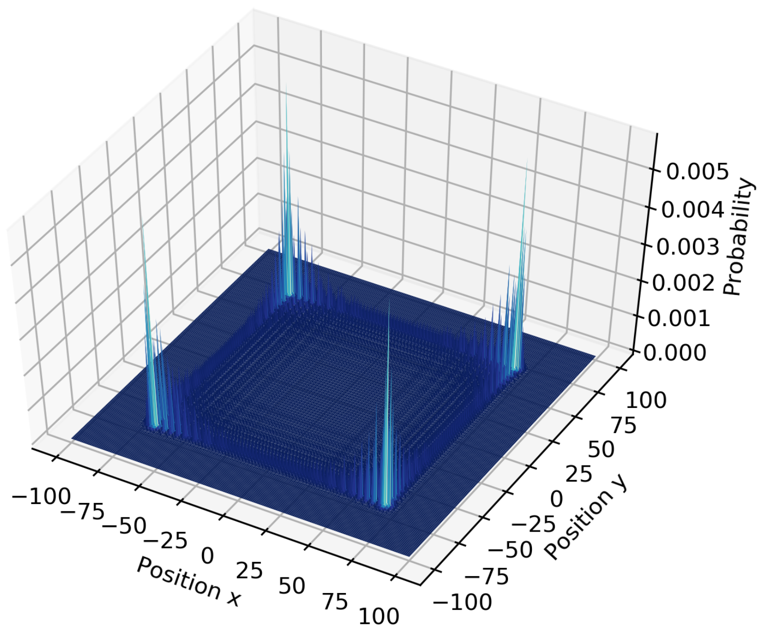

Figure 2.3: Probability distribution of a two-dimensional quantum walk with Hadamard coin after 100 steps.

As an example, on Fig. 2.3 we see the probability distribution of a quantum walk using Hadamard coin after 100 steps with initial state

$$|\psi\rangle = \frac{1}{2}(|0,0\rangle + i|0,1\rangle - i|1,0\rangle + |1,1\rangle). \tag{2.24}$$

There is also the possibility to perform quantum walks over other topologies of one- and two-dimensional meshes. We can cite, for instance, the segment and cycle meshes for the first case, presenting reflective and periodic sites on their boundaries, respectively. These topologies can also be extended for the two-dimensional case, being named as box and torus meshes. More details about these and other topologies are given by Marquezino [10].

2.2.2 Mesh Percolation

There is the possibility to consider the effects of decoherence in quantum walks by, for instance, applying the mechanism of percolations (or broken links) throughout the chosen mesh. Romanelli *et al.* [1] introduced the concepts of broken links for one-dimensional walks, consisting in, at a time t, a random site x having one or both of its links to its neighbors broken with probability p. In fig. 2.4, we have a visual representation of all possible cases of percolations in a one-dimensional mesh, where the topmost diagram represents the situation which has no percolation; the second and third diagrams depict a site x with its left and right links broken, respectively, and the last diagram shows a site with both links broken.

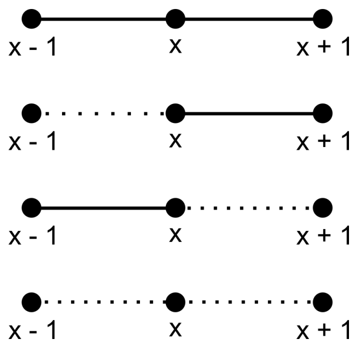

Figure 2.4: Possible cases of percolations in one-dimensional mesh. Adapted from [1].

In order to describe all those situations more formally, we can define the function

$$\mathcal{L}(i;x) = \begin{cases} (-1)^i, & \text{if the connection to } x + (-1)^i \text{ is closed} \\ 0, & \text{otherwise,} \end{cases} \tag{2.25}$$

where $i \in \mathcal{H}_c$ and $x \in \mathcal{H}_p$. Note that if $\mathcal{L}(i;x) = 0$, then $\mathcal{L}(1-i;x+(-1)^i) = 0$.

The evolution of the system must be updated accordingly, resulting in a modification of the shift operator to consider the effects of one or more percolations:

$$S_{bl} = \sum_{i=0}^{1} \sum_{x=-\infty}^{\infty} |i + \mathcal{L}(i,x)\rangle\langle i| \otimes |x + \mathcal{L}(i,x)\rangle\langle x|. \tag{2.26}$$

The decoherence effects start to happen at a time $t_d \approx 1/p$ [1], meaning that when the total time of the walk is strictly greater than this rate, the quantum properties begin to disappear and the classical behavior, the opposite. Also, as one increases the probability p, the classical behavior emerges sooner, as we can see in Fig. 2.5, which shows two walks over a line with 100 steps, simulated 10 times to get their mean probability distribution functions. The left plot exhibits the walk with $p = 0.01$, which shows the beginning of the effects of decoherence, while the right one shows the classical behavior — in this case, similar with a normal distribution — already emerged due to a higher probability of the occurrence of percolations.

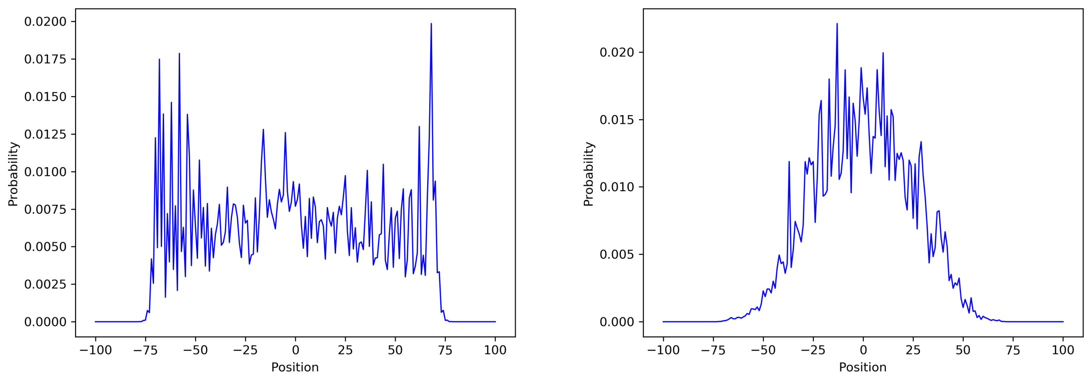

Figure 2.5: Example of quantum walks on a line with $p = 0.01$ (left) and $p = 0.1$ (right). 10 simulations were run to get their mean probability distribution functions.

The mechanism of broken links was generalized by Oliveira *et al.* [11] to consider two-dimensional meshes which, in this case, two functions are needed to describe all possible situations, one for each direction:

$$\mathcal{L}_1(i,j;x,y) = \begin{cases} (-1)^i, & \text{if the connection to } x+(-1)^i, y+(-1)^j \text{ is closed} \\ 0, & \text{otherwise} \end{cases} \tag{2.27}$$

$$\mathcal{L}_2(i,j;x,y) = \begin{cases} (-1)^j, & \text{if the connection to } x+(-1)^i, y+(-1)^j \text{ is closed} \\ 0, & \text{otherwise,} \end{cases} \tag{2.28}$$

where $i,j \in \mathcal{H}_c$ and $x,y \in \mathcal{H}_p$. Note that if $\mathcal{L}_1(i,j;x,y) = 0$, then $\mathcal{L}_1(1-i, 1-j; x + (-1)^i, y + (-1)^j) = 0$. The same must occur to $\mathcal{L}_2$. Therefore, the new shift operator is defined as:

$$\begin{aligned} S_{bl} = \sum_{i,j=0}^{1} \sum_{x,y=-\infty}^{\infty} & |i + \mathcal{L}_1(i,j;x,y), j + \mathcal{L}_2(i,j;x,y)\rangle\langle i,j| \otimes \\ & |x + \mathcal{L}_1(i,j;x,y), y + \mathcal{L}_2(i,j;x,y)\rangle\langle x,y|. \end{aligned} \tag{2.29}$$

The two-dimensional mesh percolation can also be employed on natural lattices, with the needed modifications, starting with the fact that, in this case, only one function is necessary to describe the possible situations:

$$\mathcal{L}(i,j;x,y) = \begin{cases} (-1)^i, & \text{if the connection to } x + (-1)^i(1-\delta_{i,j}), y + (-1)^i\delta_{i,j} \text{ is closed} \\ 0, & \text{otherwise,} \end{cases} \tag{2.30}$$

where $i, j \in \{0,1\}$ and, if $\mathcal{L}(i,j;x,y) = 0$, then $\mathcal{L}_1(1-i, 1-j; x + (-1)^i(1-\delta_{i,j}), y + (-1)^i\delta_{i,j}) = 0$. Hence, the shift operator for natural lattices is defined as:

$$\begin{aligned} S_{bl} = \sum_{i,j=0}^{1} \sum_{x,y=-\infty}^{\infty} & |i + \mathcal{L}(i,j;x,y), j \oplus \mathcal{L}(i,j;x,y)\rangle\langle i,j| \otimes \\ & |x + \mathcal{L}(i,j;x,y)(1-\delta_{i,j}), y + \mathcal{L}(i,j;x,y)\delta_{i,j}\rangle\langle x,y|. \end{aligned} \tag{2.31}$$

For two-dimensional walks, the relation established between the decoherence time t_d and the frequency of the broken links p is still valid, although as for this kind of quantum walk, varying coins produce different values for the standard deviation. Consequently, each coin presents different resistance to the effects of decoherence, for example, the Hadamard coin being more resistant than the Grover coin [11].

2.2.3 Multiparticle Quantum Walk

Multiparticle quantum walks have received a great deal of attention recently, because it can be used to model physical phenomena [12] and to attack important computational problems, such as being capable of universal quantum computation [13] and determining if two graphs are isomorphic [14]. The construction of the evolution operators for non-interacting multiparticle quantum walks follows the same principles described earlier for the single-particle case. After defining coin and shift operators for each particle, a final evolution operator is defined as the tensor product of the individual evolution operators. If U_1 is the evolution operator for the first particle, and U_2 the evolution operator for the second particle, then

$$U = U_1 \otimes U_2, \tag{2.32}$$

is the evolution operator for the multi-particle system.

Omar *et al.* [15], for instance, analyzed two non-interacting particles on a line, both starting at the same position but in three different coin configurations. The first configuration is

$$|\psi_0^S\rangle_{12} = |0\rangle|0\rangle_1 \otimes |1\rangle|0\rangle_2, \tag{2.33}$$

constituting a separable case where the two particles had opposing coin states. The other two cases are entangled, differing only in the relative phase

$$|\psi_0^{\pm}\rangle_{12} = \frac{1}{\sqrt{2}}(|0\rangle|0\rangle_1 \otimes |1\rangle|0\rangle_2 \pm |1\rangle|0\rangle_1 \otimes |0\rangle|0\rangle_2). \tag{2.34}$$

Ahlbrecht *et al.* [12] extended the two particle quantum walk on a line modeling a physical phenomena by considering an interaction between the particles, showing that a molecular state is formed. Thus, an operator responsible for that interaction is presented as a phase factor applied only when the particles are at the same position. That operator is defined as

$$G|i_1, x_1\rangle|i_2, x_2\rangle = \begin{cases} |i_1, x_1\rangle|i_2, x_2\rangle & \text{if } x_1 \neq x_2 \\ e^{ig}|i_1, x_1\rangle|i_2, x_2\rangle & \text{if } x_1 = x_2 \end{cases} \tag{2.35}$$

where g is a free parameter representing the interaction phase. Hence, the evolution operator for that walk must incorporate the new interaction operator so that, for each step, the effects of the interaction is taken into account. The final evolution operator is defined as

$$U = (U_1 \otimes U_2)G. \tag{2.36}$$

Due to high memory requirements, the classical simulation of interacting multi-particle quantum walks is extremely challenging even for small lattices.

2.3 High-Performance Computing with Apache Spark

Over the years, there has been a complexity increase on several computational tasks on many research fields like weather forecasting; physics simulations (classical and quantum mechanics); molecular modeling; and, more recently, data science, where knowledge is expected to be extracted from huge amounts of data by employing data visualization, statistical and machine learning techniques. Researchers have been using supercomputers/clusters in order to accomplish these tasks in a reasonable time, exploiting the parallel nature of their architectures by using message passing API (*e.g.*, MPI[1]) or parallel and distributed programming frameworks. Some of these problems require the use of Big Data processing techniques due to their data volume, variety and/or velocity characteristics.

Apache Hadoop is one of the most popular distributed frameworks for Big Data processing, providing an open source implementation of Google's MapReduce [16]. Hadoop Distributed File System (HDFS), one of its main components, makes it

[1]`http://mpi-forum.org/`

possible to easily implement a high-available, fault-tolerant, distributed file system on shared-nothing computer clusters without the need for any special hardware. However, over the last years, Hadoop has been surpassed by another framework: Apache Spark, which enhances Big Data processing tasks and is gradually replacing its predecessor. In this section, we discuss some of its main characteristics.

2.3.1 The Apache Spark Framework

According to its documentation [2], Spark is meant to be "a fast and general engine for large-scale data processing", providing:

Speed Due to its advanced execution engine and in-memory characteristic, Spark can provide 100x faster execution times than Hadoop in some cases.

Ease of use Spark allows its users to write programs in Java, Scala, Python or R, and offers interactive shells for these languages. Besides that, its core implementation contains a complete set of parallel operations.

Generality Its framework is composed of libraries that implements SQL, streaming processing, machine learning and graph processing algorithms, which can be combined in the same application code. There is also the possibility to process data stored on HDFS, Cassandra[2], HBase[3], and others.

Spark adopts a driver-workers architecture, as depicted in Fig. 2.6, where the driver is responsible for creating executors — processes that run the application — on worker nodes, and sending application code and its dependencies to them. A unit of work is called a task. Each executor executes tasks over one or more data partitions. So, data processing is highly parallelized, according to the number of nodes available. Spark needs a cluster manager to be run on, which will be responsible to acquire and set up any necessary cluster resource to the applications. By default, Spark provides a simple standalone cluster manager, but it can also be run on Apache Mesos[4] and Hadoop YARN[5].

Spark relies upon Resilient Distributed Datasets (RDD) [3]. RDD is a distributed, read-only data structure physically partitioned among executors. Each of these partitions is processed in parallel by them. Such partitioning is transparently managed by Spark, *i.e.*, application developers need not to be concerned about it. RDD are maintained in a fault-tolerant way and provide methods as transformations

[2]`https://cassandra.apache.org`
[3]`https://hbase.apache.org`
[4]`http://mesos.apache.org/`
[5]`http://hadoop.apache.org/docs/stable/hadoop-yarn/hadoop-yarn-site/YARN.html`

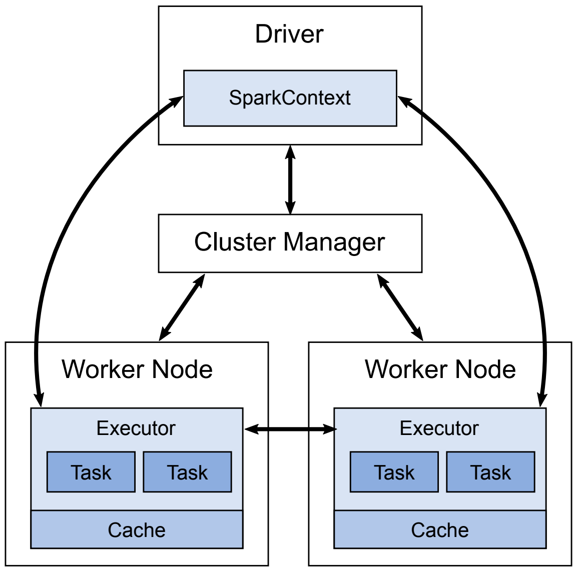

Figure 2.6: Spark's driver-workers architecture. Adapted from [2].

(map, filter, etc) and actions (collect, reduce, etc). *Transformations* can receive user-defined functions to manipulate RDD data and always produce a new RDD. On the other hand, *actions* return results to the driver. Thus, conceptually, during a typical Spark job, many RDD are created, and there exists a dependency chain between them. These dependencies are classified as *narrow* or *wide* [3]. In some transformations, each partition of a new RDD is produced from only one partition of a parent RDD. In this case, we have a *narrow* dependency between these two RDD. Other transformations produce each partition of a new RDD using data from different partitions of one or more parent RDD. In this case, the dependency between the new RDD and its parent(s) is called a *wide* dependency. Fig.2.7 presents some examples of these narrow and wide dependencies.

