QUANDOOP: UM SIMULADOR PARA CAMINHADAS QUÂNTICAS COM APOIO DE COMPUTAÇÃO PARALELA E DISTRIBUÍDA

David Santos de Souza

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
Fevereiro de 2016

QUANDOOP: UM SIMULADOR PARA CAMINHADAS QUÂNTICAS COM APOIO DE COMPUTAÇÃO PARALELA E DISTRIBUÍDA

David Santos de Souza

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.

Examinada por:

____
Prof. Franklin de Lima Marquezino, D.Sc.

____
Prof. Renato Portugal, D.Sc.

____
Prof. Ricardo Cordeiro de Farias, D.Sc.

RIO DE JANEIRO, RJ – BRASIL
FEVEREIRO DE 2016

Souza, David Santos de

Quandoop: Um Simulador para Caminhadas Quânticas com Apoio de Computação Paralela e Distribuída/David Santos de Souza. – Rio de Janeiro: UFRJ/COPPE, 2016.

XI, 59 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, 2016.

Referências Bibliográficas: p. 56 – 59.

1. Caminhadas Quânticas. 2. Map/Reduce. 3. Computação de Alto Desempenho. 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.

Aos meus pais por sempre me apoiarem.

Agradecimentos

Primeiramente a minha mãe e ao meu pai, que apesar das dificuldades, me apoiaram e me deram suporte para que eu pudesse seguir em frente e concluir mais este trabalho.

Aos professores Alexandre de Assis Bento Lima e Franklin de Lima Marquezino pelos conselhos e orientações que tornaram esse trabalho possível.

A todos que contribuíram, direta ou indiretamente, para a conclusão dessa dissertação.

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

QUANDOOP: UM SIMULADOR PARA CAMINHADAS QUÂNTICAS COM APOIO DE COMPUTAÇÃO PARALELA E DISTRIBUÍDA

David Santos de Souza

Fevereiro/2016

Orientadores: Alexandre de Assis Bento Lima
Franklin de Lima Marquezino

Programa: Engenharia de Sistemas e Computação

A computação quântica é uma área da ciência que vem se destacando nos últimos anos. Porém, como a construção de computadores quânticos para fins práticos ainda é um desafio tecnológico, por enquanto os pesquisadores dessa área precisam realizar simulações numéricas. Com cada vez mais pesquisas sendo realizadas nessa área, surge a necessidade de ferramentas que permitam aos cientistas realizarem suas simulações. A simulação desses experimentos é uma forma concreta de obter os resultados desejados, porém a sua execução pode requerer o processamento de uma grande quantidade de dados. O uso de ferramentas que permitam o processamento desses dados com alto desempenho é uma forma de viabilizar essa tarefa. Esta dissertação apresenta uma abordagem que utiliza computação de alto desempenho com memória distribuída para permitir a execução dessas simulações e um simulador genérico voltado para a área de computação quântica.

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

QUANDOOP: A SIMULATOR FOR QUANTUM WALKS SUPPORTED BY PARALLEL AND DISTRIBUTED COMPUTING

David Santos de Souza

February/2016

Advisors: Alexandre de Assis Bento Lima
Franklin de Lima Marquezino

Department: Systems Engineering and Computer Science

Quantum computing is an area of science that has been growing in recent years. However, such the building of quantum computers for practical purposes still a technological challenge, for while researchers in this field need to perform numerical simulations. With more and more research being done in this area, there is a need for tools that enable scientists to conduct their simulations. The simulation of these experiments is a tangible way to get the desired results, but its execution may require processing a large amount of data. The use of tools that allow the processing of this data with high performance is a way to make feasible this task. This dissertation presents an approach that uses high-performance computing with distributed memory to allow the execution of these simulations and a generic simulator aiming to the area of quantum computing.

Sumário

Lista de Figuras x

Lista de Tabelas xi

1 Introdução 1

1.1 Motivação 2

1.2 Organização 2

2 Caminhadas Quânticas e o Arcabouço Map/Reduce 4

2.1 Mecânica Quântica 4

2.2 Computação Quântica, Algoritmos Quânticos e Caminhada Aleatória 7

2.2.1 Computação Quântica 7

2.2.2 Algoritmos Quânticos 9

2.2.3 Caminhada Aleatória 9

2.3 Caminhadas Quânticas 9

2.4 O Arcabouço Map/Reduce 11

2.4.1 Hadoop MapReduce 13

2.4.2 HDFS 14

3 Uso do Map/Reduce para Simulação de Caminhadas Quânticas com Dois Caminhantes numa Malha Bidimensional 17

3.1 Trabalhos Relacionados: Caminhadas Quânticas 18

3.1.1 QWalk 18

3.1.2 HiPerWalk 18

3.1.3 QuIDDPro 19

3.1.4 QuTiP 19

3.2 Trabalhos Relacionados: Hadoop 20

3.2.1 Inversão Geofísica 20

3.2.2 Riscos Agregados 20

3.3 Simulação de uma Caminhada Quântica com Dois Caminhantes numa Malha Bidimensional 21

4 Quandoop **25**
4.1 Operações Básicas . . . 25
4.1.1 Produto de Kronecker . . . 27
4.1.2 Multiplicação de Matrizes . . . 28
4.1.3 Norma de uma Matriz . . . 29
4.1.4 Valor Absoluto ao Quadrado . . . 30
4.1.5 Remodelagem . . . 31
4.1.6 Soma sobre os Eixos . . . 31
4.2 Simulador . . . 32

5 Experimentos **34**
5.1 Ambiente de Execução . . . 35
5.1.1 Software . . . 35
5.1.2 Hardware . . . 35
5.1.3 Hadoop: Ajuste nas Configurações para Otimização de Desempenho . . . 35
5.2 Resultados: QW . . . 39
5.3 Resultados: Quandoop . . . 44
5.4 Resultados: Grover . . . 50

6 Conclusão **52**
6.1 Contribuições . . . 53
6.2 Trabalhos Futuros . . . 54

Referências Bibliográficas **56**

Lista de Figuras

2.1 Diagrama de execução de um algoritmo de *wordcount* utilizando Hadoop . . . 12
2.2 Diagrama de execução do Hadoop usando o *Hadoop Streaming* 13
2.3 Diagrama do tipo de entrada e saída do Hadoop . . . 14
2.4 Diagrama da arquitetura básica do Hadoop . . . 15
2.5 Diagrama da arquitetura do HDFS [9] . . . 16

5.1 Gráfico com o tempo gasto para a execução do QW . . . 41
5.2 Gráfico com o armazenamento gasto com os arquivos de entrada do QW . . . 42
5.3 Comparação dos tempos totais para a execução do código QW sequencial e da versão para Hadoop em cenários onde não ocorrem "estouro" na memória RAM utilizando o código sequencial. . . . 43
5.4 Speedup para size=30. . . . 43
5.5 Eficiência para size=30. . . . 44
5.6 Speedup para size=42. . . . 45
5.7 Eficiência para size=42. . . . 45
5.8 Distribuição de probabilidade do vetor de estados para *size* igual a 23 após 12 passos. . . . 46
5.9 Distribuição de probabilidade do vetor de estados para *size* igual a 42 após 21 passos. . . . 47
5.10 Gráfico com o tempo gasto para a execução do Quandoop . . 49
5.11 Gráfico com o tamanho dos arquivos de entrada do Quandoop 49
5.12 Comparação dos tempos totais para a execução do código sequencial e do Quandoop (escala logarítmica) em cenários onde não ocorrem "estouro" na memória RAM utilizando o código sequencial. . . . 50

Lista de Tabelas

5.1 Tempo de execução da simulação da caminhada quântica com dois caminhantes numa malha bidimensional utilizando o código sequencial. 39
5.2 Tempos para a execução da simulação da caminhada quântica com dois caminhantes numa malha bidimensional utilizando o código para Hadoop. 40
5.3 Desvio padrão e erro padrão para os tempos totais da simulação da caminhada quântica com dois caminhantes numa malha bidimensional utilizando o código para Hadoop. 40
5.4 Tempos para a execução da simulação da caminhada quântica com dois caminhantes numa malha bidimensional para size=30 utilizando quantidades diferentes de nós. 41
5.5 Tempos para a execução da simulação da caminhada quântica com dois caminhantes numa malha bidimensional para size=42 utilizando quantidades diferentes de nós. 42
5.6 Speedup e Eficiência para os dados da Tabela 5.4. 42
5.7 Speedup e Eficiência para os dados da Tabela 5.5. 44
5.8 Tempo de execução da simulação da caminhada quântica com quatro caminhantes numa linha utilizando o código sequencial, após os dados serem gerados. 47
5.9 Tempos para a execução da simulação da caminhada quântica com quatro caminhantes numa linha utilizando o Quandoop. 48
5.10 Desvio padrão e erro padrão para os tempos totais da simulação da caminhada quântica com dois caminhantes numa malha bidimensional utilizando o código para Hadoop. 48
5.11 Comparação da execução do algoritmo de Grover com n=15. 51

Capítulo 1

Introdução

O crescente avanço tecnológico provoca mudanças no nosso dia a dia assim como nas indústrias e nos centros de pesquisas. Cada vez mais cientistas utilizam tecnologias computacionais para realizarem seus experimentos. Uma das áreas de pesquisa que andou avançando bastante nos últimos tempos é a área da computação quântica. Um dos lapsos ainda presente nessa área é que os cientistas ainda não possuem computadores quânticos para executarem seus experimentos. Devido ao fato do *hardware* quântico ainda ser muito limitado, eles precisam realizar simulações numéricas utilizando computadores clássicos. Porém nem sempre é viável realizar essa simulação devido aos limites impostos pelo *hardware* dos computadores clássicos. E, apesar de ser possível realizar simulações de computação quântica em computadores clássicos, elas são ineficientes.

A computação quântica é uma área de pesquisa recente, com seu desenvolvimento iniciado na década de 80. Como apresentado por Nielsen e Chuang [1], esse desenvolvimento foi impulsionado por uma mudança na perspectiva sobre sistemas quânticos, que provocou um ressurgimento do interesse nos fundamentos da mecânica quântica, além de criar novas questões combinando física, ciência da computação e teoria da informação.

Um dos focos das pesquisas relacionadas com computação quântica está no desenvolvimento de um computador quântico. Porém, mesmo os pesquisadores ainda não possuindo um computador quântico, eles continuam realizando avanços nessa área, através de técnicas que os permitem comprovar suas hipóteses, como por exemplo, provas matemáticas, ou através da simulação dos seus experimentos. Um dos motivos que tornam a computação quântica atraente aos pesquisadores é o paralelismo quântico, que pode ser usado para resolver problemas matemáticos de uma forma mais eficiente.

1.1 Motivação

De acordo com o problema que se esteja tentando resolver, o processamento de uma grande quantidade de dados pode estar envolvido na obtenção da solução. Isso, juntamente com a característica das simulações de computação quântica em computadores clássicos serem ineficiente, torna ambientes de Computação de Alto Desempenho (HPC, do inglês *High Performance Computing*) propensos de serem utilizados como meio para viabilizar e acelerar a execução dos cálculos necessários para se obter uma solução para o problema. Em alguns casos o uso de memória compartilhada pode ser problemático, devido o alto custo do *hardware* necessário para realizar a simulação. Nesses casos o uso de memória distribuída pode nos permitir a realização dos cálculos. Um ambiente de HPC com memória distribuída pode ser proporcionado por diversas ferramentas, como as que serão citadas neste trabalho nos capítulos posteriores. Dependendo da ferramenta utilizada podemos montar uma infraestrutura utilizando apenas peças de *hardware* simples, que podem ser encontradas em qualquer loja de informática. O baixo custo em montar uma infraestrutura para utilizar uma dessas ferramentas é uma das características que as tornam atrativas.

Nossa motivação foi implementar uma solução que utilizasse um ambiente de HPC com memória distribuída para nos permitir obter uma solução para um problema da área da computação quântica que, devido a grande quantidade de dados envolvidos no cálculo de sua solução, não pudesse ser executado em computadores convencionais com uso de memória compartilhada.

Como neste trabalho focamos num problema que utilizasse muita memória RAM (do inglês *Random Access Memory*), o que impedia sua execução em computadores convencionais usando memória compartilhada, um dos fatores que motivou a escolha da ferramenta utilizada neste trabalho foi o fato dela armazenar seus dados em disco, o que nos permitiria executar simulações maiores.

1.2 Organização

Esta dissertação está dividida em 6 capítulos, onde essa introdução é primeiro deles. O Capítulo 2 apresenta conceitos sobre mecânica quântica, computação quântica, algoritmos quânticos, caminhadas aleatórias, caminhadas quânticas e sobre a ferramenta utilizada para proporcionar um ambiente de HPC para a execução dos testes. O Capítulo 3 apresenta conceitos básicos sobre caminhadas quânticas com dois caminhantes numa malha bidimensional, e sua implementação utilizando Map/Reduce. O Capítulo 4 descreve as operações que foram implementadas neste trabalho, sendo executadas utilizando o Hadoop, além de apresentar um simulador genérico. O

Capítulo 5 descreve o ambiente de execução dos experimentos e apresenta os resultados obtidos. O Capítulo 6 apresenta as nossas conclusões, além de evidenciar as principais contribuições e possíveis trabalhos futuros que possam ser efetuados a partir dos nossos resultados.

Capítulo 2

Caminhadas Quânticas e o Arcabouço Map/Reduce

A necessidade do processamento de grandes quantidades de dados vem aumentando bastante, tanto no meio acadêmico quanto na indústria. Isso gera uma demanda por equipamentos com maior capacidade de armazenamento e processamento, além de ferramentas capazes de lidar com esses dados aproveitando todos os recursos disponíveis nesses equipamentos. Assim, técnicas de processamento paralelo vem sendo bastante usadas para obter melhores resultados [2].

Este capítulo tem como objetivo fornecer conceitos básicos sobre caminhadas quânticas e apresentar a ferramenta utilizada no trabalho para realizar as simulações da nossa proposta. A Seção 2.1 apresenta conceitos sobre mecânica quântica e seus postulados. A Seção 2.2 possui uma breve explicação sobre computação quântica, algoritmos quânticos e caminhadas aleatórias. A Seção 2.3 apresenta conceitos sobre caminhadas quânticas que foram utilizados para desenvolver os códigos fonte utilizados neste trabalho. A Seção 2.4 apresenta as características da ferramenta escolhida para executar as simulações das caminhadas quânticas.

2.1 Mecânica Quântica

A mecânica clássica é a parte da física que estuda o movimento dos corpos sobre a ação de uma força. Essa parte da física tenta explicar fenômenos naturais utilizando um conjunto de leis que regem esses fenômenos. As leis físicas propostas pela mecânica clássica, no entanto, não se aplicam a corpos muito pequenos ou a corpos se movimentando numa velocidade próxima a da luz. A parte da física responsável por estudar o comportamento desses corpos é a mecânica quântica. Para criar uma conexão entre o mundo real e os formalismos matemáticos da mecânica quântica foram estabelecidos quatro postulados básicos:

- Postulado I: Representação dos Estados;
- Postulado II: Evolução dos Estados;
- Postulado III: Composição dos Estados;
- Postulado IV: Medição.

Segundo o primeiro postulado, todo sistema físico isolado pode ser representado por um vetor unitário ψ num espaço de Hilbert, o vetor de estado. Esse espaço é uma generalização do espaço euclidiano que pode possuir infinitas dimensões, e é dotado de produto interno. Os vetores abaixo são exemplos de vetores unitários num espaço bidimensional:

$$|0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \text{ e } |1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix}.$$

De acordo com o segundo postulado, a evolução de um sistema físico isolado pode ser descrita por um operador unitário. Ou seja, temos dois vetores de estado ψ relacionados entre si por um operador unitário U em dois instantes diferentes, t_1 e t_2:

$$\psi(t_2) = U\psi(t_1).$$

Um operador unitário bastante utilizado é o operador de Hadamard:

$$H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}.$$

Ao aplicarmos o operador de Hadamard sobre o vetor $|0\rangle$ obtemos:

$$H|0\rangle = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ 1 \end{pmatrix}.$$

Note que o resultado também é um vetor unitário. Esses operadores unitários são usados para evoluir a informação, ou seja, descrever os algoritmos.

Com o terceiro postulado temos que o espaço de estados de um sistema composto é gerado pelo produto tensorial entre os espaços de estados dos sistemas componentes.

Se $|\psi_1\rangle$ e $|\psi_2\rangle$ são espaços de estados de um sistema quântico, o estado do sistema composto $|\psi\rangle$ é dado por:

$$|\psi\rangle = |\psi_1\rangle \otimes |\psi_2\rangle.$$

Usamos o símbolo $\otimes$ para representar o produto tensorial. O produto tensorial entre dois vetores pode ser calculado explicitamente usando o produto de Kronecker. Sendo $A = (a_1, a_2, ..., a_m)$ e $B = (b_1, b_2, ..., b_n)$, o produto tensorial de A com B é dado por:

$$A \otimes B = (a_1b_1, a_1b_2, \cdots, a_1b_n, \cdots, a_mb_1, a_mb_2, \cdots, a_mb_n).$$

Note que ao realizarmos o produto tensorial de A com B, obtemos como resultado um vetor de dimensão mn.

Se pegarmos como exemplos os vetores $|0\rangle$ e $|1\rangle$, o produto tensorial entre eles é:

$$|0\rangle \otimes |1\rangle = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix}.$$

Sendo U_A e U_B duas matrizes unitárias:

$$U_A = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \text{ e } U_B = \begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & \frac{-1}{\sqrt{2}} \end{pmatrix}.$$

O produto tensorial de U_A com U_B é dado por:

$$U_A \otimes U_B = \begin{pmatrix} 1U_B & 0U_B \\ 0U_B & -1U_B \end{pmatrix} = \begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 & 0 \\ \frac{1}{\sqrt{2}} & \frac{-1}{\sqrt{2}} & 0 & 0 \\ 0 & 0 & \frac{-1}{\sqrt{2}} & \frac{-1}{\sqrt{2}} \\ 0 & 0 & \frac{-1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{pmatrix}.$$

Note que se realizarmos o produto tensorial entre dois vetores unitários ou entre duas matrizes unitárias, o resultado também será um vetor unitário ou uma matriz unitária.

O postulado da medição proporciona uma descrição clássica do sistema sendo medido, junto com algum estado quântico [3].

Segundo Portugal [4], uma medição projetiva é descrita por um operador Hermitiano O, chamado de observável no espaço de estado do sistema sendo medido. Esse observável tem a seguinte representação diagonal:

$$O = \sum_{\lambda} \lambda P_{\lambda},$$

onde P_{λ} é o projetor no auto-espaço de O associado ao autovalor λ.

Se o estado do sistema antes da medição é $|\psi\rangle$, a probabilidade de obter o resultado λ é:

$$p_\lambda = \langle\psi| P_\lambda |\psi\rangle.$$

Se o resultado da medição for λ, então o estado imediatamente após a medição é:

$$|\psi'\rangle = \tfrac{1}{\sqrt{p_\lambda}} P_\lambda |\psi\rangle.$$

Como exemplo podemos fazer uma medição na base computacional, que tem como observável, no espaço bidimensional e com um qubit, a matriz de Pauli Z:

$$Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}.$$

Note que $|0\rangle$ e $|1\rangle$ são autovetores de Z com autovalores igual a 1 e -1, respectivamente.

