UNIVERSIDADE DO VALE DO RIO DOS SINOS
UNIDADE ACADÊMICA DE GRADUAÇÃO
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

JORGE MATHEUS GOMES VIEGAS

ANÁLISE DE DESEMPENHO DE LINGUAGENS DE PROGRAMAÇÃO E APIS QUÂNTICAS

São Leopoldo
2021

JORGE MATHEUS GOMES VIEGAS

ANÁLISE DE DESEMPENHO DE LINGUAGENS DE PROGRAMAÇÃO E APIS QUÂNTICAS

Artigo apresentado como requisito parcial para obtenção do título de Bacharel em Ciência da Computação, pelo Curso de Ciência da Computação da Universidade do Vale do Rio dos Sinos (UNISINOS)

Orientador(a): Msc. Gilberto Irajá Müller

São Leopoldo
2021

ANÁLISE DE DESEMPENHO DE LINGUAGENS DE PROGRAMAÇÃO E APIS QUÂNTICAS

Jorge Matheus Gomes Viegas[1]
Gilberto Irajá Muller[2]

CONTEXTO: A computação quântica é a área de pesquisa que estuda a utilização das propriedades da mecânica quântica para representação e processamento de informações, expandindo a capacidade do modelo computacional clássico. O poder da computação quântica pode ser observado através de algoritmos quânticos, que destacam problemas computacionais em que o modelo quântico é superior ao modelo clássico. **PROBLEMA:** Nos últimos anos, diversas ferramentas de software como linguagens de programação, APIs e simuladores foram desenvolvidas para auxiliar o desenvolvimento de novos algoritmos quânticos, tornando difícil a escolha de estudantes e cientistas sobre o software quântico a ser utilizado. **SOLUÇÃO:** O presente estudo apresenta uma análise de desempenho de linguagens de programação e APIs quânticas utilizando algoritmos quânticos presentes na literatura. **MÉTODO PROPOSTO:** Para atingir os objetivos deste estudo, foi realizada uma pesquisa experimental utilizando a técnica *stopwatch* para a coleta de dados através de uma abordagem quantitativa. **CONCLUSÃO:** Por fim, apresentam-se a análise e discussão de resultados, considerações finais e propostas de trabalhos futuros.

Palavras-chave: Computação quântica. Linguagens de Programação Quânticas. Algoritmos Quânticos.

1 INTRODUÇÃO

A computação quântica é uma área de pesquisa transdisciplinar que estuda a aplicação de propriedades da mecânica quântica em sistemas computacionais. Nielsen e Chuang (2010) definem a computação quântica como o estudo das tarefas de processamento de informação que podem ser alcançadas utilizando sistemas quânticos. Através da utilização de propriedades da mecânica quântica, como emaranhamento e superposição, computadores quânticos são capazes de realizar tarefas específicas exponencialmente mais rápidas do que os computadores clássicos. (ZHAO, 2020). Para Svore e Troyer (2016), entender as aplicações habilitadas pela utilização da computação quântica pode alterar o cenário econômico, industrial e acadêmico contemporâneo. Pesquisadores de laboratórios da indústria e do governo estão explorando as diversas aplicações da computação quântica. IBM, Google e Microsoft são exemplos de empresas que investem em centros de pesquisa de computação quântica. Em 2021, o *Latin America Quantum Computer Center* (LAQCC), primeiro centro de pesquisa brasileiro dedicado à Computação Quântica, foi inaugurado em Salvador/Brasil. A importância do avanço da computação quântica está diretamente relacionada com limites no crescimento do modelo computacional clássico.

[1]Graduando em Ciência da Computação pela Unisinos. Email: jorgegv@edu.unisinos.br
[2]Prof. Ms. Gilberto Irajá Müller, Mestre em Computação Aplicada com atuação na área de Análise e Desenvolvimento de Aplicações para Dispositivos Móveis. Email: gimuller@unisinos.br

O modelo clássico baseia-se na Arquitetura de Von Neumann, que tem como principal característica o armazenamento de dados e instruções para o processador em memória. Na camada física, a memória é composta por componentes como transistores e capacitores, que utilizam cargas elétricas para representar a menor unidade de informação do modelo computacional clássico: o *bit*. A quantidade de transistores e capacitores contidos nos *chips* é um fator determinante para o crescimento do poder computacional do modelo clássico. Em 1965, Gordon E. Moore observou que, através das técnicas de redução de tamanho utilizadas pelas empresas fabricantes de semicondutores, o número de componentes por *chip* vinha dobrando a cada ano, com tendência em continuar na mesma crescente por décadas. Essa observação ficou conhecida como "Lei de Moore". (MOORE, 1965). Porém, segundo o próprio Moore (2003), uma barreira fundamental para este crescimento deve ser confrontada nas próximas décadas: o tamanho dos componentes está se aproximando de dimensões atômicas. Segundo Williams (2010), nestas dimensões, as leis físicas apropriadas são as da mecânica quântica e a teoria matemática que sustenta a ciência da computação moderna deixa de ser válida.

Computadores quânticos são compostos por diversos elementos que, funcionalmente, remetem aos computadores clássicos. Registradores, portas e barramentos são exemplos de componentes que estão presentes em ambos os modelos de computação. A diferença fundamental reside na camada física, onde os componentes do computador clássico têm seu funcionamento baseado nas leis da física clássica, enquanto os componentes do computador quântico utilizam as propriedades da mecânica quântica para executar operações. (GYONGYOSI; IMRE, 2019).

Os algoritmos quânticos, que são representações de circuitos quânticos, se destacam por demonstrarem o poder do modelo computacional quântico em relação ao modelo clássico. O algoritmo de Shor (fatoração de números inteiros), por exemplo, pode ser resolvido em tempo polinomial em uma máquina universal quântica, o que não se provou ser possível no modelo computacional clássico. Em 2019, pesquisadores da Google demonstraram que, em um processador quântico, uma determinada tarefa computacional pode ser resolvida em 200 segundos, enquanto em um computador clássico a mesma tarefa seria resolvida em aproximadamente 10.000 anos. Este estudo sugeriu que o modelo computacional quântico viola a tese estendida de Church-Turing, que afirma que qualquer modelo computacional pode ser simulado eficientemente por uma Máquina de Turing. (ARUTE et al., 2019). Os algoritmos quânticos são importantes para entender a capacidade computacional do modelo quântico e, por esse motivo, a busca por novos algoritmos que possam comprovar a superioridade do computador quântico deve continuar.

Para Zhao (2020), existe uma urgente necessidade de construção de uma comunidade de engenharia de software quântico, com foco em idealizar métodos, ferramentas e processos para o desenvolvimento de sistemas de software quântico. Para Svore e Troyer (2016), o desenvolvimento e estudo de bibliotecas e APIs quânticas irá acelerar o desenvolvimento de novos algoritmos quânticos. Entender as principais funcionalidades e o desempenho destas bibliotecas, linguagens e APIs seria uma contribuição na construção de uma arquitetura de software

quântico.

Recentemente, houve uma explosão na quantidade de software quântico disponível para linguagens de programação clássicas, incluindo simuladores de computação quântica. A pesquisa na área das linguagens de programação quânticas está em evolução e espera-se que diversas linguagens sejam desenvolvidas antes mesmo da disseminação do computador quântico. (GARHWAL; GHORANI; AHMAD, 2019). Apesar do reflexo positivo desse crescimento, aumenta-se a dificuldade de estudantes e pesquisadores decidirem qual linguagem utilizar, se perdendo na documentação ou simplesmente não sabendo por onde começar. (LAROSE, 2019).

Com o avanço na arquitetura de software quântico, estudar as funcionalidades e desempenho de diferentes linguagens de programação e APIs quânticas auxiliaria na introdução do modelo computacional quântico como tema de pesquisa interdisciplinar em Universidades, nas áreas de Ciência da Computação e Física.

Com base no contexto apresentado, esse estudo pode ser representado pela seguinte questão de pesquisa: **Qual o panorama das linguagens de programação e APIs quânticas ao considerar o desempenho na implementação de algoritmos quânticos recorrentes na literatura?** Para responder essa questão de pesquisa, os seguintes objetivos foram realizados:

a) implementação de algoritmos quânticos constantes na literatura de acordo com linguagens de programação e APIs quânticas;

b) análise quantitativa de desempenho de diferentes linguagens de programação e APIs quânticas;

c) proposição de um repositório no *Github*[3] com implementações de algoritmos quânticos constantes na literatura.

Além da seção de Introdução, o presente artigo está dividido em mais cinco seções. A Seção 2 aborda a fundamentação teórica sobre Computação Quântica e Arquitetura de Software Quântico, destacando os algoritmos quânticos abordados neste estudo. A Seção 3 apresenta uma análise de trabalhos relacionados e, ao final, uma comparação de linguagens de programação abordadas nesses trabalhos. Nesta seção são apresentados os critérios de seleção utilizados na escolha das linguagens de programação e APIs consideradas na análise de desempenho. A Seção 4 apresenta a metodologia utilizada para atingir os objetivos propostos. A Seção 5 aborda a apresentação, análise e discussão dos resultados. Por fim, a Seção 6 apresenta as conclusões e propostas de trabalhos futuros.

[3]Acesse o repositório em: https://github.com/jorgeviegas/quantumalgorithms

2 FUNDAMENTAÇÃO TEÓRICA

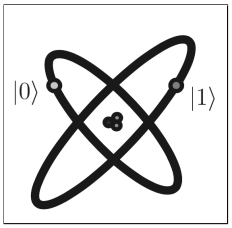
2.1 Computação Quântica

Em 1982, Richard Philips Feynman sugeriu que um computador que utilizasse propriedades da mecânica quântica poderia habilitar simulações eficientes de sistemas físicos e quânticos, pois seria impossível para um computador universal clássico representar resultados de sistemas quânticos. (FEYNMAN, 1982). Segundo Arute et al. (2019), as observações de Feymann impuseram o início pela busca por problemas que são difíceis para os computadores clássicos e fáceis para computadores quânticos, marcando o início dos estudos sobre a computação quântica.

2.1.1 Qubit

O *Quantum Bit*, ou *qubit*, é a unidade básica de informação utilizada na computação quântica. Análogo ao *bit* da computação clássica, os *qubits* são sistemas compostos de componentes físicos que utilizam estados para representação de informação. (SVORE; TROYER, 2016). Diversos sistemas quânticos, como, por exemplo, polarizações de um fóton e estados de órbita de um elétron em um átomo podem ser utilizados para indicar os estados do *qubit*. A Figura 1 representa os estados de órbita de elétrons em um átomo.

Figura 1 – Estados de órbita de elétrons em um átomo



Fonte: Nielsen e Chuang (2010)

Os dois estados básicos de um *qubit* são representados por $|0\rangle$ e $|1\rangle$, também denominados estados básicos computacionais. Os estados são representados utilizando a notação $|\rangle$, também conhecida como notação de Dirac, ou notação *bra-ket*, utilizada na representação de estados de sistemas quânticos. (DIRAC, 1939). Em um átomo, o estado de órbita de um elétron varia de acordo com seu estado energético. Quando o elétron está no seu nível mínimo de energia, chamado de estado *ground*, considera-se que o *qubit* está no estado $|0\rangle$. Quando ocorre um aumento de energia no átomo, o elétron entra no estado *excited* e muda de órbita, alterando o estado do *qubit* para $|1\rangle$. Na Equação (1) estão representados os estados básicos do *qubit* na sua forma vetorial.

$$|0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \quad |1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \tag{1}$$

Quando mede-se um *qubit*, pode-se obter um de seus possíveis estados: $|0\rangle$ ou $|1\rangle$, probabilisticamente. O estado de um *qubit* pode ser representado por uma combinação linear dos dois estados básicos, notados na Equação 2. (NIELSEN; CHUANG, 2010).