When an action method is called, Spark executes the necessary computations in a parallel/distributed way throughout the cluster, returning the results to the driver. Thus, transformations are not executed until an action call is reached. This lazy-evaluation model allows Spark to recover the sequence of RDD that were created in case of data loss caused by node failures. For execution, Spark creates a directed acyclic graph (DAG) of *stages* [3]. Each stage comprises a maximal set of consecutive transformations that generate RDD with narrow dependencies to their parents. This way, transformations inside a given stage can be pipelined, eliminating the need for materializing each RDD. Each stage is delimited by a transformation that generates a RDD that has a wide dependency to its parent(s). This is illustrated by the example of Fig. 2.8, where Spark would start the execution of stages 1 and 2, each one delimited by a wide dependency, and, then, move to the execution of stage 3. A wide dependency generates an operation called “shuffle”, which will be discussed later.

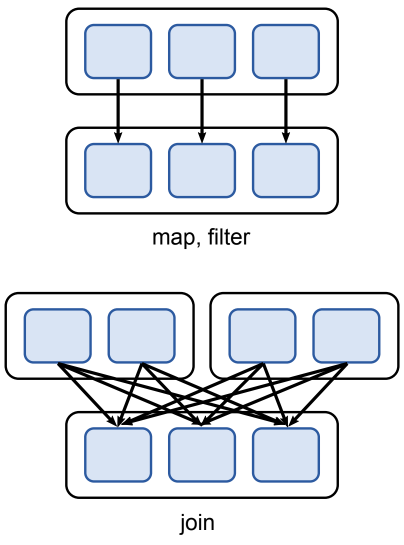

Figure 2.7: Examples of narrow (top diagrams) and wide (bottom diagrams) dependencies. Adapted from [3].

RDD can be *persisted* in different storage levels: only in main memory; only to disk or, in cases where the RDD does not completely fit in main memory, its remaining partitions are stored on disk. If a RDD is marked as persisted, when the next action is performed, Spark will compute its chain of transformations and *materialize* it, storing its data using one of the previous storage levels defined by the user. This way, the next time the persisted RDD is needed in a transformation or action, Spark will not need to recompute it, delivering better performance to iterative algorithms, for example. The storage levels also offer the possibility to replicate the partitions on two cluster nodes, providing a fault tolerant way to store the RDD. There is another way to persist a RDD: by *checkpoint*ing it. Checkpointed RDD have their dependency chain broken and their data stored in a directory, which must be informed by the user to the Spark Context object. If some variable would completely fit in the main memory of a node, Spark can also *broadcast* the desired variable across the cluster, copying its entire data to each node.

The content of the broadcasted variable can be accessed by any RDD of the application, delivering huge performance improvements for their operations.

Another important characteristic of Spark is its memory organization, that divides the Java heap-space into two regions. The first region, which by default corresponds to 60% of the heap-space, is again divided by Spark into two other regions: the execution region, reserved to all operations executed by Spark, and the storage region, used to store internal data across the cluster, like cached RDD and broadcast variables. By default, Spark creates these two regions with the same size. The rest of the heap-space is reserved mainly for user defined structures and Spark internal metadata. This characteristic of keeping as much data as possible in main memory

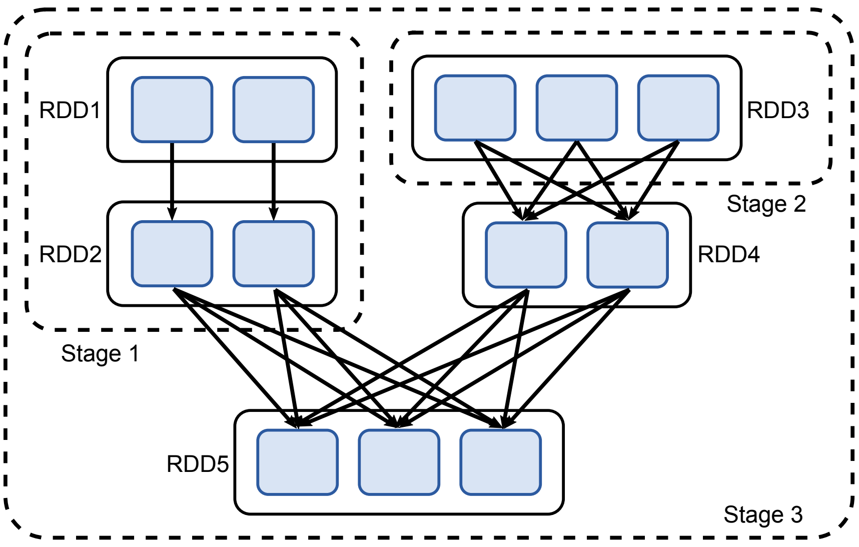

Figure 2.8: Example of execution of stages. Adapted from [3].

is what gives Spark superior performance when compared to Hadoop, which writes on disk every intermediate result produced during a MapReduce job [17].

2.3.2 Considerations when Using Spark

Despite all of those benefits, one needs to be aware of some facts when using Apache Spark in order to obtain better performance. First, transformations that generate RDD with wide dependencies to their parents have a high cost. For example, *'ByKey* methods (*e.g.*, groupByKey and reduceByKey) and some relational operations (*e.g.*, join, cartesian) typically demand data transfers between all nodes in the cluster, in contrast to transformations that generate RDD with narrow dependencies, like *filter* and *map*. This all-to-all data transfer is an operation called "shuffle" and can generate high disk and network I/O, which can heavily decrease the application performance, and should be avoided whenever possible. However, there are situations where is not possible to avoid shuffling partitions across the cluster, but the total amount of data transfered can be dramatically reduced. For example, let us consider the join operation, which produces a new RDD by combining data elements from two parent RDD that have the same key values. If none of the parent RDD have a partitioner defined, the shuffle operation will be applied on both RDD and a huge amount of data will be transferred across the cluster, as depicted in Fig. 2.9(a). In this case, by default, Spark uses a hash partitioner on both RDD's keys [18], so that data elements with the same hash key can be located at the same executor. Although, if one of the parent RDD have been previously repartitioned under a partition criterion, for instance, with the *partitionBy* transformation, the join operation will result in a child RDD with narrow dependency [4], causing a shuffle only on the other parent RDD — Fig. 2.9(b). That is because Spark knows internally that one of the parent

RDD had a partitioner defined and will distribute only the other parent RDD's data. Besides, as illustrated in Fig. 2.9(c), there are some situations where both RDD have been previously repartitioned in the same number of partitions and using the same partitioning criterion, resulting in no additional data transfers, as all data elements with equal keys have already been co-located. In cases where the RDD are generated and used only once, that strategy does not provide any gains of performance, since partitioning a RDD already requires a shuffle operation. Although, if a RDD is used multiple times, such as in iterative algorithms, it is extremely important to define a partitioner for this RDD and persist/cache it, so there will be no need to recompute it in future usages and the shuffling for its data can be avoided.

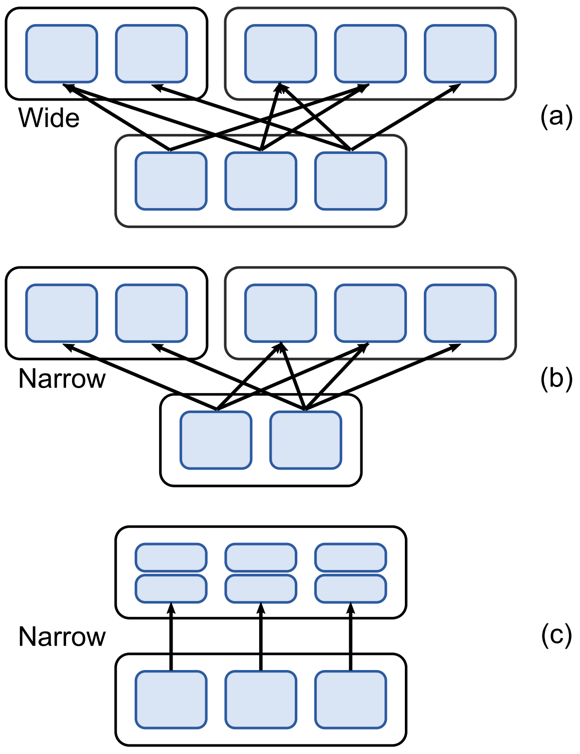

Figure 2.9: Example of possible RDD dependencies in a join transformation. Adapted from [4].

Another factor that has to be taken into account is the level of parallelism — the number of partitions — of each RDD created by the application. Spark documentation suggests it to be 2–3 times the number of computing cores in the cluster, depending on the application and on the dataset. However, when the number of partitions is too low, some irregular data distribution may occur, causing some cores to be idle waiting for the others to finish their work. This way, the cluster may not be fully utilized. Besides, depending on the size of each RDD and its partitions, the working set of the tasks can be too large, resulting in out-of-memory errors during the execution of reduce tasks or in long lasting garbage collection operations. The suggested fix for this situation is to simply increase the number of partitions, resulting in a larger number of tasks for each core in the cluster. Thus, with a higher number of partitions, the size of the working set of each task becomes smaller, reducing the memory pressure. Conversely, when the number of partitions is too high, the working set may become too small, producing a huge number of tasks that gets executed fast enough in a way that their scheduling overhead becomes relevant. The key point in this case is choosing the right ratio between the number of partitions and computing cores so the application can deliver the best performance possible.

Even though message passing APIs are commonly used for scientific computing, utilizing Spark for this purpose can make the development easier and more straightforward, mainly if the application acquires some Big Data characteristics. Many issues inherent to parallel applications like communication between nodes, fault-tolerance and data distribution are controlled by Spark. Thus, application developers can focus on their specific problems. This work uses high-performance computing with Apache Spark in order to validate its capability to simulate quantum walks and, as a future work, provide to the community a generic tool for quantum walks simulation.

2.4 Related Work

As previously stated, until a reasonably-sized universal quantum computer becomes available, researchers run numerical simulations of quantum algorithms and/or simulations of quantum circuits using classical computers. As some examples of quantum walks simulators, we can cite:

QWalk is an open-source DTQW simulator developed in C for Linux and Windows being composed of three executable files: *qw1d*, *qw2d* and *qwamplify*. The first and second executables allow the user to perform simulations over several topologies of one- and two-dimensional meshes, respectively, and provide ways of simulating decoherence by employing, for example, the mechanism of broken links [19]. The last executable is used to amplify some regions of the plots generated from the simulation of two-dimensional quantum walks for a better visualization. QWalk generates files containing the amplitude values of the final state, the probability distribution of the positions — and some of its statistics — and the *gnuplot* script that it uses to plot the probabilities. In order to use this software, the user must provide all the configuration entries of the simulation in an input file, *e.g.*, which coin, mesh and initial state will be used and, when desired, the probabilities of broken links occurrences.

HiperWalk is an open-source DTQW simulator developed in Python which takes advantages of the parallelism provided by multicore CPUs and GPUs [20]. For this, it uses the Neblina[6] programming language, which relies on OpenCL[7] as back-end. Due to the usage of graphic cards to accelerate the matrix-matrix and matrix-vector multiplications, Hiperwalk provides a huge level of parallelism, resulting in great performances for DTQW simulations. HiperWalk provides simulations of one- and two-dimensional meshes under some configuration specified by the user in an input file and also generates output files containing the amplitudes of the final state and the probability distribution along with its plot. Although HiperWalk does not offer the possibility to generate meshes with broken links, it offers simulations that QWalk does not, for example, the staggered [21] (coinless) quantum walk. In the future, HiperWalk developers aim to provide both Continuous Time and Szegedy's [22] quantum walks.

An encyclopedia of quantum information named Quantiki has a complete list containing other quantum algorithms and quantum circuits simulators available on the Internet[8]. Unfortunately, as the same as the above simulators, many of the projects on the list are capable of running simulations only on a single machine and, due to the exponential growth of the quantum structures, the simulations get restricted by that amount of available memory. To overcome this limitation, the following simulator has been developed recently:

Quandoop is a DTQW simulator developed atop of Apache Hadoop. The execution flow of Quandoop is given by, first, read, from input files, the operators and the initial state of the system; next, it performs the necessary multiplications to execute the walk; and, at the end of the simulation, generates the probability distribution. Like the other programs, the user needs to specify the characteristics of the simulation in a text file and, specially to Quandoop, the user also needs to previously build and provide all the operators. As Hadoop is developed to be properly run on a shared-nothing computer cluster, this simulator takes advantage of a greater amount of available memory, being capable of simulating larger quantum walks and a higher number of particles [8].

Notice that the aforementioned simulators complement each other in terms of functionalities and even in the number of particles in a quantum walk. The present dissertation intends to provide to the community a prototype of a DTQW simulator using HPC with the following goals:

- Gather the main functionalities already developed and combining them. The

[6]`http://www.lncc.br/~pcslara/neblina/`

[7]`https://www.khronos.org/opencl/`

[8]`https://quantiki.org/wiki/list-qc-simulators`

proposed simulator should be able to build the operators and perform simulations over different topologies for one- and two-dimensional quantum walks with the capability to consider mesh percolations and with one, two, or even a greater number of particles;

- Be extensible, allowing users with programming skills to implement, mainly, custom coins and meshes, but not restricted to these structures;
- Differently from the previous simulators, the present one expects a programmatic way to input the initial conditions of the walk, but its usage is still easy and simple, requiring few steps to start using it, as it will be shown later;

Chapter 3

Simulating Quantum Walks

In this chapter, we discuss about some characteristics of the entities of a quantum walk simulation and, in the following section, present our DTQW simulator prototype developed atop of Apache Spark, giving in-depth details of its implementation and functionalities. In the last section, we inform the necessary requisites to use the prototype and a basic set of commands to properly use it.

3.1 Our Quantum Walk Simulator

DTQW can be simulated as iterative algorithms, with each step consisting on a matrix-vector multiplication — the matrix represents the unitary evolution operator, and the vector represents the current state of the system. These entities' dimensions depend on the size of the mesh and on the number of particles being simulated, and they are constructed by means of matrix-vector and matrix-matrix multiplications, as well as tensor products.

When simulating 100 steps of a DTQW over the line, the particle can walk at most 100 positions away from the initial one. In this case, by setting the mesh with 201 positions and choosing the initial position accordingly, we can guarantee that the particle will not reach boundaries. The identity matrix in the position subspace will have dimension (201x201), and the tensor product of this matrix with a (2x2) unitary matrix results in the coin operator, with dimension (402x402). The shift operator has that same dimension, and the multiplication of both entities results in the evolution operator. Simulating a DTQW over the two-dimensional lattice is analogous. The identity matrix in the position subspace will have dimension (40,401x40,401), and the tensor product of this matrix with a (4x4) unitary matrix results in the coin operator, with dimension (161,604x161,604). As a result, the dimensions of the shift and evolution operators will also be the same. The matrices that represent the operators have cx rows and columns in the one-dimensional case, and cxy rows and columns in the two-dimensional case, as describe below:

$$row(M_{1d}) = col(M_{1d}) = cx \tag{3.1}$$

and

$$row(M_{2d}) = col(M_{2d}) = cxy, \tag{3.2}$$

where c is the dimension of the coin operator — 2 and 4, for one- and two-dimensional walks, respectively —, and x and y the size of the mesh in those coordinates. The dimension of the vectors (representing the quantum system states) is found analogously.

Regarding the sparsity of the operators, one can easily find their numbers of nonzero elements. Starting with the coin operator, we can note, from the rightmost operand of the Equation (2.14), that there is a Kronecker product between the chosen coin and an identity matrix representing the position space. Therefore, for each element of the coin, the identity matrix is replicated, resulting in a coin operator with a maximum of $4x$ nonzero elements in a one-dimensional walk and, $16xy$ nonzero elements in a two-dimensional one. For the shift operator, the Equations (2.13), (2.22) and (2.23) show a Kronecker product between the coin and position subspaces, respectively spanned by the bases $\mathcal{B}_c$ and $\mathcal{B}_p$. Thus, the number of nonzero elements of the shift operator is $2x$ and $4xy$, for one- and two-dimensional walks, respectively. As seen in equation (2.14), the multiplication done by the aforementioned operators results in the evolution operator, which sparsity is the same as the sparsity of the coin operator. Back to the previous examples, considering the 100 steps quantum walk over a line with 201 sites and using the Hadamard coin, the shift operator would have $2 * 201 = 402$ nonzero elements and the coin and evolution operators, $4 * 201 = 804$. For a quantum walk with the same number of steps, but now over a 201x201 lattice, each of these operators would have, respectively, $4 * 201 * 201 = 161,604$ and $16 * 201 * 201 = 646,146$ nonzero elements. Due to the sparsity of the operators, a quantum walk with these characteristics can be simulated in a few minutes on a single-processor, general-purpose computer by applying some optimization techniques like using special data structures to store the operators, in order to reduce memory footprint and computational time.