Medindo o estado $|\psi\rangle = \alpha |0\rangle + \beta |1\rangle$ nesse observável temos:

- $p(1) = \langle\psi| P_1 |\psi\rangle = \langle\psi \mid 0\rangle \langle 0 \mid \psi\rangle = |\alpha|^2$
- $p(-1) = \langle\psi| P_{-1} |\psi\rangle = \langle\psi \mid 1\rangle \langle 1 \mid \psi\rangle = |\beta|^2$

Após as medidas temos os estados:

- Com $\lambda = 1$:
  $|\psi'\rangle = \tfrac{1}{\sqrt{p_1}} P_1 |\psi\rangle = |0\rangle$.
- Com $\lambda = -1$:
  $|\psi'\rangle = \tfrac{1}{\sqrt{p_{-1}}} P_{-1} |\psi\rangle = |1\rangle$.

2.2 Computação Quântica, Algoritmos Quânticos e Caminhada Aleatória

2.2.1 Computação Quântica

A computação quântica estuda o uso de computadores quânticos como ferramenta para executar operações computacionais.

Um computador clássico é basicamente uma máquina que lê um conjunto de dados codificados em zeros e uns, executa cálculos e gera uma saída também codificada em zeros e uns. Esses zeros e uns são estados que podem ser representados fisicamente, e podem ser chamados de estados clássicos. Já um computador quântico, além de fazer uso direto dos fenômenos da mecânica quântica, utiliza estados quânticos. O bit (unidade mínima de informação) é, então, substituído pelo bit quântico, o *q-bit*, e os valores 0 e 1 de um bit são substituídos pelos vetores $|0\rangle$ e

$|1\rangle$. A diferença de um bit para um *q-bit* é que um *q-bit* genérico $|\psi\rangle$ pode também ser uma combinação linear dos vetores $|0\rangle$ e $|1\rangle$, com amplitudes α e β, como pode ser visto a seguir:

$$|\psi\rangle = \alpha |0\rangle + \beta |1\rangle,$$

sendo:

$$|0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \text{ e } |1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix},$$

e sendo α e β números complexos tais que:

$$|\alpha|^2 + |\beta|^2 = 1.$$

Como apresentado por Portugal et al. [5], uma interpretação física e clássica do *q-bit* é que ele está simultaneamente nos estados $|0\rangle$ e $|1\rangle$. Isso faz com que a quantidade de informação que pode ser armazenada no estado $|\psi\rangle$ seja infinita. Assim, de uma forma mais geral, um estado quântico $|\phi\rangle$ pode ser representado pela combinação linear de N estados clássicos diferentes e mutuamente exclusivos e números complexos α_i:

$$|\phi\rangle = \alpha_1 |1\rangle + \alpha_2 |2\rangle + ... + \alpha_N |N\rangle$$

Isso ocorre devido o princípio da superposição. A superposição é um princípio da mecânica quântica que afirma que uma partícula existe parcialmente em todos os estados possíveis simultaneamente. Intuitivamente, um sistema em um estado quântico $|\phi\rangle$ está em todos os estados clássicos possíveis ao mesmo tempo.

Um fenômeno da mecânica quântica que também difere drasticamente do que é previsto pela mecânica clássica é o emaranhamento (do inglês *entanglement*) de partículas. Esse fenômeno ocorre quando duas ou mais partículas estão ligadas de uma maneira que não se possa descrever uma dessas partículas de forma independente. É possível descrever apenas o emaranhado gerado por essas partículas como um todo.

A superposição e o emaranhamento podem ser utilizados juntos num computador quântico para aumentar drasticamente o desempenho dos algoritmos. Quando se sobrepõe estados de várias partículas de uma só vez, se cria a possibilidade de realizar cálculos maciçamente paralelos, ou seja, realizar todos esses cálculos de uma só vez.

2.2.2 Algoritmos Quânticos

Algoritmos quânticos são sequências de instruções que resolvem um determinado problema, igual algoritmos clássicos. A principal diferença é que em algoritmos quânticos cada uma das instruções presentes neles só pode ser executada em computadores quânticos. A computação quântica tem avançado bastante nos últimos anos apesar do fato de ainda não existir um computador quântico, ou pelo menos inteiramente quântico. Uma empresa canadense, a D-Wave, lançou em 2011 um computador quântico para venda no mercado, o D-Wave One, porém ele não é universal e não há consenso sobre se ele é escalável. Além disso, ele necessita funcionar em conjunto com computadores convencionais e possui outras limitações que restringe o seu uso.

Um dos motivos de se desenvolver algoritmos quânticos é a possibilidade de criar novos algoritmos mais eficientes que algoritmos clássicos existentes, como é o caso do algoritmo de Shor [6], que é capaz de fatorar números muito grandes com um ganho exponencial, quando comparado com o melhor algoritmo clássico conhecido.

2.2.3 Caminhada Aleatória

Uma caminhada aleatória é uma formalização matemática de um caminho formado por um conjunto de passos com direções aleatórias. Com o uso dessa técnica podemos desenvolver algoritmos mais simples e rápidos que com algoritmos já existentes.

Um exemplo simples de caminhada aleatória, detalhada por Portugal [4], é o movimento de uma partícula sobre uma reta, cuja a direção é determinada por uma moeda não viciada. Joga-se a moeda e se der coroa, a partícula dá um salto de uma unidade para a direita, se der cara, a partícula dá um salto de uma unidade para a esquerda. Esse processo é repetido a cada unidade de tempo. Como esse processo é probabilístico, não podemos saber com certeza onde estará a partícula em um instante posterior, porém podemos calcular a probabilidade p dela estar em um determinado ponto m num instante de tempo t.

2.3 Caminhadas Quânticas

Uma caminhada quântica é uma caminhada aleatória adaptada para executar num computador quântico. Uma vez que com o uso de caminhadas aleatórias diversos problemas computacionais podem ser resolvidos de forma mais eficiente [7], espera-se, com o uso da técnica de caminhada quântica, obter algoritmos eficientes para a computação quântica. As caminhadas quânticas possuem dois modelos: discreto e contínuo. Neste trabalho iremos focar no modelo discreto.

Numa caminhada quântica unidimensional uma partícula livre se movimenta a cada instante sobre uma linha infinita. Para determinarmos a direção do movimento temos que considerar um grau de liberdade adicional, que seria comparado, no caso clássico, ao resultado do lançamento de uma moeda. Numa caminhada quântica bidimensional a partícula se movimenta sobre uma malha bidimensional infinita e temos que considerar dois graus de liberdade adicionais, para decidirmos entre quatro movimentos possíveis: direita, esquerda, cima e baixo. Em ambos os tipos de caminhadas quânticas é usualmente utilizado o operador de moeda de Hadamard:

$$H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}. \tag{2.1}$$

No caso unidimensional ele é utilizado diretamente. Já no caso bidimensional o operador moeda é dado pelo produto tensorial entre dois operadores de Hadamard. Nesta seção iremos explicar o caso bidimensional, uma vez que esse foi o escolhido para realizar a primeira parte dos nossos testes.

O estado de um sistema quântico pode ser descrito por um vetor no espaço de Hilbert e a evolução do sistema é conduzida por uma operação unitária, caso o sistema esteja totalmente isolado do meio externo. Se o sistema possuir mais de uma componente, o espaço de Hilbert será o produto de *Kronecker* dos espaços de Hilbert das componentes. Sendo assim, numa caminhada bidimensional o espaço de Hilbert é dado por $\mathcal{H}_4 \otimes \mathcal{H}_\infty$, onde $\mathcal{H}_4$ é o espaço moeda e $\mathcal{H}_\infty$ é o espaço da posição da partícula. A base computacional da moeda é dada por $\{|i,j\rangle : i,j \in \{0,1\}\}$ e a base computacional do espaço posição é dada por $\{|m,n\rangle : m,n \in \mathbb{Z}\}$. Assim, o estado de um caminhante quântico num determinado tempo t é dado por:

$$|\Psi(t)\rangle = \sum_{i,j=0}^{1} \sum_{m,n=-\infty}^{\infty} \psi_{i,j;m,n}(t)\,|i,j\rangle\,|m,n\rangle. \tag{2.2}$$

A evolução do sistema do sistema é dada por:

$$U = S(C \otimes I), \tag{2.3}$$

onde U é um operador unitário, S é o operador deslocamento, C é o operador moeda e I é o operador identidade no espaço da posição.

Ao aplicarmos o operador de evolução ao estado do caminhante, obtemos:

$$|\Psi(t+1)\rangle = \sum_{i,j=0}^{1} \sum_{m,n=-\infty}^{\infty} \psi_{i,j;m,n}(t) S(C\,|i,j\rangle\,|m,n\rangle), \tag{2.4}$$

com $\psi_{i,j;m,n}(t) \in \mathbb{C}$ e $\sum_{i,j} \sum_{m,n} |\psi_{i,j;m,n}(t)|^2 = 1$.

Escrevendo de uma forma mais simplificada, temos:

$$|\Psi(t+1)\rangle = U\,|\Psi(t)\rangle. \tag{2.5}$$

Se definirmos o estado inicial da caminhada $|\Psi(0)\rangle$, podemos executar a equação (2.5) t vezes e desta forma obter o valor de $|\Psi(t)\rangle$.

Agora podemos calcular a distribuição de probabilidade, e com ela determinar a probabilidade p da partícula estar numa determinada posição. A distribuição de probabilidade para t é dada por:

$$p_{m,n}(t) = \sum_{i,j=0}^{1} |\psi_{i,j;m,n}(t)|^2. \tag{2.6}$$

A distribuição de probabilidade nos fornece a probabilidade da partícula estar em uma determinada posição na malha, assim nos permitindo prever onde a partícula se encontra após passar um determinado tempo t. Mais detalhes sobre caminhadas quânticas podem ser encontrados em [4] e [8].

2.4 O Arcabouço Map/Reduce

O Apache Hadoop [9] é um arcabouço (*framework*) que permite o processamento distribuído de grandes quantidades de dados [10] através de um conjunto de computadores, chamado de *cluster*. Ele foi desenvolvido para ser escalável desde simples servidores até milhares de máquinas. Ao invés de confiar no *hardware* para proporcionar alta disponibilidade, a biblioteca em si detecta e trata falhas na camada de aplicação, de modo a fornecer um serviço altamente disponível no topo do *cluster*.

O Hadoop utiliza um modelo simples de programação, o Map/Reduce [11]. Nesse modelo de programação o usuário especifica uma função *map* que irá processar um par de chave/valor que irá gerar um novo conjunto intermediário de pares de chave/valor, e uma função *reduce* que irá combinar todos os valores intermediários de uma mesma chave intermediária.

Devido o uso desse modelo de programação, o Hadoop poupa o usuário de ter que analisar o código em busca de problemas relacionados com paralelismo, comunicação entre processos e possíveis falhas durante a execução, como ocorre com o uso do OpenMP [12] ou do MPI (do inglês *Message Passing Interface*) [13]. Assim o Hadoop permite que o usuário foque no desenvolvimento da aplicação, escrevendo o código que realizará o processamento dos dados. O OpenMP é uma interface de programação paralela para computadores utilizando memória compartilhada. Enquanto o MPI é uma biblioteca de comunicação para computação paralela que possibilita o envio e o recebimento de mensagens entre os processos.

A Figura 2.1 é uma representação da execução de um algoritmo de *wordcount* (contador de palavra) utilizando o Apache Hadoop. No início você passa um arquivo de texto como entrada para o Hadoop. Ao começar a execução, na etapa de

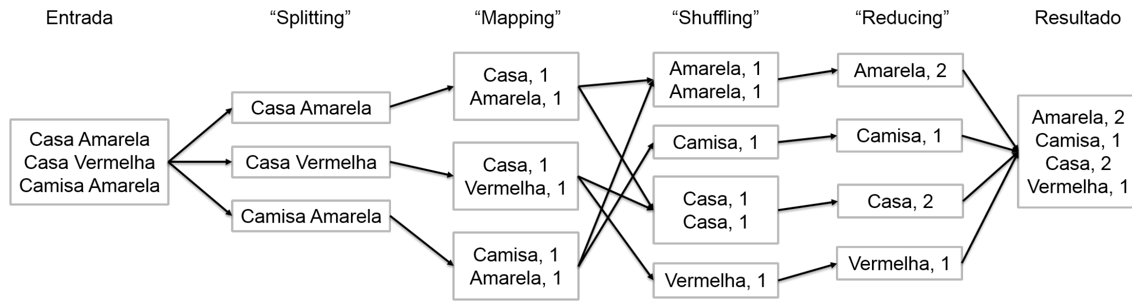

Figura 2.1: Diagrama de execução de um algoritmo de *wordcount* utilizando Hadoop

splitting (divisão) o Hadoop irá fragmentar sua entrada e enviar cada um desses fragmentos para uma tarefa *map*. Na etapa de *mapping* (mapeamento) o código da função *map* que foi escrito pelo usuário será executado, com cada uma das tarefas *map* criadas pelo Hadoop executando o mesmo código, cada tarefa com um dos fragmentos gerados na etapa de *splitting*. Assim que parte das tarefas *map* terminam sua execução o Hadoop começa a execução da etapa de *shuffling* (embaralhamento), onde ele ordena e agrupa as saídas das tarefas *map* pela chave e as distribui entre as tarefas *reduce*. Após a etapa de *shuffling* terminar o Hadoop inicia a etapa de *reducing* (redução), onde a função *reduce*, que foi escrita pelo usuário, começa ser executada. No fim é criado um arquivo com a saída das tarefas *reduce*.

O Hadoop pode ser utilizado via biblioteca ou através do *Hadoop Streaming* [14], uma ferramenta inclusa no pacote[1] distribuído através do site da Apache. Nesse pacote também se encontra a biblioteca nativa para a linguagem de programação Java. Existem bibliotecas para outras linguagens de programação, que proporcionam flexibilidade na hora de escrever o código de seu programa, similar ao que acontece com o uso da biblioteca nativa, porém o desempenho dessas bibliotecas não é muito bom quando comparado com a biblioteca nativa. O *Hadoop Streaming* permite que um executável escrito em qualquer linguagem de programação execute funções *map* e *reduce* através do Hadoop, sem a necessidade do uso de bibliotecas, utilizando apenas comandos de entrada e saída. Apesar do *Hadoop Streaming* não possuir um desempenho superior ao da biblioteca nativa, devido a uma pequena sobrecarga no processamento [15], ele geralmente apresenta um desempenho melhor que o encontrado em outras bibliotecas. A Figura 2.2 é um diagrama que representa como o Hadoop executaria suas funções *map* e *reduce* utilizando o *Hadoop Streaming*.

O Hadoop pode ser dividido basicamente em duas partes, como será visto adiante: o arcabouço Map/Reduce e o sistema de arquivos HDFS (do inglês *Hadoop Distributed File System*). Uma outra característica bem interessante do Apache Hadoop é a possibilidade de realizar a sua instalação em apenas um nó[2]. Isso nos

[1] `https://archive.apache.org/dist/hadoop/core/hadoop-1.2.1/`

[2] `http://hadoop.apache.org/docs/r1.2.1/single_node_setup.html`

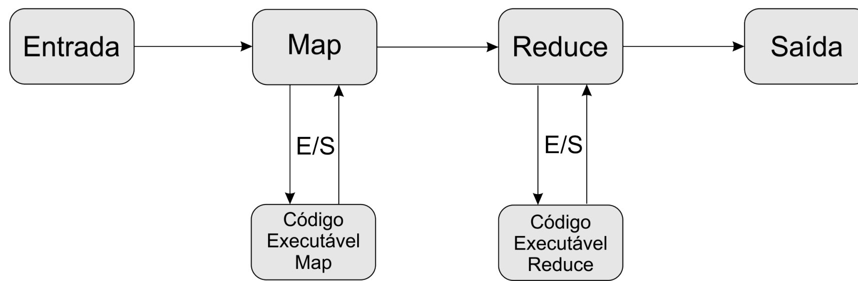

Figura 2.2: Diagrama de execução do Hadoop usando o *Hadoop Streaming*

permite executar códigos no Hadoop sem a necessidade de um *cluster*, usando apenas um computador comum, como por exemplo um notebook. Assim podemos escrever códigos e testá-los, para casos simples, mesmo quando estamos sem acesso ao *cluster*. No meio acadêmico essa característica se torna de vital importância, uma vez que a quantidade de usuários que precisam usar o *cluster* cria a necessidade de utilizar uma fila para o seu uso, o que geralmente restringe bastante o seu acesso ao usuário. Essa característica assim permite que o usuário possua mais liberdade para criar e testar os seus códigos, além de ser muito útil para propósitos de ensino e aprendizagem. Vale lembrar que a instalação do Apache Hadoop em apenas um nó limita o poder computacional, e por isso serve apenas para executarmos problemas pequenos, com o propósito de teste do código, de ensino ou aprendizagem.

2.4.1 Hadoop MapReduce

O *Hadoop MapReduce* é um arcabouço que permite escrever programas para processamento de uma grande quantidade de dados de forma paralela em grandes *clusters* com milhares de nós, de forma confiável e com tolerância a falhas.

Ao executar um código no *Hadoop MapReduce*, um *job* (trabalho) é criado. Usualmente os dados, tanto de entrada quanto de saída, são armazenados no sistema de arquivos do Hadoop, o HDFS. No início os dados da entrada são divididos em partes independentes e são processados pelas tarefas *map* paralelamente. O *framework* então ordena as saídas dos *maps*, que são passados como entrada para as tarefas *reduce*. O *Hadoop MapReduce* cuida do agendamento das tarefas, monitorando-as e re-executando aquelas que falharem.

Além das operações de mapeamento (*mapper*) e redução (*reducer*), existe também uma operação de combinação (*combiner*). Com o uso dessa operação podemos melhorar o desempenho da execução do *job*. Ela é equivalente a execução local de uma operação *reducer*, que irá minimizar a quantidade de pares de chave/valor que irão trafegar pela rede. Como padrão, você pode usar sua função *reducer* como um *combiner*. Uma função *combiner* deve obedecer as propriedades comutativa e associativa, para que o resultado final não seja alterado.

Em um *cluster* com Hadoop, os nós tipicamente funcionam como nós de processamento e de armazenamento. O arcabouço *Hadoop MapReduce* consiste de um nó principal (*master*), chamado de *JobTracker* e de nós trabalhadores (*slaves*), chamados de *TaskTracker*. O nó principal é responsável por fazer o agendamento dos *jobs* nos nós trabalhadores, monitorando e re-executando as tarefas que falharem. Os nós trabalhadores irão executar as tarefas que lhes forem passadas pelo nó principal.

O *Hadoop MapReduce* trata a entrada como um conjunto de pares de chave e valor, $< chave, valor >$, e gera em sua saída também um conjunto de pares de chave e valor. O tipo de entrada e saída de um *job* é representado na Figura 2.3.

