Received 29 November 2023; revised 7 February 2024; accepted 7 February 2024; date of publication 9 February 2024; date of current version 12 March 2024.

Digital Object Identifier 10.1109/TQE.2024.3364546

Parallelizing Quantum Simulation With Decision Diagrams

SHAOWEN LI[1], YUSUKE KIMURA[2], HIROYUKI SATO[1] (Member, IEEE), AND MASAHIRO FUJITA[1] (Life Member, IEEE)

[1]University of Tokyo, Tokyo 113-8654, Japan
[2]Fujitsu Limited, Tokyo 105-7123, Japan

Corresponding author: Shaowen Li (e-mail: li-shaowen879@satolab.itc.u-tokyo.ac.jp).

An earlier version of this article was presented at the IEEE International Conference on Quantum Software, 2023 [DOI: 10.1109/QSW59989.2023.00026].

This work was supported in part by Fujitsu Limited.

ABSTRACT Since people became aware of the power of quantum phenomena in the domain of traditional computation, a great number of complex problems that were once considered intractable in the classical world have been tackled. The downsides of quantum supremacy are its high cost and unpredictability. Numerous researchers are relying on quantum simulators running on classical computers. The critical obstacle facing classical computers in the task of quantum simulation is its limited memory space. Quantum simulation intrinsically models the state evolution of quantum subsystems. Qubits are mathematically constructed in the Hilbert space whose size grows exponentially. Consequently, the scalability of the straightforward statevector approach is limited. It has been proven effective in adopting decision diagrams (DDs) to mitigate the memory cost issue in various fields. In recent years, researchers have adapted DDs into different forms for representing quantum states and performing quantum calculations efficiently. This leads to the study of DD-based quantum simulation. However, their advantage of memory efficiency does not let it replace the mainstream statevector and tensor network-based approaches. We argue the reason is the lack of effective parallelization strategies in performing calculations on DDs. In this article, we explore several strategies for parallelizing DD operations with a focus on leveraging them for quantum simulations. The target is to find the optimal parallelization strategies and improve the performance of DD-based quantum simulation. Based on the experiment results, our proposed strategy achieves a 2–3 times faster simulation of Grover's algorithm and random circuits than the state-of-the-art single-thread DD-based simulator DDSIM.

INDEX TERMS Decision diagrams (DDs), parallelization, performance, quantum computation, simulation.

I. INTRODUCTION

In 2019, Google demonstrated quantum supremacy using its Sycamore processor, which can support 53 qubits [2]. With the hope of realizing quantum power in practice, researchers from various fields have begun to explore the possibility of using quantum computation to address problems that were once considered intractable in the classical world [3], [4], [5]. Other tech giants like IBM have also constructed their physical quantum computers.

Building infrastructure and pushing research progress requires a broad collaboration beyond just dominant companies. Nevertheless, constructing physical quantum computers still requires an inaccessible amount of resources to the majority of researchers. Consequently, quantum simulators are still an indispensable part of the quantum toolchain. Effective quantum simulations not only allow more researchers to obtain access to the quantum resources but also permit the efficient testing and verification of quantum algorithms before launching on the physical device. We have also witnessed several different teams simulating Google's quantum supremacy claim using just classical means. The statevector-based approach, i.e., using matrices and arrays to represent quantum operators and quantum states, works fine for abstract calculations. However, it requires an exponentially large memory size to be implemented on classical machines. This drawback considerably limits the number of qubits that can be simulated. Companies and research institutions with abundant resources can leverage supercomputers of

© 2024 The Authors. This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License.
For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

thousands of nodes and petabytes of memory to increase the simulation scale. However, this becomes impractical to general researchers.

Quantum algorithms are expressed as a unitary evolution of a quantum state. Quantum states and quantum evolution are, respectively, modeled using vectors and matrices in the Hilbert space. Both vectors and matrices grow exponentially with respect to the number of subsystems, i.e., qubits. The evolution can be mathematically modeled using a sequence of matrix–vector multiplications. A straightforward method to simulate quantum algorithms is to represent quantum states and operators as one- and 2-D arrays, and perform a sequence of matrix–vector multiplication following the structure of the simulated quantum circuits. The critical downside of this approach is the exponential growth of the array size which limits the simulation scale. Binary decision diagrams (BDDs) are a canonical tool for solving the state explosion issue in model-checking and formal verification. They have also been engineered to simulate quantum circuits. In BDDs, nodes represent matrices or submatrices, and edges represent quantum state transitions. DDs are memory efficient in the sense that identical submatrices across all matrices can be represented using a unique node. Variants of BDDs have been proposed with different targets in mind, for example, Quantum information decision diagram (QuIDD) [6] and quantum decision diagram (QDD) [7]. The original BDDs are modified by attaching weights to edges and creating more terminals. The discussion in this article is based on quantum multiple-valued DDs (QMDDs) [8]. The added features of QMDDs are the explicit support of complex-valued entries and multiple-valued basis states.

Using DDs in quantum simulation mitigates the memory pressure, however, they tend to be slower than the straightforward statevector-based approach. We argue that the time inefficiency is due to the lack of parallelization in DD operations. Parallelization has proven its success in large-scale scientific computations. We also consider it a potent tool in quantum simulation. Both statevector and tensor network-based quantum simulations have been parallelized in previous works to boost the performance. However, it is yet to be answered how DDs can be efficiently parallelized. We consider this absence due to the following factors. First, DDs are proposed with the goal of avoiding data duplication. As a result, the typical tradeoff between time and space requires synchronization mechanisms for correct updates. Second, DDs rely on a few auxiliary data structures for better performance, such as the operation cache and the unique table [9]. Parallelization strategies for these data structures are not unique and can exhibit different performance characteristics in different applications. Third, parallelism can be achieved via different primitives. They also affect the performance of the simulation task.

This article answers these questions by investigating different strategies for parallelizing QMDD-based quantum simulation on a shared-memory machine. Three parallelization primitives are examined in this article: *tasks*, *threads*, and *fibers*. Fiber-based concurrent programming is common in writing game engines. However, it has not been explored for the task of quantum simulation. Our experimental results answer when and how fibers can benefit DD-based quantum simulation. Meanwhile, the effects of the operation cache and the unique table are examined, revealing results not suggested before. One example of our findings is the different performance of global and local caches under different levels of parallelization, as well as different numbers of qubits. We summarize our findings here: parallelization can affect DD-based quantum simulation. Whether it benefits the performance depends on the characteristics of the simulated circuit (e.g., randomness) and the simulation scale (e.g., the number of qubits). When the number of qubits is large, using fibers combined with thread local operation caches is the optimal solution. On our machine, this threshold is found to be above 30 qubits. For an intermediate number of qubits, e.g., 20 to 30, using fibers with a global operation cache beats other alternatives. For a small number of qubits, adding parallelization incurs extra costs and hurts the performance. We have found that for certain quantum algorithms with highly random circuits, the operation cache hit ratio becomes nearly 0 and should be removed. We observe 2–3 times faster simulation results of the Grover's algorithm and random circuits in the experiment using our proposed strategies compared with the state-of-the-art single-thread DD simulator DDSIM [10], [1].

The rest of this article is organized as follows. Section II reviews simulation approaches and the basic of fibers. Section III covers the design of our parallel simulation strategy. Section IV provides the experiment results. Finally, Section V concludes this article.

II. REVIEW OF SIMULATION APPROACHES AND FIBERS

A. STATEVECTOR

The most straightforward approach to simulating quantum circuits is storing quantum states and gates using 1-D and 2-D arrays. The obvious disadvantage of this approach is its exponential memory usage: given n qubits, it needs 2^n amplitudes to describe the composite system. The simulation scale (i.e., the number of qubits) entirely depends on the memory size. Nevertheless, given sufficient memory, this approach is often faster than others, as arrays can be stored contiguously in memory to accelerate memory accesses. A plain array-based approach is easy to implement while only supporting a few qubits. Distributed and supercomputers are adopted to extend memory size with multithreading processing [11], [12], [13], [14]. Fang et al. [15] proposed to partition the circuit into hierarchical subcircuits with the iterative construction of smaller state vectors. State vectors can also be compressed. Different compression techniques offer different compression ratios at the cost of accuracy [16], [17].