However, when a multiparticle quantum walk is taken into account, its evolution operator is built by applying the Kronecker product on the evolution operator of each particle, resulting in a matrix with huge dimensions. Hence, the Equations (3.1) and (3.2) can be generalized for a multiparticle quantum walk as follows:

$$row(M_{1d}) = col(M_{1d}) = (cx)^p \tag{3.3}$$

and

$$row(M_{2d}) = col(M_{2d}) = (cxy)^p, \tag{3.4}$$

where p is the number of particles in the walk. The sparsity of the operators can be found in a similar way, also raising its formulas to the power of p. Considering the interaction between particles employed by Ahlbrecht *et al.* [12], its correspondent operator — described by Equation (2.35) — can have its dimension found straightforwardly by applying the previous equations. Also, one can note that this operator can have its number of nonzero elements found by the same way as it was for the shift operator. For example, to simulate only 15 steps of a two-particle quantum walk on a two-dimensional mesh, the size of the evolution operator would be $(14,776,336 \times 14,776,336)$. As long as the mesh size or the number of particles increases, the matrices and vectors grow exponentially. Therefore, it is very difficult or even impossible to simulate such walks on a single-processor, general-purpose computer due to high memory usage and long execution time. Thus, using HPC tools, like Spark, can make it possible to simulate larger quantum walks. In order to test this hypothesis and, on a near future, to develop a generic DTQW simulator tool, a prototype[1] was developed atop of Apache Spark version 2.2.0, written in Python[2] version 3.6. The prototype is composed by a Python package containing modules that represent entities of a quantum walk, *i.e.*, mesh, coin, operator, state and probability distribution function (PDF), along with all the necessary operations.

3.1.1 Operators and States

As seen before, the base of a DTQW is composed by a initial state of the quantum system that keeps evolving after t steps, *i.e.*, t applications of an unitary operator to the state. Our prototype provides a Python module that contains classes to represent those entities, which inherit from a mathematical *Base* class, just for generality purpose. Due to Spark's in-memory characteristic, the principal idea is to maximize main memory usage, so this Base class accepts only RDD as data storage. This class also implements some utility methods to *persist*, *unpersist*, *materialize* and *checkpoint* its contained RDD. This RDD stores the data of a mathematical object in a coordinate format composed of a $(c_1, ..., c_m, v)$ tuple, where each c corresponds to a dimension index in a m-dimensional object and v, its correspondent value. Due to the sparse nature of quantum structures, when employing this representation technique, only nonzero elements can be retained, avoiding unnecessary memory usage. Regarding operators, their coordinates are described as (i, j, v) tuples, with i and j meaning the row and column numbers, respectively. As states are represented

[1]The software can be downloaded from: `https://github.com/alfabr90/dtqw`

[2]`https://www.python.org/`

by column vectors, is unnecessary to indicate their column indexes, simplifying their coordinates in a way that only the row index and value are stored. The Base class has other properties that store the shape (dimension) and the number of nonzero elements of the mathematical object that it represents. The latter property is only obtained after a materialization of the RDD which, in this implementation, uses the Spark's *count* action, providing a way to find the sparsity of the mathematical object.

We know that, when multiplying the coin operator by the shift operator, results in an unitary operator responsible for the evolution of the system, which is given by t multiplications of the initial state by this operator. Therefore, the *Operator* class implements matrix-matrix and matrix-vector multiplications. The former is based on a MapReduce algorithm described in [23]. There, two steps composed of a map and a reduce operations are necessary and, as a prerequisite, the first and second operands must be in (i, j, v_1) and (j, k, v_2) coordinate representations, respectively. The first map-reduce step is responsible for transforming the coordinates in key-value pairs of the form $(j, (i, v_1))$ and $(j, (k, v_2))$, so the column index of the first matrix can match the line index of the second. Then, a join is performed on both matrices based on the same keys generating elements in the form of $((i, k), v_1 v_2)$. The second step sums all values related to the same key — now (i, j) —, producing the elements of the resulting matrix.

The Algorithm 1 represents our implementation made with Spark, which the former step was implemented using the *join* transformation, generating $(j, ((i, v_1), (k, v_2)))$ elements, followed by a *map* transformation to produce $((i, k), v_1 v_2)$ elements by applying a user defined function — declared on line 1. For the second step, the *reduceByKey* transformation was used. This method also receives a user defined function, in this case, a summation — declared on line 4 —, that is applied to every group of elements with the same key. As matrix-vector multiplications work in a similar way, this algorithm was also used in each step of the walk. The difference here is that the map user defined function generates i keys, instead of (i, k).

The join and reduceByKey transformations create wide dependent RDD and are characterized by generating intermediate "shuffle" operations, decreasing the application performance due to the high cost of transferring all data across the nodes. As a matrix-vector multiplication must be done between the evolution operators and the actual state of the system in every step of a quantum walk, too much data would be transferred during such operations. These wide transformations could not be avoided for this implementation, so we adopt a strategy to reduce the amount of data to be transferred. After the construction of the operators, the RDD of this entity is repartitioned using the *partitionBy* method. This way, it is not necessary to transfer the entire operator's RDD in each step. Instead, only the RDD of each

Algorithm 1 Matrix-matrix multiplication

```
 1: function MAP($m$)
 2:     return $((m[1][0][0], m[1][1][0]), m[1][0][1] * m[1][1][1])$
 3: end function

 4: function SUM($a$, $b$)
 5:     return $a + b$
 6: end function

 7: function MATRIXMATRIXMULTIPLICATION($operator_1$, $operator_2$)
 8:     $rdd \leftarrow operator_1$.JOIN($operator_2$)
 9:     $rdd \leftarrow rdd$.MAP($map$)
10:     $rdd \leftarrow rdd$.REDUCEBYKEY($sum$)
11:     return OPERATOR($rdd$)
12: end function
```

state are transferred — which have much less data than the RDD of the evolution operators —, dramatically lowering the amount of disk and network I/O operations.

Also, as previously stated, an important feature of Apache Spark is RDD persistence, which stores its data in main memory — and/or disk, if desired — after an action method is called, avoiding possible unnecessary recalculations of their partitions. This feature is applied by this prototype to every operator built and, specially for the evolution operator, this avoids their recalculation for those matrix-vector multiplications during each step of the quantum walk.

3.1.2 Coins

The prototype is also composed of a module that provides classes to represent all the coins that were presented in this work: the one- and two-dimensional Hadamard coins, the Grover coin and the Fourier coin.

The implementation to build the coin operator is based on the Kronecker product between the chosen coin and the identity matrix in Equation (2.14). Even though the Kronecker product is a costly operation, in this case, each operand has characteristics that allows us to perform it in an efficient way. First, the identity matrix has only its main diagonal filled with nonzero elements, so we can store it by creating a RDD that contains only its row-indexes. Second, as the shapes of the previous coins are really small, their data can be stored in the driver node as NumPy[3] arrays and broadcast to each worker node, so it can be fast accessed by the RDD. To effectively build the coin operator, we need to replicate the identity matrix for each element of the coin and store the resulting (i, j, v) coordinates. The Algorithm 2 illustrates this process, starting with the *range* method of the SparkContext object to build

[3] `http://www.numpy.org/`

a RDD with each row-index of the aforementioned identity matrix. As the larger operand — the identity matrix — is contained in a RDD, we can apply a function to each of its elements in order to produce the above coordinates based on each element of the coin. In this case, the Kronecker product would be performed in an inverse way, but efficiently. To accomplish this, we could use the Spark's map transformation, which applies an user defined function to each element of the RDD but, as the return of the function is a collection of coordinates, we would have to flatten the result by applying an additional transformation. Instead, we use the Spark's *flatMap* transformation, which is similar to the previous one, but already flattens any returned collections. Hence, we provided to the flatMap a function (Algorithm 3) that accesses the broadcast coin and, for each of its elements, produces a (i, j, v) coordinate calculated as follows:

$$(i, j, v) = (ms + p, ns + q, C_{m,n}), \tag{3.5}$$

where s is the size of mesh, (m, n) the coordinates of each coin element and (p, q) the coordinates of each identity matrix element which, in this case, are equal and corresponded by the actual element of the RDD. Note that the multiplications ms and ns represent the replication of the identity matrix by a coin element.

Algorithm 2 Creation of the coin operator

```
Require: $meshSize$                          ▷ number of sites of the mesh
1: function CREATECOINOPERATOR($sparkContext$)
2:     $rdd \leftarrow sparkContext$.RANGE($meshSize$)
3:     $rdd \leftarrow rdd$.FLATMAP($map$)
4:     return OPERATOR($rdd$)
5: end function
```

Algorithm 3 User defined function for coin operator

```
Require: $coinSize$                          ▷ dimension of the coin
Require: $meshSize$                          ▷ number of sites of the mesh
Require: $broadcastCoin$                     ▷ coin data that has been broadcast
1: function MAP($p$)
2:     for $m \leftarrow 0, coinSize$ do
3:         for $n \leftarrow 0, coinSize$ do
4:             $i \leftarrow m * meshSize + p$
5:             $j \leftarrow n * meshSize + p$
6:             $v \leftarrow broadcastCoin[m][n]$
7:         end for
8:     end for
9:     return LIST($i, j, v$)
10: end function
```

The above implementation is employed in both one- and two-dimensional walks, since the variables *coinSize* and *meshSize* be filled accordingly to the dimension of the walk. For instance, in one-dimensional walks, they must equal, respectively, 2 and the number of sites in the x coordinate, while in two-dimensional walks, they equal 4 and the result of a multiplication of the numbers of sites in both x and y coordinates.

In order to build a custom coin, the user can easily do this extending the top-level Coin class and provide the custom coin elements in a NumPy array.

3.1.3 Meshes

Meshes are also represented by their own module inside the prototype's package and are responsible for building the shift operator.

Starting with the one-dimensional case, our implementation of the shift operator is based on Equation (2.13), where a Kronecker product is applied between two outer products: one for each coin space element and another one for each position space element. If we would have to perform all these operations separately, the shift operator could take a really long time to be built. In this case, as both outer products are applied to two elementary bases, each resulting matrix will have just one nonzero element. For example, $|0\rangle\langle 1| = \left(\begin{smallmatrix}1\\0\end{smallmatrix}\right)\left(\begin{smallmatrix}0 & 1\end{smallmatrix}\right) = \left(\begin{smallmatrix}0 & 1\\0 & 0\end{smallmatrix}\right)$. Hence, we can state that the basis element of the ket and the one of the bra will correspond to the row and column-indexes of the resulting matrix, respectively, avoiding two unnecessary outer products. As the position space can be spanned by a basis that composes an identity matrix, it can be stored in a RDD containing only its row-indexes. Considering these premises, the shift operator can be implemented in a similar way than the coin operator was, being described by Algorithm 4. The RDD of the position space can be transformed in a way that each of its elements produce a (i, j, v) coordinate for each element of the coin space. The flatMap transformation was again used, receiving a user defined function (Algorithm 5) to generate the coordinates by the following calculation:

$$(i, j, v) = (cs + (p + l) \bmod s, cs + p, 1), \tag{3.6}$$

where s is the size of mesh and c and p the elements of coin and position space bases, respectively. Just for convenience, l represents the shifting $(-1)^c$ of the particle over the mesh. As the coin space basis can also be represented as an identity matrix, it is not even necessary to broadcast its small content. Therefore, a simple loop is enough to cover its elements. Again, similar as it was for the coin operator, the product cs represents the replication of the position space element by a coin space element. One important thing to note is the modulo operation after incrementing/decrementing the position p. Even though the mesh size comprises t steps to the left and to the right, when the calculation reaches the border of the mesh, there will be no additional site to be considered by the above formula, so we must cycle the mesh for this special case.

Algorithm 4 Creation of the shift operator

```
Require: $meshSize$                                  ▷ number of sites of the mesh
1: function CREATESHIFTOPERATOR($sparkContext$)
2:     $rdd \leftarrow sparkContext$.RANGE($meshSize$)
3:     $rdd \leftarrow rdd$.FLATMAP($map$)
4:     return OPERATOR($rdd$)
5: end function
```

Algorithm 5 User defined function for shift operator of line meshes

```
Require: $coinSize$                                  ▷ dimension of the coin: 2
Require: $meshSize$                                  ▷ number of sites of the mesh
1: function MAP($p$)
2:     for $c \leftarrow 0, coinSize$ do
3:         $l \leftarrow$ POW($-1, c$)
4:         $i \leftarrow c * meshSize + ((p + l) \bmod meshSize)$
5:         $j \leftarrow c * meshSize + p$
6:         $v \leftarrow 1$
7:     end for
8:     return LIST($i, j, v$)
9: end function
```

For the two-dimensional mesh, the operations and process of building the shift operator are the same as for the previous case but, due to a higher dimension, the coin and position spaces are, each one, in composite form, as described in Equations (2.22) and (2.23). In order to consider these compositions, additional Kronecker products are needed, so we have to change the generation of the coordinates in the user defined function passed to the flatMap transformation.

For diagonal lattices, the calculations to generate the coordinates are described as follows and fully represented in the Algorithm 6:

$$\begin{pmatrix} i \\ j \\ v \end{pmatrix} = \begin{pmatrix} (c_1k + c_2)s_1s_2 + ((p_1 + l_1) \bmod s_1)s_2 + ((p_2 + l_2) \bmod s_2) \\ (c_1k + c_2)s_1s_2 + p_1s_2 + p_2 \\ 1 \end{pmatrix}. \quad (3.7)$$

Here, c_n, p_n and s_n still correspond to the coin space, the position space and the mesh size, respectively, with each subscription representing a dimension. Analogously to the one-dimensional mesh, l_1 and l_2 represent the shifting of the particle in both directions, *i.e.*, $(-1)^{c_1}$ and $(-1)^{c_2}$. Due to the higher dimension, the additional Kronecker product for the coin space is represented by $c_1k + c_2$, with k being the coin space size, while the additional one for the position space is represented by

$((p_1 + l_1) \bmod s_1)s_2 + ((p_2 + l_2) \bmod s_2)$ — for the row-index — and $p_1 s_2 + p_2$ — for the column-index. Note that the additional modulo operation is also needed to comprise the border extrapolation, as also occurred in the one-dimensional case.

For natural lattices, the previous modifications are also needed, and the process of generating the coordinates are very similar, being based on the following equation:

$$\begin{pmatrix} i \\ j \\ v \end{pmatrix} = \begin{pmatrix} \substack{(c_1k+c_2)s_1s_2+((p_1+l_1(1-\delta_{c_1,c_2})) \bmod s_1)s_2 \\ +((p_2+l_1\delta_{c_1,c_2}) \bmod s_2)} \\ (c_1k + c_2)s_1s_2 + p_1s_2 + p_2 \\ 1 \end{pmatrix}. \tag{3.8}$$

As described by the Algorithm 7, the Kronecker delta (δ_{c_1,c_2}) is implemented by a XNOR gate applied to both coin values.