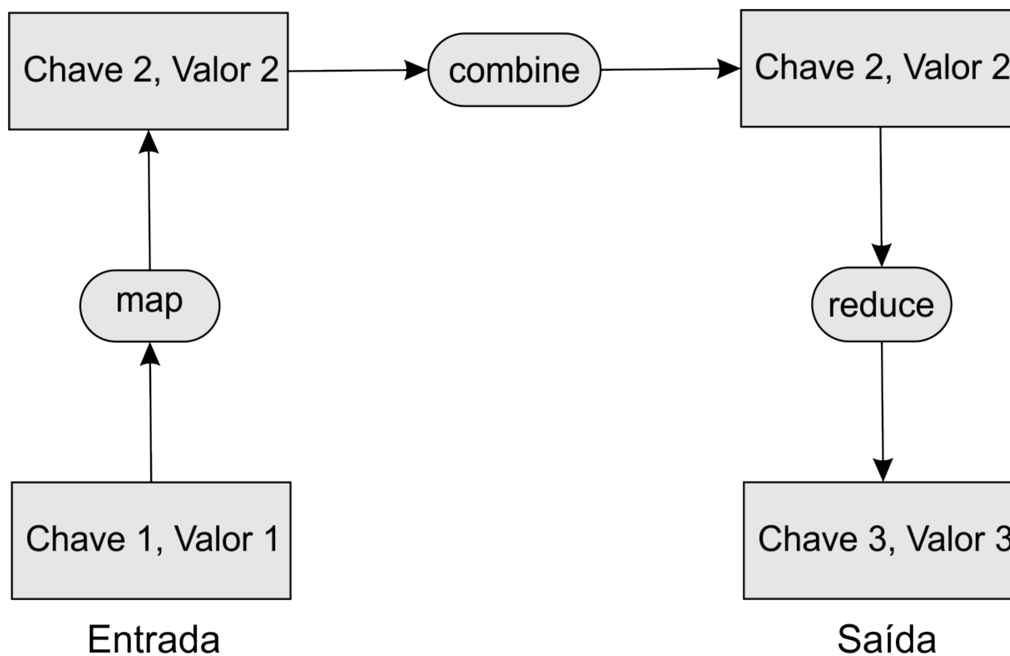

Figura 2.3: Diagrama do tipo de entrada e saída do Hadoop

2.4.2 HDFS

O HDFS (*Hadoop Distributed File System*) [16] é um sistema de arquivos distribuído utilizado pelas aplicações que utilizam o Hadoop. O HDFS é uma implementação de código aberto do GFS (*Google File System*) [17]. Assim como o *Hadoop MapReduce*, o HDFS possui um nó principal (*master*), chamado *Namenode* e nós escravos (*slaves*), chamados *Datanodes*. O *Namenode* é responsável por gerenciar os metadados do sistema de arquivos, enquanto os *Datanodes* são responsáveis por armazenar os dados.

A Figura 2.4 apresenta um diagrama resumido da arquitetura do Hadoop. Nela podemos ver que todos os nós, principal ou escravos, são usados tanto pelo *Hadoop MapReduce* quanto pelo HDFS.

Algumas das características do HDFS que vale a pena ressaltar são:

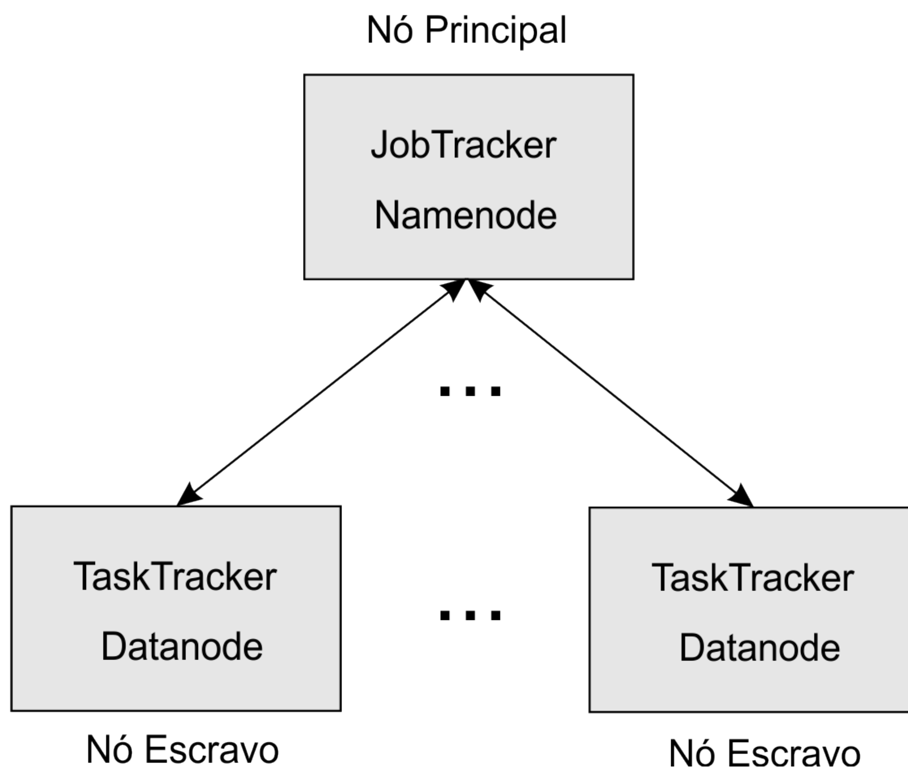

Figura 2.4: Diagrama da arquitetura básica do Hadoop

- Como geralmente todos nós, exceto o *master*, são *Datanodes*, ao aumentar a quantidade de nós, automaticamente aumenta-se a quantidade de armazenamento;

- O *Namenode* e os *Datanodes* possuem servidores web embutidos que permitem monitorar o atual estado do *cluster*;

- O Hadoop suporta comandos em modo texto para interagir diretamente com o HDFS;

- O HDFS possui diversos parâmetros de configuração que podem ser usados para se obter melhor desempenho e confiabilidade de acordo com o tamanho do *cluster* ou do tipo de problema executado nele;

- Os dados são replicados e ficam espalhados pelos *Datanodes* de forma que otimize o desempenho durante a execução dos *jobs*. Por padrão são criadas 3 réplicas de cada dado.

A Figura 2.5 é um diagrama que mostra a arquitetura do HDFS. Nela podemos ver os dados espalhados entre os *Datanodes*. O *Namenode* fica responsável por gerenciar operações como abertura, fechamento e renomeação de arquivos, além de determinar o mapeamento dos blocos de dados dos *Datanodes*, enquanto estes ficam responsáveis por operações de leitura e escrita.

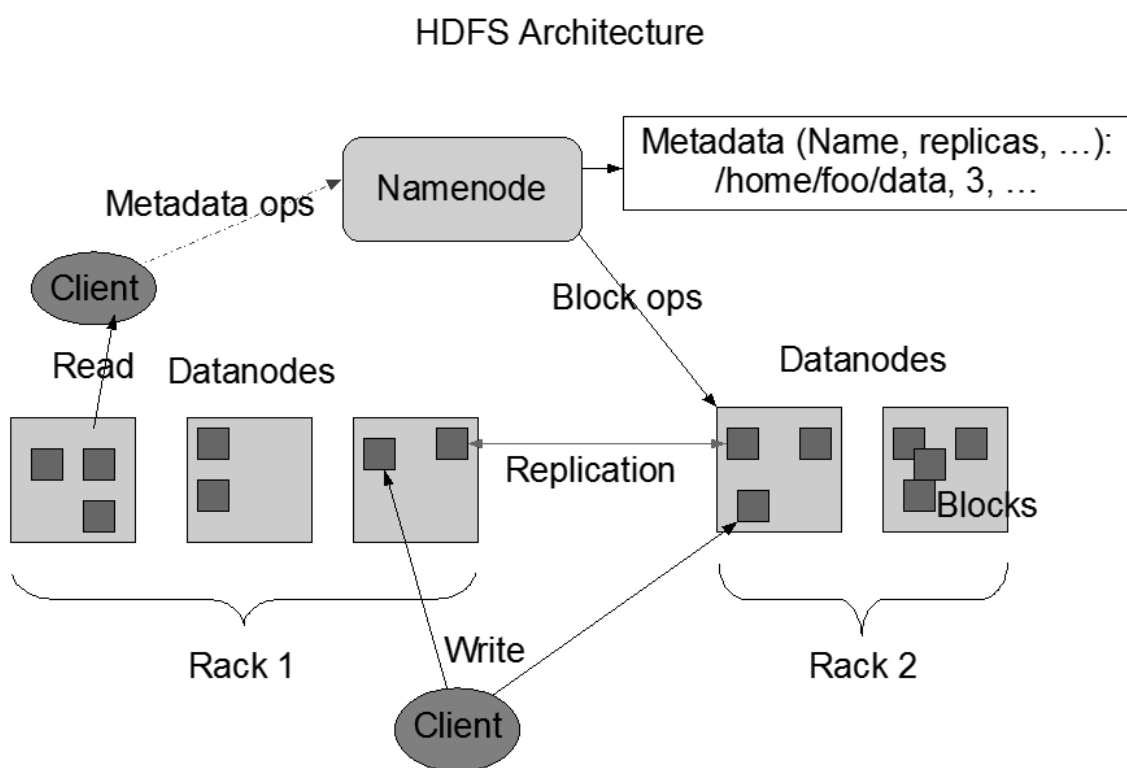

Figura 2.5: Diagrama da arquitetura do HDFS [9]

Capítulo 3

Uso do Map/Reduce para Simulação de Caminhadas Quânticas com Dois Caminhantes numa Malha Bidimensional

Como os pesquisadores da área de computação quântica ainda não possuem computadores quânticos para executarem seus algoritmos e realizar seus testes, um dos recursos que eles utilizam é a simulação. A simulação é um processo que permite imitar um sistema ou operação do mundo real. Através de simulações, por exemplo, podemos gerar uma animação computacional que simula o movimento das ondas do oceano. Um problema de utilizar simulação como meio de executar um algoritmo quântico é o fato de não podermos usufruir de recursos que estarão presentes em computadores quânticos, como a superposição e o emaranhamento, que permitem que a execução do algoritmo quântico seja rápida. Estas entre outras características fazem com que a simulação de algoritmos quânticos em computadores clássicos seja um processo demorado [18].

O tempo de execução e os recursos utilizados durante uma simulação de uma caminhada quântica são fatores limitantes que podem impedir que o resultado final seja alcançado. A dimensão do espaço posição e a quantidade de partículas (caminhantes) são exemplos de características de uma caminhada quântica que influenciam bastante na quantidade de recursos computacionais usados durante a simulação. Na nossa proposta nós escolhemos utilizar apenas caminhadas quânticas que utilizam muita memória *RAM* quando simuladas num computador utilizando um código sequencial, para tirar proveito da capacidade do Hadoop de lidar com grandes quantidades de dados. Por possuir essas características, a escolhida para realizar nossos primeiros testes foi a caminhada quântica com duas partículas numa

malha bidimensional.

Este capítulo tem como objetivo fornecer conceitos básicos sobre caminhadas quânticas com dois caminhantes e sua implementação utilizando Map/Reduce. A Seção 3.1 possui alguns trabalhos relacionados com caminhadas quânticas. A Seção 3.2 possui alguns trabalhos que utilizam o Hadoop. A Seção 3.3 descreve como foi realizada a simulação da caminhada quântica com dois caminhantes numa malha bidimensional utilizando o Hadoop.

3.1 Trabalhos Relacionados: Caminhadas Quânticas

Existem vários simuladores de caminhada quântica, como o Qwalk [19] e o HiPerWalk [20]. Além de outros simuladores mais genéricos como o QuIDDPro [21] e o QuTIP [22], que podem ser usados para simular uma caminhada quântica. Simuladores como o QuTiP e o HiPerWalk realizam os cálculos de forma paralela aproveitando os recursos de multiprocessamento dos computadores em que estão sendo executados, porém eles não foram desenvolvidos para ambientes com memória distribuída, o que os deixam limitados com o poder computacional de apenas uma máquina.

3.1.1 QWalk

O QWalk é um simulador de caminhada quântica desenvolvido utilizando a linguagem de programação C. Ele pode ser usado tanto em ambientes com Linux quanto em ambientes com Microsoft Windows. No arquivo compactado, que pode ser baixado na página do desenvolvedor, estão presentes o código fonte e executáveis compilados para uso no Microsoft Windows. O QWalk é composto por 3 ferramentas, que possuem as seguintes finalidades:

- Simulação de caminhadas quânticas em malhas unidimensionais;
- Simulação de caminhadas quânticas em malhas bidimensionais;
- Amplificação de certas regiões de uma função de onda, para facilitar a visualização.

3.1.2 HiPerWalk

O HiPerWalk é um programa de código livre que permite ao usuário realizar simulações de caminhada quântica em grafos usando HPC (*High Performance Computing*), Computação de Alto Desempenho. O usuário pode usar os recursos de

paralelização presentes no computador, como placas aceleradoras, múltiplos núcleos do processador e GPGPU (*General Purpose Graphics Processing Unit*), Unidade de Processamento Gráfico de Propósito Geral, para acelerar a execução sem a necessidade de conhecimentos prévios de programação paralela.

Esse simulador ainda encontra-se em desenvolvimento e utiliza as linguagens Python, OpenCL, Neblina e Gnuplot. A saída do simulador é a estatística principal da distribuição de probabilidade associada com a caminhada quântica. Essa saída é colocada em arquivos de dados e as plotagens são automaticamente geradas pelo simulador. A versão atual calcula caminhada quântica em tempo discreto com e sem o uso de moeda. Caminhada quântica com tempo contínuo e caminhada de Szegedy estarão disponíveis em breve, segundo as informações presentes no manual de usuário [20].

3.1.3 QuIDDPro

O QuIDDPro é uma interface computacional rápida, escalável e de fácil uso para simulação de circuitos quânticos genéricos. Ele suporta vetores de estado, matrizes de densidade e operações relacionadas usando o QuIDD (*Quantum Information Decision Diagram*). Diferentes do que acontece com pacotes como Matlab e Octave, o QuIDDPro não sofre sempre da explosão exponencial no tamanho das matrizes necessárias para simular circuitos quânticos. Como resultado, os autores encontraram que o QuIDDPro é mais rápido e usa menos memória quando comparado com outros métodos de simulação genéricos para alguns circuitos úteis com muito mais de 10 q-bits.

3.1.4 QuTiP

O QuTiP (*Quantum Toolbox in Python*) é um programa de código aberto para simular a dinâmica de sistemas quânticos abertos, codificado utilizando a linguagem de programação Python. Com ele é possível representar sistemas quânticos genéricos e realizar cálculos e simulações nesses sistemas. Segundo os autores, o QuTiP visa proporcionar simulações numéricas para uma grande variedade de problemas computacionais em sistemas quânticos com uma interface amigável e eficiente, incluindo aqueles com dependência temporal arbitrária comumente encontrados em uma ampla gama de aplicações de física, como ótica quântica, íons aprisionados, circuitos supercondutores e ressonadores nanomecânicos quânticos.

Esse simulador pode utilizar os múltiplos núcleos de processamento encontrados em todos os computadores modernos, para executar tarefas simultaneamente, reduzindo o tempo necessário para finalizar a execução.

3.2 Trabalhos Relacionados: Hadoop

Devido as características presentes no Apache Hadoop, ele vem sendo utilizado por pesquisadores como meio de obter a solução para problemas que envolvem cálculos numéricos cuja solução é obtida realizando o processamento de uma grande quantidade de dados. Abordagens para resolução de problemas como o cálculo de inversão geofísica [23] e o cálculo de riscos agregados [24] já foram apresentadas usando o Hadoop como ferramenta para acelerar esse processo.

3.2.1 Inversão Geofísica

A geofísica é uma área de pesquisa intensa, cujas considerações são acompanhadas por modelos e análise de dados, que são tarefas que exigem alto poder computacional. O conhecimento da distribuição de velocidade em meio geológico é uma das coisas mais importantes na exploração mineira. O processo de reconstrução da distribuição de velocidade é chamado de problema inverso. Resolver um problema inverso é difícil devido a relação não linear entre a distribuição de valor dos parâmetros elásticos e os tempos para a propagação das ondas.

Segundo Krauzowicz et al. [23], a ideia principal em utilizar o Apache Hadoop foi procurar regiões de perspectiva pela análise dos dados gerados com seus valores de avaliação. E eles ainda tiram vantagem dos resultados obtidos com o Apache Hadoop, enviando eles de volta para o algoritmo de inversão e testando, para obter uma solução mais precisa.

3.2.2 Riscos Agregados

No domínio de análise de risco em larga escala, uma grande quantidade de dados precisa ser rapidamente processada e milhões de simulações necessitam serem executadas com alta velocidade. Para isso os dados devem ser gerenciados eficientemente e o paralelismo explorado pelos algoritmos que realizam as simulações. Segundo Yao et al. [24], seu trabalho foi motivado em explorar técnicas para o emprego de computação de alto desempenho, não só para acelerar a simulação mas para processar e gerenciar os dados de forma mais eficiente para a análise de risco agregado. Eles desenvolveram um algoritmo paralelo implementado usando Map/Reduce para realizar a análise de riscos agregados com alto desempenho.

3.3 Simulação de uma Caminhada Quântica com Dois Caminhantes numa Malha Bidimensional

Como pode ser observado na Seção 2.3, uma caminhada quântica pode ser vista como uma sequência de operações algébricas. Essa sequência pode ser dividida em duas partes, onde a primeira parte é responsável por gerar os dados e a segunda parte é responsável por calcular o vetor de estados para um determinado tempo, além da distribuição de probabilidade para esse vetor. A primeira parte pode ser considerada opcional uma vez que podemos possuir os dados necessários previamente calculados, e queremos apenas calcular o vetor de estados para um determinado tempo e encontrar a distribuição de probabilidade. Lembrando que a distribuição de probabilidade é usada para estimar a localização do caminhante após um determinado espaço de tempo.

Para realizar a simulação de uma caminhada quântica com o auxílio do Hadoop e assim avaliar a nossa abordagem, primeiro tivemos que implementar algumas operações, sendo escritas de acordo com o modelo Map/Reduce. O funcionamento dessas operações é descrito na Seção 4.1.

Com essas operações prontas, o próximo passo foi criar um código que funcionasse como um *script*, e que fizesse o uso dessas operações nos permitindo executar a simulação da caminhada quântica. Ao arquivo que possui esse código demos o nome de QW[1], de *Quantum Walk*. Esse e os demais programas presentes neste trabalho foram codificados usando a linguagem de programação Java[2] e seguindo suas convenções de codificação [25]. Nele usamos a biblioteca padrão do Hadoop para fazer a manipulação dos arquivos no HDFS. Todos os dados gerados por esse programa são escritos diretamente no HDFS, para acelerar o processo de escrita dos arquivos e para que possam ser acessados pelas operações. Lembrando que os arquivos passados como entrada para um *job* do Hadoop devem estar no HDFS, senão o Hadoop é incapaz de acessá-los.

A caixa de texto a seguir contém o algoritmo utilizado no arquivo QW, para gerar os dados e executar a simulação de uma caminhada quântica com duas partículas numa malha bidimensional. Esse código foi desenvolvido baseado num trabalho de Ahlbrecht et al. [26]. Esse artigo descreve um experimento onde ocorre interação entre duas partículas ao executar uma caminhada quântica. Alguns dos passos abaixo foram realizados para o algoritmo entrar em conformidade com os dados descritos no artigo.

[1] `https://github.com/david-ufrj/qw-hadoop/tree/master/QW`
[2] `https://docs.oracle.com/javase/specs/`

QW

1. **Define** $SIZE,\ STEPS$
2. **Cria** $Hadamard_A[2,2]\ e\ Hadamard_B[2,2]$
3. $Hadamard = Hadamard_A \otimes Hadamard_B$
4. **Apaga** $Hadamard_A\ e\ Hadamard_B$
5. **Cria** $Identidade[SIZE^2]$
6. $Moeda_{W1} = Hadamard \otimes Identidade$
7. **Apaga** $Hadamard\ e\ Identidade$
8. **Cria** $Deslocamento_{W1}[4 * SIZE^2, 4 * SIZE^2]$
9. $W1A = Deslocamento_{W1} \otimes Moeda_{W1}$
10. $W1B = Deslocamento_{W1} \otimes Moeda_{W1}$
11. **Apaga** $Deslocamento_{W1}\ e\ Moeda_{W1}$
12. **Cria** $Identidade_{W2A}[4 * SIZE^2]$
13. $W2A = W1A \otimes Identidade_{W2A}$
14. **Apaga** $W1A\ e\ Identidade_{W2A}$
15. **Cria** $Identidade_{W2B}[4 * SIZE^2]$
16. $W2B = Identidade_{W2B} \otimes W1B$
17. **Apaga** $Identidade_{W2B}\ e\ W1B$
18. **Cria** $G[16 * SIZE^4, 16 * SIZE^4]$
19. **Cria** $Psi_t[16 * SIZE^4]$
20. **Para** i de 1 até STEPS **faça** :
21. $\quad Psi_t = G * Psi_t$
22. $\quad Psi_t = W2B * Psi_t$
23. $\quad Psi_t = W2A * Psi_t$
24. $Norma = Norma\ de\ Psi_t$
25. $PDF =$ *Distribuição de Probabilidade de* Psi_t
26. **Copia** $Psi_t,\ Norma\ e\ PDF\ para\ o\ disco\ local$
27. **Apaga** $W2A,\ W2B,\ G,\ Psi_t,\ Norma,\ e\ PDF$

Em relação ao algoritmo, os comandos foram utilizados com o seguinte significado:

- *Define*: atribuir valores a constantes utilizadas no código;
- *Cria*: criar arquivos diretamente no HDFS;

- *"="*: executar alguma operação utilizando como entrada os arquivos representados pelos nomes à direita do sinal. O resultado é armazenado no diretório representado pelo nome à esquerda do sinal;

- *Apaga*: remover um arquivo ou diretório.