B. TENSOR NETWORK

Tensors are a mathematical concept that encapsulates the idea of multilinear maps. A collection of tensors

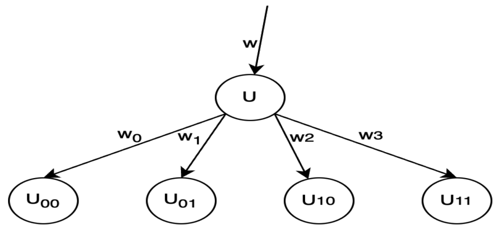

FIGURE 1. QMDD subtree.

connected by edges is a tensor network. The edges specify which indexes are contracted [18]. Tensor networks are a well-established tool for analyzing quantum systems, and they have also been adapted for simulating quantum circuits [19], [20], [21]. Quantum states and quantum gates are represented by low-rank tensors. The output state amplitudes of a quantum circuit are calculated by contracting the network, and it is generally known that the speed of the contraction is determined by the contraction order [22]. Finding the optimal contraction order is an NP-hard problem. Huang et al. [23] proposed to slice the network into portions with no dependency and contract them in parallel. Reinforcement learning is utilized in [24] to extract efficient contraction orders. Gray and Kourtis [25] implemented randomized protocols and conceived the tensor network as a hypergraph to find a close-to-optimal contraction order. One downside of tensor networks is their inability to support intermediate state measurement. As its space cost is exponential with the treewidth of the underlying graph, plus the fact that tensors are still stored using arrays, the qubit count it can simulate tends to be lower than DDs [26], [27].

C. DECISION DIAGRAM AND ITS CONCURRENCY

BDDs are proposed originally for representing Boolean functions as a graphical depiction of applying te Shannon decomposition [28]. They also provide an efficient means for modeling digital circuits [29], [30]. When we extend binary bits to quantum bits, BDDs are no longer the best choice for modeling quantum circuits. Variants have been proposed to emulate quantum circuits still using DDs but with added features to support complex amplitudes and higher dimensions. Our work is based on QMDDs [8]. QMDDs support complex-valued entries by attaching complex weights to edges. Quantum states and quantum operations are represented using QMDDs with two and four child nodes, respectively. This naturally corresponds to dividing a state vector and a gate matrix into two and four submatrices. Fig. 1 is an example of QMDD representing the matrix

$$U = \left[\begin{array}{c|c} U_{00} & U_{01} \\ \hline U_{10} & U_{11} \end{array}\right]. \tag{1}$$

Weights on the path from the root to a terminal are multiplied to get the values at the corresponding position. If Fig. 1 represents a 2×2 matrix U, the value of U_{00}, which is a single element, is $w \times w_0$. QMDDs achieve memory efficiency

Algorithm 1: sum(A, B).

```
Input: root node A, B
Output: sum C
if $A$ is terminal then
    $B.w \leftarrow B.w + A.w$
    return $B$
else if $B$ is terminal then
    $A.w \leftarrow A.w + B.w$
    return $A$
multiply the weights of all child nodes of $A$ by $A.w$
multiply the weights of all child nodes of $B$ by $B.w$
$C_{00} \leftarrow$ sum($A_{00}$, $B_{00}$)
$C_{01} \leftarrow$ sum($A_{01}$, $B_{01}$)
$C_{10} \leftarrow$ sum($A_{10}$, $B_{10}$)
$C_{11} \leftarrow$ sum($A_{11}$, $B_{11}$)
return $C$
```

Algorithm 2: Multiply(A, B).

```
Input: root node A, B
Output: product C
if $A$ is terminal then
    $B.w \leftarrow B.w * A.w$
    return $B$
else if $B$ is terminal then
    $A.w \leftarrow A.w * B.w$
    return $A$
multiply the weights of all child nodes of $A$ by $A.w$
multiply the weights of all child nodes of $B$ by $B.w$
$C_{00} \leftarrow$ sum(multiply($A_{00}$, $B_{00}$), multiply($A_{01}$, $B_{10}$))
$C_{01} \leftarrow$ sum(multiply($A_{00}$, $B_{01}$), multiply($A_{01}$, $B_{11}$))
$C_{10} \leftarrow$ sum(multiply($A_{10}$, $B_{00}$), multiply($A_{11}$, $B_{10}$))
$C_{11} \leftarrow$ sum(multiply($A_{10}$, $B_{01}$), multiply($A_{11}$, $B_{11}$))
return $C$
```

via the exploitation of redundancy. For example, submatrices containing identical values can be represented using a single node, regardless of its size.

The structure of QMDDs makes the implementation of linear algebra computations straightforward. For example, assume two matrices A and B are of the form

$$A = \left[\begin{array}{c|c} A_{00} & A_{01} \\ \hline A_{10} & A_{11} \end{array}\right] \quad B = \left[\begin{array}{c|c} B_{00} & B_{01} \\ \hline B_{10} & B_{11} \end{array}\right]. \tag{2}$$

Then their product is

$$\left[\begin{array}{c|c} A_{00} \cdot B_{00} + A_{01} \cdot B_{10} & A_{00} \cdot B_{01} + A_{01} \cdot B_{11} \\ \hline A_{10} \cdot B_{00} + A_{11} \cdot B_{10} & A_{10} \cdot B_{01} + A_{11} \cdot B_{11} \end{array}\right]. \tag{3}$$

We can translate this into the pseudocodes for computing the sum and multiplication between two QMDDs, as shown in Algorithms 1 and 2.

The proposal of DDs provides both symbolic and compact data representation which exhibit its major advantage in space efficiency. Given sufficient sharing of gate matrices and state vectors, replacing the statevector-based representation with DDs can yield substantially lower memory usage.

This leads to researchers' attention to applying DDs to quantum simulation which is a memory-intensive task. Nevertheless, its time efficiency compared with the array-based simulation is not yet conclusive due to the difficult parallelization incurred by its higher level of data sharing. Zulehner and Wille [31] revisited the potential of reordering simulation operations in DD-based simulation. In the conventional array-based simulation, it tends to first multiply gate matrices with the current state vector since matrix-vector multiplication is typically cheaper than matrix–matrix multiplication. Zulehner et al. [31] observed that DD's symbolic representation makes quantum gates structurally more compact than state vectors, thus combining them first can have the prospect of more efficient simulation. They further discuss and compare the strategies for deciding how many gates to be combined. The combination of quantum gates is also leveraged in our design. Hillmich et al. [32] recognized the potential as well as the obstacles of adding concurrency in DD-based quantum simulation. They identify that subnodes of DDs can be processed in parallel to speed up simulation. However, the improvement is not guaranteed due to the possibility of computing identical operations more than once during concurrent execution. They propose to customize the operation cache into separate multiplication and addition caches to address this issue. We extend this direction by proposing new parallelization schemas and studying the effects of operation caches in a concurrent context. Our experiments reveal that operation caches can be either made thread-local or removed to speed up the concurrent simulation of certain circuits, which has not been recognized before in previous work.

D. FIBERS

Fibers are lightweight execution context similar to threads. The fundamental difference between fibers and threads is their scheduling mechanism. Modern operating systems run multiple processes with their own execution context at any given moment. However, not all processes are running in parallel. The OS scheduler switches between them quickly so that they appear to be running in parallel. When threads are used as the execution context, they do not need to yield to other threads to allow them to run because the OS scheduler preemptively switches among them. There is a list of points at which the OS will save the state of one thread then resume another thread. These include IO, interrupts, sleeps, etc. The overheads associated with this context switching process begin to become prohibitive when the number of threads is large [33]. One proposed solution is to move the context switching decision from the kernel space to the user space and this effectively leads to the concept of cooperative scheduling and the implementation of fibers. Fibers include switching as a part of computation and deliberately decide when to relinquish control. This allows keeping an excess number of execution contexts. DD operations can generate many tasks with unbalanced workloads depending on the structure of the tree and the caching result. We find that using fibers can reduce the overheads of maintaining and switching among many execution contexts.

