TDAG: Tree-based Directed Acyclic Graph Partitioning for Quantum Circuits

***Abstract*—We propose the Tree-based Directed Acyclic Graph (TDAG) partitioning for quantum circuits, a novel quantum circuit partitioning method which partitions circuits by viewing them as a series of binary trees and selecting the tree containing the most gates. TDAG produces results of comparable quality (number of partitions) to an existing method called ScanPartitioner (an exhaustive search algorithm) with an 95% average reduction in execution time. Furthermore, TDAG improves compared to a faster partitioning method called QuickPartitioner by 45% in terms of quality of the results with minimal overhead in execution time.**

I. Introduction

QUANTUM computing is a developing computing field which applies the principles of quantum mechanics to solve problems more efficiently than optimal classical techniques. Quantum algorithms provide exponential speedups for problems in physics, chemistry, and mathematics, with promising applications in a wide variety of other areas. However, extracting the benefits of quantum computation has proven difficult due to the high error rates and relatively low qubit counts of Noisy Intermediate-Scale Quantum (NISQ) computers. Further, it has been challenging to design and implement large quantum circuits.

A promising method for reducing some of the complexity associated with quantum computing is quantum circuit partitioning. This technique splits a quantum circuit into smaller, more manageable pieces which can more easily be simulated by NISQ computers [1] [2] [3] [4], or more easily processed by classical algorithms [5] [6]. Many quantum circuit partitioning methods focus on assigning qubits to partitions such that the number of required telegate operations is minimized. This is called qubit partitioning. Because this approach splits multi-qubit gates, performing optimization on the resulting sub-circuits is difficult. An alterative partitioning scheme efficiently assigns quantum gates to partitions, such that the number of teledata operations is minimized. In this scheme, called gate partitioning, each partition can be considered separately, which allows applications in both circuit distribution and optimization.

A very common approach is partitioning a circuit by performing a balanced min-cut on the circuit's weighted coupling graph. This method, however, does not account for varying qubit connectivity across different regions of the circuit, which limits partition quality [7]. As a result, many recent works based on this approach [7] [8] [9] or another global approach [10] incorporate some measure of localized information when selecting cuts.

Several existing algorithms perform circuit partitioning for the purposes of circuit optimization, but these approaches sacrifice a significant amount of efficiency for the sake of performance, or vice versa. We chose to compare against a pair of algorithms, one of which is optimized for quality of result, and the other for speed. ScanPartitioner, which is proposed as part of QGo [5], uses a greedy approach with exhaustive search to select partitions. This approach produces good quality results, but has a fairly large time complexity of $O(gn^k)$. The other method, called QuickPartitioner [11], considers gates in topological order and progressively creates partitions. QuickPartitioner has a time complexity of approximately $O(gn)$, but produces significantly lower quality results than ScanPartitioner.

Thus, we propose Tree-based Directed Acyclic Graph (TDAG) partitioning, a novel iterative gate partitioning method which uses localized heuristics to partition a circuit. The algorithm accepts a circuit of n qubits with g gates and a constant $k \leq n$, and partitions the circuit into the minimum number of partitions, j, such that each partition has $\leq k$ qubits. This partitioning scheme reduces the scaling of quantum synthesis-based optimization of a circuit from exponential in n to exponential in k. By minimizing the number of partitions, the algorithm provides synthesis tools with the most information to optimize the resulting sub-circuits. Minimizing the number of partitions also has the effect of reducing the teleportation cost of a distributed circuit partitioned this way.

TDAG partitions circuits by viewing the gates as nodes in a series of binary trees and selecting the tree containing the most gates. The algorithm produces results of comparable quality to ScanPartitioner while having a time complexity of only $O(gn4^k)$. Since ScanPartitioner has a complexity of $O(gn^k)$, TDAG has a significant advantage both for circuits with a large number of qubits and for partitioning with large values of k. A comparison of TDAG, ScanPartitioner, and QuickPartitioner across a set of benchmark circuits is provided for $k = 4$ and 5. These benchmarks include Quantum Fourier Transform (QFT), quantum arithmetic, quantum approximate optimization algorithm, and Ising model simulation circuits. In comparison with ScanPartitioner, which produces the highest quality results, TDAG produces an equivalent quality of result and an average run time reduction of 94.5%. In comparison with QuickPartitioner, TDAG produces an average quality of result improvement of 45.2% with only an average 4% increase in run time. Furthermore, for larger values of k (up to 16), we observe a time improvement ratio of nearly 100% compared with ScanPartitioner for several circuits, demonstrating the superior time complexity of TDAG. Performance on hardware mapped circuits (a 20 qubit QFT circuit) was also measured, again showing a near 100% improvement in run time against ScanPartitioner and up to 69% improvement in quality against QuickPartitioner.