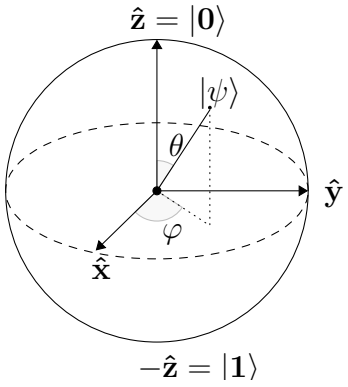
$$|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle \tag{2}$$

Na Equação 2, ψ representa o estado do *qubit*, onde α e β são as amplitudes probabilísticas de cada estado, caracterizados como números complexos. Quando mede-se um *qubit*, pode-se obter o resultado $|0\rangle$ com probabilidade $|\alpha|^2$ ou o estado $|1\rangle$ com probabilidade $|\beta|^2$. Como as probabilidades devem somar um, pode-se observar que $|\alpha|^2 + |\beta|^2 = 1$, ou $|\psi\rangle = 1$. Considerando que α e β são números complexos, pode-se reescrever a Equação 2 em sua forma polar:

$$|\psi\rangle = \cos\frac{\theta}{2}\,|0\rangle + e^{i\varphi}\sin\frac{\theta}{2}\,|1\rangle \tag{3}$$

A Equação 3 define geometricamente o estado de um *qubit*, podendo ser representado por um ponto em uma esfera tridimensional. A Figura 2 representa o estado de um *qubit* na Esfera de Bloch, comumente utilizada na representação de *qubits*.

Figura 2 – Esfera de Bloch



Fonte: Elaborada pelo autor.

Em uma comparação, se aplicados à esfera, as duas únicas posições possíveis para os *bits* clássicos seriam os pólos da esfera, representados por $\hat{\mathbf{z}}$ e $-\hat{\mathbf{z}}$, enquanto o restante da superfície da esfera demonstra o espaço de possíveis estados de um *qubit*. Ao analisar a esfera de Bloch, quando o vetor estado ψ está mais próximo do pólo $\hat{\mathbf{z}}$, a amplitude associada ao estado $|0\rangle$ é maior do que a amplitude associada ao estado $|1\rangle$, indicando que a probabilidade de uma medição ter como resultado o estado $|0\rangle$ é maior. Quando mais próximo do pólo $-\hat{\mathbf{z}}$, a probabilidade do estado $|1\rangle$ se torna maior. Porém, quando o vetor estado ψ está exatamente entre os dois pólos, paralelo aos eixos $\hat{x}$ e $\hat{y}$, significa que simultaneamente o *qubit* está em dois estados, resultando em uma superposição.

A superposição quântica é uma propriedade que atesta que uma partícula pode existir em

uma variedade de diferentes estados ao mesmo tempo. Porém, devido a restrições da mecânica quântica, apenas um de seus estados pode ser observado em um instante específico. Nos sistemas físicos quânticos, um *qubit* pode estar em superposição quando, por exemplo, um elétron está entre os extremos da órbita de um átomo. Quando medido, o *qubit* colapsa para um de seus estados básicos ($|0\rangle$ ou $|1\rangle$), probabilisticamente. (SANGHVI; SHAH; VARADAN, 2014). Matematicamente, a superposição pode ser representada como uma combinação linear dos estados básicos, como representa a Equação 4.

$$|\psi\rangle = \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle \tag{4}$$

O estado representado pela Equação 4 significa que, quando medido, o *qubit* resultará em 0 em 50% das vezes e 1 em 50% das vezes. Este estado também é representado como $|+\rangle$. (NIELSEN; CHUANG, 2010).

No modelo computacional clássico, um par de *bits* pode representar quatro estados diferentes: 00, 01, 10 e 11. Analogamente, no modelo quântico, um sistema composto de dois *qubits* pode representar quatro estados computacionais básicos, representados por $|00\rangle$, $|01\rangle$, $|10\rangle$ e $|11\rangle$. Assim, como os estados de um único *qubit*, os estados de sistemas de múltiplos *qubits* também podem estar em superposição. A Equação 5 representa a combinação linear dos possíveis estados de um sistema de dois *qubits*. (NIELSEN; CHUANG, 2010).

$$|\psi\rangle = \alpha_{00}|00\rangle + \alpha_{01}|01\rangle + \alpha_{10}|10\rangle + \alpha_{11}|11\rangle \tag{5}$$

2.1.2 Algoritmos Quânticos

Para Svore e Troyer (2016), o grande poder do computador quântico está na capacidade de aplicar as propriedades da mecânica quântica para solucionar problemas computacionais clássicos, como, por exemplo, criptografia, simulação de sistemas físicos e solução de sistemas lineares. Os algoritmos quânticos são algoritmos que se utilizam das propriedades quânticas como o emaranhamento e superposição através de uma combinação de operações e transformações em *qubits*, representadas por um circuito quântico para resolver um problema computacional. Um quadro com as principais operações (portas) quânticas pode ser visto no Apêndice A.

Nielsen e Chuang (2010) afirmam que a essência do *design* de muitos algoritmos quânticos reside na escolha inteligente de funções e transformações dos estados quânticos, habilitando a obtenção de informações que não poderiam ser alcançadas em um computador clássico. A escolha das funções e transformações descrita por Nielsen e Chuang (2010) é parte fundamental dos algoritmos de Deutsch e Deutsch-Josza, que destacam uma das principais vantagens dos algoritmos quânticos em relação aos algoritmos clássicos: o paralelismo quântico.

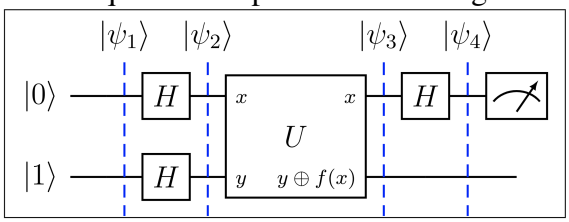
O paralelismo quântico é uma propriedade que permite que computadores quânticos avaliem uma função $f(x)$ para muitos valores diferentes de x simultaneamente. Ao colocar os *qubits* em um estado de superposição utilizando a porta de Hadamard, quando aplicada uma função $f(x)$,

pode-se obter resultados de múltiplas diferentes entradas em apenas uma avaliação da função. Um dos primeiros exemplos da utilização do paralelismo quântico é o algoritmo introduzido por David Elieser Deutsch em 1985. (DEUTSCH, 1985).

O Algoritmo de Deutsch ficou conhecido por ser o primeiro exemplo da vantagem do computador quântico em relação ao computador clássico para resolver problemas específicos. Para definir o problema, considera-se uma função booleana $f(x)$ que recebe um *bit* como entrada e retorna sempre o valor de um *bit* como saída. A função pode ser definida como $f(x) : \{0, 1\} \rightarrow \{0, 1\}$, garantida como sendo balanceada ou constante. Uma função constante sempre retorna o mesmo *bit*, independente da entrada. Uma função balanceada varia sua saída, retornando o valor 0 em exatamente metade das saídas e o valor 1 na outra metade. O problema se dá em descobrir se a função $f(x)$ é balanceada ou constante no menor número de execuções da função $f(x)$.

Na computação clássica, no pior caso, necessitam-se $2^n/2 + 1$ execuções da função $f(x)$ para descobrir se a função é balanceada ou constante. Em um computador quântico, o problema de Deutsch pode ser resolvido em apenas uma execução da função. A Figura 3 representa o circuito quântico que implementa o algoritmo de Deutsch.

Figura 3 – Circuito quântico implementando o algoritmo de Deutsch



Fonte: Elaborada pelo autor.

O circuito representado na Figura 3 tem como entrada dois *qubits*, respectivamente nos estados $|0\rangle$ e $|1\rangle$, formando um sistema de estado $|\psi_1\rangle = |01\rangle$. A primeira operação do circuito é aplicar a porta Hadamard (H) ao sistema de dois *qubits*, obtendo um estado de superposição $|\psi_2\rangle$. A próxima operação é executar a porta U, que representa a função booleana $f(x)$ em sua forma quântica. A forma quântica da função recebe dois *qubits* como parâmetro e retorna dois *qubits* como resultado. Essa porta quântica, através da operação XOR $(\oplus)$, inverte o valor do segundo *qubit* se o resultado de f (quando recebe o valor do primeiro *qubit* como parâmetro) seja $|1\rangle$, e não altera o valor do segundo *qubit* se o resultado de f atuando no valor do primeiro *qubit* seja $|0\rangle$. Após executar a porta U, o estado do sistema quântico é representado por $|\psi_3\rangle$.

O passo seguinte é aplicar novamente a porta Hadamard (H) no primeiro *qubit*, o que faz com que o *qubit* retorne a um dos dois estados básicos computacionais. Neste momento, o estado do sistema é representado por $|\psi_4\rangle$. O último passo do algoritmo é medir o primeiro *qubit*, fazendo-o colapsar para $|0\rangle$ ou $|1\rangle$. Ao medir o primeiro *qubit*, pode-se determinar uma propriedade global da função $f(x)$ com apenas uma execução da porta U. Se o *qubit* colapsar para $|0\rangle$, a função é constante; caso contrário, é balanceada.

Os algoritmos quânticos abordados neste estudo foram selecionados baseando-se nos seguintes critérios: (a) citados na literatura; (b) considerados educacionais ou introdutórios à computação quântica; e (c) possuam implementação nas linguagens de programação e APIs selecionadas. A partir desses critérios, a seguir, apresentam-se os algoritmos quânticos considerados na análise de desempenho.

2.1.3 Algoritmo de Bernstein-Vazirani

Formulado por Ethan Bernstein e Umesh Vazirani, o algoritmo é uma variação do algoritmo de Deutsch. Dada uma função $f(x) : \{0,1\}^n \rightarrow \{0,1\}$ que tem como propriedade sempre retornar o resultado do produto *bitwise* entre x e uma sequência de bits a, o problema se dá em encontrar a sequência de bits a. A função $f(x)$ pode ser descrita na Equação 6. (CLEVE et al., 1998).

$$f(x) = (a \cdot x) \oplus b \tag{6}$$

A solução clássica para o problema de Bernstein-Vazirani necessita de $\mathcal{O}(n)$ execuções da função $f(x)$, enquanto o algoritmo quântico consegue encontrar a sequência a em apenas $\mathcal{O}(1)$ execução da função $f(x)$.

2.1.4 Algoritmo de Deutsch-Jozsa

O algoritmo de Deutsch-Jozsa é uma generalização do algoritmo de Deutsch para funções booleanas de n *qubits*. As funções, que podem ser definidas como $f(x) : \{0,1\}^n \rightarrow \{0,1\}$, são garantidas como sendo balanceadas ou constantes, assim como no problema original. O circuito que implementa a solução para o problema utiliza $n+1$ *qubits*, sendo n o tamanho em *qubits* do valor de entrada x da função.

Em sua solução clássica, no pior caso, necessitam-se $2^n/2+1$ execuções da função $f(x)$ para descobrir se a função é balanceada ou constante. A solução quântica, assim como no algoritmo de Deutsch, consegue avaliar a propriedade global da função em apenas uma execução ($\mathcal{O}(1)$).

2.1.5 Algoritmo de Grover

Proposto por Lov Krumar Grover em 1996, é um algoritmo de busca quântico. Dada uma lista de n itens não-ordenados, o problema se dá em encontrar um item que satisfaça uma propriedade única dentre os itens da lista. Pode ser definido através de uma função $f(x) : \{0,1\}^n \rightarrow \{0,1\}$, que retorne 1 para apenas um valor de x. O algoritmo busca o valor do parâmetro x que retorne o valor 1. (CLEVE et al., 1998).

Foi descrito por Grover como um algoritmo onde a "mecânica quântica ajuda a encontrar uma agulha em um palheiro". (GROVER, 1997). O algoritmo quântico consegue encontrar o

item em $O(\sqrt{N})$, apresentando uma melhora quadrática em relação ao algoritmo clássico de busca, que necessita de $O(N)$ iterações.

2.1.6 Algoritmo de Shor

O Algoritmo de Shor, dentre os algoritmos quânticos educacionais, é o que mais apresenta possível utilidade, pois propõe uma solução para o problema da fatoração de números inteiros, técnica que pode ser utilizada para quebrar chaves de criptografia RSA.