III. DESIGN OF PARALLEL SIMULATION ENGINE

The operation cache and unique table [9] are two core components of DD libraries. They contribute to DDs' time and space efficiency, respectively. The unique table stores a unique node representing all identical submatrices and each DD containing such a submatrix keeps an identifier referring to this node, e.g., the key in the table. This substantially reduces the memory consumption incurred in the statevector approach which does not explore the repetitiveness of submatrices. The uniqueness also allows caching calculation results. Since the identical matrices share a unique identifier, we can readily check whether the operation performed on two matrices has been conducted before. The operation cache allows the query of results using the matrix identifiers. In a parallel simulation engine, the first question to answer is how to synchronize these data structures with the least negative effects on performance. Synchronization is necessary to guarantee correct calculation, however, it can come with a considerable sacrifice of performance without careful tuning. These points are significant issues in designing a simulator. One goal of this study is to find when and how we can remove the need for synchronization in DD-based parallel quantum simulation. The second goal is finding the parallelization schema that most appropriately fits quantum simulation. We propose the architecture in Fig. 2. In general, we find fibers have better performance than threads- and tasks-based parallelization. The performance improvement comes from their cooperative scheduling, lightweight bookkeeping, and flexible load balancing. We examine each of these in this section. The unique table needs to be global, whereas the choice between a global and local cache depends on the characteristics of the simulated circuit. Work stealing can offer automatic load balancing. This is especially important because it is not uncommon to have significantly unbalanced DDs. Free fibers which are assigned subtrees with few child nodes can take over nodes from busy fibers. Our experiments show this architecture uses resources better and accelerates simulations by 2–3 times than a single-threaded DD-based simulation. This section illustrates each design choice.

A. THREAD LOCAL VERSUS GLOBAL UNIQUE TABLE

DDs achieve a higher memory efficiency by using the unique table. A single copy of identical nodes is stored in the unique table and shared among DDs. Thus, we avoid duplicating the storage of the same matrices. The unique table is essential to achieve better memory efficiency as long as the matrix is not entirely random and where no submatrices can be shared. The unique table needs to support fast inserts and lookups since it is on the critical path of the simulation. In our prototype system, the unique table is implemented using a hashtable and we resolve the hash collision with chaining. We leave the study of other hashtable variations for future work. We need to decide between the alternatives of thread

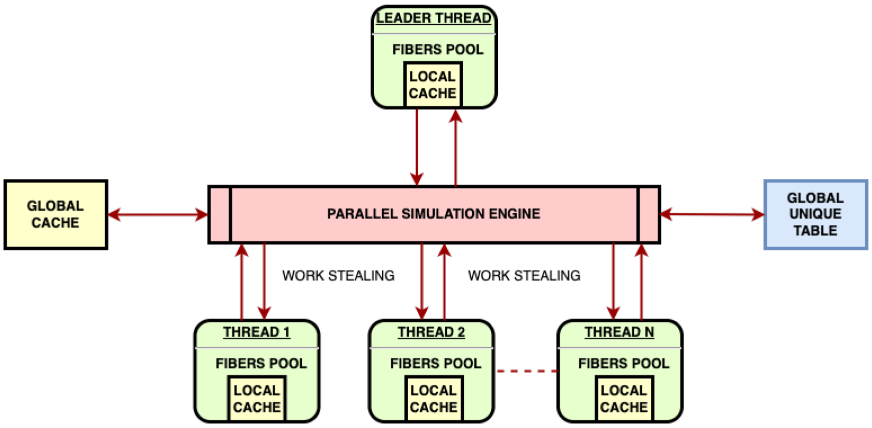

FIGURE 2. **Proposed parallel simulation architecture.**

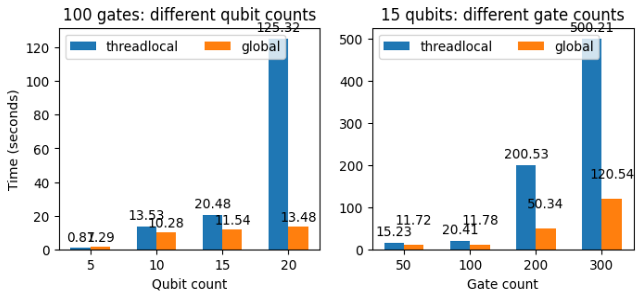

FIGURE 3. **Comparison between thread local and global unique table.**

local and global unique tables. A thread local unique table can be accessed with no synchronization. However, it incurs duplication and higher memory consumption as the same node not present in a thread's local table may appear in other tables. The higher memory cost restricts the supported qubit number. Besides the extra memory consumption, the fundamental issue of using a local unique table is that it renders the operation cache hard to hit. The operation cache uses the memory addresses of operand nodes as the unique identifier for equality checking, and if the result is present in the cache, the complete computation can be eliminated. Achieving a high operation cache hit ratio is essential in a high-performance DD-based quantum simulation. Using thread local unique tables generates different addresses for the same nodes if they are created by different threads. This does not impact the computation accuracy; however, the cache hit ratio dramatically drops.

We conduct an experiment to see the variation in performance from using a thread local or global unique table. Fig. 3 compares a thread-local and global unique table in a random circuit. The circuit consists of 100 gates randomly sampled from the set of RX, RY, RZ, and CNOT. The running time shown in the figure is the average value of ten circuits. We conclude that a thread local unique table is inferior to a global unique table for both higher qubit and gate counts. The former suffers from a considerable drop in the cache hit ratio. For example, in a circuit with 100 gates and 20 qubits, the multiplication operation cache hit ratio drops from 17.54% to nearly none. Therefore, a global unique table is used in the rest of the comparison. The synchronization is managed by updating the pointer pointing to the next entry in the linked list using compare-and-swap. A gradually increasing number of hashtable entries are preallocated on each thread's stack to avoid frequent and fragmented memory allocations. Other techniques for implementing a concurrent hashtable with better performance can be applied [34].

B. THREAD LOCAL VERSUS GLOBAL OPERATION CACHE

Having a unique copy of identical nodes in DDs allows us to cache previous calculation results and later query the results using the unique node identifier easily. The elimination of the complete calculation is central to the DD library's performance as well as the simulation speed. In our implementation, the node address is used as its identifier. The equality of two submatrices can be simply checked by comparing their addresses. This is nontrivial to realize in state vector-based approaches. The equality of two matrices cannot be easily checked without iterating through all of their elements.

When the operation cache successfully serves the query, we manage to skip all subsequent calculations. Similar to the alternatives we have explored in the case of a unique table, the operation cache can be either global or local. A global operation cache shares calculation results conducted by any thread with others, whereas a local operation cache only serves its owner. The global cache can theoretically cache more computations with an additional cost of synchronization. However, our experiment demonstrates that this does not necessarily apply to the task of quantum simulation since quantum simulations exhibit both time and spatial locality. First, besides some circuit identities, most quantum gates do not commute, and thus, must be applied in sequence. Furthermore, each quantum gate (i.e., DD) is constructed with tensor products between identity gates and the target gate. This makes the resulting matrix contain many subidentities that can preserve subtrees from the input state to the output state. The locality supports the use of thread local caches.

QMDDs divide each matrix into four submatrices, and most cache hits happen among the calculations on these

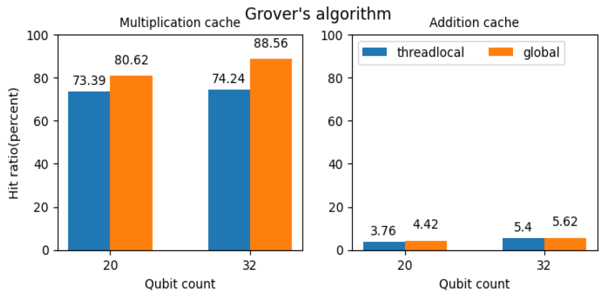

FIGURE 4. Comparison between thread local and global operation cache.

