國立臺灣大學電機資訊學院資訊工程學系暨研究所

碩士論文

Department of Computer Science & Information Engineering

College of Electrical Engineering & Computer Science

National Taiwan University

Master Thesis

Quokka: 基於檔案之量子計算模擬器

Quokka: a File-based Quantum Computing Simulator

徐乃威

Nai-Wei Hsu

指導教授: 洪士灝 博士

Advisor: Shih-Hao Hung Ph.D.

中華民國 112 年 6 月

June, 2023

doi:10.6342/NTU202300661

Acknowledgements

首先，我要特別感謝洪士灝老師。感謝您提供了頂級的設備和研究環境，讓我能夠充分利用這些資源進行深入的研究。在一次次的討論中，您給予我寶貴的建議和指導，使我的研究能夠不斷進步和完善。

衷心感謝我的家人，特別是他們在經濟上對我無私的支援。您們的無私奉獻和信任，讓我能夠專注於研究工作，不受經濟壓力的困擾。您們一直是我堅強的後盾，給予我力量和勇氣，讓我堅持不懈地追求學術的道路。

此外，我要特別感謝我的女友。在我遇到困難和挫折時，您的陪伴和理解讓我感到溫暖和安慰，我深深感激您無私的付出。

感謝我的學長。在繁忙的學業中抽出寶貴的時間，陪我修改論文、投影片，不計其數的深夜討論讓我受益匪淺。您們的專業知識和經驗分享，為我的研究提供了寶貴的指引和啟示。

感謝您們的支持和鼓勵，是您們的無私奉獻和幫助讓我能夠順利完成這項研究。在未來的人生道路上，我將倍加努力，不負您們的期望和厚愛。

再次衷心地感謝您們！

doi:10.6342/NTU202300661

doi:10.6342/NTU202300661

摘要

隨著量子運算發展出解決各式應用的方法與軟硬體快速地發展, 許多具有高度平行化的量子模擬框架陸續被推出。量子模擬需要大量的記憶體和計算能力，尤其是當需要模擬的 qubit 數目非常大時。為了解決這個問題，基於多節點的解決方案被推出以應對大 qubit 的案例。然而多節點的模擬意味著使用者除了應付記憶體增加的費用外，還要額外對計算單元進行付費。這樣的惡性循環使得量子電腦不能被普及到普羅大眾。在此篇論文，我們提出一個新穎的解決方案，Quokka，緩和上述的額外成本。Quokka 在軟體層面以資料流的角度下去實作，在硬體層面可透過ＳＳＤ這個特化的硬體來達到最佳的運算效能。由於資料串流需要更多的調參，我們定義了針對不同硬體做調整的格式並給出如何在特定硬體下做出各種適配的調整。在我們的實驗中，我們能達到 memory-based 版本的 0.8 倍，並在價格中省下了 80 倍預算，這是之前的論文所媲美不了的全新研究。

關鍵字：量子計算、量子電路模擬、量子電路模擬器、平行計算、效能分析

doi:10.6342/NTU202300661

doi:10.6342/NTU202300661

Abstract

With the growing demand for quantum computing in a variety of applications, numerous highly parallel quantum circuit simulation (QCS) frameworks have been introduced. These frameworks utilize multiple machines to provide the memory capacity required for large-scale QCS. Unfortunately, such approaches are expensive and inefficient as the performance is bounded by the bandwidth of the interconnection network. In this thesis, we develop a storage-based quantum circuit simulator is proposed to provide a cost-effective alternative to existing memory-based schemes. To efficiently utilize multi-core CPUs and solid-state disk arrays, a qubit representation is developed, together with a threading model, to enable efficient simulation of quantum circuits of different qubit sizes with parameterized configurations. Moreover, several performance issues are addressed by proposed strategies. The experimental results show that the proposed storage-based simulation provides a platform-aware approach to increase the qubit size supported by the simulator. The size of quantum circuit is bounded by the volume sizes of the adopted

doi:10.6342/NTU202300661

SSDs and the resulted performance is limited by the maximum bandwidth of the storage system. Thanks to the combination of high capacity, good data bandwidth, and low cost of SSDs, as well as the expandability of I/O devices, our approach provides a cost-effective and flexible solution for large-scale quantum circuit simulation. As a result, the proposed approach reduced costs by 80 times compared to memory-based version.

Keywords: Quantum Computing, Quantum Circuit Simulation, Quantum Circuit Simulator, Parallel Computing, Performance Analysis

doi:10.6342/NTU202300661

Contents

doi:10.6342/NTU202300661

doi:10.6342/NTU202300661

List of Figures

doi:10.6342/NTU202300661

doi:10.6342/NTU202300661

List of Tables

doi:10.6342/NTU202300661

doi:10.6342/NTU202300661

Chapter 1 Introduction

Quantum circuit simulation emulating quantum computing systems on classical computers is a popular way to evaluate quantum circuits for novel algorithms. Besides, while physical quantum computers are available as cloud computing services, they are vulnerable to noise interference, regarding the noisy intermediate-scale quantum [12]. In such a case, quantum circuit simulation is helpful for the development and validation of quantum algorithms.

Typical quantum circuit simulations, which adopt the Schrödinger-style *full-state* simulation [10, 20, 30, 31] producing intermediate results for validation and debug purposes, demand for a large memory space during the simulations. This poses a great challenge since the required memory space grows exponentially [22, 27] with deeper quantum circuits or larger qubit sizes. Given an N-qubit circuit, it requires O(2^N) memory space to keep the states. To respond to the need, existing works leverage large-scale supercomputers to provide sufficient memory for keeping the states. For example, the Fugaku supercomputer [25] comprises 158,976 machine nodes and 4.7 petabytes of memory to cover the simulations of 48-qubit circuits. Unfortunately, the supercomputer-based approaches place an entry barrier for average users and hence, hinder the promotion of the development of quantum computing.

doi:10.6342/NTU202300661

In this work, we propose a storage-based quantum circuit simulator that adopts the full-state simulation and extends the memory space with the secondary memory, such as solid-state drives (SSDs), as a remedy for the growing memory space problem. The storage-based solution can be deployed on commodity machines, enjoying the parallel computing power released by many-core processors. Besides, as storage devices have larger sizes than memories, the storage-based solution is able to enlarge the simulation limit. For example, the estimated qubit limit of QuEST [20] is 33-qubit in double precision on our hardware platform with 256GB of main memory, whereas our storage-based simulator is able to achieve the 39-qubit quantum circuit simulation with 16TB SSD on the same machine.

To further show that the storage-based simulator is a better design alternative, Table 1.1 compares the specifications and performance data between the typical memory-based and our proposed storage-based quantum circuit simulations, where the experiments are done on our hardware platform listed in Section 4.1 and QuEST is used to represent the state-of-the-art, typical quantum circuit simulator. The *cost-delay product* (CDP) [29] is used as a figure of merit to quantify the cost efficiency of the quantum circuit simulation, where the *delay* is defined as the averaged simulation time for the Hadamard gate. Our experimental data show that while the storage-based solution has a slightly higher delay (because of its lower memory bandwidth), it achieves up to 61 times more cost efficient than the memory-based alternative.

The contributions achieved by this work are listed as follows.

1. A novel quantum circuit simulator with storage devices is proposed, achieving higher cost efficiency than the memory-based alternative. Our work is open source to the

doi:10.6342/NTU202300661

Public[1]. To the best of our knowledge, the proposed solution is one of the pioneer works of using secondary memory to enlarge the qubit simulation limit. More about the storage-based solutions are compared and discussed in Section 2.

2. A profiling and logging module is embedded in our proposed full-state simulator to report the performance statistics of simulated quantum gates (e.g., the counts of simulated gates) and the elapsed time of simulated gates. The example of the collected performance data is provided in Section 3.1.

3. A data representation scheme for N-qubit is devised to facilitate parallel simulations on a many-core processor platform. The related performance tuning suggestions are presented to facilitate the practical use of our proposed simulator onto different hardware platforms.

The remainder of this paper is organized as follows. Section 2 presents the background and the existing works of quantum circuit simulations. Section 3 presents the design of the proposed work. Section 3.5 describes the important aspect for building the proposed parallel quantum circuit simulator. Section 4 reports the delivered performance results. Section 5 concludes this work.

[1]The proposed storage-based simulator can be downloaded via the link: https://github.com/drazermega7203/Quokka

doi:10.6342/NTU202300661

Table 1.1: Comparison of the tradeoffs between the memory- and storage-based quantum circuit simulations. A lower CDP means a better cost efficiency for quantum circuit simulation.

	Memory capacity	**Bandwidth (GB/s)**	**Cost (USD/GB)**	**Delay (ms/H gate)**	**Cost-delay product (CDP)**
Storage-based quantum circuit simulation over NVMe SSD (PCIe 4.0)	2 TB	7.0	0.09 [5]	0.74∼398.4	**0.06∼35.85**
Memory-based quantum circuit simulation over DRAM (DDR4)	32 GB	28.8	7.17 [2]	0.31∼305.6	2.22∼2191.15
Relative ratio	0.01	4.1	80	0.4∼0.7	37∼61

doi:10.6342/NTU202300661

Chapter 2 Background

This section briefly introduces full-state quantum circuit simulation (QCS) in Chapter 2.1. While QCS can be used to support the development of quantum algorithms and quantum computers, there are challenges for implementing QCS in practice, which are discussed in Chapter 2.2.

2.1 Quantum Circuit Simulation

A pure state in quantum mechanics is represented by a state vector or wave function that contains complete information about the quantum system, and it can be used to compute the probabilities of the possible outcomes measured on the system. A quantum circuit consists of a sequence of gate operations and qubit data, where the operations performed on the data across different qubits to accomplish a high-level application logic. From the mathematical point of view, a gate operation, which can be represented as a matrix operation, is a tensor product performing on one or more qubits and changes the quantum states of the qubits. Furthermore, in order to model the noises in the quantum circuits, the density matrix representation can be used.

doi:10.6342/NTU202300661

2.1.1 State Vector

In the context of an N-qubit quantum system, the state vector is typically denoted as $|\Psi\rangle = \Sigma_{i=0}^{2^N-1}\alpha_i |i\rangle$, where $\alpha_i \in \mathbb{C}$ represents the probability amplitude of state $|i\rangle$. It can be represented as a column vector in a Hilbert space, which is a complex vector space endowed with an additional mathematical structure for describing quantum systems. In a 20-qubit system, if each probability amplitude is stored in double precision, it would take 16 bytes per amplitude. Therefore, to store the full state vector, it would require 16 MB of memory. Similarly, for a 40-qubit system, it would require 16 TB of memory to store the full state vector. When a measurement is performed on a quantum system, the state vector falls down to one of the possible measurement outcomes. The measurement process can be defined mathematically by a projection operator, which can be expressed as a matrix projecting the state vector onto the measurement outcome.

