

Quantum Approaches to Sequence Alignment

Vivatsathorn Thitasirivit 6432158421

2023/1 2110581 Bioinformatics I

Department of Computer Engineering,
Faculty of Engineering, Chulalongkorn University

Table of Contents

1. Backgrounds	1
2. Related Research	2
2.1. Research 1	2
2.1.1. Research Backgrounds	2
2.1.2. Approaches & Methodology	2
2.1.3. Results & Discussions	3
2.2. Research 2	4
2.2.1. Research Backgrounds	4
2.2.2. Approaches & Methodology	4
2.2.3. Results & Discussions	4
2.3. Research 3	5
2.3.1. Research Backgrounds	5
2.3.2. Approaches & Methodology	5
2.3.3. Results & Discussions	5
3. Bioinformatics and Classical Algorithms	6
3.1. Sequence Alignment	6
3.1.1. Global Alignment	7
3.1.2. Local Alignment	7
4. Classical Algorithms Parallelization Techniques	8
4.1. Sequential NW Algorithm	8
4.2. Parallel NW Algorithm	9
4.2.1. Block-parallel NW Algorithm	10
4.2.2. Block-parallel Implementation in CPU	12
4.2.3. Block-parallel Implementation in GPU	13
4.2.4. Block mapping Technique	14
5. Quantum Computing and Quantum Algorithms	15
5.1. Fundamental Quantum Gates	16
5.2. Grover's Algorithm	17
5.3. Quantum Fourier Transform	17
6. Proposed Methods	18
6.1. Sequence Encoding	19
6.2. Quantum Analogous	19
6.2.1. Time-domain comparison with XOR	19
6.2.2. Utilizing quantum adder for XOR comparison	21
6.2.3. Time-domain (computational basis) comparison with Swap Test	21
6.2.4. Frequency-domain comparison with Swap Test	23
6.2.5. Swap Test comparison with dual domain (Combination)	25
6.2.6. Time-domain Permutation Swap Test	25
6.2.7. Quantum Phase Estimation for frequency domain	26
7. Comparison with Related Researches	26
8. Alternative Approaches	27
8.1. Qudit Encoding	27
9. Quantum Circuit Generation with IBM Qiskit	28
9.1. About <code>ibm_brisbane</code>	29
9.2. Preparing Basic Operators	30
9.2.1. String Encoding	30

9.2.2.	State Initialization	30
9.2.3.	n -bit Quantum XOR	31
9.2.4.	n -bit Swap Test	31
9.2.5.	Initialize Basic Operators	31
9.3.	Time-domain comparison with XOR	32
9.4.	Time-domain comparison with Swap Test	32
9.5.	Frequency-domain comparison with Swap Test	33
9.6.	Dual domain comparison with Swap Test	34
10. Quantum Algorithm Results & Discussion		35
10.1.	Quantum Algorithm Simulation	35
10.1.1.	16-bit Time-domain comparison with XOR	36
10.1.2.	16-bit Time-domain comparison with Swap Test	37
10.1.3.	16-bit Frequency-domain comparison with Swap Test	38
10.1.4.	32-bit Frequency-domain comparison with Swap Test	40
10.1.5.	8x8 16-bit Frequency-domain comparison with Swap Test	42
10.1.6.	8-bit Dual domain comparison with Swap Test	43
10.2.	Results on Quantum Device	44
10.2.1.	8-bit String Comparison	44
10.2.2.	16-bit String Comparison	45
10.2.3.	32-bit String Comparison	46
10.2.4.	40-bit String Comparison	47
10.3.	Complexity Analysis	48
10.3.1.	Initialization and Transformation	48
10.3.2.	Comparison Operators	48
10.3.3.	Quantum Fourier Transform	48
10.3.4.	Overall Analysis	48
10.4.	Noise Analysis	49
10.4.1.	Qubit Width and SNR Comparison	49
10.4.2.	Logical Barrier and Circuit Optimization	50
10.4.3.	Sample Size	51
10.5.	Results on Quantum Device (Improved)	52
10.5.1.	Improved 8-bit String Comparison Test	52
10.5.2.	Improved 16-bit String Comparison Test	52
10.5.3.	Improved 32-bit String Comparison Test	53
10.5.4.	Improved 40-bit String Comparison Test	53
11. Summary		54
12. Further Research		55
12.1.	Mitigating Noise	55
12.2.	Interpreting QFT Result	55
12.3.	Alternative Problem Statements	56
12.4.	Quantum RNA Folding	56
12.5.	Motif Finding	57
12.5.1.	Algorithm Suitability Rationales	57
12.5.2.	Method Adaptation for Motif Finding	57
12.5.3.	Challenges in Quantum Motif Finding	58
12.5.4.	Proposed Quantum Algorithms	58
12.5.5.	Impact on Genomic Research	58
A Quantum Circuits		61
1.1.	16-bit Time-domain comparison with XOR	61
1.2.	16-bit Time-domain comparison with Swap Test	62
1.3.	16-bit Frequency-domain comparison with Swap Test	63
1.4.	8-bit Dual domain comparison with Swap Test	64
B Classical Algorithms C Source Code		65

2.1.	<code>main.c</code>	65
2.2.	<code>bio_cpu.h</code>	68
2.3.	<code>bio_cpu.c</code>	70
2.4.	<code>fasta.h</code>	77
2.5.	<code>fasta.c</code>	78

List of Figures

1	Sequential matrix sweeping	8
2	F's Cell dependency (Blue = $F[i][j]$, Green = Dependent cells)	9
3	F's Wavefront anti-diagonal (Blue = Current wavefront, Green = Dependent cells, Red = To-be-completed cells)	9
4	F's Wavefront anti-diagonal iterations	10
5	F split into blocks	10
6	Access pattern within block	11
7	Block access pattern and dependency	11
8	Block access pattern (Anti-diagonal)	12
9	Classical algorithms comparison	13
10	Very Large Array Block Decomposition	14
11	Block Indexing Array	14
12	Typical XOR Operation	19
13	XOR with output qubit	20
14	Time-domain comparison with XOR	20
15	XOR with output qubit	20
16	1-bit Test Swap Quantum Circuit	21
17	Time-domain comparison with Swap Test	22
18	Frequency-domain comparison with Swap Test	25
19	Time-and-Frequency-domain comparison with Swap Test	25
20	Generalized Quantum Phase Estimation Circuit	26
21	<code>ibm_brisbane</code> 's lattice structure (Courtesy: <i>IBM Quantum Platform</i>)	29
22	Quasiprobability Distribution	36
23	Quasiprobability Distribution	37
24	Quasiprobability Distribution	38
25	Quasiprobability Distribution	39
26	Quasiprobability Distribution	39
27	Quasiprobability Distribution	40
28	Quasiprobability Distribution (Indistinguishable)	41
29	8×8 Quasiprobability Distribution	42
30	Quasiprobability Distribution	43
31	8-bit String Running Results Comparison	44
32	16-bit String Running Results Comparison	45
33	32-bit String Running Results Comparison	46
34	40-bit String Running Results Comparison	47
35	8-bit String Running Results Mode Comparison	50
36	8-bit String Running Results Shots Comparison	51
37	Quasiprobability Distribution Comparison (8 bits)	52
38	Quasiprobability Distribution Comparison (16 bits)	52
39	Quasiprobability Distribution Comparison (32 bits)	53
40	Quasiprobability Distribution Comparison (40 bits)	53
41	Full Circuit (Decomposed 1 level)	61
42	Full Circuit (Decomposed 1 level)	62
43	Full Circuit (Decomposed 1 level)	63
44	Full Circuit (Top-level)	64

List of Tables

1	SNR Comparison between 8, 16, 32 and 40 bits	49
2	SNR Comparison from 2 modes (8-bit)	50
3	SNR Comparison from 2 shots (8-bit)	51

1. Backgrounds

Sequence Alignment is a fundamental procedure in bioinformatics, particularly in comparing DNA, RNA, or protein sequences to identify similar regions which can denote functional and structural relationships among the sequences. Classical algorithms: Smith-Waterman for local alignment and Needleman-Wunsch for global alignment, which both have $O(mn)$ time complexity, have provided sufficient optimizations for classical computing. However, as biological databases continue to expand exponentially, more computational power is required significantly.

Quantum computers utilize superposition, quantum entanglement, interference, quantum annealing, and other properties to solve specific problems with their probabilistic approaches and their parallelism by nature computationally faster than classical computers.

In quantum computing algorithms, there are many mechanics and algorithm implementations that can be applied to a classical algorithm to gain speed ups by taking advantage of its physical parallelism (which is different from classical architectural parallelism), where computations are performed under state of quantum coherence. The final results can be achieved by observation of probabilistic coherence states of a system (a computational problem), which the action will cause a quantum decoherence, namely wave function collapsing. As stated, there are many “flavors” of quantum mechanics applied in the process: Black-Box, Oracle, Quantum Fourier Transform, Search algorithm, Quantum Annealing (adiabatic computation), State Optimization, etc. There are also various quantum hardware technologies: superconducting qubits, trapped ions, quantum dots, Nuclear Magnetic Resonance, and many other researches underway.

Grover’s algorithm is one of the most significant algorithms primarily designed for database searching (finding inputs matching output of a function), reducing analogous search time from $O(n)$ to $\Omega(\sqrt{n})$ time.

Researchers have been exploring quantum algorithms’ potential in bioinformatics fields, e.g., sequence reconstruction, sequence alignment, multiple sequence alignment. Quantum supremacy promises speed-ups which could significantly reduce computational time for solving specific problems compared to classical computation, making large-scale sequence comparison more feasible. This involves encoding sequence data in quantum states harnessing quantum properties to search the solution space more efficiently.

Nevertheless, in the present time, quantum approaches have a number of challenges. Quantum computers hardware and implementations are still relatively new, with prominent noise and error rates that can impact computation results. Consequently, implementing quantum algorithms pose many challenges in the meantime; however, as the field matures, the possibility will become higher and more promising.

2. Related Research

2.1. Research 1

Quantum Pattern Recognition for Local Sequence Alignment
Prousalis K., & Konofaos N. (2017).

2.1.1. Research Backgrounds

As common backgrounds to many researches harnessing quantum computing power to classical algorithms, this research utilizes fundamental technique: Quantum Fourier Transform (QFT) and more applied technique: Schützhold's quantum pattern recognition algorithm,¹ which the latter is relatively new in the field. Those quantum algorithm blueprints are applied to speed up classical Smith-Waterman algorithm for sequence local alignment.

2.1.2. Approaches & Methodology

The authors utilized classical Smith-Waterman algorithm, QFT and Schützhold's quantum pattern recognition algorithm. The concept is to create an unstructured black and white cells for Smith-Waterman's substitution matrix and uses pattern recognition to recognize and locate desired regions.

Over recent years, there's been an increasing interest in quantum pattern recognition due to its potential in locating and identifying certain patterns within unstructured data sets. Quantum methods offer a means to analyze large databases of biological data generated from genome projects, and sequence alignment, a process of arranging sequences of discrete objects to identify similarities, becomes a significant focus.

Using a spatial light modulator, an incoming optical mode is mapped to thousands of outgoing optical modes over a complex 2-dimensional image. This image contains the unstructured data set. Quantum algorithms are used to identify specific patterns, employing quantum parallelism and the QFT to enhance sequence alignment processes. The classical Smith-Waterman algorithm's working was improved version through the integration of Schützhold's pattern recognition quantum algorithm.

¹<https://arxiv.org/abs/quant-ph/0208063>

2.1.3. Results & Discussions

1. The QFT has been identified as a critical tool in quantum algorithms. It is the quantum equivalent of the classical Discrete Fourier Transform and finds use in several quantum algorithms.
2. The Schützhold's quantum algorithm greatly improves the efficiency of the Smith-Waterman algorithm for local alignment.
3. The complexity analysis suggests that while the Smith-Waterman algorithm has its own challenges and costs, integrating it with quantum algorithms, especially the proposed method, makes it exponentially faster than the classical version.
4. However, a potential drawback is the necessity to load the entire data set into a quantum memory, which might pose challenges.

2.2. Research 2

A Linear Time Quantum Algorithm for Pairwise Sequence Alignment

Khan, Md. R. I., Shahriar, S., & Rafid, S. F. (2023).

<http://arxiv.org/abs/2307.04479>

2.2.1. Research Backgrounds

Recently, quantum computing has offered promising avenues for speeding up sequence alignment. Techniques such as the QiBAM, which leverages Grover's algorithm for a quadratic speedup, and methods that combine dot-matrix plotting with quantum pattern recognition, have been proposed.

2.2.2. Approaches & Methodology

The authors proposed "Graph-Based Quantum Alignment" method. For a pair of sequences, a 2D graph is created based on methodologies from Needleman and Wunsch (2002). The next step involves generating paths through this graph using the "Edit Distance" metric. The goal is to find a path with minimal mismatches. A unique method was made to calculate the path cost or profit by assigning values to each edge of the graph. The path with the highest profit corresponds to the optimal path. The Grover's search algorithm is employed to identify the optimal alignment.

2.2.3. Results & Discussions

1. The proposed quantum algorithm achieved alignment in linear time.
2. The algorithm consistently identified the optimal alignment for DNA sequences, surpassing the accuracy and precision of current randomized and heuristic approaches.
3. A quadratic speed-up relative to deterministic algorithms was observed (multiplicative mn to additive (linear, $m + n$)).

2.3. Research 3

Multi-sequence alignment using the Quantum Approximate Optimization Algorithm

Madsen, S. Y., Marqversen, F. K., Rasmussen, S. E., & Zinner, N. T. (2023).

<http://arxiv.org/abs/2308.12103>

2.3.1. Research Backgrounds

This research utilizes unique approaches for alignment multiple sequences at once, generalizing traditional pairwise alignment methodology. For example, Ising model, a mathematical model of ferromagnetism for a quantum system, and optimized quantum circuitry are used. It also has potentials to be developed using adiabatic computing: quantum annealing/QA for more speed up. Classical approximation approach is also transformed into quantum problem, thus making the algorithm faster compared to solving for exact solution.

2.3.2. Approaches & Methodology

The goal was to encode the Multi-sequence alignment (MSA) cost within the Hamiltonian operator of a quantum system. The cost function was transformed into a Quadratic Unconstrained Binary Optimization (QUBO) model. This model, suitable for binary optimization, was then mapped to the Ising model, aligning it with quantum computation standards. A problem-specific cost function was mapped to a Hamiltonian and then encoded within a Parameterized Quantum Circuit (PQC). Classical optimizers were employed to navigate the quantum cost landscape, trying to minimize the cost function iteratively. Quantum Approximate Optimization Algorithm (QAOA), a specific Variational Quantum Annealing (VQA), was utilized, designed primarily for combinatorial optimization problems. It is inspired by the principles of quantum annealing.

2.3.3. Results & Discussions

1. A native 2-sequence example of MSA was used as a toy model to illustrate the implementation of the QAOA. Probability distributions from simulations of this model, when layered repetitively, showed potential promise, especially with increasing repetitions.
2. When tested on actual quantum hardware, results differed. The distributions exhibited significant noise, suggesting that present-day hardware might not be sufficient for high-probability sampling using the QAOA designed for MSA.
3. Results from the Aspen M-3 quantum hardware showed a substantial level of randomness, emphasizing the presence of noise and the need for better circuit-compilation techniques.

3. Bioinformatics and Classical Algorithms

3.1. Sequence Alignment

Sequence alignment is a method for arranging, measuring and identifying the similarity between two string sequences (pairwise). To compare multiple sequences, multiple sequence alignment (MSA) process is used. The results of DNA, RNA and protein alignment may reveal functional, structural and evolutionary relationship between origin sequences in the sequences' similar regions.

There are different schemes for measuring sequence similarity by alignment types: local alignment and global alignment. Local alignment only matches the region with highest similarity while the latter attempts to match the similarity of the whole sequences. The local and global alignment processes are commonly used in closely related sequences and more divergent sequences respectively.

The common classical methods for both alignment are *Needleman-Wunsch* (NW) algorithm for global alignment and *Smith-Waterman* (SW) algorithm for local alignment, which both are optimization problem utilizing dynamic programming techniques. The scoring matrices with match score, mismatch score and gap score (indel) is used in the score calculation filling the score matrix. One of the differences between two algorithms is that NW algorithm allows negative values while SW algorithm only allows the minimum value of zero.

There are also some similar approaches for string matching problems which are a superset of this particular sequence alignment problem. Some solutions can find exact optimal position and the best score, while some can only approximate them or solve partially.

Edit Distance is one of the methods used for string matching problems. There are many variants of this method, e.g., Hamming Edit Distance, Levenshtein Edit Distance, Graph Edit Distance, etc. *Pattern Matching* is also one of the methods similar to string matching problems.

3.1.1. Global Alignment

The concept of NW algorithm has three steps: initialization, matrix F filling and backtracking while:

$$\begin{aligned} F(0, j) &= dj \\ F(i, 0) &= di \\ F(i, j) &= \max \begin{cases} F(i - 1, j - 1) + S(A_i, B_j) \\ F(i, j - 1) + d \\ F(i - 1, j) + d \end{cases} \end{aligned} \quad (1)$$

and

$$S(a, b) = \begin{cases} c_1 & \text{if } a = b, \\ -c_2 & \text{if } a \neq b. \end{cases} \quad (2)$$

Usually the algorithm's preset values are $c_1 = c_2 = 1$ and $d = -2$. Moreover, the algorithm can be improved from $O(mn)$ time to $O(n)$ using wavefront anti-diagonal parallelization and do each wavefront in parallel using GPU computation blocks.

3.1.2. Local Alignment

The local alignment method uses SW algorithm which is similar to the NW algorithm, but restricts negative values to zero whenever encountered. Its complexity is also $O(mn)$ and $O(n)$ time in serial and parallel computation.

$$\begin{aligned} F(0, j) &= 0 \\ F(i, 0) &= 0 \\ F(i, j) &= \max \begin{cases} F(i - 1, j - 1) + S(A_i, B_j) \\ F(i, j - 1) + d \\ F(i - 1, j) + d \\ 0 \end{cases} \end{aligned} \quad (3)$$

and

$$S(a, b) = \begin{cases} c_1 & \text{if } a = b, \\ -c_2 & \text{if } a \neq b. \end{cases} \quad (4)$$

4. Classical Algorithms Parallelization Techniques

This section is an extra part of exploring pairwise sequence alignment algorithms in both quantum and classical approaches.

For classical algorithms comparison, algorithms with dynamic programming approach, i.e., Needleman-Wunsch, Smith-Waterman and Levenshtein Edit Distance algorithms, will be stated. These algorithms are very similar in terms of cell dependency and problem formulations, so Needleman-Wunsch (NW) algorithm will be used as a basic template.

4.1. Sequential NW Algorithm

In classical NW algorithm, the backtracking can be completed in $O(n)$ time and the matrix filling state can be completed in $O(n^2)$ time. This raises the question whether the algorithm can be parallelized and how. Fig. 1 illustrates on basic sequential sweeping throughout the matrix filling phase. The obvious problem was if the score matrix size is large enough to not fit in the CPU's cache, the memory access would be very slow, lacking spatial locality.

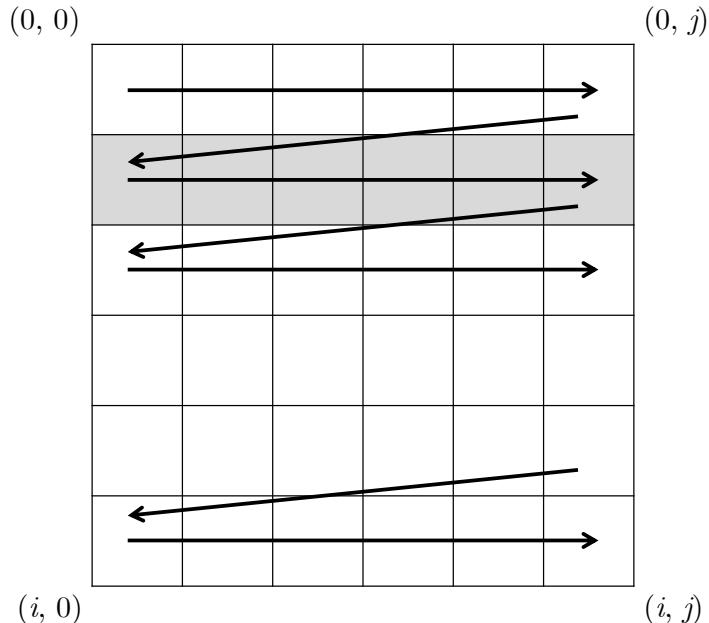


Figure 1: Sequential matrix sweeping

4.2. Parallel NW Algorithm

In the matrix F filling state, we can observe that the cell $F[i][j]$ is dependent of $F[i-1][j-1]$, $F[i-1][j]$ and $F[i][j-1]$, which makes the traversal direction be the diagonal of F as shown in Fig. 2.

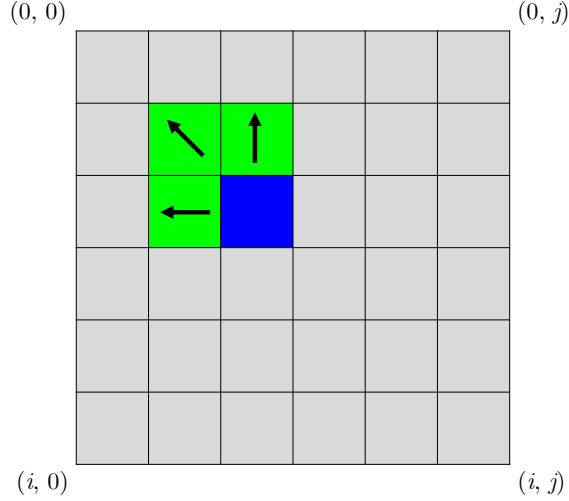


Figure 2: F 's Cell dependency
(Blue = $F[i][j]$, Green = Dependent cells)

Therefore, the wavefront perpendicular to the traversal direction is the anti-diagonal of F . Fig. 3 demonstrates wavefront anti-diagonal traversal and dependency when it is in 5th iteration. Overall iterations are shown in Fig. 4.