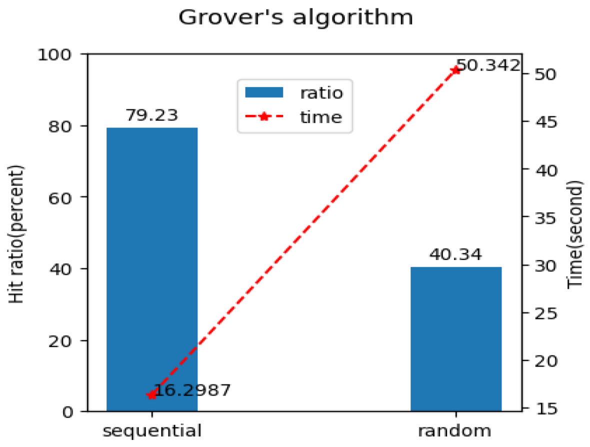

FIGURE 5. Grover's algorithm with different processing orders.

submatrices since quantum matrices tend to be structured (e.g., symmetric). The majority of these calculations are done within a single thread. This also implies the use of thread local caches. We observe that the hit ratio of using thread local caches is only lower than that of a global cache to a slight extent but comes with no cost of synchronization. Notably, the importance of operation caches also varies according to quantum algorithms. For example, the Grover's algorithm repeatedly applies the same Grover iteration. This makes the operation cache important. They become less useful in more random circuits. We can leverage certain linear algebra properties to improve the cache efficiency. Multiplication is a typical linear operation. This allows us to normalize the operands. In the case of QMDDs, the operation cache only needs to store node addresses but not their edge weights. This makes the multiplication cache more likely to hit. Instead, addition commutes. Thus the addition cache needs to check both orders of the operands. However, since weights are floating-point complex values, the cache hit ratios depend on the tolerance used in equality checking and the parallelization strategy. This means it is less likely to hit in the addition cache and its hit ratio will be remarkably lower unless we set a larger threshold and sacrifice the accuracy. In Fig. 4, we present a comparison in the Grover's algorithm. The gap between the local and global cache hit ratio is small. The local cache relieves the burden of synchronization and leads to better performance. We also see that the hit ratio of the multiplication is tens of times higher than the local cache. The hit ratios for multiplication and addition are virtually zero for a random circuit, so they are not shown. We suggest that, from the cost-effectiveness perspective, assigning a larger memory space to the multiplication cache is more sensible.

C. FIBERS VERSUS THREADS

Fibers are a lightweight execution context. They are similar to operating system threads in the sense that they are both abstract concept and implemented as a data structure encapsulating data necessary for execution. Fibers are in the userspace and their implementation relies on the underlying OS threads, thus, they share some data with the underlying threads. Multiple fibers can be scheduled onto the same thread. These fibers have their own stacks but share a single address space with the underlying thread. This means the same amount of memory can support more fibers, and the context switch among fibers is faster since the operating system does not need to switch the page table.

The principal difference between fibers and threads is how they are scheduled. Fibers use cooperative scheduling. The idea is that they deliberately yield to others at a chosen point. It should be noted that using fibers does not provide extra concurrency. The concurrency level still depends on the number of operating system threads (hardware threads, to be precise). Within a single thread, only one fiber can execute at a single point in time. The benefit of this is that no extra synchronization needs to be handled at the level of fibers as long as threads are appropriately synchronized. Use cases of fibers often involve many blockings or small computations. When a fiber blocks due to I/O or locking, it can voluntarily yield to others. This is less costly than a context switch between threads. The primary advantage fibers exhibit in quantum simulation is its automatic load balancing. Depending on the pattern of a quantum gate matrix, its corresponding DD can be imbalanced. We approach this by creating a large number of fibers for carrying out computation on subtrees. Threads can be kept busy by always having the next fiber available when they complete one.

D. PARALLELIZATION SCHEMA

We analyze three strategies for parallelizing the simulation process: 1) task-based outer parallelization; 2) thread-based inner parallelization; and 3) fiber-based inner parallelization.

Consider a quantum circuit representing the state evolution of $|\text{output}\rangle = U_3U_2U_1U_0|\text{input}\rangle$. The task-based outer parallelization launches a fixed number of worker threads and creates tasks for each multiplication $U_{i+1}U_i$ or $U_i|\text{state}\rangle$. The noncommutativity of multiplication induces dependencies among tasks. Task dependencies are handled by constructing a task graph with nodes representing operations and edges representing task prerequisites. Fig. 6 shows a toy example. In the task graph, we use *MulMV* and *MulMM* to represent the operation of matrix–vector multiplication and matrix–matrix multiplication. Such a task graph does

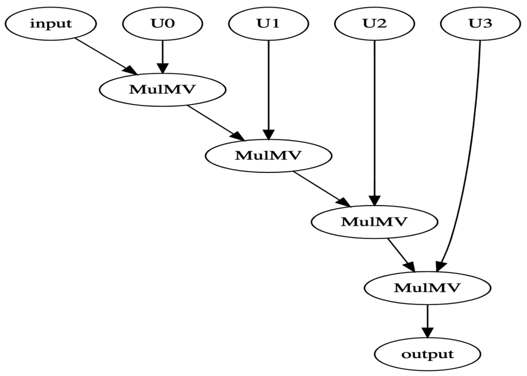

FIGURE 6. Task graph with linear dependency.

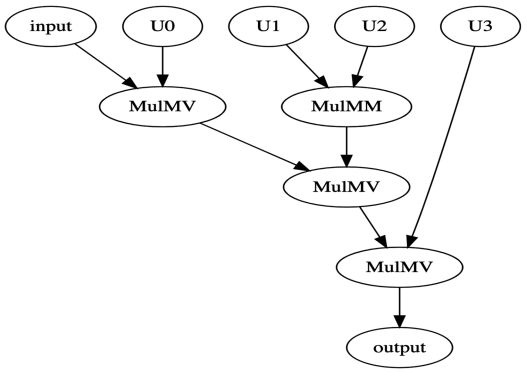

FIGURE 7. Task graph without linear dependency.

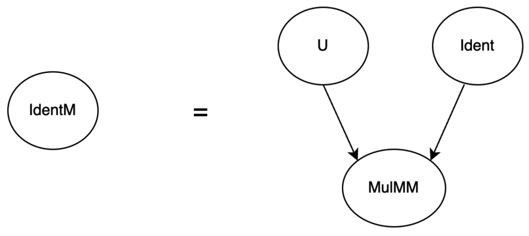

FIGURE 8. Equivalence between IdentM and a matrix-matrix multiplication.

not offer any true concurrency: later threads must wait for the completion of previous threads. For example, all *MulMV* nodes must be executed in a linear sequence. We improve this by leveraging the fact that multiplication is associative. Thus worker threads can simulate different parts of the circuit simultaneously as long as the order is respected. We use associativity to construct the task graph in Fig. 7. Dependencies among *MulMV* nodes are removed, and thus, they can be executed in parallel. We call this *outer parallelization* since subtrees of a single DD are all processed by the same thread. The whole task graph is processed in parallel by multiple threads. Nevertheless, we find this approach performs poorly in our experiment.

Past works have observed that matrix–vector multiplication is generally faster than matrix–matrix multiplication for simulating quantum circuits [31]. This also applies to DDs. Therefore, it is more efficient to reduce the dimension faster and earlier by allocating more resources (i.e., worker threads) to compute matrix–vector multiplication rather than matrix–matrix multiplication. However, *MulMV* nodes in Fig. 7, the only type of node generating a vector and reducing the dimension, were taken by a single thread.

The solution is constructing a task graph in which dependencies are set such that tasks are processed in batches following the flow of the circuit. This means worker threads work jointly in a subpart of the circuit (i.e., gates closer to each other) to accelerate the dimension reduction and to avoid working on arbitrary parts of the circuit (i.e., gates further apart). We construct such a task graph by introducing a node of type *reduce*. It serves as a *hub* in the graph. The *reduce* nodes segregate the graph and enforce worker threads to work on earlier parts of the circuit before later parts can be processed. Fig. 9 shows an example of such a graph. We use the node IdentM to mean a unitary gate matrix multiplied by an identity matrix to save space. The equivalence is shown in Fig. 8. In this example, we add three reduce hub nodes. These hubs enforce worker threads to concentrate on smaller parts of the circuit so that caches are more likely to hit. The update of Fig. 9 from Fig. 7 is that worker threads are no longer allowed to work on arbitrary parts of the circuit. They are forced to first reach the reduced node and then proceed to the next subregion of the circuit. In our experiment, we find the inclusion of *reduce* nodes improves the performance by approximately two times compared with the other two kinds of task graph.