Na linha 1 definimos o valor das constantes *SIZE* e *STEPS*, que representam, respectivamente, a dimensão da malha bidimensional e a quantidade de passos da simulação. A quantidade de passos da simulação representa o tempo.

Na linha 2 criamos duas matrizes de Hadamard. Houve a necessidade de criar duas matrizes com o mesmo conteúdo devido a forma como a operação que será realizada sobre elas foi implementada. Como o Hadoop, na hora de executar as funções *map* e *reduce*, enxerga todos os arquivos de entrada como se fosse apenas um arquivo, temos que diferenciar os arquivos de entrada adicionando um marcador em cada linha. Assim, para realizar uma operação de multiplicação de matrizes ou do produto de Kronecker, precisamos criar duas matrizes, com mesmo conteúdo, diferenciando-as apenas por um marcador no início de cada linha. Esse processo será explicado com maior detalhes na Seção 4.1.

Na linha 3 executamos o produto de Kronecker e geramos uma matriz de Hadamard com dimensão quatro. Fazemos isso pois a dimensão do espaço moeda de uma caminhada quântica numa malha bidimensional possui dimensão quatro, como pode ser observado na Seção 2.3.

Nas linhas 4, 7, 11, 14 e 17 apagamos os arquivos que não serão mais usados para liberar espaço, maximizando a quantidade de recursos livres.

Na linha 5 criamos uma matriz identidade que é usada na linha 6 para realizar o produto de Kronecker com a moeda de Hadamard.

Na linha 8 criamos a matriz *Deslocamento*, que possui os dados para a realização dos movimentos das partículas.

Na linha 9 aplicamos o produto de Kronecker entre a matriz *Deslocamento* e a matriz de moeda, gerando uma matriz que é responsável por controlar o movimento das duas partículas.

Na linha 10 criamos uma matriz idêntica a que foi gerada na linha 9, exceto pelo marcador no início de cada linha.

Na linha 12 e na linha 15 criamos mais duas matrizes identidade. Elas são usadas, respectivamente, nas linhas 13 e 16, onde realizamos o produto de Kronecker gerando finalmente as matrizes que representam as duas partículas, $W2A$ e $W2B$.

Na linha 18 criamos um operador de interação, G. Se usássemos apenas as matrizes $W2A$ e $W2B$, quando as partículas se encontrassem na mesma posição para um mesmo valor de tempo, não ocorreria nada. Esse operador faz com que as partículas mudem o sentido de seu movimento quando ocorre interação entre elas.

Na linha 19 criamos o vetor de estados Psi_t. Ao criá-lo definimos o seu estado inicial, ou seja, seu valor para o tempo igual a zero.

Na linha 20 criamos um laço de repetição que será executado $STEPS$ vezes. Na última iteração do laço obtemos o valor de Psi_t para o tempo igual ao valor definido em $STEPS$. Como a multiplicação de matrizes é uma operação realizada apenas entre duas matrizes, para realizarmos a multiplicação entre as três matrizes e o vetor, precisamos realizar três multiplicações de matrizes. Note nas linhas 21, 22 e 23, que apesar da operação a ser realizada seja $Psi_t = W2A * W2B * G * Psi_t$, executamos a multiplicação das matrizes da direita para esquerda. Como o vetor de estados Psi_t é pequeno em relação as outras matrizes, ao executarmos a multiplicação nessa sequência, reduzimos o tempo necessário para executar as operações. Lembrando que isso só é possível pois a operação de multiplicação de matrizes possui associatividade, ou seja, a ordem das multiplicações não altera o produto.

Na linha 24 calculamos a norma do vetor de estados Psi_t para verificar se ele ainda é um operador unitário. Se ele não for, significa que há algum problema com os valores dos operadores usados na multiplicação da linha 21. O resultado dessa operação nos mostra quão confiável é o resultado obtido para Psi_t.

Na linha 25 calculamos a distribuição de probabilidade do vetor de estados Psi_t. Com o resultado dessa operação podemos estimar a posição das partículas. Uma forma para se fazer isso seria utilizar alguma ferramenta para plotar gráficos.

Na linha 26 copiamos Psi_t, $Norma$ e PDF para um diretório local. Após isso, na linha 27, removemos o restante dos dados gerados durante a execução.

Capítulo 4

Quandoop

Alguns testes preliminares foram realizados utilizando o código do algoritmo descrito na Seção 3.3. Após eles decidimos que além das operações básicas que utilizam o Hadoop, adicionaríamos na nossa contribuição um programa de simulação mais genérico que executasse alguma sequência de operações que fosse recorrentemente utilizada na área de computação quântica. A esse simulador demos o nome de Quandoop.

Este capítulo tem como objetivo descrever as operações básicas desenvolvidas para serem executadas no Hadoop e apresentar o nosso simulador, o Quandoop. A Seção 4.1 descreve o funcionamento das operações básicas desenvolvidas para esse trabalho. A Seção 4.2 apresenta o simulador Quandoop.

4.1 Operações Básicas

Quando escrevemos um código sequencial para executar uma caminhada quântica, geralmente utilizamos bibliotecas, presentes na linguagem de programação escolhida, para realizar operações como o produto de Kronecker ou a multiplicação de matrizes. Além, é claro, das estruturas de armazenamento de dados aceitas como argumento de entrada para as funções dessas bibliotecas. Porém quando trabalhamos com matrizes grandes um problema que surge é o "estouro" da memória *RAM*. As estruturas de armazenamento de dados, aceitas pelas bibliotecas que possuem funções algébricas, geralmente armazenam os dados na memória *RAM* do computador, o que torna a quantidade dessa memória um fator limitante em problemas que utilizam uma grande quantidade de dados.

O Hadoop, por outro lado, utiliza como entrada dados armazenados em disco. Com isso esse limite é expandido, uma vez que o espaço de armazenamento em discos geralmente é bem maior que o tamanho da memória *RAM* do computador. Assim, ao desenvolvermos códigos capazes de realizar as operações algébricas presentes no nosso problema utilizando o modelo de programação aceito pelo Hadoop,

o Map/Reduce, podemos realizar essas operações sobre uma quantidade maior de dados. E por causa das características do Hadoop, obtemos ganho com o paralelismo e sem a necessidade de nos preocupar com problemas de consistência de dados ou de comunicação entre processos, uma vez que o arcabouço já cuida desses problemas.

Para executarmos uma simulação de uma caminhada quântica duas operações são bastante usadas: o produto de Kronecker e a multiplicação de matrizes. Por esse motivo essas foram as duas primeiras operações que decidimos implementar. Além delas ainda implementamos a operação de norma para vetores e outras operações que são usadas em conjunto para calcular a distribuição de probabilidade para dados armazenados num vetor.

Antes de começarmos a implementação das operações, primeiro tivemos que definir um formato para os arquivos onde serão armazenados as matrizes. As matrizes envolvidas na simulação de uma caminhada quântica são matrizes esparsas, ou seja, são matrizes cuja maioria de seus elementos possuem o valor zero. Sendo assim, por questões de eficiência, qualquer elemento cujo valor seja zero não estará presente no arquivo. Com isso reduzimos bastante o tamanho do arquivo gerado e diminuímos o tempo de processamento dessas matrizes. Outro fator que foi considerado foi o tipo dos valores a serem armazenados nesse arquivo. Numa caminhada quântica as matrizes guardam valores pertencentes ao espaço de Hilbert, que são números complexos. Assim as operações foram implementadas para tratar os valores dos elementos das matrizes como números complexos.

Como apenas parte dos valores da matriz está presente no arquivo (valores não nulos), precisamos adicionar os índices de cada elemento (linha e coluna) para podermos identificá-los corretamente. Como o Hadoop trata os arquivos de entrada de um *job* como se fosse apenas um arquivo, que seria a junção de todos os arquivos de entrada, tivemos que adicionar um marcador em cada linha do arquivo para podermos diferenciar a matriz à esquerda da operação da matriz à direita da operação. Isso só foi necessário devido a existência de operações que utilizam duas matrizes como entrada: o produto de Kronecker e a multiplicação de matrizes. Foi necessário também adicionar uma linha no início do arquivo, um *header*, que contém as informações do tipo da matriz, se a matriz está à esquerda ou à direita do operador, e suas dimensões. Lembrando que o Hadoop só é capaz de realizar a execução de um *job* recebendo como entrada arquivos armazenados no HDFS, então quando falarmos de arquivos de entrada para uma operação, estamos falando de arquivos armazenados no HDFS.

O formato usado para os arquivos de entrada do produto de Kronecker, multiplicação de matrizes, norma, valor absoluto ao quadrado e remodelagem foi o seguinte:

- *Header* (#TIPO,M,N):

- "#": marcador que indica que esta linha é o *header*;
- TIPO: "A" ou "B". "A" é a matriz à esquerda do operador. "B" é a matriz à direita do operador;
- M e N: são as dimensões da matriz. "M" é a quantidade de linhas e "N" é a quantidade de colunas. Um vetor deve ser representado como uma matriz coluna, ou seja, o seu valor para "N" deve ser "1".

- Outras linhas (TIPO,M,N,REALjIMAGINARIO):
  - TIPO: "A" ou "B". "A" é a matriz à esquerda do operador. "B" é a matriz à direita do operador;
  - M e N: são as dimensões da matriz. "M" é a quantidade de linhas e "N" é a quantidade de colunas. Um vetor deve ser representado como uma matriz coluna, ou seja, o seu valor para "N" deve ser "0" para todos os elementos.
  - REAL e IMAGINARIO: A parte real e a parte imaginária do valor de cada elemento da matriz. O caractere "j" é um separador usado no código das operações para distinguir a parte real da parte imaginária.

Esse formato será referenciado neste trabalho a partir desse ponto como sendo o formato padrão para os arquivos de entrada.

O formato usado para os arquivos de entrada da operação de soma sobre os eixos é parecido ao utilizado pelas outras operações, exceto os valores e a quantidade de eixos. Os valores são números reais, e enquanto os arquivos de entradas das outras operações possuíam sempre dois eixos, "M" e "N", os arquivos de entrada dessa operação podem possuir vários eixos.

O código fonte e o executável para essas operações[1] estão disponíveis para *download*. Junto com eles está um arquivo *README* com as informações necessárias para executar as operações ou recompilar o seu código fonte.

4.1.1 Produto de Kronecker

O produto de Kronecker é uma operação que é realizada entre duas matrizes e é denotado pelo símbolo $\otimes$. Nele cada elemento de uma matriz será multiplicado por cada elemento da outra matriz, com cada produto gerando um novo elemento da matriz resultado. Essa matriz resultado é uma matriz cuja dimensão é o produto das dimensões das matrizes envolvidas, ou seja, se aplicarmos o produto de Kronecker entre uma matriz $A_{m\text{x}n}$ e uma matriz $B_{p\text{x}q}$ obteremos uma matriz $C_{mp\text{x}nq}$.

[1] `https://github.com/david-ufrj/qw-hadoop/tree/master/Operations`

O produto de Kronecker entre uma matriz $A_{m\text{x}n}$ e uma matriz $B_{p\text{x}q}$ é dado por:

$$A_{m\text{x}n} \otimes B_{p\text{x}q} = \begin{bmatrix} a_{11}b_{11} & a_{11}b_{12} & \cdots & a_{1n}b_{1q} \\ a_{11}b_{21} & a_{11}b_{22} & \cdots & a_{1n}b_{2q} \\ \vdots & \vdots & \ddots & \vdots \\ a_{21}b_{11} & a_{21}b_{12} & \cdots & a_{2n}b_{1q} \\ a_{21}b_{21} & a_{21}b_{22} & \cdots & a_{2n}b_{2q} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1}b_{p1} & a_{m1}b_{p2} & \cdots & a_{mn}b_{pq} \end{bmatrix}$$

A nossa versão para essa operação recebe como entrada um diretório que deve conter os dados de duas matrizes e pelo menos dois arquivos, cada um com seu *header*. Uma das matrizes deve ser do tipo "A" e a outra do tipo "B". O arquivo de saída dessa operação, o resultado, também está no formato padrão, podendo ser usado como entrada para outras operações que aceitem esse formato. Ao executar essa operação o usuário ainda deve prover um diretório onde será armazenado o resultado e o tipo da matriz do resultado, se é uma matriz "A" ou "B". A escolha do tipo da matriz resultado é necessária para que o usuário possa utilizá-la diretamente como entrada de outra operação.

O código fonte da operação possui uma constante, *NUMBER_ELEMENTS_IN_-MEMORY*, usada para limitar o número de elementos que serão armazenados na memória *RAM* durante parte dos cálculos. Aumentando esse valor podemos aumentar o desempenho apresentado na execução dessa operação, porém se aumentarmos muito pode ocorrer o "estouro" da memória *RAM*. O valor dessa constante deve ser alterado com cuidado, analisando a quantidade de memória *RAM* disponível nos nós utilizados pelo Hadoop.

4.1.2 Multiplicação de Matrizes

A multiplicação de matrizes é uma operação realizada entre duas matrizes e que só pode ocorrer se o número de colunas da matriz à esquerda do operador for igual ao número de linhas da matriz à direita do operador. A operação consiste em multiplicar uma linha de uma matriz por uma coluna da outra matriz. As dimensões da matriz obtida como resultado da multiplicação são dadas pelo número de linhas da matriz à esquerda do operador e pelo número de colunas da matriz à direita do operador. Assim, ao realizarmos a multiplicação entre uma matriz $A_{m\text{x}n}$ e uma matriz $B_{n\text{x}q}$ obtemos uma matriz $C_{m\text{x}q}$.

A multiplicação entre uma matriz $A_{m\text{x}n}$ e uma matriz $B_{n\text{x}q}$ é dada por:

$$A_{m\text{x}n}B_{p\text{x}q} = \begin{bmatrix} a_{11}b_{11} + a_{12}b_{21} + \cdots + a_{1n}b_{n1} & \cdots & a_{11}b_{1q} + a_{12}b_{2q} + \cdots + a_{1n}b_{nq} \\ a_{21}b_{11} + a_{22}b_{21} + \cdots + a_{2n}b_{n1} & \cdots & a_{21}b_{1q} + a_{22}b_{2q} + \cdots + a_{2n}b_{nq} \\ \vdots & \ddots & \vdots \\ a_{m1}b_{11} + a_{m2}b_{21} + \cdots + a_{mn}b_{n1} & \cdots & a_{m1}b_{1q} + a_{m2}b_{2q} + \cdots + a_{mn}b_{nq} \end{bmatrix}$$

Essa operação já possui um algoritmo implementado para uso no Hadoop, como pode ser visto em Leskovec et al. [27]. Baseado nesse algoritmo implementamos a nossa versão fazendo as alterações necessárias para usá-la com arquivos no nosso formato padrão de entrada. Ela recebe como entrada um diretório que deve conter os dados de duas matrizes e pelo menos dois arquivos, cada um com seu *header*. Uma das matrizes deve ser do tipo "A" e a outra do tipo "B". O arquivo de saída dessa operação, o resultado da multiplicação, também encontra-se no formato padrão, podendo ser usado como entrada para outras operações que aceitem esse formato. Ao executar essa operação o usuário ainda deve prover um diretório onde será armazenado o resultado e o tipo da matriz do resultado, se é uma matriz "A" ou "B". Da mesma maneira que ocorreu no produto de Kronecker, a escolha do tipo da matriz resultado é necessária para que o usuário possa utilizá-la diretamente como entrada de outra operação.

4.1.3 Norma de uma Matriz

A norma de uma matriz é uma função que associa um valor não-negativo para uma matriz e que satisfaz as seguintes propriedades, sendo M e N matrizes e λ um número:

- $||M|| = 0 \Leftrightarrow M = 0$
- $||\lambda M|| = |\lambda|||M||$
- $||M + N|| \leq ||M|| + ||N||$

Um número complexo z pode ser escrito como $z = a + bi$, onde a e b são números reais e i é uma unidade imaginária, com $i^2 = -1$.

O cálculo da norma de uma matriz de números complexos pode ser realizado desta forma:

$$||M_{m\text{x}n}|| = \sqrt{\sum_{j=1}^{m} \sum_{k=1}^{n} (a_{jk}^2 + b_{jk}^2)},$$

sendo a_{jk} e b_{jk}, respectivamente, a parte real e a parte imaginária do valor do elemento $M_{j,k}$.

Para verificar se os operadores utilizados são unitários, após as operações de multiplicação ocorridas nos passos da caminhada, calculamos a norma do vetor de estados e verificamos se ela foi preservada. A implementação dessa operação neste trabalho ocorreu com essa finalidade.

A nossa versão dessa operação recebe como entrada um diretório que deve conter os dados do vetor de estados, com o arquivo estando no formato padrão de entrada. O usuário também deve passar o diretório onde o resultado será armazenado. O resultado será um número. Como o estado inicial é representado por um vetor unitário, se o resultado da norma for 1 ou muito próximo de 1 (devido a aproximação numérica durante os cálculos), significa que as matrizes usadas nos cálculos são unitárias.

4.1.4 Valor Absoluto ao Quadrado

A operação que chamamos de valor absoluto ao quadrado na verdade é a realização de duas operações sequencialmente: calcula-se o valor absoluto de um número, seja ele real ou complexo, e depois eleva esse valor ao quadrado.

O valor absoluto de um número real é o seu valor desconsiderando o sinal, ou seja, sendo a um número real:

- Valor absoluto de $-a$ é a;
- Valor absoluto de a é a.

O valor absoluto de um número complexo é calculado de uma forma um pouco diferente. Se considerarmos um número complexo $z = a + bi$, onde a é a parte real, b a parte imaginária e i a unidade imaginária, o valor absoluto de z é dado por:

$$\sqrt{a^2 + b^2}$$

Essa operação foi implementada para ser usada no processo do cálculo da distribuição de probabilidade. Na Seção 2.3, a equação 2.6 é utilizada para realizar o cálculo da distribuição de probabilidade. Podemos dividi-la em duas partes: a primeira parte sendo responsável por calcular o valor absoluto elevado ao quadrado de cada elemento do vetor de estados, e a segunda parte sendo responsável por realizar o somatório.

A entrada dessa operação é um diretório contendo um arquivo no formato padrão de entrada. O usuário também deve passar o diretório onde o resultado será armazenado. O resultado dessa operação não se encontra no formato padrão de entrada, pois os valores de cada elemento são números reais.

4.1.5 Remodelagem

A operação que chamamos de remodelagem realiza apenas a alteração do formato do vetor passado como entrada, sem alterar os seus dados. Essa operação foi implementada para que possamos alterar o formato do vetor de estados para criarmos um *array* que possua as reais dimensões presentes no problema. O vetor de estados pode possuir dados referentes a diversos espaços moeda e posição, tudo agrupado num *array* com apenas um eixo. Essa operação altera o formato desse *array*, dando-lhe o formato real apresentado no problema, com a quantidade correta de eixos. Realizamos essa operação sobre o vetor de estados para que possamos realizar a soma sobre os eixos corretos utilizando a operação de soma sobre os eixos desenvolvida para esse trabalho.