2.1.2 Gate Operations

A quantum gate operation performs on one or more qubits and affects the states of the quantum system. A single-qubit gate operation G applied on the j-th qubit of the state vector can be denoted as $G_j |\Psi\rangle := I_{N-1} \otimes ... \otimes I_{j+1} \otimes G \otimes I_{j-1} \otimes ... \otimes I_0 |\Psi\rangle$, where G_j is an $2^N * 2^N$ matrix multiply on the state vector. In a general view, this matrix multiplication can be represented as follows.

$$\begin{bmatrix} \alpha'_{\star...\star 0_j \star...\star} \\ \alpha'_{\star...\star 1_j \star...\star} \end{bmatrix} = G \begin{bmatrix} \alpha_{\star...\star 0_j \star...\star} \\ \alpha_{\star...\star 1_j \star...\star} \end{bmatrix}$$

doi:10.6342/NTU202300661

where $\star$ refer to any bit value when represent index i of α_i in binary representation. Furthermore, when considering a two-qubit unitary gate U_{jk} for updating the state vector on both the j-th and k-th qubits, the matrix form can be rewritten as follows.

$$\begin{bmatrix} \alpha'_{\star\ldots\star 0_j \star\ldots\star 0_k \star\ldots\star} \\ \alpha'_{\star\ldots\star 0_j \star\ldots\star 1_k \star\ldots\star} \\ \alpha'_{\star\ldots\star 1_j \star\ldots\star 0_k \star\ldots\star} \\ \alpha'_{\star\ldots\star 1_j \star\ldots\star 1_k \star\ldots\star} \end{bmatrix} = U_{jk} \begin{bmatrix} \alpha_{\star\ldots\star 0_j \star\ldots\star 0_k \star\ldots\star} \\ \alpha_{\star\ldots\star 0_j \star\ldots\star 1_k \star\ldots\star} \\ \alpha_{\star\ldots\star 1_j \star\ldots\star 0_k \star\ldots\star} \\ \alpha_{\star\ldots\star 1_j \star\ldots\star 1_k \star\ldots\star} \end{bmatrix}$$

2.1.3 Density Matrix

Given a set of N-qubit quantum systems with a state vector $|\Psi_s\rangle = \Sigma_{i=0}^{2^N-1}\alpha_i |i\rangle$, the corresponding density matrix can be defined as $\rho_s = |\Psi_s\rangle \langle\Psi_s|$ to describe the same quantum system. The key advantage of the density matrix is that it cannot only be used to describe systems that are not in pure states, such as systems that are subject to decoherence or other forms of environmental noise but help to investigate entanglement properties of composite quantum systems. It is a Hermitian, positive semi-definite matrix for characterizing the state of a mixed quantum state, which is represented by a weighted sum of pure states and the weights correspond to the probabilities of finding the system in each of the pure states. To apply a gate as the view of the density matrix, it can be leveraged by the Choi-Jamiolkowski isomorphism [11] as follows.

$$\begin{aligned} \rho' &= \Sigma_s p_s G_i(\Sigma_{j=0}^{2^N-1}\Sigma_{k=0}^{2^N-1}\alpha_{j,k,s} |j\rangle \langle k|)G_i^\dagger \\ &\rightarrow \Sigma_s p_s G_{i+N}^\dagger G_i(\Sigma_{j=0}^{2^{2N}-1}\alpha_{j,s} |j\rangle) \end{aligned} \tag{2.1}$$

doi:10.6342/NTU202300661

For a 20-qubit system, unlike state vector, storing the full density matrix for this system would require storing $2^{20} \times 2^{20} = 2^{40}$ complex numbers, which is much larger than storing the state vector. If each complex number is also stored in double precision, it would require 1 TB for storage. As Equation 2.1 indicated, applying a gate operation on a density matrix of an N-qubit system requires twice the computation compared to a 2N-qubit system stored in a state vector. This is due to the density matrix containing information about the correlations between the different qubits, making it a more computationally expensive representation of a quantum state compared to the state vector. In our work, we present a gate operation that can be applied to a density matrix. However, due to the computational overhead involved, we recommend avoiding the use of density matrices unless necessary.

2.2 Challenges in Quantum Circuit Simulation

With the swift evolution of quantum computing and optimization algorithms, the quantum circuit simulator and software on classic computers offer valuable guidance for understanding a given quantum program. In practice, the following are challenges in designing quantum circuit simulators.

1. **Performance.** In the field of quantum circuit simulators, a substantial amount of research is focused on enhancing the efficiency of quantum operations. For homomorphic systems, some studies [17, 20] have adopted a multi-threaded approach to optimize CPU utilization. To take advantage of the single instruction multiple data (SIMD) features, some simulators [1, 19, 30, 31] have been developed to support the AVX instruction set. With the widespread adoption of graphical processing units (GPUs), some simulators [10, 14, 20] have proposed a heterogeneous-based

doi:10.6342/NTU202300661

approach to leverage the thousands of cores available in GPUs for massive parallelism. At a higher level, for sequential gate operations on adjacent qubits, some performance tricks such as gate fusion [19, 30] and qubit reordering [26] can be used to reduce the frequency of gate operations and improve the access pattern of gate operations, respectively.

2. **Scalability.** To cope with the exponentially growing memory demands for larger-scale simulations, the distributed implementation approach has been proposed [16, 19, 21, 24, 33]. In such an approach, the quantum state, divided into multiple sub-states, is simulated independently on a separate computer or node. However, the communication overhead and hardware cost remain a significant challenge for the system developers. To attain better cost-effectiveness, the storage-based methodology referred to as SnuQS [26] utilizes secondary devices as storage units and develops a sub-circuit partitioning algorithm to reduce the IO frequency for their RAID-0 device. Unfortunately, apart from utilizing RAID-0, which has a low level of reliability, there is no advanced optimization for IO, which results in significantly reduced versatility and suboptimal performance for pure single-gate operators.

3. **Comprehension.** In order to facilitate comprehension of the intricate behavior of quantum systems and gain insights into their properties and dynamics, various software tools have been developed that incorporate visualization techniques [3, 7, 8], statistical analysis methods [32, 34], and debugging tools [18, 23]. Visualization tools render a given quantum circuit expressed in programs or description files as an impressive illustration, deepening understanding of quantum circuits. Some interactive tools can also convert visual images back into description files, enabling the adjusted circuit to run on various platforms. To manage the simulation and in-

doi:10.6342/NTU202300661

teractive verification of quantum algorithms, external modules such as statistical analysis and debugging tools are integrated into the simulator.

Overall, the development of quantum circuit simulation still heavily relies on high-spec machines, which greatly limits the accessibility of quantum computing. Furthermore, in the studies that rely on storage to reduce the budget, there is a lack of effective optimization of system IO transmission. This has prompted us to add heuristic profiling modules to do the deep performance analysis from a gate perspective. In light of these considerations, we propose to design the simulator with multi-functional properties of high performance, scalability, and cost-efficiency.

doi:10.6342/NTU202300661

Chapter 3 Methodology

In this chapter, an SSD-based quantum circuit simulation scheme is proposed to enlarge the size of quantum circuits. The overall architecture is first introduced in Chapter 3.1. The qubit representation developed in this work to enable large-scale, parallel quantum circuit simulation over storage devices is described in Chapter 3.2. Furthermore, the parallel simulation paradigms inherited from the qubit representation are presented in Chapter 3.3.

3.1 Architectural Overview

The three-layer architecture of the proposed quantum circuit simulator is illustrated in Figure 3.1. The simulator accepts the user inputs to perform the quantum circuit simulation, and the quantum states are kept in the storage devices and buffered by the memory. The three layers are described as follows.

1. **The input circuit and configuration.** The inputs to the simulator include a quantum circuit file and a configuration file. The quantum circuit file can be transformed from the popular OpenQASM format [13] to specify the involved quantum gates in the circuit file. The configuration file provides the settings for the simulations, without the recompilation of the simulator to take effect. In particular, the configuration

doi:10.6342/NTU202300661

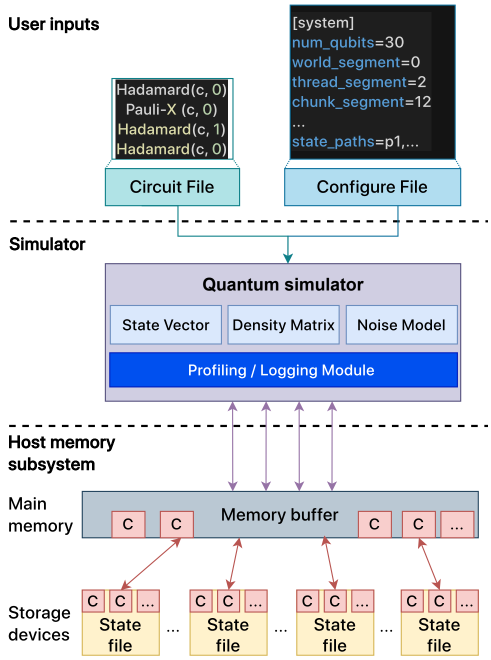

Figure 3.1: The architecture of the quantum circuit simulator.

file can be used to turn on/off the noise model, to set up the simulation parameters (e.g., number of threads), to enable/disable the performance profiling during the simulation, and to log the execution status/intermediate states during the simulation.

2. **The simulator.** The proposed simulator adopts the Schrödinger-style algorithm to perform a gate-by-gate simulation [15, 28]. Given an N qubit circuit, the simulator uses 2^N complex amplitudes in the storage devices, and these state data are retrieved when needed. They are buffered in the main memory and updated in place during the gate-by-gate simulation. A matrix representation to represent state vectors or density matrix is used to facilitate the simulation. The matrix representation helps the traversing of quantum program since it provides a clear view of quantum states.

   Note that a profiling/logging module is built within the simulator for analyzing the simulation performance of the simulated quantum gates. The module is able to collect the dynamic performance data, such as the simulated gate sequence, the counts

doi:10.6342/NTU202300661

of simulated gates, and the simulation time for each gate. Later, the quantum circuit optimization(s) can be judiciously performed based on the collected dynamic performance data. This module can further broaden the optimizations to be performed, as a complement to the static optimizations performed by typical quantum compilers. The example output of the profile data is shown in Figure 3.2 for the simulation of the H-CNOT-H-X gate sequence.

3. **The host memory subsystem.** The states of qubits are stored in the *state files* in the storage devices. These files are opened as ordinary files and are accessed via the standard Linux I/O system calls. Memory buffers are allocated by the simulator for buffering the qubit states for further manipulations. Threads are created with the OpenMP library in the simulator to manipulate the quantum states in parallel. The basic units for content updates of the qubit state are called *chunks*, which are transferred back and forth between the secondary memory and the main memory, as illustrated in Figure 3.1. The following subsections introduce the method used for partitioning the qubit states (e.g., into chunks) and the scheme for parallel qubit state processing.

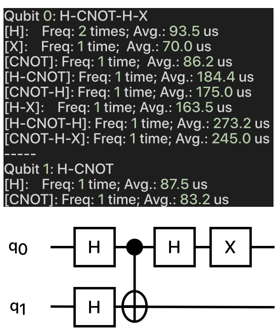

Figure 3.2: An example output of the profiling data offered by our simulator.

doi:10.6342/NTU202300661

3.2 Multithreading and State Files

To facilitate parallel simulations of quantum circuits, a representation of an N-qubit has been designed, as shown in Figure 3.3. When an N-qubit system is considered for the simulations, it is saved in the state files on the storage devices.

When accessing state information from a state file stored in secondary storage, various configuration details need to be considered, such as the batch size for each read and write operation and how tasks should be allocated across threads. For an N-qubit system, as previously mentioned, storing the state vector requires $16 * 2^N$ bytes. Setting the chunk size $B = 2^C$ as the size of each I/O (e.g. 4KB, 8KB, 16KB) from storage can help determine an optimal chunk size for achieving the best simulation performance. Additionally, computation tasks must be allocated across threads for parallel computing. If the state vector were stored in a single file, the system lock for read and write could significantly impact performance. To avoid this, the state vector can be partitioned into state files, with the number of state files based on the number of threads. Each thread only gets access to a single file at a time to ensure efficient computation.

A given N-qubit can then be partitioned into three segments, File, Middle, and Chunk segments. With the partition scheme, the N-qubit can be used as indexes to their states in the files. The first three qubits (belonging to the file segment) form the state file name; taking the 11-qubit in Figure 3.3 as an example, "101" is the name of the state file for the states of the combinations of the rest 8-qubit, such as "110 01110." The partition scheme can be modified via the configuration file to facilitate the parallel execution on the different host environments.

doi:10.6342/NTU202300661

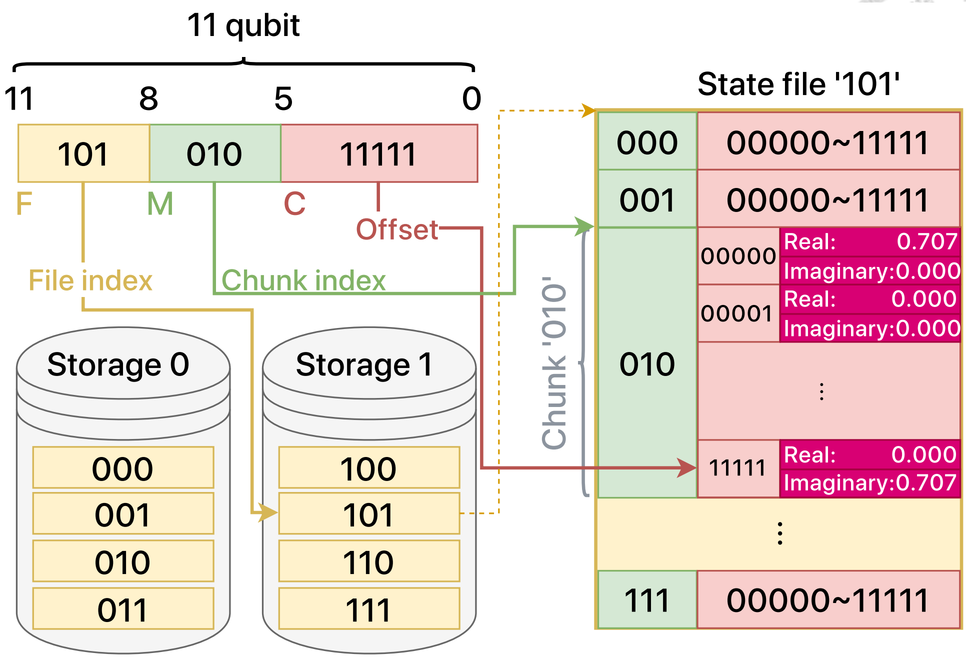

Figure 3.3: An example of the representation of qubits, where the 11-qubit is partitioned into three segments, **F**ile, **M**iddle, and **C**hunk segments.