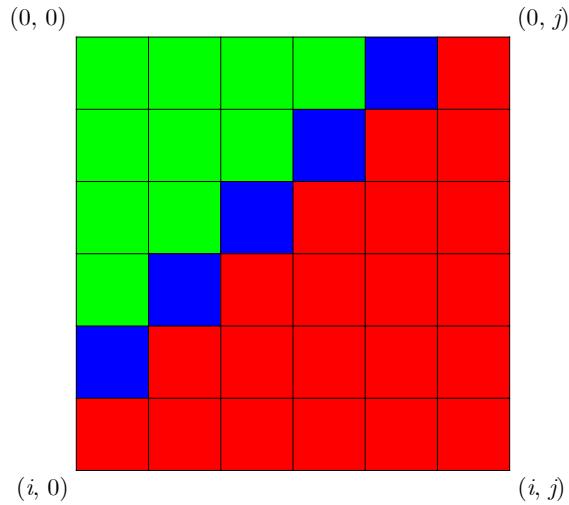


Figure 3: F 's Wavefront anti-diagonal (Blue = Current wavefront, Green = Dependent cells, Red = To-be-completed cells)

Those areas in the wavefront can be theoretically parallelized with enough threads and the threads can join if and only if all operations on elements in the wavefront are completed as whole as the next wavefront depends on them.

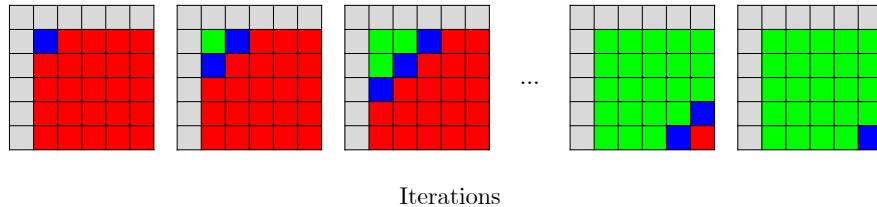


Figure 4: F's Wavefront anti-diagonal iterations

Although the wavefront anti-diagonal's elements can be directly parallelized, it would still cause the cache's spatial locality problem as the CPU has to load the whole matrix in a cache but it would not fit if the matrix is too large.

Also, the overhead cost to spawn those threads is much higher than the actual operations in the loop which do simple addition, score function lookup and comparison.

4.2.1. Block-parallel NW Algorithm

To fix both of the stated issues: cache and overhead, you can split the computation into blocks as shown in Fig. 5.

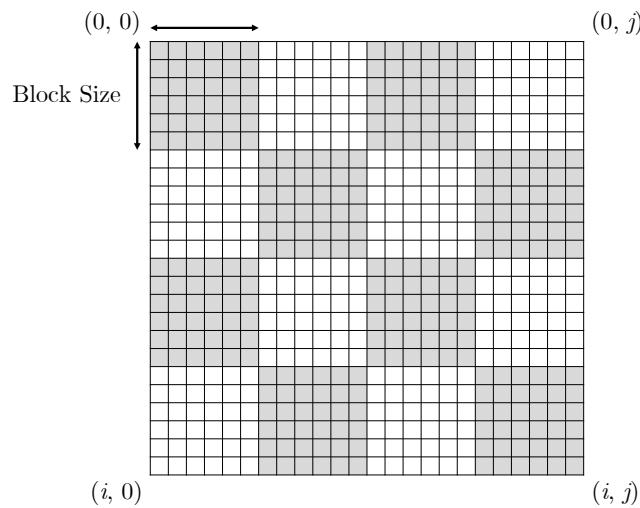


Figure 5: F split into blocks

This only solves the cache's spatial locality problem as only a block of size smaller than the set threshold `MAX_BLOCK_SIZE` is loaded into the cache at that time, so it is less likely to cause cache misses.

Within a block, the sweeping does not have to be wavefront as it does not have to be parallelized due to thread overheads over small computations. Letting the compiler optimize sequential access can make the overall block operation runs faster. The access pattern within a block is shown in Fig. 6.

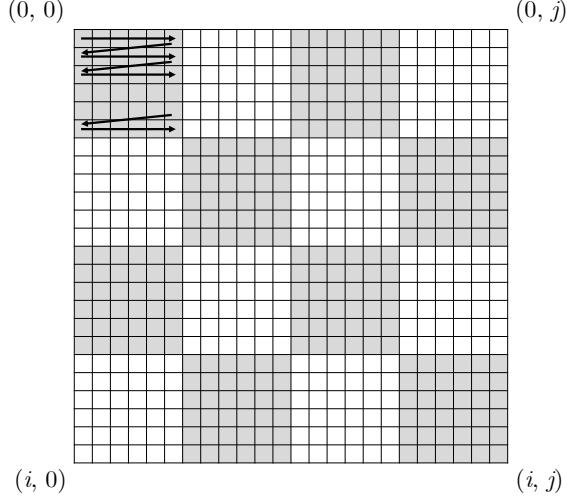


Figure 6: Access pattern within block

To improve it further, the task is now large enough to be parallelized without overhead and cache locality problem. To parallelize without violating dependency, we can layout blocks in wavefront anti-diagonal in the same manner as previously proposed.

Fig. 7 shows the block access pattern. The magenta-colored regions illustrate the current wavefront blocks' dependency of the previously finished blocks, hence not violating the cells dependency.

In some architectures, especially in modern architectures, there are cache blocks that are not shared between threads and such cache memory is typically larger.

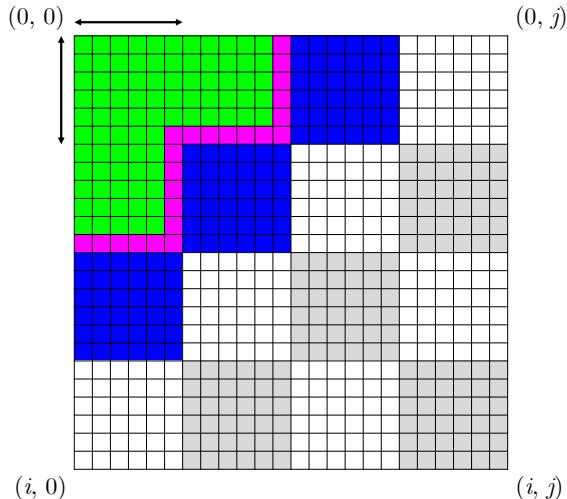


Figure 7: Block access pattern and dependency

4.2.2. Block-parallel Implementation in CPU

To implement a block-parallel algorithm, in this case: using CPU threading in C and OpenMP's for loop parallelization, we have to access blocks in the diagonals in 2D (N^2) and access elements within blocks in 2D (M^2). Total number of accesses would be still in n^2 like in sequential counterpart, but theoretically faster because of simultaneous access.

```

for diagonal in all_diagonals do
|
| < Begin Parallelize for region >
| for block in diagonal do
| |
| | for i in block_rows do
| | | for j in block_cols do
| | | | < Matrix filling sequential algorithm >
| | | end
| | end
| end
| < End Parallelize for region >
|
end

```

Fig. 8 shows the access pattern described by the pseudocode.

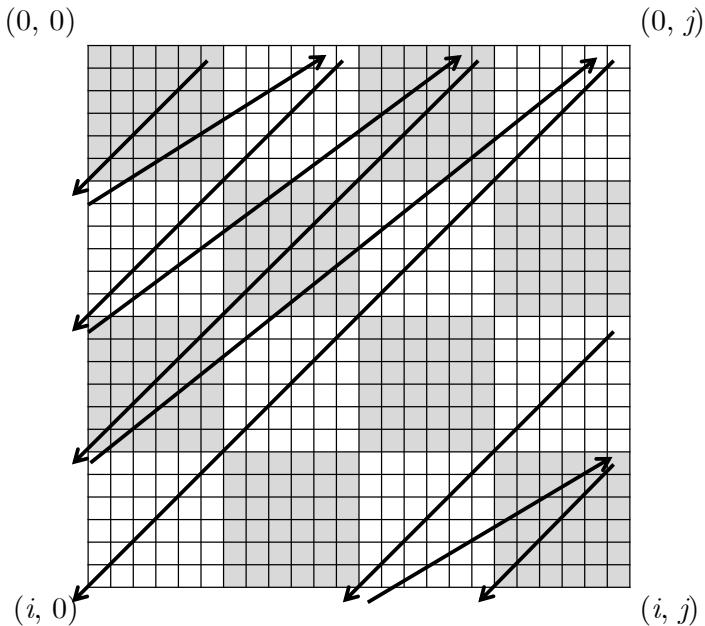


Figure 8: Block access pattern (Anti-diagonal)

Here's the result of comparison of various methods and techniques on algorithm running on CPU (Fig. 9).

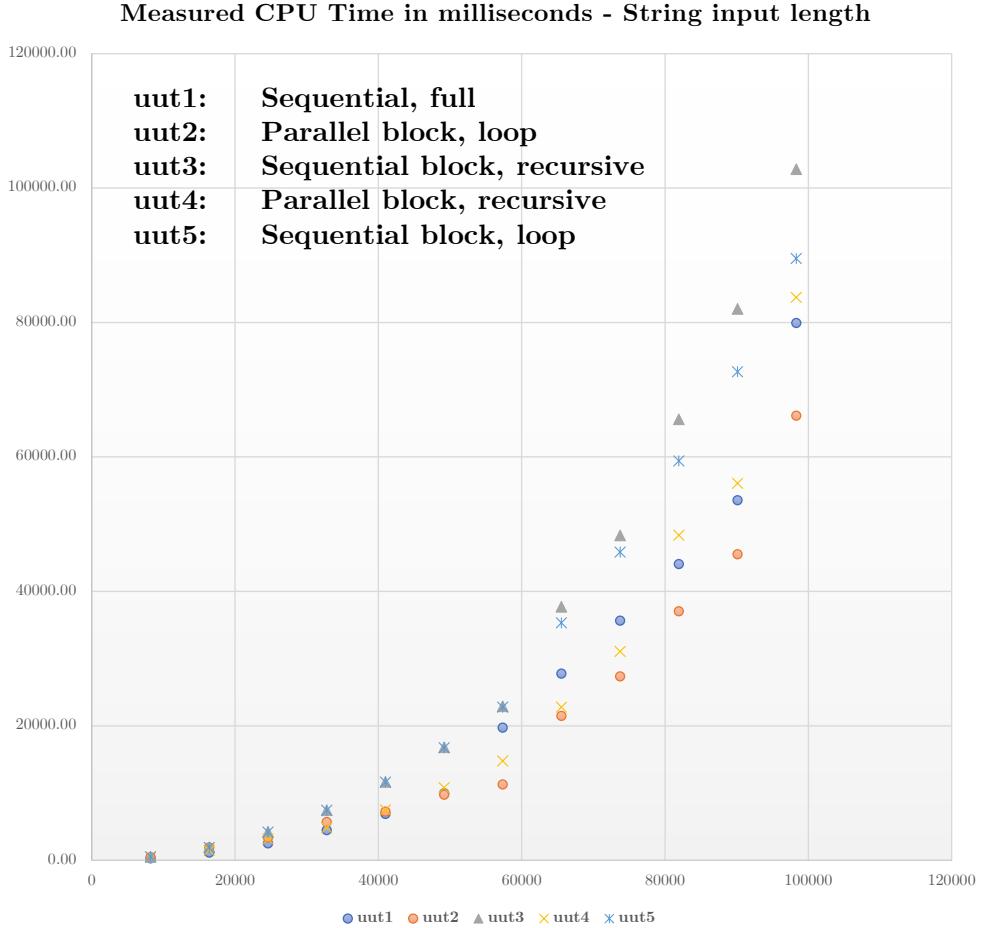


Figure 9: Classical algorithms comparison

4.2.3. Block-parallel Implementation in GPU

AMD's OpenCL and NVIDIA's CUDA framework can be used to parallelize the algorithm by loading the work into GPU memory and compute using defined kernels. There will be some limitations in terms of memory and manual paging technique which is similar to swap region used by an operating system. For blocks that are currently in the computation: current waveform and previous waveform, they can be loaded into the system memory. For unused/finished/unfinished blocks, they can be offloaded to the disk.

This reduces the simultaneous memory usage dramatically from quadratic space to linear space. During the backtracking, this technique can also be used. Computers with low memory can also utilize this technique to process large strings' pairwise alignment. However, this technique contains disk access which is much slower than memory access, so it has some tradeoffs for speed, but still enables parallelization which recursive dynamic programming approach cannot provide.

4.2.4. Block mapping Technique

As the scoring matrix is dense and not sparse, changing matrix's data structure is not applicable; therefore, other techniques were used. To implement memory offloading, a large array that is unable to be stored in the main memory can be decomposed to smaller blocks, preferably with the same size as the computational blocks as shown in Fig. 10. This way, each block can be stored separately. In conjunction with block-parallel computing, only blocks currently in the computation and referenced blocks are loaded at a time.

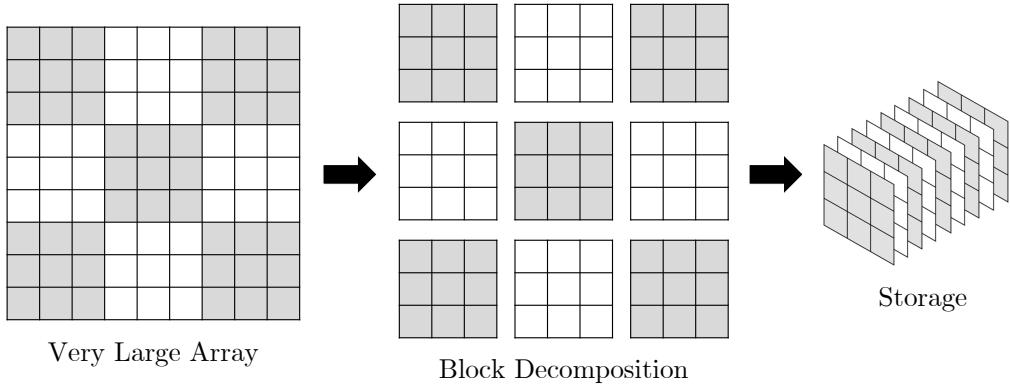


Figure 10: Very Large Array Block Decomposition

Subsequently, an index array is introduced for indexing with target as a pointer to each block header (the first element in each block) as shown in Fig. 11.

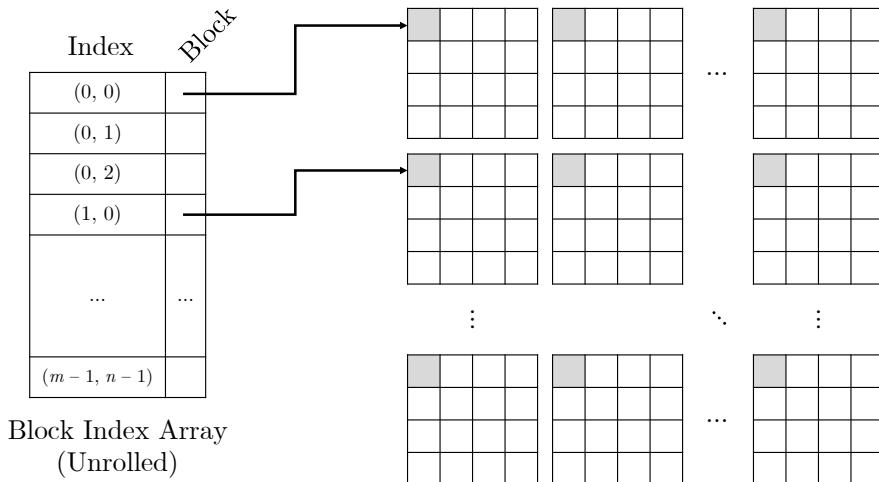


Figure 11: Block Indexing Array

5. Quantum Computing and Quantum Algorithms

A classical computer uses voltages flowing through circuits and logical gates which is mainly controlled by classical mechanics principles. A quantum computer, on the other hand, uses various properties of quantum mechanics, e.g., superposition, entanglement, coherence and decoherence, to perform naturally parallel computations.

Similar to classical computers which use *bits* as a foundation, quantum computers use *qubits*. By mathematical definition, a qubit lies in a 2-dimensional Hilbert space: \mathcal{H}_2 where the orthonormal basis is $\{|0\rangle, |1\rangle\}$ which are two basic 1-qubit bases. A qubit is, therefore, a unit vector in \mathcal{H}_2 . If a state is a basis and not a linear combination of itself or other bases, then the state is a *pure state* or in *superposition*. Otherwise, it is a *mixed state*. A state (qubit) can be written as a linear combination of $|0\rangle$ and $|1\rangle$ as follows.

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle ; \alpha, \beta \in \mathbb{C} \text{ and } \|\alpha\|^2 + \|\beta\|^2 = 1. \quad (5)$$

$\|\alpha\|^2$ and $\|\beta\|^2$ describes probability of the state being either state in a superposition. The fundamental states are as follows.

$$\begin{aligned} |0\rangle &= \begin{pmatrix} 1 \\ 0 \end{pmatrix} \\ |1\rangle &= \begin{pmatrix} 0 \\ 1 \end{pmatrix} \end{aligned} \quad (6)$$

The quantum coherence and decoherence principles are that if a coherent state $|\psi\rangle$ is measured or interacted with external environment, the state would collapse (decoherence) into a pure state. A multi-qubit system is very much like a classical n -bit system:

$$|q\rangle = |q_1, \dots, q_n\rangle = \bigotimes_{i=1}^n \alpha_i |0\rangle + \beta_i |1\rangle \in \mathbb{C}^{2^n} \quad (7)$$

A qubit can be manipulated (transformed) by unitary matrices U (which is equivalent to a quantum gate in a quantum computer) where $U^\dagger U = I$ and $\|U|\psi\rangle\| = \||\psi\rangle\|$.

Quantum circuit model is an extension to a classical Boolean circuit implementing qubits, reversible unitary gates and measurement (wave collapse). There are several assumptions to be made.

1. A quantum computer is an extension to a classical computer which is responsible for state preparation and gates (operators) controlling.
2. $|0\rangle$ and $|1\rangle$ maps correspondingly to classical bit 0 and 1.
3. Gates can be applied to any groups of qubits, and there exists a universal set of gates (fundamental blocks) implemented by the hardware.
4. Measurement can be made at any point, which collapses one or more qubits in the process.

5.1. Fundamental Quantum Gates

Name	Circuit notation	Matrix
Identity		$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$
Hadamard		$\frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$
Pauli-X (Bit flip)		$\sigma_X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$
Pauli-Y		$\sigma_Y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}$
Pauli-Z (Phase flip)		$\sigma_Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$
P-Rotation		$e^{-i \frac{\theta P}{2}} = \cos \frac{\theta}{2} I - i \sin \frac{\theta}{2} \sigma_P$
Controlled-NOT (CNOT, CX)		$\begin{pmatrix} I & 0 \\ 0 & X \end{pmatrix}$
Controlled-U (CU)		$\begin{pmatrix} I & 0 \\ 0 & U \end{pmatrix}$
SWAP		$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$
Toffoli gate (CCNOT)		
Fredkin gate (CSWAP)		

5.2. Grover's Algorithm

Grover's algorithm is a quantum algorithm for “unstructured search problems” offering quadratic improvement over classical algorithms. With current technology of quantum hardware, compared to classical hardware, the algorithm does not likely have practical advantages over classical computing. However, Grover's algorithm has many potential use cases with advanced technology in the future.

To implement the Grover's algorithm in high-level manner, first, prepare the search state $|\psi_0\rangle = |0_1, \dots, 0_n\rangle$ to a uniform superposition of all possible states by applying each qubit with Hadamard gate.

$$|s\rangle = \frac{1}{\sqrt{N}} \sum_{i=0}^{N-1} |i\rangle = \left(\bigotimes_{i=1}^n H_i \right) |0_1, \dots, 0_n\rangle, \text{ where } N = 2^n. \quad (8)$$

Then, the search oracle $U_f : (-1)^{O(i)|i\rangle}$ inverts the phase of state $|i\rangle$, which marks the correct search state. Finally, the diffuser operator inverts back the state while amplifying the inverted state, so that the amplitude is higher.

$$U_s = 2|s\rangle\langle s| - I = H^{\otimes n}(2|0\rangle\langle 0| - I)H^{\otimes n} \quad (9)$$

The oracle and diffusion search operators repeats for $\pi/4\sqrt{N}$ times to get optimal search time and probability.

5.3. Quantum Fourier Transform

The Quantum Fourier Transform is a quantum analogue of classical Discrete Fourier Transform/Fast Fourier Transform. In this writing, the details of the QFT will not be elaborated.

Definition of QFT ($N = 2^n$) in terms of state vectors and state construction:

$$\begin{aligned} \text{QFT } |x\rangle &\mapsto \frac{1}{\sqrt{N}} \sum_{k=0}^{N-1} e^{2\pi i x k / N} |k\rangle \\ &\leftrightarrow \frac{1}{\sqrt{N}} \bigotimes_{k=1}^n \left(|0\rangle + e^{2\pi i x / 2^k} |1\rangle \right) \end{aligned} \quad (10)$$

6. Proposed Methods

Since the Quantum RAM (qRAM) and quantum annealing hardware testing are either doesn't exist in real world yet or cannot be accessed, the algorithm implemented will be theoretical. For practical experiment. Due to the limitations, the input and verification function will be controlled (not generalized).

The generalized version of the sequence alignment problems are string matching problems. There are numerous classical algorithms for those problems. Some algorithms can exactly solve the problem while some can approximate the solution or only tell the feasibility. These are some classical algorithms involving string matching problems and possibility of implementation in a quantum computer.