Essa operação pode receber como entrada tanto um diretório contendo um arquivo no formato padrão de entrada quanto um diretório contendo um arquivo no formato da saída da operação valor absoluto ao quadrado. O usuário também deve passar o diretório onde o resultado será armazenado e o novo formato que o *array* possuirá após a operação. Os valores do novo formato devem ser separados pelo caractere "vírgula". Como cada partícula possui o seu espaço moeda e posição, se o problema do usuário possuir três partículas se movimentando numa malha bidimensional de tamanho 30, por exemplo, o formato a ser passado seria:

2,2,30,30,2,2,30,30,2,2,30,30

4.1.6 Soma sobre os Eixos

A operação de soma sobre os eixos realiza o somatório ao longo dos eixos passados na entrada. Utilizamos essa operação para realizar a parte do somatório da equação 2.6, na Seção 2.3. Essa operação ainda aceita como entrada índices pertencentes ao *array*, com a função de fixar uma determinada posição para alguns eixos. Essa opção de fixar valores nos serviu para assumir valores específicos como resultado da medição de alguns subespaços, nos permitindo gerar um arquivo com dados numa dimensão onde pudesse ser gerado um gráfico da distribuição de probabilidade, para facilitar na análise da solução do problema.

Neste trabalho, as operações *valor absoluto ao quadrado*, *remodelagem* e *soma sobre os eixos* foram usadas, nessa ordem, para obtermos o valor da distribuição de probabilidade a partir do vetor de estados.

Essa operação recebe como entrada um diretório contendo um arquivo num formato similar ao arquivo padrão de entrada, com apenas duas diferenças: ao invés de duas dimensões, linhas e colunas, esse arquivo pode possuir várias dimensões (eixos); e os valores são números reais. O usuário também deve passar o diretório

onde o resultado será armazenado e os eixos sobre os quais a soma será realizada. Opcionalmente o usuário ainda pode escolher índices para fixar uma determinada posição para alguns eixos do *array*. Os eixos especificados devem estar separados pelo caractere "vírgula", e o primeiro eixo é representado pelo número um. Para usar a opção de passar índices para fixar uma posição, o usuário deve saber as dimensões do *array*, que se encontra no *header* do arquivo de entrada. O usuário deve passar como entrada uma sequência de números inteiros separados pelo caractere "vírgula" com a mesma quantidade de eixos do arquivo de entrada, substituindo os índices não fixos pelo caractere "?".

Se, por exemplo, o usuário possuir um *array* com as dimensões 2,2,5,5,2,2,5,5 e quiser realizar a soma sobre os eixos 1, 2, 5 e 6 e fixar a posição 2 para os eixos 7 e 8, ele deve passar na entrada: $1, 2, 5, 6$ e $?, ?, ?, ?, ?, ?, 2, 2$. O resultado será um arquivo com um *array* com dois eixos.

4.2 Simulador

No âmbito da simulação de algoritmos quânticos uma operação bastante usada é a multiplicação de uma sequência de operadores unitários, representados por matrizes unitárias U, e um vetor ψ. Como uma das operações desenvolvidas para uso neste trabalho foi exatamente a de multiplicação de matrizes, decidimos que esse seria o tipo de simulação realizada pelo nosso programa. Então, podemos dizer que basicamente o nosso simulador executa t vezes a seguinte sequência de multiplicação de matrizes:

$$\psi(t) = U_1 * U_2 * U_3 ... U_n * \psi(t-1),$$

onde t é um número inteiro positivo referente ao número de passos da simulação, n a quantidade de operadores unitários U presentes no problema e $\psi(t)$ o vetor de estados após t passos, onde $\psi(0)$ seria estado inicial do problema.

Por questão de otimização, essa sequência de multiplicações de matrizes é realizada da direita para a esquerda, pelo mesmo motivo apresentado na Seção 3.3, sobre o funcionamento do algoritmo do QW. A forma como foi implementado o algoritmo do Quandoop foi muito similar ao algoritmo do QW, após a parte que gera os dados, então optamos por não descrevê-lo no trabalho. O código fonte e o executável do Quandoop[2] estão disponíveis para *download*.

O simulador possui um arquivo de configuração, *config.properties*, que possui parâmetros que devem ser preenchidos para que o programa possa executar e outros parâmetros opcionais. Esse arquivo deve ser colocado no mesmo diretório onde se encontra o arquivo executável do Quandoop. Os parâmetros são os seguinte:

[2] `https://github.com/david-ufrj/qw-hadoop/tree/master/Quandoop`

- *steps*: A quantidade de passos que serão executados na simulação. Deve ser um número inteiro positivo;

- *paths*: O caminho para um arquivo local que contenha os caminhos para cada um dos diretórios das matrizes U e do vetor ψ. Esse arquivo deve conter um caminho por linha e os caminhos devem ser preenchidos na mesma ordem da sequência da multiplicação, com o caminho do arquivo do vetor ψ sendo o último a ser preenchido. Todas as matrizes U devem ser matrizes do tipo "A", do formato padrão de entrada usado neste trabalho, e o vetor ψ deve ser do tipo "B";

- *workDir*: O caminho que será usado no HDFS para o programa armazenar seus dados durante a execução;

- *jarDir*: O diretório local onde estão os arquivos *.jar* necessários para a execução do Quandoop, o *quandoop.jar* e o *operations.jar*;

- *outputDir*: O caminho para um diretório local onde o resultado será armazenado no final da execução. Esse diretório não precisa ser criado, o programa já faz isso. Se esse diretório já existir, ele deve estar vazio, ou todos os dados presentes nele serão apagados;

- *dimensions* (opcional): As dimensões do subespaço de Hilbert. Devem ser números inteiros positivos e estar separados pelo caractere "vírgula";

- *measurement* (opcional): Os índices dos subespaços de Hilbert que se deseja medir. Devem ser números inteiros positivos e estar separados pelo caractere "vírgula";

- *saveStates* (opcional): Permite que sejam salvos estados parciais da simulação. Cada passo múltiplo desse valor será salvo junto ao resultado final. Deve ser um número inteiro positivo menor que o do parâmetro *steps*.

Uma vez que os parâmetros necessários foram preenchido de forma correta e os arquivos contendo os dados de entrada estão no formato padrão utilizado neste trabalho, o Quandoop pode ser executado. No início o Quandoop copia os arquivos de entrada para o HDFS, para que possam ser usados pelas nossas operações, e depois configura algumas de suas variáveis de acordo com os valores presentes no arquivo de configuração. Após isso ele realiza alguns cálculos e começa a executar a simulação. Ao chegar no fim da execução, o resultado ($\psi(t)$, distribuição de probabilidade para $\psi(t)$, norma e estados parciais) será copiado para a pasta especificada pelo usuário e todos os dados usados pelo Quandoop presentes HDFS serão apagados.

Capítulo 5

Experimentos

Os computadores atuais ainda possuem uma arquitetura hierárquica que permite que o processador apenas tenha acesso a dados presentes na memória principal, a memória RAM. Assim, os dados presentes em discos (memória secundária) devem ser carregados para essa memória antes de poderem ser usados pelo processador. A memória RAM é ainda capaz de executar um número bem maior de operações e possui um tempo de acesso bem inferior quando comparada aos discos. Essas características, entre outras, permite que programas executados utilizando como entrada dados armazenados na memória RAM apresentem um desempenho bem superior aqueles que precisam acessar os dados em discos. Sendo assim devemos ter em mente que, se a quantidade de dados a ser processados podem ser armazenados na memória RAM, uma versão sequencial do programa pode ter um desempenho bem superior ao apresentado pela versão que utiliza o Hadoop.

Para realizar os nossos experimentos primeiro implementamos códigos sequenciais escritos em Python utilizando as bibliotecas *numpy*[1] e *scipy.sparse*[2], para realizar de forma eficiente os cálculos executados sobre matrizes esparsas. Utilizamos esse código para avaliar como seria a execução do problema de forma sequencial e qual seria o limite imposto pela configuração do *hardware* utilizado, além de comparar com a nossa implementação usando o Hadoop.

Este capítulo tem como objetivo descrever o ambiente de execução e apresentar os resultados obtidos com os testes. A Seção 5.1 descreve o ambiente de execução, apresentando os *softwares* e *hardwares* utilizados durante os testes. A Seção 5.2 apresenta os resultados obtidos com a execução do QW. A Seção 5.3 apresenta os resultados obtidos com a execução do Quandoop . A Seção 5.4 apresenta os resultados obtidos com a execução do código do algoritmo de Grover, ainda não mencionado neste trabalho.

[1] `http://www.numpy.org/`
[2] `http://docs.scipy.org/doc/scipy/reference/sparse.html`

5.1 Ambiente de Execução

5.1.1 Software

Para executar os experimentos utilizando os códigos sequenciais utilizamos um notebook com o sistema operacional Microsoft Windows 7 Pro e Python versão 2.7.9, e uma Workstation HP com o sistema operacional Ubuntu 15.04 e Python versão 2.7.9.

Para executar os experimentos com as operações desenvolvidas neste trabalho, utilizamos um cluster com o sistema operacional Rocks Cluster [28] versão 6.1.1, Apache Hadoop versão 1.2.1 e Java 1.7.0_51.

5.1.2 Hardware

O notebook utilizado possui um processador Intel Core 2 Duo P7350 com 2.0 GHz (2 núcleos) e 4 GB de RAM. A Workstation HP possui um processador Intel XEON E5-1620 com 3.6 GHz (4 núcleos com Hyper-Threading [29]), 28 GB de RAM com *ECC*[3] (do inglês Error-Correcting Code) e uma placa gráfica Nvidia Quadro K600 com 1 GB de RAM.

O cluster utilizado é composto por 32 computadores com as mesmas configurações de *hardware* (1 *frontend*[4] e 31 nós). Os computadores possuem um processador Intel Core i7 3770 com 3.4 GHz (4 núcleos com Hyper-Threading), 8 GB de RAM, disco rígido de 1 TB com 7200 RPM (Rotações Por Minuto) e placa de rede Intel 82579LM Gigabit. Apenas uma das máquinas, o *frontend*, possui uma segunda placa de rede ethernet 10/100 para acesso externo. Esse cluster está em funcionamento no laboratório NUMPEX-Comp, localizado no Polo Xerém da Universidade Federal do Rio de Janeiro.

5.1.3 Hadoop: Ajuste nas Configurações para Otimização de Desempenho

A Apache Hadoop possui diversas configurações que podem ser alteradas para se obter um melhor desempenho, levando em conta o número de nós e o *hardware* presente neles. Utilizamos como base para realizarmos os ajustes as informações apresentadas em Tannir [30] e as presentes no site da Apache.

Para realizar os ajustes, alteramos os parâmetros presentes em três arquivos localizados na pasta *conf* do diretório raiz onde foi instalado o Hadoop.

[3]Memórias RAM com *ECC* podem detectar e corrigir erros comuns com dados internos corrompidos.

[4]Servidor que administra o sistema, controlando, monitorando e distribuindo as tarefas entre os usuários.

Os parâmetros alterados no arquivo *core-site.xml* foram:

- fs.default.name = hdfs://numpex-hpc:9000
  - Definimos o *frontend* do cluster como sendo o *Namenode*.
- io.file.buffer.size = 65536
  - Alteramos para 64 KB o tamanho do *buffer* utilizado pelo Hadoop durante operações de entrada e saída.

Os parâmetros alterados no arquivo *hdfs-site.xml* foram:

- dfs.name.dir = /state/partition1/hadoop
  - Escolhemos um diretório que fosse local a cada nó para armazenar os metadados do HDFS.
- dfs.data.dir = /state/partition1/hadoop/data
  - Escolhemos um diretório que fosse local a cada nó para armazenar os dados do HDFS.
- dfs.block.size = 134217728
  - Alteramos para 128 MB o tamanho dos blocos de dados utilizados pelo HDFS.
- dfs.namenode.handler.count = 30
  - Alteramos para 30 o número de *threads* utilizadas pelo *Namenode*.
- dfs.datanode.handler.count = 6
  - Alteramos para 6 o número de *threads* utilizadas pelos *Datanodes*.

Os parâmetros alterados no arquivo *mapred-site.xml* foram:

- mapred.job.tracker = numpex-hpc:8021
  - Definimos o *frontend* do cluster como sendo o *JobTracker*.
- mapred.local.dir = /state/partition1/hadoop
  - Escolhemos um diretório que fosse local a cada nó para armazenar os dados intermediários gerados durante a execução do Hadoop MapReduce.
- mapred.tasktracker.map.tasks.maximum = 5

- Alteramos para 5 o número de tarefas *map* utilizadas simultaneamente pelos *TaskTrackers*.

- mapred.tasktracker.reduce.tasks.maximum = 6
  - Alteramos para 6 o número de tarefas *reduce* utilizadas simultaneamente pelos *TaskTrackers*.

- mapred.map.tasks = 155
  - Alteramos para 155 o valor do número máximo de tarefas *map* por *job*. Esse valor foi definido de acordo com a quantidade de núcleos de processamento com Hyper-Threading que todos os nós possuem juntos. A conta realizada é: calcula a quantidade de núcleos reais menos um e multiplica por 1.5. Arredonda esse valor para que seja um número inteiro e depois multiplica pela quantidade de nós que possuem um processador com as mesmas características desse.

- mapred.reduce.tasks = 186
  - Alteramos para 186 o valor do número máximo de tarefas *reduce* por *job*. Esse valor foi definido de acordo com a quantidade de núcleos de processamento com Hyper-Threading que todos os nós possuem juntos. A conta realizada é: calcula a quantidade de núcleos reais e multiplica por 1.5. Arredonda esse valor para que seja um número inteiro e depois multiplica pela quantidade de nós que possuem um processador com as mesmas características desse.

- mapred.reduce.parallel.copies = 15
  - Alteramos para 15 o número de transferências paralelas durante a etapa de *shuffling*. Essa configuração deve ser alterada de acordo com a velocidade da rede utilizada pelo cluster.

- mapred.submit.replication = 5
  - Alteramos o valor para 5. Esse valor deve ser a raiz quadrada do número de nós utilizados pelo Hadoop.

- mapred.child.java.opts = -Xmx600m
  - Alteramos para 600 MB a quantidade máxima de RAM utilizada por cada tarefa do Hadoop MapReduce. Esse valor deve ser menor que: a quantidade de memória RAM de cada nó menos o valor de *io.sort.mb*. Pega esse valor e divide pela

soma dos valores de *mapred.tasktracker.map.tasks.maximum* e *mapred.tasktracker.reduce.tasks.maximum*.

- io.sort.mb = 250
  - Alteramos para 250 MB o tamanho do *buffer* utilizado para realizar a ordenação dos arquivos.
- io.sort.factor = 25
  - Alteramos para 25 a quantidade de fluxos para serem combinados enquanto os arquivos são ordenados.
- io.sort.spill.percent = 0.95
  - Alteramos a porcentagem do limite utilizado pelo *buffer* para 95%. Após atingir esse valor o conteúdo do *buffer* é passado para o disco.
- mapred.jobtracker.taskScheduler = org.apache.hadoop.mapred.CapacityTaskScheduler
  - Alteramos o agendador de tarefas para um que fosse mais eficiente em lidar com *jobs* que possuem arquivos de entrada grandes.
- mapred.jobtracker.taskScheduler.maxRunningTasksPerJob = 248
  - Alteramos para 248 o valor da quantidade máxima de tarefas *map* e *reduce* que podem ser executadas simultaneamente. Esse valor foi definido pela quantidade de núcleos de processamento totais presentes em todos os nós utilizados pelo Hadoop.
- mapred.reduce.slowstart.completed.maps = 0.4
  - Alteramos para 0.4 a quantidade de tarefas *map* presentes em um *job* que devem ser completadas antes que a etapa de *reduce* seja agendada para o *job*.
- mapred.compress.map.output = true
  - Ativamos a compactação dos dados de saída das tarefas *map*. Essa configuração ajuda a reduzir o tempo de escrita dos dados de arquivos grandes em discos.
- mapred.map.output.compression.codec = org.apache.hadoop.io.compress.GzipCodec
  - Escolhemos o *codec* de compressão Gzip por ele nos prover uma boa relação entre tempo para compressão e tamanho final do arquivo.

- mapred.output.compression.type = BLOCK
  - Alteramos o tipo de compressão utilizada para *BLOCK* para melhorar a taxa de compressão.

5.2 Resultados: QW

Começamos nossos testes executando os códigos sequenciais escritos em Python com duas finalidades: obter o tempo de execução e conhecer o maior valor para a variável *size*, responsável por definir o tamanho da malha bidimensional, como explicado na Seção 3.3. O valor da variável *size* é limitado pela quantidade de memória RAM disponível no computador. A quantidade de passos da simulação foi definido como sendo a metade do valor da variável *size*, arredondando quando o valor obtido não for um número inteiro. Como na condição inicial as partículas encontram-se no meio da malha, não corremos o risco das partículas ultrapassarem os limites impostos pelo tamanho da malha durante a caminhada.

Em seguida realizamos os testes usando o Hadoop, com as mesmas finalidades do código sequencial. Porém na versão do código do Hadoop, o valor da variável *size* é limitado pela quantidade de espaço disponível no HDFS. No nosso caso, com 31 nós, possuímos 27.15 TB. Como usamos o fator de replicação três, padrão do Hadoop, o espaço efetivo que temos para uso é 9.05 TB. A quantidade de passos da simulação foi definida da mesma forma que no código sequencial.

Os resultados obtidos foram gerados a partir da média aritmética dos tempos de três execuções para cada valor da variável *size*, tanto para o código sequencial quanto para o código que utiliza o Hadoop.

A Tabela 5.1 possui o tempo, a quantidade de passos da simulação e o valor máximo (suportado pelos computadores antes de "estourar"a memória) para a variável *size* executando o código sequencial no notebook e na Workstation HP.

Tabela 5.1: Tempo de execução da simulação da caminhada quântica com dois caminhantes numa malha bidimensional utilizando o código sequencial.

Computador	*size*	Passos	Tempo de Execução (segundos)
Notebook	23	12	102.50
Workstation HP	42	21	656.41

A Tabela 5.2 possui o tempo de geração dos dados, o tempo de execução dos passos da simulação, o tempo total (o tempo de geração somado com o tempo de execução), a quantidade de recurso utilizado e o valor da variável *size* para os testes realizados usando o Hadoop. Devido ao tempo disponível na fila para o uso do cluster, o valor máximo da variável *size* utilizado foi sessenta. Dividimos esse valor pela metade duas vezes, para realizar mais dois testes, e também executamos para

os mesmos valores usados no código sequencial. O recurso utilizado é referente ao espaço usado de disco, sem considerar a replicação, ou seja, representa o tamanho dos arquivos necessários para a execução.

Tabela 5.2: Tempos para a execução da simulação da caminhada quântica com dois caminhantes numa malha bidimensional utilizando o código para Hadoop.

size	Passos	Recurso (GB)	Geração (s)	Execução (s)	Tempo Total (s)
15	8	0.25	256.11	1743.72	1999.83
23	12	1.49	292.67	2786.00	3078.67
30	15	4.39	371.36	3914.38	4285.74
42	21	17.41	576.82	7063.86	7640.69
60	30	74.99	1617.80	14029.70	15647.50

A Tabela 5.3 possui o desvio padrão e o erro padrão para os tempos totais obtidos com a execução do QW.