This paper is organized as follows: Section II describes the behavior of TDAG. Section III discusses how the partitioners were evaluated along with results. Section IV discusses future research direction and concludes the article.

II. Proposed Method: Tree-Based Direct Acyclic Graph (TDAG) Partitioning

TDAG attempts to decompose a circuit of n qubits with g gates into blocks at most k qubits containing as many multi-qubit gates as possible. It follows these steps: 1: TDAG begins by enumerating all qubit groups which could be included together in a partition. 2: A candidate partition is then created for each group of qubits by greedily collecting all gates which include only the grouped qubits, starting from the beginning of the circuit. 3: These candidate partitions are scored based on multi-qubit gate count, and the best partition is removed from the active region and transformed into a partition. This process is repeated until the active region is empty. All of these steps are self-explanatory with the exception of the qubit group enumeration algorithm (step 1), which we now explain in detail.

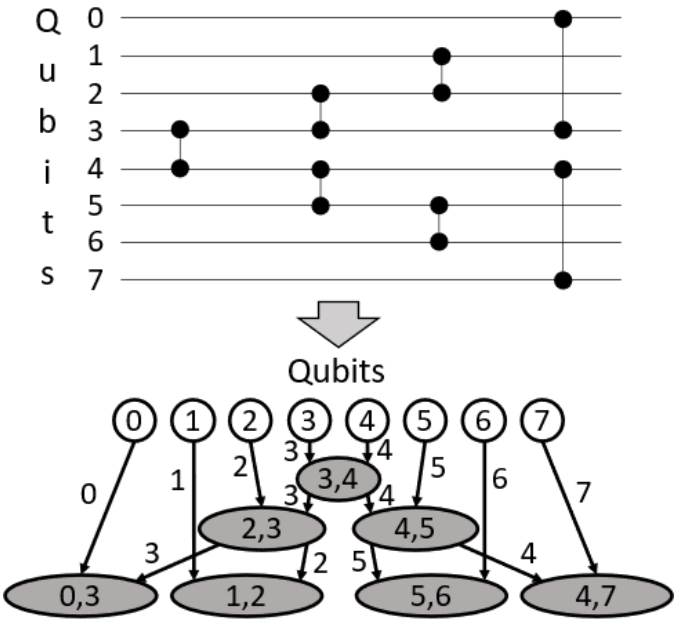


Fig. 1: Converting a Quantum Circuit into a Direct Acyclic Graph

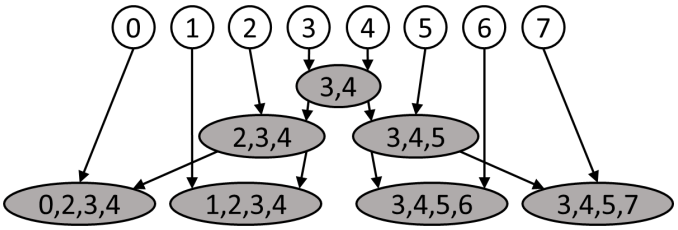


Fig. 2: Gate Dependency Calculation

The group enumeration algorithm operates on the Directed Acyclic Graph (DAG) representation of the circuit. Figure 1 demonstrates the conversion of a quantum circuit into a DAG representation. The nodes at the top of the DAG and all edges represent qubits, while the remaining nodes each represent a generic two-qubit quantum gate. The DAG is constructed by iterating along each qubit in the circuit and creating a chain of nodes representing the gates applied to the qubit. In the case that a gate has already been encountered, an edge connecting to the existing node for that gate is created instead. The group enumeration algorithm is composed of two steps. The first step calculates the qubit dependencies of each gate up to k dependencies, while the second step finds all possible groups of interacting qubits up to size k by viewing the circuit as a series of binary trees.

A. Gate Dependencies

The gate dependency calculation performs a modified breadth-first search over the DAG of the circuit starting at each qubit node. The nodes in the search queue are visited in topological order. When a node is visited, the dependencies of the node are merged into the dependencies of each child node. The behavior of the algorithm on the example circuit from Figure 1 is shown in Figure 2. In the first step, each qubit node is visited and merged into the list of dependencies for the first gate along the qubit. Gates are added to the visitation queue when they are encountered. The first gate to be visited is the gate between qubits 3 and 4, which has dependencies $\{3, 4\}$. These dependencies are copied to the gate between 2 and 3 and the gate between 4 and 5. These gates are visited next and their dependencies are similarly copied into each of their children. Because all gates remaining in the queue are dependent on at least k qubits, the algorithm then terminates.