A fatoração de números inteiros pode ser reduzida em tempo polinomial ao problema de encontrar o período de uma função $f(x) = a^r \mod N$, onde N é o inteiro a ser fatorado. Dados dois números inteiros a e N, em que o máximo divisor comum entre eles seja 1, o problema se dá em encontrar o menor número inteiro r que satisfaça a Equação 7. (MOSCA, 2008).

$$a^r \equiv 1 \pmod N \tag{7}$$

O algoritmo, proposto por Peter Williston Shor em 1994, apresenta solução em tempo polinomial, o que ainda não se fez possível em computadores clássicos. (SHOR, 1994).

2.1.7 Algoritmo de Simon

Dada uma função $f(x) : \{0,1\}^n \rightarrow \{0,1\}^m$, em que m seja maior que n e que necessariamente seja uma função ou um-para-um ou dois-para-um, o problema consiste em encontrar uma sequência de *bits* que defina a relação de invariância XOR da função dois-para-um para os dois valores de entrada (x e x') que retornam o mesmo valor de y. A relação de invariância pode ser descrita na Equação 8. (SIMON, 1997).

$$f(x) = f(x') \Leftrightarrow x' = x \oplus s \tag{8}$$

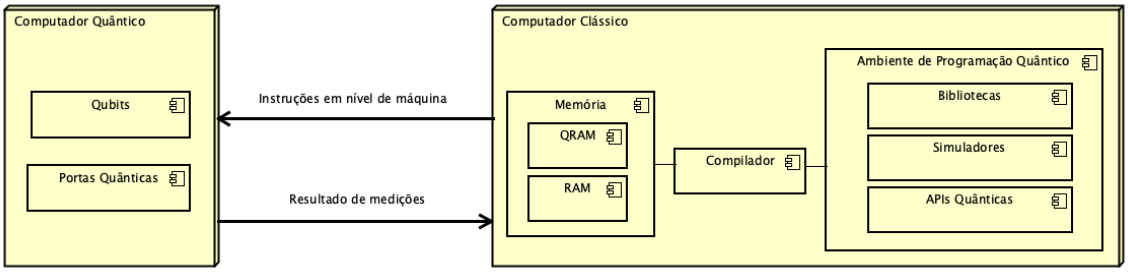
Para encontrar a sequência de *bits* s que satisfaça a equação, a solução clássica para o problema de Simon necessita de $2^{n-1} + 1$ execuções da função f, sendo n o tamanho em *bits* da entrada da função. O algoritmo quântico de Simon é capaz de encontrar a sequência s em apenas $O(n)$ execuções da função f, sendo exponencialmente mais rápido na comparação com sua solução clássica.

2.2 Arquitetura de Software Quântico

Häner et al. (2018) afirmam que o computador quântico será controlado por um computador clássico, atuando como um co-processador, assim como o modelo de processamento da GPU. Computadores quânticos serão encapsulados logicamente em um computador clássico que con-

trola um sistema quântico, assim realizando uma mistura de computação clássica e quântica. Para Svore et al. (2006), é seguro afirmar que computadores clássicos irão monitorar e controlar computadores quânticos através de um canal de comunicação bidirecional. Os modelos de computação quântica são baseados no QRAM, um modelo de arquitetura híbrida para sistemas de computação quânticos proposto por Emanuel Knill em 1996. (KNILL, 1996). A Figura 4 representa a arquitetura do modelo QRAM.

Figura 4 – Arquitetura QRAM



Fonte: Elaborada pelo autor.

A arquitetura QRAM é divida em dois componentes: quântico e clássico. No componente quântico, também chamado de Unidade de Processamento Quântico (QPU), residem os componentes de hardware quânticos como *qubits* e portas lógicas quânticas. No componente clássico, os algoritmos quânticos são transformados em instruções quânticas e armazenados em uma memória compartilhada, juntamente com as instruções clássicas resultantes do processo de compilação. Em tempo de execução, as instruções são enviadas sequencialmente à QPU, que retorna os resultados em bits clássicos. (KNILL, 1996). O modelo pode ser considerado uma extensão das arquiteturas de computadores clássicos que utilizam memórias de acesso randômico (RAM), pois além de manter a capacidade do modelo clássico, adiciona a capacidade de executar um conjunto restrito de operações quânticas na QPU. (BETTELLI; CALARCO; SERAFINI, 2003).

Svore et al. (2006) propõem uma arquitetura de software quântico constituída de camadas, em que um programa quântico escrito em uma linguagem de programação de alto nível é transformado em um algoritmo composto por um conjunto de instruções de baixo nível, que pode ser executado em uma QPU. Nesta arquitetura, que tem como base os conceitos fundamentais da QRAM, diversas ferramentas de software como compiladores, simuladores e linguagens de programação de alto nível são dispostas em camadas. (SVORE et al., 2006).

2.2.1 Linguagens de Programação

Ghezzi e Jazayeri (1997) definem linguagens de programação como notações formais para descrever algoritmos a serem executados em um computador. As linguagens de programação são a maneira com que os humanos comunicam suas ideias com o computador.

Dentro do estudo sobre as características das linguagens de programação, Tucker e Noonan (2009) definem as principais categorias de princípios de projeto de linguagens, dentre elas a sintaxe e a semântica. A sintaxe de uma linguagem de programação se refere à estrutura do texto de um programa, sendo uma descrição precisa de todos os programas gramaticamente corretos. Assim como uma linguagem natural, a sintaxe é descrita por um conjunto de regras que devem ser seguidas por um programa escrito na linguagem. A semântica de uma linguagem de programação é uma definição precisa do significado de qualquer programa que seja correto. O efeito de cada comando sobre o estado de uma computação em um programa é dado pela semântica da linguagem.

Linguagens de programação podem ser classificadas em paradigmas de programação de acordo com suas características, como sintaxe e semântica. Para Tucker e Noonan (2009), um paradigma de programação é um padrão de resolução de problemas que se relaciona a um determinado gênero de programas e linguagens. Um paradigma pode ser visto como um padrão de ideias que são utilizadas para guiar um conjunto de atividades relacionadas. Dentre os principais paradigmas, destacam-se:

a) **Imperativo ou Procedural:** as linguagens de programação do paradigma imperativo têm seu funcionamento baseado na Arquitetura de Von Neumann, em que as instruções e os dados são armazenados em memória. Através da execução sequencial de instruções, os valores armazenados na memória são alterados, representando o estado da execução do programa. São características das linguagens imperativas os comandos de atribuição, estruturas de controle, estruturas de dados e abstração procedural. Linguagens notáveis do paradigma imperativo são o COBOL, a linguagem C e o Fortran;

b) **Orientado a Objetos:** as principais características das linguagens orientadas a objeto são a classificação de tipos abstratos de dados, a herança e a troca de mensagens. Os tipos abstratos de dados, criados para facilitar a organização dos programas, formam a base do modelo orientado a objeto. Neste contexto, os tipos abstratos foram nomeados classes. As instâncias destas classes são chamadas de objetos, o que origina o nome do paradigma. As classes podem encapsular subprogramas chamados métodos, que definem operações específicas em objetos de uma classe. A herança surgiu como uma forma de reutilizar tipos abstratos de dados, uma vez que uma classe pode herdar características e estender suas funcionalidades sem alterar o tipo abstrato original. Algumas das principais linguagens que seguem o paradigma orientado a objeto são Smalktalk, C++, Java e C#. (SEBESTA, 2012);

c) **Funcional:** no paradigma funcional, os programas são modelados como um conjunto de funções matemáticas que são utilizadas para resolver um problema computacional. As funções, componentes centrais destas linguagens, são uma representação matemática que mapeiam um valor de entrada em um valor de saída. Diferentemente dos demais paradigmas, o paradigma funcional não contempla a noção de estado de um programa, pois não

conta com os comandos de atribuição de valores à variáveis em memória. Pelo fato de não contemplar variáveis e atribuições em memória, as repetições no paradigma funcional são obtidas através de recursão. Programas escritos em linguagens de programação funcionais são definições de funções e sua execução consiste em avaliar o resultado das aplicações destas funções. Lisp, Prolog, Scheme, Haskell e ML são alguns exemplos de linguagens funcionais. (SEBESTA, 2012).

2.2.2 Linguagens de Programação Quânticas

Devido à arquitetura híbrida proposta para sistemas de computação quântica, Knill (1996) também propôs estruturas de pseudocódigos quânticos, compostos de instruções que são parte executadas em um computador quântico e parte em um computador clássico. As principais características propostas pelos pseudocódigos no modelo QRAM são a definição de tipos de dados para registradores quânticos e a representação de operações quânticas. Os modelos de pseudocódigo de Knill (1996) influenciaram o estudo e criação de diversas linguagens de programação quânticas. Smith, Curtis e Zeng (2016) definem três categorias de linguagens de programação quânticas:

a) **Linguagens de domínio específico embutidas (ESDL) :** são linguagens com propósitos específicos que são embutidas em linguagens hospedeiras, devendo obedecer as estruturas primárias como condicionais, funções e iterações de sua hospedeira. São criadas para facilitar o nível de abstração das linguagens em determinados domínios, como, por exemplo, a programação quântica. Um exemplo de ESDL é o Quipper, proposto por Green et al. (2013). O Quipper é uma linguagem de programação quântica funcional embutida, composta de uma coleção de tipos de dados, operadores e bibliotecas de funções para a linguagem hospedeira Haskell;

b) **Linguagens de programação quânticas de alto nível:** estas linguagens são análogas às linguagens clássicas, como a imperativa linguagem C e a orientada a objeto Java. Esta categoria de linguagens deve encapsular as abstrações matemáticas da mecânica quântica e da álgebra linear necessárias para a escrita de algorimos quânticos. A linguagem QCL, proposta por Ömer (1998), é reconhecidamente a primeira linguagem de programação quântica de alto nível. A sintaxe da QCL é derivada da linguagem imperativa C;

c) **Representações quânticas intermediárias de baixo nível:** são linguagens de baixo nível utilizadas na representação de algoritmos, principalmente na forma de circuitos. Um exemplo de linguagem nesta categoria é o QASM, linguagem utilizada como notação de descrição de circuitos quânticos. Linguagens de baixo nível mais completas, que podem expressar fluxos de controle, são utilizadas como linguagem de máquina, código alvo do processo de compilação. Um exemplo de linguagem quântica intermediária de baixo ní-

vel completa é o Quil, que pode ser usada para descrever um conjunto de instruções a serem executadas pela QPU.

Segundo Svore et al. (2006), o maior desafio da arquitetura de software quântico reside na criação de uma linguagem de programação quântica de alto nível que seja capaz de encapsular os princípios da mecânica quântica de uma forma natural, de maneira que físicos e cientistas da computação possam desenvolver e validar um número maior de algoritmos quânticos.

2.2.3 Simuladores

Na arquitetura de quatro camadas proposta por Svore et al. (2006), as camadas de maior nível são independentes do hardware, enquanto as camadas de baixo nível são específicas para cada implementação física. Devido à complexidade dos sistemas de hardware quânticos, simuladores têm um papel muito importante no projeto e na validação de algoritmos e circuitos quânticos. A proposta de arquitetura em camadas facilita a abstração, pois o mesmo código de alto nível pode ser executado por qualquer implementação de hardware quântico, sejam simuladores ou as próprias implementações físicas. (SVORE; TROYER, 2016).

Pelo fato do modelo matemático dos estados quânticos, portas quânticas e medições envolverem álgebra linear, um aspecto fundamental de qualquer simulação de sistema quântico eficiente em um computador clássico é a exploração de estruturas em matrizes e vetores. Diversos simuladores em modelos computacionais clássicos foram propostos e algumas linguagens de programação contemplam simuladores locais em suas bibliotecas.

3 TRABALHOS RELACIONADOS

O Quadro 1 apresenta os critérios utilizados na pesquisa de trabalhos relacionados.