1. ***Hamming Edit Distance (Possible, Implementable)***
2. Levenshtein Edit Distance (Theoretical with qRAM)
3. Needleman-Wunsch (Theoretical with qRAM)
4. Smith-Waterman (Theoretical with qRAM)
5. Cosine Similarity (Possible)
6. Graph Edit Distance (Theoretical)
7. ***Pattern Matching Approximation with QFT
(Quantum analogous of DFT/FFT)***
8. BLAST Database Search Matching (Never proven as of now)
9. Knuth-Morris-Pratt string search algorithm (Never proven as of now)

As the Hamming Edit Distance is possible and can be implemented for experiments as of now, the quantum analogous will be made. There are various primitive algorithms which can be applied to the design of the problem. The mathematical perspectives will not be written in this writings, only the circuitry aspects will be shown.

The concept is to convert the string into a time domain and a frequency domain. Two strings are compared in both domains. Comparing in time domain shows the pairwise difference between two strings. Meanwhile, comparing in the frequency domain shows the similarity in substring and pattern frequency of both strings. The latter can also show the phase difference which represents circular shifting within the strings, and can be potentially meaningful if there are practical way to measure the phase. The frequency counterpart will be discussed in the next section.

We are currently in “Noisy Intermediate-Scale Quantum Era” (NISQ). As the quantum computer has many limitations as of now, some constraints must me made so that the solution is possible. The string comparison is pairwise and constrained to equal length only.

6.1. Sequence Encoding

Before solving the problem with a quantum computer, the input has to be converted into a quantum input. In this sense, 3 to 4 quantum registers are prepared for 2 input sequences and 1-2 output results. We also need 1-2 classical registers for measurement of respective output quantum registers.

Since, in this case, a DNA sequence of length n is a string described by $\{A, T, C, G\}^n$. Therefore it can be encoded into a minimum length of 2 bits: $\{00, 01, 10, 11\}$ or a one-hot encoding of 4-bit length: $\{0001, 0010, 0100, 1000\}$.

Another possibility is reverse-readable normal and one-hot encoding which might be beneficial for future applications where reverse reading is a part of a problem: $\{0000, 0110, 1001, 1111\}$ and $\{00011000, 00100100, 01000010, 10000001\}$. It would also help the accuracy of pattern matching.

Assuming 2 input quantum registers, 1 output quantum register and 1 output classical register, we would need $3kn$ qubits, where k is encoding length per character. For example, a 63-qubit quantum computer can only support maximum string length of 10 characters each ($3 \times 2 \times 10$).

Now the problem is encoded into an initial quantum state, the quantum algorithms can be applied.

6.2. Quantum Analogous

In classical version, hamming distance calculation utilizes “Difference between character” in general, which the specialized version of binary uses XOR operation to determine if each character, in pairwise, is different. The 1’s are then counted to determine the hamming distance.

6.2.1. Time-domain comparison with XOR

To implement a quantum XOR operation building block on 2 qubits. Normally we have 2 input qubits $|a\rangle$ and $|b\rangle$, the Controlled-NOT (CNOT) operation is utilized here. After the CNOT is applied, the state $|a\rangle$ remains the same, while $|b\rangle$ evolves into $|a \oplus b\rangle$.

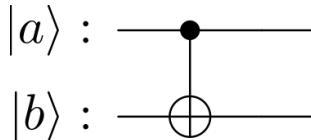


Figure 12: Typical XOR Operation

This design has some drawbacks as if we want to use $|b\rangle$ later in the circuit, it would not be possible. To alleviate this issue, another set of qubit is introduced to store the

output result. By applying $\text{CNOT}(|a\rangle, |b\rangle) \otimes |0\rangle$ yielding state $|a\rangle \otimes |a \oplus b\rangle \otimes |0\rangle$. We can apply another CNOT from $|a \oplus b\rangle$ to the output qubit. After doing so, as CNOT is its own inverse. We can apply the last CNOT on the original qubit pair. This way, the $|a\rangle$ and $|b\rangle$ conserves.² The circuit for XOR with output qubit conserving input is shown as follows.

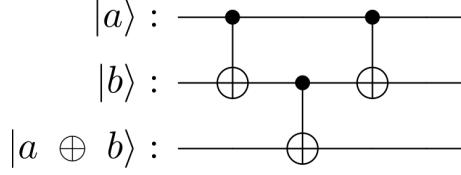


Figure 13: XOR with output qubit

As a result, basic quantum hamming distance calculation can be achieved. The general circuit diagrams (block and 1-bit example decomposed circuits) are illustrated as follows.

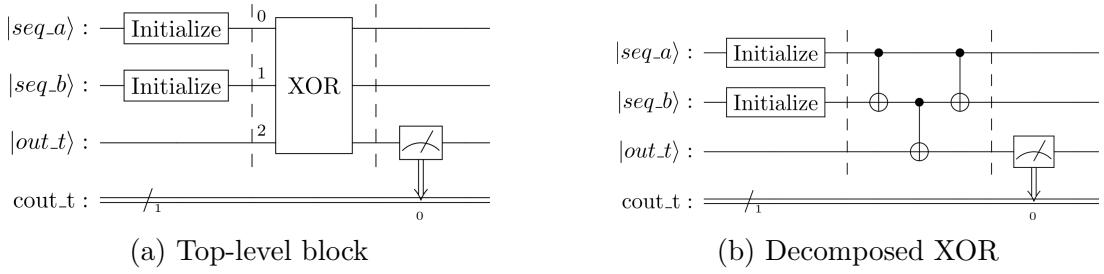


Figure 14: Time-domain comparison with XOR

For more intuitive examples, a 2-character (4-bit) hamming distance is shown as follows.

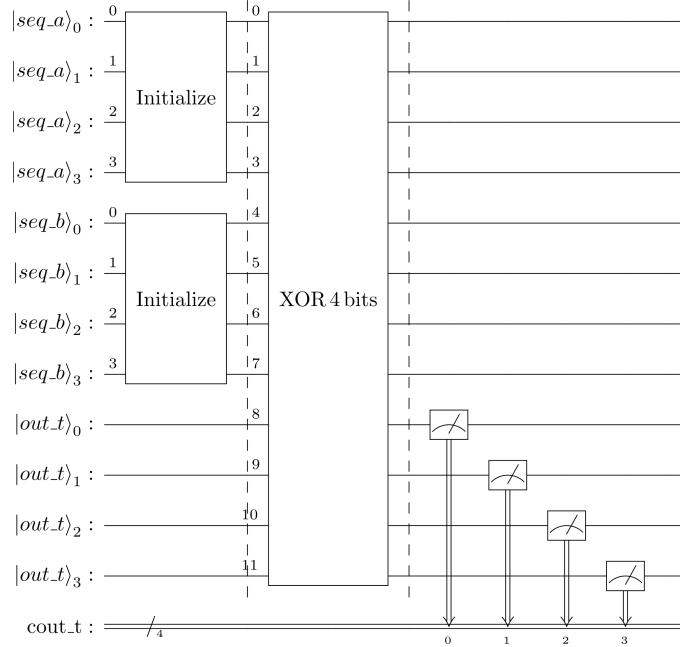


Figure 15: XOR with output qubit

²This utilizes the fact that unitary operators make quantum computing reversible.

The results in the output quantum register are then measured into the classical register. The classical bit position with 1 means that the string pair are different in that position. Otherwise, the bit is 0, it is the same. Counting the occurrence of 1's can determine the hamming distance.

Moreover, using kn -bit output which is 1:1 to the input string length, the exact position of mismatch can be determined. This standard XOR hamming distance can potentially be extended to quadratically compare $m \times n$ each character, and determine the indels' number and position collectively.

6.2.2. Utilizing quantum adder for XOR comparison

In the previous implementation, the calculation uses kn output qubits to determine both number and position. If the position does not matter and output requires only a distance, a quantum accumulator can be implemented instead of 1:1 positional mapping.

To implement a quantum accumulator to the algorithm, the accumulator can be attached to the XOR computation phase. The resulting number of qubits can be reduced to $\log_2(kn)$. Quantum adder can be used in the accumulator: either *ripple carry adder* implementation or *QFT adder* implementation.

Adding numbers using phase rotation and approximating the resulting phase using Quantum Phase Estimation (QPE) algorithm is also a possibility.

6.2.3. Time-domain (computational basis) comparison with Swap Test

Using traditional XOR does not use any of the quantum properties to determine the similarity of two strings. Instead of using XOR for bit comparisons, the **Swap Test** utilizing superposition is used in place of XOR part.

The Swap Test utilizes Controlled-SWAP (CSWAP or Fredkin) gate with the control qubit in superposition (using Hadamard gate) as shown in Fig. 16.

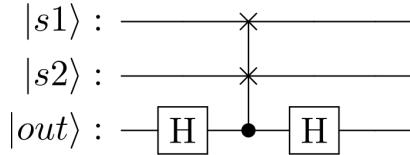


Figure 16: 1-bit Test Swap Quantum Circuit

To read the output, it is not straightforward 1 or 0 like standard XOR. The output can be determined by running the circuit many times, e.g., 1024 shots and reading the quasiprobability of the final measurement of $|out\rangle$. If $|s1\rangle$ and $|s2\rangle$ are the same, i.e., $|00\rangle$ and $|11\rangle$, then $\text{Pr}[out = 0] = 1$ and $\text{Pr}[out = 1] = 0$. Otherwise, $\text{Pr}[out = 0] = 0.5$ and $\text{Pr}[out = 1] = 0.5$, meaning for 1024 shots, theoretically, 0 is shown 512 times and 1 is shown 512 times.

Swap Test operation can be described as follows.

$$\begin{aligned}
& |0\rangle \otimes |a\rangle \otimes |b\rangle \\
& \rightarrow \frac{1}{\sqrt{2}} (|0\rangle + |1\rangle) |a\rangle |b\rangle \\
& = \frac{1}{\sqrt{2}} (|0\rangle |a\rangle |b\rangle + |1\rangle |a\rangle |b\rangle) \\
& \rightarrow \frac{1}{\sqrt{2}} (|0\rangle |a\rangle |b\rangle + |1\rangle |b\rangle |a\rangle) \\
& = \frac{1}{\sqrt{2}} \left(\frac{|0\rangle + |1\rangle}{\sqrt{2}} |a\rangle |b\rangle + \frac{|0\rangle - |1\rangle}{\sqrt{2}} |b\rangle |a\rangle \right) \\
& = \frac{1}{2} \left(|0\rangle \otimes (|a\rangle |b\rangle + |b\rangle |a\rangle) + |1\rangle \otimes (|a\rangle |b\rangle - |b\rangle |a\rangle) \right)
\end{aligned} \tag{11}$$

For more qubits, we can apply $H^{\otimes kn}$ to the $|out\rangle$ which is initialized as default ($|0\rangle^{\otimes kn}$) at the beginning. Therefore, the improved version of the circuit diagrams are shown as follows (shown as 2-bit version).

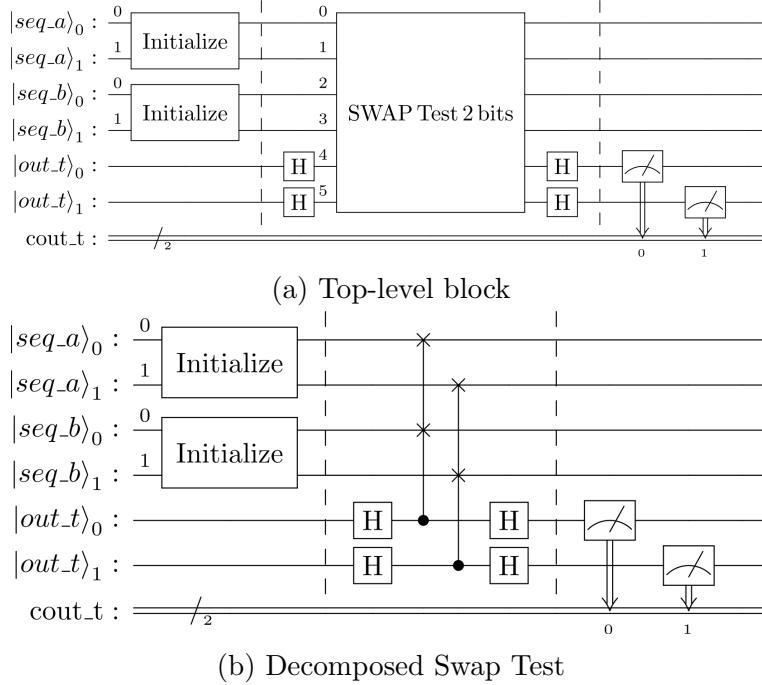


Figure 17: Time-domain comparison with Swap Test

For more bits, the analogy is the same. To determine the position of the mismatch, after running the circuits many times, the probability of each position with mismatch is theoretically $\frac{1}{2^m}$ where m is number of 1's occurrences. Any with mismatch will show on the quasiprobability graph with $|...0...\rangle$ and $|...1...\rangle$ at the same bit position with roughly equal probability from the quantum computer or a noisy simulator.

6.2.4. Frequency-domain comparison with Swap Test

Measuring difference in the frequency domain can yield more information about similarity of two strings. It can show similarity in pattern, shifting and repetition, which in computational basis cannot show.

To perform measurement in frequency domain, we apply a QFT operation in each string (state) separately. Then, a swap test is applied. After the computation, Inverse QFT (QFT^\dagger) operation is lastly applied to revert the states back to their original states, so we can use the states for future computations in the circuit.

In the time-domain swap test, each bit from each sequence is compared using H-CSWAP one by one, but in frequency domain, the frequency (signature) difference is compared after QFT operation.

The post-initial state with $H^{\otimes n}$ and QFT $|s_m\rangle$ is as follows.

$$(\text{H}|0\rangle)^{\otimes n} \otimes \text{QFT}|s_1\rangle \otimes \text{QFT}|s_2\rangle \quad (12)$$

Recalling from the time-domain swap test with addition of QFT in the circuit, the behavior is as follows.

$$\frac{1}{2} \left(|0\rangle \otimes (|a\rangle|b\rangle + |b\rangle|a\rangle) + |1\rangle \otimes (|a\rangle|b\rangle - |b\rangle|a\rangle) \right) \quad (13)$$

with definition of QFT ($N = 2^n$):

$$\begin{aligned} \text{QFT}|x\rangle &\mapsto \frac{1}{\sqrt{N}} \sum_{k=0}^{N-1} e^{2\pi i x k / N} |k\rangle \\ &\leftrightarrow \frac{1}{\sqrt{N}} \bigotimes_{k=1}^n \left(|0\rangle + e^{2\pi i x / 2^k} |1\rangle \right) \end{aligned} \quad (14)$$

while in each $|a\rangle$ and $|b\rangle$ terms,

$$\begin{aligned} |a\rangle &= \text{QFT}|s_1\rangle \\ &= |0\rangle + e^{2\pi i s_1 / 2^k} |1\rangle \\ &\text{and} \\ |b\rangle &= \text{QFT}|s_2\rangle \\ &= |0\rangle + e^{2\pi i s_2 / 2^k} |1\rangle \\ &\text{for } k = 1, 2, \dots, n. \end{aligned} \quad (15)$$

As a result, $|a\rangle|b\rangle$ and $|b\rangle|a\rangle$ are as the following terms.

$$\begin{aligned} |a\rangle|b\rangle &= |0\rangle|0\rangle + e^{2\pi i s_2 / 2^k} |0\rangle|1\rangle + e^{2\pi i s_1 / 2^k} |1\rangle|0\rangle + e^{2\pi i (s_1+s_2) / 2^k} |1\rangle|1\rangle \\ |b\rangle|a\rangle &= |0\rangle|0\rangle + e^{2\pi i s_1 / 2^k} |0\rangle|1\rangle + e^{2\pi i s_2 / 2^k} |1\rangle|0\rangle + e^{2\pi i (s_1+s_2) / 2^k} |1\rangle|1\rangle \end{aligned} \quad (16)$$

Therefore,

$$\begin{aligned}
|a\rangle|b\rangle + |b\rangle|a\rangle &= 2|0\rangle|0\rangle + \left(e^{2\pi i s_2/2^k} + e^{2\pi i s_1/2^k}\right)|0\rangle|1\rangle \\
&\quad + \left(e^{2\pi i s_1/2^k} + e^{2\pi i s_2/2^k}\right)|1\rangle|0\rangle + 2e^{2\pi i(s_1+s_2)/2^k}|1\rangle|1\rangle \\
&= 2|0\rangle|0\rangle + \left(e^{2\pi i s_2/2^k} + e^{2\pi i s_1/2^k}\right)(|0\rangle|1\rangle + |1\rangle|0\rangle) \\
&\quad + 2e^{2\pi i(s_1+s_2)/2^k}|1\rangle|1\rangle
\end{aligned} \tag{17}$$

and

$$\begin{aligned}
|a\rangle|b\rangle - |b\rangle|a\rangle &= \left(e^{2\pi i s_2/2^k} - e^{2\pi i s_1/2^k}\right)|0\rangle|1\rangle \\
&\quad + \left(e^{2\pi i s_1/2^k} - e^{2\pi i s_2/2^k}\right)|1\rangle|0\rangle \\
&= \left(e^{2\pi i s_2/2^k} - e^{2\pi i s_1/2^k}\right)(|0\rangle|1\rangle - |1\rangle|0\rangle)
\end{aligned} \tag{18}$$

Finally, the final measured state of the frequency-domain swap test can be described by the following state expression.

$$\frac{1}{\sqrt{N}} \bigotimes_{k=1}^n \left(|0\rangle|\psi_1\rangle + |1\rangle|\psi_2\rangle \right) \tag{19}$$

where

$$|\psi_1\rangle = |0\rangle|0\rangle + \frac{e^{2\pi i s_2/2^k} + e^{2\pi i s_1/2^k}}{2}(|0\rangle|1\rangle + |1\rangle|0\rangle) + e^{2\pi i(s_1+s_2)/2^k}|1\rangle|1\rangle \tag{20}$$

and

$$|\psi_2\rangle = \frac{e^{2\pi i s_2/2^k} - e^{2\pi i s_1/2^k}}{2}(|0\rangle|1\rangle - |1\rangle|0\rangle) \tag{21}$$

In conclusion, measuring in frequency domain with swap test is similar to measuring in computational basis, but with QFT and QFT^\dagger .

The circuit corresponding the QFT-swap-test (2-bit example) can be constructed as illustrated in Fig. 18.

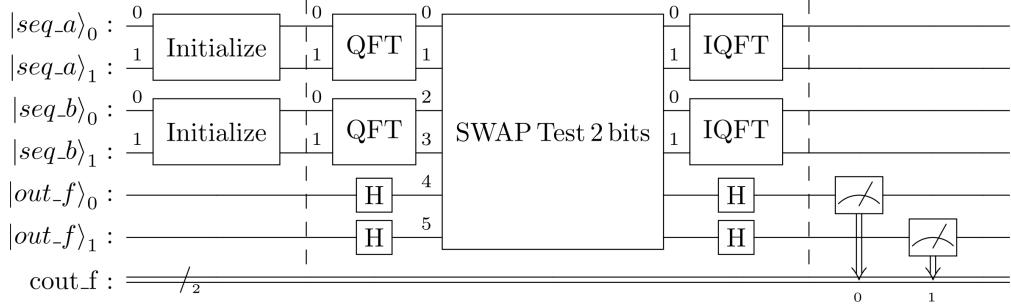


Figure 18: Frequency-domain comparison with Swap Test

Likewise, the analogy of extending the circuit with string length of other than 2 qubits is the same by scaling the computation blocks to corresponding data qubit width.

6.2.5. Swap Test comparison with dual domain (Combination)

Measuring similarity/difference in both time domain and frequency domain is possible. You can either measure in separate circuit, or measure both in the same circuit sequentially, but this implementation requires resetting and reinitializing the input states.

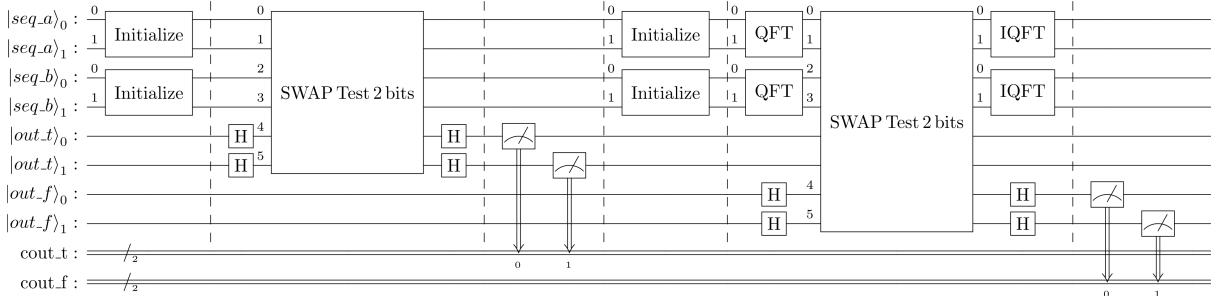


Figure 19: Time-and-Frequency-domain comparison with Swap Test

6.2.6. Time-domain Permutation Swap Test

To determine all differences between two strings, a permutation of gates in $s_1 \{0, 1, \dots, n - 1\} \times s_2 \{0, 1, \dots, n - 1\}$ can also be achieved; although, this method increases the quantum gate numbers from linear (n) to quadratic (n^2).

6.2.7. Quantum Phase Estimation for frequency domain

Another possible implementation if we want to measure phase difference between two strings as we treated them as signals would be using the QPE algorithm to determine phase. Creating a specialized version of U quantum realization of a classical comparison function is non-trivial. Instead, Fig. 20 shows a generalized version of QPE with base-2 fractional sum and QFT^\dagger to complete its reverse.