B. Qubit Group Calculation

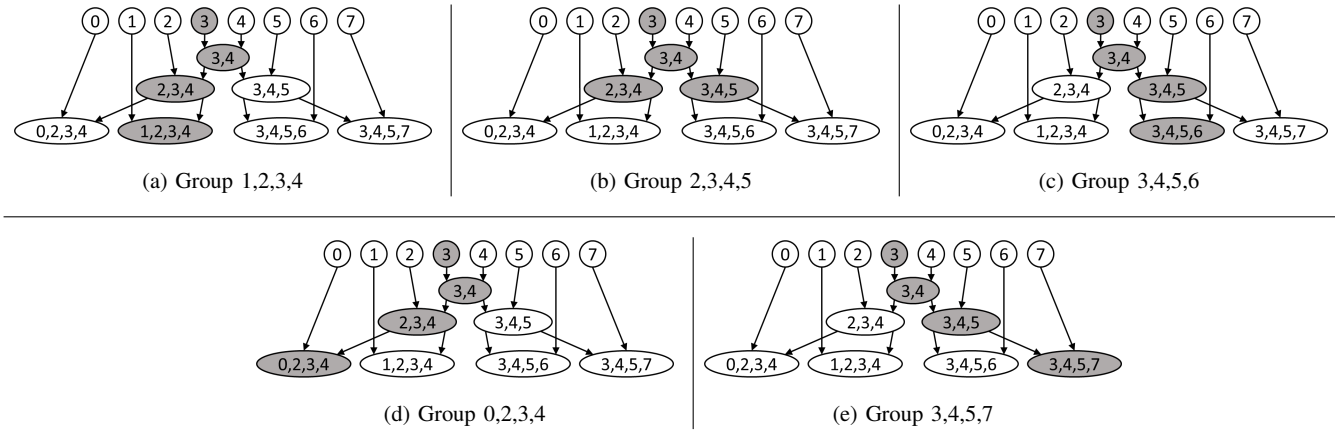
The qubit group calculation algorithm finds qubit groups by accumulating qubits while traversing all possible binary trees with a given root node. Figure 3 shows all possible trees rooted at qubit 3 with 4 nodes. Four of the trees (a, c, d, and e) consist of a simple route down the graph until a total of four qubits are encountered. The fifth tree (b) reveals a more complicated pattern which consists of movement down one side of the graph, then backtracking and moving down the other side of the graph. The qubit group for each path is calculated as the union of the dependencies for all gates included in the path.

Algorithm 1 Qubit Group Enumeration for Proposed Method

```
1:  function EnumerateGroups(circuit, depend)
2:      result ← ∅
3:      for all qubit ∈ circuit do
4:          results.add(Enumerate(qubit, set{qubit}, depend))
5:      end for
6:      return results
7:  end function
8:
9:  function Enumerate(target, input, depend)
10:     output ← ∅
11:     result ← ∅
12:     for all group ∈ input do
13:         if |group ∪ depend[target]| < k then
14:             result.add(group ∪ depend[target])
15:             if depend[target] ⊈ group then
16:                 output.add(group ∪ depend[target])
17:             end if
18:         end if
19:     end for
20:     while any |group| < k for all group ∈ output do
21:         target ← target.next()
22:         input ← output
23:         output ← Enumerate(target.otherqubit, input)
24:         result.add(output)
25:     end while
26:     return result
27: end function
```

Algorithm 1 shows the qubit group calculation algorithm. The algorithm receives the circuit to be partitioned and the gate dependencies as inputs, and returns the set of possible qubit groups. The circuit paths are enumerated by recursively exploring the circuit in an order similar to a depth first search. The recursive function (line 9) receives a target gate and qubit, and a set of incomplete qubit groups. The first loop in the function appends the dependent qubits for the target gate to each group with less than k qubits. The new groups are added to both a result set and the set of inputs to the next recursive call. In order to prevent redundant branching, only groups which are updated with new qubits in the previous step (evaluated on line 15) are passed to the next call. If a new group contains more than k qubits, it is discarded.