The parameters used to represent the configurations of the parallel quantum circuit simulations are defined in Table 3.1. N refers to the number of qubits (i.e., N-qubit) to be simulated and the three segments (F, M, and C in Figure 3.3) constitute an N-qubit, implying 2^N complex amplitudes for the states of the N-qubit system and the relationship of $N = F + M + C$. T denotes two to the power of T meaning the total number of threads is equal to 2^T. F represents the number of files that would be opened in the simulation, which is equal to 2^F. C is related to the transmission size of the IO system call. For example, the number of amplitudes accessed by each thread is 2^C, and if a state vector is stored as an eight-byte double-precision number, the chunk size of the specific chunk segment is 2^{C+4}, where the additional one unit is reserved for the complex number, and the additional three units are allocated for double-precision storage. M is used to indicate the total number of chunks for each thread is 2^M. The remaining parameter $targ$ denotes the target qubit operated by the current quantum gate, and $tIdx$ is used as an index to uniquely pointing to one of the threads in the quantum circuit simulation.

doi:10.6342/NTU202300661

When the simulator accesses the state files, it reads/write the entire chunk from/to the SSD via the I/O interface, and the chunk size affects the performance. The optimal chunk size for achieving the best simulation performance depends on the IO subsystem. On the other hand, the value of T is determined by the underlying classical processor, as the simulator creates enough threads to utilize the physical cores provided by the processor. Thus, 2^T should be equivalent to or more than the number of physical cores in typical scenarios. For instance, given 64 physical cores in the system, T should be set to 6 at least to create 64 or more threads. More threads can be useful to take advantage of the hyperthreading feature offered by modern processors and reduce the I/O waits. For the threads to access the state vector stored in the files with minimum interference, the simulator creates at least one file for each thread. Thus, F is set equally to T by default.

In practice, the setting of parameters C and F should depend on the input quantum circuit (number of qubits, N) and the classical computer that is used to perform the simulation. We have developed an empirical method to determine the partition of an N-qubit state. As illustrated in Figure 3.4, given a classical computer with 2^T cores to simulate an N-qubit system, the workflow tries to optimize the setting for C and F based on an empirical tuning process, which is presented and discussed in Chapter 3.5. Note that we assume that $N > T$, as the proposed simulator aims to support large quantum circuit simulation where the state vector/density matrix is too large to be stored in the main memory, and N is typically larger than 15 with density matrix.

doi:10.6342/NTU202300661

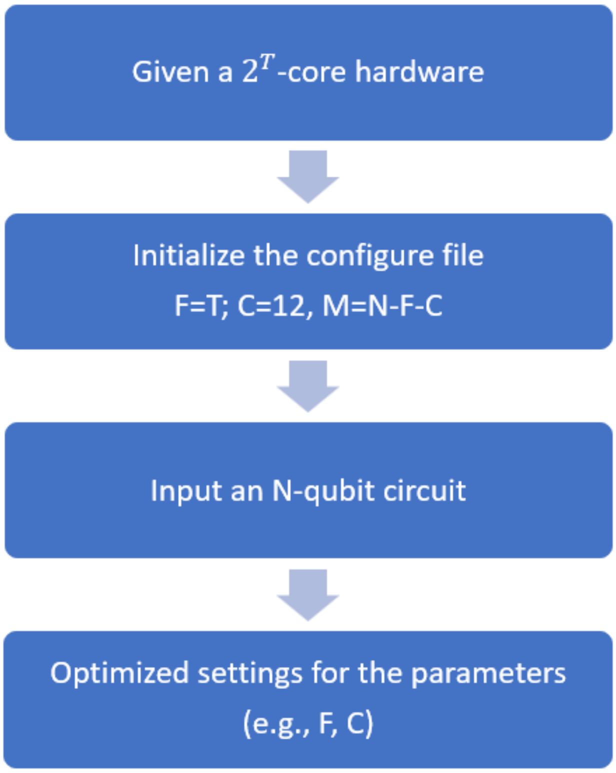

Figure 3.4: An empirical workflow for determining a proper partition for an N-qubit system of the representation shown in Fig. 3.3.

Algorithm 1 The main function for storage-based quantum circuit simulator.

```
set_Ini(Config File)
set_Circuit(Circuit File)
#pragma parallel
for g ← 1 to total_gate do
    gateOps ← gateMap[g]
    Decoder(gateOps, controls, targets)
    #pragma omp barrier
end for
```

3.3 Data Access Patterns in Parallel Simulation

Algorithm 1 is the main function for our storage-based quantum circuit simulator. The purpose of the simulator is to simulate the behavior of quantum circuits using classical computers. The algorithm is composed of a series of steps that are repeated for each gate in the quantum circuit being simulated.

doi:10.6342/NTU202300661

Table 3.1: The parameters used by the proposed quantum circuit simulator.

Parameter (Symbol)	Description
Number of Qubits (N)	The total number of qubits simulated.
Target Qubit ($targ$)	The qubit that is targeted for a specific quantum operation or measurement.
Thread Segment (T)	The total number of threads is 2^T.
Thread Index ($tIdx$)	The index for each distinct thread.
File Segment (F)	The total number of state files is 2^F.
Middle Segment (M)	The total number of chunks is 2^M.
Chunk Segment (C)	The basic unit of transfer for standard IO is 2^C.

- Lines 1-2: The algorithm starts by setting the initial state of the quantum circuit and reading the circuit description from a configuration file and a circuit file, respectively.

- Line 3: We initiate parallel threads here for the following iterations in the for-loop.

- Line 4: The algorithm then starts a loop that runs for each gate in the circuit. The loop index ”g” ranges from 1 to ”$total_gate$,” which is the total number of gates in the circuit.

- Line 5: For each gate, the algorithm first looks up the corresponding gate information such as operations, control-qubits, and target-qubits from a pre-defined gate map.

- Line 6: The algorithm then calls the Decoder function with the gate operations, control-qubits, and target-qubits. The Decoder function is responsible for performing the actual gate operation on the quantum state.

- Line 7: The algorithm then waits for all threads to finish using an OpenMP barrier.

- Line 7: The loop continues for the next gate until all gates have been simulated.

doi:10.6342/NTU202300661

Algorithm 2 The main function for storage-based quantum circuit simulator with Density matrix method available.

```
1:  set_Ini(Config File)
2:  set_Circuit(Circuit File)
3:  #pragma parallel
4:  for g ← 1 to total_gate do
5:      gateOps ← gateMap[g]
6:      Decoder(gateOps, controls, targets)
7:      #pragma omp barrier
8:      if isDensity then
9:          Decoder_Density(gateOps, controls, targets)
10:         #pragma omp barrier
11:     end if
12: end for
```

If the density matrix method is added to the simulator, the main function would need to be modified to include an additional gate operation to perform density matrix method in each iteration of the for loop. The modified main function would look like Algorithm 2, where Lines 8-11 are added to perform density matrix method when the user chooses to do so.

3.3.1 Decoder

Algorithm 3 is a function called the decoder that takes as input a gate operation, control and target qubits, and returns the resulting state vector after applying the gate operation to the system. The decoder function is used in a quantum circuit simulator to implement the gates in the circuit.

The decoder function is designed as a switch-case statement, where the gate operation is used to determine which gate function to call. The function can handle different types of gates, including 1-qubit gates, 2-qubit unitary gates, controlled gates, swap gates, 3-qubit unitary gates, and measurement gates. Each gate function may takes the target and control qubits as input and applies the corresponding gate operation to the qubits.

doi:10.6342/NTU202300661

Algorithm 3 The Decoder function.

Input: gateOps, controls qubit(s), targets qubit(s)
Result: state vector after applied gate.

```
switch gateOps do
    case 1-Qubit Gate
        one_qubit_gate(gateOps, target qubit)
    case 2-Qubit Unitary Gate
        two_qubit_unitary_gate(target qubits)
    case Control-Target Gate
        controlled_gate(gateOps, control qubit, target qubit)
    case Swap gate
        swap_gate(target qubits)
    case 3-Qubit Unitary Gate
        three_qubit_unitary_gate(target qubits)
    case Measure
        measure(target qubit)
```

If the simulator includes the density matrix method, the decoder function will require modification to distinguish between the first and second gates applied. The resulting modified function, decoder2, is shown in Algorithm 4

Algorithm 4 The Decoder_Density function.

Input: gateOps, controls qubit(s), targets qubit(s)
Result: state vector after applied gate.

```
switch gateOps do
    case 1-Qubit Gate
        one_qubit_gate_d(gateOps, target qubit)
    case 2-Qubit Unitary Gate
        two_qubit_unitary_gate_d(target qubits)
    case Control-Target Gate
        controlled_gate_d(gateOps, control qubit, target qubit)
    case Swap gate
        swap_gate_d(target qubits)
    case 3-Qubit Unitary Gate
        three_qubit_unitary_gate_d(target qubits)
```

3.3.2 1-Qubit Gates in Memory-based Simulation

Algorithm 5 is the implementation of a one-qubit gate in QuEST and other memory-based quantum circuit simulators. The algorithm applies a 1-qubit gate operation to a

doi:10.6342/NTU202300661

Algorithm 5 The one_qubit_gate function

```
Global: N-qubit system.
Input: gateOps, target qubit
Result: state vector after applied gate.
for task ← task_start[tid] to task_end[tid] do
    [s0, s1] ← map(task, target qubit)
    [s0, s1] ← simulate_gate(gateOps, [s0, s1])
end for
```

Algorithm 6 The one_qubit_gate_d function.

```
Global: N-qubit system.
Input: gateOps, target qubit.
Result: state vector after applied gate.
for task ← task_start[tid] to task_end[tid] do
    [s0, s1] ← map(task, target qubit)
    [s0, s1] ← simulate_gate_d(gateOps, [s0, s1])
end for
```

state vector by iterating over each element in the vector and performing the necessary calculations.

For an N-qubit system, the state vector holds 2^N probability amplitudes, each representing a specific state. When applying a single qubit gate, there are $2^N/2$ pairs of states that need to be updated. In a conventional memory-based simulation, these $2^N/2$ tasks can be distributed to multiple threads using the task_start[tid] and task_end[tid] (i.e., one single task is equal to two states). After all iterations are completed, an OpenMP barrier is applied to synchronize the threads in Algorithm 2.

Within each iteration of the Algorithm 5, the variable ”task” is mapped to a specific pair of states using division and modulo operations. The division and modulo operations can be simplified by add with another for loop. This mapping process is based on the location of the target qubit and shown in Algorithm 7.

The distance between pairs of states required by gateOps is stored in target_offset and then organized into ”task blocks.” The task blocks are determined by the target qubit's

doi:10.6342/NTU202300661

Algorithm 7 The map function in 1-Qubit Gate function for memory-based simulation.

Input: task, target_qubit
Result: [s0, s1] pair of state

```
target_Offset ← 1 « target_qubit
half_task_block ← target_Offset
task_mod_mask ← half_task_block-1
task_block ← 2 * target_Offset
s0 ← ((task≫target_qubit)≪(target_qubit+1)) | (task&task_mod_mask)
s1 ← s0 | target_Offset
return [s0, s1]
```

Algorithm 8 Simplifying the map function with nested loops.

Input: gateOps, target_qubit
Result: state vector after applied gate.

```
target_offset ← 1 « target_qubit
half_task_block ← target_offset
num_tasks_per_thread = 2^(N−1) / num_threads
if num_tasks_per_thread < half_task_block then
    task_offset ← num_tasks_per_thread;
else
    task_offset ← half_task_block;
end if
for task ← task_start[tid]; task < task_end[tid]; task+=task_offset do
    s0 ← (task≫target_qubit)≪(target_qubit+1)
    s1 ← s0 + target_offset
    for i=0; i < task_offset; i++ do
        simulate_gate(gateOps, [s0, s1])
        s0++
        s1++
    end for
end for
```


doi:10.6342/NTU202300661

position. If the target qubit is in the upper half of a task block, all the states in that block have the target qubit set to zero. Conversely, if the target qubit is in the bottom half of a task block, all the states in that block have the target qubit set to one. To map the task to the corresponding state pair, the algorithm uses the target_Offset variable, which is calculated as the value obtained by shifting 1 to the left by the target_qubit position. The half_task_block variable is set to the target_Offset, and the task_block variable is set to twice the half_task_block. The variable s0 represents the state index in the pair, and it is computed by integer dividing the task by half_task_block and adding the remainder of the task divided by half_task_block. The variable s1 is then obtained by adding target_Offset to s0. The algorithm returns the pair of states [s0, s1] that corresponds to the given task and target_qubit.