Algorithm 6 User defined function for shift operator of diagonal lattices

```
Require: $coinSize$                                  ▷ dimension of the coin: 2
Require: $meshSize$                                  ▷ number of sites of the mesh
Require: $s_1$                                       ▷ number of sites of the x coordinate
Require: $s_2$                                       ▷ number of sites of the y coordinate
 1: function MAP($p$)
 2:     $p_1 \leftarrow p \bmod s_1$
 3:     $p_2 \leftarrow p/s_1$
 4:     for $c_1 \leftarrow 0, coinSize$ do
 5:         $l_1 \leftarrow$ POW$(-1, c_1)$
 6:         for $c_2 \leftarrow 0, coinSize$ do
 7:             $l_2 \leftarrow$ POW$(-1, c_2)$
 8:             $i \leftarrow (c_1 * coinSize + c_2) * meshSize + ((p_1 + l_1) \bmod s_1) * s_2 + (p_2 + l_2) \bmod s_2$
 9:             $j \leftarrow (c_1 * coinSize + c_2) * meshSize + p_1 * s_2$
10:             $v \leftarrow 1$
11:         end for
12:     end for
13:     return LIST$(i, j, v)$
14: end function
```

As our prototype also simulates the effects of decoherence by mesh percolations (broken links), the package contains a special module with a class responsible for this decoherence technique: the *RandomBrokenLinks* class. Given a probability p that represents the occurrence of broken links and the number of edges of the used mesh, the class generates a collection containing all the edges that are broken. As this method of broken links generation only considers the edges numbers, its implementation becomes extremely simple, due to the fact that it does not need to worry about whether the mesh has one or two dimensions. To accomplish this, the total number of edges is used to generate a RDD storing each edge number. This RDD is then transformed using a map operation which its user defined function checks

Algorithm 7 User defined function for shift operator of natural lattices

```
Require: $coinSize$                    ▷ dimension of the coin: 2
Require: $meshSize$                    ▷ number of sites of the mesh
Require: $s_1$                         ▷ number of sites of the x coordinate
Require: $s_2$                         ▷ number of sites of the y coordinate
 1: function MAP($p$)
 2:     $p_1 \leftarrow p \bmod s_1$
 3:     $p_2 \leftarrow p/s_1$
 4:     for $c_1 \leftarrow 0, coinSize$ do
 5:         $l \leftarrow$ POW$(-1, c_1)$
 6:         for $c_2 \leftarrow 0, coinSize$ do
 7:             $delta \leftarrow$ XNOR$(c_1, c_2)$
 8:             $i \leftarrow (c_1 * coinSize + c_2) * meshSize + ((p_1 + l * (1 - delta)) \bmod s_1) * s_2 + (p_2 + l * delta) \bmod s_2$
 9:             $j \leftarrow (c_1 * coinSize + c_2) * meshSize + p_1 * s_2$
10:             $v \leftarrow 1$
11:         end for
12:     end for
13:     return LIST$(i, j, v)$
14: end function
```

if a random generated number is lesser than p and produces a tuple composed of the correspondent edge number and a boolean value to represent the aforementioned condition. Next, we apply the filter transformation in order to retain only the edges that were, in fact, broken, *i.e.*, the boolean value holding true.

When the simulator needs to consider mesh percolations, the process of building the shift operator is modified in order to consider the broken edges. This way, the initial RDD is not built containing each mesh site, but storing each edge number. Besides, while building the operator, the Mesh class must convert the edge number to the appropriate position in order to correctly generate the coordinates of the operator. Therefore, the RDD generated by the RandomBrokenLinks class has its items collected as a Python *dict* structure using the action collectAsMap and broadcast to the cluster nodes, so it can be fast accessed by the user defined function that generates the coordinates of the shift operator. The conversion of an edge number to a position is based on the topology of the mesh and the directions of the particle's movement.

Starting with the one-dimensional meshes, we know that, when the coin is evaluated as $|0\rangle$, the particle moves to the right, incrementing its position. When $|1\rangle$, the opposite movement is performed. Hence, the edges are incrementally numbered following this left-to-right direction, starting with the leftmost edge. The last edge has the same number of the first, as it was a cycled mesh — depicted on Fig. 3.1 — to consider the border extrapolation. Concretely, to convert the edge number to a position, our implementation is based on the following calculation:

$$x = (e - c - l) \bmod s, \tag{3.9}$$

with s being the size of the mesh, c the coin value, e the current edge number and l the shifting $(-1)^c$ of the particle. This calculation factors out the particle's shifting and determines the position of the particle by identifying if it was moving to the right or to left, based on the coin state. A modulo operation is applied to consider the border edges. Hence, the user defined function described by the Algorithm 5 must be modified accordingly, now being based on Equation (2.26) and considering the conversion of edge numbers into positions, resulting in the Algorithm 8.

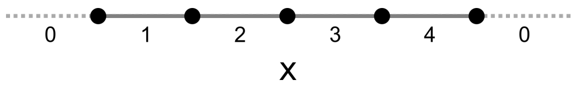

Figure 3.1: Example of edges numbering with an one-dimensional mesh of size 5.

Algorithm 8 User defined function for shift operator of line meshes with broken links

```
Require: $coinSize$                      ▷ dimension of the coin: 2
Require: $meshSize$                      ▷ number of sites of the mesh
Require: $broadcastEdges$                ▷ collection of broken links that has been broadcast
 1: function MAP($e$)
 2:     for $c \leftarrow 0, coinSize$ do
 3:         $l \leftarrow$ POW$(-1, c)$
 4:         $p \leftarrow (e - c - l) \bmod meshSize$
 5:         if $e$ in $broadcastEdges$ then
 6:             $l \leftarrow 0$
 7:         end if
 8:         $i \leftarrow (c + l) * meshSize + (p + l) \bmod meshSize$
 9:         $j \leftarrow (1 - c) * meshSize + p$
10:         $v \leftarrow 1$
11:     end for
12:     return LIST$(i, j, v)$
13: end function
```

For two-dimensional meshes, the previous principle is also used, although some adaptations must be performed. When considering the diagonal mesh, as the particle moves only diagonally, the number of sites that can be occupied by the particle is inferior than the sites of the mathematical mesh, as illustrated on Fig. 3.2(a). Also, notice that the number of edges traversed by the particle equals the number of positions of the grid. Thus, the conversion of the edge number to a position for diagonal meshes can be done as:

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} (e \bmod s_1 - c_1 - l_1) \bmod s_1 \\ (\frac{e}{s_1} - c_2 - l_2) \bmod s_2 \end{pmatrix}, \tag{3.10}$$

where s_1 and s_2 represent the sizes of each dimension, c_1 and c_2 each coin value and e, the current edge number. This calculation is similar to the one-dimensional case, with the addition of a modulo operation and a division for each cartesian coordinate. These operations are needed because, for each shifting, be it through the main diagonal or not, the edge number is incremented or decremented by s_1 units. For this case, the user defined function is described by Algorithm 9.

Algorithm 9 User defined function for shift operator of diagonal lattices with broken links

```
Require: $coinSize$                ▷ dimension of the coin: 2
Require: $meshSize$                ▷ number of sites of the mesh
Require: $s_1$                     ▷ number of sites of the x coordinate
Require: $s_2$                     ▷ number of sites of the y coordinate
Require: $broadcastEdges$          ▷ collection of broken links that has been broadcast
 1: function MAP($e$)
 2:     for $c_1 \leftarrow 0, coinSize$ do
 3:         $l_1 \leftarrow$ POW$(-1, c_1)$
 4:         for $c_2 \leftarrow 0, coinSize$ do
 5:             $l_2 \leftarrow$ POW$(-1, c_2)$
 6:             $p_1 \leftarrow (e \bmod s_1 - c_1 - l_1) \bmod s_1$
 7:             $p_2 \leftarrow ($FLOOR$(e/s_1) - c_2 - l_2) \bmod s_2$
 8:             if $e$ in $broadcastEdges$ then
 9:                 $bl_1 \leftarrow 0$
10:                 $bl_2 \leftarrow 0$
11:             else
12:                 $bl_1 \leftarrow l_1$
13:                 $bl_2 \leftarrow l_2$
14:             end if
15:             $i \leftarrow ((c_1 + bl_1) * coinSize + c_2 + bl_2) * meshSize + ((p_1 + bl_1) \bmod s_1) * s_2 + (p_2 + bl_2) \bmod s_2$
16:             $j \leftarrow ((1 - c_1) * coinSize + (1 - c_2)) * meshSize + p_1 * s_2 + p_2$
17:             $v \leftarrow 1$
18:         end for
19:     end for
20:     return LIST$(i, j, v)$
21: end function
```

When a natural mesh is considered, all the possible positions that the particle can be located at coincide with the mathematical grid, resulting in a higher number of positions in relation to the previous case, and being the double of the number of edges traversed by the particle. In this case, the particle moves vertically when both

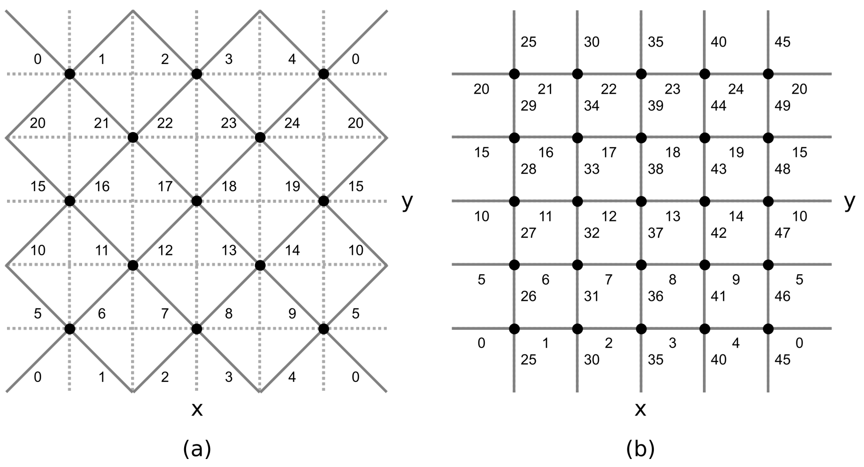

Figure 3.2: Example of edges numbering with two-dimensional meshes (diagonal and natural) of size 5x5.

coin states are equal, and moves horizontally, otherwise. The edge numbering is done for both directions separately, starting with the horizontal (x coordinate) and then, with the vertical (y coordinate), as illustrated on Fig. 3.2(b). Thus, the conversion of an edge number to a position can be done considering two conditions, as follows:

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} (\frac{e-s_1s_2}{s_1}) \\ ((e-s_1s_2) \bmod s_2 - c_1 - l) \bmod s_2 \end{pmatrix}, \tag{3.11}$$

if $e >= s_1s_2$, *i.e.*, for vertical edges, and

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} (e \bmod s_1 - c_1 - l) \bmod s_1 \\ \frac{e}{s_1} \end{pmatrix}, \tag{3.12}$$

for the horizontal ones. The user defined function for building the shift operator of a natural lattice with broken links is defined by Algorithm 10. Notice that only one coin is being considered for this case, since the calculation of the positions is based only on it. Due to this, the second coin is conditioned by the first, *i.e.*, if an edge is a vertical one, both coins are equal and, otherwise, they are different.

Similarly for coins, in order to provide extensibility, the prototype allows the user to develop his/her custom mesh, by extending the top-level Mesh class and implementing the method responsible for the shift operator building.

3.1.4 Discrete Time Quantum Walk

The *DiscreteTimeQuantumWalk* class is the main entity of the prototype, receiving the instantiated coins and meshes and being responsible for the evolution of the system, executing the quantum walk. The walk method is responsible for performing the multiplications of each state by the interaction operator, if any, and each particle's

Algorithm 10 User defined function for shift operator of natural lattices with broken links

```
Require: $coinSize$                         ▷ dimension of the coin: 2
Require: $meshSize$                         ▷ number of sites of the mesh
Require: $s_1$                              ▷ number of sites of the x coordinate
Require: $s_2$                              ▷ number of sites of the y coordinate
Require: $broadcastEdges$   ▷ collection of broken links that has been broadcast
 1: function MAP($e$)
 2:     for $c_1 \leftarrow 0, coinSize$ do
 3:         $l \leftarrow$ POW$(-1, c_1)$
 4:         if $e \geq s_1 * s_2$ then
 5:             $c_2 \leftarrow c_1$
 6:             $p_1 \leftarrow$ FLOOR$((e - s_1 * s_2)/s_1)$
 7:             $p_2 \leftarrow ((e - s_1 * s_2) \bmod s_2 - c_1 - l) \bmod s_2$
 8:         else
 9:             $c_2 \leftarrow$ not $c_1$
10:             $p_1 \leftarrow (e \bmod s_1 - c_1 - l) \bmod s_1$
11:             $p_2 \leftarrow$ FLOOR$(e/s_1)$
12:         end if
13:         $delta \leftarrow$ XNOR$(c_1, c_2)$
14:         if $e$ in $broadcastEdges$ then
15:             $l \leftarrow 0$
16:         end if
17:         $i \leftarrow ((c_1 + l) * coinSize +$ ABS$(c_2 + l, coinSize)) * meshSize + ((p_1 + l *$
    $(1 - delta)) \bmod s_1) * s_2 + ((p_2 + l * delta) \bmod s_2$
18:         $j \leftarrow ((1 - c_1) * coinSize + (1 - c_2)) * meshSize + p_1 * s_2 + p_2$
19:         $v \leftarrow 1$
20:     end for
21:     return LIST$(i, j, v)$
22: end function
```

evolution operator. This class is also responsible for building these operators.

As it has been described in Equation (2.14), the evolution operator is built by multiplying the shift operator by the coin operator. For quantum walks with only one particle, this is done really straightforwardly, because the Operator class already implements the multiplication between operators. Although, when considering multiparticle quantum walks, the walk operator is composed of Kronecker products between the evolution operator of each particle, as stated in Equation (2.32). Our implementation for this situation is different: the evolution operator of the composed system is applied as a sequence of operators, each corresponding to an individual particle. This way, as the Kronecker product is not applied, the number of columns of each particles' operator would be different than the number of rows of the system state and, therefore, it would not be possible to perform the multiplications. The solution for this case is to perform the following algebraic manipulation:

$$U_1 \otimes \cdots \otimes U_N = (U_1 \otimes I_2 \otimes \cdots \otimes I_N) \cdots (I_1 \otimes \cdots \otimes I_{N-1} \otimes U_N), \tag{3.13}$$

which is mathematically equivalent, although computationally more efficient in this case. Despite the additional matrix-vector multiplications done for each step of the walk due to this change, evolution operators become sparser than those of the original method, resulting in an overall reduced number of operations and memory usage. Fig. 3.3 exhibits a chart with the number of nonzero elements of the evolution operators for a quantum walk with two particles over a lattice. The size of each lattice dimension was derived from the number of steps, as the particle must not trespass the borders of the mesh, as to the left as to the right. The upper line of the chart contains the values of the evolution operator when it is built by direct Kronecker products with each particle's evolution operator, while the lower line contains the sum of the nonzero elements of both particles' evolution operators when built applying the algebraic manipulation of Equation (3.13).

To effectively build the set of evolution operators, our implementation consists in three parts, detailed in Algorithm 11. The first part of this code comprises the walk operator of the first particle. As its evolution operator starts the chain of Kronecker products, all the other operands consist in identity matrices, so they will result in a new identity matrix with a much bigger shape, with dimension $(row(I_n)^{N-1} \times col(I_n)^{N-1})$. Then, as described in lines 3–5, we just need to store the final identity matrix in a RDD and perform only one Kronecker product with the evolution operator of the first particle and the aforementioned identity matrix. The evolution operator must be broadcast to the cluster nodes and the RDD transformed using the flatMap operation, which its user defined function, detailed in Algorithm 12, replicates the current element of the RDD by each element of the particle's evolution

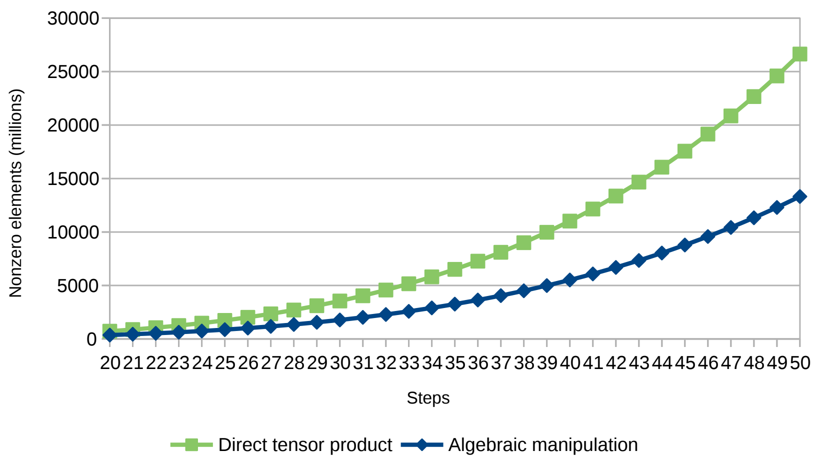

Figure 3.3: Comparison of the number of nonzero elements of evolution operators when simulating quantum walks with two particles over a lattice.

operator. The generation of the coordinates is defined as follows:

$$(i, j, v) = (ms + p, ns + q, U_{m,n}), \tag{3.14}$$

where s is the size of the resulting identity matrix, (m, n) the coordinates of each particle's evolution operator element and (p, q) the coordinates of each identity matrix element which, in this case, are equal and correspond the actual element of the RDD. The multiplications ms and ns represent the replication of the identity matrix by a particle's evolution operator element.