The algorithm then finds the next gate along the target qubit and performs a recursive call targeting the new gate and the other qubit the gate interacts with, alongside the relevant partial groups (lines 21



(a) Group 1,2,3,4 (b) Group 2,3,4,5 (c) Group 3,4,5,6

(d) Group 0,2,3,4 (e) Group 3,4,5,7

Fig. 3: Possible Trees Rooted Qubit 3

to 23). The return value from this call provides the partial groups for the next recursive call. Each returned set is also merged into the result set (line 24). Recursive calls along the target qubit are made until the return set is empty (line 20), at which point the function returns the accumulated result set. At the end of the call, the qubit groups for all possible binary trees rooted at the target gate have been accumulated in the result set. An initial recursive call is performed on the first gate along each qubit (line 4). This allows the algorithm to explore the circuit from all possible starting points. It is important to note that this algorithm is guaranteed not to modify the behavior of the circuit, because the order in which the gates are applied is not disturbed by the partitioning process.

III. Results

TDAG, ScanPartitioner, and QuickPartitioner were each applied to a set of benchmark circuits, measuring run time and number of partitions produced for each run. Another partitioner has been proposed by Clark et al. [12] which shows good performance compared to both of the partitioners from BQSKit. This partitioner is included in the first part of the analysis, but excluded from the latter part for clarity due to the similarity to ScanPartitioner. The CNOT gate count, qubit count, and a brief description of each circuit is provided in Table I. The analysis was performed at $k = 4$ and 5, both because 4 and 5 are accepted values for this parameter and also to provide a direct comparison with Clark et al. Each test was performed 10 times and the results were averaged, with the exception of the more time-consuming Quantum Fourier Transform passes on ScanPartitioner. It should be noted that all four algorithms are deterministic, and thus produce produce a constant number of partitions given a fixed set of parameters. All tests were performed using a computer with an AMD Ryzen 5 5600X processor and 32GB of RAM.

Unlike ScanPartitioner and QuickPartitioner, TDAG does not group together qubits which cannot interact in the active portion of the circuit, which means that it will occasionally produce partitions with fewer average gates per block. In most cases, the resulting blocks can be processed to obtain results similar to the other partitioning methods by applying a simple merging algorithm which groups adjacent blocks that contain no more than k qubits when combined. This approach was applied to the quality measurements for all partitioners to allow them to be accurately compared.

Table II shows the data for the four partitioners on all benchmark circuits, while Table III provides a summary of the data. The results demonstrate that TDAG produces very similar quality results to ScanPartitioner and Clark et al., while producing a 45.2% quality improvement over QuickPartitioner. TDAG is also shown to have a run time competitive with QuickPartitioner, ScanPartitioner, and Clark et al. across the benchmark circuits. A significant difference in the run time of ScanPartitioner and the other methods is observed for the 20 qubit QFT circuit, where ScanPartitioner takes significantly longer than the other methods. As a result, TDAG shows an average time improvement ratio of 94.5% against ScanPartitioner. TDAG also achieves a 61.4% time improvement compared to Clark et al. [12].

Despite the large difference in average time taken, all three methods show similar run time for most circuits at $k = 4$ and 5. In fact, ScanPartitioner is faster than both QuickPartitioner and TDAG in a few cases, but this is primarily due to the relative simplicity of the method. The high time complexity of ScanPartitioner becomes evident in the larger, more complex circuits. Notably, the 20 qubit QFT circuit shows clear exponential growth in execution time for ScanPartitioner, while QuickPartitioner and TDAG appear constant. Additionally, TDAG produced a higher quality result compared with QuickPartitioner for all circuits except the 5 qubit QFT circuit, where it was the same, and was within 1 partition of ScanPartitioner for all circuits except the 20 qubit Quantum Fourier Transform circuit, where it was two partitions ahead.

A. Performance as k Varies

In addition to the analysis across all circuits at $k = 4$ and 5, a more detailed analysis was performed on four important quantum circuits (multiply_10, qaoa_10, qft_20, and TFIM_32) for values of k up to 16. These circuits were partitioned using QuickPartitioner, ScanPartitioner, and TDAG. The execution time and number of partitions were measured for each partitioning and plotted.

The results are shown in Table 4. The wider range of k values provided by this table provides a better indication of the general performance of the three partitioners. For example, QuickPartitioner seems to be constant in k, with the time for most circuits showing no trend when k is varied. ScanPartitioner demonstrates an exponential increase for all four circuits, with the trend for the 10 qubit multiplier and 32 qubit TFIM circuit possibly appearing greater than exponential. TDAG appears to be constant in k for most test circuits, although some exponential growth is exhibited in the TFIM circuit. TDAG and ScanPartitioner consistently outperform QuickPartitioner in terms of quality, with QuickPartitioner producing three times as many partitions as the other two in the TFIM circuit.