Finally, if we combine this map function we get Algorithm 8, we can see that there's intrinsically locality inside task_block. Also, there are 2 for-loop in Algorithm 8, this is the break point of our storage-based simulation.

3.3.3 1-Qubit Gates in Storage-based Simulation

In storage-based simulation, for efficiency, the state vector needs to be brought into memory chunk-by-chunk before calculation, which needs some modification from Algorithm 8 to Algorithm 9. The algorithm checks the task_block to determine how to access the storage with efficiency. If the task_block size is smaller than the chunk size, then the task_block needed to be processed for the target qubit is inside a chunk (Lines 8-15); otherwise, modifying a task_block requires accesses to multiple chunks. It is also possible that the target qubit points to a task_block which resides across two files on different SSDs, and the algorithm will have the threads access the files in pairs to maximize the IO

doi:10.6342/NTU202300661

bandwidth (Lines 17-29). Finally, if the task_block is less than or equal to the size of device, the algorithm will have the threads access the state vector in chunks simultaneously and perform the gate operation in parallel.

The partitioning of an N-qubit into the three segments facilitates the accessing of the quantum states by the parallel threads. Using a single qubit gate simulation as an example. When a thread attempt to do a gate operation, the operation may be on the chunk segment, the middle segment, or the file segment. As shown Algorithm 9, according to the position of the target qubit, *targ*, the file access pattern is quite different.

The primary objective of this algorithm is to distribute tasks evenly among all threads. It consists of three parts, namely chunk, middle, and file segments. Initially, we obtain the function handler based on the gate operation by opcode. This gate function performs the operation entirely in memory, which is similar to the behavior of other simulators. The operation flow is simple: first, read a chunk of state into memory, perform the gate operation, and then write the results back to the storage. However, in some cases, the gate operation may require reading in multiple chunks of state and writing back multiple chunks of state.

The following describes the details:

- **Within the chunk segment.** The target qubits required by the simulation for certain operations or measurements are at the positions belonged to a chunk segment. These chunks have the same prefix formed by the qubit string of F and M. The corresponding pairs of basis vectors are accessed via the typical Linux file operations (which will be described in the next section) from the state file, and the memory regions are allocated for buffering the contents of the 2^C chunks. Figure 3.5 illus-

doi:10.6342/NTU202300661

Algorithm 9 The storage-based algorithm utilized for the case study of single-gate.

```
void (*gate)(void*, uint64_t) ← get_gate(opCode);
int tId ← omp_get_thread_num();                                  ▷ Thread ID
void* buffer;                                  ▷ Independent for Different Thread ID
task_block ← 1 « (target_qubit+1) * sizeof(double);
uint64_t chunkSize ← 2^C * sizeof(double);
uint64_t fileSize ← 2^N/2^F * sizeof(double);
if task_block < chunkSize then                                   ▷ Chunk Segment
    uint64_t offsetInMem ← 2^targ;
    uint64_t fileOff ← 0;
    for j ← 0 to 2^M − 1 do
        read(tId, buffer, chunkSize, fileOff);
        gate(buffer, offsetInMem);
        write(tId, buffer, chunkSize, fileOff);
        fileOff ← fileOff + chunkSize;
    end for
else if task_block > fileSize then                               ▷ File Segment
    if tId bit-wise and (1 << 2^(targ−M−C)) = 1 then return;
    end if
    uint64_t offsetInMem ← 2^C;
    uint64_t fileOff ← 0;
    int targFile ← tIdx bit-wise xor (1 << 2^(targ−M−C));
    for j ← 0 to 2^(M−1) − 1 do
        read(tId, buffer, chunkSize, fileOff);
        read(targFile, buffer+chunkSize, chunkSize, fileOff);
        gate(buffer, offsetInMem);
        write(tId, buffer, chunkSize, fileOff);
        write(targFile, buffer+chunkSize, chunkSize, fileOff);
        baseOff ← baseOff + chunkSize;
    end for
else                                                             ▷ Middle Segment
    uint64_t targOff ← 1 « target_qubit * sizeof(double);
    uint64_t offsetInMem ← 2^C;
    uint64_t fileOff ← 0;
    for j ← 0 to 2^(M+C−targ−1) − 1 do
        for k ← 0 to 2^(targ−C) − 1 do
            read(tId, buffer, chunkSize, fileOff);
            read(tId, buffer+chunkSize, chunkSize, targOff);
            gate(buffer, offsetInMem);
            write(tId, buffer, chunkSize, fileOff);
            write(tId, buffer+chunkSize, chunkSize, targOff);
            fileOff ← fileOff + chunkSize;
            targOff ← targOff + chunkSize;
        end for
        fileOff ← fileOff + 2^(targ−1);
        targOff ← targOff + 2^(targ−1);
    end for
end if
```


doi:10.6342/NTU202300661

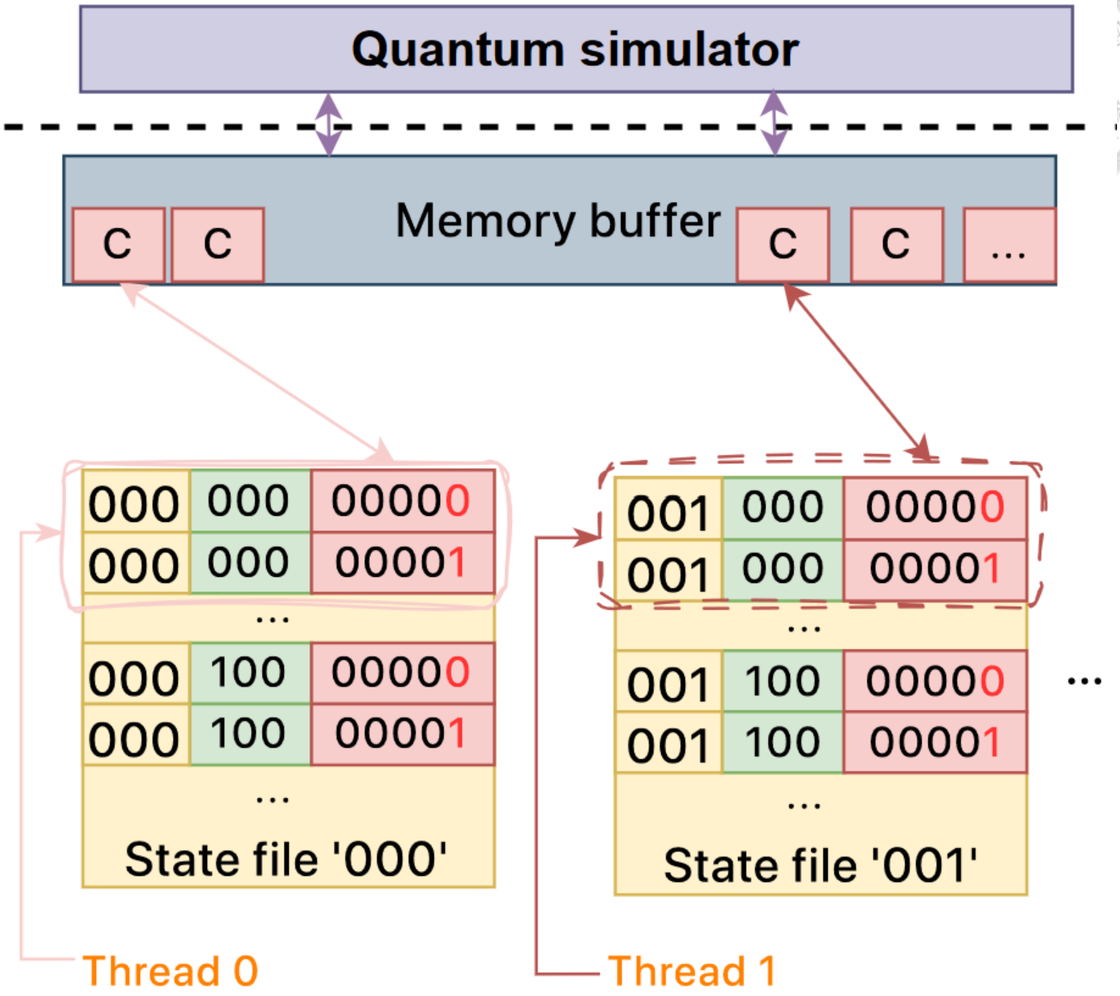

Figure 3.5: The qubit representation and the corresponding accesses to the qubit states by the threads, using the 11-qubit system as an example, where the qubit(s) to be accessed $targ$ is within a chunk segment.

trates the concept, where different threads have the accesses to their respective state files, and the thread index is identical to the file name (the binary bit string). The number of chunks in a state file can be determined by 2^M, where M is the middle segment in the simulation. This also represents the total number of read and write operations for a single thread.

- **Within the file segment.** The target qubits are at the positions of a file segment, which involves the accesses across different state files. The related states of the dispersed chunk segments should be brought to the memory buffer first before they

doi:10.6342/NTU202300661

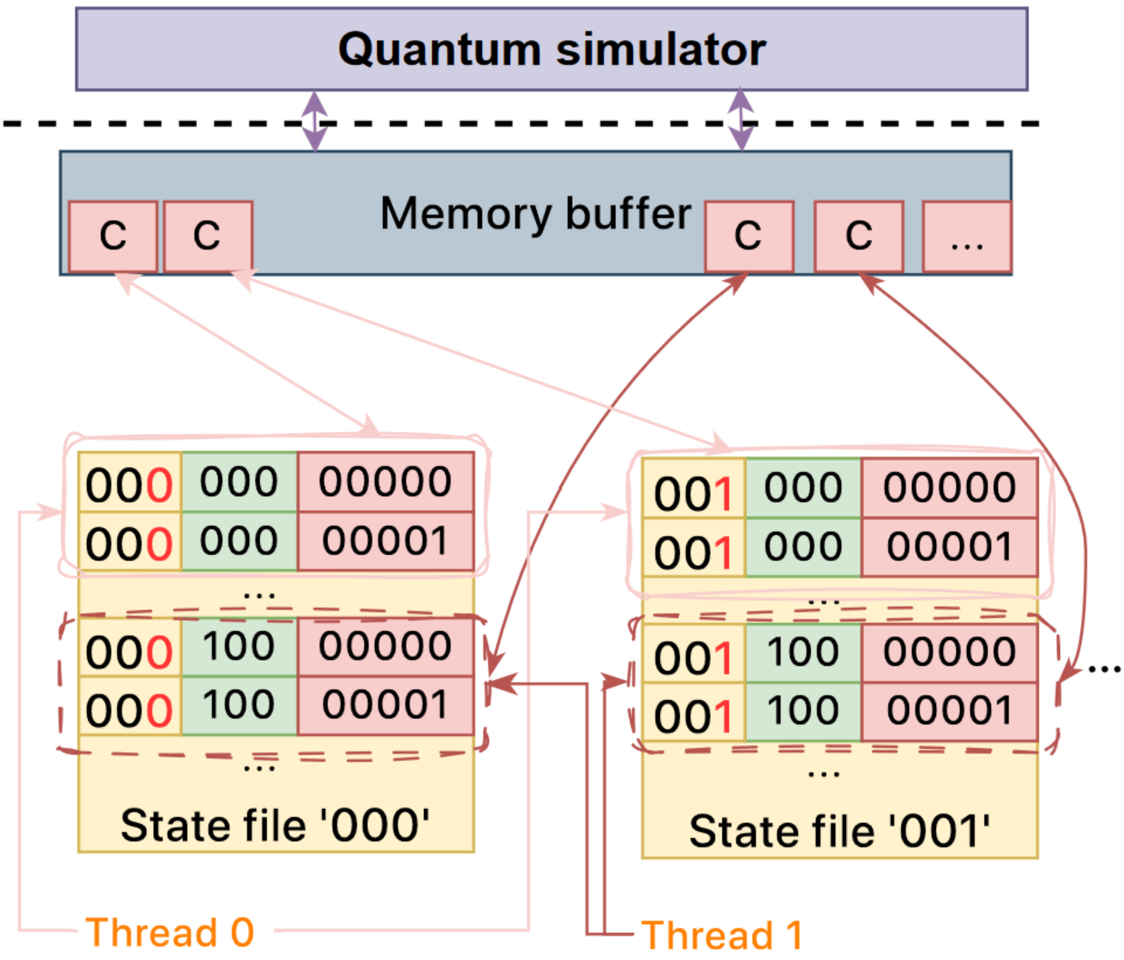

Figure 3.6: The qubit representation and the corresponding accesses to the qubit states by the threads, using the 11-qubit system as an example, where target qubit is located within the file segment (in different state files).