Tabela 5.3: Desvio padrão e erro padrão para os tempos totais da simulação da caminhada quântica com dois caminhantes numa malha bidimensional utilizando o código para Hadoop.

size	Passos	Desvio Padrão (s)	Erro Padrão (s)
15	8	133.41	77.02
23	12	12.04	6.95
30	15	43.24	24.96
42	21	77.37	44.67
60	30	130.96	75.61

Como já era esperado, podemos observar na Tabela 5.3 que o tempo de execução variou muito pouco, uma vez que durante a execução o cluster só estava executando o nosso código.

Para ficar mais fácil de visualizar os dados presentes na Tabela 5.2, eles foram colocados em dois gráficos, Figura 5.1 e Figura 5.2, um para o tempo e outro para os recursos utilizados, respectivamente.

Observando a Figura 5.1 e a Figura 5.2 podemos perceber que a variação do tempo em relação a quantidade de dados começou a ficar similar apenas com valores da variável *size* maiores que 30. Isso ocorreu pois até esse valor o cluster ainda estava sendo subutilizado pelo Hadoop.

Para comparar os tempos obtidos usando o código sequencial com os tempos obtidos usando o Hadoop, criamos o gráfico presente na Figura 5.3. Nele podemos observar que o tempo necessário para completar a execução da simulação utilizando o código sequencial é bem menor que utilizando a versão para Hadoop. Essa comparação foge da nossa proposta e foi realizada apenas no intuito de verificar a viabilidade de utilizar o Hadoop em casos onde não ocorre "estouro" na memória RAM.

Para avaliar o ganho com o paralelismo proporcionado pelo Hadoop, executamos testes com o mesmo valor da variável *size* e com a mesma quantidade de passos,

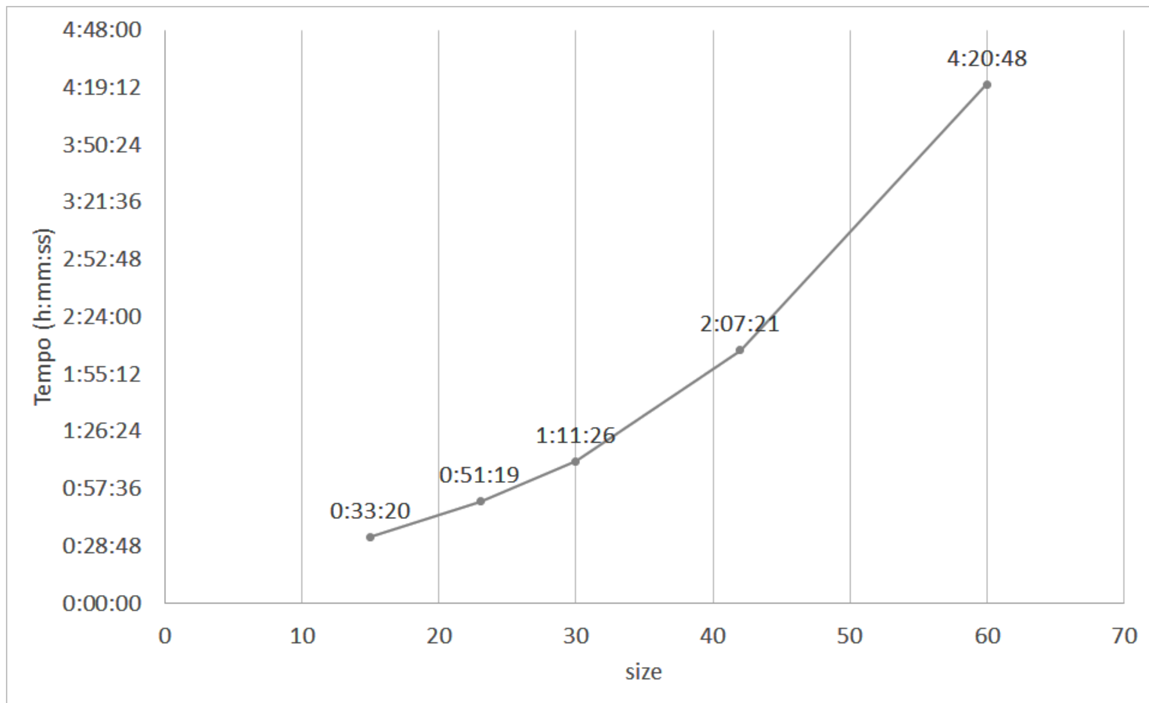

Figura 5.1: Gráfico com o tempo gasto para a execução do QW

alterando apenas a quantidade de nós utilizados pelo Hadoop. Para esses testes utilizamos 1, 2, 4, 8, 16 e 31 nós, os valores 30 e 42 para a variável *size* e apenas um passo, para ambos os valores da variável *size*. Esses valores para a variável *size* foram escolhidos após compararmos os gráficos de tempo e de recurso utilizado, Figura 5.1 e Figura 5.2. Os tempos obtidos foram colocados na Tabela 5.4 e na Tabela 5.5.

Tabela 5.4: Tempos para a execução da simulação da caminhada quântica com dois caminhantes numa malha bidimensional para size=30 utilizando quantidades diferentes de nós.

Quantidade de nós	Geração (s)	Execução (s)	Tempo Total (s)
1	559.85	651.98	1211.83
2	402.17	508.41	910.58
4	346.26	415.01	761.27
8	339.92	414.35	754.27
16	324.12	356.55	680.67
31	356.70	419.13	775.83

Esses resultados podem ser melhor visualizados nas tabelas de *speedup* e eficiência (Tabela 5.6 e Tabela 5.7), e nos gráficos de *speedup* (Figuras 5.4 e Figura 5.6) e eficiência (Figuras 5.5 e Figura 5.7).

Ao observarmos o gráfico presente na Figura 5.4 podemos perceber que houve ganho ao adicionar mais nós ao Hadoop, porém somente até 16 nós. O teste realizado com 31 nós apresentou um *speedup* pior que o teste realizado com 16 nós. O gráfico presente na Figura 5.5 nos mostra que a eficiência cai a medida que aumentamos

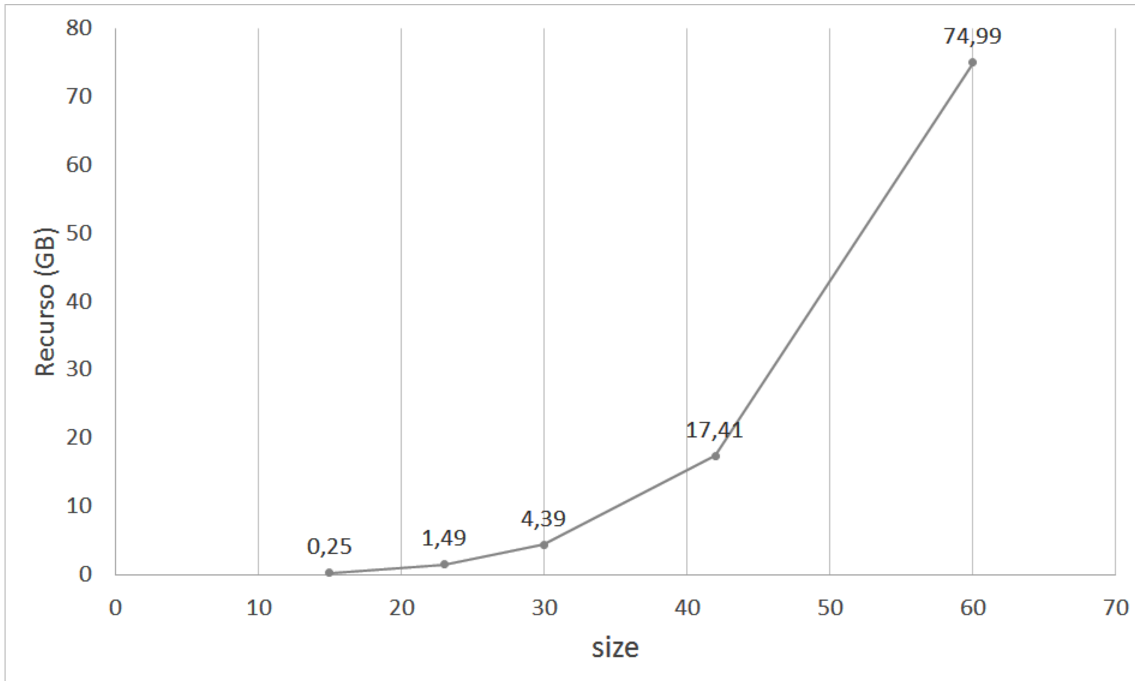

Figura 5.2: Gráfico com o armazenamento gasto com os arquivos de entrada do QW

Tabela 5.5: Tempos para a execução da simulação da caminhada quântica com dois caminhantes numa malha bidimensional para size=42 utilizando quantidades diferentes de nós.

Quantidade de nós	Geração (s)	Execução (s)	Tempo Total (s)
1	1639.91	1681.97	3321.87
2	1078.46	991.03	2069.48
4	725.81	720.13	1445.95
8	534.28	551.23	1085.51
16	472.90	476.38	949.28
31	594.96	495.56	1090.52

Tabela 5.6: Speedup e Eficiência para os dados da Tabela 5.4.

Quantidade de nós	Speedup	Eficiência
1	1.00	1.00
2	1.33	0.67
4	1.59	0.40
8	1.61	0.20
16	1.78	0.11
31	1.56	0.05

a quantidade de nós, atingindo um valor bem baixo usando 31 nós. Essa queda no *speedup* e na eficiência ocorre devido a quantidade de dados passados na entrada, que ainda é pouca para serem processados utilizando muitos nós.

Observando o gráfico presente na Figura 5.6 podemos perceber que houve ganho ao adicionar mais nós ao Hadoop, porém somente até 16 nós. O teste realizado com 31 nós apresentou um *speedup* pior que o teste realizado com 16 nós. O gráfico presente na Figura 5.7 nos mostra que a eficiência cai a medida que aumentamos a quantidade de nós, atingindo um valor baixo usando 31 nós. Essa queda no *speedup* e

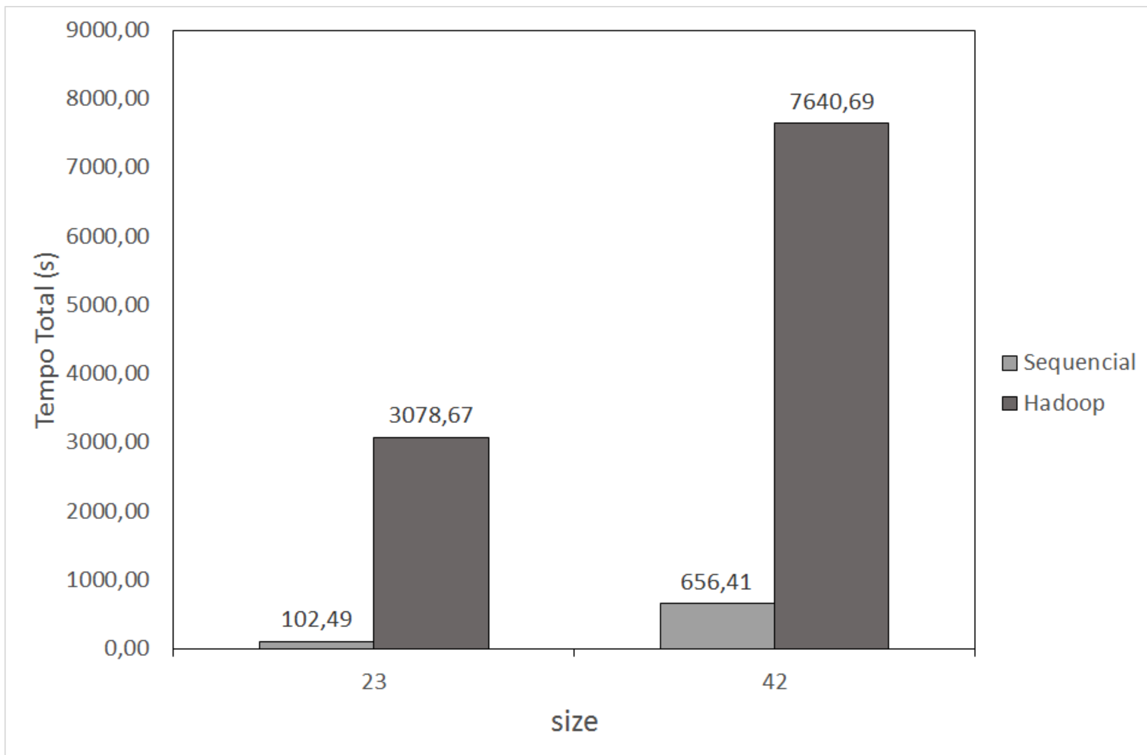

Figura 5.3: Comparação dos tempos totais para a execução do código QW sequencial e da versão para Hadoop em cenários onde não ocorrem "estouro" na memória RAM utilizando o código sequencial.

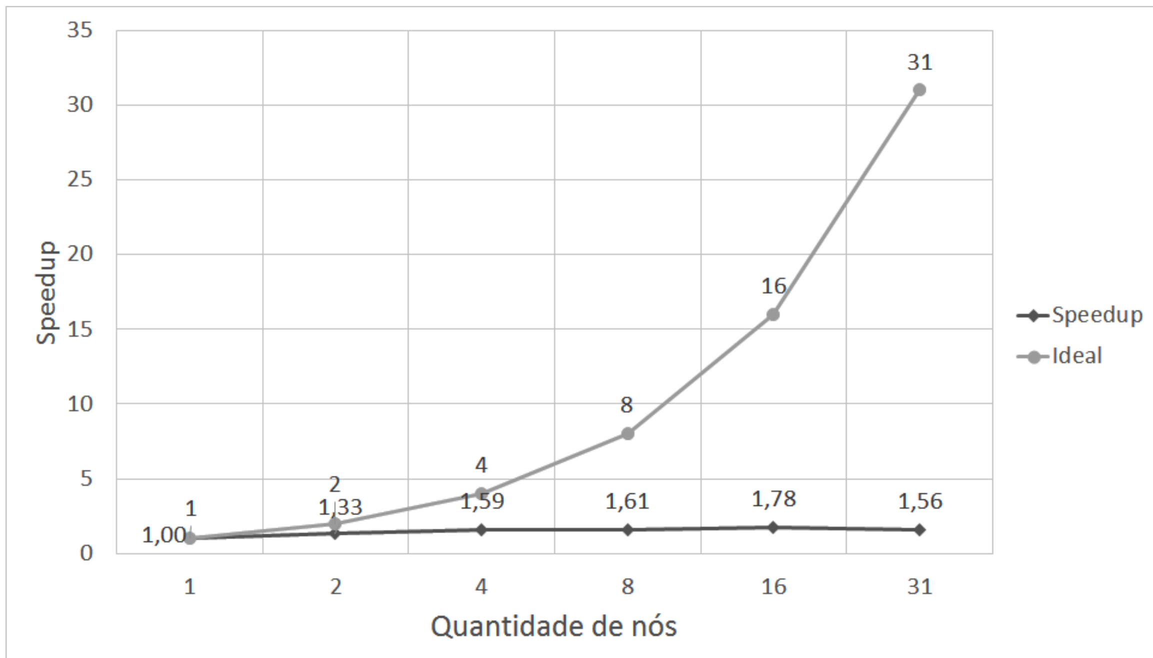

Figura 5.4: Speedup para size=30.

na eficiência ocorre devido a quantidade de dados passados na entrada, que continua sendo pouca para serem processados utilizando muitos nós.

Comparando os resultados de *speedup* para os valores 30 e 42 da variável *size* podemos observar que todos os valores de *speedup* para *size* igual a 42 foram melhores

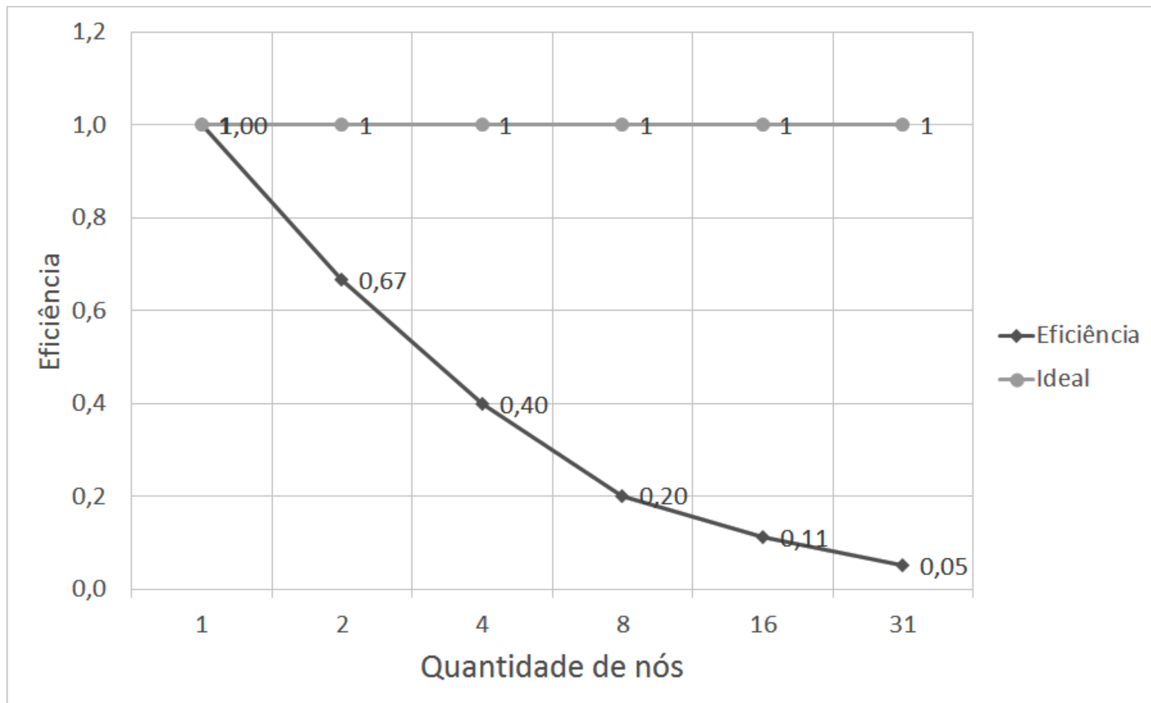

Figura 5.5: Eficiência para size=30.

Tabela 5.7: Speedup e Eficiência para os dados da Tabela 5.5.

Quantidade de nós	Speedup	Eficiência
1	1.00	1.00
2	1.61	0.80
4	2.30	0.57
8	3.06	0.38
16	3.50	0.22
31	3.05	0.10

que os valores obtidos com *size* igual a 30. A eficiência também foi melhor para *size* igual a 42, atingindo o dobro do valor obtido com *size* igual a 30, utilizando 16 e 31 nós.

A Figura 5.8 e a Figura 5.9 são gráficos gerados a partir da distribuição de probabilidade do resultado obtido no fim da execução, presente no vetor de estados para os valores máximos da variável *size*, executando o código sequencial no notebook e na Workstation HP, respectivamente. Essa distribuição de probabilidade é pertencente a uma das partículas dado que a outra colapsou para o centro da malha. Essas figuras foram geradas utilizando a biblioteca *matplotlib*[5] para Python.