Algorithm 11 Creation of the walk operator

```
1: function CREATEWALKOPERATOR($sparkContext$, $particle$)
2:     if $particle = 1$ then
3:         $size \leftarrow$ number of elements of the $(N-1)$ composite identity matrices
4:         $rdd \leftarrow sparkContext$.RANGE($size$)
5:         $rdd \leftarrow rdd$.FLATMAP($mapFirst$)
6:     else
7:         $size \leftarrow$ number of elements of the $(particle-1)$ identity matrices
8:         $rdd \leftarrow sparkContext$.RANGE($size$)
9:         $rdd \leftarrow rdd$.FLATMAP($mapLast$)
10:        if $particle \neq N$ then
11:            $size \leftarrow$ number of elements of the $(N-particle)$ identity matrices
12:            $rdd \leftarrow rdd$.FLATMAP($mapOther$)
13:        end if
14:    end if
15:    return OPERATOR($rdd$)
16: end function
```

Regarding the second section of the code, in lines 7–9, we build the walk operator for the last particle of the system. This case is roughly the same of the previous,

Algorithm 12 User defined function of the walk operator for the first particle

```
Require: $size$                ▷ number of elements of the $(N-1)$ identity matrices
Require: $rows, cols$          ▷ number of rows and columns of the particle's evolution
    operator
Require: $broadcastEvolutionOperator$   ▷ particle's evolution operator data that
    has been broadcast
 1: function mapFirst($p$)
 2:     for $m \leftarrow 0, rows$ do
 3:         for $n \leftarrow 0, cols$ do
 4:             $i \leftarrow m * size + p$
 5:             $j \leftarrow n * size + p$
 6:             $v \leftarrow broadcastEvolutionOperator[m][n]$
 7:         end for
 8:     end for
 9:     return list($i, j, v$)
10: end function
```

except for the fact that the particle ends the chain of Kronecker products. Therefore, the only Kronecker product must be performed in an inverse way, with the coordinates of the operators being generated as follows:

$$(i, j, v) = (ps + m, qs + n, U_{m,n}), \tag{3.15}$$

where s is now the size of the evolution operator, (p, q) the coordinates of the resulting identity matrix of the previous particles — corresponded by the actual element of the RDD — and (m, n) the coordinates of each evolution operator element. The implementation of this case is detailed in Algorithm 13.

Algorithm 13 User defined function of the walk operator for the last particle

```
Require: $size$                ▷ number of elements of the $(N-1)$ identity matrices
Require: $rows, cols$          ▷ number of rows and columns of the particle's evolution
    operator
Require: $broadcastEvolutionOperator$   ▷ particle's evolution operator data that
    has been broadcast
 1: function mapLast($p$)
 2:     for $m \leftarrow 0, rows$ do
 3:         for $n \leftarrow 0, cols$ do
 4:             $i \leftarrow p * rows + m$
 5:             $j \leftarrow p * cols + n$
 6:             $v \leftarrow broadcastEvolutionOperator[m][n]$
 7:         end for
 8:     end for
 9:     return list($i, j, v$)
10: end function
```

The third part of our code comprises the other particles of the system. For the nth particle, two Kronecker products must be performed. The first needs to be applied to the resulting identity matrix that correspond to the $(n-1)$th particles and the current particle's evolution operator. This case is already comprised by the previous section and does not need to be reimplemented. The other Kronecker product is performed with the result of the previous operation and the other $(N-n)$ identity matrices, corresponded by lines 11–12 of the Algorithm 11. For these particles, the generation of the coordinates is based on Equation 3.14, with the following differences:

$$(i, j, v) = (ps + m, qs + m, V_{p,q}), \tag{3.16}$$

where s and m are the size and the coordinate of the resulting identity matrix for the next $(N-n)$ particles, and (p, q) and $V_{p,q}$ the coordinates and value of each element of the previous resulting matrix, which is already stored in (i, j, v) coordinates. The implementation for this case is detailed in Algorithm 14.

Algorithm 14 User defined function of the walk operator for each other particle

```
Require: $size$          ▷ number of elements of the $(N - particle)$ identity matrices
1: function MAPOTHER($p$)      ▷ $p$ is already a $(i, j, v)$ coordinate of the previous transformation
2:     for $m \leftarrow 0, size$ do
3:         $i \leftarrow p[0] * size + m$
4:         $j \leftarrow p[1] * size + m$
5:         $v \leftarrow p[2]$
6:     end for
7:     return LIST($i, j, v$)
8: end function
```

The interaction operator built by the DiscreteTimeQuantumWalk class is based on Equation (2.35), which changes the phase of the particles only when they are located at the same positions. This operator is only built when simulating multiparticle quantum walks and the free parameter g informed is nonzero. As the coin and position spaces are considered for each particle in this equation, we can use the premises mentioned in the shift operator building process to be the base of our implementation. Hence, this interaction operator works like a diagonal matrix, with its elements being e^{ig} when their row-indexes correspond to particles' positions that are equal, and 1.0, otherwise. Thus, this operator can be built by converting the row-indexes to their correspondent particles' positions and then, checking whether all of them are equal or not. Our implementation consists in storing the possible indexes — which can be found by Equations (3.1) and (3.2) — in a RDD and transforming it using a map operation to generate the (i, j, v) coordinates, as represented by Algorithm 15. Given the index, the user defined function passed to that transformation must calculate the correspondent positions of each particle and check whether they are equal to decide if the value of the element will be 1.0 or e^{ig}. From Equation 2.35, Kronecker products are applied between the particles' states, which means that, for each particle added to the system, the number of indexes in the interaction operator is multiplied by that particle's state dimension. Besides, as each particle's state is, by themselves, compositions of coin and position spaces, for higher dimensional quantum walks, additional Kronecker products are necessary to represent the bigger coin and position spaces, increasing the size of the system state. The calculation to find the positions for one- and two-dimensional walks can be performed applying the inverse of those operations, being described as follows:

$$\begin{pmatrix} x_1 \\ \vdots \\ x_n \\ \vdots \\ x_N \end{pmatrix} = \begin{pmatrix} \frac{m}{(cs)^{N-1}} \bmod s \\ \vdots \\ \frac{m}{(cs)^{N-n}} \bmod s \\ \vdots \\ m \bmod s \end{pmatrix} \tag{3.17}$$

and

$$\begin{pmatrix} (x_1, y_1) \\ \vdots \\ (x_n, y_n) \\ \vdots \\ (x_N, y_N) \end{pmatrix} = \begin{pmatrix} \left(\frac{m}{(c_1c_2s_1s_2)^{N-1}s_2} \bmod s_1, \frac{m}{(c_1c_2s_1s_2)^{N-1}} \bmod s_2\right) \\ \vdots \\ \left(\frac{m}{(c_1c_2s_1s_2)^{N-n}s_2} \bmod s_1, \frac{m}{(c_1c_2s_1s_2)^{N-n}} \bmod s_2\right) \\ \vdots \\ \left(\frac{m}{s_2} \bmod s_1, m \bmod s_2\right) \end{pmatrix}. \tag{3.18}$$

Generally speaking, according to the first equation, a position x_n is obtained by dividing the row-index m by $(cs)^{N-n}$, which corresponds to the dimension of the remaining $N-n$ composed particles' states, resulting in a new index for the actual particle. Then, the modulo s operation is performed, so that we can get the desired x_n position. The second equation just extends the previous one for a two-dimensional quantum walk, where $c_1c_2s_1s_2$ represents a bigger dimension of each state and an additional multiplication by s_2 to consider the second cartesian coordinate y of the current particle. The next step is to verify if the positions are equal. In a positive case, the generated coordinate contains the phase e^{ig} and, otherwise, 1.0. As this operator is a diagonal matrix, the coordinates i and j are the same, and correspond to the current element in the RDD. The user defined functions for one- and two-dimensional cases are represented by Algorithms 16 and 17.

Algorithm 15 Creation of the interaction operator

```
Require: $size$                                         ▷ number of indexes
 1: function CREATEINTERACTIONOPERATOR($sparkContext$)
 2:     $rdd \leftarrow sparkContext$.RANGE($size$)
 3:     $rdd \leftarrow rdd$.MAP($map$)
 4:     return OPERATOR($rdd$)
 5: end function
```

Algorithm 16 User defined function of the interaction operator for one-dimensional walks

```
Require: $phase$                                        ▷ chosen phase for the particles
Require: $coinSize$                                     ▷ dimension of the coin: 2
Require: $meshSize$                                     ▷ number of sites of the mesh
Require: $N$                                            ▷ number of particles in the system
 1: function MAP($m$)
 2:     $positions \leftarrow$ list
 3:     for $p \leftarrow 0, N$ do
 4:         $stateSize \leftarrow$ POW($coinSize * meshSize, N - 1 - p$)
 5:         $pos \leftarrow$ ROUND($m/stateSize$) mod $meshSize$
 6:         $positions$.APPEND($pos$)
 7:         if $positions[0] \neq pos$ then
 8:             return LIST($m, m, 1$)
 9:         end if
10:     end for
11:     return LIST($m, m, phase$)
12: end function
```

Algorithm 17 User defined function of the interaction operator for two-dimensional walks

```
Require: $phase$                                        ▷ chosen phase for the particles
Require: $coinSize$                                     ▷ dimension of the composed coin space: 4
Require: $meshSize$                                     ▷ number of sites of the mesh
Require: $s_1$                                          ▷ number of sites of the x coordinate
Require: $s_2$                                          ▷ number of sites of the y coordinate
Require: $N$                                            ▷ number of particles in the system
 1: function MAP($m$)
 2:     $positions \leftarrow$ list
 3:     for $p \leftarrow 0, N$ do
 4:         $stateSize \leftarrow$ POW($coinSize * meshSize, N - 1 - p$)
 5:         $posX \leftarrow$ ROUND($m/(stateSize * s_2)$) mod $s_1$
 6:         $posY \leftarrow$ ROUND($m/stateSize$) mod $s_2$
 7:         $pos \leftarrow$ LIST($posX, posY$)
 8:         $positions$.APPEND($pos$)
 9:         if $positions[0][0] \neq posX$ or $positions[0][1] \neq posY$ then
10:             return LIST($m, m, 1$)
11:         end if
12:     end for
13:     return LIST($m, m, phase$)
14: end function
```

3.1.5 Probability Distribution Functions

After the evolution of the system, the user can perform a measurement in order to get the outcome of a DTQW: the probability distribution function of the particle's position. The *State* class provides methods that find probabilities associated to certain types of measurement. These methods returns objects of the following classes: *JointPDF*, *CollisionPDF* and *MarginalPDF*. The first class represents the probability distribution found when a full measurement of the system is performed, regarding all dimensions and particles of the quantum walk; the second contains only the probability values of sites where all particles are, at the same time, located at, and the third represents the full measurement of a single particle. These classes extend the Base class, which means that they also store all the probability values in a RDD and are represented by coordinate format. Besides, all the PDF classes of the prototype provide the *plot* method, which uses matplotlib[4] as back-end to generate images with their probability values.

We know that the probability distribution can be found based on Equation (2.16), but it only describes the process for one-dimensional quantum walks and with just one particle. As our prototype provides one- and two-dimensional walks with an arbitrary number of particles, we have to develop ways to measure the system state under these configurations. Next, we provide the generalized formulas to find the probability distribution of the entire system, *i.e.*, the joint PDF, for one- and two-dimensional walks with N particles:

$$p_{x_1,\dots,x_N,t} = \sum_{n=1}^{N} \sum_{i_n=0}^{1} |\psi_{i_1,x_1;\dots,i_N,x_N}(t)|^2 \tag{3.19}$$

and

$$p_{x_1,y_1,\dots,x_N,y_N,t} = \sum_{n=1}^{N} \sum_{i_n,j_n=0}^{1} |\psi_{i_1,j_1,x_1,y_1;\dots;i_N,j_N,x_N,y_N}(t)|^2. \tag{3.20}$$

Since we have the state of system represented as (i, v) coordinates and stored in a RDD, a possible path to find the probability distribution is transforming the that RDD to convert the row-index i to particles' positions and then, summing the corresponding values v when the positions are equal. From Equations (2.11) and (2.18), one can note that a general state is composed of a Kronecker product between the coin space and the position space bases. The application of this operation directly interferes in the dimension of the system state, by means that for each Kronecker product applied by an additional coin or position space, the size of the state is multiplied by its dimension. Besides, when considering multiparticle quantum walks, the size of the state is raised to the power of the number of particles. Thus, in order to accomplish

[4]`https://matplotlib.org/`

the conversion of each row-index to a position, we must apply a transformation to the RDD with an inverse calculation of those operations as the same as it is done for the interaction operator. The Algorithm 18 details our implementation for this measurement, where the chosen method is the Spark's map transformation, and the user defined function passed to this method converts the current item of the RDD to the particles' positions based on Equations (3.17) and (3.18), for one- and two-dimensional walks. Even though the positions are properly obtained, each of them is replicated by the size of their coin space basis, *e.g.*, twice for one-dimensional walks and four times for two-dimensional ones. The next step is, therefore, to sum their corresponding values $|v|^2$ in order to get the probability values. For this, the reduceByKey transformation can be used, receiving a summation as its user defined function. As this method needs to be applied in a key-value pair RDD, the previous map transformation must produce the coordinates in the following structure: $((x_1, ..., x_N), v')$, for the one-dimensional case, and $(((x_1, y_1), ..., (x_N, y_N)), v')$, for the two-dimensional one. After the application of the reduceByKey, another map transformation is used to get the original coordinate format, by flattening the above keys. The above map transformations are detailed in Algorithms 19 and 20.

Algorithm 18 Measurement of the entire system

```
1: function MEASURESYSTEM($state$)
2:     $rdd \leftarrow state$.MAP($map$)
3:     $rdd \leftarrow rdd$.REDUCEBYKEY($sum$)
4:     $rdd \leftarrow rdd$.MAP($unmap$)
5:     return JOINTPDF($rdd$)
6: end function
```

Algorithm 19 User defined function of the measurement of the system for one-dimensional walks

```
Require: $coinSize$                          ▷ dimension of the coin: 2
Require: $meshSize$                          ▷ number of sites of the mesh
Require: $N$                                 ▷ number of particles in the system
1: function MAP($m$)          ▷ $m$ is already a $(i, v)$ coordinate of the previous
   transformation
2:     $positions \leftarrow$ list
3:     for $p \leftarrow 0, N$ do
4:         $stateSize \leftarrow$ POW($coinSize * meshSize, N - p$)
5:         $pos \leftarrow$ ROUND($m[0]/stateSize$) mod $meshSize$
6:         $positions$.APPEND($pos$)
7:     end for
8:     $v \leftarrow$ ABS($m[1]$) $*$ ABS($m[1]$)
9:     return LIST($positions, v$)
10: end function
```

Algorithm 20 User defined function of the measurement of the system for two-dimensional walks

```
Require: $coinSize$            ▷ dimension of the composed coin space: 4
Require: $meshSize$            ▷ number of sites of the mesh
Require: $s_1$                 ▷ number of sites of the x coordinate
Require: $s_2$                 ▷ number of sites of the y coordinate
Require: $N$                   ▷ number of particles in the system
 1: function MAP($m$)          ▷ $m$ is already a $(i, v)$ coordinate of the previous transformation
 2:     $positions \leftarrow$ list
 3:     for $p \leftarrow 0, N$ do
 4:         $stateSize \leftarrow$ POW($coinSize * meshSize, N - p$)
 5:         $posX \leftarrow$ ROUND($m/(stateSize * s_2)$) mod $s_1$
 6:         $posY \leftarrow$ ROUND($m/stateSize$) mod $s_2$
 7:         $pos \leftarrow$ LIST($posX, posY$)
 8:         $positions$.APPEND($pos$)
 9:     end for
10:     $v \leftarrow$ ABS($m[1]$) * ABS($m[1]$)
11:     return LIST($positions, v$)
12: end function
```

As previously stated, the CollisionPDF class is responsible to store the probability values that corresponds to the same positions where all particles are located at. Thus, to find its values, we need to measure the system for all particles when their positions are equal, as described bellow:

$$\hat{p}_{x,t} = \sum_{n=1}^{N} \sum_{i_n=0}^{1} |\psi_{i_1,x;\ldots;i_N,x}(t)|^2 \tag{3.21}$$

and

$$\hat{p}_{x,y,t} = \sum_{n=1}^{N} \sum_{i_n,j_n=0}^{1} |\psi_{i_1,j_1,x,y;\ldots;i_N,j_N,x,y}(t)|^2. \tag{3.22}$$

Here, $\hat{p}_{x,t}$ and $\hat{p}_{x,y,t}$ represent the probability to find all particles at the same position x, for one-dimensional walks, and (x, y), for two-dimensional ones, when measuring the system at time t. Instead of recalculating those values, our implementation (Algorithm 21) operates upon the RDD of a previously built JointPDF object, transforming it using the Spark's *filter* operation, in order to retain only the probability values where all particles' positions are equal. This method receives a condition statement through an user defined function, which is applied to each element of the RDD and creates a new one containing only the elements that the condition statement validates as true. For this case, the condition statement is really simple, looping through the positions of all particles — $(x_1, ..., x_N)$ or $((x_1, y_1), ..., (x_N, y_N))$ — checking if they are equal. Notice that there is no need to keep all particles' positions, as they are equal, so a map transformation is used to produce a RDD with the position of just one particle with its correspondent value. The filter and map operations are detailed in Algorithms 22 and 23. An important consideration to be done is that the elements of that RDD do not sum 1.0, as every probability distribution function must do. Although, it can be normalized in order to acquire this characteristic.