can be updated. When accessing to the chunks across different files, it is important to orchestrate the thread accessing chunks in order to avoid the lock-step simulation among threads (for accessing chunks in the same file). A straightforward solution is to have half of the threads wait while the gate operation is being applied. Since this simulation is mostly I/O bound, the performance is not affected much if only half of the threads are performing I/O operations. The remaining steps are identical to those in the middle segment, where two required chunks of states are fetched, the gate operation is performed, and then the updated chunks of states are written back to the state files.

doi:10.6342/NTU202300661

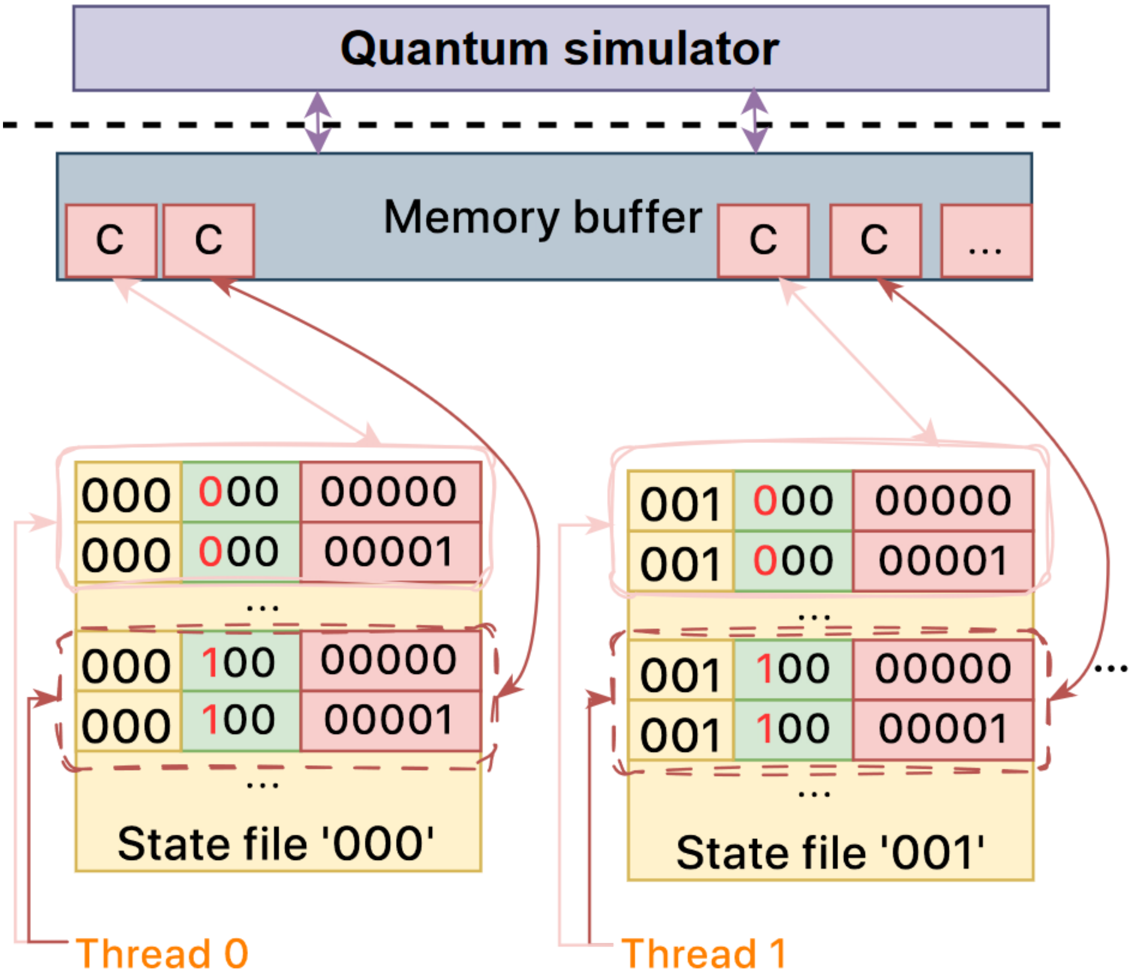

Figure 3.7: The qubit representation and the corresponding accesses to the qubit states by the threads, using the 11-qubit system as an example, where the target qubit falls within the middle segment.

- **Within the middle segment.** In cases where the states needed for operations or measurements are located outside of a single chunk but still in the same file, the data must be aligned in memory for computation after two consecutive reads. These chunks have the same prefix formed by the qubit string of F, and the corresponding pairs of basis vectors are accessed from the state file. Figure 3.7 illustrates the concept, where different threads have the accesses to their respective state files, and the thread index is identical to the file name (the binary bit string). The number of read and write operations for a single thread in the middle segment is the same as in the chunk segment. However, since we apply the gate function for every two

doi:10.6342/NTU202300661

read and write operations, the number of gate functions applied in this case is half compared to the chunk segment.

While this algorithm does not determine the optimal simulation configuration, including factors like chunk size for I/O, file system format, and I/O mode, we have developed a performance optimization strategy that can be applied on any host.

3.4 Extension to Multi-Qubit Gates

When dealing with multi-qubit gates that involve K qubits, each qubit can be located in one of three segments. This means that there are a total of 3^K cases that need to be considered. However, by employing a linear mapping technique described in Appendix 5, we can significantly reduce the number of cases to $\binom{K+2}{2} = \frac{(K+1)(K+2)}{2}$. This reduction in cases simplifies the problem and allows for more efficient analysis.

Once the reduction is applied, the remaining task involves rearranging the pattern of input/output (I/O) operations in chunks. This step aims to optimize the data flow and enhance the overall performance. By carefully organizing the I/O operations, we can utilize the bandwidth, thereby improving the efficiency of the system.

This approach not only reduces the computational complexity by reducing the number of cases but also enhances the scalability of the system. With fewer cases to consider, the system becomes more manageable and adaptable to larger qubit systems. This advancement in handling multi-qubit gates has significant implications for quantum computing, as it allows for more efficient and effective processing of complex quantum algorithms.

doi:10.6342/NTU202300661

3.5 Performance Optimization Strategies

Chapter 3.5 discusses the performance issues that we observed in Algorithm 9 and proposes strategies to resolve the issues.

3.5.1 Contention for the File Descriptor

The parallel quantum circuit simulation is facilitated largely by the parallel IO operations done between the main memory and the storage devices, where the states of qubits are kept in the state files and the main memory is served as the buffer for the quantum circuit simulator to operate on. Instead of using the read/write functions provided by the standard C library, such as `fread` and `fwrite`, the Linux standard system calls `pread` and `pwrite` are used to handle the accesses to the state files in the multithreaded environment. While accessing to these files, their file contents are buffered in the *page cache* maintained by the underlying operating system to shorten the data access time. However, the access time degrades when the data size grows larger than that of the page cache.

The virtual file system (VFS) of Linux systems provides a unified interface for user-space file I/O operations without considering the heterogeneity of the underlying I/O devices. The VFS exposes different interfaces to user-space applications for file I/O. For example, a standard I/O interface leverages the buffering/caching mechanisms in between an user-space application and a storage hardware (left of Figure 3.8), whereas a direct I/O interface (right of Figure 3.8) reduces the caching mechanism in the hope to provide a shorter data accessing latency (especially for an I/O bound system) and to avoid potential data consistency issue. Nevertheless, neither of the both methods can always deliver the

doi:10.6342/NTU202300661

best performance for a quantum circuit simulator that could be used to simulate different qubit sizes (e.g., N could be 5, 10, or 40).

Under such a circumstance, it would be important if the simulator can be aware of the workload, and perform the simulation accordingly to achieve higher performance. Our empirical studies suggest that when simulating a smaller qubit system (e.g., $N = 5$), the standard I/O interface can be used as the main memory is often large enough to accommodate the page cache data. On the other hand, the direct I/O is desired when handling a larger qubit system to provide a shorter accessing latency (since the page cache miss rate is high due to the capacity miss). We have implemented the simulator with the two I/O methods to achieve a high performance simulation under different circumstances. However, in practical applications, there are some potential issues while applying the direct I/O solution. Further results about how to configure the simulator into the standard I/O or the direct I/O mode are presented in Section 4.2. An automatic mode switching mechanism is possible by monitoring the page cache miss rate at runtime; this is part of our future work.

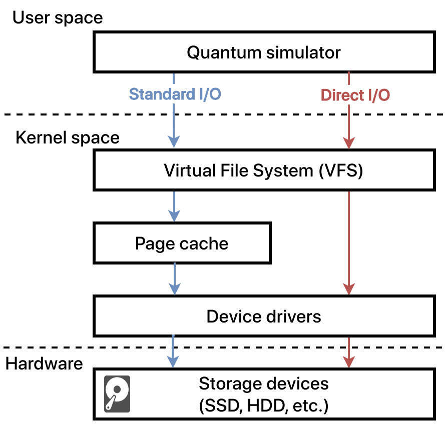

Figure 3.8: Two different types of file reads and writes: standard I/O and direct I/O.

The system calls can access the same file concurrently, thanks to their thread-safe support. Nevertheless, the implicit synchronization mechanism is implemented to ensure the correctness of the concurrent accesses to the same file, which hinders the delivered

doi:10.6342/NTU202300661

performance of the parallel simulation. Therefore, as shown in the previous section, our design assigns a file to a single thread whenever possible to alleviate the synchronization overhead. The experimental results are presented and discussed in Chapter 4.3.

3.5.2 Choices of Filesystem Formats

The qubit states are kept in the state files. While doing the simulation, the qubit states should be pulled out of the state files into the memory buffers, or should be pushed into the storage devices. The data blocks of the to-be-processed file kept in the file system are either pulled or pushed from or to the storage devices. In Linux systems, where the fourth extended file system *Ext4* is considered as a default file system in Linux systems, the addresses to the data blocks are stored in the *inode* data structure and the address pointers (known as *extents*) refer to a range of contiguous physical blocks in the storage devices. Based on our experimental analysis, a small inode size tends to decrease the delivered I/O performance. This is because 1) more inodes are created to indexing the actual file contents, which decreases the storage device size allocated for the file data; 2) the *extent tree* traversal is more frequent to find the physical block addresses. Hence, a larger inode size is desired to provide a larger storage space for file contents and less extent tree traversal time. To this end, the two options `largefile` and `largefile4` can be used to enlarge the inode size into 1 and 4 MiB, respectively.

3.5.3 Selection of Configuration Parameters

To unleash the performance offered by the multicore hardware and the memory subsystem, it is important to use proper parameter settings for the quantum circuit simulations.

doi:10.6342/NTU202300661

The important parameters and their effects on the performance are listed as follows.

1. **Chunk segment (C).** As the simulator operates on the state data chunk by chunk, the chunk size should be determined judiciously to reduce the number of system calls for file I/O. These chunks are buffered in the memory and a proper chunk size setting (to ensure that the chunks can be accommodated by the memory buffer) results in better performance.

2. **File segment (F).** As described in Section 3.5.1, concurrent reads/writes from/to a state file would incur the thread synchronization overhead. To eliminate the overhead, it is possible to let a thread to handle one or more state files. Nevertheless, increasing the number of the files would inevitably add the burden to the underlying system to maintain the multiple files (for a thread). Furthermore, when there are multiple storage devices, the distribution of the state files onto the devices could also be taken into account in order to achieve the best performance.

3. **Thread segment (T) and simultaneous multithreading (SMT) effect.** A larger number of threads could potentially improve simulation speed by allowing more works to be done in parallel. Nevertheless, it would incur higher thread management cost, especially when the thread number is larger than the physical hardware core number. Thus, it is crucial to set the T value properly to deliver the best simulation performance. When the T value is considered, the SMT feature would be taken into account as one of the possible options for a larger T. SMT is a hardware multithreading support found in modern processors, and is useful to technique for improving system throughput by allowing more hardware logical threads running on a physical hardware core simultaneously. Nevertheless, the outcome of enabling

doi:10.6342/NTU202300661

the SMT feature is unpredictable depending highly on the runtime system status.

Observing that improper different parameter settings could lead to poor performance, as an example, our experimental results in Section 3.5 provide the insight (by showing the procedure) to find a proper setting for the quantum circuit simulation on our experimental environment.