The vital problem of the *outer parallelization* is its poor cache utilization. As threads may take arbitrary task nodes, DDs processed by a single thread can come from remote parts of the circuit, and thus, exhibit low similarities. This means cached results are rarely reused, leading to a lower cache hit ratio. In Fig. 5, we run the Grover's algorithm with a single thread using the same task graph but different processing orders: one for sequentially multiplying DDs and one for picking DDs randomly. It illustrates that random processing decreases the cache hit ratio and leads to a longer execution time.

Inner parallelization does not simulate the entire circuit concurrently. Quantum gates are applied to the start state vector in sequence following the circuit order, and only one *main* thread carries this out. What distinguishes this strategy is that different threads will process subtrees of a single DD instead of working on different parts of the circuit. We create fibers when the operation is recursively applied on child nodes until a lower bond of qubit count is reached to avoid having too many fibers. We determine this threshold by benchmarking the time spent in constructing two fibers for two DDs of a certain qubit number (this involves the creation of data structures and the addition to the fiber manager's task queue) and the time spent in multiplying these two DDs

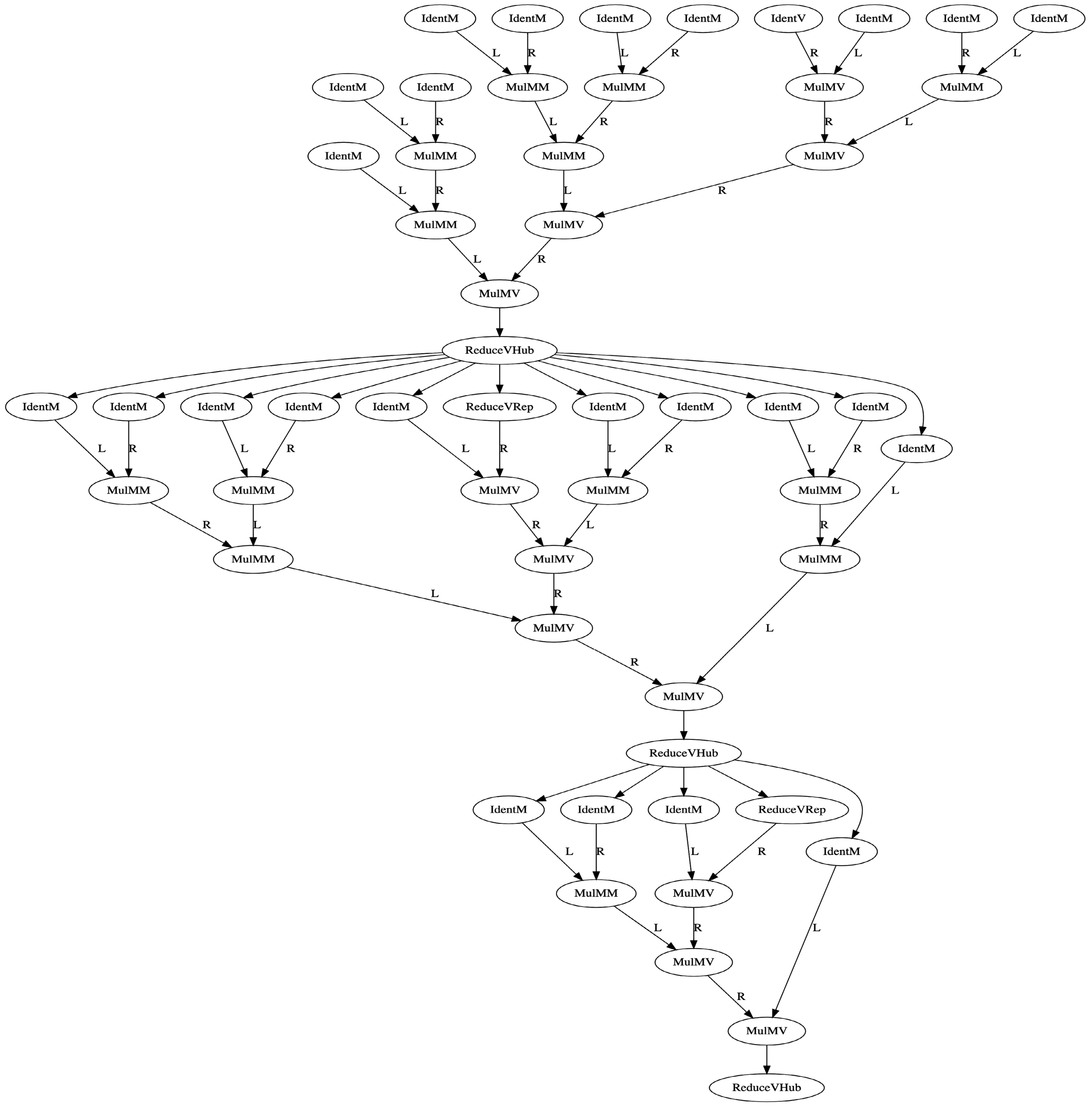

FIGURE 9. Task graph with *reduce* nodes.

without using fibers. If the latter is faster, it suggests we should stop creating new fibers at this number of qubits. The results on our machine are provided in Table 5. We fix this threshold to 4 qubits in all experiments. *Inner parallelization with fibers* offers the following advantages. First, fibers are dynamic, whereas task graphs are static. This means we can query the operation cache first and avoid launching a new fiber if it returns a hit. Second, inner parallelization boosts cache efficiency. Quantum circuits tend to evolve the input state gradually. At each step, quantum gates are applied to the state vector locally. This means a majority of subtrees in the DD remain unchanged across each gate. Third, workloads on different subtrees are imbalanced (i.e., different tree sizes) due to the gate matrix pattern. Fibers can be inexpensively created, destroyed, and migrated among threads. Free threads can pick fibers from the fiber pool and this realizes an automatic load balancing. In our experiment, we discover that worker threads spend over 50% of time idling and waiting for tasks in *outer parallelization*. In *fiber-base inner parallelization*, this falls to approximately 10%.

IV. EXPERIMENTS

To evaluate the effectiveness of each parallelization strategy, we implement a QMDD-based quantum simulator in

TABLE 1. Results on Grover's Algorithm

Qubit	Local Cache				Global Cache				
	single	fibers	OpenMP	taskgraph	fibers	OpenMP	taskgraph	DDSIM	Qiskit Aer
10	0.701	1.231	2.451	0.122	0.054	0.512	0.964	0.353	**0.053**
12	0.854	3.789	3.732	0.255	**0.044**	0.863	0.863	0.324	1.788
15	0.833	3.995	4.125	1.786	**0.037**	0.076	0.984	0.475	538.869
18	0.856	3.865	5.915	178.315	**0.017**	0.187	76.253	0.466	OOM
21	1.061	4.124	6.663	1756.357	**0.148**	0.474	679.356	0.874	
24	1.223	5.062	7.027	5928.232	**0.368**	1.668	4486.656	3.129	
27	7.654	2.132	8.095	TIMEOUT[2]	**1.442**	6.055	TIMEOUT	13.254	
28	8.734	**2.967**	10.323		3.065	9.346		19.074	
29	10.825	5.075	16.627		**4.869**	21.364		29.465	
30	16.298	**10.157**	15.486		10.422	50.331		49.016	
31	51.677	20.722	107.316		**20.341**	176.749		62.346	
32	94.687	**50.865**	193.427		91.261	385.245		89.123	
33	130.568	**45.338**	504.925		720.365	OOM[1]		137.456	
34	330.462	**82.455**	OOM		2400.412	OOM		197.957	

[1] Out Of Memory
[2] Exceed 7200 seconds
The bold values represent the fastest case in the row.