Quadro 1 – Critérios de pesquisa de trabalhos relacionados

Termos pesquisados	*quantum programming languages, quantum computing platforms, quantum algorithms*
Área de pesquisa	Computação ("COMP") ou física ("PHYS")
Publicações	Periódicos, conferências, revistas científicas, dissertações de mestrado e teses de doutorado
Período de análise	A partir de 2018
Bancos de dados científicos	Arvix, Google Scholar, IEEExplore, Nature, Scopus e Springer

Fonte: Elaborado pelo autor.

Com base nos resultados da pesquisa por trabalhos relacionados, os seguintes artigos correlatos foram selecionados.

3.1 Apresentação de Trabalhos Relacionados

3.1.1 Overview and Comparison of Gate Level Quantum Software Platforms

LaRose (2019) realizou uma pesquisa sobre ambientes de computação quântica, comparando as plataformas de software disponíveis. O objetivo do trabalho é prover uma visão geral dos ambientes computacionais relacionados às plataformas. A motivação do estudo vem da dificuldade de estudantes e pesquisadores em decidir qual ambiente de computação quântica deve ser usado, se perdendo na documentação ou não sabendo por onde começar.

A pesquisa compara as plataformas em diferentes aspectos, como: instalação, documentação e tutoriais, sintaxe das linguagens, suporte das bibliotecas, hardware quântico e compiladores quânticos. As plataformas consideradas para a pesquisa foram: pyQuil da Rigetti, QISKit da IBM, ProjectQ da ETH Zurich e Quantum Development Kit da Microsoft. LaRose (2019) também avaliou o desempenho dos simuladores locais das plataformas QISKit e ProjectQ e dos hardwares quânticos disponíveis das plataformas pyQuil e QISKit. A pesquisa seguiu uma abordagem qualitativa na análise das características de linguagens de programação e uma abordagem quantitativa na comparação de desempenho de simuladores locais.

Os resultados obtidos indicam que existem diferentes sugestões para diferentes perfis de estudantes ou pesquisadores, dependendo de seu nível de experiência com programação. LaRose (2019) ainda conclui que todas as plataformas estudadas obtiveram conquistas significativas na área da computação quântica e oferecem excelentes utilitários para pesquisadores e estudantes iniciarem na programação quântica.

3.1.2 Quantum Programming Language: A Systematic Review of Research Topic and Top Cited Languages

Garhwal, Ghorani e Ahmad (2019) realizaram uma pesquisa com o objetivo de comparar linguagens de programação quânticas de alto nível, levando em consideração suas características e funcionalidades. A pesquisa tem como objetivo a comparação de linguagens implementadas em hardware quântico, não abordando simuladores e emuladores quânticos. A pesquisa abordou como questões centrais as diferentes linguagens de programação quânticas, as tendências na área de pesquisa de software quântico e a comparação e classificação de linguagens em paradigmas de programação.

Em sua pesquisa bibliográfica, Garhwal, Ghorani e Ahmad (2019) analisaram artigos, periódicos e conferências publicados entre 1985 e 2018, nos bancos de dados científicos Arvix, Google Scholar, IEEExplose, Science Direct e Web of Knowledge. A pesquisa contempla um breve histórico do desenvolvimento de linguagens de programação quânticas, além da análise e comparação de características das linguagens e sua classificação dentre os paradigmas imperativo, funcional e orientado a objetos.

3.1.3 Quantum Programming Languages

Heim et al. (2020) realizaram um estudo sobre linguagens de programação quânticas, destacando aspectos importantes da programação quântica e como ela difere da programação clássica. O estudo objetiva apresentar o estado da arte na área das linguagens de programação quânticas, destacando suas principais características e provendo exemplos de código fonte. A pesquisa tem como motivação a crescente relevância do estudo das linguagens de programação quânticas e sua importância na arquitetura do modelo computacional quântico.

Em seu estudo, Heim et al. (2020) destacam casos de uso relevantes de cada linguagem, demonstrando a importância das ferramentas de software como simuladores, bibliotecas, documentação e materiais de aprendizagem. As linguagens selecionadas para um estudo aprofundado seguindo uma abordagem qualitativa foram: Q#, QISKit, Cirq, Quipper e Scaffold. Estas linguagens foram selecionadas por representar diferentes paradigmas de programação, facilitando a escolha de um determinado paradigma pela comunidade científica ou acadêmica. Apesar de não terem sido escolhidas baseadas em sua popularidade ou volume de uso, Heim et al. (2020) afirmam que muitas características das linguagens abordadas também estão presentes em diversas outras linguagens que não foram objeto de estudo. O estudo também apresenta um algoritmo de exemplo expresso em cada linguagem juntamente com uma coleção de imagens de contêineres Docker que possibilitam fácil acesso às plataformas computacionais utilizadas.

3.2 Análise de Trabalhos Relacionados

O Quadro 2 ilustra um comparativo de linguagens de programação abordadas nos trabalhos relacionados.

Quadro 2 – Comparativo de linguagens de programação citadas nos trabalhos relacionados

Linguagens abordadas	**Overview and Comparison of Gate Level Quantum Software Platforms**	**Quantum Programming Language: A Systematic Review of ResearchTopic and Top Cited Languages**	**Quantum Programming Languages**
Cirq			✓
LanQ		✓	
Linguagem Q		✓	
ProjectQ	✓		
Q#	✓	✓	✓
QASM		✓	
QCL		✓	
QISKit	✓		✓
QML		✓	
QuaFL		✓	
Quill	✓		
Quipper		✓	✓
Q$\|SI\rangle$		✓	
Scaffold			✓
Strawberry Fields	✓	✓	
qGCL		✓	

Fonte: Elaborado pelo autor.

Com base na pesquisa de trabalhos relacionados, percebe-se que não foi abordada a análise de desempenho de linguagens de programação e APIs quânticas (simuladores) utilizando

algoritmos presentes na literatura, que são objetivos deste estudo. A escolha das linguagens de programação e APIs abordadas neste estudo considerou os seguintes critérios: (a) citadas nos trabalhos relacionados; (b) disponham de simuladores locais; (c) disponham de documentação; (d) relevância dos laboratórios de computação quântica em que a linguagem foi criada; (e) possuam implementação dos algoritmos constantes na literatura. A partir desses critérios, a seguir, apresentam-se as linguagens de programação e APIs quânticas consideradas na análise de desempenho:

a) **Cirq:** criado em 2018 nos laboratórios de computação quântica da Google, o Cirq é uma biblioteca de código aberto baseada na linguagem Python. Seu objetivo é prover as abstrações necessárias para a escrita, manipulação e otimização de circuitos quânticos, incluindo o suporte para a execução dos circuitos em dispositivos e simuladores quânticos. (HEIM et al., 2020). Um exemplo de código é mostrado no Apêndice B;

b) **PyQuil:** é uma biblioteca para a linguagem Python, desenvolvida pela Rigetti em 2016. O objetivo do PyQuil é prover as abtrações de alto nível necessárias na construção de circuitos para a linguagem de baixo nível Quil. Juntamente com máquinas virtuais, compiladores e simuladores, as linguagens Quil e PyQuil fazem parte da plataforma de software quântico da Rigetti, chamada Forest. (SMITH; CURTIS; ZENG, 2016). Um exemplo de código é mostrado no Apêndice C;

c) **Q#:** é uma linguagem de domínio específico quântico de alto nível, desenvolvida como um projeto de código aberto pela Microsoft em 2018. Faz parte do kit de desenvolvimento quântico (QDK), que ainda conta com simuladores, compiladores e bibliotecas que auxiliam no desenvolvimento de software quântico. A principal característica da linguagem Q# é o fato de ser uma linguagem *stand-alone*, que não depende de uma linguagem hospedeira para ser executada. (SVORE et al., 2018). Um exemplo de código é mostrado no Apêndice D;

d) **Qiskit:** é uma biblioteca para a linguagem Python, parte do kit de desenvolvimento de software quântico (SDK) de código aberto desenvolvido nos laboratórios de computação quântica da IBM. É composto de bibliotecas de software baseadas na linguagem Python, incluindo a linguagem de baixo nível OpenQASM, compiladores e simuladores quânticos. (ABRAHAM et al., 2019). Um exemplo de código é mostrado no Apêndice E.

Dentre as linguagens selecionadas, três são bibliotecas desenvolvidas para a linguagem Python. Segundo Hidary (2019), a escolha do Python para desenvolvimento de linguagens e bibliotecas de programação quântica deve-se à sua curva de aprendizagem. Características da linguagem como tipagem dinâmica e fácil instalação permitem ao programador focar no problema a ser resolvido sem se preocupar com detalhes técnicos.

4 MATERIAIS E MÉTODOS

Segundo Wazlawick (2009), o método de pesquisa consiste na sequência de passos necessários para demonstrar que o objetivo proposto foi atingido. Para Gerhardt e Silveira (2009), metodologia é o estudo dos métodos e instrumentos necessários para uma pesquisa ser realizada, validando os meios escolhidos para se chegar ao fim proposto.

A natureza do presente estudo é aplicada, pois objetiva gerar conhecimentos de aplicação prática para a solução de um problema específico. O conhecimento gerado através da pesquisa tem caráter científico, pois apresenta as duas características necessárias: a determinação de um objeto de investigação e a explicitação de um método de pesquisa. (GERHARDT; SILVEIRA, 2009).

A abordagem da pesquisa é quantitativa, pois utiliza da matemática para enfatizar atributos mensuráveis, como: tempo de execução e recursos computacionais consumidos por algoritmos em diferentes linguagens de programação e APIs quânticas. (GERHARDT; SILVEIRA, 2009).

Do ponto de vista dos objetivos deste estudo, a pesquisa é classificada como exploratória, pois aprimora ideias presentes em trabalhos relacionados e tem como objetivo a produção de conhecimento, proporcionando maior familiaridade sobre o tema das linguagens de programação e APIs quânticas. (GIL, 2002).

Em relação à classificação de procedimentos, esse estudo é caracterizado como experimental, pois define formas de observação de efeitos de variáveis no objeto de estudo. Como objeto de estudo, são consideradas as seguintes variáveis de desempenho de linguagens de programação e APIs quânticas: os diferentes algoritmos quânticos e o número de *qubits* utilizados na experimentação. (GIL, 2002).

4.1 Técnica de coleta de dados

4.1.1 Método *stopwatch*

O método *stopwatch*, que consiste na utilização do relógio embutido no computador para estimar o tempo decorrido da execução de um programa, foi utilizado durante a execução dos experimentos. O método é considerado uma estimativa, pois não leva em consideração possíveis interrupções, preempções e tempo de I/O durante a execução de um programa. A utilização do método se dá através da medição do tempo do relógio do sistema antes e depois da execução do programa a ser medido. (STEWART, 2002).

O Q#, por ser uma linguagem de domínio específico *stand-alone*, não conta com bibliotecas que auxiliem na medição de tempo. Por este motivo, foi utilizado um programa auxiliar escrito em C# que encapsula o programa escrito em Q# e implementa o método *stopwatch*. O programa auxiliar escrito em C# utiliza a classe *Stopwatch* do pacote *System.Diagnostics*, que fornece as operações necessárias para a medição do tempo decorrido. A Figura 5 mostra a utilização do

método *stopwatch* na linguagem C#.

Figura 5 – Exemplo de código utilizando o método *Stopwatch* na linguagem C#

```
Stopwatch stopWatch = new Stopwatch();
stopWatch.Restart();
var deutschJoszaResult = await RunDeutschJosza.Run(sim, numQubits, 1);
stopWatch.Stop();
```

Fonte: Elaborada pelo autor.