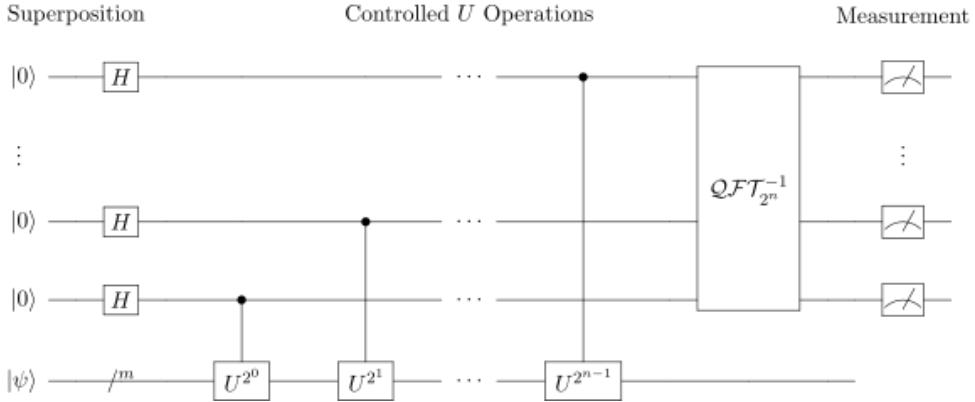


Figure 20: Generalized Quantum Phase Estimation Circuit

The details of QPE algorithm will not be covered in this writings. The QPE is used in Shor's algorithm to determine periodicity which is one of the most important a algorithm in quantum computation.

7. Comparison with Related Researches

1. The proposed method is similar to Research 1 as it utilizes QFT for pattern recognition, but the proposed method has simpler implementation without quadratic comparison like Research 1 has proposed.
2. Graph Algorithm proposed by Research 2 was not implemented due to technical difficulties.
3. Research 3 proposed Annealing for optimization which is very different from the proposed method of simplifying algorithms, using hamming distance and QFT.

8. Alternative Approaches

8.1. Qudit Encoding

A qudit is generalized d -level quantum system. In particular, a qubit is 2-level quantum system, a qutrit is 3 level, and a ququart is 4 level. The state of a qudit is similarly described by a vector in a d -dimensional complex Hilbert space \mathcal{H}_d and can be written as:

$$|\psi\rangle = \sum_{i=0}^{d-1} c_i |i\rangle \quad ; \quad \sum_{i=0}^{d-1} \|c_i\|^2 = 1 \quad (22)$$

Operations on a single qudit can be done by unitary operators $U_{d \times d}$ where $U^\dagger U = I$. A quantum system of qudits can be described as follows.

$$|q\rangle = |q_1, \dots, q_n\rangle = \bigotimes_{i=1}^n \sum_{i=0}^{d-1} c_i |i\rangle \in \mathbb{C}^{d^n} \quad (23)$$

Qudit encoding can help reducing size of encoded string significantly. In DNA sequence case, normally, 4 bases can be minimally encoded using 2 qubits, but can be fully represented using only 1 qudit.

For example, in a 2-qubit system, $\{\text{A, T, C, G}\}$ maps minimally to $\{|00\rangle, |01\rangle, |10\rangle, |11\rangle\}$. In a 1-qudit system, the bases can map minimally to $\{|0\rangle, |1\rangle, |2\rangle, |3\rangle\}$.

Current quantum technology implements superconducting non-linear harmonic oscillator for a qubit, which the wavefunction in each energy level represent each state. A superposition of a state can be directly mapped to superposition of wavefunction in a harmonic oscillator.

However, more possibilities in each qudit mean more noise that can occur in a system which makes the computing less robust as the current technology is advancing.

9. Quantum Circuit Generation with IBM Qiskit

There are many Quantum Computing framework to use, including Cirq, Microsoft Q#, Tensorflow Quantum (for QCNN ML) and IBM Qiskit. IBM Qiskit provides free monthly credit of 10-minutes worth of quantum computer runtime with additional starter's credit from IBM Cloud worth of 5 minutes computation time. IBM Quantum Computers are also more accessible than other providers. With numerous documentations and learning resources, IBM Qiskit and IBM Cloud fit the needs.

IBM Qiskit framework can be run on Python, i.e., Jupyter Notebook. As of now, IBM's quantum computer's processor with most qubits (that I can access) is IBM Eagle (IBM Osprey has 433 qubits). It comes with staggering 127 qubits. Within 2023, IBM is planned to release a 1000-qubit processor.

Once algorithm design phase and circuit design phase are completed, you can build a circuit from your computational model using Qiskit in Python.

You can also decompose, transpile (translate + compile) and send the circuit to the IBM's runtime provider to run on either simulators or real quantum computers directly via Qiskit API. After running, the result of simulation or computation can be acquired from the finished job, which you can run data post-processing on your classical machine.

To convert an classical input bits, the input bits are prepared in a State Preparation phase. By default, a qubit is always initialized to 0. To make it 1, you can prepare it by appending a NOT to it. In Qiskit framework, you can also prepare the state using a statevector (a vector of quasiprobability of each state) as well.

9.1. About `ibm_brisbane`

The `ibm_brisbane` quantum computer has a total of 127 qubits in the system utilizing IBM's Eagle r3 processor. Superconducting qubits, operating using Josephson junction at approximately 25 mK, technology is used in this processor. It has Heavy-Hex lattice qubits structure which mitigates errors by limiting connectivity to nearest neighbors and reducing cross-talk between qubits. Fig. 21 shows the lattice structure and readout assignment error scales of `ibm_brisbane` at the time of testing stated circuits (3 Nov 2023 at 17:33:27 UTC).

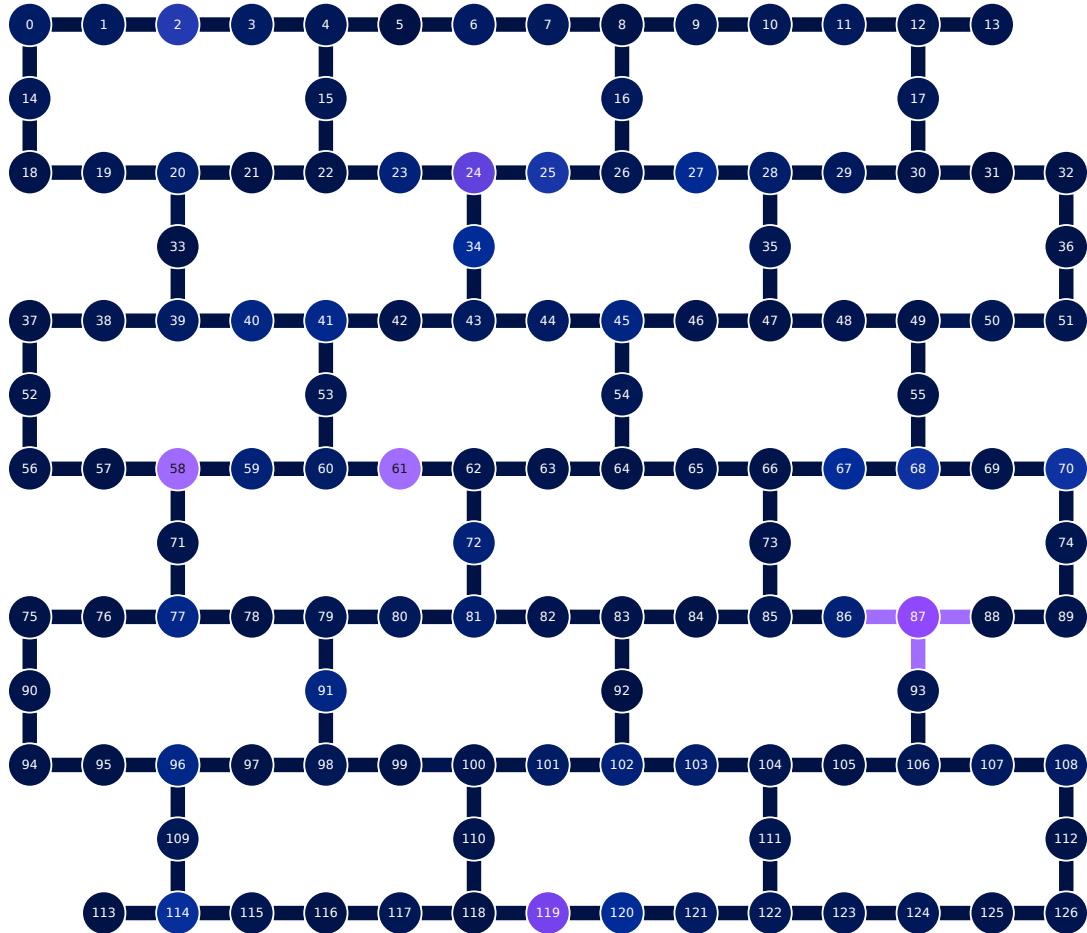


Figure 21: `ibm_brisbane`'s lattice structure (Courtesy: *IBM Quantum Platform*)

9.2. Preparing Basic Operators

In the designed algorithms, all computation units can be constructed from primitive gates. To support dynamic scaling, the blocks are parametric and can be scaled by bits. These are operators needed to be prepared.

9.2.1. String Encoding

Map each character of a sequence to binary version by encoding using either 2-bit normal or 4-bit one hot encoding.

```
def encode_dna(dna_seq: str):
    # DNA_MAP = {
    #     'A': '0001',
    #     'T': '0010',
    #     'C': '0100',
    #     'G': '1000'
    # }
    DNA_MAP = {
        'A': '00',
        'T': '01',
        'C': '10',
        'G': '11'
    }
    s = ''
    for c in dna_seq:
        s += DNA_MAP[c]
    return s
```

9.2.2. State Initialization

Generate a quantum subcircuit for initializing a state using most-significant bit (MSB) representation.

```
def make_init(bits: int, label: str = None):
    if label is None:
        _circ = QuantumCircuit(bits, name=f'Initialize')
        _circ.reset(range(0, bits))
        return _circ

    if len(label) != bits:
        raise Exception('Number of bits must equal label length')
    _circ = QuantumCircuit(bits, name=f'Initialize')
    _circ.reset(range(0, bits))
    for i, c in enumerate(reversed(label)):
        if c == '1':
            _circ.x(i)
    return _circ
```

9.2.3. *n*-bit Quantum XOR

Generate a template for *n*-bit Quantum XOR operation with $|s1\rangle$, $|s2\rangle$, $|out\rangle$.

```
def make_xor(bits: int):
    _qreg_s1 = QuantumRegister(bits, '\\ket{s1}')
    _qreg_s2 = QuantumRegister(bits, '\\ket{s2}')
    _qreg_out = QuantumRegister(bits, '\\ket{out}')
    _circ = QuantumCircuit(_qreg_s1, _qreg_s2, _qreg_out, name=f'XOR {bits} bits')
    for i in range(bits):
        _circ.cx(_qreg_s1[i], _qreg_s2[i])
        _circ.cx(_qreg_s2[i], _qreg_out[i])
        _circ.cx(_qreg_s1[i], _qreg_s2[i])
    return _circ
```

9.2.4. *n*-bit Swap Test

Generate a template for *n*-bit Swap Test operation with $|s1\rangle$, $|s2\rangle$, $|out\rangle$.

```
def make_swap_test(bits: int):
    _qreg_s1 = QuantumRegister(bits, '\\ket{s1}')
    _qreg_s2 = QuantumRegister(bits, '\\ket{s2}')
    _qreg_out = QuantumRegister(bits, '\\ket{out}')
    _circ = QuantumCircuit(_qreg_s1, _qreg_s2, _qreg_out, name=f'SWAP Test {bits} bits')
    _circ.cswap(_qreg_out, _qreg_s1, _qreg_s2)

    return _circ
```

9.2.5. Initialize Basic Operators

```
SEQ1 = 'AAAAAAA'
SEQ2 = 'GGGGGGG'

if len(SEQ1) != len(SEQ2):
    raise Exception('Sequence does not have equal length!')

STRLEN = len(SEQ1)
ENCODED_LENGTH = 2
BITS = STRLEN * ENCODED_LENGTH

enc1 = encode_dna(SEQ1)
enc2 = encode_dna(SEQ2)

circ_init_seq1 = make_init(BITS, enc1)
circ_init_seq2 = make_init(BITS, enc2)
circ_cmp_xor = make_xor(BITS)
circ_cmp_st = make_swap_test(BITS)

circ_qft = QFT(BITS)
circ_iqft = circ_qft.inverse()
```

9.3. Time-domain comparison with XOR

```
qreg_seq1 = QuantumRegister(BITS, '\\ket{seq_a}')
qreg_seq2 = QuantumRegister(BITS, '\\ket{seq_b}')
qreg_out_t = QuantumRegister(BITS, '\\ket{out_t}')
creg_out_t = ClassicalRegister(BITS, 'cout_t')

circ1 = QuantumCircuit(qreg_seq1, qreg_seq2, qreg_out_t, creg_out_t)

circ1.append(circ_init_seq1, qreg_seq1)
circ1.append(circ_init_seq2, qreg_seq2)

circ1.barrier()
circ1.append(circ_cmp_xor, [*qreg_seq1, *qreg_seq2, *qreg_out_t])

circ1.barrier()
circ1.measure(qreg_out_t, creg_out_t)

circ1.draw(output='latex', scale=1.0)
```

9.4. Time-domain comparison with Swap Test

```
qreg_seq1 = QuantumRegister(BITS, '\\ket{seq_a}')
qreg_seq2 = QuantumRegister(BITS, '\\ket{seq_b}')
qreg_out_t = QuantumRegister(BITS, '\\ket{out_t}')
creg_out_t = ClassicalRegister(BITS, 'cout_t')

circ2 = QuantumCircuit(qreg_seq1, qreg_seq2, qreg_out_t, creg_out_t)

circ2.append(circ_init_seq1, qreg_seq1)
circ2.append(circ_init_seq2, qreg_seq2)

circ2.barrier()
circ2.h(qreg_out_t)
circ2.append(circ_cmp_st, [*qreg_seq1, *qreg_seq2, *qreg_out_t])
circ2.h(qreg_out_t)

circ2.barrier()
circ2.measure(qreg_out_t, creg_out_t)

circ2.draw(output='latex', scale=1.0)
```

9.5. Frequency-domain comparison with Swap Test

```
qreg_seq1 = QuantumRegister(BITS, '\\ket{seq_a}')
qreg_seq2 = QuantumRegister(BITS, '\\ket{seq_b}')
qreg_out_f = QuantumRegister(BITS, '\\ket{out_f}')
creg_out_f = ClassicalRegister(BITS, 'cout_f')

circ3 = QuantumCircuit(qreg_seq1, qreg_seq2, qreg_out_f, creg_out_f)

circ3.append(circ_init_seq1, qreg_seq1)
circ3.append(circ_init_seq2, qreg_seq2)

circ3.barrier()
circ3.append(circ_qft, qreg_seq1)
circ3.append(circ_qft, qreg_seq2)
circ3.h(qreg_out_f)
circ3.append(circ_cmp_st, [*qreg_seq1, *qreg_seq2, *qreg_out_f])
circ3.h(qreg_out_f)
circ3.append(circ_iqft, qreg_seq1)
circ3.append(circ_iqft, qreg_seq2)

circ3.barrier()
circ3.measure(qreg_out_f, creg_out_f)

circ3.draw(output='latex', scale=1.0)
```

9.6. Dual domain comparison with Swap Test

```
qreg_seq1 = QuantumRegister(BITS, '\\ket{seq_a}')
qreg_seq2 = QuantumRegister(BITS, '\\ket{seq_b}')
qreg_out_t = QuantumRegister(BITS, '\\ket{out_t}')
qreg_out_f = QuantumRegister(BITS, '\\ket{out_f}')
creg_out_t = ClassicalRegister(BITS, 'cout_t')
creg_out_f = ClassicalRegister(BITS, 'cout_f')

circ4 = QuantumCircuit(qreg_seq1, qreg_seq2, qreg_out_t, qreg_out_f,
                      creg_out_t, creg_out_f)

circ4.append(circ_init_seq1, qreg_seq1)
circ4.append(circ_init_seq2, qreg_seq2)

circ4.barrier()
circ4.h(qreg_out_t)
circ4.append(circ_cmp_st, [*qreg_seq1, *qreg_seq2, *qreg_out_t])
circ4.h(qreg_out_t)

circ4.barrier()
circ4.measure(qreg_out_t, creg_out_t)

circ4.barrier()
circ4.append(circ_init_seq1, qreg_seq1)
circ4.append(circ_init_seq2, qreg_seq2)

circ4.barrier()
circ4.append(circ_qft, qreg_seq1)
circ4.append(circ_qft, qreg_seq2)
circ4.h(qreg_out_f)
circ4.append(circ_cmp_st, [*qreg_seq1, *qreg_seq2, *qreg_out_f])
circ4.h(qreg_out_f)
circ4.append(circ_iqft, qreg_seq1)
circ4.append(circ_iqft, qreg_seq2)

circ4.barrier()
circ4.measure(qreg_out_f, creg_out_f)

circ4.draw(output='latex', scale=1.0)
```

10. Quantum Algorithm Results & Discussion

10.1. Quantum Algorithm Simulation

Due to limited credits and very long wait time to use the quantum computer, an official IBM's Matrix Product State (100 qubits) sampler with noise simulation is used for now. The sampler is set to 1024 shots.

IBM Cloud provides many simulator services based on requirements free of charge with zero wait time.

1. Statevector simulator (`simulator_statevector`): Supports 50 qubits using Schrodinger wavefunction calculation model. Supports most unitary gates.
2. Stabilizer simulator (`simulator_stabilizer`): Supports up to 5,000 qubits but usable with Clifford gates only.
3. Extended Stabilizer simulator (`simulator_extended_simulator`): Supports 63 qubits with more gates.
4. MPS simulator (`simulator_mps`): Supports 100 qubits using Matrix Product State model.
5. QASM simulator (`ibmq_qasm_simulator`): Supports 32 qubits using context-aware calculation.

The MPS simulator has the best compromise supporting every gates used in the algorithm and the 100-qubit supports make the maximum string length be up to 16 character (32 bit) each for single domain calculation. For dual domain computation, strings with maximum length of 12 character (24 bit) are supported.

To simplify computation with reasonable result readings, 8-character (16-bit) strings was used in the simulations and experiments. Also, 32-bit strings input was experimented to compare with 16-bit strings.

Note that the primitive used in this experiment was Sampler (generates quasiprobability distribution for the circuit). Another primitive (did not use) available would be Estimator which calculates the expectation values from the circuit.

10.1.1. 16-bit Time-domain comparison with XOR

Sequences ATGCTTGC and TGCCTGCA are the test sample.

```
SEQ1 0001111001011110
SEQ2 0111101001111000
OUT  0110010000100110
OUT  25638  (decimal)
```

As you can see, the bit pair XOR operation works flawlessly. The number of occurrences of 0's and 1's are 10 and 6 respectively. So, the hamming distance is 6. Here is also the corresponding quasiprobability graph of measurement (in decimal).

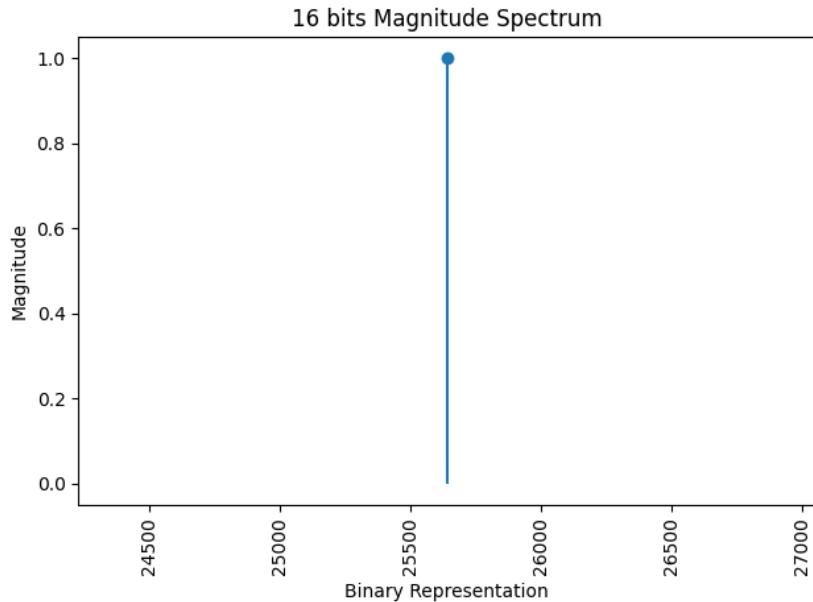


Figure 22: Quasiprobability Distribution

The graph shows that the measurement is exact ($\text{Pr}[\dots] = 1.0$).

10.1.2. 16-bit Time-domain comparison with Swap Test

Sequences ATGCTTGC and TGCCTGCA are the test sample.

```
SEQ1 0001111001011110  
SEQ2 0111101001111000
```

The result in a quasiprobability graph representation is not trivial, but the raw data can be processed. Taking all non-zero-magnitude points and do an `OR(array)` operation. The result is exactly the same as using XOR comparison, but the circuit is much smaller than the XOR counterpart.

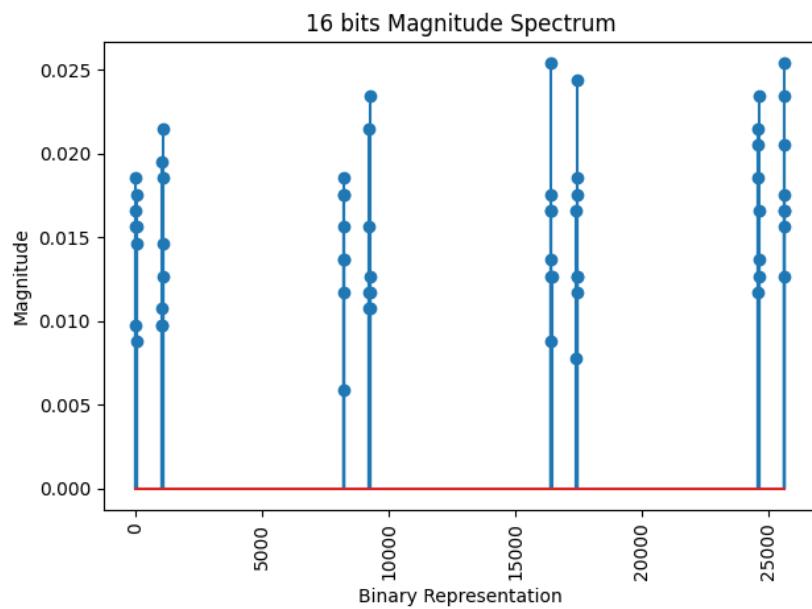


Figure 23: Quasiprobability Distribution

10.1.3. 16-bit Frequency-domain comparison with Swap Test

Sequences ATGCTTGC and TGCCTGCA are the test sample.