Algorithm 21 Measurement of the collisions

```
1: function MEASURESYSTEM($jointPDF$)
2:     $rdd \leftarrow jointPDF$.FILTER($filter$)
3:     $rdd \leftarrow rdd$.MAP($map$)
4:     return COLLISIONPDF($rdd$)
5: end function
```

Algorithm 22 User defined functions of the measurement of the collisions for one-dimensional walks

```
Require: $N$                                  ▷ number of particles in the system
 1: function FILTER($m$)           ▷ $m$ is $(x_1, ..., x_N, v)$ coordinate of the previous transformation
 2:     for $p \leftarrow 0, N$ do
 3:         if $m[0] \neq m[p]$ then
 4:             return false
 5:         end if
 6:     end for
 7:     return true
 8: end function

 9: function MAP($m$)              ▷ $m$ is $(x_1, ..., x_N, v)$ coordinate of the previous transformation
10:     return LIST($m[0], m[N]$)
11: end function
```

The last class that represent probability distributions provided by the prototype is the MarginalPDF. This class stores the probability distribution of an arbitrary particle and the process of this partial measurement is very similar to the one that results in a JointPDF object. Here, the difference is that only the positions of the particle k being measured are taken into account, as described by the following equations:

$$\tilde{p}^{(k)}_{x,t} = \sum_{n=1}^{N} \sum_{i_n=0}^{1} \sum_{\substack{x_n=0, \\ n \neq k}}^{size-1} |\psi_{i_1,x_1;\ldots;i_N,x_N}(t)|^2 \tag{3.23}$$

and

$$\tilde{p}^{(k)}_{x,y,t} = \sum_{n=1}^{N} \sum_{i_n,j_n=0}^{1} \sum_{\substack{x_n,y_n=0, \\ n \neq k}}^{size-1} |\psi_{i_1,j_1,x_1,y_1;\ldots;i_N,j_N,x_N,y_N}(t)|^2. \tag{3.24}$$

Algorithm 23 User defined functions of the measurement of the collisions for two-dimensional walks

```
Require: $N$                                   ▷ number of particles in the system
 1: function FILTER($m$)   ▷ $m$ is a $(x_1, y_1, ..., x_N, y_N, v)$ coordinate of the previous transformation
 2:     for $p \leftarrow 0, N, step \leftarrow 2$ do
 3:         if $m[0] \neq m[p]$ or $m[1] \neq m[p+1]$ then
 4:             return false
 5:         end if
 6:     end for
 7:     return true
 8: end function

 9: function MAP($m$)   ▷ $m$ is a $(x_1, y_1, ..., x_N, y_N, v)$ coordinate of the previous transformation
10:     return LIST($m[0], m[1], m[2 * N]$)
11: end function
```

From the previous equation, $\tilde{p}^{(k)}_{x,t}$ and $\tilde{p}^{(k)}_{x,y,t}$ represent the probability to find the k-th particle at position x, for one-dimensional walks, and (x, y), for two-dimensional ones, when measuring the system at time t. Therefore, our implementation — based on Algorithms 19 and 20 — just need to find the positions of the desired particle and, consequently, compose the keys for the RDD with these positions. The reduceByKey is still necessary to sum all values of the state that correspond to the particle being measured. In particular for the two-dimensional case, a map operation is performed to flatten the (x, y) position, similar to the one that was used during the measurement of the entire system.

3.2 How to Use

In order to properly use our prototype, the user must have the Apache Spark downloaded[5] and unzipped on his/her computer. As our prototype is implemented in Python, another way to get Spark is installing it with *pip*, the recommended tool for installing Python packages. To do this, the user needs to have Python and pip installed and execute the following command in a terminal: *pip install pyspark==2.2.0.post0*. Note that Spark 2.2.0 needs Java version 8 to be run and Python versions 2.7+ and 3.4+, so these prerequisites must be satisfied.

As previously informed, Spark can be launched using different resource managers, but the simplest way to do so is using its Standalone Mode. The user can manually start the master and worker nodes, but its easier to use the launch scripts provided by Spark [24]. In order to start the slaves, the worker nodes URL must be provided

[5] `https://spark.apache.org/downloads.html`

in the conf/slaves file inside Spark directory. Besides, the unzipped Spark directory must be accessible by each cluster node. To launch a custom application, such as a Python script using our prototype to simulate quantum walks, the user may execute the *spark-submit* script, passing, as arguments, the driver URL, the Python script location and some configuration parameters, if desired. It is worth noting that, for instance, when using our prototype, its package files must be in the same directory of the user's Python script, so they can be imported. A good way to test the applications is executing the spark-submit script without configuring any worker nodes and setting "local[*]" as the master URL. This way, Spark is launched in a single machine, allocating all its available cores. More information about spark-submit can be found in [25].

When simulating quantum walks with our prototype, the user must write a Python script where the SparkContext class must be instantiated, a coin and a mesh must be chosen, the initial state of the system must be created and, then, the DiscreteTimeQuantumWalk class instantiated. Although, first of all, their correspondent packages must be imported. This can be done as illustrated by Listing 1, where the correspondent modules of the Hadamard coin for two-dimensional walk and the diagonal lattice were imported.

Listing 1 Importing the necessary prototype packages

```python
# These ones will be necessary further ahead
import math
import cmath

from pyspark import SparkContext
from dtqw.coin.coin2d.hadamard2d import Hadamard2D
from dtqw.mesh.mesh2d.diagonal.lattice import LatticeDiagonal
from dtqw.math.state import State
from dtqw.dtqw import DiscreteTimeQuantumWalk
```

With the necessary packages imported, the user can instantiate the aforementioned classes. The first to be instantiated is the SparkContext. This class is the access point to the Spark's functionalities and, consequently, all classes provided by the prototype need it to work. Next, the user can instantiate the coin and the mesh, providing its size, as detailed in Listing 2. For one-dimensional walks, the Mesh class expects an integer, while for two-dimensional ones, it is necessary to provide a Python tuple containing the size for each cartesian coordinate. Namely for lattices, the prototype already considers the possible number of steps of the particle to the left and to the right and adjust itself to the correct size. Besides, the user must not provide a size smaller than the number of steps, so the particle does not trespass the borders of the mesh.

Listing 2 Instantiating the SparkContext, coin and mesh

```
sparkContext = SparkContext()

# In this example, the walk will last 30 steps
size = steps = 30

coin = Hadamard2D(sparkContext)
mesh = LatticeDiagonal(sparkContext, (size, size))
```

Now, the user must create the initial state of the system. As previously stated, operators and states are stored in RDD and have their elements represented as coordinates. Therefore, the state must be created from a RDD containing its coordinates. The user can accomplish this by building a collection of coordinates and passing it to the SparkContext *parallelize* method, which returns a RDD containing the parallelized data. This step of the prototype usage is exemplified in Listing 3.

Listing 3 Creating the initial state of the system

```
# Example of two particles in an entangled state
# The particle will start the walk from the middle of the mesh
# |psi> = (|1,1>|0,0>|0,0>|0,0> - |0,0>|0,0>|1,1>|0,0>) / sqrt(2)
num_particles = 2

# The indexes are calculated based on the coins and positions of
# both particles
i1 = 193836192
i2 = 27697263
v = 1.0 / math.sqrt(2)

coordinates = ((i1, v), (i2, -v))

rdd = sparkContext.parallelize(coordinates)

# As this is a two-particle quantum walk, the shape of the state
# comprises both particles' shapes
coin_size = coin.size
mesh_size = mesh.size[0] * mesh.size[1]
shape = (coin_size * mesh_size * coin_size * mesh_size, 1)

initial_state = State(rdd, shape, mesh, num_particles)
```

Next (Listing 4), the DiscreteTimeQuantumWalk class is instantiated, receiving the coin and mesh chosen by the user, the number of particles in the system and the number of partitions, which will be applied to the operators and states generated by this class. The *walk* method is, then, called to perform the walk, based on the steps and the initial state provided. The collisional phase is also passed to this method so that the DiscreteTimeQuantumWalk object can build the interaction operator.

Listing 4 Executing the discrete time quantum walk

```
# The operators and states will be partitioned in 2400 chunks
num_partitions = 2400

dtqw = DiscreteTimeQuantumWalk(
        sparkContext, coin, mesh, num_particles, num_partitions
)

# The collision phase for the interaction operator
phase = cmath.pi

final_state = dtqw.walk(steps, initial_state, phase)
```

Finally, after doing the walk, the final state of the system can be measured in order to provide its probability distributions. The user just need to call the method *measure* of the State class to get all the PDF objects. Listing 5 illustrates this process, along with the plotting of the distributions.

Listing 5 Measuring the final state and plotting the PDF

```
joint_pdf, collision_pdf, marginal_pdf = final_state.measure()

# There is no need to call the plot method
# for the joint probability as its dimension is higher than two.
# In this case (two particles on a two-dimensional mesh),
# it is represented by a four-dimensional array
collision_pdf.plot('./collision_measurement')
marginal_pdf[0].plot('./particle1_measurement')
marginal_pdf[1].plot('./particle2_measurement')
```

Chapter 4

Experiments

This chapter details the evaluation of our approach of using Apache Spark for simulating DTQW. In the first section, we describe the adopted methodology when building and profiling the sets of experiments that helped us to do this evaluation, and show the execution environment specifications where the experiments were run. Next, we present and analyze the results of the simulations.

4.1 Methodology and Execution Environment

In order to evaluate our approach of using Apache Spark to simulate DTQW, we built some sets of simulations of two interacting particles walking on diagonal lattice.

The first set of experiments consists in performing short walks simulations utilizing few worker nodes and different levels of parallelism. The objective here is to know how Spark behaves with a reasonable low amount of data and to identify some relevant characteristic related to the number of partitions for DTQW simulations. Quantum walks of 10, 15 and 20 steps were performed, each one utilizing one, two and three worker nodes. Besides, each simulation was replicated considering different levels of parallelism, *i.e.*, the number of partitions, which was applied a range of 1–8 times the total number of cores available for each simulation.

With the next set of experiments, we intend to check how Spark scales when increasing the number of steps of the quantum walks while also increasing the amount of available resources. Here, we choose to incrementally double the resources, but we could not do this for the number of steps, as the structures of quantum walks grow exponentially based on this number. Instead, we had to calculate, in respect to each number of steps, the number of nonzero elements for the biggest structures — the both particles' evolution operators and the interaction operator — and select the approximate steps that produced a doubling-rate of the total number of elements. This set of experiments comprises simulations of 20, 25, 30, 35, 42 and 50 steps quantum walks utilizing one, two, four, eight, 16 and 32 worker nodes, respectively.

Each of the above configurations used a number of partitions equal to eight times the number of available cores.

A third set of experiments was built in order to observe the speedup produced by Spark for quantum walks simulations. To do so, we used the same number of worker nodes of the previous setups, while keeping the size of the dataset of a 30 steps quantum walk. For most of the setups, an eight partitions-per-core multiplier was used, similar as it was for the previous experiments, except for the scenarios that one and two worker nodes were selected, where higher multipliers had to be used due to the large amount of data being processed when having a reduced amount of memory available. This is needed to ensure that the working set processed by Spark is not big enough to cause out-of-memory errors. For both of these cases, the total number of partitions was 2400, when only one worker node was used, and 1440, for two nodes.

In the last set of experiments, we attempt to evaluate the capability of Spark to process huge amounts of data with a reduced amount of available resources. For this scenario, we ran simulations of quantum walks with 30, 40 and 50 steps, on one, two and four worker nodes, respectively. As a few number of worker nodes were used, the large data produced by these configurations would not be highly divided if we chose a small number of partitions, what could generate memory errors. Therefore, we chose a very high set of multipliers (50, 75 and 150), resulting in 2400, 14400 and 28800 partitions, in order to guarantee that all data would be divided in small chunks.

For each setup of the experiments described above, we profiled the execution times and the memory usage of each operator building, each step done and each of the probability distributions calculated. Also, all the experiments were executed five times, so the mean and the standard deviation of those both metrics could be obtained.

All the simulations were run on Lobo Carneiro supercomputer, which contains 252 nodes connected by a 56 Gbs InfiniBand network. Each node has two Intel Xeon E5-2670v3 processor summing up 48 cores due to HyperThreading and 64 GB RAM. All nodes share a 500 TB disk with Intel Lustre parallel file system. On Lobo Carneiro, the job submission is handled by PBS-Pro[1]. Lobo Carneiro is located at High Performance Computer Center (NACAD) - COPPE, Federal University of Rio de Janeiro (UFRJ). As each node provides 60 GB of RAM to be used, 50 GB were allocated for the executor's Java Virtual Machine (JVM), so the 10 GB left could be used by Python processes. The used Python interpreter was the Intel Python[2] interpreter version 3.6.2. No configuration parameter regarding Spark's memory

[1]`http://pbspro.org/`

[2]`https://software.intel.com/en-us/distribution-for-python`

organization was changed, assuming its defaults.

4.2 Results

We start this section presenting the results for the first set of experiments, which is composed by a collection of setups regarding simulations of shorter walks using one, two and three worker nodes, varying the number of partitions by a range of 1–8 times the number of cores. The following charts exhibit, for each number of partitions, three groups of operations. The first constitutes in the total time to build all operators, with the legend "Operators building". The second set of operations represents the execution time to perform the evolution of the system, *i.e.*, all the matrix-vector multiplications. This set is named as "Walk execution". The last one is the elapsed time to calculate the joint and each particle's marginal PDF, and find the collision probabilities, being labeled as "Measurement". Each group contains the mean and the standard deviation of its profiled time.

Fig. 4.1, 4.2 and 4.3 exhibit the profiled execution times for quantum walks with 10, 15 and 20 steps, respectively, when just one worker node was used. In the 10 steps case, we notice, for the first half of partitions-per-core multipliers, that their corresponding execution times started to increase when the multiplier also gets higher. When the simulations were performed using the other half of multipliers, the prototype delivered low execution times, except for the last scenario.

Regarding the 15 steps walk, we can note that the execution time of the three groups of operations did not change too much for the selected numbers of partitions, presenting small variations in their overall execution times. The lowest values were achieved when the first two multipliers were used, although, in general, for this case, any of the proposed level of parallelism would be appropriate.

When the quantum walk becomes longer (20 steps), and utilizing just one worker node, a noticeable gain in performance was achieved when the number of partitions was increased to the double of the number of cores, but the significant improvement happened when this multiplier valued three or more. This is a scenario where Spark delivers better performance when the number of partitions is increased. Using a higher number of partitions facilitates a better data division, providing a better distribution of partitions for the tasks among executors, what helps to generate a better utilization of the cores.

The previous setups were again executed, although now using two worker nodes, as illustrated on Fig. 4.4, 4.5 and 4.6. This time, for the 10 steps simulations, the overall execution times were delivered when the multipliers valued two and eight. The other cases produced a linear increasing time. Here, we highlight the best performances with simulations that used 96, 288 and 672 partitions.

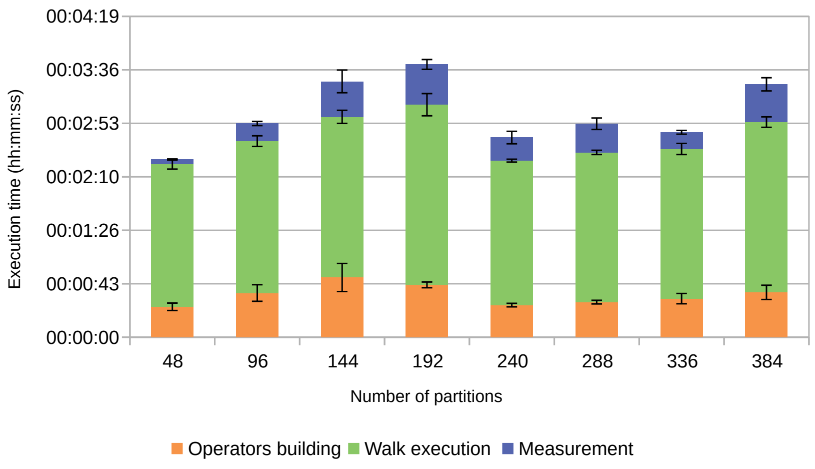

Figure 4.1: Execution times for 10 steps quantum walks on one worker node with different number of partitions.