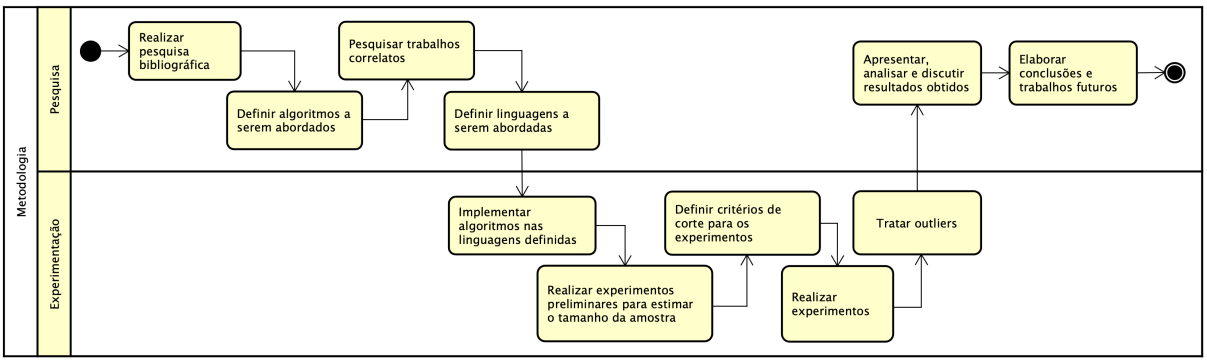
As linguagens que são embutidas ou bibliotecas da linguagem Python possibilitam a escrita de programas com instruções quânticas e clássicas, tornando possível a utilização dos módulos do Python em conjunto com as instruções específicas de cada linguagem quântica. O módulo *time* do Python fornece rotinas auxiliares para a medição de tempo decorrido, incluindo o método *time.perf_counter*, que foi utilizado.

4.1.2 Detecção de Outliers

Zimek e Filzmoser (2018) definem *outliers* como observações que aparentam um desvio incoerente em um conjunto de dados, que podem levar a análises inconsistentes. No presente estudo, os *outliers* foram identificados através da validação dos dados obtidos utilizando o método Box-plot. O método consiste em dividir a amostra em quartis, definindo um intervalo interquartis (IQR), que é então utilizado para definir valores máximos e mínimos dentro da amostra. Observações situadas acima do máximo ou abaixo do mínimo foram então consideradas *outliers* e desconsideradas na etapa de análise de resultados. (DAWSON, 2011).

A Figura 6 ilustra as etapas da metodologia de pesquisa e experimentação utilizadas neste estudo.

Figura 6 – Desenho da metodologia



Fonte: Elaborada pelo Autor.

5 APRESENTAÇÃO E ANÁLISE DE RESULTADOS

Nesta seção são abordadas a apresentação, análise e discussão dos resultados. Na subseção de apresentação, define-se o tamanho da amostra, pontos de corte dos experimentos e o ambiente de execução dos experimentos. Já na subseção de análise, são apresentados os resultados obtidos de cada algoritmo com suas respectivas análises e observações. A seção é finalizada com a discussão das observações realizadas. No contexto do presente estudo, um **cenário** se refere à implementação de um algoritmo quântico em uma linguagem de programação.

5.1 Apresentação

5.1.1 Estimativa de tamanho de amostra e pontos de corte

Através de medições preliminares, foram definidos o tamanho de amostra e os pontos de corte das medições. O cenário utilizado para o cálculo de estimativa de tamanho de amostra foi a implementação do algoritmo de Deutsch-Jozsa na linguagem Qiskit utilizando 10 *qubits*. A partir daí, foram realizadas 30 medições preliminares com um intervalo de confiança (IC) de 95% e margem de erro de 5%, resultando em um tamanho estimado de amostra de 42 medições. (ALMEIDA, 2014).

Segundo Almeida (2014), a coleta de amostras na área da Ciência da Computação experimental pode se tornar um processo caro. No presente estudo, por exemplo, se executadas todas as medições do tamanho estimado de amostra para cenários com tempo de execução maior que 300 segundos, as medições levariam mais de 210 horas para serem executadas. Por este motivo, foram definidos pontos de corte baseados no tempo de execução das medições preliminares, descritos abaixo:

a) Para cenários em que o tempo obtido através de medições foi menor ou igual a 60 segundos, foram executadas 42 medições;

b) Para cenários em que o tempo obtido através de medições foi maior que 60 segundos e menor que 300 segundos, foi executada uma única medição;

c) Para cenários em que o tempo de execução foi maior que 300 segundos, a execução do programa foi interrompida e as medições não foram consideradas na análise.

5.1.2 Ambiente de execução dos experimentos

Os experimentos foram realizados em um MacBook Pro 2019 com as configurações detalhadas no Quadro 3.

Quadro 3 – Configuração utilizada nos experimentos

Sistema Operacional	macOS Big Sur 11.3
Memória principal	DDR4 16GB 2400 MHz
Memória secundária	Apple SSD 256 GB PCI-Express
CPU	Intel Core i7 2,6 GHz 6-Core
GPUs	Radeon Pro 555X 4GB e Intel UHD Graphics 630 1536 MB

Fonte: Elaborado pelo autor.

O Quadro 4 apresenta as características das linguagens de programação, APIs e simuladores quânticos abordados nos experimentos.

Quadro 4 – Linguagens de programação abordadas nos experimentos

Linguagem	**Versão**	**Paradigmas suportados**	**Classificação**	**Linguagem clássica**	**Simulador**
Cirq	0.11.0	Imperativo e Funcional	Biblioteca (API)	Python	*pure state simulator*
PyQuil	2.28.1	Imperativo	Biblioteca (API)	Python	QVM (local)
Q#	0.17.2105	Imperativo e Orientado a Objetos	Linguagem de alto nível	C#	*QuantumSimulator*
Qiskit	0.25.4	Imperativo e Funcional	Biblioteca (API)	Python	*qasm_simulator*

Fonte: Elaborado pelo autor.

5.2 Análise

5.2.1 Algoritmo de Bernstein-Vazirani

A implementação do algoritmo na biblioteca Qiskit não apresentou resultados consistentes na comparação com cenários similares. O tempo de execução se manteve constante independentemente do número de *qubits*, comportamento que viola a teoria da representação de *qubits* em estruturas de memória clássicas. Em cenários com algoritmos similares, a biblioteca suportou um número máximo de 15 *qubits*, enquanto no algoritmo de Bernstein-Vazirani, mesmo em execuções com 500 *qubits*, o tempo se manteve constante. Por este motivo, o cenário do algoritmo de Bernstein-Vazirani implementado na biblioteca Qiskit foi desconsiderado da análise. A Tabela 1 apresenta um comparativo de desempenho do algoritmo de Bernstein-Vazirani.

Tabela 1 – Comparativo de desempenho do algoritmo de Bernstein-Vazirani

Qubits	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**
Cirq	0,0025	0,0033	0,0042	0,0040	0,0052	0,0058	0,0072	0,0076	0,0103	0,0142	0,0229	0,0403	0,0688	0,1355
PyQuil	0,0158	0,0174	0,0195	0,0370	0,1169	0,5152	1,9798	8,9378	40,0303	223,0088	-	-	-	-
Q#	-	-	0,0028	0,0026	0,0046	0,0046	0,0033	0,0048	0,0068	0,0055	0,0095	0,0073	0,0072	0,0080

Qubits	**15**	**16**	**17**	**18**	**19**	**20**	**21**	**22**	**23**	**24**	**25**	**26**	**27**	**28**
Cirq	0,2677	0,5828	1,3162	2,3274	4,4849	9,8072	18,9189	39,3824	79,3850	165,1642	-	-	-	-
PyQuil	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Q#	0,0092	0,0150	0,0245	0,0593	0,1008	0,2379	0,4987	1,0246	2,0694	4,1216	8,3368	16,9274	34,5844	117,642

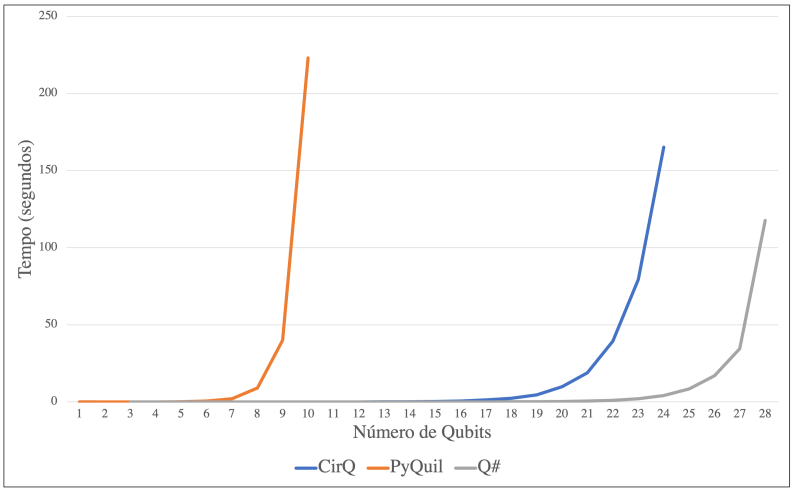
* Os tempos representam a média amostral em segundos (s).

Fonte: Elaborada pelo autor.

A linguagem Q# suportou o maior número de *qubits* antes do ponto de corte, com um máximo de 28 *qubits*. No cenário com número máximo de *qubits* suportado por mais de uma

linguagem (24), o Q# foi 39 vezes mais rápido do que o Cirq. A biblioteca PyQuil obteve o pior desempenho do algoritmo, com um máximo de 10 *qubits* suportados. O Gráfico 1 apresenta o tempo de execução do algoritmo de Bernstein-Vazirani.

Gráfico 1 – Desempenho do algoritmo de Bernstein-Vazirani



Fonte: Elaborado pelo Autor.

Pode ser observado no Gráfico 1 um comportamento assintótico exponencial em relação ao número de *qubits*. Este comportamento se justifica pela complexidade da operação de produto tensorial realizada nos simuladores, que aumenta exponencialmente em relação ao número de *qubits* do circuito. (SILVA, 2018).

5.2.2 Algoritmo de Deutsch-Jozsa

A Tabela 2 apresenta um comparativo de desempenho do algoritmo de Deutsch-Jozsa.

Tabela 2 – Comparativo de desempenho do algoritmo de Deutsch-Jozsa

Qubits	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**
Cirq	0,0023	0,0030	0,0037	0,0040	0,0050	0,0063	0,0072	0,0101	0,0121	0,0178	0,0278	0,0497	0,0959	0,1817
PyQuil	0,0073	0,0095	0,0187	0,0488	0,1706	0,6373	2,5608	11,6261	39,7587	190,0843	-	-	-	-
Q#	-	-	0,0030	0,0020	0,0023	0,0031	0,0034	0,0041	0,0045	0,0057	0,0063	0,0071	0,0075	0,0089
Qiskit	0,0125	0,0167	0,0217	0,0353	0,0553	0,0952	0,1681	0,3596	0,7996	2,3090	6,2794	14,1137	32,7595	92,8678

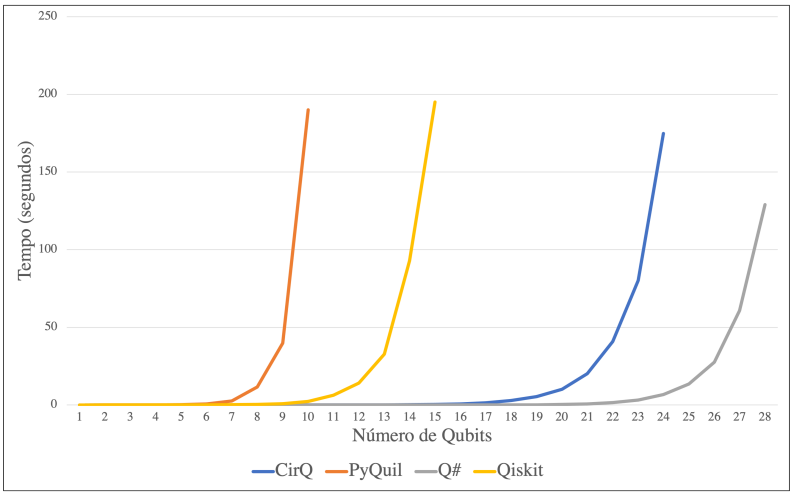
Qubits	**15**	**16**	**17**	**18**	**19**	**20**	**21**	**22**	**23**	**24**	**25**	**26**	**27**	**28**
Cirq	0,3333	0,6697	1,4254	2,8252	5,3883	10,1397	20,1739	40,9157	80,1459	174,8623	-	-	-	-
PyQuil	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Q#	0,0116	0,0160	0,0252	0,0501	0,1302	0,3287	0,7108	1,5282	3,2288	6,7063	13,5695	27,6173	60,8920	129,0020
Qiskit	195,1136	-	-	-	-	-	-	-	-	-	-	-	-	-

* Os tempos representam a média amostral em segundos (s).