B. Performance on Hardware-Mapped Circuits

In addition to analyzing the performance of the algorithms on the original set of benchmark circuits, we also tested the algorithms on

TABLE I: Benchmark Circuits Used in This Work

Circuit	Description	CNOT Count	Qubit Count
adder_9	Quantum adder	98	9
heisenberg_8	50 step Heisenberg model simulation	2100	8
hlf_10	Hidden linear function circuit	56	10
multiply_10	Quantum multiplier	163	10
qaoa_10	Quantum approximate optimization algorithm	85	10
qft_5	Quantum Fourier transform circuit	33	5
qft_10	Quantum Fourier transform circuit	216	10
qft_20	Quantum Fourier transform circuit	380	20
TFIM_8	100 step transverse-field Ising model simulation	56	8
TFIM_16	100 step transverse-field Ising model simulation	240	16
TFIM_32	100 step transverse-field Ising model simulation	992	32
wstate_27	W-state preparation circuit	52	27
qft_20_grid	qft_20 mapped with grid connectivity	487	20
qft_20_tokyo	qft_20 mapped onto the IBM Tokyo	484	20

TABLE II: Benchmarks of Partitioning Methods for k at 4 and 5

		Quick [11]		Scan [5]		Clark et al. [12]		TDAG (Proposed)	
Circuit	k	Time (s)	Partitions	Time (s)	Partitions	Time (s)	Partitions	Time (s)	Partitions
adder_9	4	0.04	15	0.04	7	0.07	7	0.04	7
adder_9	5	0.03	7	0.06	6	0.08	6	0.04	6
heisenberg_8	4	1.18	349	0.99	225	2.19	225	1.15	225
heisenberg_8	5	1.24	294	1.00	150	2.75	150	1.25	150
hlf_10	4	0.02	15	0.03	8	0.04	8	0.02	8
hlf_10	5	0.02	10	0.05	5	0.04	5	0.02	6
multiply_10	4	0.06	19	0.10	15	0.13	15	0.06	15
multiply_10	5	0.05	11	0.12	8	0.14	8	0.06	8
qaoa_10	4	0.03	16	0.05	9	0.06	9	0.03	9
qaoa_10	5	0.03	9	0.08	6	0.07	6	0.03	6
qft_5	4	0.01	3	0.01	3	0.02	3	0.01	3
qft_5	5	0.01	1	0.01	1	0.02	1	0.01	1
qft_10	4	0.08	27	0.12	18	0.17	18	0.08	18
qft_10	5	0.07	17	0.16	12	0.20	12	0.10	12
qft_20	4	0.21	71	30.82	45	0.52	45	0.23	45
qft_20	5	0.21	51	152.29	33	0.63	32	0.24	31
TFIM_8	4	0.05	8	0.05	7	0.09	7	0.05	7
TFIM_8	5	0.05	5	0.05	5	0.10	5	0.06	5
TFIM_16	4	0.22	44	0.22	31	0.51	32	0.23	31
TFIM_16	5	0.25	35	0.23	22	0.58	22	0.26	22
TFIM_32	4	0.96	184	0.90	125	3.49	126	1.02	124
TFIM_32	5	1.00	134	0.92	86	3.80	87	1.04	85
wstate_27	4	0.03	19	0.04	13	0.11	26	0.06	13
wstate_27	5	0.03	14	0.04	9	0.10	18	0.06	9

TABLE III: Performance Improvement of TDAG Compared with Existing Works

		Partitions	Time (s)	
	k	TDAG w.r.t. Quick [11]	TDAG w.r.t. Scan [5]	TDAG w.r.t. Clark et al. [12]
Improvement Ratio	4	34.42%	91.05%	59.68%
	5	42.01%	97.96%	62.95%
Average Improvement Ratio		**45.21%**	**94.52%**	**61.37%**

the 20 qubit QFT circuit mapped to existing quantum hardware. We considered two connectivity types: a grid-shaped coupling graph, and the coupling graph representing the IBM Tokyo quantum computer, which is a grid with some diagonal connections inserted. The results are shown in Figure 5.

The table indicates that mapping a fully connected circuit to a grid architecture reduces, but does not eliminate, the exponential scaling exhibited by ScanPartitioner. Although ScanPartitioner performs better than in the unmapped case, TDAG still shows an average 99.9% performance improvement. Additionally, as before, TDAG matches the result quality of ScanPartitioner and runs in similar time to QuickPartitioner, with an 80% improvement in result quality compared to QuickPartitioner.