For frequency-domain comparison of strings with considerable length, a pattern on the frequency domain is shown clearly. Although deriving the meaning of those patterns are neither clear or trivial, the patterns on the quasiprobability graph of measurement looks very promising and should have some hidden meaning to it.

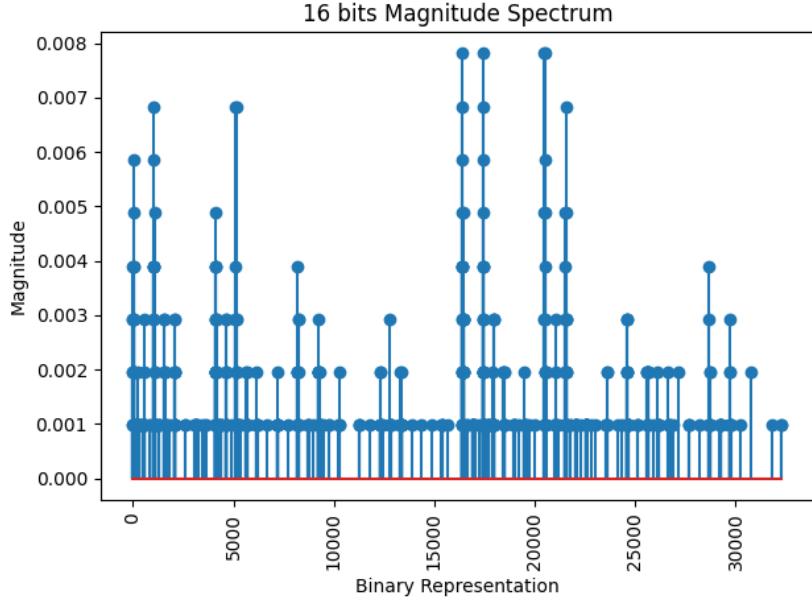


Figure 24: Quasiprobability Distribution

The data was observed to be transformed to a pattern matching problem. With more work, we should be able to derive the meaning from the chart.

Sequences AAAAGGGG and GGGGAAAA are the test sample.

As you can observe from the literal string, it has very low complexity. They are similar in low frequency and very different in high frequency. The pattern observation might show more feasibility to it. Note that A is encoded as 00 and G is encoded as 11. It is not aligned at all in the time domain, but you can see it is a circular shift of each other, so the frequency domain shows it.

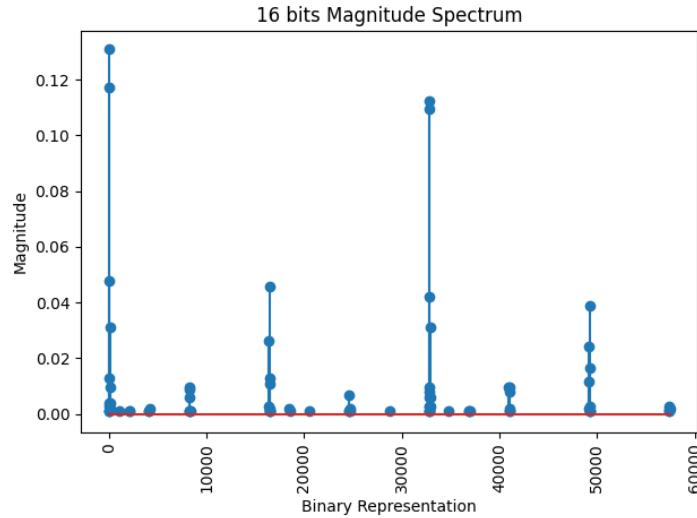


Figure 25: Quasiprobability Distribution

Sequences AAAAAAAA and GGGGGGGG are the test sample. The pattern is becoming more obvious.

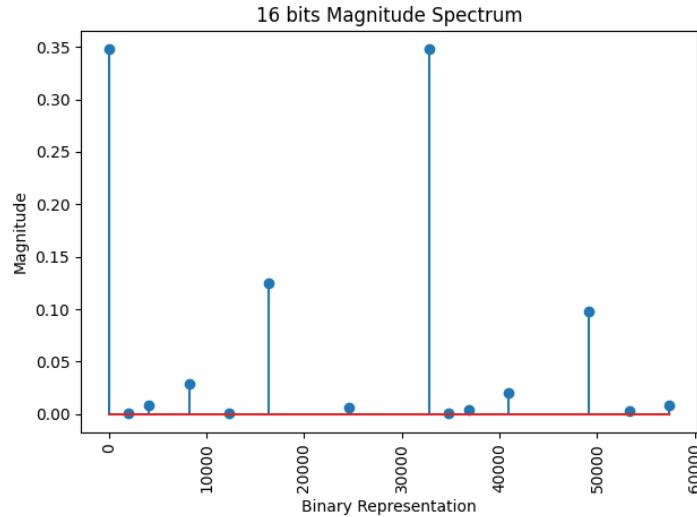


Figure 26: Quasiprobability Distribution

10.1.4. 32-bit Frequency-domain comparison with Swap Test

Sequences ATGCTTGCAGGGGGGGG and TGCCTGCAGGGGGGGG are the test sample.

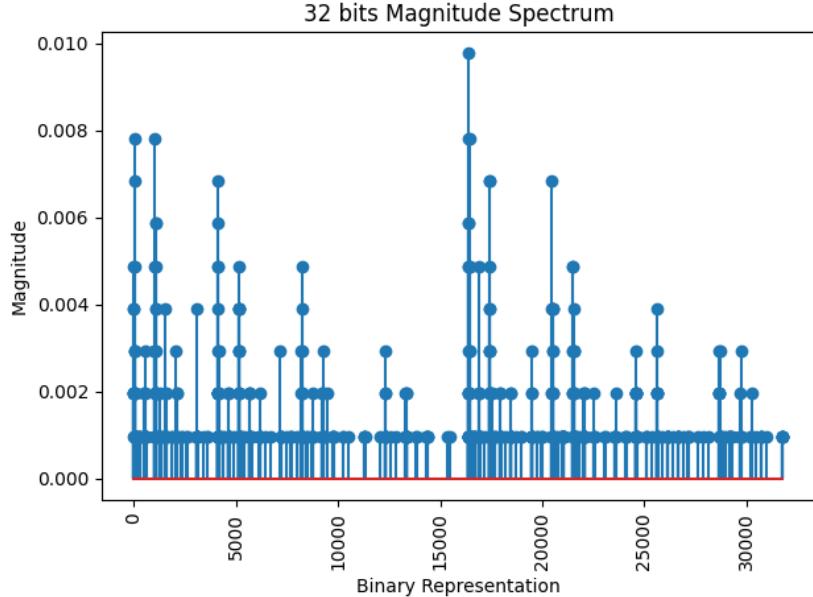


Figure 27: Quasiprobability Distribution

As seen in Fig. 27, the overall plot is similar to Fig. 24. Like padding in Classical Fourier Transform, the overall behaviour across the frequency domain is similar, but with higher resolution (by interpolation).

Sequences ATGCTTGCCTAAAGT and TGCCTGCACGCGCGCA are the test sample.

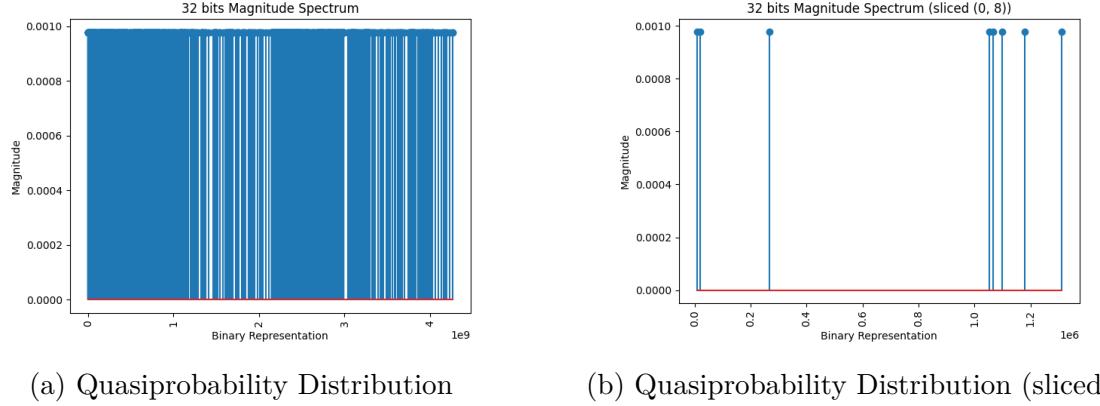


Figure 28: Quasiprobability Distribution (Indistinguishable)

As illustrated in Fig. 28, When the sequence is general and very dispersed from each other, probability of each outcome of the swap test of QFT in long bit length is close to each other and near zero. With the real quantum computer machine or a noise simulator, the readings will be inaccurate as the magnitude will be hidden within noises. For tested 32-bit string on the MPS simulator, the maximum magnitude read was approximately 0.001.

10.1.5. 8x8 16-bit Frequency-domain comparison with Swap Test

In this experiment, a pair of strings from a predetermined string pool are introduced to the initialization phase of each circuit. For example, a string pool \mathbb{P} having order $|\mathbb{P}|$ is predetermined. Every combination from $\mathbb{P} \times \mathbb{P}$ is tested in the circuit. In this case, \mathbb{P} is of order 8; therefore, there exists $\frac{8 \times 8}{2} = 32$ theoretical combinations from \mathbb{P} .

In this case,

$$\mathbb{P} = \{ \text{AAAAAAA}, \text{TTTTTTT}, \text{GGGGGGG}, \text{AGAGAGAG}, \\ \text{GAGAGAGA}, \text{AGTCGGCT}, \text{GCTACAGT}, \text{TGACATCG} \}$$

Fig. 29 illustrates results from running the circuit with Sampler on IBM's MPS simulator using string pairs from 8×8 Cartesian product $\mathbb{P} \times \mathbb{P}$.

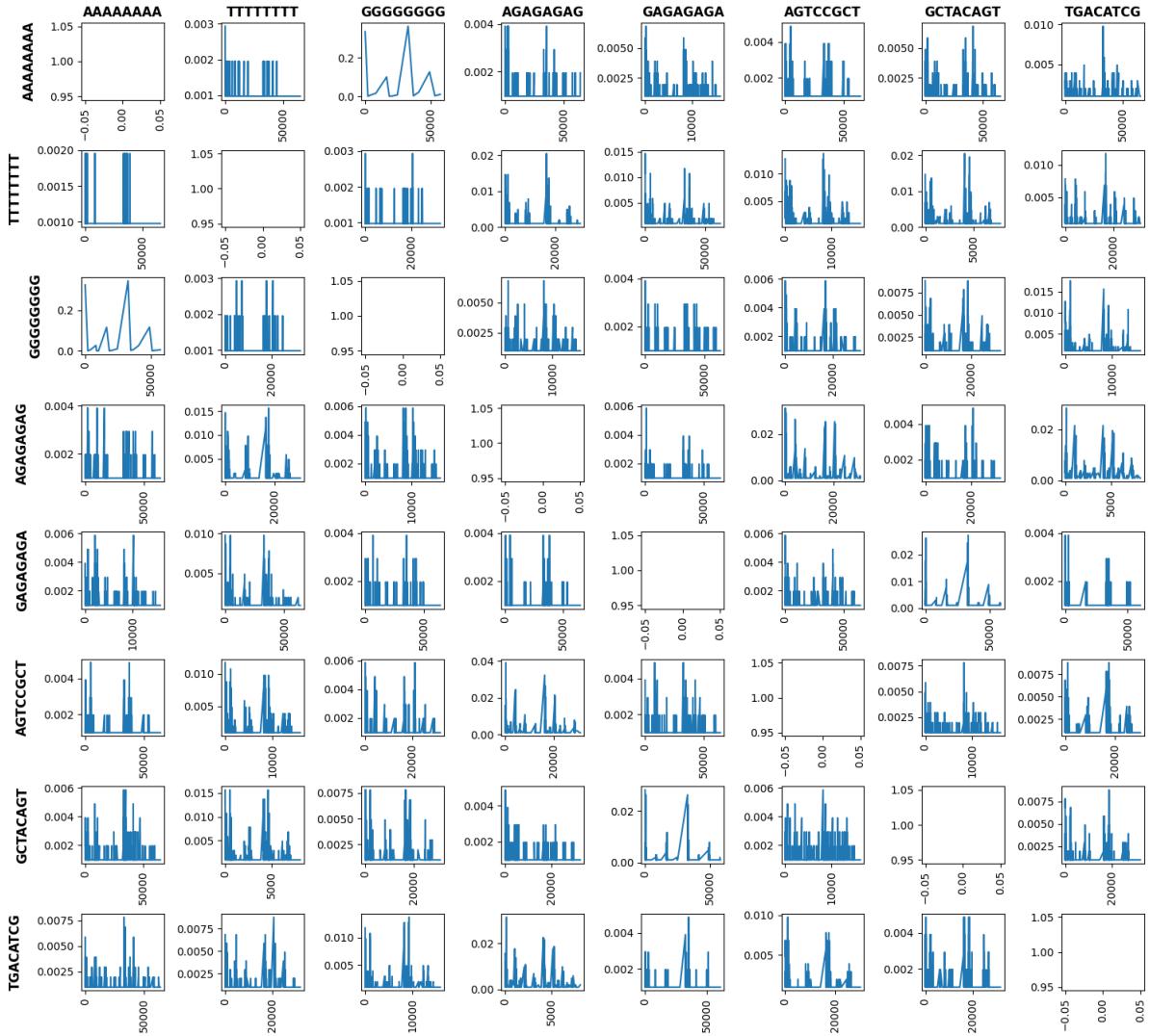


Figure 29: 8×8 Quasiprobability Distribution

10.1.6. 8-bit Dual domain comparison with Swap Test

Note that this only for experimentation and this method is not bit-efficient and decoding the result is complex. Sequences ATGC and TGCC are the test sample.

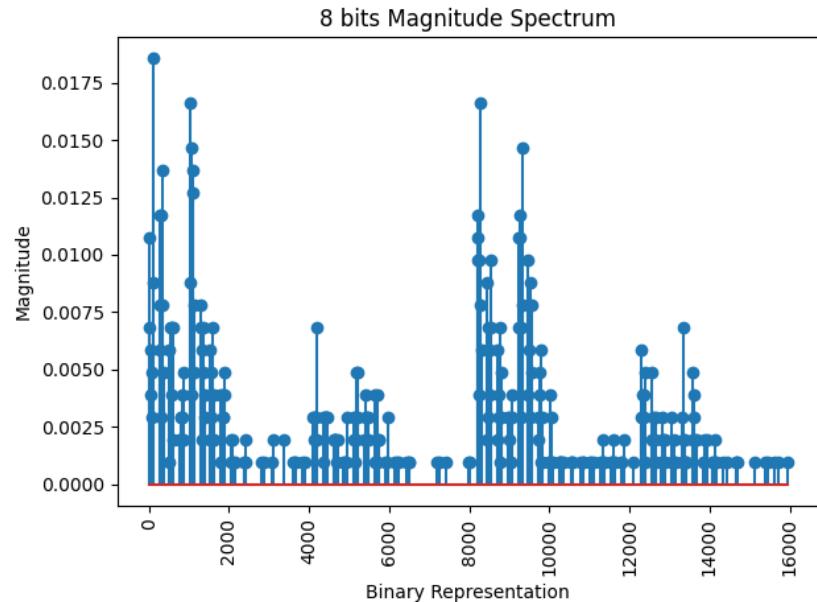


Figure 30: Quasiprobability Distribution

10.2. Results on Quantum Device

Time-domain XOR Comparison, Time-domain Swap Test and Frequency-domain Swap Test for 8-bit, 16-bit, 32-bit and 40-bit strings are executed on IBM’s MPS Simulator (codename `mps_simulator`) and IBM’s 127-qubit Eagle r3 processor quantum computer (codename `ibm_brisbane`). In this experiment, Sampler primitive with sample size of 1024 shots is used.

10.2.1. 8-bit String Comparison

Sequences TTGC and TGCT are the test sample.

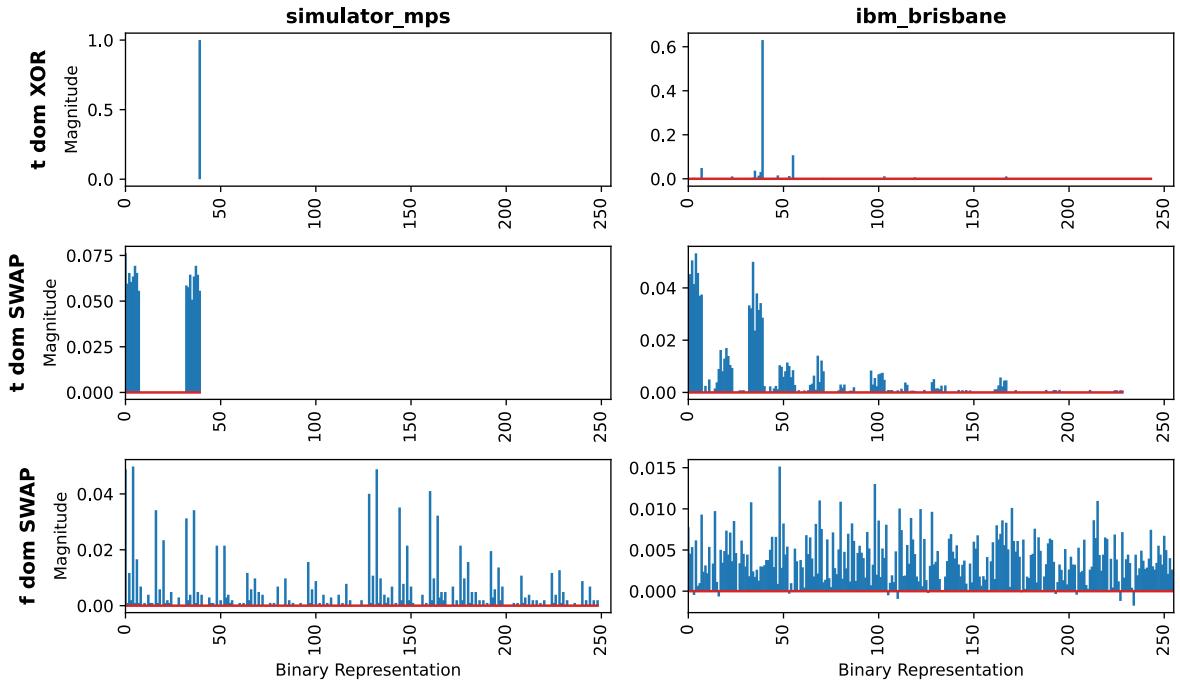


Figure 31: 8-bit String Running Results Comparison

As Fig. 31 has shown, on the quantum computer, there are noise and reading errors introduced to the circuit. Both time-domain comparison has similar high-probability regions. However, for frequency-domain comparison, the noise and errors have similar magnitude to the expected output quasiprobability distribution from the simulator.

10.2.2. 16-bit String Comparison

Sequences ATGCTTGC and TGCCTGCA are the test sample.

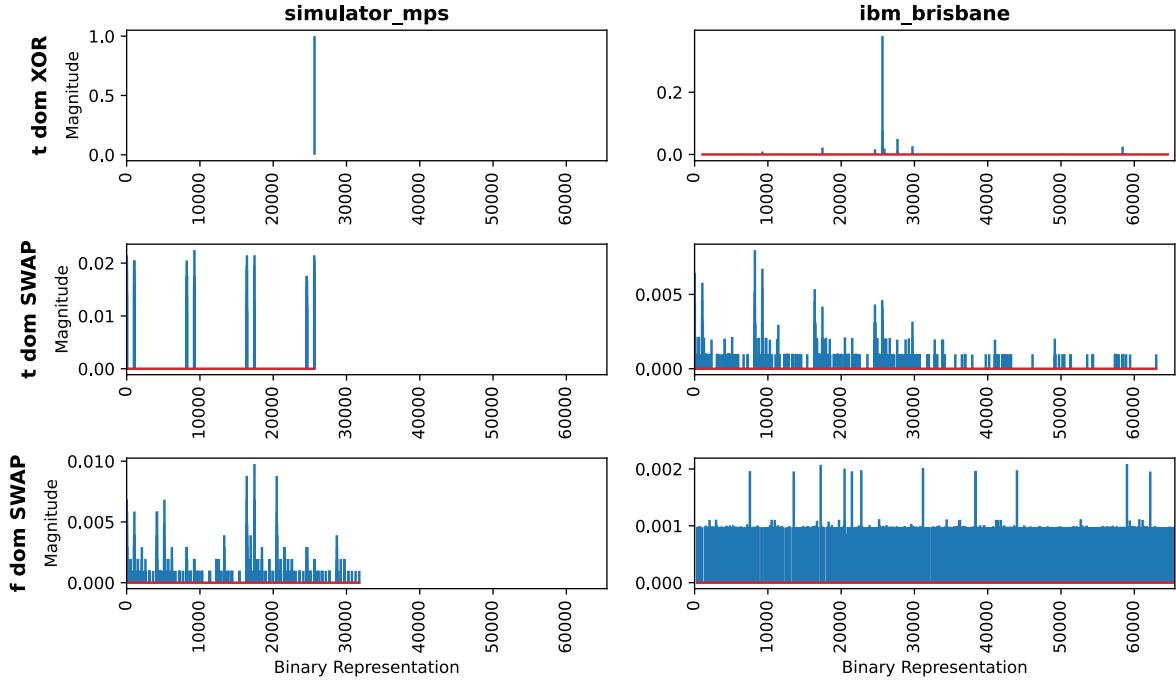


Figure 32: 16-bit String Running Results Comparison

Similar to 8-bit comparison, both time-domain comparison has similar high-probability regions, but frequency-domain comparison's data is likely uninterpretable.