3.5.4 Reducing Idle Time with Barrier-free State File Parity Index Finder

In Section 3.3, we discussed stalling half of the threads to avoid file locks and maintain the purity of each thread's duty. However, upon closer inspection of the file segment, we can see that if we allow pairs of threads to work on pairs of state files, we may not need to stall half of the threads during gate application. Figure 3.6 illustrates this concept. Both thread 0 and thread 1 need to access chunks within two files, namely "000" and "001". Thread 0 is responsible for handling the chunks of the middle segment of "000", whereas thread 1 is for the chunks within the middle segment "100". The above assignment scheme is able to assign the proper file index of each thread without the need for additional thread management or thread barriers since the Linux file system is responsible to handle the access order of the chunks within the same state file. This assignment scheme is achieved by Algorithm 10, which is introduced as follows.

Algorithm 10 is used by each simulation thread for calculating the location of the first half-paired of state vectors in a state file (e.g., "000" for thread 1 in Figure 3.6), based on the $tIdx$ (the identifier of each thread), $targ$, and bit shifting operations (lines 1-3). This algorithm is achieved by using the binary numbers to index the files and threads. In this

doi:10.6342/NTU202300661

Algorithm 10 Find out the corresponding pairs of files for each thread when target qubit in file segment and has at least one qubit in middle segment.

1: int tIdx ← omp_get_thread_num(); ▷ Thread ID
2: int targShift ← *targ* − $(N - F)$;
3: int targMask ← 1 << targShift;
4: **if** ($tIdx$ bit-wise AND *targMask*) = 0 **then** ▷ baseFile = tIdx
5:     int baseFile ← *tIdx*;
6:     int corrFile ← *baseFile* bit-wise XOR *targMask*;
7: **else**
8:     int baseFile ← *baseFile* bit-wise XOR *targMask*;
9:     int corrFile ← *tIdx*;
10: **end if**
11: return baseFile and corrFile

case, each simulation thread can calculate the file location for its own use by itself, avoiding the need for thread communications/synchronizations. Similarly, the second-half state vectors can be obtained by using the previously calculated intermediate value along with an XOR operation (line 6 and 8). By utilizing the proposed segmentation and algorithms, the simulator can effectively execute diverse analyses and extend its functionality to include multi-qubit gates such as CNOT, Toffoli, and others in a prompt manner.

After combining Algorithm 10 and Algorithm 9, we proposed Algorithm 11. In this version, we maximize performance without stalling any thread if there is at least one qubit in the middle segment. We partition each pair of state files into an upper half and a lower half, allowing every two threads to cooperate on the same pair of two state files with higher performance.

doi:10.6342/NTU202300661

Algorithm 11 Updated Version combined Algorithm 9 and Algorithm 10

```
void (*gate)(void*, uint64_t) ← get_gate(opCode);
int tId ← omp_get_thread_num();                          ▷ Thread ID
void* buffer;                                            ▷ Independent for Different Thread ID
uint64_t chunkSize ← 2^C * sizeof(double);
if targ < C then                                         ▷ Chunk Segment
    same as Algorithm 9
else if M > targ − C then                                ▷ Middle Segment
    same as Algorithm 9
else                                                     ▷ File Segment
    if M = 0 then                                        ▷ No Middle Segment
        same as Algorithm 9 in File Segment
    else                                                 ▷ At Least 1 Qubit in Middle Segment
        int tIdx ← omp_get_thread_num();                 ▷ Thread ID
        int targShift ← targ − (N − F);
        int targMask ← 1 << targShift;
        baseFile ← (targMask >> (targShift+1) << targShift+1) | targMask bit-wise
AND (targShift-1)
        int corrFile ← baseFile bit-wise XOR targMask;
        if (tIdx bit-wise AND targMask) = 0 then         ▷ baseFile = tIdx
            uint64_t fileOff ← 0;
        else
            uint64_t fileOff ← 2^(N−F−1) * sizeof(double);
        end if
        for j ← 0 to 2^(M−1) − 1 do
            read(baseFile, buffer, chunkSize, fileOff);
            read(corrFile, buffer+chunkSize, chunkSize, fileOff);
            gate(buffer, offsetInMem);
            write(baseFile, buffer, chunkSize, fileOff);
            write(corrFile, buffer+chunkSize, chunkSize, fileOff);
            baseOff ← baseOff + chunkSize;
        end for
    end if
end if
```


doi:10.6342/NTU202300661

Chapter 4 Evaluation

In this chapter, we evaluate the proposed storage-based QCS and the performance optimization strategies mentioned in Chapter 3. To begin with, we elucidate the construction of the experimental setup. Then, we evaluate our optimal result in comparison with the fast and widely-used QuEST simulator. To get the highest performance for other specific systems, We provide several fine-tuning techniques to guide users to adjust the configuration in detail. Furthermore, we give our understanding and recommendations for each method in the respective subsection.

4.1 Experimental Setup

A multithreading quantum circuit simulation environment has been built to provide many working threads to manipulate the quantum states during the simulations. The simulation environment is built upon the AMD Threadripper processor with 64 processor cores (128 logical hardware threads enabled by the SMT technology), as listed in Table 4.1, where the processor has a large last-level CPU cache of 256 MB. For the memory subsystem of the machine, the simulation hardware has eight 32GB DDR4 memory modules, consisting of a total of 256GB main memory. Besides, to support the simulation of a larger qubit size, the eight storage devices form a total of 16TB of the secondary memory,

doi:10.6342/NTU202300661

each of the storage device having up to 7.0 GB/s of memory bandwidth over the PCIe 4.0 bus (x4 lanes; 8GB/s; larger than the theoretical SSD bandwidth). The multithreaded simulation is enabled by the OpenMP library (ver. 4.5) [9], and the simulator is built by GCC ver. 9.4.0, running on top of Ubuntu Linux 20.04 LTS.

Table 4.1: Configuration of hardware and software for experiments.

Name	Description
CPU	AMD Ryzen Threadripper PRO 3995WX 64-Cores @ 2.7GHz(64C/128T)
RAM	HX436C18FB3K2/64, 256GB(8x32GB) DDR4 3600MHz
Storage	Kingston SFYRD/2000G, 16TB (8*2TB on PCIe 4.0 bus)
OS	Ubuntu 20.04 LTS(kernel version 5.15.0-58-generic)
Compiler	GCC version 9.4.0
API	OpenMP version 4.5

4.2 Comparison to Existing Works

We performed a series of comparative analyses between our simulator and the multithread simulator QuEST [20], known for its high performance, as shown in Figure 4.1. To ensure fairness and consistency, each measurement of the Hadamard gate is the average execution time derived from 10 runs, with the removal of the highest and lowest observations, and double precision floating point numbers are adopted. For the sake of the comprehensive evaluation with regard to the utilization of the main memory, we set the qubit range from 21 to 39.

Comparing with the SOTA Memory-based Simulator, QuEST. The figure demonstrates that the storage-based simulation scheme has comparative performance against the memory-based solution. For instance, when $N = 26$, it takes 24ms to do the simulation with our storage-based approach, whereas it costs 21ms to do the same simulation by QuEST (the state-of-the-art memory-based simulator), meaning that the storage-based

doi:10.6342/NTU202300661

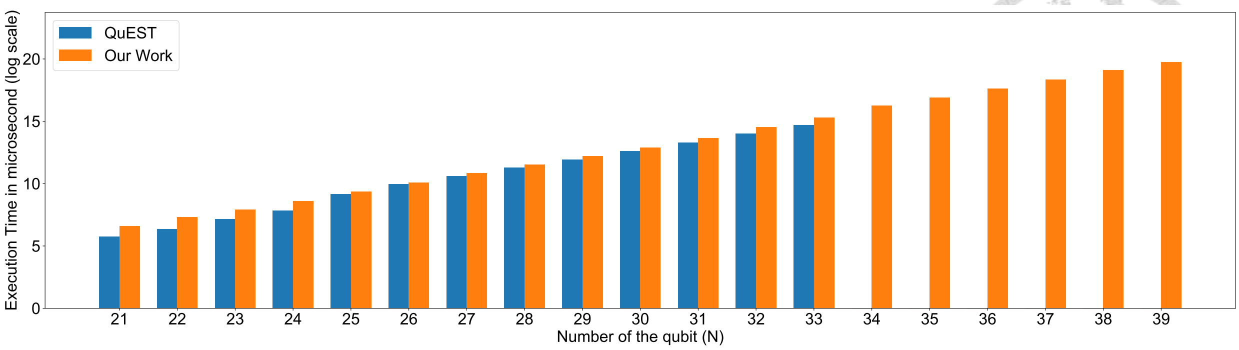

Figure 4.1: A comparison between QuEST and our simulator is performed for the Hadamard gate simulation in second, with a range of qubits from 21 to 39.

method is 0.88 times as fast as the memory-based counterpart. It is important to note that as the number of qubits grows, the memory-based approach cannot handle the simulation effectively and can lead to unknown behaviors. In our setup, when $N = 34$, the state-of-the-art work cannot respond as expected because the state data size exceeds the main memory size and the operating system is busying in swapping out data from memory to the secondary storage to make room for the requested state file data. This phenomenon renders that the memory-based scheme is not suitable for handling a large qubit simulation. On the contrary, our storage-based solution is able to handle a larger qubit. The storage-based simulation is limited by the size of a single storage device and the total bandwidth of the PCIe bus on the motherboard of the host system.

Comparing with the SOTA Storage-based Simulator, SnuQS. SnuQS [26] is the other one of the pioneer works for the storage-based quantum circuit simulation, and it also supports the simulation of up to 41-qubit quantum programs with larger RAID-0 enabled SSDs (e.g., taking 1.44 hours to handle the 41-qubit H gate simulation). To give a rough comparison with SnuQS, the simulation time of the H gate simulation for a 39-qubit system is approximated by scaling down the time linearly, and it turns out that it takes about 0.36 hours to do the 39-qubit simulation[1]. On the other hand, our proposed simulator costs

[1]Based on the experimental results provided by SnuQS the simulation time grows linearly with the increase of the simulated qubit size.

doi:10.6342/NTU202300661

about 0.11 hours to do the H gate simulation for a 39-qubit system, and we believe our approach is comparative to the state-of-the-art work. As this work adopts an orthogonal approach for storage-based simulation, we believe our methodology can work together with the SnuQS methods to achieve an even higher simulation performance. More about the comparison of our method and SnuQS's method is provided in Section 2.2.

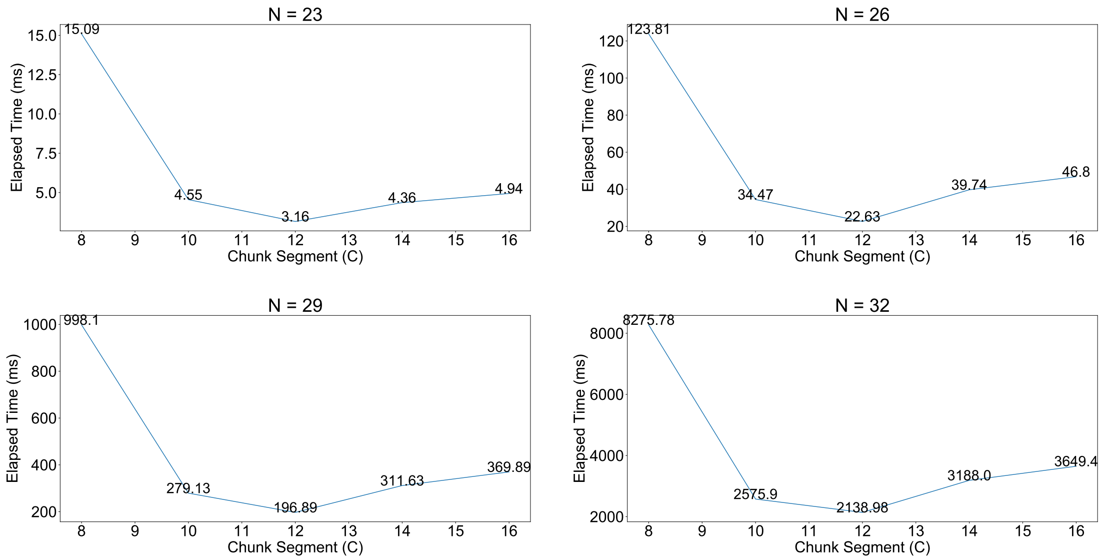

Figure 4.2: Ultizing 64 cores to explore a proper chuck segment value.

4.3 Evaluation of Optimization Strategies

To achieve the best simulation performance on a given computer, the parameters listed in Section 3.5.3 should be selected judiciously. The following paragraphs use the Hadamard gate as an example workload to evaluate the delivered simulation performance. The reported performance data are the average simulation time for ten runs, excluding the outliers.