5.3 Resultados: Quandoop

Como descrito na Seção 4.2, o Quandoop recebe como entrada arquivos contendo matrizes unitárias e um vetor unitário, e realiza a execução da simulação a partir

[5] `http://matplotlib.org/`

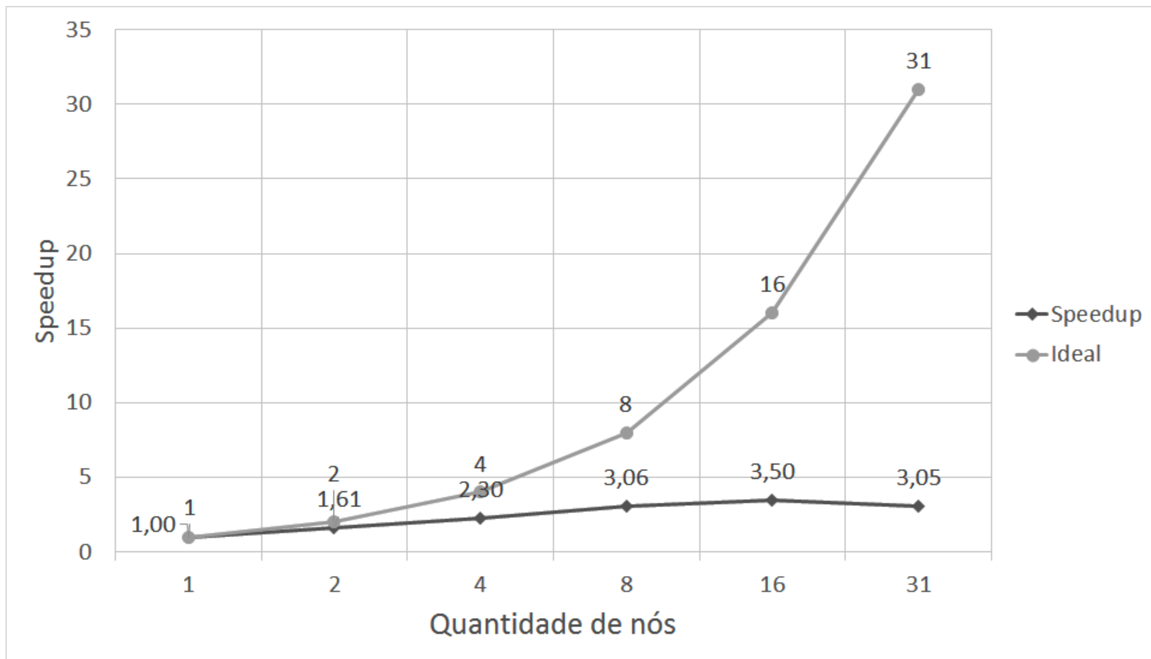

Figura 5.6: Speedup para size=42.

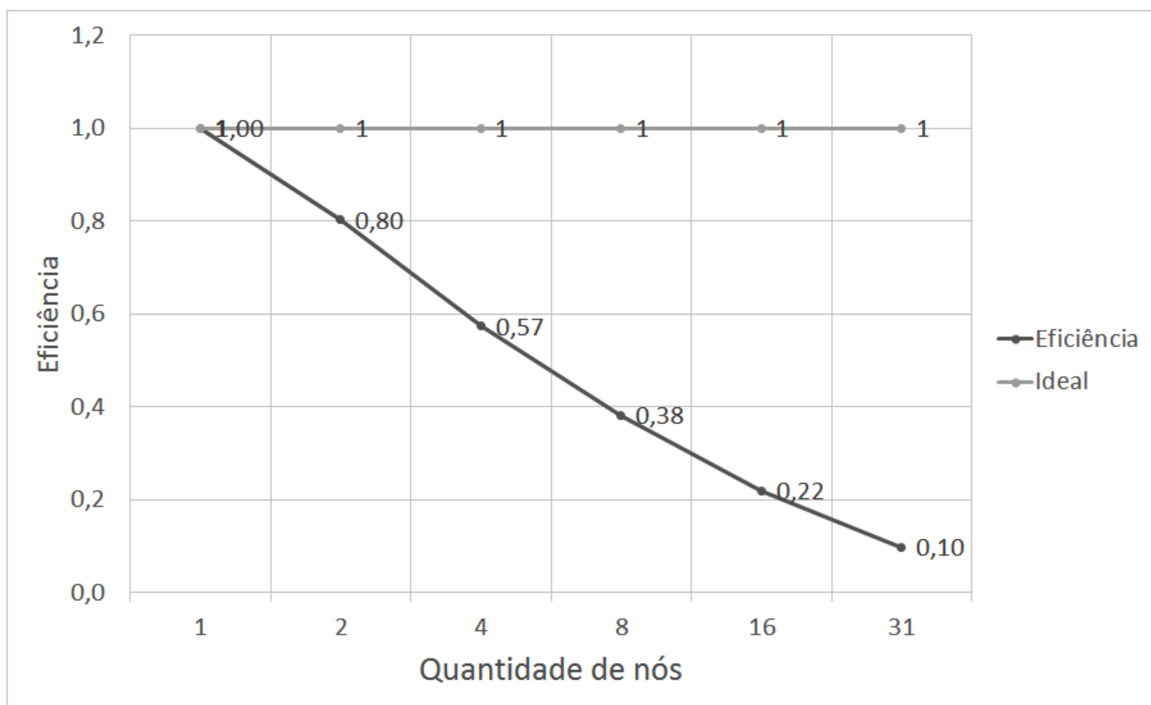

Figura 5.7: Eficiência para size=42.

de sucessivas sequências de multiplicações. Porém para executarmos o Quandoop primeiro temos que possuir esses arquivos de entrada. Com o único propósito de gerar esses dados, implementamos o código QWD[6], de *Quantum Walk Data*.

O QWD gera os arquivos com os dados necessários para realizar a simulação de uma caminhada quântica unidimensional com quatro caminhantes. Nesse código

[6] `https://github.com/david-ufrj/qw-hadoop/tree/master/QWD`

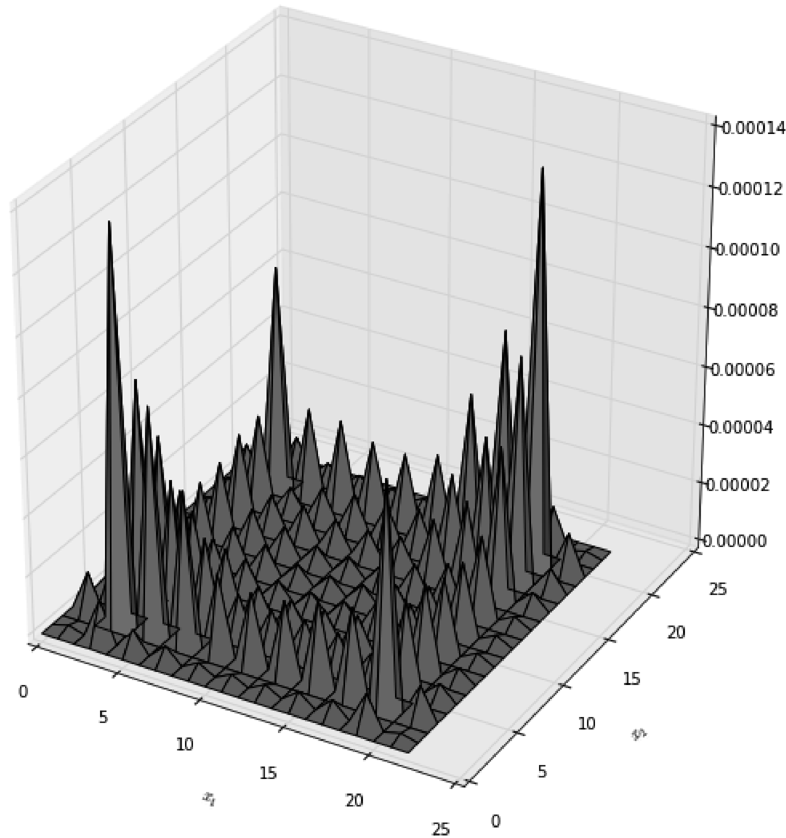

Figura 5.8: Distribuição de probabilidade do vetor de estados para *size* igual a 23 após 12 passos.

podemos alterar o valor da variável *size* para gerarmos diferentes entradas para o Quandoop. Não iremos entrar em detalhes sobre o código devido a sua similaridade com o código QW e porque esse código foi implementado apenas com o fim de gerar dados, para assim podermos testar o Quandoop.

Começamos nossos testes executando os códigos sequenciais escritos em Python com duas finalidades: obter o tempo de execução e conhecer o maior valor para a variável *size*, responsável por definir o tamanho da linha sobre a qual as quatro partículas irão se movimentar. O valor da variável *size* é limitado pela quantidade de memória RAM disponível no computador. A quantidade de passos da simulação foi definido como sendo a metade do valor da variável *size*, arredondando quando o valor obtido não for um número inteiro. Assim como nos testes realizados com o QW, a condição inicial para as quatro partículas foi definida como sendo o centro da linha, para não corrermos o risco de ultrapassarem os limites impostos pelo tamanho da linha durante a caminhada.

Em seguida realizamos os testes usando o Hadoop, com as mesmas finalidades do código sequencial. Porém, como dito na seção anterior, na versão do código que utiliza o Hadoop, o valor da variável *size* é limitado pela quantidade de espaço

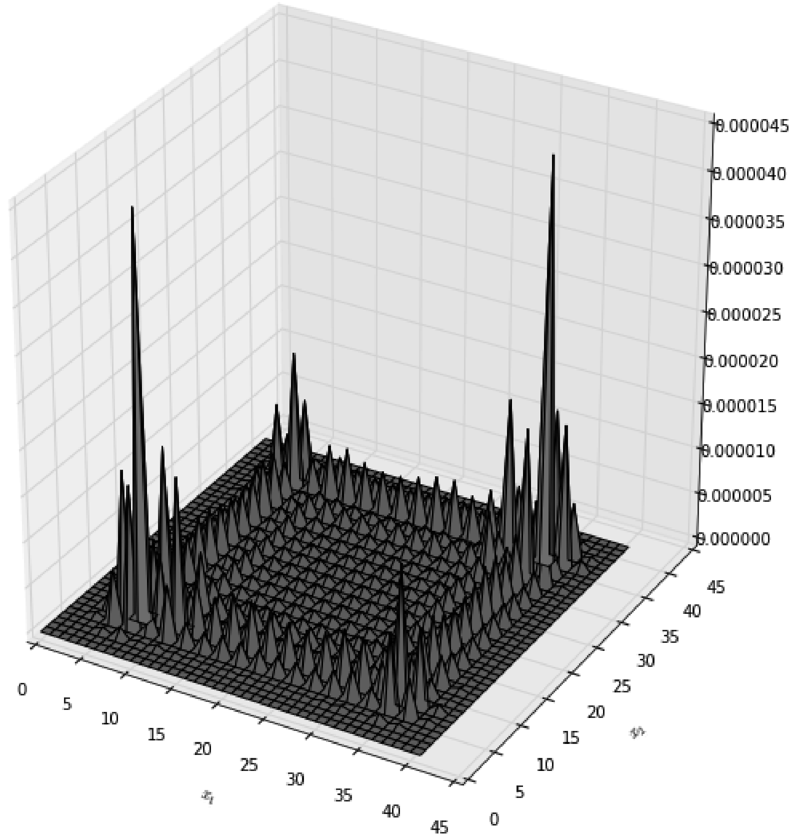

Figura 5.9: Distribuição de probabilidade do vetor de estados para *size* igual a 42 após 21 passos.

disponível no HDFS. A quantidade de passos da simulação foi definida da mesma forma que no código sequencial. Definimos o valor para o parâmetro *saveStates* para um terço da quantidade de passos. Assim junto com o resultado para cada valor de *size*, teríamos três estados parciais da simulação. Quando o valor obtido para esse parâmetro não foi um número inteiro, pegamos apenas a parte inteira desse valor. Assim não corremos o risco de não satisfazer as restrições desse parâmetro.

Os resultados obtidos foram gerados a partir da média aritmética dos tempos de três execuções para cada valor da variável *size*, tanto para o código sequencial quanto para o Quandoop.

A Tabela 5.8 possui o tempo, a quantidade de passos da simulação e o valor máximo para a variável *size* executando o código sequencial no notebook e na Workstation HP.

Tabela 5.8: Tempo de execução da simulação da caminhada quântica com quatro caminhantes numa linha utilizando o código sequencial, após os dados serem gerados.

Computador	*size*	Passos	Tempo de Execução (segundos)
Notebook	27	14	17.02
Workstation HP	51	26	108.54

A Tabela 5.9 possui o tempo de execução da simulação, a quantidade de recurso

utilizado, o valor da variável *size* utilizado no QWD para gerar os dados de entrada e o valor do parâmetro *saveStates* para os testes utilizando o Quandoop. Devido ao tempo disponível na fila para o uso do cluster, o valor máximo da variável *size* utilizado foi sessenta. Dividimos esse valor pela metade duas vezes, para realizar mais dois testes, e também executamos o Quandoop para os mesmos valores usados no código sequencial. O recurso utilizado é referente ao tamanho dos arquivos de entrada, gerados pelo QWD.

Tabela 5.9: Tempos para a execução da simulação da caminhada quântica com quatro caminhantes numa linha utilizando o Quandoop.

size	Passos	saveStates	Recurso (GB)	Tempo de Execução (s)
15	8	2	0.23	2230.62
27	14	4	2.60	3951.67
30	15	5	4.00	4275.88
51	26	8	34.80	11825.86
60	30	10	68.30	15626.56

A Tabela 5.10 possui o desvio padrão e o erro padrão para os tempos totais obtidos com a execução do Quandoop.

Tabela 5.10: Desvio padrão e erro padrão para os tempos totais da simulação da caminhada quântica com dois caminhantes numa malha bidimensional utilizando o código para Hadoop.

size	Passos	Desvio Padrão (s)	Erro Padrão (s)
15	8	65.90	38.05
27	14	107.71	62.18
30	15	56.29	32.50
51	26	79.16	45.70
60	30	112.04	64.69

Observando a Tabela 5.10 podemos perceber que o tempo de execução variou muito pouco, assim como no teste com o QW, uma vez que durante a execução o cluster só estava executando o nosso código.

Para ficar mais fácil de visualizar os dados presentes na Tabela 5.9, eles foram colocados em dois gráficos, Figura 5.10 e Figura 5.11, um para o tempo e outro para os recursos utilizados, respectivamente.

Observando a Figura 5.10 e a Figura 5.11 podemos perceber que a variação do tempo entre 15 e 30 (1.92x) é bem menor que a variação da quantidade da dados passados na entrada (17.39x). Entre 30 e 60 a variação do tempo aumentou (3.65x), enquanto a variação da quantidade de dados foi foi praticamente a mesma (17.08x).

Para comparar os tempos obtidos usando o código sequencial com os tempos obtidos usando o Quandoop, criamos o gráfico presente na Figura 5.12. O gráfico foi colocado na escala logarítmica pois os tempos obtidos com o código sequencial não estavam sendo mostrados. Nele podemos observar que o tempo necessário para completar a execução da simulação utilizando o código sequencial é bem menor que

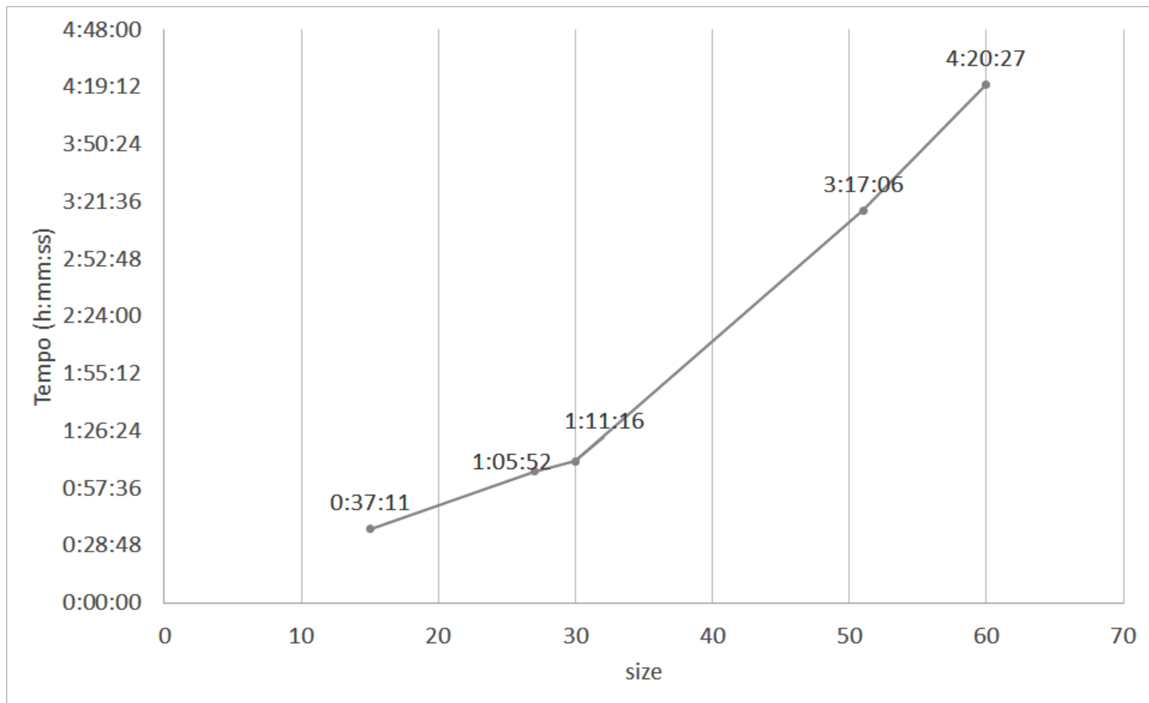

Figura 5.10: Gráfico com o tempo gasto para a execução do Quandoop

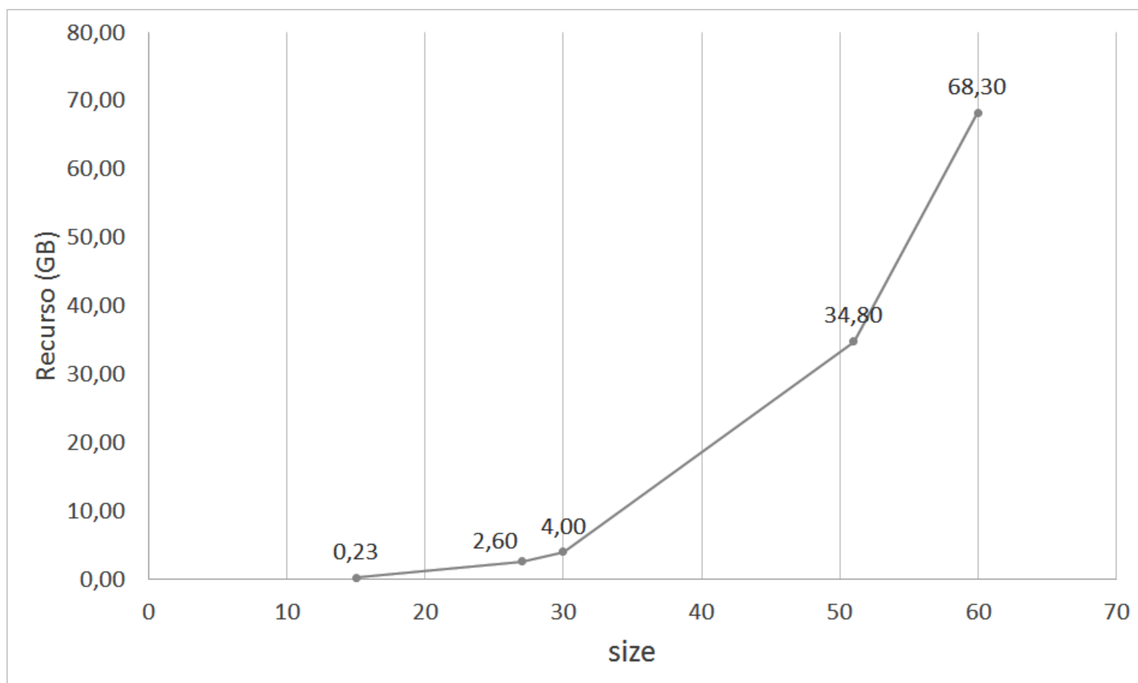

Figura 5.11: Gráfico com o tamanho dos arquivos de entrada do Quandoop

utilizando o Quandoop. Essa comparação foge da nossa proposta e foi realizada apenas no intuito de verificar a viabilidade de utilizar o Quandoop em casos onde não ocorre "estouro" na memória RAM.