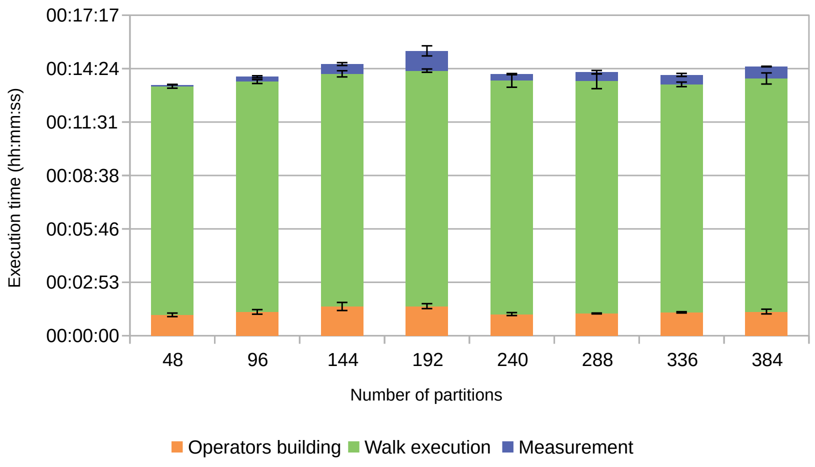

Figure 4.2: Execution times for 15 steps quantum walks on one worker node with different number of partitions.

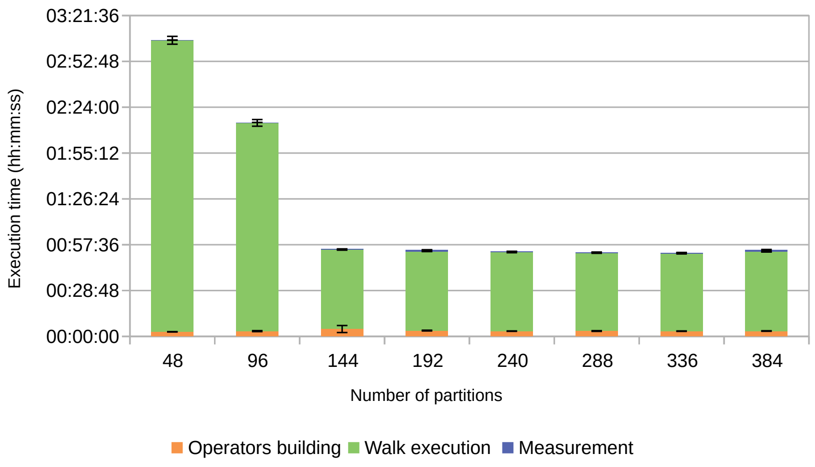

Figure 4.3: Execution times for 20 steps quantum walks on one worker node with different number of partitions.

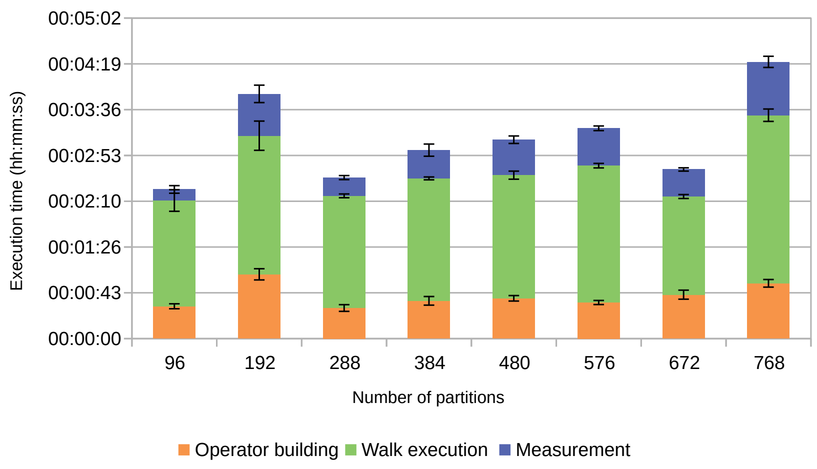

Figure 4.4: Execution times for 10 steps quantum walks on two worker nodes with different number of partitions.

The 15 steps simulations exhibit the case where Spark delivers the best execution times when the smallest multipliers were used and, as long as the number of partitions was being increased, the execution times were also getting higher. Here, the available resources were sufficient to comfortably handle simulations, requiring only few partitions to deliver the best performances.

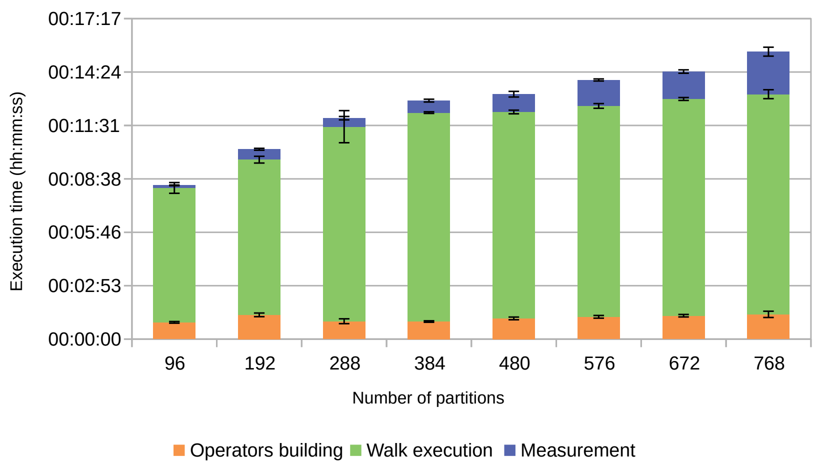

Figure 4.5: Execution times for 15 steps quantum walks on two worker nodes with different number of partitions.

Similar as it happened when one worker node was used, the 20 steps simulations presented a huge performance gain when moving from 96 to 192 (or higher) partitions, with the lowest execution times being produced by the 2–3 partitions-per-core multipliers. Two worker nodes being used for this problem size was not enough to produce better performance when the number of particles equals the number of available cores. It is worth noting that the execution times started to increase for the highest multipliers.

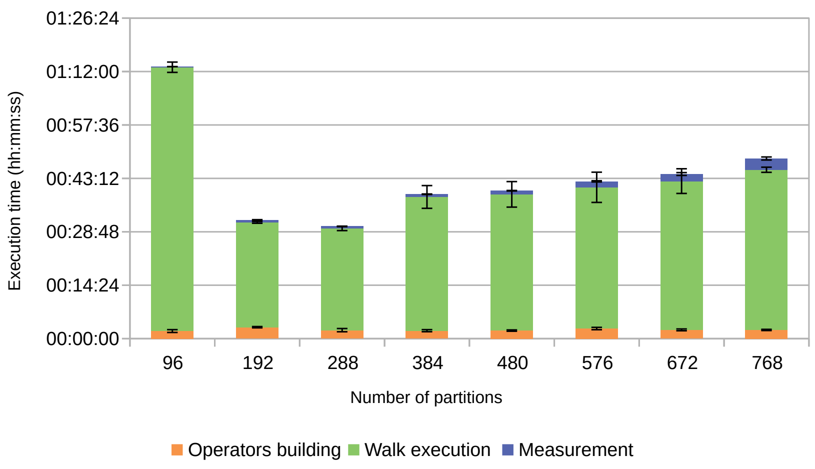

Figure 4.6: Execution times for 20 steps quantum walks on two worker nodes with different number of partitions.

The third group of simulations comprises the use of three nodes and has the execution times of its setups presented on the following charts. Fig. 4.7 contains the times for the 10 steps quantum walks simulations, where Spark presented, once more, a linear increasing execution times for the majority of the numbers of partitions. Although, when using the greatest numbers of partitions — multipliers seven and eight —, the times to execute the walk were the lowest of these setups. When using 2–3 partitions-per-core multipliers, Spark delivered the best overall performance, considering the three groups of operations ("Operators building", "Walk execution" and "Measurement").

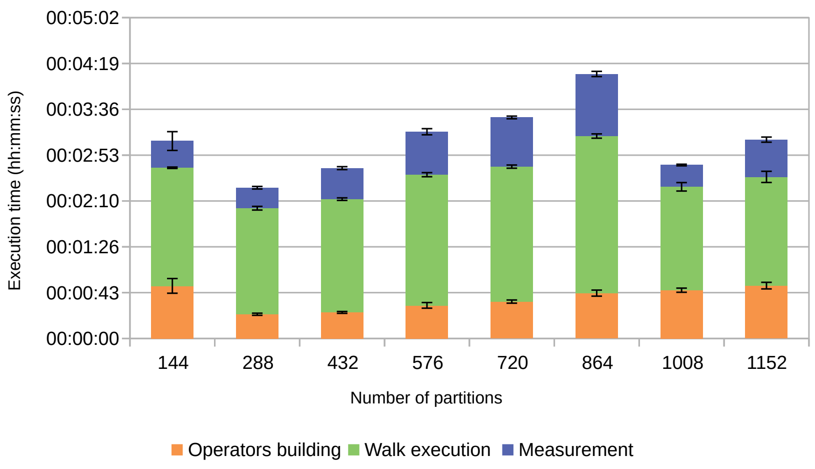

Figure 4.7: Execution times for 10 steps quantum walks on three worker nodes with different number of partitions.

Observing Spark's behavior for 15 steps quantum walks on Fig. 4.8, we noticed a very similar shape in comparison to their equivalent setups when using two worker nodes applying 1–6 multipliers. The previous situation that presented good overall performance for seven and eight multipliers occurred again, but much more evident for the former rate.

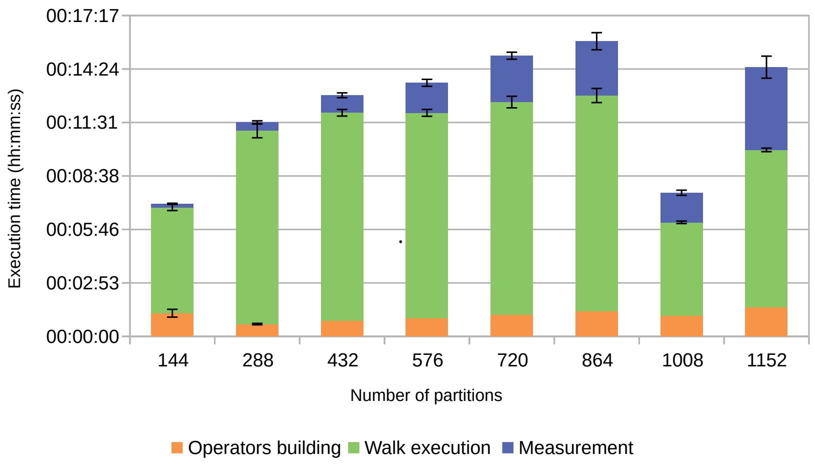

Figure 4.8: Execution times for 15 steps quantum walks on three worker nodes with different number of partitions.

With 20 steps quantum walks (Fig. 4.9), the best overall execution times started right away with the first partitions-per-core multiplier, linearly increasing as this rate was also getting higher. For the last multipliers, a good performance was again delivered.

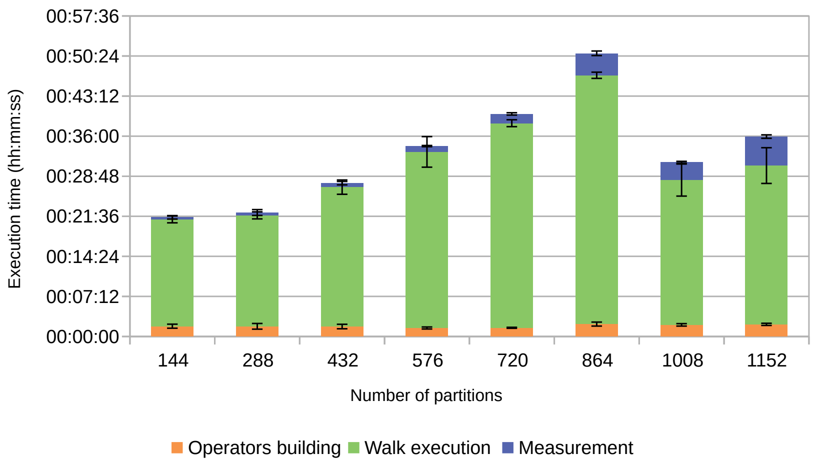

Figure 4.9: Execution times for 20 steps quantum walks on three worker nodes with different number of partitions.

The next set of experiments intends to check whether Spark is capable to provide a reasonable scalability when doubling the available resources. To accomplish this, we chose a set of steps which its correspondent numbers of nonzero elements could increase in an approximate doubling-rate. The number of partitions for each configuration was defined as eight times the number of cores, in an attempt to produce working sets with an appropriate size to avoid possible out-of-memory errors. Using these scenarios, Spark was able to scale-up with no execution issues.

We profiled the memory footprint for these setups, depicted in Fig. 4.10, which shows that all the operators could fit into main memory, more precisely, the Spark's storage region, which is responsible to store the cached RDD and broadcast variables. Interestingly, for the three largest walks, the required memory to store the operators, final state and all the PDF, did not keep up with the double-increasing rate of the available memory, what could allow us to simulate even largest walks for these specific setups.

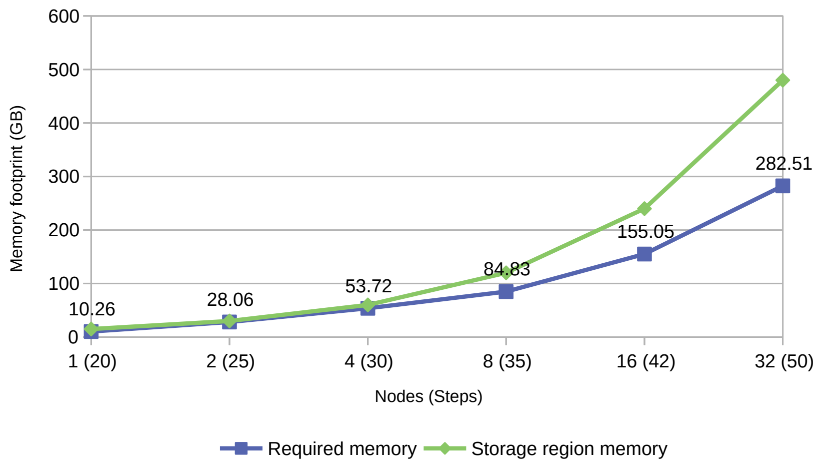

Figure 4.10: Memory footprint for the scalability experiments.

Even though Spark showed us its capability to scale-up with no problems in relation to the number of stored elements, the execution times of the simulations did not keep in a constant rate, which is required to achieve a linear scalability. Although, Spark produced execution times in a nearly linear rate, as can be seen in Fig. 4.11. The exception of this case was a particular situation that occurred for the 30 steps walk — using 192 cores (4 worker nodes) —, which its execution time was even higher than the next configuration, and presented a higher standard deviation than all of the other setups. Even though all the functionalities of the prototype are parallelized, increasing only the number of worker nodes, and consequently, the number of cores and the amount of memory, was not enough to attain a linear scalability.

The third set was intended to see how is the speedup produced by Spark, where we fixated a 30 steps quantum walks while doubling the number of working nodes, from one to 32. From Fig. 4.12, we see that Spark produced a sub-linear speedup, although presenting results close to the linear (ideal) when utilizing one, two, four and eight worker nodes. The performance started to degrade as a higher number of worker nodes was used, indicating that the amount of data was not enough to require a higher utilization of the cluster. We also highlight that the provided speedup when

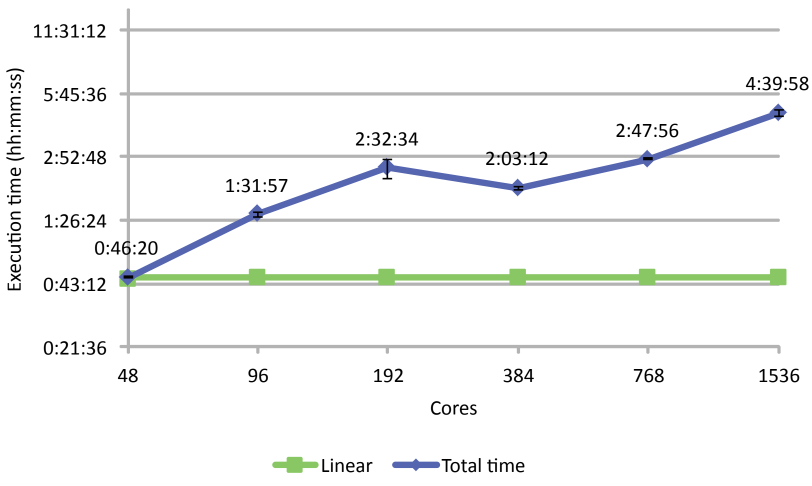

Figure 4.11: Scalability of Spark when doubling the number of cores and the problem size.

utilizing 32 worker nodes was roughly the same as when using eight worker nodes.