C. Complexity

The efficiency of TDAG is explained by the dramatic reduction in worst-case number of qubit groups returned by the group enumeration method. The process of finding circuit paths is equivalent to enumerating all binary trees of k nodes. This process can, in turn, be represented in terms of a Dyck language. A Dyck language is a collection of balanced strings composed of a pair of symbols, such that the number of occurrences of the second symbol never exceeds the first and the total number of occurrences of each symbol is equal. The Dyck language representing possible paths through the circuit is composed of the symbols F and B. The F symbol moves forward along the current qubit and then branches to the other qubit in the next gate. The B symbol moves back to the previous qubit. The F operation used by the enumeration algorithm skips gates until a gate with at least one new qubit is found. This means that at most $k-1$ pairs of symbols will be present in the resulting Dyck words. The total number of unique Dyck words of length $2(k-1)$ is equal to the $k-1$th Catalan number [13]. The growth rate of the Catalan numbers is bounded above by 4^k. Thus, the total number of groups of length k returned by starting from a single qubit in the

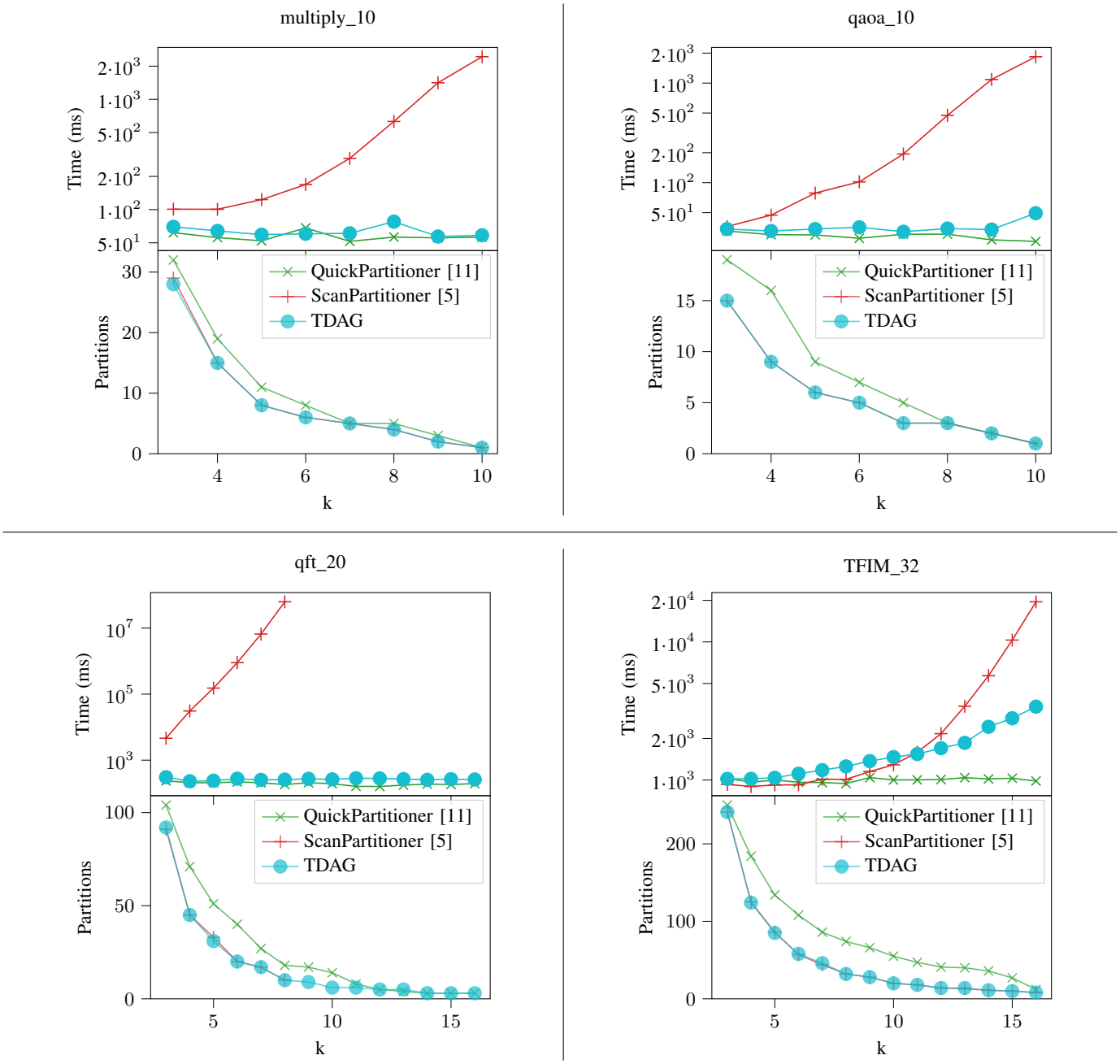


Fig. 4: Performance for Partitioning Methods Across Select Benchmarks

enumeration algorithm is $O(4^k)$ worst-case. Because this process is repeated for each qubit in the circuit, the worst-case number of groups returned by the enumeration algorithm is $O(n4^k)$. As each partition contains at worst one gate, and this operation is performed after each partition, the total time complexity is $O(gn4^k)$. This is smaller than the $O(gn^k)$ of ScanParitioner, particularly for large values of n or k.

It is important to note that the complexity of ScanPartitioner and TDAG are strongly affected by the topology of their respective graphs. Thus, for ScanPartitioner, time complexity is dependent on the degree of the qubit coupling graph. For example, grid-shaped (HLF, multiplier, and QAOA circuits) and fully-connected (QFT circuits) graphs have a worst-case group count of approximately $O(n4^k)$ and $O(n^k)$, respectively. This produces the exponential time complexity with increasing k seen in the benchmarks. An interesting case is the linearly connected circuits (TFIM and W-state circuits), which produce exponential complexity despite the $O(nk)$ worst-case group count for linearly connected circuits. This is caused by ScanPartitioner's implementation of the group enumeration algorithm, which searches the graph more than necessary in this case.

The time complexity of TDAG depends on the local connectivity of the circuit, specifically the beginning of the active region of the circuit. When the circuit is strongly interconnected, the dependency list for each successive gate grows exponentially, reducing the number of nodes in the search tree to $O(lg(k))$. This reduces the number of possible groups to $O(nk^2)$. Conversely, the worst-case behavior $O(n4^k)$ is produced if the dependency list for each successive gate grows by only one qubit.