Fonte: Elaborada pelo autor.

Novamente, a linguagem que suportou o maior número de *qubits* foi o Q#, com 28 *qubits*. A biblioteca Cirq foi a que mais se aproximou do Q#, com número máximo de 24 *qubits*. No cenário com o maior número de *qubits* suportado por mais de uma linguagem (24), a linguagem Q# foi 25 vezes mais rápida que o Cirq. O Gráfico 2 apresenta o tempo de execução do algoritmo de Deutsch-Jozsa.

Gráfico 2 – Desempenho do algoritmo de Deutsch-Jozsa



Fonte: Elaborado pelo Autor.

As quatro linguagens abordadas apresentam um comportamento assintótico exponencial em relação ao número de *qubits*, como pode se observar no Gráfico 2. O comportamento assintótico das linguagens Q# e Cirq começa a crescente exponencial a partir de um número maior de *qubits* em comparação com o comportamento das demais linguagens.

5.2.3 Algoritmo de Grover

A Tabela 3 apresenta o comparativo de desempenho do algoritmo de Grover.

Tabela 3 – Comparativo de desempenho do algoritmo de Grover

Qubits	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**
CirQ	0,0032	0,0046	0,0054	0,0063	0,0076	0,0095	0,0102	0,0119	0,0148	0,0195	0,0272	0,0441	0,0775	0,1426
PyQuil	0,0060	0,0084	0,0099	0,0186	0,0380	0,1391	0,5811	2,6083	12,5186	60,3491	-	-	-	-
Q#	-	-	0,0050	0,0066	0,0085	0,0094	0,0099	0,0106	0,0134	0,0058	0,0131	0,0094	0,0057	0,0231
Qiskit	0,0137	0,0246	0,0720	0,2238	0,6456	1,6999	4,4459	13,6375	27,9736	63,2195	171,8272	-	-	-

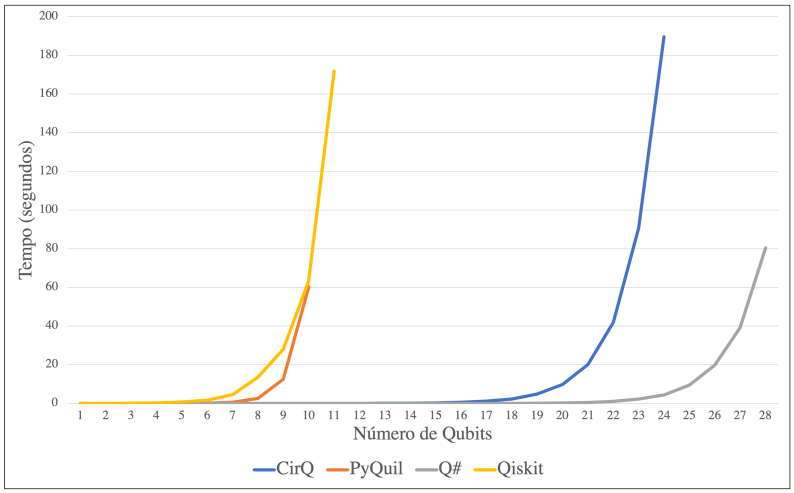
Qubits	**15**	**16**	**17**	**18**	**19**	**20**	**21**	**22**	**23**	**24**	**25**	**26**	**27**	**28**
CirQ	0,2825	0,5426	1,1243	2,2560	4,8161	9,7674	20,0535	41,8487	90,6747	189,5509	-	-	-	-
PyQuil	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Q#	0,0260	0,0299	0,0361	0,0474	0,0847	0,1935	0,4752	1,0156	2,1896	4,3699	9,4858	19,7452	39,1347	80,409
Qiskit	-	-	-	-	-	-	-	-	-	-	-	-	-	-

* Os tempos representam a média amostral em segundos (s).

Fonte: Elaborada pelo autor.

Ao analisar a Tabela 3, novamente destacam-se negativamente o desempenho das bibliotecas Qiskit e PyQuil em relação às demais. Enquanto Cirq e Q# simularam o algoritmo de Grover com mais de 20 *qubits*, PyQuil e Qiskit suportaram no máximo 10 e 11 *qubits*, respectivamente. No cenário do algoritmo de Grover com 10 *qubits*, as bibliotecas Qiskit e PyQuil apresentaram um desempenho próximo, com apenas 4,7% de vantagem para a biblioteca PyQuil. O Gráfico 3 apresenta o tempo de execução do algoritmo de Grover.

Gráfico 3 – Desempenho do algoritmo de Grover



Fonte: Elaborado pelo Autor.

Ao analisar o Gráfico 3, percebe-se a proximidade dos comportamentos assintóticos das bibliotecas PyQuil e Qiskit. As linguagens Q# e Cirq novamente se destacam pelo número de *qubits* suportados antes de seus comportamentos assintóticos iniciarem a ascendente exponencial.

5.2.4 Algoritmo de Shor

O algoritmo de Shor, por ter como parâmetro dois números inteiros a e N, não varia com o número de *qubits* como os outros algoritmos. Por conta dos requisitos de memória necessários para simular o algoritmo de Shor, as bibliotecas e linguagens de programação recomendam o número 15 como parâmetro N em experimentos, pois têm como fatores números primos relativamente baixos (3 e 5). Nos experimentos realizados, o parâmetro $N = 15$ foi utilizado, variando os valores de a. Pela definição do algoritmo, para $N = 15$, os valores de a devem ser menores que 15 e primos entre si (o máximo divisor comum entre a e N deve ser 1). A Tabela 4 apresenta o comparativo de desempenho do algoritmo de Shor.

Tabela 4 – Comparativo de desempenho do algoritmo de Shor

Parâmetro a	**7**	**8**	**11**	**13**
Cirq	1,6171	1,6408	1,7799	1,9133
PyQuil	1,9593	1,6999	2,9009	3,5229
Q#	2,5723	2,1275	2,4245	2,9224
Qiskit	1,4230	0,9822	1,1601	1,4361

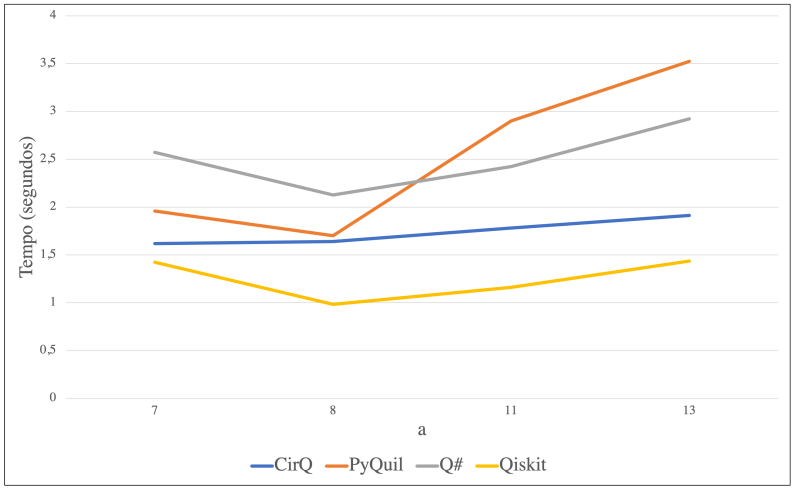
* Tempos da média amostral em segundos (s).

Fonte: Elaborada pelo autor.

Destaca-se o desempenho da biblioteca Qiskit, que apresentou os melhores tempos de execução no algoritmo de Shor, contrastando com o desempenho inferior obtido nos demais algo-

ritmos. Também observa-se que as linguagens PyQuil, Q# e Qiskit executaram o algoritmo em um tempo inferior no cenário $a = 8$ em relação ao tempo obtido no cenário $a = 7$. O Gráfico 4 apresenta o tempo de execução do algoritmo de Shor.

Gráfico 4 – Desempenho do algoritmo de Shor



Fonte: Elaborado pelo Autor.

Diferentemente do gráfico dos demais algoritmos, o Gráfico 4 não apresenta um comportamento assintótico exponencial em relação ao parâmetro a. A biblioteca Qiskit obteve um desempenho superior às demais linguagens em todos os cenários do algoritmo de Shor.

5.2.5 Algoritmo de Simon

A Tabela 5 apresenta o comparativo de desempenho do algoritmo de Simon.

Tabela 5 – Comparativo de desempenho do algoritmo de Simon

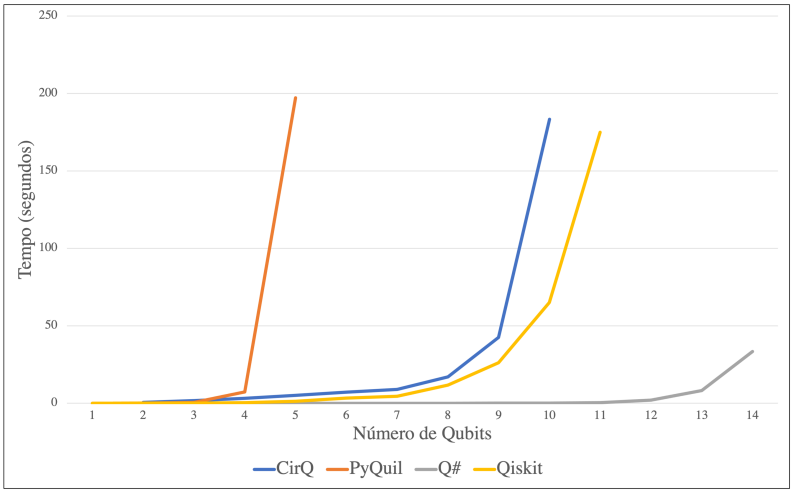
Qubits	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**
CirQ	-	0,5678	1,7322	3,118	5,0432	7,0925	8,8892	17,0566	42,5357	183,3877	-	-	-	-
PyQuil	0,0003	0,0312	0,5603	7,2436	197,1568	-	-	-	-	-	-	-	-	-
Q#	-	0,0010	0,0024	0,0036	0,0042	0,0048	0,0056	0,0101	0,0209	0,0793	0,4192	1,9467	8,1802	33,4786
Qiskit	0,0179	0,0275	0,1366	0,4429	1,3238	3,3236	4,5755	11,6858	26,1807	65,1205	174,9935	-	-	-

* Os tempos representam a média amostral em segundos (s).

Fonte: Elaborada pelo autor.

A Tabela 5 demonstra uma peculiaridade do algoritmo de Simon: para executar o circuito, o simulador necessita de $2n + 1$ *qubits*, onde n é o tamanho em *qubits* da função de entrada. A linguagem Q#, por exemplo, manteve o número máximo de *qubits* em 28 nos demais algoritmos (com exceção do algoritmo de Shor), e na execução do algoritmo de Simon obteve um número máximo de 14 *qubits*. Destaca-se também o número máximo de *qubits* obtido pela biblioteca Qiskit, que se manteve estável em 11 em relação aos outros algoritmos. O Gráfico 5 apresenta o tempo de execução do algoritmo de Simon.

Gráfico 5 – Desempenho do algoritmo de Simon



Fonte: Elaborado pelo Autor.

Pelo número de *qubits* necessário para a simulação do algoritmo de Simon, o comportamento assintótico exponencial observado inicia em números menores de *qubits* em relação aos demais algoritmos. Também percebe-se a proximidade dos comportamentos assintóticos das linguagens Qiskit e Cirq.