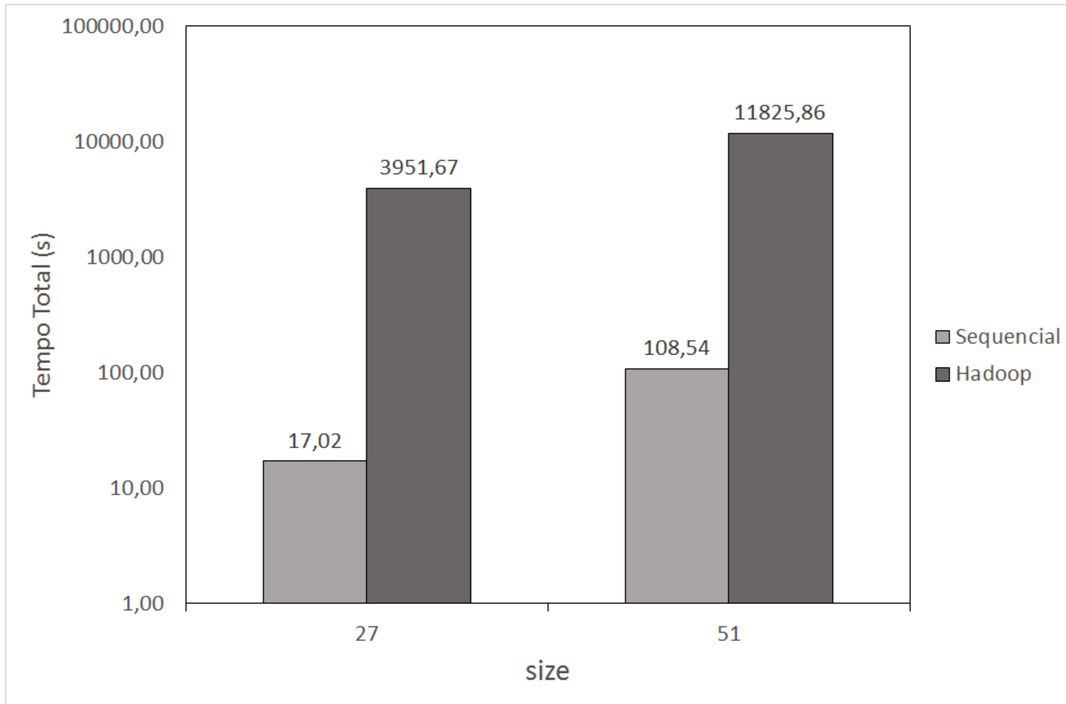

Figura 5.12: Comparação dos tempos totais para a execução do código sequencial e do Quandoop (escala logarítmica) em cenários onde não ocorrem "estouro" na memória RAM utilizando o código sequencial.

5.4 Resultados: Grover

O algoritmo de Grover [31] é um algoritmo quântico de busca cujo objetivo é encontrar um elemento marcado em uma dada lista de elementos. Esse algoritmo possui um ganho quadrático de tempo quando comparado ao melhor algoritmo clássico para resolver o mesmo problema.

Em Lara [32] é descrito uma técnica onde é possível gerar cada elemento de uma matriz dinamicamente, ou seja, sem a necessidade de armazená-lo. Com isso é possível realizar operações utilizando matrizes cujo o tamanho excederia a quantidade de memória RAM presente na maioria dos computadores. O autor do trabalho mencionado escreveu o algoritmo de Grover utilizando essa técnica e a linguagem de programação Neblina [33]. Essa linguagem de programação permite utilizar de forma transparente placas gráficas por meio de OpenCL [34] (do inglês *Open Computing Language*) para realizar esses cálculos de forma paralela.

Se a matriz de evolução utilizada na simulação do algoritmo de Grover não fosse gerada dinamicamente e fosse armazenada, para o valor de $n = 20$ seriam necessários em torno de 8 TB para armazenar essa matriz. Mais detalhes sobre essa técnica e sobre a simulação do algoritmo de Grover utilizando ela podem ser vistos no trabalho realizado por Lara. A matriz a seguir é um exemplo dessa matriz para o valor de $n = 2$.

$$\begin{pmatrix} \frac{-1}{2} & \frac{1}{2} & \frac{-1}{2} & \frac{1}{2} \\ \frac{1}{2} & \frac{-1}{2} & \frac{-1}{2} & \frac{1}{2} \\ \frac{1}{2} & \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & \frac{1}{2} & \frac{-1}{2} & \frac{-1}{2} \end{pmatrix}.$$

Escrevemos um código para ser executado no Hadoop que utiliza a técnica de gerar um elemento da matriz dinamicamente, para assim compararmos o desempenho com a solução que utiliza Neblina. Escrevemos também um código sequencial em Python para ser executado utilizando o notebook citado neste capítulo. O código para Neblina foi executado utilizando a placa gráfica da Workstation HP.

Dividimos o código[7] para ser executado no Hadoop em duas partes: um código que executa a multiplicação de matrizes com cada elemento sendo gerado dinamicamente (*Grover.java*) e um código que realiza uma chamada ao código anterior para cada passo da simulação (*GS.java*).

Os resultados obtidos foram gerados a partir da média aritmética dos tempos de três execuções para cada valor da variável n (utilizada para definir o tamanho da lista), tanto para o código sequencial quanto para o código que utiliza o Hadoop e o que utiliza Neblina.

A Tabela 5.11 possui os tempos de execução e o *speedup* com relação a implementação sequencial em Python do algoritmo de Grover com o valor de entrada igual a 15.

Tabela 5.11: Comparação da execução do algoritmo de Grover com n=15.

Implementação	Tempo de Execução (segundos)	*Speedup*
Sequencial	308505.89	1x
Hadoop	8204.86	37.6x
Neblina	258.64	1192.82x

Observando a Tabela 5.11 podemos perceber que houve um bom ganho de desempenho graças ao paralelismo proporcionado pelo Hadoop, porém o ganho ao utilizar Neblina foi muito superior, mesmo utilizando uma placa gráfica modesta.

[7] `https://github.com/david-ufrj/qw-hadoop/tree/master/GS`

Capítulo 6

Conclusão

Os avanços na área de computação quântica aumentam cada vez mais a necessidade de ferramentas que permitam aos pesquisadores realizarem seus experimentos. Como dito no Capítulo 3, a simulação é uma das formas que os pesquisadores possuem para executar seus experimentos mesmo não possuindo computadores quânticos. O objetivo deste trabalho foi desenvolver ferramentas que permitissem, através do uso do Apache Hadoop, a realização de simulações que necessitassem de uma grande quantidade de memória RAM, não presente em computadores convencionais. Inicialmente implementamos as operações necessárias utilizando o modelo de programação Map/Reduce, para podermos resolvê-las utilizando as vantagens proporcionadas pelo Hadoop, como por exemplo, o paralelismo e o armazenamento dos dados de entrada em disco. Uma vez que essas operações estavam prontas, implementamos códigos que realizam chamadas a essas operações na sequência desejada, funcionando como um *script*, retornando no fim de sua execução o resultado para o experimento proposto.

Para avaliarmos a nossa proposta, escolhemos um problema presente na área da computação quântica que necessitasse processar uma grande quantidade de dados para obter o resultado: a simulação de caminhadas quânticas com dois caminhantes numa malha bidimensional. Nessa simulação a quantidade de dados possui um crescimento quártico em relação ao tamanho da malha. Após fazermos alguns ajustes nas configurações do Hadoop para aumentar a eficiência com que ele iria utilizar o nosso cluster, começamos a execução dos experimentos. A primeira coisa que podemos concluir ao analisar as tabelas e gráficos presentes na Seção 5.2 é que para os casos onde a quantidade necessária de dados para realizar a simulação podem ser armazenados na memória RAM, sem ocorrer "estouro" na mesma, é melhor utilizar uma solução com memória compartilhada, pois a nossa proposta foi menos eficiente que as abordagens convencionais. Porém em casos onde a quantidade de dados é grande demais para serem armazenados na memória RAM, a nossa abordagem que utiliza o Hadoop teve desempenho de acordo com o esperado, nos permitindo

executar a simulação para casos onde nem mesmo uma *workstation* com 28 GB de RAM poderia realizar a simulação.

A análise dos gráficos de *speedup* e eficiência nos permitiu tirar as seguintes conclusões:

- Com um valor maior para a variável *size*, utilizada para definir o tamanho da malha, que implica numa maior quantidade de dados, obtivemos melhores valores de *speedup*;
- Um valor maior para a variável *size* também nos proporcionou uma maior eficiência durante a execução da simulação;
- Para ambos os valores utilizados para a variável *size*, o *speedup* piorou utilizando 31 nós, atingindo seu maior valor utilizando 16 nós.

Apesar dos valores obtidos utilizando 31 nós para os valores escolhidos para a variável *size* terem sido piores que os valores obtidos utilizando 16 nós, se tivéssemos utilizado um valor maior para essa variável poderíamos ter obtido um melhor resultado utilizando 31 nós, uma vez que o gasto extra com as estruturas necessárias para realizar o paralelismo, utilizadas internamente pelo Hadoop, teria sido compensada pelo ganho de paralelismo adicionado com essa quantidade de nós. Testes de *speedup* utilizando valores maiores que 42 para a variável *size* não foram realizados devido o tempo na fila de utilização do cluster não ser o suficiente para executar o teste utilizando apenas um nó.

Ao analisarmos as tabelas e gráficos referentes a Seção 5.3, do Quandoop, podemos inferir as mesmas conclusões obtidas com o código do QW, já que o código sequencial executou mais rápido que o Quandoop e que o nosso simulador nos permitiu executar a simulação para casos onde computadores convencionais não conseguiriam executar o código sequencial.

O último experimento que executamos foi uma avaliação da execução do algoritmo de Grover com seus elementos sendo gerados dinamicamente e comparando com uma solução existente que utiliza a linguagem de programação Neblina. Apesar de termos obtido um bom ganho com o paralelismo através do Hadoop, a solução utilizando Neblina se mostrou bem mais eficiente para resolver problemas que possam ter seus dados gerados de forma dinâmica, ou seja, sem a necessidade de armazená-los.

6.1 Contribuições

Esta dissertação apresentou uma solução utilizando o modelo de programação Map/Reduce para executar experimentos numéricos onde é necessário o proces-

samento de uma grande quantidade de dados. Essa solução foi implementada e avaliada neste trabalho, e está disponível para *download* nos *links* presentes nesta dissertação.

Além da solução, alcançada utilizando as operações desenvolvidas para esse trabalho, também implementamos um simulador genérico, o Quandoop, que realiza uma simulação caracterizada por uma sequência de operações de multiplicação de matrizes, recorrentemente utilizada na área de computação quântica.

6.2 Trabalhos Futuros

Os resultados obtidos nos mostraram que com a nossa abordagem podemos obter soluções para problemas onde a grande quantidade de dados presentes nele não nos permitiria resolvê-los utilizando abordagens com memória compartilhada em computadores convencionais. Portanto, como trabalhos futuros, poderia ser realizada uma pesquisa em busca por problemas que enfrentem essa limitação, e assim, implementar novas operações que utilizem o Hadoop, agregando-as ao conjunto desenvolvido neste trabalho, no intuito de ampliar a gama de problemas que possam ser resolvidos utilizando a nossa abordagem.

Propomos também que seja implementada uma versão híbrida do Quandoop, que realize os cálculos de forma sequencial para casos que utilizem poucos dados no processo de obtenção da solução, e utilize o Hadoop em casos maiores.

O Apache Spark[1] é um arcabouço para ser usado em clusters, que proporciona computação de alto desempenho de forma similar ao Apache Hadoop, sendo uma alternativa para o mesmo, também oferecendo escalabilidade e tolerância à falhas, além de suportar o modelo de programação Map/Reduce. Outra característica do Spark que vale a pena ressaltar é o suporte à linguagem de programação Python, que nos últimos anos vem sendo bastante usada no meio científico. Devido as otimizações presentes no Spark, ele pode apresentar um desempenho até dez vezes superior ao Hadoop [35]. Um dos motivos dessa diferença de desempenho ocorre devido ao custo pago pelo recarregamento dos dados em cada iteração, quando usamos o Hadoop. O Spark possui um *cache* global que melhora o desempenho, reduzindo o tempo de acesso aos dados [36]. Assim, como trabalho futuro, também propomos a migração das operações implementadas neste trabalho para o Spark. Como ele também suporta o modelo Map/Reduce, será possível reimplementar essas operações sem alterar muito o algoritmo e a lógica de programação. Dessa forma poderíamos aproveitar as otimizações presentes no Spark para reduzir o tempo de execução dos cálculos.

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

Propomos ainda a implementação de um módulo para integrar o Quandoop com o HiPerWalk.

Referências Bibliográficas

[1] NIELSEN, M. A., CHUANG, I. L. *Quantum Computation And Quantum Information*. 10th anniversary ed. New York, Cambridge University Press, 2010.

[2] LEE, K.-H., LEE, Y.-J., CHOI, H., et al. "Parallel data processing with MapReduce: a survey", *AcM sIGMoD Record*, v. 40, n. 4, pp. 11–20, Dec 2011.

[3] KAYE, P., LAFLAMME, R., MOSCA, M. *An Introduction to Quantum Computing*. Oxford, Oxford University Press, 2007.

[4] PORTUGAL, R. *Quantum Walks and Search Algorithms*. New York, Springer Science and Business Media, 2013.

[5] PORTUGAL, R., LAVOR, C. C., CARVALHO, L. M., et al. "Uma Introdução à Computação Quântica". In: *Sociedade Brasileira de Matemática Aplicada e Computacional*, São Carlos - SP, Brasil, 2004.

[6] SHOR, P. W. "Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer", *SIAM journal on computing*, v. 26, n. 5, pp. 1484–1509, 1997.

[7] MOTWANI, R., RAGHAVAN, P. *Randomized algorithms*. New York, Cambridge University Press, 1995.

[8] MARQUEZINO, F. L. *Análise, simulações e aplicações algorítmicas de caminhadas quânticas*. Tese de D.Sc., Laboratório Nacional de Computação Científica, Petrópolis, RJ, Brasil, 2010.

[9] "Projeto Apache Hadoop". `https://hadoop.apache.org/`, . Acessado: 17-03-2015.

[10] WHITE, T. *Hadoop: The definitive guide*. 3 ed. Massachusetts, O'Reilly Media, Inc., 2012.

[11] DEAN, J., GHEMAWAT, S. "MapReduce: simplified data processing on large clusters", *Communications of the ACM*, v. 51, n. 1, pp. 107–113, Jan 2008.

[12] CHAPMAN, B., JOST, G., PAS, R. V. D. *Using OpenMP: portable shared memory parallel programming*. Cambridge, MA, MIT press, 2008.

[13] GROPP, W., LUSK, E., SKJELLUM, A. *Using MPI: portable parallel programming with the message-passing interface*. 1 ed. Cambridge, MA, MIT press, 1999.

[14] "Hadoop Streaming". `http://hadoop.apache.org/docs/r1.2.1/streaming.html`, . Acessado: 18-03-2015.

[15] DING, M., ZHENG, L., LU, Y., et al. "More convenient more overhead: the performance evaluation of Hadoop streaming". In: *Proceedings of the 2011 ACM Symposium on Research in Applied Computation*, pp. 307–313, New York, Mar 2011.

[16] SHVACHKO, K., KUANG, H., RADIA, S., et al. "The hadoop distributed file system". In: *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pp. 1–10, Washington, May 2010.

[17] GHEMAWAT, S., GOBIOFF, H., LEUNG, S.-T. "The Google file system". In: *ACM SIGOPS operating systems review*, v. 37, pp. 29–43, Oct 2003.

[18] NIELSEN, M. A., CHUANG, I. L. "Quantum algorithms". In: *Quantum Computation And Quantum Information*, 10th anniversary ed., pp. 172–173, New York, Cambridge University Press, 2010.

[19] MARQUEZINO, F. L., PORTUGAL, R. "The QWalk simulator of quantum walks", *Computer Physics Communications*, v. 179, n. 5, pp. 359–369, 2008.

[20] "HiPerWalk, High-Performance Quantum Walk Simulator". `http://qubit.lncc.br/qwalk/hiperwalk.pdf`. Users Manual, 2014.

[21] VIAMONTES, G. F., MARKOV, I. L., HAYES, J. P. *Quantum circuit simulation*. Springer Science and Business Media, 2009.

[22] JOHANSSON, J. R., NATION, P. D., NORI, F. "QuTiP: An open-source Python framework for the dynamics of open quantum systems", *Computer Physics Communications*, v. 183, n. 8, pp. 1760–1772, Feb 2012.

[23] KRAUZOWICZ, L., SZOSTEK, K., DWORNIK, M., et al. "Numerical calculations for geophysics inversion problem using apache hadoop technology". In: *Computer Networks*, v. 291, *Communications in Computer and Information Science*, Springer, pp. 440–447, 2012.

[24] YAO, Z., VARGHESE, B., RAU-CHAPLIN, A. "High performance risk aggregation: addressing the data processing challenge the hadoop mapreduce way". In: *Proceedings of the 4th ACM workshop on Scientific cloud computing*, pp. 53–60, New York, Jun 2013.

[25] "Code Conventions for the Java TM Programming Language". `http://www.oracle.com/technetwork/java/codeconvtoc-136057.html`. Acessado: 06-08-2015.

[26] AHLBRECHT, A., ALBERTI, A., MESCHEDE, D., et al. "Molecular binding in interacting quantum walks", *New Journal of Physics*, v. 14, n. 7, pp. 073050, Jul 2012.

[27] LESKOVEC, J., RAJARAMAN, A., ULLMAN, J. D. *Mining of massive datasets*. Cambridge, Cambridge University Press, 2014.

[28] "About Rocks Cluster". `http://central6.rocksclusters.org/roll-documentation/base/6.1.1/roll-base-usersguide.pdf`. Base Users Guide, 2014.

[29] MAGRO, W., PETERSEN, P., SHAH, S. "Hyper-Threading Technology: Impact on Compute-Intensive Workloads", *Intel Technology Journal*, v. 6, n. 1, pp. 1, Feb 2002.

[30] TANNIR, K. *Optimizing Hadoop for MapReduce*. Birmingham, Packt Publishing Ltd, 2014.

[31] GROVER, L. K. "A fast quantum mechanical algorithm for database search". In: *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, pp. 212–219, New York, 1996.

[32] DA SILVA LARA, P. C. *Otimização de Funções Contínuas Usando Algoritmos Quânticos*. Tese de D.Sc., Laboratório Nacional de Computação Científica, Petrópolis, RJ, Brasil, 2015.

[33] "Neblina Programming Language". `http://www.lncc.br/~pcslara/neblina/`. Acessado: 18-09-2015.

[34] "The open standard for parallel programming of heterogeneous systems". `https://www.khronos.org/opencl/`. Acessado: 18-09-2015.

[35] ZAHARIA, M., CHOWDHURY, M., FRANKLIN, M. J., et al. "Spark: Cluster Computing with Working Sets". In: *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, v. 10, p. 10, 2010.

[36] GU, L., LI, H. "Memory or time: Performance Evaluation for Iterative Operation on Hadoop and Spark". In: *High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing (HPCC_EUC), 2013 IEEE 10th International Conference on*, pp. 721–727. IEEE, 2013.