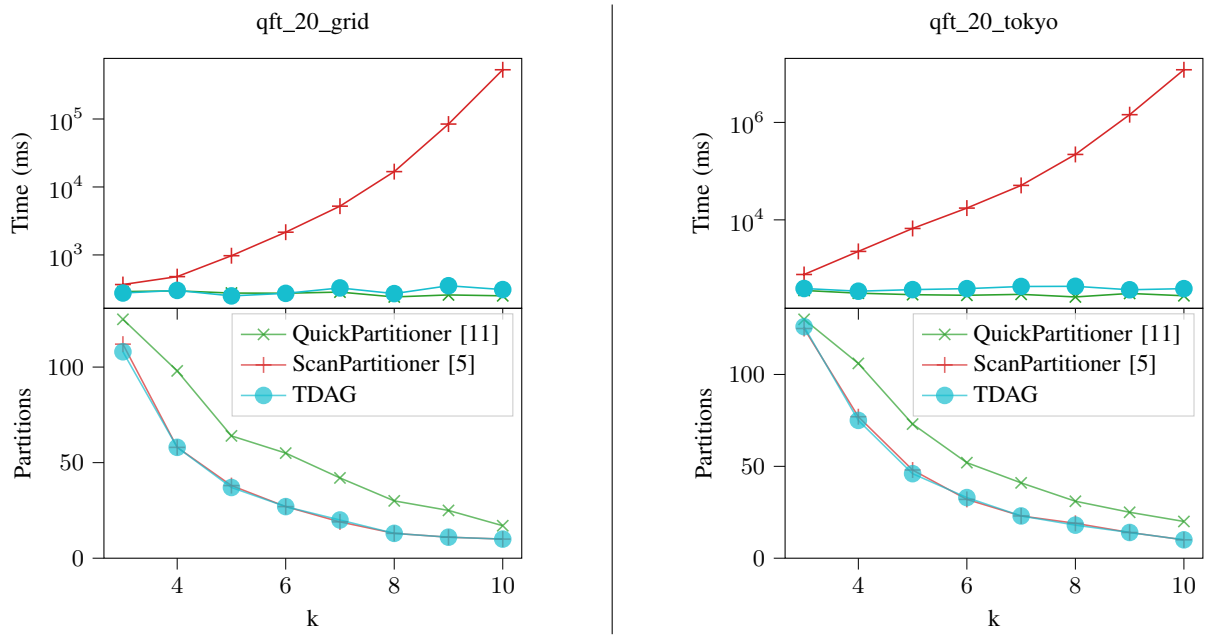


Fig. 5: Performance of Partitioning Methods on Mapped Circuits

IV. Conclusion

Applying circuit partitioning to synthesis-based optimization provides a promising avenue to address the exponential scaling problems which normally accompany quantum synthesis. Partition quality significantly affects the execution time and quality of the optimized circuit. TDAG outperforms existing partitioning methods by applying more efficient partitioning techniques, achieving a result quality equal to exhaustive methods and execution time similar to faster, simpler methods. In a benchmark test against a fast method and an exhaustive solution, TDAG shows a 45% quality improvement against the fast method, and almost identical quality to the exhaustive method. The execution time of TDAG scales significantly better than the exhaustive method with an 95% average time improvement, and is faster than the fast method in several cases.

Future improvements for TDAG should address the fact that it does not group together qubits which can not interact in the active part of the circuit. For very shallow circuits, the reprocessing technique discussed in the results section may not work because TDAG fails to find sufficiently large qubit groups due to reduced tree depth. This problem has proven difficult to address for TDAG, but we believe it could be resolved by slightly modifying the qubit enumeration algorithm while preserving the positive characteristics of TDAG.

References

[1] W. Tang, T. Tomesh, M. Suchara, J. Larson, and M. Martonosi, "Cutqc: Using small quantum computers for large quantum circuit evaluations," in *Proceedings of the 26th ACM International conference on architectural support for programming languages and operating systems*, 2021, pp. 473–486.

[2] Z. Davarzani, M. Zomorodi-Moghadam, M. Houshmand, and M. Nouri-baygi, "A dynamic programming approach for distributing quantum circuits by bipartite graphs," *Quantum Information Processing*, vol. 19, no. 10, pp. 1–18, 2020.