5.3 Discussão

A linguagem Q# obteve, no geral, o melhor desempenho dentre as linguagens de programação e APIs quânticas abordadas. O simulador local da linguagem Q# foi capaz de executar cenários com até 28 *qubits*, sendo a linguagem com o simulador local de maior capacidade. O simulador local da linguagem Cirq também obteve desempenho significativo em relação às outras linguagens, sendo possível executar cenários com até 24 *qubits*. A biblioteca Qiskit obteve um desempenho abaixo do esperado, pois é a biblioteca mais consolidada considerando-se o número de repositórios no Github, citações em artigos e número de exemplos e tutoriais disponíveis para a biblioteca. A IBM, empresa pioneira na disponibilização de ambientes quânticos através da Internet, através de seus esforços na área da pesquisa da computação quântica, também contribuiu para a expectativa gerada sob a biblioteca. A Tabela 6 apresenta o número máximo de *qubits* suportado em cada linguagem por algoritmo.

Tabela 6 – Número máximo de *qubits*

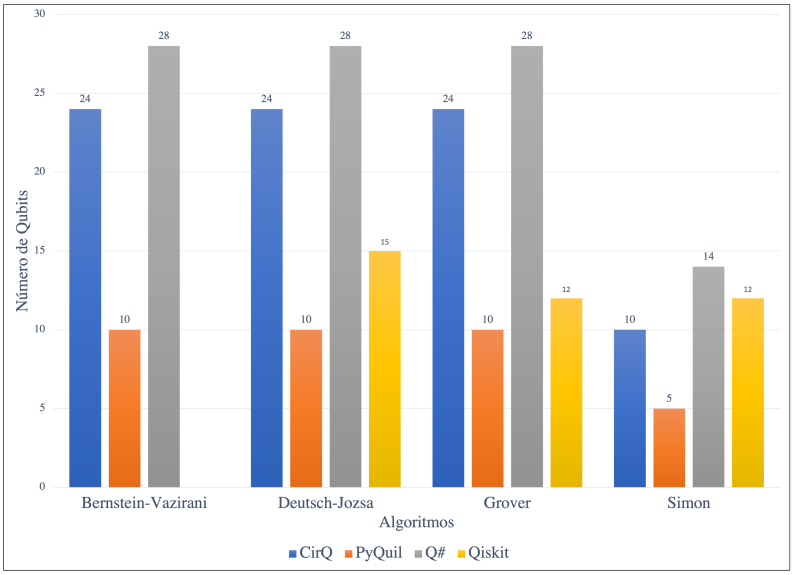
Algoritmo	**Cirq**	**Pyquil**	**Q#**	**Qiskit**
Bernstein-Vazirani	24	10	28	-
Deutsch-Jozsa	24	10	28	15
Grover	24	10	28	12
Simon	10	5	14	12

Fonte: Elaborada pelo autor.

A Tabela 6 indica que a linguagem Qiskit suportou um numero máximo de 15 *qubits*. Em uma comparação com os resultados obtidos pelo trabalho correlato, LaRose (2019), em sua análise de desempenho de simuladores quânticos, obteve um número máximo de 12 *qubits* no simulador da linguagem Qiskit. Também destaca-se que LaRose (2019) obteve um número máximo de 28 *qubits* no simulador ProjectQ, mesma capacidade apresentada pela linguagem Q# no presente estudo.

O Gráfico 6 apresenta o número máximo de *qubits* suportado em cada algoritmo pelas linguagens de programação por algoritmo.

Gráfico 6 – Número máximo de *qubits* obtido em cada linguagem por algoritmo



Fonte: Elaborado pelo Autor.

Ao analisar os gráficos de tempo de execução obtidos, nota-se um padrão recorrente nos cenários: um comportamento assintótico exponencial em relação ao número de *qubits*. O motivo deste fenômeno é a complexidade de representar estados quânticos utilizando estruturas de dados clássicas.

Pelo fato dos *qubits* serem capazes de entrar em um estado de superposição, ou seja, estarem em mais de um estado ao mesmo tempo, os simuladores locais também necessitam representar estes estados quânticos em estruturas de dados clássicas. Para representar n *qubits* em estruturas de memória clássicas, são necessários 2^n bits, uma relação exponencial. Ao aplicar as operações quânticas para alterar o estado dos *qubits*, diversas operações de produto tensorial (espaço vetorial) são aplicadas à todas as combinações de estado possíveis, levando à uma complexidade $\mathcal{O}(2^n)$. Por este motivo, existe uma limitação dos simuladores locais em relação ao número de *qubits*. (NIELSEN; CHUANG, 2010).

6 CONCLUSÕES E TRABALHOS FUTUROS

A computação quântica se mostra uma área de pesquisa muito promissora. A capacidade do modelo computacional quântico, demonstrada pelos algoritmos quânticos, desperta o interesse de empresas, instituições governamentais e Universidades. Mesmo com a arquitetura de hardware quântico em constante mudança, diversas linguagens de programação e APIs quânticas estão disponíveis, reforçando a importância da arquitetura de software quântico para o avanço da computação quântica.

Este estudo abordou a análise de desempenho de linguagens de programação e APIs quânticas utilizando algoritmos quânticos recorrentes na literatura, motivado pela dificuldade de estudantes e pesquisadores decidirem qual linguagem utilizar. Para responder à questão de pesquisa e, consequentemente, atingir os objetivos deste estudo, foi realizada uma pesquisa bibliográfica sobre Computação Quântica e Arquitetura de Software Quântico, com ênfase nas linguagens de programação e APIs quânticas. A pesquisa de trabalhos relacionados apresentou três trabalhos correlatos, que auxiliaram na definição das linguagens de programação e APIs quânticas a serem abordadas na análise de desempenho. Na etapa de coleta de dados, foram realizados 19 cenários de análise de desempenho, considerando diferentes algoritmos, linguagens de programação e número de *qubits*.

A análise dos resultados indicou que a linguagem Q# obteve o melhor desempenho dentre as linguagens abordadas, em tempo médio de execução e número máximo de *qubits*. Para estudantes e pesquisadores com experiência prévia em desenvolvimento de software, sugere-se a utilização da linguagem Q#. Para estudantes e pesquisadores sem experiência prévia, a sugestão é a biblioteca Cirq. Um repositório no *Github*[4] contendo a implementação dos diferentes algoritmos nas linguagens de programação e APIs quânticas abordadas foi publicado com o objetivo de contribuir com a comunidade de computação quântica.

Destaca-se que, a partir da análise e discussão dos resultados, foi possível responder a questão de pesquisa desse estudo, sendo que os objetivos propostos foram cumpridos, destacando-se a análise de desempenho de linguagens de programação e APIs quânticas utilizando algoritmos quânticos recorrentes na literatura.

6.1 Trabalhos Futuros

Como trabalhos futuros, sugerem-se:

a) proposição de uma taxonomia de linguagens de programação e APIs quânticas;

b) análise quantitativa de desempenho de ambientes computacionais de hardware quântico disponíveis através de computação em nuvem (Microsoft Azure Quantum, IBM Q Experience, Google Quantum Computing Service);

[4] Acesse o repositório em: https://github.com/jorgeviegas/quantumalgorithms

c) análise quantitativa comparativa entre o desempenho de simuladores locais e ambientes computacionais de hardware quântico disponíveis através de computação em nuvem.

Referências

ABRAHAM, H. et al. **Qiskit: An Open-source Framework for Quantum Computing**. 2019.

ALMEIDA, V. Métodos quantitativos para ciência da computação experimental. 2014.

ARUTE, F. et al. Quantum supremacy using a programmable superconducting processor. **Nature**, v. 574, n. 7779, p. 505–510, Oct 2019. ISSN 1476-4687. Disponível em: <https://doi.org/10.1038/s41586-019-1666-5>.

BETTELLI, S.; CALARCO, T.; SERAFINI, L. Toward an architecture for quantum programming. **The European Physical Journal D - Atomic, Molecular and Optical Physics**, Springer Science and Business Media LLC, v. 25, n. 2, p. 181–200, Aug 2003. ISSN 1434-6079. Disponível em: <http://dx.doi.org/10.1140/epjd/e2003-00242-2>.

CLEVE, R. et al. Quantum algorithms revisited. **Proceedings of the Royal Society of London. Series A: Mathematical, Physical and Engineering Sciences**, The Royal Society, v. 454, n. 1969, p. 339–354, Jan 1998. ISSN 1471-2946. Disponível em: <http://dx.doi.org/10.1098/rspa.1998.0164>.

DAWSON, R. How significant is a boxplot outlier? **Journal of Statistics Education**, Taylor Francis, v. 19, n. 2, p. null, 2011. Disponível em: <https://doi.org/10.1080/10691898.2011.11889610>.

DEUTSCH, D. Quantum theory, the Church-Turing principle and the universal quantum computer. **Proceedings of the Royal Society of London Series A**, v. 400, n. 1818, p. 97–117, jul. 1985.

DIRAC, P. A. M. A new notation for quantum mechanics. **Mathematical Proceedings of the Cambridge Philosophical Society**, Cambridge University Press, v. 35, n. 3, p. 416–418, 1939.

FEYNMAN, R. P. Simulating physics with computers. **International Journal of Theoretical Physics**, v. 21, n. 6, p. 467–488, Jun 1982. ISSN 1572-9575. Disponível em: <https://doi.org/10.1007/BF02650179>.

GARHWAL, S.; GHORANI, M.; AHMAD, A. Quantum programming language: A systematic review of research topic and top cited languages. **Archives of Computational Methods in Engineering**, Dec 2019. ISSN 1886-1784. Disponível em: <https://doi.org/10.1007/s11831-019-09372-6>.

GERHARDT, T. E.; SILVEIRA, D. T. **Métodos de pesquisa**. [S.l.]: Plageder, 2009.

GHEZZI, C.; JAZAYERI, M. **Programming Language Concepts**. 3rd. ed. USA: John Wiley &; Sons, Inc., 1997. ISBN 0471104264.

GIL, A. C. Como elaborar projetos de pesquisa. In: **Como elaborar projetos de pesquisa**. [S.l.: s.n.], 2002. p. 175–175.

GREEN, A. S. et al. Quipper. **ACM SIGPLAN Notices**, Association for Computing Machinery (ACM), v. 48, n. 6, p. 333–342, Jun 2013. ISSN 1558-1160. Disponível em: <http://dx.doi.org/10.1145/2499370.2462177>.

GROVER, L. K. Quantum mechanics helps in searching for a needle in a haystack. **Physical Review Letters**, American Physical Society (APS), v. 79, n. 2, p. 325–328, Jul 1997. ISSN 1079-7114. Disponível em: <http://dx.doi.org/10.1103/PhysRevLett.79.325>.

GYONGYOSI, L.; IMRE, S. A survey on quantum computing technology. **Computer Science Review**, v. 31, p. 51 – 71, 2019. ISSN 1574-0137. Disponível em: <http://www.sciencedirect.com/science/article/pii/S1574013718301709>.

HEIM, B. et al. Quantum programming languages. **Nature Reviews Physics**, v. 2, n. 12, p. 709–722, Dec 2020. ISSN 2522-5820. Disponível em: <https://doi.org/10.1038/s42254-020-00245-7>.

HIDARY, J. D. Development libraries for quantum computer programming. In: ____. **Quantum Computing: An Applied Approach**. Cham: Springer International Publishing, 2019. p. 61–79. ISBN 978-3-030-23922-0. Disponível em: <https://doi.org/10.1007/978-3-030-23922-0_6>.

HäNER, T. et al. A software methodology for compiling quantum programs. **Quantum Science and Technology**, IOP Publishing, v. 3, n. 2, p. 020501, Feb 2018. ISSN 2058-9565. Disponível em: <http://dx.doi.org/10.1088/2058-9565/aaa5cc>.

KNILL, E. Conventions for quantum pseudocode. 6 1996.

LAROSE, R. Overview and comparison of gate level quantum software platforms. **Quantum**, Verein zur Forderung des Open Access Publizierens in den Quantenwissenschaften, v. 3, p. 130, Mar 2019. ISSN 2521-327X. Disponível em: <http://dx.doi.org/10.22331/q-2019-03-25-130>.

MOORE, G. E. Cramming more components onto integrated circuits. **Electronics**, v. 38, n. 8, April 1965.