C++ compiled with *GCC 11.4.0*. We implement our task graph engine. We use *Boost fibers* and *OpenMP*, [35] for *fiber-based* and *thread-based* parallelization. Work-stealing is enabled for load balancing. The implementation as well as the benchmark scripts (under. /scripts) is open-sourced for the proof of concept.[1]

In the experiments, we compare with the state-of-the-art QMDD-based simulator DDSIM and *Qiskit Aer*, [36] with *default qubit* state vector-based backend. We conduct the experiments on a server with AMD Ryzen 9 7950X 16-core processor and 128 GB RAM. The Linux kernel is 5.19.0. We set the timeout limit to 7200 s. The Linux kernel manages out-of-memory abortions. We fix the number of threads to 16 in fiber, taskgraph, and OpenMP experiments. This corresponds to the number of cores in our experiment machine. Each thread is pinned to a physical core to minimize the impacts of thread migration. Our experiment parameters are summarized in table [6].

Table 1 presents the result of the Grover's algorithm. Our implementation first creates the unitary gate for one complete Grover iteration, the number of Grover's iterations is computed by $\lfloor \frac{\pi\sqrt{N}}{4} \rfloor$($N$ is the number of qubit). The oracle we use can be described by $U|x\rangle = (-1)^{f(x)}|x\rangle$, where $f(x) = 1$ only for a single input. The number of threads is fixed to 16 for all multithread cases. Each thread is pinned to a separate core to remove the impacts from thread migration. The results show that *taskgraph* performs poorly for both local and global cache. Substantial overheads come from traversing the task graph and waiting for dependencies. The cache hit ratio of the Grover's algorithm is expected to be high because of its repeated applications of the Grover's iteration. However, we find the hit ratio is below 10% in *taskgraph* due to worker threads randomly working on different parts of the circuit. This pollutes the cache content and leads to unnecessary evictions. Another issue is the imbalanced workloads from different quantum gates. For quantum gate matrices with complex structures and random entries, we observe that many threads are idle while the others are kept busy. This suggests task-based parallelization is not suitable for simulating quantum algorithms. *Qiskit Aer* uses the state vector. It outflows the memory above 18 qubits on our server. Qiskit Aer provides *max_memory_mb* to set the maximum size of memory to store the state vector. In our experiment, we set this to 0 so it will be set to the maximum allowable system memory size. We also assign the minimum out-of-memory score to the experiment process to ensure Linux does not kill it too early. It is also slower than all the other DD-based approaches.

Another common approach for adding parallelization is with OpenMP. When *OpenMP* is used for parallelization, using a global cache accelerates the simulation for small qubit counts. The global cache simulates more than ten times faster than the local cache when the number of qubits is below 24. Local caches start to boost the performance when the qubit count gets larger. We also observe the same phenomenon in the case of fibers. The profiling results suggest the following reasons. First, higher qubit counts generally require a longer execution time and more cache accesses. This makes the costs associated with locking the cache more severe. Second, the size of DDs also increases with qubit count. Therefore, more subtrees share the same structure, thus the same cache bucket. Using separate local caches alleviates cache contention.

The performance of DDSIM is among the first tier. Our single-thread implementation targets on matching DDSIM to serve as a reasonable baseline. Therefore, their results are comparable. DDSIM uses a single thread. However, it is superior to the 16-thread *OpenMP* implementation with a global cache. This confirms the negative impacts of unduly synchronization. In our experiments, when the qubit count is below 27, fibers should be used in conjunction with a global cache. This offers the optimal performance among all strategies we have benchmarked. With more qubits, fibers should

[1] https://github.com/Fujitsu-UTokyo-QDD/QDD/tree/journal

TABLE 2. Results on Random Circuit (Number of Gates = 200)

Qubit	No Cache				Local Cache				Global Cache				
	single	fibers	OpenMP	taskgraph	single	fibers	OpenMP	taskgraph	fibers	OpenMP	taskgraph	DDSIM	Qiskit Aer
10	0.04	0.02	0.26	1.466	0.9	10.55	5.29	6.54	0.13	1.85	12.54	0.05	**0.02**
12	0.15	0.06	1.05	2.63	1.06	11.73	6.05	10.34	0.44	7.68	254.24	0.19	**0.03**
14	0.64	0.16	3.99	15.67	1.75	10.97	8.92	88.23	1.41	28.85	873.44	6.61	**0.04**
16	3.14	**0.57**	14.73	164.63	5.01	12.58	18.79	364.52	4.48	95.34	2034.23	55.13	1.42
18	35.85	**5.38**	82.55	1042.53	42.22	17.65	92.23	3442.46	26.09	460.02	TIMEOUT	258.16	17.72
20	147.69	**21.18**	314.51	5654.52	159.45	35.65	305.23	TIMEOUT	78.19	1353.54		436.24	82.55
22	4804.88	**631.77**	6226.24	TIMEOUT	4935.11	656.17	6403.21		731.65	TIMEOUT		6942.45	725.07
24	TIMEOUT	**1258.33**	TIMEOUT		TIMEOUT	1358.23	TIMEOUT		1658.24			TIMEOUT	2593.53
26		**1484.64**				1648.35			1994.23				OOM
28		**2346.35**				2476.42			2857.34				
30		**3994.44**				4124.53			4572.42				

The bold values represent the fastest case in the row.

TABLE 3. Results on Clifford + T circuit (Number of Gates = 200)

Qubit	No Cache				Local Cache				Global Cache				
	single	fibers	OpenMP	taskgraph	single	fibers	OpenMP	taskgraph	fibers	OpenMP	taskgraph	DDSIM	Qiskit Aer
10	0.03	0.02	0.24	1.50	0.03	0.02	0.21	1.44	0.11	1.54	11.20	0.04	**0.01**
12	0.14	0.05	1.02	2.45	0.17	0.04	0.99	2.13	0.34	6.33	56.34	0.12	**0.02**
14	0.71	0.12	3.21	14.94	0.64	0.1	2.99	11.45	1.01	19.21	102.29	4.51	**0.04**
16	2.98	0.44	12.53	142.52	2.56	**0.34**	9.23	133.34	3.22	53.35	299.34	45.93	0.92
18	24.54	5.21	76.95	995.84	19.23	**3.52**	60.26	792.23	19.23	323.34	1231.04	198.17	12.72
20	125.65	20.28	304.23	3753.61	88.26	**15.23**	245.23	3001.69	52.21	998.03	4472.02	326.26	62.55
22	3782.23	572.45	4723.62	6932.22	2993.02	**509.92**	3245.32	4728.01	642.92	3234.12	TIMEOUT	2942.26	487.04
24	4923.34	1002.56	6993.52	TIMEOUT	3829.27	**787.92**	5782.29	6723.34	1198.28	4992.01		6782.92	1654.91
26	6933.12	1492.31	TIMEOUT		4672.29	**998.27**	TIMEOUT	TIMEOUT	1762.92	TIMEOUT		TIMEOUT	OOM
28	TIMEOUT	1992.23			6237.21	**1223.49**			2233.23				
30		3423.45			TIMEOUT	**2563.91**			3994.24				

The bold values represent the fastest case in the row.

be used with local caches. This combination is the only one superseding DDSIM in our experiments, and it reduces the simulation time by 3–4 times.