10.2.3. 32-bit String Comparison

Sequences ATGCTTGCAGGGGGGG and TGCCTGCACGCGCGCA are the test sample.

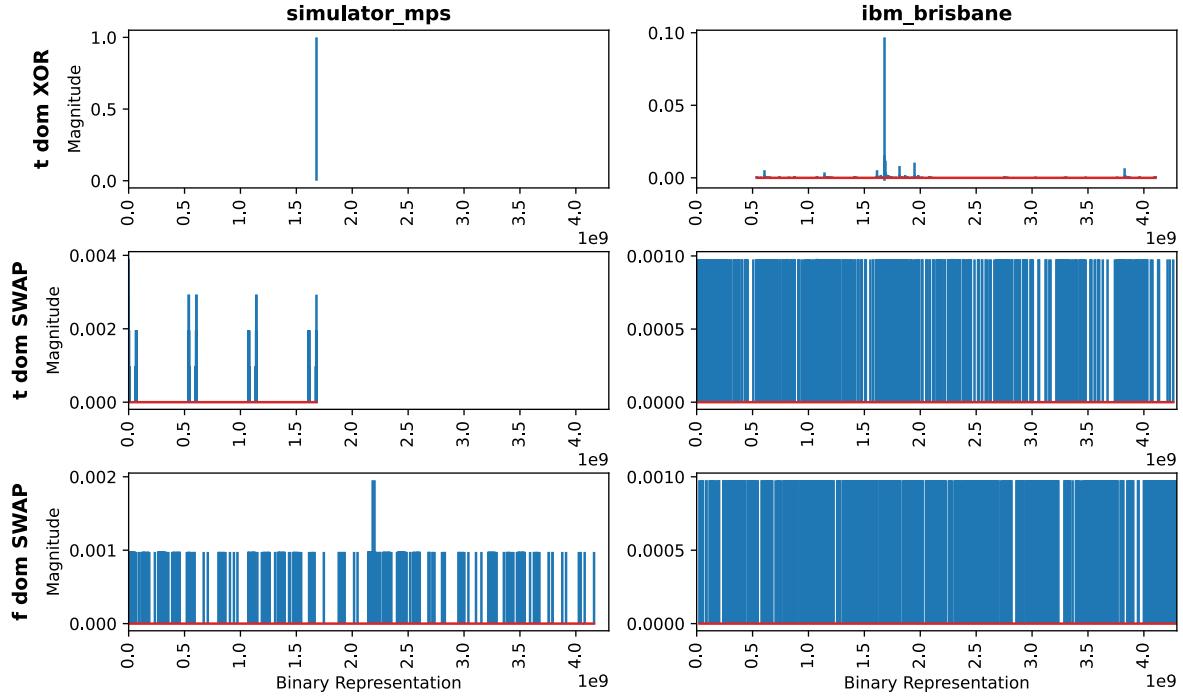


Figure 33: 32-bit String Running Results Comparison

As the number of bits increases, the noise acting on superposition is higher relative to measured probability. The only interpretable data with similar high-probability regions is time-domain comparison with XOR circuit. The results of Swap Tests are indistinguishable from noise.

10.2.4. 40-bit String Comparison

Sequences ATGCTTGCAGGGGGGGACAG and TGCCTGCACCGCGCATCAG are the test sample.

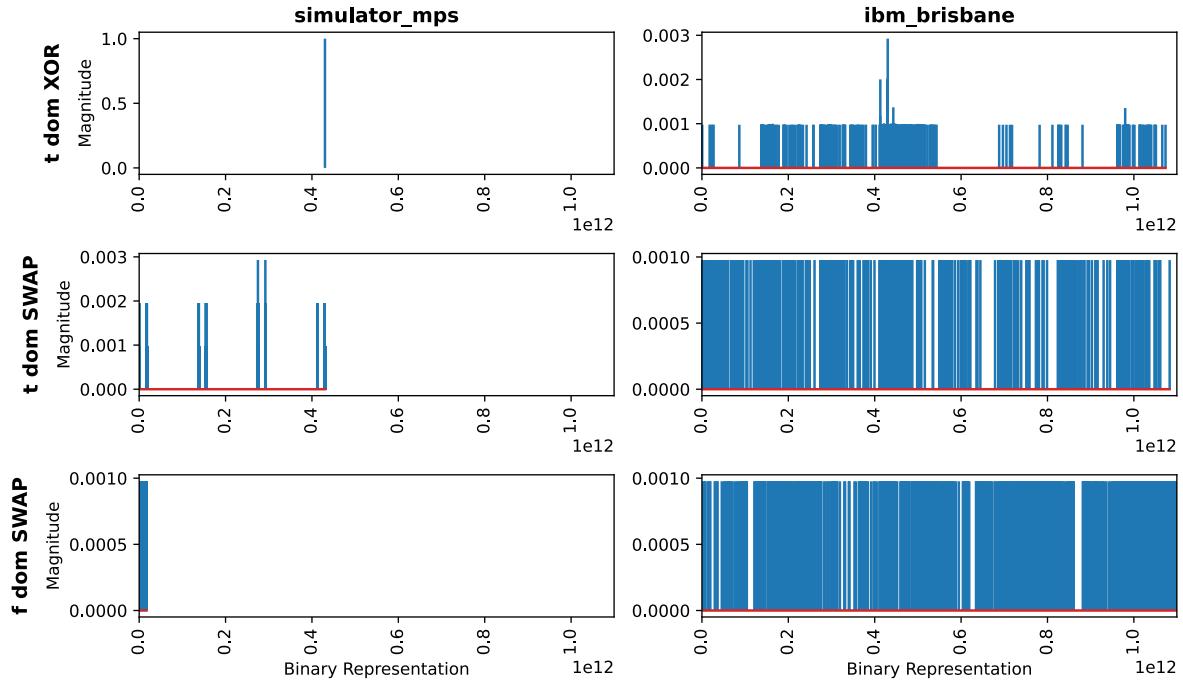


Figure 34: 40-bit String Running Results Comparison

For 40-bit string, time-domain comparison with XOR circuit starts to show higher magnitude of noise from other regions other than the expected output region. The results of Swap Tests are indistinguishable from noise.

10.3. Complexity Analysis

10.3.1. Initialization and Transformation

Qubits initialization, Hadamard transformation and measurements, in common, uses $O(n)$ gates and has $O(1)$ time complexity.

10.3.2. Comparison Operators

For XOR Comparator (Using 3 CNOT gates per qubit) and Swap Test Comparator (Using 2 H gates and 1 CSWAP gate per qubit), they are linear to number of input qubits. Hence, this part of the circuit also uses $O(n)$ gates with $O(n)$ time complexity as there are limitations to parallelize CSWAP gates and has to be executed sequentially.

10.3.3. Quantum Fourier Transform

The QFT algorithm used in the circuit requires only $O(n^2)$ gates and has time complexity of $O(\log^2 n)$

10.3.4. Overall Analysis

1. Time-domain comparison, the overall gate and time complexity is $O(n)$.
2. Frequency-domain comparison, the circuit requires $O(n^2)$ gates and has time complexity of $O(n)$.

Although complexity of the circuit can be determined, it cannot be directly compared to the known classical algorithms known as the statement to the problem is addressed differently and do not have exact result like in classical algorithms.

Due to the limitations of number of qubits on current quantum computers, the measurement of execution time of the circuit may not provide a reliable indication due to the limited number of qubits on a single system. Also, with noisy behavior of the current quantum computers, the result cannot be interpreted correctly.

10.4. Noise Analysis

In a quantum device, noise introduced to the system can be from a number of sources: decoherence, control errors, cross-talk, thermal noise, measurement errors, interference, loss of entanglement and many other sources from hardware.

To roughly approximate the amount of noise produced in the quantum device, in this experiment: `ibm_brisbane`, the Signal-to-Noise Ratio (SNR) in decibel (dB) was computed. SNR is computed by accumulating total differences between the source signal and the noisy signal as described in Eqn. 24.

$$\text{SNR (dB)} = 10 \log_{10} \frac{\sum A_{\text{signal}}^2}{\sum A_{\text{noise}}^2} \quad (24)$$

The simulated output was treated as source signal data and the output from quantum device was treated as noisy signal.

10.4.1. Qubit Width and SNR Comparison

	8 bits	16 bits	32 bits	40 bits
t dom XOR	8.125 dB	4.006 dB	0.873 dB	0.021 dB
t dom SWAP	6.912 dB	1.012 dB	-2.564 dB	-2.762 dB
f dom SWAP	0.749 dB	-1.461 dB	-3.002 dB	-3.010 dB

Table 1: SNR Comparison between 8, 16, 32 and 40 bits

As seen in Table 1, as the number of qubit width increases, the SNR decreases.

10.4.2. Logical Barrier and Circuit Optimization

The original experiments' circuits had their modules, split by barriers to indicate sections. As a result, barriers prevent the circuit optimizer to optimize globally; therefore, the circuits' noise may not be minimized by the noise-adaptive compiler. Also, the default optimization level of Qiskit is 1 (available from 0 to 3). The optimization level was later set to 3 (maximum optimization which significantly increased compilation time).

To compare between the original and improved transpilation process, 8-bit String Comparison between simulation, device with barrier and device without barrier distributions comparison is performed.

Fig. 35 shows the comparison between three modes of running the 8-bit String Comparison circuits (with simulator reference).

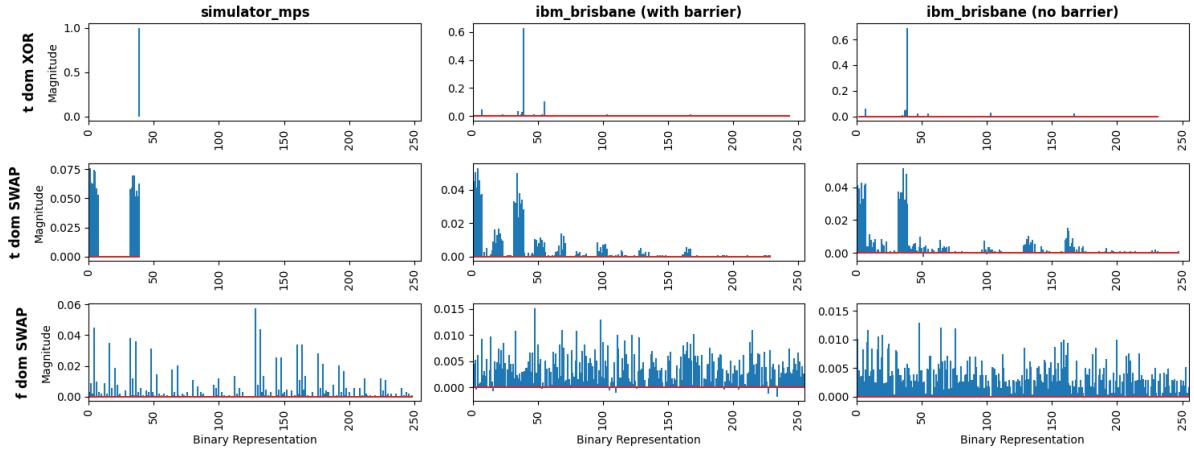


Figure 35: 8-bit String Running Results Mode Comparison

Table 2 shows the comparison of SNR between three modes of running the 8-bit String Comparison circuits (with simulator reference).

	With Barrier	No Barrier + Opt.	Improvement Gain
t dom XOR	8.125 dB	9.736 dB	19.8%
t dom SWAP	6.901 dB	6.815 dB	-1.3%
f dom SWAP	0.817 dB	0.836 dB	2.3%

Table 2: SNR Comparison from 2 modes (8-bit)

10.4.3. Sample Size

As the quasiprobability result is sampled from multiple shots of measurement from the device running the circuit, sample size also affects the accuracy of the distribution.

Fig. 36 shows the comparison between two shots of running the 8-bit String Comparison circuits (with simulator reference).

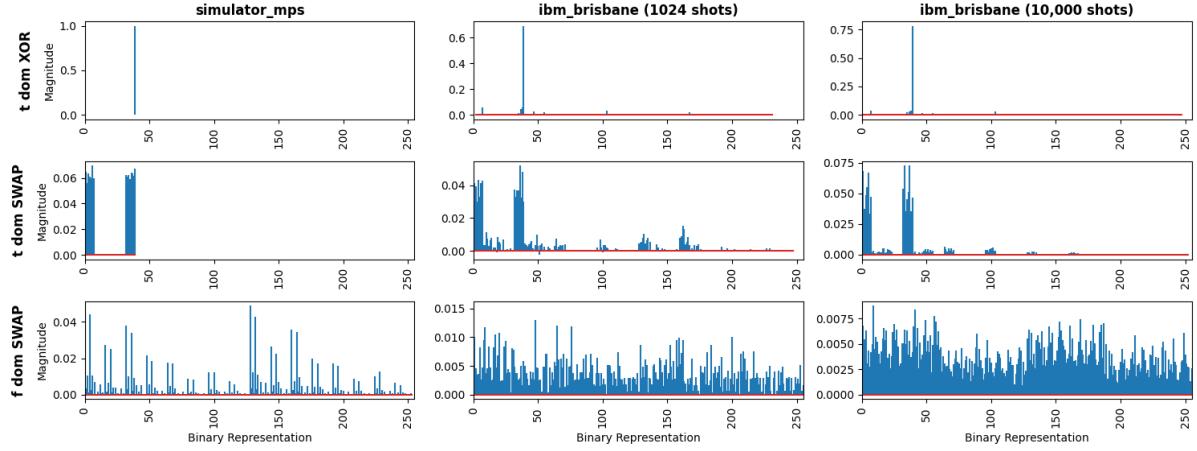


Figure 36: 8-bit String Running Results Shots Comparison

Table 3 shows the comparison of SNR between two shots of running the 8-bit String Comparison circuits (with simulator reference).

	1024 shots	10,000 shots	Improvement Gain
t dom XOR	9.736 dB	12.833 dB	31.8%
t dom SWAP	7.206 dB	10.718 dB	48.7%
f dom SWAP	0.964 dB	0.758 dB	-21.4%

Table 3: SNR Comparison from 2 shots (8-bit)

10.5. Results on Quantum Device (Improved)

To test the improvement when the circuits are optimized and sampled with more shot counts (from 1024 shots to 10,000 shots. Note that the optimization level used for 8-bit circuit was 3 and for other circuits was 2. The transpiler could not compile the 16-bit, 32-bit and 40-bit circuits with level-3 optimization.

10.5.1. Improved 8-bit String Comparison Test

Sequences TTGC and TGCT are the test sample.

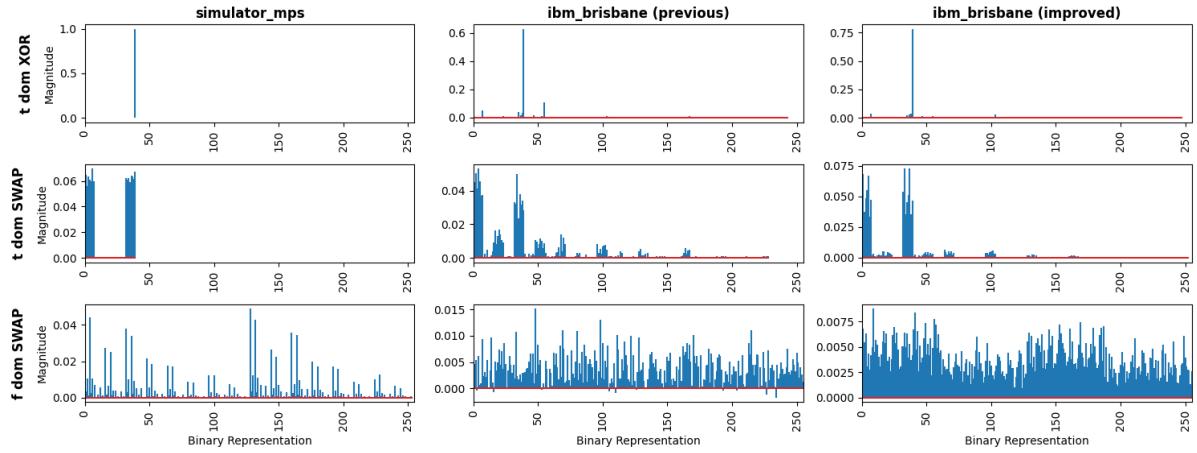


Figure 37: Quasiprobability Distribution Comparison (8 bits)

10.5.2. Improved 16-bit String Comparison Test

Sequences ATGCTTGC and TGCCTGCA are the test sample.

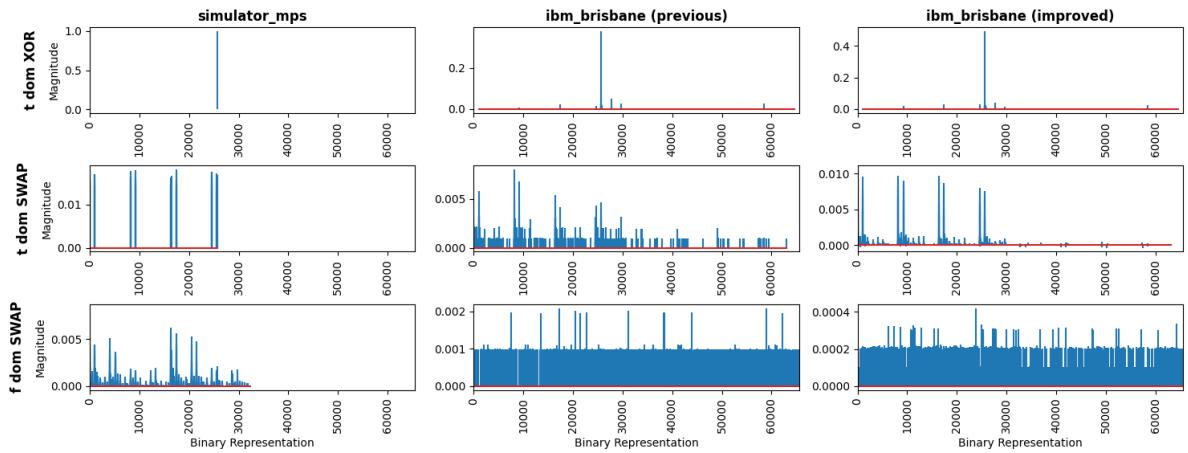


Figure 38: Quasiprobability Distribution Comparison (16 bits)

10.5.3. Improved 32-bit String Comparison Test

Sequences ATGCTTGCAGGGGGGGGG and TGCCTGCACGCGCGCA are the test sample.

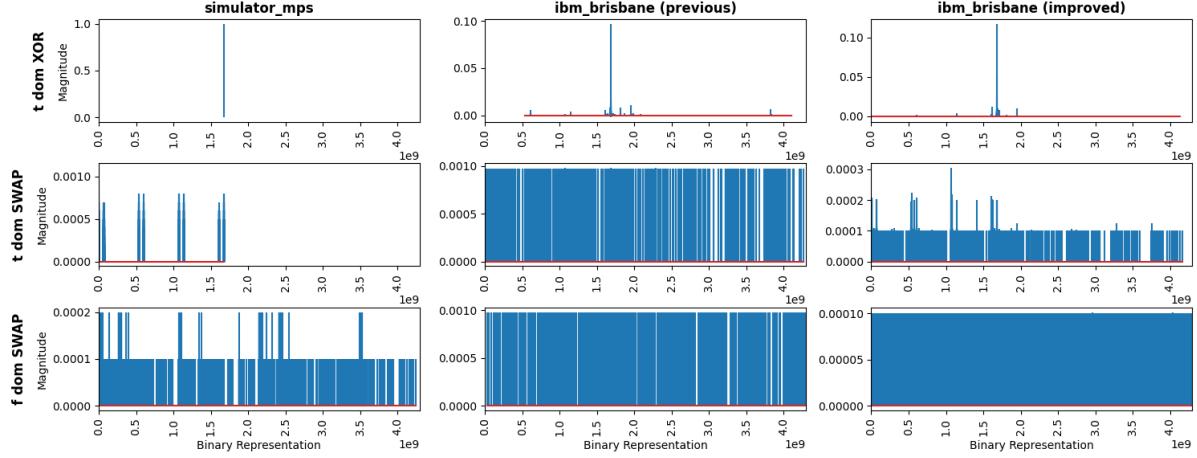


Figure 39: Quasiprobability Distribution Comparison (32 bits)

10.5.4. Improved 40-bit String Comparison Test

Sequences ATGCTTGCAGGGGGGGGGACAG and TGCCTGCACGCGCGCATCAG are the test sample.