Chunk segment. Figure 4.2 illustrates the simulation times under different chunk segment sizes (C=8, 10, 12, 14, 16) and qubit sizes (N=23, 26, 29, 23) combinations, where there are 64 threads (T=6), each accessing a respective file (F=6). Overall, when

doi:10.6342/NTU202300661

the chunk segment is set to twelve, the simulations deliver the best performance. Simply choosing a value could result in poor performance. For example, it is intuitive to use the file system block size, which is 4,096 bytes in our system, as the chunk segment size (C=8)[2]. But, as illustrated in the figure, when the chunk segment is set to eight, its performance is up to 5.4x slower than the performance for the setting of twelve (chunk size = 64KB), in the experiment of N=26. It is important to note that the delivered performance is affected by multiple factors, such as the file system block size, the virtual file system caching mechanism, the virtual memory management (e.g., aggressiveness of swap out memory pages), the caching mechanism of the storage devices, and the block size of the storage devices. Hence, it would be a good idea to test for a proper setting when the simulations are to be performed on a new computer.

File segment. To experiment the synchronization overhead of the concurrent accesses to the state files, as described in Section 3.5.1, this experiment uses 64 threads (T=6) to perform the concurrent reads from 8 (F=3) and 64 (F=6) files, respectively, with and without the *largefile* setting.

Table 4.2 shows that the synchronization overhead is eliminated by allowing each thread to access a file (the settings with F=6), and the performance is improved by a factor of 2.8 (N=35) and 2.3 (N=35, largefile). Note that when multiple threads access the same file, each thread has to wait for its turn to access the file descriptor to avoid inconsistent state [6], and this synchronization overhead can be removed by using the one-file-for-one-thread scheme.

Thread segment. To discover a better T value, this experiment tests the T values

[2]When C is eight, it means the chunk size of 4,096 bytes (2^{8+4} = 4,096). Please refer to Section 3.2 for details.

doi:10.6342/NTU202300661

Table 4.2: Evaluating the performance impact of different file segments (unit: s).

	31	32	33	34	35
C=12, F=3, T=6	1.52	5.51	17.55	65.23	132.24
C=12, F=6, T=6	0.99	2.90	6.14	29.36	45.72
[Largefile] C=12, F=3, T=6	1.21	2.66	5.29	46.10	95.75
[Largefile] C=12, F=6, T=6	**0.96**	**2.11**	**4.26**	**21.92**	**40.71**

Table 4.3: Evaluating a proper thread segment value (unit: s).

Qubit	T = 6	T = 7	T = 8	T = 9	T = 10
31	**1.0**	1.1	1.3	1.3	1.2
32	**2.1**	2.3	2.9	2.8	3.0
33	**4.3**	4.8	5.6	5.7	6.4
34	**21.9**	27.0	28.7	29.1	27.4
35	**40.7**	76.4	88.1	85.5	83.8
36	**97.4**	148.2	189.5	174.6	190.4
37	**186.5**	293.5	366.1	376.5	363.2
38	**541.1**	761.8	803.4	766.2	729.0
39	**1127.3**	1190.9	1559.3	1462.2	1460.7

ranging from six (64 threads) to ten (1,024 threads). The performance data suggest that the simulation system delivers the best performance when the T value is agreed with the physical core number (64). Turning on the SMT feature (T=7; via turning off the SMT switch in the system BIOS) does not help the performance. Further increasing the thread number ($T > 6$) would hurt the performance since the simulation system is bounded by I/O bandwidth when T is six according to our preliminary analysis. Using a larger number of threads would incur a higher context-switching overhead, and the increased I/O requests made by the threads contend with the I/O transfer channels.

Direct IO. Table 4.4 lists the performance delivered by the storage-based simulations with and without the use of direct I/O for state data reads/writes, under the setting of 64 threads ($F = T = 6$) and 64KB ($C = 12$) of the chunk size. As mentioned in Section 3.5.1, simply using a specific I/O method cannot always deliver the best perfor-

doi:10.6342/NTU202300661

Table 4.4: The elapsed time of the standard IO and direct IO (unit: s).

Qubit	Memory Requirement	Standard IO	Direct IO	Relative Performance
31	32 GiB	**1.0**	2.0	0.5
32	64 GiB	**2.1**	3.5	0.6
33	128 GiB	**4.3**	6.0	0.7
34	256 GiB	21.9	**11.4**	**1.9**
35	512 GiB	40.7	**22.4**	**1.8**
36	1 TiB	87.3	**44.7**	**2.0**
37	2 TiB	182.1	**91.4**	**2.0**
38	4 TiB	377.3	**197.1**	**1.9**
39	8 TiB	764.2	**385.0**	**2.0**

mance. When the state data can be accommodated by the main memory ($N \leq 33$), the standard I/O method is 2x faster than the direct I/O alternative since the state data (maintained by the page cache) are hit on the memory. On the other hand, the direct I/O method performs 2x better than the standard I/O method for a larger qubit size ($N > 33$) because it provides direct access to the underlying storage devices without incurring the virtual memory management overhead (caused by the page cache for not being able to handle a large data). It is important to note that the empirical data shown in Table 4.3 suggest the quantum circuit simulation is an I/O bound workload, and under such a circumstance, the direct I/O can reduce the I/O latency without the intervention of the caching mechanism. This mechanism should be enabled for the simulation of a large qubit system.

doi:10.6342/NTU202300661

doi:10.6342/NTU202300661

Chapter 5 Conclusion

In this work, a storage-based quantum circuit simulator is proposed to provide a cost-effective alternative to the memory-based scheme. The proposed approach is up to 61 times more efficient than the memory-based counterpart, in terms of the cost-delay product. A qubit representation is developed, together with a threading model, to enable efficient simulation of quantum circuits of different qubit sizes with parameterized configurations. Moreover, several performance issues are addressed by proposed strategies. The experimental results show that the proposed storage-based simulation provides a platform-aware approach to increase the qubit size supported by the simulator. The size of quantum circuit is bounded by the volume sizes of the adopted SSDs and the resulted performance is limited by the maximum bandwidth of the storage system. Thanks to the combination of high capacity, good data bandwidth, and low cost of SSDs, as well as the expandability of I/O devices, our approach provides a cost-effective and flexible solution for large-scale quantum circuit simulation.

The use of additional SSD expansion cards can potentially improve data bandwidth and effectively reduce the IO time, leading to enhanced efficiency for the proposed method. Thus, we would be interested in finding ways to attach as many SSD expansion cards as possible with high bandwidth I/O interfaces. It is possible to adopt NVMe over Fabric (NVMe-oF) [4] to connect external SSDs via high-speed network interfaces.

doi:10.6342/NTU202300661

We have mentioned the lifespan issue of Today's NAND-based SSDs, which limits the application of the proposed method. While it is possible to adopt a more durable nonvolatile memory technology, such as phase change memory (PCM), when it becomes affordable, we can also extend our method to support remote memories via remote direct memory access (RDMA) over high-speed network interfaces. By combining several aforementioned techniques, we may better address the limitations observed in this thesis and result in overall performance improvements.

doi:10.6342/NTU202300661

References

[1] Cirq is a Python library for writing, manipulating, and optimizing quantum circuits and running them against quantum computers and simulators. https://github.com/quantumlib/Cirq.

[2] DRAM HX436C18FB3K2/64. https://amazon.com/dp/B089QV3HX2.

[3] Ibm quantum composer. https://quantum-computing.ibm.com/composer/.

[4] NVM Express. 2021. NVM Express over Fabric 1.1a. Retrieved from. https://nvm-express.org/specifications/.

[5] NVMe SSD SFYRD/2000G. https://amazon.com/dp/B09K36S11S.

[6] pread(2) - Linux manual page. https://linux.die.net/man/2/pread.

[7] A quantum computer is the ultimate black box. our quantum user interface is here to help. https://qui.research.unimelb.edu.au/.

[8] Quirk is a toy quantum circuit simulator, intended to help people in learning about quantum computing. https://github.com/Strilanc/Quirk.

[9] The OpenMP API specification for parallel programming specification. https://www.openmp.org/.

doi:10.6342/NTU202300661

[10] T. Alexander, N. Kanazawa, D. J. Egger, L. Capelluto, C. J. Wood, A. Javadi-Abhari, and D. C. McKay. Qiskit pulse: programming quantum computers through the cloud with pulses. Quantum Science and Technology, 5(4):044006, aug 2020.

[11] M.-D. Choi. Completely positive linear maps on complex matrices. Linear Algebra and its Applications, 10(3):285–290, 1975.

[12] J. Chow, O. Dial, and J. Gambetta. IBM Quantum breaks the 100-qubit processor barrier. https://research.ibm.com/blog/127-qubit-quantum-processor-eagle.

[13] A. W. Cross, L. S. Bishop, J. A. Smolin, and J. M. Gambetta. Open quantum assembly language, 2017.

[14] T. cuQuantum development team. cuquantum, Apr. 2023. If you use this software, please cite it as below.

[15] H. De Raedt, F. Jin, D. Willsch, M. Willsch, N. Yoshioka, N. Ito, S. Yuan, and K. Michielsen. Massively parallel quantum computer simulator, eleven years later. Computer Physics Communications, 237:47–61, 2019.

[16] K. De Raedt, K. Michielsen, H. De Raedt, B. Trieu, G. Arnold, M. Richter, T. Lippert, H. Watanabe, and N. Ito. Massively parallel quantum computer simulator. Computer Physics Communications, 176(2):121–136, 2007.

[17] V. Gheorghiu. Quantum++: A modern c++ quantum computing library. PLOS ONE, 13(12):e0208073, dec 2018.

[18] Y. Huang and M. Martonosi. Statistical assertions for validating patterns and finding

doi:10.6342/NTU202300661

bugs in quantum programs. In Proceedings of the 46th International Symposium on Computer Architecture. ACM, jun 2019.

[19] T. Häner and D. S. Steiger. 0.5 petabyte simulation of a 45-qubit quantum circuit. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. ACM, nov 2017.

[20] T. Jones, A. Brown, I. Bush, and S. Benjamin. Quest and high performance simulation of quantum computers. Scientific Reports, 9, 07 2019.

[21] R. LaRose. Distributed memory techniques for classical simulation of quantum circuits, 2018.

[22] Y. A. Liu, X. L. Liu, F. N. Li, H. Fu, Y. Yang, J. Song, P. Zhao, Z. Wang, D. Peng, H. Chen, C. Guo, H. Huang, W. Wu, and D. Chen. Closing the "quantum supremacy" gap. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. ACM, nov 2021.

[23] S. A. Metwalli and R. V. Meter. A tool for debugging quantum circuits, 2022.

[24] J. Niwa, K. Matsumoto, and H. Imai. General-purpose parallel simulator for quantum computing. Phys. Rev. A, 66:062317, Dec 2002.

[25] R. Okazai, T. Tabata, S. Sakashita, K. Kitamura, N. Takagi, H. Sakata, T. Ishibashi, T. Nakamura, and Y. Ajima. Supercomputer Fugaku CPU A64FX Realizing High Performance, High Density Packaging, and Low Power Consumption. https://www.fujitsu.com/global/documents/about/resources/publications/technicalreview/2020-03/article03.pdf.

doi:10.6342/NTU202300661

[26] D. Park, H. Kim, J. Kim, T. Kim, and J. Lee. Snuqs: Scaling quantum circuit simulation using storage devices. In Proceedings of the 36th ACM International Conference on Supercomputing, ICS '22, New York, NY, USA, 2022. Association for Computing Machinery.

[27] K. D. Raedt, K. Michielsen, H. D. Raedt, B. Trieu, G. Arnold, M. Richter, T. Lippert, H. Watanabe, and N. Ito. Massively parallel quantum computer simulator. Computer Physics Communications, 176(2):121–136, jan 2007.

[28] K. D. Raedt, K. Michielsen, H. D. Raedt, B. Trieu, G. Arnold, M. Richter, T. Lippert, H. Watanabe, and N. Ito. Massively parallel quantum computer simulator. Computer Physics Communications, 176(2):121–136, jan 2007.

[29] E. Roloff, M. Diener, E. D. Carreño, F. B. Moreira, L. P. Gaspary, and P. O. Navaux. Exploiting price and performance tradeoffs in heterogeneous clouds. In Companion Proceedings of The10th International Conference on Utility and Cloud Computing, UCC '17 Companion, page 71 – 76, New York, NY, USA, 2017. Association for Computing Machinery.

[30] M. Smelyanskiy, N. P. D. Sawaya, and A. Aspuru-Guzik. qhipster: The quantum high performance software testing environment, 2016.