The issue of a single-thread DD simulator becomes more evident in a random circuit. In Table 2, a pure random circuit with 200 gates is tested. We randomly sample 200 gates from the gate set of *RX*, *RY*, *RZ*, and CNOT. Each gate's rotation angle and target qubit are also uniformly distributed. DDs are generally unsuitable for highly random circuits as the operation cache barely helps, and random memory accesses incurred from traversing the decision tree further hurt the performance. Consequently, the *no cache* case performs the best for larger qubit counts. *Global cache* performs worse than *local cache* as it cannot obtain a higher cache hit ratio but adds synchronization overheads. *Qiskit*'s performance is satisfactory. It is faster than DDSIM for all qubit counts before it outflows the memory. We cannot observe any advantage from using *OpenMP* than a single-thread implementation, suggesting that naive parallelization can, in fact, hurt the performance. Purely random circuits are not common in practice. A better approach to universal quantum computation is using Clifford (those generated by CNOT, Hadamard, and Phase gates) and T gates. These gates are more controlled in rotation angles; thus, the circuits have potentially more repetitive patterns to be exploited by DDs. This has been observed in Table 3, where we randomly sampled from a set of Clifford and T gates. Using fibers with local caches still performs better than the global cache in the experiment. This illustrates the relatively large impact of synchronization on performance even with more data sharing.

Our fiber-based implementation also accelerates the simulation of the Shor's algorithm. The Qiskit Aer execution is killed by the OS out-of-memory killer for factoring 253 while our approach can simulate with a speed-up of several orders of magnitude. The implementation of the Shor's algorithm is based on [37]. Note that the execution time of the Shor's algorithm is highly dependent on implementation details and the choice of the coprime. In Table 4, we only show the results

TABLE 4. Results on Shor's Algorithm

Number_Coprime	fibers(local cache)	Qiskit Aer
15_2	**0.235**	1.463
21_2	**1.523**	80.342
33_5	**8.323**	2242.643
69_2	**724.654**	3543.223
253_2	**1564.342**	OOM

The bold values represent the fastest case in the row.

TABLE 5. Time of Fiber Construction in Milliseconds

Number of qubit	Fiber construction	DD multiplication
1	89	34
2	78	62
3	102	90
4	94	143
5	88	285
6	101	412
7	92	934
8	87	1672

The bold values represent the fastest case in the row.

TABLE 6. Summary of Experiment Parameters

Parameter	Value
CPU	AMD Ryzen 9 7950X
Physical Memory	128GB
Physical Cores	16
Linux Version	5.1.1
GCC Version	11.4.0
Worker threads(including OpenMP)	16
oom_score_adj	-1000

with local caches which is the best among other alternatives in our benchmark.

V. CONCLUSION

In this article, we examined DD-based quantum simulation. Using QMDDs as an alternative to the basic BDDs in quantum simulators has been around for a while. Although its memory efficiency is superior to state vectors and tensor networks, its lack of parallelization strategies limits its scale. Components of DD libraries, such as the unique table and operation cache, complicate the adaptation to parallelization. Consequently, traditional thread-based strategies bring little, if any, performance improvement. Their synchronization costs occasionally are even harmful to the results. To address these problems, we present a comprehensive overview of the tradeoffs of several parallelization strategies. Furthermore, we propose a design that allows a faster simulation than the state-of-the-art single-threaded simulator DDSIM.

A. GARBAGE COLLECTION

Our prototype simulation engine does not implement garbage collection which limits the supported simulation scale. We believe that garbage collection is essential to support more qubits. However, supporting garbage collection in DD-based quantum simulation is challenging for the following reasons. First, the performance impact of the stop-the-world garbage collection is dramatic although it is easier to implement. On the other hand, a pause-free GC may not guarantee a complete free of memory and can be harder to implement correctly. Second, the alternatives between collecting unused nodes and leaving them for potential future usage require careful benchmarking to decide. Collecting unused nodes too aggressively in some quantum algorithms that exhibit a high reusage of nodes may be detrimental to performance. For a general-purpose quantum simulator, the designers remain unknown to the actual simulation circuits, therefore, it is challenging for them to tune the garbage collector.

B. HYBRID SIMULATION

We observe that when the qubit count drops below a certain number, the state vector-based simulation performs better than the DD-based simulation. The memory saving and caching benefits of the DD must outweigh its extra complexity over the state vector-based approach. Therefore, we propose to replace the DD nodes with the 2-D array-based matrices for the lower part of the tree to combine the benefits of these two approaches. The state vector-based approach has the following pros. First, its gate entries are stored linearly in the memory so that the hardware caches can be effectively used to speed up the memory access. In DDs, we trace child nodes using pointers which lead to random memory access. Second, computations over arrays can be easily parallelized. There is neither a unique table nor an operation cache to be synchronized. The threshold for switching to state vector-based simulation requires a study.

C. DISTRIBUTED SIMULATION

To further extend the simulation scale, it is central to leverage distributed resources. It is not easy to extend DD-based simulation to a distributed environment by using libraries, such as OpenMPI due to the existence of the unique table and operation cache. There are two alternatives. One is to maintain a global table that allows all machines to read and update. The other is to let each computing node maintain the table locally. A global table allows all node to share the work, however, we face the challenge of keeping the table consistent and its associated costs. Local tables are easier to maintain correctly but suffer from the waste of work performed by other nodes. Whether conducting distributed quantum simulations using DDs can improve the performance remains to be explored.

REFERENCES

[1] S. Li, Y. Kimura, H. Sato, J. Yu, and M. Fujita, “Parallelizing quantum simulation with decision diagrams,” in *Proc. IEEE Int. Conf. Quantum Softw.*, Los Alamitos, CA, USA, 2023, pp. 149–154, doi 10.1109/QSW59989.2023.00026.

[2] F. Arute et al., “Quantum supremacy using a programmable superconducting processor,” *Nature*, vol. 574, pp. 505–510, 2019, doi: 10.1038/s41586-019-1666-5.

[3] P. S. Emani et al., “Quantum computing at the frontiers of biological sciences,” *Nature Methods*, vol. 18, no. 7, pp. 701–709, Jan. 2021, doi: 10.1038/s41592-020-01004-3.

[4] F. Tennie and T. Palmer, “Quantum computers for weather and climate prediction: The good, the bad and the noisy,” *Bull. Amer. Meteorol. Soc.*, vol. 104, no. 2, doi: 10.1175/BAMS-D-22-0031.1.

[5] I. Hull, O. Sattath, E. Diamanti, and G. Wendin, “Quantum technology for economists,” Dec. 2020.

[6] G. Viamontes, I. Markov, and J. Hayes, “High-performance QuIDD-based simulation of quantum circuits,” in *Proc. Des.*, 2004, pp. 1354–1355, doi: 10.1109/DATE.2004.1269084.

[7] A. Abdollahi and M. Pedram, “Analysis and synthesis of quantum circuits by using quantum decision diagrams,” in *Proc. Des. Automat. Test Eur. Conf.*, vol. 1, 2006, pp. 1–6, doi: 10.1109/DATE.2006.244176.

[8] D. Miller and M. Thornton, “QMDD: A decision diagram structure for reversible and quantum circuits,” in *Proc. 36th Int. Symp. Multiple-Valued Log.*, 2006, pp. 30–30, doi: 10.1109/ISMVL.2006.35.

[9] Akers, “Binary decision diagrams,” *IEEE Trans. Comput.*, vol. C-27, no. 6, pp. 509–516, Jun. 1978, doi: 10.1109/TC.1978.1675141.

[10] T. Grurl, J. Fuß, and R. Wille, “Noise-aware quantum circuit simulation with decision diagrams,” *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 42, no. 3, pp. 860–873, Mar. 2023, doi: 10.1109/TCAD.2022.3182628.

[11] H. D. Raedt et al., “Massively parallel quantum computer simulator, eleven years later,” *Comput. Phys. Commun.*, vol. 237, pp. 47–61, Apr. 2019, doi: 10.1016/j.cpc.2018.11.005.

[12] M. Smelyanskiy, N. P. D. Sawaya, and A. Aspuru-Guzik, “qHiPSTER: The quantum high performance software testing environment,” 2016, *arXiv:1601.07195*, doi: 10.48550/arXiv.1601.07195.

[13] N. Khammassi, I. Ashraf, X. Fu, C. G. Almudever, and K. Bertels, “QX: A high-performance quantum computer simulation platform,” in *Proc. Des., Automat. & Test Eur. Conf. & Exhib.*, 2017, pp. 464–469, doi: 10.23919/DATE.2017.7927034.

[14] S. Imamura et al., “mpiQulacs: A distributed quantum computer simulator for A64FX-based cluster systems,” 2022, *arXiv:2203.16044*, doi: 10.48550/arXiv.2203.16044.