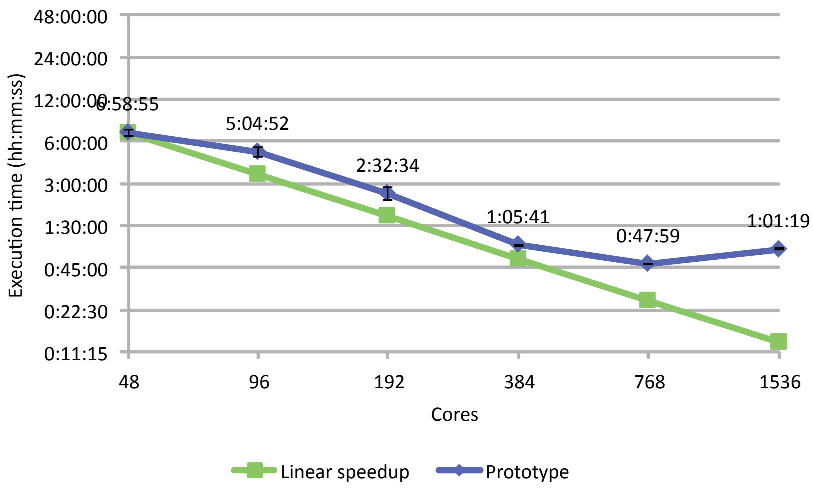

Figure 4.12: Speedup produced by Spark for a 30 steps quantum walk, while doubling the number of cores.

We also plot the efficiency for those experiments on Fig. 4.13. From this chart, one can see that Spark provided, for the majority of the number of nodes, an above-average efficiency, with the best values of almost 70% when utilizing two and four worker nodes, and 80% for eight nodes. A loss of efficiency is recognized when selecting a higher number of nodes, with values of 55% and 21%, corroborating with the reduced speedup of the previous chart.

Moving to the last set of experiments, we aimed to discover whether Spark is capable of handling large datasets while having to work with few computational resources. For a higher increase in the number of nodes, an even higher increase in the number of nonzero elements were chosen. As the expected number of elements

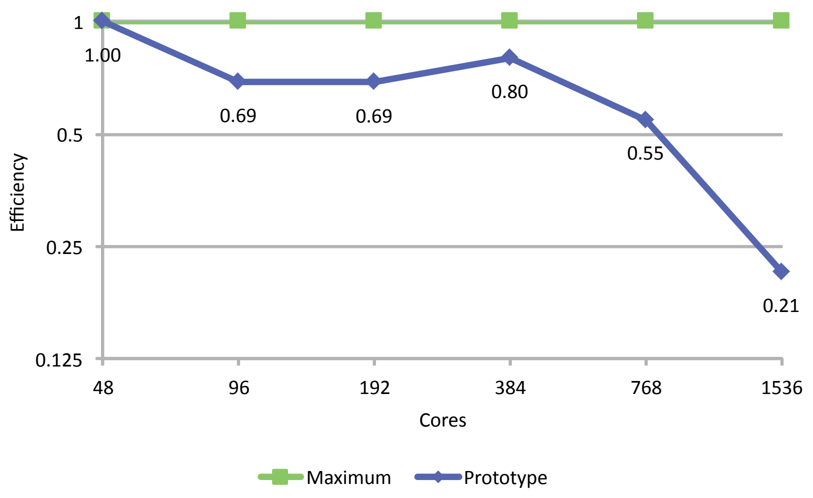

Figure 4.13: Efficiency of Spark for a 30 steps quantum walk, while doubling the number of cores.

for these experiments would require a higher amount of memory in comparison with what would be available, we had, once more, to choose very high numbers of partitions, so the performed transformations on the RDD could produce small working sets, avoiding any memory issues. On Fig. 4.14, we plot the required memory for these quantum walks and the available memory of Spark's storage region for each configuration. As we can see, the required memory for this set of experiment is too much larger than the available space, which resulted in Spark spilling to disk the partitions that could not fit into their storage region. Fig. 4.15 exhibits the profiled execution times for these setups, which presented long lasting simulation times as expected, due to the reduced number of cores to process these quantum walks and the necessity of Spark to read the partitions that were spilled to disk, instead of accessing them directly from the memory.

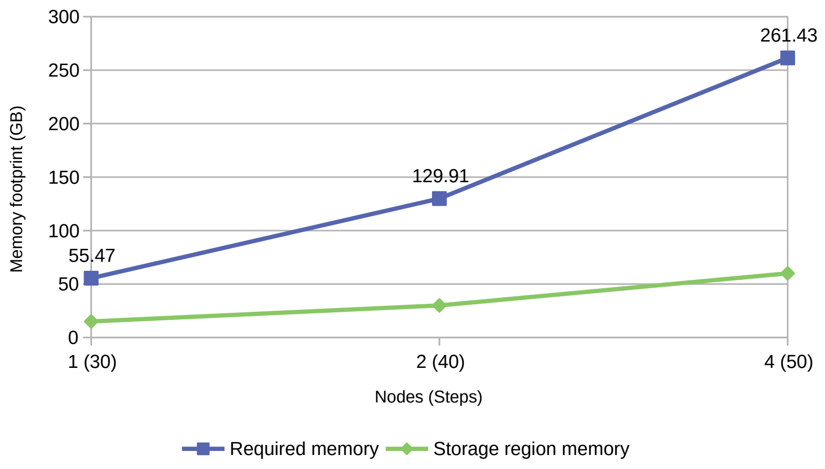

Figure 4.14: Memory footprint of large quantum walks processed by Spark when possessing few computational resources.

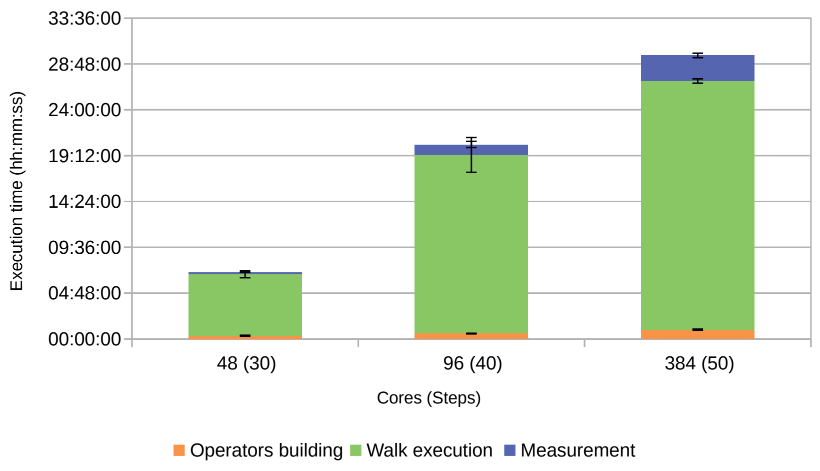

Figure 4.15: Execution times of large quantum walks processed by Spark when possessing few computational resources.

Chapter 5

Final Remarks

In this work, we used Apache Spark to simulate quantum walks with two interacting particles on two-dimensional lattices, which can be considered as a problem with Big Data characteristics due to the exponential growth of their data structures. To do so, a prototype was developed atop of Apache Spark to validate its capability to simulate larger quantum walks and to provide a generic simulation tool for this kind of application.

This chapter starts with the conclusions and some observations regarding the usage of Apache Spark to simulate DTQW. Next, future work comprising additional features and possible enhancements to the developed prototype are presented.

5.1 Conclusions

To evaluate our approach, an extensive set of experiments were run on a HPC cluster. The first experiments intended to observe how Spark behaves when simulating short quantum walks when few worker nodes were utilized. Also, we applied a range of partitions-per-core multipliers for each setup in order to determine the best numbers of partitions for these smaller scenarios. Here, we noted that, in general, for small problem sizes and using few computational resources (computing cores and available memory), the number of partitions recommended by Spark, *i.e.*, 2–3 times the number of cores, are sufficient to achieve the best performances. Of course, depending on the available memory and processing power, this ratio (partitions-per-core) must be changed accordingly. For instance, when using fewer nodes and the problem size gets larger, a higher number of partitions becomes necessary in order to Spark successfully complete the simulations and delivering better performance.

A few other configurations were used to know whether Spark is capable to scale-up when both the problem size and the number of cores were increased. For this set of experiments, we noticed that Spark could handle all problem sizes with no execution issues. In order to present a linear scalability, the execution times of all these configurations should be kept constant. However, that was not possible using our prototype and Spark's default configurations.

Another set of experiments were executed to observe the speedup values provided by Spark, consisting in keeping constant the problem size, while doubling the number of worker nodes. Here, Spark presented better results. Even though we achieved a sub-linear speedup, when utilizing one, two, four and eight worker nodes, the prototype presented speedup values close to the ideal (linear). Also, we profiled the efficiency of Spark considering the available resources. According to the results, Spark allowed our prototype to achieve, in general, an above-average efficiency, with the highest value of 80%.

A few last simulations were performed in an attempt to check if Spark could handle large datasets while possessing few computational resources. The simulations were run successfully when using an appropriate number of partitions. Here, longer execution times were expected, due to a low number of computing cores and to the available memory not being enough to hold all the dataset.

To conclude, we noticed that Spark delivered good performance when simulating quantum walks and handled most simulations without issues. Moreover, our simulator allowed the size of quantum walk instances to be increased beyond the possibilities of single-processor, general-purpose computers.

5.2 Future Work

For future work, we first propose an extension of the experiments that considered the scalability and speedup produced by Spark. Even though reasonable results regarding these metrics were achieved with the present sets of experiments, the new setups could comprise different partitions-per-core multipliers, in order to detect if using different numbers of partitions would result in better or worse performances for those metrics. Consequently, this would indicate what configurations could be considered to achieve better scalability and a speedup even closer to the linear.

Regarding the implementation of our prototype, some enhancements can still be made in order to offer a more complete set of features. First, a possible extension is to add more modules in the code to consider other types of meshes. For instance, to consider DTQW over general graphs. GraphX[1], the Apache Spark's API for graphs, may be employed to that purpose. Even though the prototype requires few steps to run a quantum walk simulation, we propose a construction of a web interface to integrate with the our software in order to provide a more user-friendly utilization, specially for researchers with few programming knowledges. Another option for these users, is the capability of the prototype to read input files with the

[1]`http://spark.apache.org/graphx/`

characteristics of the quantum walk to be simulated. Allowing the simulation of continuous-time or staggered quantum walks are also important additions to our simulator. Besides, the simulator can be extended to consider another important quantum algorithm, differentiating from the quantum walks presented so far: the Grover's search algorithm for unstructured databases, which searches for a marked element quadratically faster than the classical counterpart. A last contribution to our simulator can be the implementation of a resource manager, which would store the resources information of the cluster being used and, based on some heuristic and given some characteristics of the simulation, generate more accurate numbers of partitions. This addition may help to accomplish the previously suggested set of experiments.

Bibliography

[1] ALEJANDRO, R., SIRI, R., ABAL, G., et al. “Decoherence in the quantum walk on the line”, *Physica A: Statistical Mechanics and its Applications*, v. 347, pp. 137–152, 03 2005.

[2] THE APACHE FOUNDATION. “Spark 2.2.0 Documentation”. 2017. Available at: <`https://spark.apache.org/docs/2.2.0/`>. Online.

[3] ZAHARIA, M., CHOWDHURY, M., DAS, T., et al. “Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing”. In: *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pp. 15–28. USENIX, 2012. ISBN: 978-931971-92-8.

[4] HOLDEN KARAU, R. W. *High Performance Spark*. Sebastopol, CA, USA, O’Reilly Media, 2017.

[5] FEYNMAN, R. P. “Simulating Physics with Computers”, *International Journal of Theoretical Physics*, v. 2, n. 6/7, 1982.

[6] MONTANARO, A. “Quantum algorithms: an overview”, *npj Quantum Information*, v. 2, n. 15023, 2016.

[7] VENEGAS-ANDRACA, S. E. “Quantum walks: a comprehensive review”, *Quantum Information Processing*, v. 11, n. 5, pp. 1015–1106, 2012.

[8] DAVID SANTOS DE SOUZA, FRANKLIN DE LIMA MARQUEZINO, A. D. A. B. L. “A Classical Simulator of Quantum Walks on Computer Clusters”. In: *4th Conference of Computational Interdisciplinary Sciences*, pp. 84(1)–84(10), 2016.

[9] PORTUGAL, R. *Quantum walks and search algorithms*. New York, Springer Science & Business Media, 2013.

[10] MARQUEZINO, F. L. *Análise, simulações e aplicações algorítmicas de caminhadas quânticas*. Doctoral Thesis, National Laboratory for Scientific Computing, Petrópolis, Brazil, 2010.

[11] OLIVEIRA, A. C., PORTUGAL, R., DONANGELO, R. "Decoherence in Two-Dimensional Quantum Walks", *Physical Review A*, v. 74, n. 012312, 07 2006.

[12] AHLBRECHT, A., ALBERTI, A., MESCHEDE, D., et al. "Molecular binding in interacting quantum walks", *New Journal of Physics*, v. 14, n. 073050, 2012.

[13] RUDINGER, K., GAMBLE, J. K., WELLONS, M., et al. "Noninteracting multiparticle quantum random walks applied to the graph isomorphism problem for strongly regular graphs", *Phys. Rev. A*, v. 86, n. 022334, Aug 2012.

[14] CHILDS, A. M., GOSSET, D., WEBB, Z. "Universal Computation by Multiparticle Quantum Walk", *Science*, v. 339, n. 6121, pp. 791–794, 2013. doi: 10.1126/science.1229957.

[15] OMAR, Y., PAUNKOVIĆ, Y., L, S., et al. "Quantum Walk on a Line with Two Entangled Particles", *Physical Review A*, v. 74, n. 042304, 2006.

[16] DEAN, J., GHEMAWAT, S. "MapReduce: Simplified Data Processing on Large Clusters", *6th Symposium on Operating System Design and Implementation*, pp. 137–150, 2004.

[17] SHI, J., QIU, Y., MINHAS, U. F., et al. "Clash of the Titans: MapReduce vs. Spark for Large Scale Data Analytics", *Proc. VLDB Endow.*, v. 8, n. 13, pp. 2110–2121, 2015.

[18] ZAHARIA, M., KARAU, H., KONWINSKI, A., et al. *Learning Spark*. Sebastopol, CA, USA, O'Reilly Media, 2015.

[19] MARQUEZINO, F., PORTUGAL, R. "The QWalk Simulator of Quantum Walks", *Computer Physics Communications*, v. 179, pp. 359–369, 04 2008.

[20] LEÃO, A. B. *Um novo simulador de alta performance de caminhadas quânticas*. Master Thesis, National Laboratory for Scientific Computing, 2015.

[21] FALK, M. D. "Quantum Search on the Spatial Grid", 03 2013.

[22] SZEGEDY, M. "Quantum Speed-Up of Markov Chain Based Algorithms". In: *Proceedings of the 45th Annual IEEE Symposium on Foundations of Computer Science*, FOCS '04, pp. 32–41, Washington, DC, USA, 2004. IEEE Computer Society. ISBN: 0-7695-2228-9. doi: 10.1109/FOCS.2004.53.

[23] ULLMAN, J. D., RAJARAMAN, A., LESKOVEC, L. *Mining of Massive Datasets*. Cambridge, England, Cambridge University Press, 2014.

[24] THE APACHE FOUNDATION. “Spark Standalone Mode”. 2017. Available at: <`https://spark.apache.org/docs/2.2.0/spark-standalone.html`>. Online.

[25] THE APACHE FOUNDATION. “Submitting Applications”. 2017. Available at: <`https://spark.apache.org/docs/2.2.0/submitting-applications.html`>. Online.