[31] D. S. Steiger, T. Häner, and M. Troyer. ProjectQ: an open source software framework for quantum computing. Quantum, 2:49, jan 2018.

[32] A. Suau, G. Staffelbach, and A. Todri-Sanial. qprof: a gprof-inspired quantum profiler, 2021.

[33] D. B. Trieu. Large-Scale Simulations of Error-Prone Quantum Computation

doi:10.6342/NTU202300661

Devices. Dr. (univ.), Universität Wuppertal, Jülich, 2009. Record converted from VDB: 12.11.2012; Universität Wuppertal, Diss., 2009.

[34] C.-H. Wu, C.-Y. Hsieh, J.-Y. Li, and J. C.-M. Li. qatg: Automatic test generation for quantum circuits. In 2020 IEEE International Test Conference (ITC), pages 1–10, 2020.

doi:10.6342/NTU202300661

doi:10.6342/NTU202300661

Appendix A — One-qubit Gate Derivation

By incorporating chunks into the memory-based algorithm, we have simplified the management of the state vector when bring states into memory. For line 15-23 in Algorithm 12, we can extract the operation and encapsulated into Algorithm 13. From here, the states have to be brought to memory before calculated, line 15-23 and 30-34 in Algorithm 12 are then be formulated into line 15-17 and 24-32 Algorithm 14, respectively.

Furthermore, the use of chunks enhances memory locality. Instead of repeatedly accessing the entire state vector, we only need to load and work with a specific chunk at a given time. This optimized access pattern improves cache utilization and reduces memory access latency, leading to overall improved performance.

At last, we further take the storage limit of one device into consideration and line 19-36 in Algorithm 14 are then be formulated into line 16-47 in Algorithm 15.

Finally, in order to account for the storage limit of a single device, we have incorporated this consideration into our approach. As a result, the corresponding lines 19-36 in Algorithm 15 have been reformulated and now appear as lines 16-47 in Algorithm 9.

doi:10.6342/NTU202300661

Algorithm 12 one_qubit_gate(): Chunk-based simulation.

```
Input: task, target_qubit
Result: [s0, s1] pair of state
target_offset ← 1 « target_qubit
half_task_block ← target_offset
task_mod_mask ← half_task_block-1
num_tasks_per_thread = 2^(N−1) / num_threads
if num_tasks_per_thread < half_task_block then
    task_offset ← num_tasks_per_thread;
else
    task_offset ← half_task_block;
end if
task_per_chunk ← 2^(C−1)                    ▷ states_per_chunk = 2*task_per_chunk
if task_per_chunk ≥ task_offset then
    for task ← task_start[tid]; task < task_end[tid]; task+=task_per_chunk do
        s0 ← (task≫target_qubit)≪(target_qubit+1) | (task&task_mod_mask)
        s1 ← s0 + target_offset
        for j = 0; j < task_per_chunk; j+=task_offset do
            for i=0; i < task_offset; i++ do
                simulate_gate(gateOps, [s0, s1])
                s0++
                s1++
            end for
            s0 += task_offset
            s1 += task_offset
        end for
    end for
else
    for task ← task_start[tid]; task < task_end[tid]; task+=task_offset do
        s0 ← (task≫target_qubit)≪(target_qubit+1)
        s1 ← s0 + target_offset
        for j = 0; j < task_offset; j+=task_per_chunk do
            for i=0; i < task_per_chunk; i++ do
                simulate_gate(gateOps, [s0, s1])
                s0++
                s1++
            end for
            s0 += task_per_chunk
            s1 += task_per_chunk
        end for
    end for
end if
```


doi:10.6342/NTU202300661

Algorithm 13 Chunk_ops([s0, s1], task_per_chunk, task_offset): Calculation inside contagious memory of size 2*task_per_chunk

Global: gateOps
Input: [s0, s1], task_per_chunk, task_offset num_tasks ← task_per_chunk incre ← task_offset

```
for j = 0; j < num_tasks; j+=incre do
    for i=0; i < incre; i++ do
        simulate_gate(gateOps, [s0, s1])
        s0++
        s1++
    end for
    s0 += incre
    s1 += incre
end for
```


doi:10.6342/NTU202300661

Algorithm 14 one_qubit_gate(): Simulation with read/write.

```
Input: task, target_qubit
Result: [s0, s1] pair of state
target_offset ← 1 « target_qubit
half_task_block ← target_offset
task_mod_mask ← half_task_block-1
num_tasks_per_thread = 2^(N−1) / num_threads
if num_tasks_per_thread < half_task_block then
    task_offset ← num_tasks_per_thread;
else
    task_offset ← half_task_block;
end if
task_per_chunk ← 2^(C−1)                          ▷ states_per_chunk = 2*task_per_chunk
if task_per_chunk ≥ task_offset then
    for task ← task_start[tid]; task < task_end[tid]; task+=task_per_chunk do
        s0 ← (task≫target_qubit)≪(target_qubit+1) | (task&task_mod_mask)
        s1 ← s0 + target_offset
        read 2*task_per_chunk states from s0.          ▷ read states_per_chunk
        Chunk_ops(s0, s1, task_per_chunk, task_offset)
        write 2*task_per_chunk states to s0.           ▷ write states_per_chunk
    end for
else                                               ▷ target_offset > task_per_chunk
    for task ← task_start[tid]; task < task_end[tid]; task+=task_offset do
        s0 ← (task≫target_qubit)≪(target_qubit+1)
        s1 ← s0 + target_offset
        for j = 0; j < task_offset; j+=task_per_chunk do
            read task_per_chunk states from s0.        ▷ read half of states_per_chunk
            read task_per_chunk states from s1.        ▷ read half of states_per_chunk
            for i=0; i < task_per_chunk; i++ do
                simulate_gate(gateOps, [s0, s1])
                s0++
                s1++
            end for
            write task_per_chunk states to s0.         ▷ write half of states_per_chunk
            write task_per_chunk states to s1.         ▷ write half of states_per_chunk
            s0 += task_per_chunk
            s1 += task_per_chunk
        end for
    end for
end if
```


doi:10.6342/NTU202300661

Algorithm 15 one_qubit_gate(): Simulation with modified read/write.

```
Input: task, target_qubit
Result: [s0, s1] pair of state
target_offset ← 1 « target_qubit
half_task_block ← target_offset
task_mod_mask ← half_task_block-1
num_tasks_per_thread = 2^(N−1) / num_threads
if num_tasks_per_thread < half_task_block then
    task_offset ← num_tasks_per_thread;
else
    task_offset ← half_task_block;
end if
task_per_chunk ← 2^(C−1)                          ▷ states_per_chunk = 2*task_per_chunk
if task_per_chunk ≥ task_offset then
    for task ← task_start[tid]; task < task_end[tid]; task+=task_per_chunk do
        s0 ← (task≫target_qubit)≪(target_qubit+1) | (task&task_mod_mask)
        s1 ← s0 + target_offset
        read 2*task_per_chunk states from s0.                ▷ read states_per_chunk
        Chunk_ops(s0, s1, task_per_chunk, task_offset)
        write 2*task_per_chunk states to s0.                 ▷ write states_per_chunk
    end for
else                                              ▷ target_offset > task_per_chunk
    for task ← task_start[tid]; task < task_end[tid]; task+=task_offset do
        s0 ← (task≫target_qubit)≪(target_qubit+1)
        s1 ← s0 + target_offset
        for j = 0; j < task_offset; j+=2*task_per_chunk do
            read 2*task_per_chunk states from s0.            ▷ read states_per_chunk
            read 2*task_per_chunk states from s1.            ▷ read states_per_chunk
            for i=0; i < 2*task_per_chunk; i++ do
                simulate_gate(gateOps, [s0, s1])
                s0++
                s1++
            end for
            write 2*task_per_chunk states to s0.             ▷ write states_per_chunk
            write 2*task_per_chunk states to s1.             ▷ write states_per_chunk
            s0 += 2*task_per_chunk
            s1 += 2*task_per_chunk
        end for
    end for
end if
```


doi:10.6342/NTU202300661

doi:10.6342/NTU202300661

Appendix B — Gate Transformation

In multi-qubit gate, operations are applied to different qubit $\{q_k\}$in order. In this scenario, where q_i and q_{i+1} are not ordered, it becomes necessary to consider all cases whether $q_i < q_{i+1}$ or $q_i > q_{i+1}$. However, by rearranging the matrix in a specific manner, it is possible to reduce the complexity of these cases.

B.1 Two-qubit Gate Cases

In two-qubit gate case, it is simple to be recognize that either $q_0 > q_1$ or $q_0 < q_1$. If the former case is preferred, reordering the matrix elements based on the values of q0 and q1. By rearranging the matrix, we can ensure that the elements are ordered in a specific way that simplifies the simulation process.

For example, let's assume we have a matrix A representing the unitary two-qubit quantum gate, and q_0 and q_1 are the indices of the qubits involved. We can reorder the indices of A such that q_0 is always greater than q_1. This reordering can be done by permuting the rows and columns of A based on the values of q_0 and q_1. Applied a permutation matrix

doi:10.6342/NTU202300661

P to the vector can be seen as follows.

$$A = \begin{bmatrix} a_{00,00} & a_{00,01} & a_{00,10} & a_{00,11} \\ a_{01,00} & a_{01,01} & a_{01,10} & a_{01,11} \\ a_{10,00} & a_{10,01} & a_{10,10} & a_{10,11} \\ a_{11,00} & a_{11,01} & a_{11,10} & a_{11,11} \end{bmatrix}, v = \begin{bmatrix} v_{q_0=0,q_1=0} \\ v_{q_0=0,q_1=1} \\ v_{q_0=1,q_1=0} \\ v_{q_0=1,q_1=1} \end{bmatrix}$$

$$Av = v_{updated}$$

$$Pv_{updated} = PAv = PAP^{-1}Pv = (PAP^{-1})(Pv), P = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = P^{-1}$$

$$PAP^{-1} = \begin{bmatrix} a_{00,00} & a_{00,10} & a_{00,01} & a_{00,11} \\ a_{10,00} & a_{10,10} & a_{10,01} & a_{10,11} \\ a_{01,00} & a_{01,10} & a_{01,01} & a_{01,11} \\ a_{11,00} & a_{11,10} & a_{11,01} & a_{11,11} \end{bmatrix}$$

Note that both v and $v_{updated}$ are permuted with P, which means that the order before (Pv) and after $(Pv_{updated})$ the computation is the same. (PAP^{-1}) is the rearranged matrix we would like to find, which is the same operation of A also applying on both q_0 and q_1 but in different order, q_1, q_0.

Once the matrix is rearranged, the simulation process becomes more efficient. Instead of considering all possible permutations of q_0 and q_1, which leads to 3 types of segments for each qubit involved and 9 cases in total, we can now iterate through the cases in a specific order and only 6 cases left, taking advantage of its reordered structure.

doi:10.6342/NTU202300661

B.2 Generalized Cases

Besides the two-qubit gate, there are more generalized cases that can be considered for three-qubit gates and gates involving even more qubits. By extending the concept to higher-order gates, we can explore the simulation of more complex quantum circuits.

For three-qubit gates, the number of possible combinations increases significantly, up to 6 permutations are possible. In order to cope with all of these possible permutations, 27 cases composed of 3 types of segments for each qubit involved are considered. This grows exponentially when number of qubits involved in the gate increase, which make the larger multi-qubit gate infeasible in this situation.

After rearrange the matrix of three-qubit gates, the number of cases dramatically decrease to only 10 cases. This can be compute by

$$\#\text{cases for K-qubit gate} = \binom{K+2}{2} = \frac{(K+1)(K+2)}{2}$$

The explanation is simple, we are separating N qubit with 2 bars for 3 types of segments.

Given a K-qubit gate matrix A involved $\{q_k\}$ these qubits. Let $\delta(k)$ as the permutation of $\{q_k\}$ with $\{q_{\delta(k)}\}$ in order. Then the elements of the same gate operation A' correspond to $\{q_{\delta(k)}\}$ is

$$A'_{\{q_{\delta(k)}\}_i,\{q_{\delta(k)}\}_j} = A_{\{q_k\}_i,\{q_k\}_j} = A_{ij}$$

For each element A_{ij}, transform to binary representation, then reorder each bit based on the permutation of $\delta(k)$. The gate operation A' is the same operation as matrix A with qubits in different order.

doi:10.6342/NTU202300661

By reducing the complexity of the cases and optimizing the simulation process through matrix rearrangement, we can potentially improve the efficiency of the quantum circuit simulator. Future work could explore algorithms and techniques for automatically identifying the optimal matrix rearrangement for a given circuit, taking into account the specific qubit interactions and their order.

doi:10.6342/NTU202300661