[3] P. Andres-Martinez and C. Heunen, "Automated distribution of quantum circuits via hypergraph partitioning," *Physical Review A*, vol. 100, no. 3, p. 032 308, 2019.

[4] O. Daei, K. Navi, and M. Zomorodi-Moghadam, "Optimized quantum circuit partitioning," *International Journal of Theoretical Physics*, vol. 59, no. 12, pp. 3804–3820, 2020.

[5] X.-C. Wu, M. G. Davis, F. T. Chong, and C. Iancu, *Qgo: Scalable quantum circuit optimization using automated synthesis*, 2020. DOI: 10.48550/ARXIV.2012.09835. [Online]. Available: https://arxiv.org/abs/2012.09835.

[6] T. Patel, E. Younis, C. Iancu, W. de Jong, and D. Tiwari, *Robust and resource-efficient quantum circuit approximation*, 2021. DOI: 10.48550/ARXIV.2108.12714. [Online]. Available: https://arxiv.org/abs/2108.12714.

[7] E. Nikahd, N. Mohammadzadeh, M. Sedighi, and M. S. Zamani, "Automated window-based partitioning of quantum circuits," *Physica Scripta*, vol. 96, no. 3, p. 035 102, 2021.

[8] J. M. Baker, C. Duckering, A. Hoover, and F. T. Chong, "Time-sliced quantum circuit partitioning for modular architectures," in *Proceedings of the 17th ACM International Conference on Computing Frontiers*, 2020, pp. 98–107.

[9] B. Fang, M. Y. Özkaval, A. Li, Ü. V. Catalyürek, and S. Krishnamoorthy, "Efficient hierarchical state vector simulation of quantum circuits via acyclic graph partitioning," in *2022 IEEE International Conference on Cluster Computing (CLUSTER)*, IEEE, 2022, pp. 289–300.

[10] D. Dadkhah, M. Zomorodi, S. E. Hosseini, P. Plawiak, and X. Zhou, "Reordering and partitioning of distributed quantum circuits," *IEEE Access*, vol. 10, pp. 70 329–70 341, 2022.

[11] B. N. Laboratory, *Bqskit*, https://github.com/BQSKit/bqskit, 2022.

[12] J. Clark, H. Thapliyal, and T. S. Humble, "A novel approach to quantum circuit partitioning," in *2022 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, IEEE, 2022, pp. 450–451.

[13] E. Barcucci and M. C. Verri, "Some more properties of catalan numbers," *Discrete Mathematics*, vol. 102, no. 3, pp. 229–237, 1992.