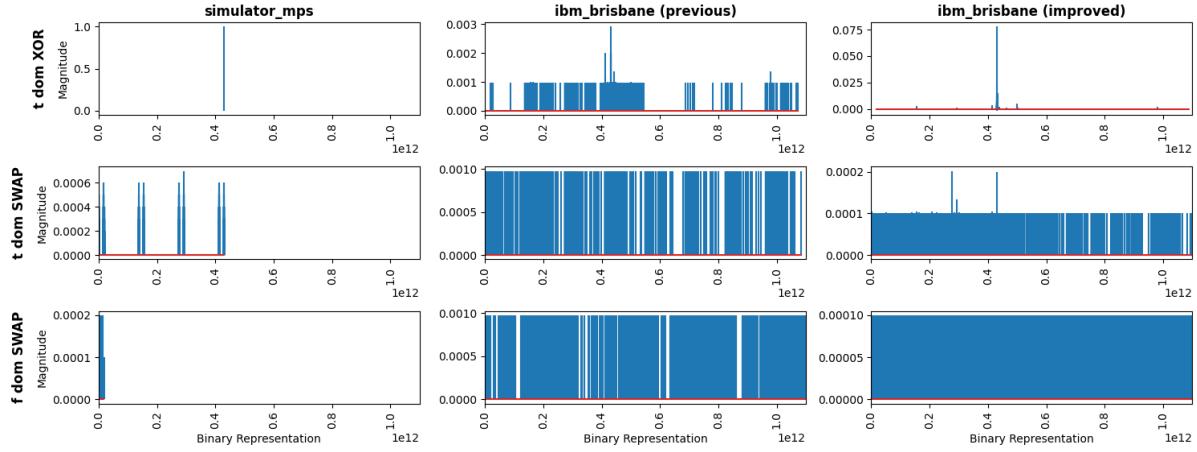


Figure 40: Quasiprobability Distribution Comparison (40 bits)

As illustrated in Figs. 37–40, some conclusions can be drawn as follows.

1. “t dom XOR” has improvement on 8, 16, 32 and 40 bits.
2. “t dom SWAP” has improvement on 8 and 16 bits. 32 bits has considerable amount of noise compared to the result, but the result is still visible.
3. “f dom SWAP” does not have any improvement at all.

11. Summary

In this study, we explored a spectrum of quantum computing approaches to sequence alignment, integrating advanced methods like pattern recognition, 2D graph analysis, Quantum RAM, VQA, and QUBO into our core algorithm. We juxtaposed these against classical dynamic programming techniques, enhancing performance through parallelization. Our innovative methods for aligning sequences and estimating mismatches employed time and frequency domain analyses using quantum algorithms, facilitated by IBM Qiskit.

Our experimental results, derived from simulations and runs on IBM's Eagle r3 quantum device, included comprehensive comparisons of quasiprobability distributions, and strategic improvements in circuit design for optimized performance metrics such as speed, qubit coherence, and noise reduction. We also addressed noise-related challenges by analyzing the impacts of various optimization levels, the presence of logical barriers, and the size of sampling on the integrity of our results.

However, theoretical limitations were encountered with certain algorithms that employ oracle and black-box functions, which are currently impractical to implement on actual quantum hardware. Addressing these limitations is critical for the advancement of quantum analogues to classical sequence alignment problems, setting a clear trajectory for future research in this domain.

12. Further Research

There are many more quantum computing approaches and techniques that can be applied to bioinformatics. There are inevitably a number of areas that could be of interest but were not thoroughly discovered and studied within the limited time frame of this writing. Here are proposals on some topics that are feasible for quantum computing and bioinformatics fields, improving efficiency, resilience, and performance.

12.1. Mitigating Noise

As shown and discussed in Section 10., the noise detectable in the current quantum device is relatively high (commonly SNR < 20dB). To combat with noise problem, many methods can be introduced to the circuit and system which can improve SNR of the algorithm as follows.

1. Applying Grover's Diffusion operator (inversion-amplification) to the circuit to amplify the amplitude of signal and reduce the amplitude of unwanted noise from the system.
2. Introducing Error Correction Code (ECC)/Quantum Error Correction (QEC) for fault-tolerance computing.
3. Improving the quantum computing device at hardware and control levels.

12.2. Interpreting QFT Result

The Quantum Fourier Transform is a cornerstone in quantum computing, instrumental for encoding the frequency domain characteristics of quantum states. In the context of sequence alignment, QFT is applied to approximate the frequency domain mismatch of two encoded bitstrings. While its implementation can provide rich data, the interpretation of QFT results is not straightforward and warrants further research.

Interpreting the output of the QFT is crucial for converting quantum information into meaningful biological insights. The frequency domain data can reveal underlying patterns and mismatches in genetic sequences, which are essential for understanding genomic functions and relationships.

The primary challenge in interpreting QFT results lies in the quantum nature of the data, which includes complex amplitudes and phase information that do not have direct classical analogs. Interpreting the results from QFT in both magnitude difference and phase difference rely on more advanced and robust quantum algorithms, e.g., QPE which utilizes oracle and fractional binary summation. Future research should aim to demystify the quantum information provided by QFT, making it as actionable as possible for biological research and applications.

12.3. Alternative Problem Statements

Traditional sequence alignment algorithms often reformulate the biological problem into a string matching task, which is then addressed using various computational techniques. However, this translation may not always capture the nuances of biological sequences. Exploring alternative problem statements that stay closer to the biological nature of the problem could lead to new insights and solutions.

The string matching approximation simplifies the alignment problem but may overlook biological factors such as the functional consequences of certain mutations or the three-dimensional structure of DNA and proteins. These limitations suggest the need for alternative frameworks that can account for the multi-faceted nature of sequence alignment.

Without converting the problem to string matching, quantum algorithms can potentially exploit the unique properties of biological sequences directly:

1. **Energy-Based Models:** Utilizing quantum systems to model the energy landscapes of biological molecules, providing a more natural representation of sequence alignment as an energy minimization problem. Quantum annealing or the Variational Quantum Eigensolver (VQE) could be explored to find the ground state of a biological system that corresponds to the optimal alignment.
2. **Graph-Theoretic Approaches:** Representing sequences as quantum graphs where alignment is viewed as a problem of finding the optimal graph matching or subgraph isomorphism. Graph-theoretic problems could be tackled by quantum algorithms designed for graph problems, such as the quantum walk algorithm, which could be adapted for sequence alignment.
3. **Quantum Machine Learning:** Developing quantum machine learning models that learn to align sequences by directly processing the biological data without the need to recast it as a string matching problem. Quantum machine learning models may employ quantum versions of neural networks or kernel methods that can be trained on quantum representations of the sequence data.

12.4. Quantum RNA Folding

Quantum RNA Folding represents a novel and promising frontier in computational biology. Leveraging the principles of quantum computing, this approach aims to address the complexities of RNA folding, which involves predicting the secondary and tertiary structures from its nucleotide sequence. The RNA folding problem is particularly challenging due to the vast number of possible configurations, making traditional computational methods resource-intensive. Quantum computing, with its inherent parallelism (by interference) and ability to represent multiple states simultaneously, offers a potential solution to these challenges. The quantum superposition and entanglement properties could enable the evaluation of numerous possible folding patterns in a fraction of the time required by classical computers.

At present, quantum algorithms for RNA folding are in the nascent stages. Some preliminary models have been proposed, but extensive research is required to realize practical and scalable quantum algorithms for RNA folding.

The implications of a successful quantum approach to RNA folding are vast, with potential applications in drug discovery, understanding genetic diseases, and synthetic biology. As RNA plays a crucial role in many cellular processes, better folding algorithms could significantly advance our understanding of life at a molecular level.

As the field of quantum computing matures, the prospects for Quantum RNA Folding will become clearer. Future research should focus on algorithmic development, error mitigation strategies, and the integration of quantum algorithms with existing bioinformatics pipelines. The ultimate goal is to harness quantum computing to reveal insights into RNA biology that were previously impossible to obtain with classical methods.

12.5. Motif Finding

While the proposed method using time-domain and frequency-domain swap tests within QFT has been initially developed for sequence alignment, its underlying principles may be more aptly suited for motif finding. Motif finding involves searching for recurring patterns across multiple sequences, which can often be represented as a problem of identifying common primitives within different frequency components.

12.5.1. Algorithm Suitability Rationales

The swap test's capability to compare quantum states could be leveraged to identify motifs by detecting similarities in the frequency domain representations of sequences. Since motifs are essentially recurring patterns, they may manifest as peaks in the frequency domain that are consistent across different sequences.

12.5.2. Method Adaptation for Motif Finding

To adapt the time-domain and frequency-domain swap tests for motif finding, the following modifications and considerations could be made:

1. **Targeted Frequency Analysis:** Modify the QFT to focus on frequency ranges where motifs are more likely to appear, possibly through a preprocessing step that narrows down the search space.
2. **Pattern Recognition in Quantum States:** Develop algorithms that recognize patterns within the quantum states that correspond to potential motifs.

3. **Thresholding and Scoring:** Implement a quantum thresholding scheme to differentiate between significant and insignificant frequency domain matches, which would help in scoring potential motifs.

12.5.3. Challenges in Quantum Motif Finding

There will be some challenges in Quantum motif finding. For instance, motifs can vary in length and may not be perfectly conserved, which facilitates a flexible approach to pattern matching. Additionally, in the current quantum technologies, the noise inherent in current quantum systems could obscure the subtle frequency domain signatures of motifs.

12.5.4. Proposed Quantum Algorithms

Future research can focus on proposing specific quantum algorithms that can efficiently execute the motif finding process. For example, Algorithms that can handle variable-length patterns in the quantum domain; quantum clustering algorithms that group similar frequency domain features, indicating the presence of motifs; and quantum error correction methods tailored to enhance the fidelity of motif detection.

12.5.5. Impact on Genomic Research

The application of this method to motif finding could revolutionize genomic research by enabling the rapid identification of regulatory elements and protein-binding sites. This, in turn, could have significant implications for understanding gene expression and cellular function.

References

- [1] J. A. Aborot and H. N. Adorna, *String matching using quantum fourier transform*.
- [2] D. Cantone, S. Faro, A. Pavone, and C. Viola, “Quantum circuits for fixed substring matching problems,” Aug. 2023. [Online]. Available: <http://arxiv.org/abs/2308.11758>.
- [3] A. V. Cherkas and S. A. Chivilikhin, “Quantum adder of classical numbers,” vol. 735, Institute of Physics Publishing, Aug. 2016. DOI: 10.1088/1742-6596/735/1/012083.
- [4] D. J. Cornwell, “The amplified quantum fourier transform: Solving the local period problem,” Sep. 2010. [Online]. Available: <http://arxiv.org/abs/1010.0033>.
- [5] T. Faorlin, *Quantum fourier transform*, 2020.
- [6] D. M. Fox, C. M. MacDermaid, A. M. Schreij, M. Zwierzyna, and R. C. Walker, “Rna folding using quantum computers,” *PLoS Computational Biology*, vol. 18, 4 Apr. 2022, ISSN: 15537358. DOI: 10.1371/journal.pcbi.1010032.
- [7] G. Goos, J. Hartmanis, J. Van, *et al.*, *Lncs 3251 - grid and cooperative computing - gcc 2004*, 2004.
- [8] J. Gruska, *QUANTUM COMPUTING*.
- [9] M. Incudini, F. Tarocco, R. Mengoni, A. D. Pierro, and A. Mandarino, “Computing graph edit distance with algorithms on quantum devices,” Nov. 2021. DOI: 10.1007/s42484-022-00077-x. [Online]. Available: <http://arxiv.org/abs/2111.10183> <http://dx.doi.org/10.1007/s42484-022-00077-x>.
- [10] P. K. and K. N., *Quantum pattern recognition for local sequence alignment*, 2017.
- [11] M. R. I. Khan, S. Shahriar, and S. F. Rafid, “A linear time quantum algorithm for pairwise sequence alignment,” Jul. 2023. [Online]. Available: <http://arxiv.org/abs/2307.04479>.
- [12] M. Khan and A. Miranskyy, “String comparison on a quantum computer using hamming distance,” Jun. 2021. [Online]. Available: <http://arxiv.org/abs/2106.16173>.
- [13] B. Langmead, *Global alignment*. [Online]. Available: www.langmead-lab.org/teaching-materials.
- [14] B. Langmead, *Local alignment*. [Online]. Available: www.langmead-lab.org/teaching-materials.
- [15] Y. S. Lee, Y. S. Kim, and R. L. Uy, “Serial and parallel implementation of needleman-wunsch algorithm,” *International Journal of Advances in Intelligent Informatics*, vol. 6, pp. 97–108, 1 Mar. 2020, ISSN: 25483161. DOI: 10.26555/ijain.v6i1.361.
- [16] O. B. Lindvall, *Quantum methods for sequence alignment and metagenomics*.
- [17] Lukac and Perkowski, “Book introduction and quantum mechanics,” in.
- [18] S. Y. Madsen, F. K. Marqversen, S. E. Rasmussen, and N. T. Zinner, “Multi-sequence alignment using the quantum approximate optimization algorithm,” Aug. 2023. [Online]. Available: <http://arxiv.org/abs/2308.12103>.

- [19] V. Menon and A. Chattopadhyay, “Quantum string comparison method,” May 2020. [Online]. Available: <http://arxiv.org/abs/2005.08950>.
- [20] D. Musk, “A comparison of quantum and traditional fourier transform computations,” 2020. DOI: 10.22541/au.160614804.47667838/v1. [Online]. Available: <https://doi.org/10.22541/au.160614804.47667838/v1>.
- [21] L. Ruiz-Perez and J. C. Garcia-Escartin, “Quantum arithmetic with the quantum fourier transform,” Nov. 2014. DOI: 10.1007/s11128-017-1603-1. [Online]. Available: <http://arxiv.org/abs/1411.5949%20http://dx.doi.org/10.1007/s11128-017-1603-1>.
- [22] K. Suksen, N. Benchasattabuse, and P. Chongstitvatana, “Compact genetic algorithm with quantum-assisted feasibility enforcement,” *ECTI Transactions on Computer and Information Technology*, vol. 16, pp. 422–435, 4 Dec. 2022, ISSN: 22869131. DOI: 10.37936/ecti-cit.2022164.247821.
- [23] G. D. Varsamis, I. G. Karafyllidis, K. M. Gilkes, *et al.*, *Quantum gate algorithm for reference-guided dna sequence alignment*.
- [24] Q. Wang, R. Li, and M. Ying, “Equivalence checking of sequential quantum circuits,” Nov. 2018. DOI: 10.1109/TCAD.2021.3117506. [Online]. Available: <http://arxiv.org/abs/1811.07722%20http://dx.doi.org/10.1109/TCAD.2021.3117506>.

A Quantum Circuits

1.1. 16-bit Time-domain comparison with XOR

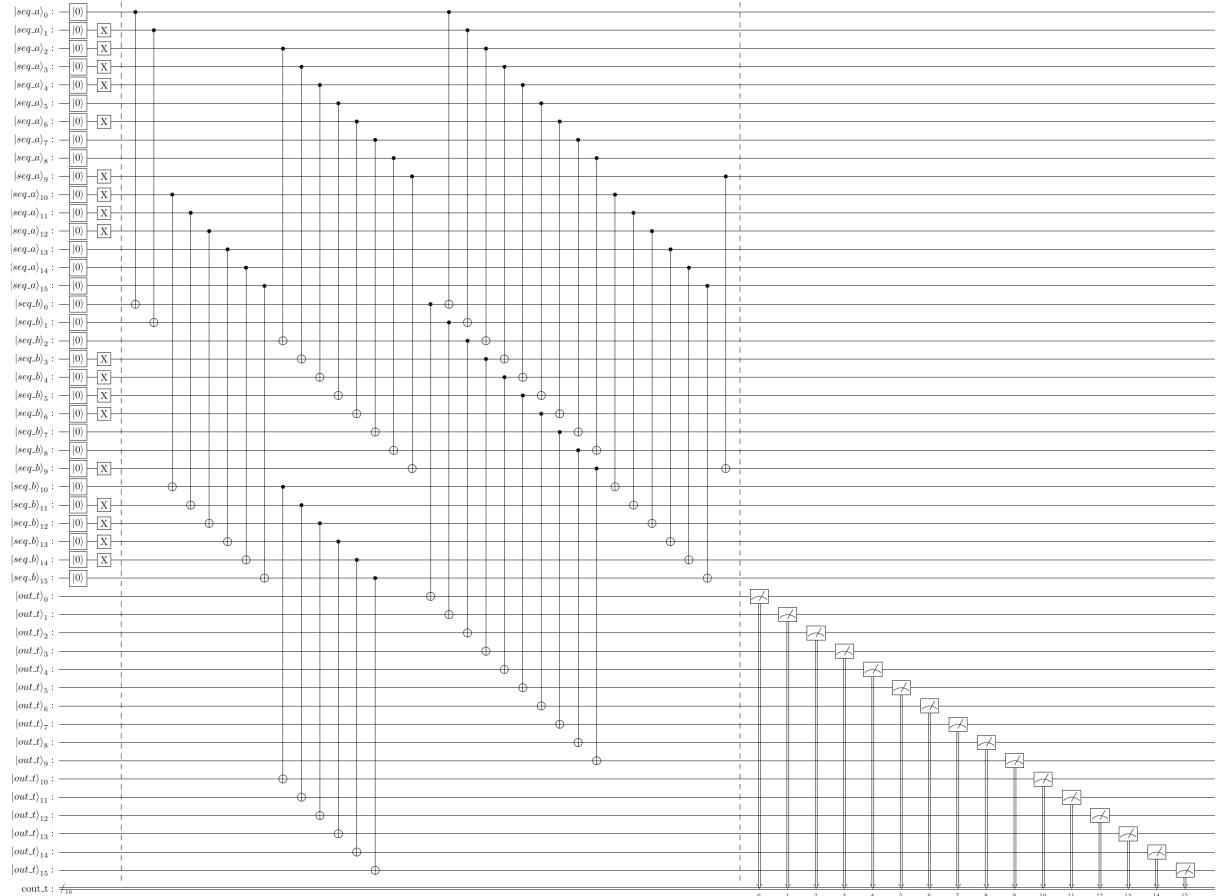


Figure 41: Full Circuit (Decomposed 1 level)

1.2. 16-bit Time-domain comparison with Swap Test

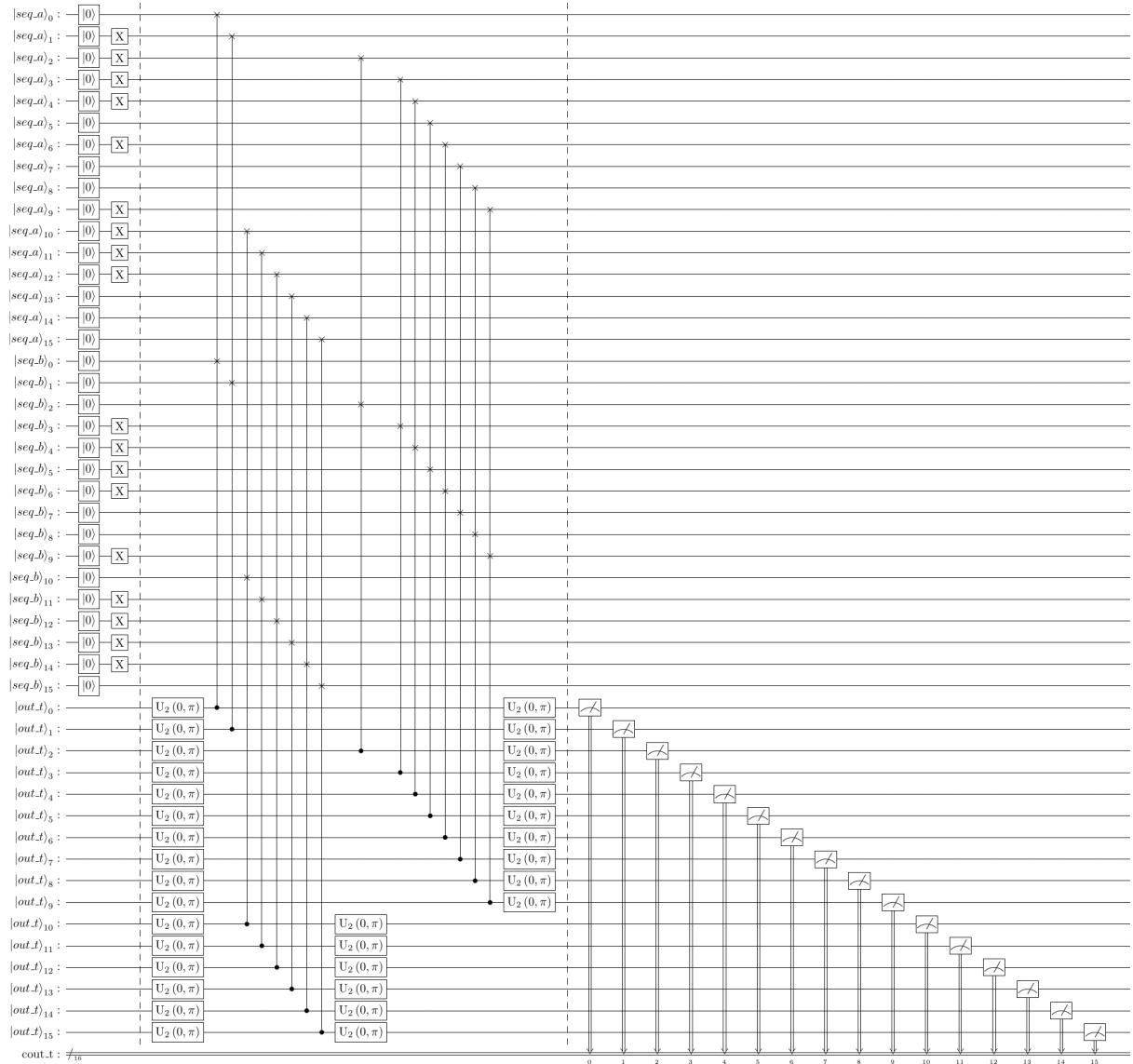


Figure 42: Full Circuit (Decomposed 1 level)

1.3. 16-bit Frequency-domain comparison with Swap Test



Figure 43: Full Circuit (Decomposed 1 level)