[15] B. Fang, M. Y. Özkaya, A. Li, Ü. V. Çatalyürek, and S. Krishnamoorthy, “Efficient hierarchical state vector simulation of quantum circuits via acyclic graph partitioning,” in *Proc. IEEE Int. Conf. Cluster Comput.*, 2022, pp. 289–300, doi: 10.1109/CLUSTER51413.2022.00041.

[16] X.-C. Wu et al., “Full-state quantum circuit simulation by using data compression,” in *Proc. Int. Conf. High Perform. Computing, Netw., Storage Anal.*, 2019, pp. 1–24, doi: 10.1145/2F3295500.3356155.

[17] X.-C. Wu, S. Di, F. Cappello, H. Finkel, Y. Alexeev, and F. T. Chong, "Amplitude-aware lossy compression for quantum circuit simulation," 2018, *arXiv:1911.04034*, doi: 10.48550/arXiv.1811.05140.

[18] J. C. Bridgeman and C. T. Chubb, "Hand-waving and interpretive dance: An introductory course on tensor networks," *J. Phys. A: Math. Theor.*, vol. 50, no. 22, May 2017, Art. no. 223001, doi: 10.1088/1751-8121/aa6dc3.

[19] R. Orús, "Tensor networks for complex quantum systems," *Nature Rev. Phys.*, vol. 1, no. 9, pp. 538–550, Aug. 2019, doi: 10.1038/2Fs42254-019-0086-7.

[20] Y. Pang, T. Hao, A. Dugad, Y. Zhou, and E. Solomonik, "Efficient 2D tensor network simulation of quantum systems," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2020, pp. 1–14, doi: 10.1109/SC41405.2020.00018.

[21] T. Nguyen, D. Lyakh, E. Dumitrescu, D. Clark, J. Larkin, and A. McCaskey, "Tensor network quantum virtual machine for simulating quantum circuits at exascale," *ACM Trans. Quantum Comput.*, vol. 4, no. 1, pp. 1–21, Oct. 2022, doi: 10.1145/3547334.

[22] I. L. Markov and Y. Shi, "Simulating quantum computation by contracting tensor networks," *SIAM J. Comput.*, vol. 38, no. 3, pp. 963–981, 2008, doi: 10.1137/050644756.

[23] C. Huang et al., "Efficient parallelization of tensor network contraction for simulating quantum computation," *Nature Comput. Sci.*, vol. 1, no. 9, pp. 578–587, Sep. 2021, doi: 10.1038/s43588-021-00119-7.

[24] E. Meirom, H. Maron, S. Mannor, and G. Chechik, "Optimizing tensor network contraction using reinforcement learning," in *Proc. 39th Int. Conf. Mach. Learn.*, 2022, pp. 15278–15292, doi: 10.48550/arXiv.2204.09052.

[25] J. Gray and S. Kourtis, "Hyper-optimized tensor network contraction," *Quantum*, vol. 5, Mar. 2021, Art. no. 410, doi: 10.22331/q-2021-03-15-410.

[26] J. Chen, F. Zhang, C. Huang, M. Newman, and Y. Shi, "Classical simulation of intermediate-size quantum circuits," 2018, *arXiv:1805.01450*, doi: 10.48550/arXiv.1805.01450.

[27] S. Boixo, S. V. Isakov, V. N. Smelyanskiy, and H. Neven, "Simulation of low-depth quantum circuits as complex undirected graphical models," 2017, *arXiv:1712.05384*, doi: 10.48550/arXiv.1712.05384.

[28] R. E. Bryant, "Symbolic Boolean manipulation with ordered binary-decision diagrams," *ACM Comput. Surv.*, vol. 24, no. 3, pp. 293–318, Sep. 1992, doi: 10.1145/136035.136043.

[29] C. A. J. van Eijk, "A BDD-based verification method for large synthesized circuits," *Integration*, vol. 23, no. 2, pp. 131–149, 1997, doi: 10.1016/S0167-9260(97)00018-7.

[30] R. Drechsler, "Polynomial circuit verification using BDDs," in *Proc. 5th Int. Conf. Elect., Electron., Commun., Computer Technol. Optim. Techn.*, 2021, pp. 49–52, doi: 10.1109/ICEECCOT52851.2021.9707932.

[31] A. Zulehner and R. Wille, "Matrix-vector vs. matrix–matrix multiplication: Potential in DD-based simulation of quantum computations," in *Proc. Des., Automat. Test Eur. Conf. Exhib.*, 2019, pp. 90–95, doi: 10.23919/DATE.2019.8714836.

[32] S. Hillmich, A. Zulehner, and R. Wille, "Concurrency in DD-based quantum circuit simulation," in *Proc. 25th Asia South Pacific Des. Automat. Conf.*, 2020, pp. 115–120. doi: 10.1109/ASP-DAC47756.2020.9045711.

[33] C. Li, C. Ding, and K. Shen, "Quantifying the cost of context switch," in *Proc. Workshop Exp. Computer Sci.*, New York, NY, USA, 2007, pp. 1–4, doi: 10.1145/1281700.1281702.

[34] T. Maier, P. Sanders, and R. Dementiev, "Concurrent hash tables: Fast and general(?)!," *ACM Trans. Parallel Comput.*, vol. 5, no. 4, pp. 1–32, Feb. 2019, doi: 10.1145/3309206.

[35] L. Dagum and R. Menon, "OpenMP: An industry standard API for shared-memory programming," *Comput. Sci. Eng.*, vol. 5, no. 1, pp. 46–55, Jan./Mar. 1998, doi: 10.1109/99.660313.

[36] H. A. et al., "Qiskit: An open-source framework for quantum computing," *ACM Trans. Quantum Comput.*, H. S. Travis and Y. Mingsheng, Eds., vol. 3, no. 3, Sep. 2022, doi: 10.1145/3543987.

[37] S. Beauregard, "Circuit for Shor's algorithm using 2n+3 qubits," *Quantum Info. Comput.*, vol. 3, no. 2, pp. 175–185, Mar. 2003, doi: 10.48550/arXiv.quant-ph/0205095.

Shaowen Li received the B.S. degree in computer engineering from the University of British Columbia, Vancouver, BC, Canada, in 2019, and the M.S. degree in electrical engineering and information systems, in 2021, from the University of Tokyo, Tokyo, Japan, where he is currently working toward the Ph.D. degree in computer science.

His research interests include operating system, blockchain, and quantum computing.

Yusuke Kimura received the B.S., M.S., and Ph.D. degrees in electronic engineering from the University of Tokyo, Tokyo, Japan, in 2015, 2017, and 2020, respectively.

He is currently a Researcher with Quantum Laboratory, Fujitsu Research, Tokyo. His research interests include verification of digital systems, program synthesis, and quantum computing simulation.

Hiroyuki Sato (Member, IEEE) received the B.Sc., M.Sc., and D.Sc. degrees in computer science from the Department of Information Science, University of Tokyo, Tokyo, Japan, in 1985, 1987, and 1990, respectively.

In 1990, he was an Assistant Professor with Kyushu University, Fukuoka, Japan. He is currently an Associate Professor with the Information Technology Center, University of Tokyo. His research interests include security and Internet trust.

Masahiro Fujita (Life Member, IEEE) received the Ph.D. degree in information engineering from the University of Tokyo, Tokyo, Japan, in 1985.

He joint Fujitsu, Tokyo, in 1985. From 1993 to 2000, he was assigned to Fujitsu Laboratories of America, Santa Clara, CA, USA, where he was the Director of the VLSI CAD research group. Since 2000, he has been a Professor with the VLSI Design and Education Center, University of Tokyo, until he retired in 2022. He is currently a Researcher with University of Tokyo. His research interests include hardware/software codesigns targeting embedded systems and formal analysis, verification, and synthesis of cyber–physical systems. Recently he is also working on hardware implementation of neural network-based information processing systems, such as learned image compression.

Dr. Fujita has been a program and steering committee member in many prestigious international conferences and journals.