Moore, G. E. No exponential is forever: but "forever"can be delayed! [semiconductor industry]. In: **2003 IEEE International Solid-State Circuits Conference, 2003. Digest of Technical Papers. ISSCC.** [S.l.: s.n.], 2003. p. 20–23 vol.1.

MOSCA, M. **Quantum Algorithms**. 2008.

NIELSEN, M. A.; CHUANG, I. L. **Quantum Computation and Quantum Information: 10th Anniversary Edition**. [S.l.]: Cambridge University Press, 2010.

SANGHVI, N.; SHAH, A.; VARADAN, V. The concept and future of quantum computing. **International Journal of Computer Applications**, Citeseer, v. 106, n. 4, p. 30–33, 2014.

SEBESTA, R. W. **Concepts of programming languages**. [S.l.]: Boston: Pearson,, 2012.

SHOR, P. Algorithms for quantum computation: discrete logarithms and factoring. In: **Proceedings 35th Annual Symposium on Foundations of Computer Science**. [S.l.: s.n.], 1994. p. 124–134.

SILVA, W. J. N. da. **Uma introdução à Computação Quântica**. Tese (Doutorado) — Universidade de São Paulo, 2018.

SIMON, D. On the power of quantum computation. **SIAM Journal on Computing**, v. 26, n. 5, p. 1474–1483, 1997. Cited By 416. Disponível em: <https://www.scopus.com/inward/record.uri?eid=2-s2.0-0009438294&doi=10.1137%2fS0097539796298637&partnerID=40&md5=88ac08054b617f30c22e0d85c59ac2b8>.

SMITH, R. S.; CURTIS, M. J.; ZENG, W. J. **A Practical Quantum Instruction Set Architecture**. 2016.

STEWART, D. B. Measuring execution time and real-time performance. In: **In: Proceedings of the Embedded Systems Conference (ESC SF**. [S.l.: s.n.], 2002. p. 1–15.

SVORE, K. et al. Q: Enabling scalable quantum computing and development with a high-level dsl. In: **Proceedings of the Real World Domain Specific Languages Workshop 2018**. New York, NY, USA: Association for Computing Machinery, 2018. (RWDSL2018). ISBN 9781450363556. Disponível em: <https://doi.org/10.1145/3183895.3183901>.

SVORE, K. M. et al. A layered software architecture for quantum computing design tools. **Computer**, v. 39, n. 1, p. 74–83, 2006.

SVORE, K. M.; TROYER, M. The quantum future of computation. **Computer**, v. 49, n. 9, p. 21–30, 2016.

TUCKER, A.; NOONAN, R. **Linguagens de programação: princípios e paradigmas**. McGraw-Hill, 2009. ISBN 9788577260447. Disponível em: <https://books.google.com.br/books?id=v3OrPgAACAAJ>.

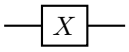
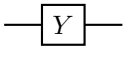
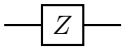
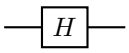
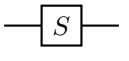
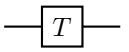
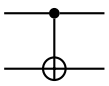
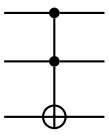
WAZLAWICK, R. S. **Metodologia de pesquisa para ciência da computação**. [S.l.]: Elsevier, 2009.

ZHAO, J. **Quantum Software Engineering: Landscapes and Horizons**. 2020.

ZIMEK, A.; FILZMOSER, P. There and back again: Outlier detection between statistical reasoning and data mining algorithms. **WIREs Data Mining and Knowledge Discovery**, v. 8, n. 6, p. e1280, 2018. Disponível em: <https://onlinelibrary.wiley.com/doi/abs/10.1002/widm.1280>.

ÖMER, B. **A procedural formalism for quantum computing**. [S.l.], 1998.

APÊNDICE A – PRINCIPAIS PORTAS QUÂNTICAS

Porta	Representação	Descrição	Matriz
Pauli-X		A porta Pauli-X é equivalente à porta NOT dos circuitos clássicos. Inverte-se o valor do *qubit*, transformando $\|0\rangle$ em $\|1\rangle$ e $\|1\rangle$ em $\|0\rangle$.	$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$
Pauli-Y		A porta Pauli-Y é utilizada para rotacionar em ângulo de 180 graus em torno do eixo Y na esfera de Bloch.	$\begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}$
Pauli-Z		A porta Pauli-Z é utilizada para rotacionar em ângulo de 180 graus em torno do eixo Z na esfera de Bloch.	$\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$
Hadamard		A porta Hadamard é utilizada para criar um estado de superposição em um *qubit*.	$\frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$
Phase		A porta S é utilizada para rotacionar em ângulo de 90 graus em torno do eixo Z na esfera de Bloch, possibilitando um universo de superposição na representação polar.	$\begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix}$
$\pi/8$		A porta T é utilizada para rotacionar em ângulo de 45 graus em torno do eixo Z na esfera de Bloch, possibilitando um universo de superposição na representação polar.	$\begin{bmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{bmatrix}$
Controlled NOT (CNOT)		A porta NOT controlada inverte o segundo *qubit* se o primeiro *qubit* estiver no estado $\|1\rangle$.	$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$
Toffoli		A porta Toffoli inverte o valor do terceiro *qubit* se os dois primeiros *qubits* estiverem no estado $\|1\rangle$.	$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$

Fonte: Elaborado pelo autor.

APÊNDICE B – EXEMPLO DE CÓDIGO DA BIBLIOTECA CIRQ

```
1
2  import random, cirq, time
3
4  def main(qubit_count):
5      circuit_sample_count = 3
6
7      input_qubits = [cirq.GridQubit(i, 0) for i in range(qubit_count)]
8      output_qubit = cirq.GridQubit(qubit_count, 0)
9
10     secret_bias_bit = random.randint(0, 1)
11     secret_factor_bits = list(map(int,list(endianNotation(qubit_count))))
12
13     oracle = make_oracle(input_qubits, output_qubit, secret_factor_bits,
       secret_bias_bit)
14     circuit = make_bernstein_vazirani_circuit(input_qubits, output_qubit,
       oracle)
15     simulator = cirq.Simulator()
16     result = simulator.run(circuit, repetitions=circuit_sample_count)
17     frequencies = result.histogram(key='result', fold_func=bitstring)
18     most_common_bitstring = frequencies.most_common(1)[0][0]
19
20 def make_oracle(input_qubits, output_qubit, secret_factor_bits,
       secret_bias_bit):
21
22     if secret_bias_bit:
23         yield cirq.X(output_qubit)
24
25     for qubit, bit in zip(input_qubits, secret_factor_bits):
26         if bit:
27             yield cirq.CNOT(qubit, output_qubit)
28
29
30 def make_bernstein_vazirani_circuit(input_qubits, output_qubit, oracle):
31     # Solves for factors in f(a) = a factors + bias (mod 2) with one query
       .
32
33     c = cirq.Circuit()
34
35     # Initialize qubits.
36     c.append(
37         [
38             cirq.X(output_qubit),
39             cirq.H(output_qubit),
40             cirq.H.on_each(*input_qubits),
41         ]
```

```
    )

    # Query oracle.
    c.append(oracle)

    # Measure in X basis.
    c.append([cirq.H.on_each(*input_qubits), cirq.measure(*input_qubits,
    key='result')])

    return c


def bitstring(bits):
    return ''.join(str(int(b)) for b in bits)
```

APÊNDICE C – EXEMPLO DE CÓDIGO DA BIBLIOTECA PYQUIL

```
1
2  from grove.bernstein_vazirani.bernstein_vazirani import BernsteinVazirani,
       create_bv_bitmap
3
4  from itertools import product
5  import pyquil.api as api
6  import time
7
8  qvm = api.QVMConnection()
9
10 bernsteinVazirani = BernsteinVazirani()
11
12 def generateBitMap(numberOfQubits) :
13     endianRep = "{0:b}".format(numberOfQubits)[::-1]
14     bitMapLength = len(endianRep)
15     zeroPadding = '0' * (numberOfQubits - bitMapLength)
16     endianRep = endianRep + zeroPadding
17     print("Bitmap: " + endianRep)
18     return endianRep
19
20 def runBernsteinVazirani(numberOfQubits) :
21     bitMap = generateBitMap(numberOfQubits)
22     bitMapFunction = create_bv_bitmap(bitMap, '1')
23     timeBefore = time.perf_counter()
24     bernsteinVazirani.run(qvm, bitMapFunction).get_solution()
25     timeAfter = time.perf_counter()
26     totalElapsedTime = timeAfter - timeBefore
27     formattedTotalElapsedTime = "{:.4f}".format(totalElapsedTime)
28     print("[" + str(numberOfQubits) + "] Time: " + str(
       formattedTotalElapsedTime))
```

APÊNDICE D – EXEMPLO DE CÓDIGO DA LINGUAGEM Q#

```
namespace BernsteinVaziraniNamespace {

    open Microsoft.Quantum.Canon;
    open Microsoft.Quantum.Intrinsic;
    open Microsoft.Quantum.Measurement;
    open Microsoft.Quantum.Arrays;
    open Microsoft.Quantum.Convert;

    operation LearnParityViaFourierSampling(Uf : ((Qubit[], Qubit) => Unit)
    , n : Int) : Bool[] {
        use queryRegister = Qubit[n];
        use target = Qubit();

        X(target);

        within {
            ApplyToEachA(H, queryRegister);
        } apply {
            H(target);
            Uf(queryRegister, target);
        }

        let resultArray = ForEach(MResetZ, queryRegister);
        Reset(target);
        return ResultArrayAsBoolArray(resultArray);
    }

    internal operation ParityOperationImpl(pattern : Bool[], queryRegister
    : Qubit[], target : Qubit) : Unit {
        if (Length(queryRegister) != Length(pattern)) {
            fail "Length of input register must be equal to the pattern
    length.";
        }

        for (patternBit, controlQubit) in Zipped(pattern, queryRegister) {
            if (patternBit) {
                Controlled X([controlQubit], target);
            }
        }
    }

    function ParityOperation(pattern : Bool[]) : ((Qubit[], Qubit) => Unit)
    {
        return ParityOperationImpl(pattern, _, _);
    }
```

```
42
43      operation RunBernsteinVazirani (nQubits : Int, patternInt : Int) : Int
        {
44          let pattern = IntAsBoolArray(patternInt, nQubits);
45
46          let result = LearnParityViaFourierSampling(ParityOperation(pattern)
        , nQubits);
47          return BoolArrayAsInt(result);
48      }
49  }
```

APÊNDICE E – EXEMPLO DE CÓDIGO DA BIBLIOTECA QISKIT

```
import matplotlib.pyplot as plt
import numpy as np
import time
from qiskit import IBMQ, Aer
from qiskit.providers.ibmq import least_busy
from qiskit import QuantumCircuit, ClassicalRegister, QuantumRegister,
    transpile, assemble
import qiskit
from qiskit.visualization import plot_histogram

backendObject = Aer.get_backend('qasm_simulator')

def buildBernsteinVaziraniCircuit(n, bitMask) :

    s = bitMask
    print(s)

    bv_circuit = QuantumCircuit(n+1, n)

    bv_circuit.h(n)
    bv_circuit.z(n)

    for i in range(n):
        bv_circuit.h(i)

    bv_circuit.barrier()

    s = s[::-1] # reverse s to fit qiskit's qubit ordering
    for q in range(n):
        if s[q] == '0':
            bv_circuit.i(q)
        else:
            bv_circuit.cx(q, n)

    bv_circuit.barrier()

    for i in range(n):
        bv_circuit.h(i)

    for i in range(n):
        bv_circuit.measure(i, i)

    return bv_circuit

def runBernsteinVaziraniAlgorithm(numberOfQubits) :
```

```
45      bitMask = generateEndianBitMap(numberOfQubits)
46      bernsteinVaziraniCircuit = buildBernsteinVaziraniCircuit(numberOfQubits
        , bitMask)
47      quantumObject = assemble(bernsteinVaziraniCircuit)
48      results = backendObject.run(quantumObject).result()
```