1.4. 8-bit Dual domain comparison with Swap Test

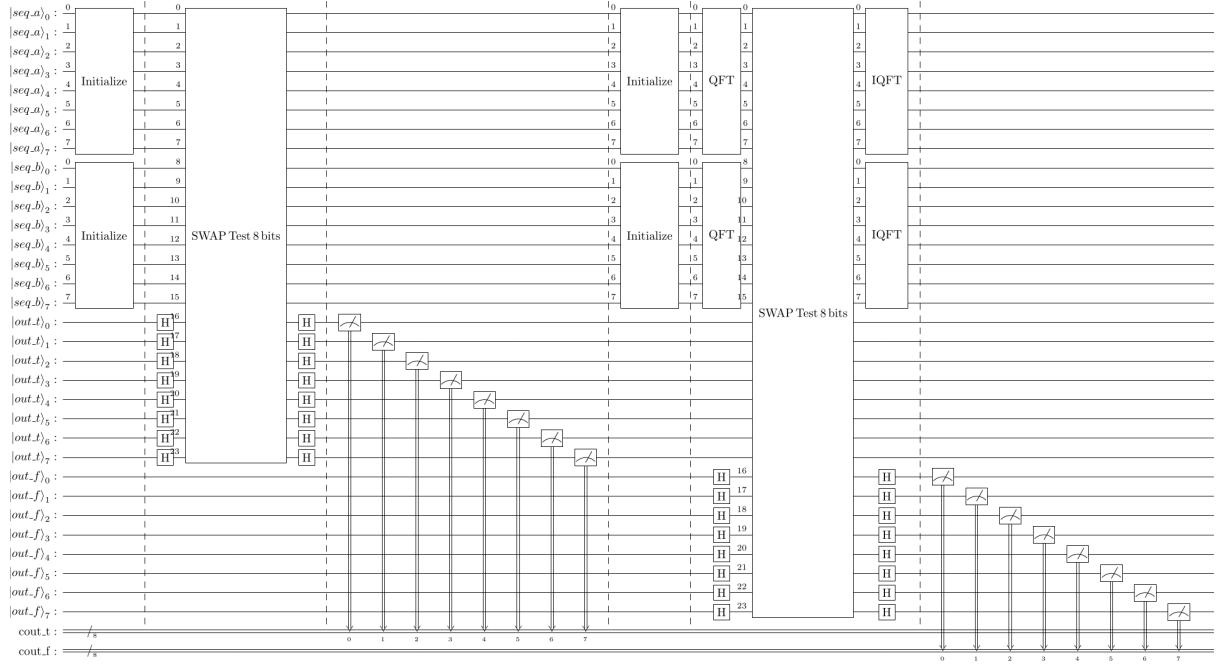


Figure 44: Full Circuit (Top-level)

B Classical Algorithms C Source Code

2.1. main.c

```
#include <stdio.h>
#include "bio_cpu.h"
#include "fasta.h"

#define MAX_STR_LEN (4000ull * 1000ull * 1000ull)

char join_char = '*';
score_t match, mismatch, gap_penalty;
sequence_t seq_a, seq_b;
void **F;

int main(int argc, char **argv) {
    if (argc != 6 && argc != 7)
        exit(1);

    FASTAFILE *ffp;

    // First Sequence
    ffp = OpenFASTA(argv[4]);
    if (ffp != NULL) {
        ReadFASTA(ffp, &seq_a.data, &seq_a.len);
        CloseFASTA(ffp);
    } else {
        seq_a.data = argv[4];
        seq_a.len = strlen(seq_a.data, MAX_STR_LEN);
    }

    // Second Sequence
    ffp = OpenFASTA(argv[5]);
    if (ffp != NULL) {
        ReadFASTA(ffp, &seq_b.data, &seq_b.len);
        CloseFASTA(ffp);
    } else {
        seq_b.data = argv[5];
        seq_b.len = strlen(seq_b.data, MAX_STR_LEN);
    }

    if (seq_a.len < 2 || seq_b.len < 2) {
        if (seq_a.data != argv[4] && seq_a.data != NULL)
            free(seq_a.data);
        if (seq_b.data != argv[5] && seq_b.data != NULL)
            free(seq_b.data);
        exit(2);
    }

#ifndef _OPENMP
    omp_set_max_active_levels(1);
#endif

    match = strtod(argv[1], NULL);
    mismatch = strtod(argv[2], NULL);
    gap_penalty = strtod(argv[3], NULL);
```

```

if (argc == 7) join_char = *argv[6];

// seq_a.len >= seq_b.len always, swap if not
if (seq_a.len < seq_b.len) {
    size_t tmp = seq_a.len;
    seq_a.len = seq_b.len;
    seq_b.len = tmp;

    char *stmp = seq_a.data;
    seq_a.data = seq_b.data;
    seq_b.data = stmp;
}

char *o1 = NULL;
char *o2 = NULL;
char *om = NULL;
size_t len;
score_t score;

nw_generate();

#endif DEBUG_PRINT
size_t len_a = seq_a.len + 1;
size_t len_b = seq_b.len + 1;

printf("      ");
for (size_t j = 0; j < len_b; ++j) {
    printf("%4zu", j);
}
printf("\n");

for (size_t i = 0; i < len_a; ++i) {
    printf("%4zu", i);
    for (size_t j = 0; j < len_b; ++j) {
        printf("%4.0f", ((score_t (*)[len_b]) F)[i][j]);
    }
    printf("\n");
}
#endif
nw_backtrack(&o1, &o2, &om,
              &len, &score);

printf("%zu\n", len);
printf("%.12f\n", score);
printf("%s\n", o1);
if (argc == 7)
    printf("%s\n", om);
printf("%s\n", o2);

free(F);
free(o1);
free(o2);
free(om);

if (seq_a.data != argv[4] && seq_a.data != NULL)
    free(seq_a.data);
if (seq_b.data != argv[5] && seq_b.data != NULL)

```

```
    free(seq_b.data);
    exit(0);
}
```

2.2. bio_cpu.h

```
#ifndef ALGO_CPU_BIO_CPU_H
#define ALGO_CPU_BIO_CPU_H

#include <stdlib.h>
#include <string.h>
#include <stdarg.h>

#ifndef _OPENMP
#include <omp.h>
#endif

#define DIV_MOD(T, Q, R, A, B)          T Q = (A) / (B); T R = (A) % (B);
#define MAX_BLOCK_WIDTH                64ULL // to fit 64KB Cache with double
#define MAX_BLOCK_SIZE                 (MAX_BLOCK_WIDTH * MAX_BLOCK_WIDTH)

typedef double score_t;

typedef struct {
    void ***ptr;
    size_t full_h;
    size_t full_w;
    size_t i_begin;
    size_t j_begin;
    size_t i_end;
    size_t j_end;
} block_t;

typedef struct {
    char *data;
    size_t len;
} sequence_t;

#if defined(__GNUC__)
__attribute__((always_inline))
#elif defined(_MSC_VER)
__forceinline
#endif
extern score_t default_comparator(char a, char b);

#ifndef max
extern score_t max(score_t a, score_t b);
#endif

extern int find_max(size_t argc, ...);

extern void **new_matrix(size_t row, size_t col);

extern char *new_string(size_t len);

extern void nw_generate(void);

extern void impl_nw_generate_1(void);

extern void impl_nw_generate_1_block_recur(block_t *block,
```

```
        int is_subdivide);

extern void impl_nw_generate_1_block_iter(block_t *block);

extern void nw_backtrack(char **out_a,
                         char **out_b,
                         char **out_match,
                         size_t *aligned_len,
                         score_t *score);

extern void sw_generate(void ***F,
                       const char *seq_a, const char *seq_b,
                       size_t len_a, size_t len_b,
                       score_t gap_penalty);

extern void sw_backtrack(char **out_a, char **out_b, char **out_match,
                        size_t *aligned_len, score_t *score,
                        void **F,
                        const char *seq_a, const char *seq_b,
                        size_t len_a, size_t len_b,
                        score_t gap_penalty);

#endif
```

2.3. bio_cpu.c

```
#include "bio_cpu.h"

extern char join_char;
extern score_t match, mismatch, gap_penalty;
extern sequence_t seq_a, seq_b;
extern void **F;

#if defined(__GCC__)
__attribute__((always_inline))
#elif defined(_MSC_VER)
__forceinline
#endif
score_t default_comparator(char a, char b) { return (a == b) ? match :
mismatch; }

#ifndef max
score_t max(score_t a, score_t b) { return (a > b) ? a : b; }
#endif

int find_max(size_t argc, ...) {
    int max, current;

    va_list args;
    va_start(args, argc);

    max = va_arg(args, int);

    for (size_t i = 1; i < argc; ++i) {
        current = va_arg(args, int);
        if (current > max)
            max = current;
    }

    va_end(args);
    return max;
}

void **new_matrix(size_t row, size_t col) {
    score_t (*mat)[col] = calloc(1, sizeof(score_t[row][col]));
    if (mat == NULL)
        exit(1);
    return (void **) mat;
}

char *new_string(size_t len) {
    char *str = (char *) calloc(len + 1, sizeof(char));
    if (str == NULL)
        exit(1);
    return str;
}

void nw_generate(void) {
    impl_nw_generate_1();
}

void impl_nw_generate_1(void) {
```

```

size_t len_a_new = seq_a.len + 1;
size_t len_b_new = seq_b.len + 1;
F = new_matrix(len_a_new, len_b_new);
score_t (*pF)[len_b_new] = (score_t (*)[len_b_new]) F;

#ifndef _OPENMP
#pragma omp parallel default(none) shared(pF, gap_penalty, len_a_new,
    len_b_new)
{
#endif

// Initialize
#ifndef _OPENMP
#pragma omp for nowait schedule(static)
#endif
for (size_t i = 1; i < len_a_new; ++i)
    pF[i][0] = gap_penalty * (score_t) i;

// Initialize
#ifndef _OPENMP
#pragma omp for nowait schedule(static)
#endif
for (size_t j = 1; j < len_b_new; ++j)
    pF[0][j] = gap_penalty * (score_t) j;

#ifndef _OPENMP
}
#endif

// Block calculation starts at i=0, j=0.
size_t i_begin = 0;
size_t j_begin = 0;
block_t init_block = {&F, len_a_new, len_b_new,
    i_begin, j_begin,
    len_a_new, len_b_new};

impl_nw_generate_1_block_recur(&init_block, 0);
//    impl_nw_generate_1_block_iter(&init_block);
}

void impl_nw_generate_1_block_iter(block_t *block) {
    size_t full_hx = block->i_end;
    size_t full_wx = block->j_end;

    DIV_MOD(size_t, full_i, rem_i,
        full_hx, MAX_BLOCK_WIDTH)
    DIV_MOD(size_t, full_j, rem_j,
        full_wx, MAX_BLOCK_WIDTH)

    size_t count_i = full_i + (rem_i != 0);
    size_t count_j = full_j + (rem_j != 0);

    size_t block_hx = full_hx < MAX_BLOCK_WIDTH ? full_hx : MAX_BLOCK_WIDTH
    ;
    size_t block_wx = full_wx < MAX_BLOCK_WIDTH ? full_wx : MAX_BLOCK_WIDTH
    ;
}

```

```

score_t (*pF)[block->full_w] = (score_t (*)[block->full_w]) (*(block->
ptr));

// Access blocks in anti-diagonal pattern (for parallelism)
#ifndef _OPENMP
#pragma omp parallel default(none) \
shared(count_i, count_j, rem_i, rem_j, block, block_hx, block_wx, pF,
seq_a, seq_b, gap_penalty)
#endif
for (size_t h_block_i = 0; h_block_i < (count_i + count_j - 1); ++
h_block_i) {
    size_t j_start = h_block_i < count_j ? 0 : h_block_i - count_j + 1;
    size_t j_end = h_block_i < count_i ? h_block_i + 1 : count_i;

#ifndef _OPENMP
#pragma omp for schedule(static)
#endif
    for (size_t h_block_j = j_start; h_block_j < j_end; ++h_block_j) {
        size_t block_i = h_block_j;
        size_t block_j = h_block_i - h_block_j;

        size_t ii_start = block_i * block_hx;
        size_t jj_start = block_j * block_wx;
        size_t ii_stop = rem_i != 0 && block_i == count_i - 1 ?
                        block_i * block_hx + rem_i :
                        (block_i + 1) * block_hx;
        size_t jj_stop = rem_j != 0 && block_j == count_j - 1 ?
                        block_j * block_wx + rem_j :
                        (block_j + 1) * block_wx;

        // Access within the block in row sweeping pattern (better
        // cache locality)
        for (size_t i = ii_start; i < ii_stop; ++i) {
            for (size_t j = jj_start; j < jj_stop; ++j) {
                if (i == 0 || j == 0) continue;
                score_t _match = pF[i - 1][j - 1] +
                                default_comparator(seq_a.data[i - 1],
                                                   seq_b.data[j - 1]);
                score_t _delete = pF[i - 1][j] + gap_penalty;
                score_t _insert = pF[i][j - 1] + gap_penalty;
                pF[i][j] = max(_match, max(_delete, _insert));
            }
        }
    }
}

void impl_nw_generate_1_block_recur(block_t *block,
                                    int is_subdivide) {
    size_t hx = block->i_end - block->i_begin;
    size_t wx = block->j_end - block->j_begin;

    if (is_subdivide && hx * wx >= MAX_BLOCK_SIZE) {
        size_t new_hx = hx / 2;
        size_t new_wx = wx / 2;

        // partitioning blocks 0-3: top-left, top-right, bottom-left,
        // bottom-right
    }
}

```

```

block_t b = {0};

b.ptr = block->ptr;
b.full_w = block->full_w;
b.full_h = block->full_h;

// block 0
b.i_begin = block->i_begin;
b.j_begin = block->j_begin;
b.i_end = block->i_begin + new_hx;
b.j_end = block->j_begin + new_wx;
impl_nw_generate_1_block_recur(&b, 1);

#ifndef _OPENMP
#pragma omp parallel sections default(none) shared(block, new_hx, new_wx)
{
#pragma omp section
{
#endif
// block 1
block_t b1 = {
    block->ptr,
    block->full_h,
    block->full_w,
    block->i_begin,
    block->j_begin + new_wx,
    block->i_begin + new_hx,
    block->j_end
};
impl_nw_generate_1_block_recur(&b1, 1);

#ifndef _OPENMP
}
#endif
#pragma omp section
{
#endif
// block 2
block_t b2 = {
    block->ptr,
    block->full_h,
    block->full_w,
    block->i_begin + new_hx,
    block->j_begin,
    block->i_end,
    block->j_begin + new_wx
};
impl_nw_generate_1_block_recur(&b2, 1);

#ifndef _OPENMP
}
}
#endif
// block 3
b.i_begin = block->i_begin + new_hx;
b.j_begin = block->j_begin + new_wx;
b.i_end = block->i_end;
b.j_end = block->j_end;
impl_nw_generate_1_block_recur(&b, 1);

```

```

    } else {
        score_t (*pF)[block->full_w] = (score_t (*)[block->full_w]) (*(block->ptr));

#ifndef DIAGONAL_ARRAY_ACCESS
    for (size_t i = block->i_begin; i < block->i_end; ++i) {
        for (size_t j = block->j_begin; j < block->j_end; ++j) {
            if (i == 0 || j == 0) continue;
            score_t _match = pF[i - 1][j - 1] +
                default_comparator(seq_a.data[i - 1],
                                   seq_b.data[j - 1]);
            score_t _delete = pF[i - 1][j] + gap_penalty;
            score_t _insert = pF[i][j - 1] + gap_penalty;
            pF[i][j] = max(_match, max(_delete, _insert));
        }
    }
#else
    for (size_t i = 0; i < (hx + wx - 1); ++i) {
        size_t j_start = i < wx ? 0 : i - wx + 1;
        size_t j_end = i < hx ? i + 1 : hx;

        for (size_t j = j_start; j < j_end; ++j) {
            size_t diag_i = block->i_begin + (j);
            size_t diag_j = block->j_begin + (i - j);

            score_t _match = (pF)[diag_i - 1][diag_j - 1] +
                default_comparator(seq_a.data[diag_i - 1],
                                   seq_b.data[diag_j - 1]);
            score_t _delete = (pF)[diag_i - 1][diag_j] + gap_penalty;
            score_t _insert = (pF)[diag_i][diag_j - 1] + gap_penalty;
            (pF)[diag_i][diag_j] = max(_match, max(_delete, _insert));
        }
    }
#endif
}
}

void nw_backtrack(char **out_a,
                  char **out_b,
                  char **out_match,
                  size_t *aligned_len,
                  score_t *score) {

    score_t (*pF)[seq_b.len + 1] = (score_t (*)[seq_b.len + 1]) F;
    size_t max_len = seq_a.len + seq_b.len;
    *score = 0;

    char *tmp_a = new_string(max_len);
    char *tmp_b = new_string(max_len);

    size_t i = seq_a.len;
    size_t j = seq_b.len;
    size_t idx = 0;

    while (i > 0 && j > 0) {
        score_t ms = default_comparator(seq_a.data[i - 1], seq_b.data[j - 1]);
        if (pF[i][j] == pF[i][j - 1] + gap_penalty) {

```

```

        tmp_a[idx] = '-';
        tmp_b[idx] = seq_b.data[j - 1];
        *score += gap_penalty;
        --j;
    } else if (pF[i][j] == pF[i - 1][j] + gap_penalty) {
        tmp_a[idx] = seq_a.data[i - 1];
        tmp_b[idx] = '-';
        *score += gap_penalty;
        --i;
    } else if (pF[i][j] == pF[i - 1][j - 1] + ms) {
        tmp_a[idx] = seq_a.data[i - 1];
        tmp_b[idx] = seq_b.data[j - 1];
        *score += ms;
        --i;
        --j;
    }
    ++idx;
}

while (j > 0) {
    tmp_a[idx] = '-';
    tmp_b[idx] = seq_b.data[j - 1];
    *score += gap_penalty;
    --j;
    ++idx;
}

while (i > 0) {
    tmp_a[idx] = seq_a.data[i - 1];
    tmp_b[idx] = '-';
    *score += gap_penalty;
    --i;
    ++idx;
}

*out_a = new_string(idx);
*out_b = new_string(idx);
*out_match = new_string(idx);

#endif _OPENMP
#pragma omp parallel sections default(none) shared(out_a, out_b, out_match,
    tmp_a, tmp_b, idx, join_char)
{
#pragma omp section
    for (size_t k = 0; k < idx; ++k) (*out_a)[k] = tmp_a[idx - k - 1];
#pragma omp section
    for (size_t k = 0; k < idx; ++k) (*out_b)[k] = tmp_b[idx - k - 1];
}
for (size_t k = 0; k < idx; ++k) (*out_match)[k] = ((*out_a)[k] == (*
    out_b)[k]) ? join_char : ' ';
#else
    for (size_t k = 0; k < idx; ++k) {
        (*out_a)[k] = tmp_a[idx - k - 1];
        (*out_b)[k] = tmp_b[idx - k - 1];
        (*out_match)[k] = ((*out_a)[k] == (*out_b)[k]) ? join_char : ' ';
    }
#endif

```

```
*aligned_len = idx;  
free(tmp_a);  
free(tmp_b);  
}
```

2.4. fasta.h

Huge Credits to <https://github.com/casalebrunet/fasta> on GitHub.

```
/* fasta.h
 * Declarations for simple FASTA i/o library
 * SRE, Sun Sep  8 05:37:38 2002 [AA2721, transatlantic]
 * CVS $Id$ */
#include <stdio.h>

#define FASTA_MAXLINE 512      /* Requires FASTA file lines to be <512
   characters */

typedef struct fastafile_s {
    FILE *fp;
    char buffer[FASTA_MAXLINE];
} FASTAFILE;

extern FASTAFILE *OpenFASTA(char *seqfile);

extern int ReadFASTA(FASTAFILE *fp, char **ret_seq, size_t *ret_L);

extern void CloseFASTA(FASTAFILE *ffp);
```

2.5. fasta.c

Huge Credits to <https://github.com/casalebrunet/fasta> on GitHub.

```
/* Simple API for FASTA file reading
 * for Bio5495/BME537 Computational Molecular Biology
 * SRE, Sun Sep  8 05:35:11 2002 [AA2721, transatlantic]
 * CVS $Id$ */
 */

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <ctype.h>

#include "fasta.h"

FASTAFILE *OpenFASTA(char *seqfile) {
    FASTAFILE *ffp;

    ffp = (FASTAFILE *) malloc(sizeof(FASTAFILE));
    ffp->fp = fopen(seqfile, "r"); /* Assume seqfile exists &
                                    readable! */
    if (ffp->fp == NULL) {
        free(ffp);
        return NULL;
    }
    if ((fgets(ffp->buffer, FASTA_MAXLINE, ffp->fp)) == NULL) {
        free(ffp);
        return NULL;
    }
    return ffp;
}

int ReadFASTA(FASTAFILE *ffp, char **ret_seq, size_t *ret_L) {
    char *s;
    char *seq;
    size_t n;
    size_t nalloc;

    /* Peek at the lookahead buffer; see if it appears to be a valid FASTA
     * descline.
    */
    if (ffp->buffer[0] != '>') return 0;

    /* Everything else 'til the next descline is the sequence.
     * Note the idiom for dynamic reallocation of seq as we
     * read more characters, so we don't have to assume a maximum
     * sequence length.
    */
    seq = (char *) malloc(sizeof(char) * 128); /* allocate seq in
                                                blocks of 128 residues */
    nalloc = 128;
    n = 0;
    while (fgets(ffp->buffer, FASTA_MAXLINE, ffp->fp)) {
        if (ffp->buffer[0] == '>') break; /* a-ha, we've reached the
                                         next descline */
    }
}
```

```

for (s = ffp->buffer; *s != '\0'; s++) {
    if (!isalpha(*s)) continue; /* accept any alphabetic character */
    */

    seq[n] = *s;                      /* store the character, bump
length n */
    n++;
    if (nalloc == n)                  /* are we out of room in seq? if so
, expand */
    {
        /* (remember, need space for the final
'\0')*/
        nalloc += 128;
        char *tmp = (char *) realloc(seq, sizeof(char) * nalloc);
        if (tmp == NULL) {
            free(seq);
            CloseFASTA(ffp);
            exit(3);
        }
        seq = tmp;
    }
}
seq[n] = '\0';

*ret_seq = seq;
*ret_L = n;
return 1;
}

void CloseFASTA(FASTAFILE *ffp) {
    fclose(ffp->fp);
    free(ffp);
}

```