POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Project report

Optimizing SWAP Gate Usage in Dependent Quantum Circuits using a Topological Approach

Course of Code Transformation and Optimization

Authors: Daniel Comotti, Giorgio Miani, Francesco Micucci

Advisor: Prof. Agosta Giovanni

Academic year: 2023-2024

1. Introduction

Quantum computing (QC), a field that originated in the 1980s, has only recently gained significant traction as advancements in quantum technology accelerate. Despite operating in the current NISQ (Noisy Intermediate-Scale Quantum) era—where quantum computers are limited in qubit capacity and affected by noise—quantum research continues to thrive due to QC's potential to solve complex problems that classical computers struggle with.

Our work specifically addresses gate-based quantum computing, where qubits are controlled by applying quantum gates. The gates define the behavior of a quantum circuit, enabling it to perform specific computational tasks. A significant constraint, however, is that quantum gates can only be applied to physically adjacent qubits. To overcome this limitation, SWAP gates are frequently used to reposition distant qubits to make them adjacent. Unfortunately, the repeated use of these SWAP gate sequences can introduce computational noise, especially in circuits composed of several dependent modules. This challenge forms the basis of our project. In complex quantum circuits comprising multiple dependent modules, it is often essential to transfer output qubits from one module to another for subsequent processing. To address this, we aim to create an efficient qubit-mapping strategy that reduces the reliance on SWAP gates.

2. Project Groundwork

Our research builds upon previous studies that investigated methods for mapping multiple independent quantum circuits onto a single quantum computer topology, trying to enhance throughput while preserving output quality [1]. This research focuses on a hardware system H equipped with m qubits and a collection of quantum circuits represented as $C = \{C_1, C_2, \ldots, C_N\}$. The objective is to allocate groups of up to $k \leq N$ circuits to the hardware simultaneously, adhering to the following criteria: (i) the total number of qubits required for each group must not exceed m; (ii) circuits within each group must utilize distinct qubits; and (iii) the noise level for each circuit's configuration must remain within ε of its optimal noise threshold.

To facilitate this, a compatibility graph is built, where each vertex corresponds to a specific circuit and its associated layout (i.e., its placement on the hardware). An edge between two vertices indicates that the corresponding circuits are distinct and have layouts separated by at least b qubits in the topology. The edges have weights based on the mapomatic score, which evaluates how closely a layout corresponds to a circuit's optimal configuration. By identifying the maximum clique within this compatibility graph, we can determine which circuits can be executed concurrently.

Our goal is to extend this approach to scenarios involving dependent circuits and to shift the primary mapping metric from noise to the number of SWAP gates required. More details regarding this extension can be found in Section 3.

3. Implementation

We examine a general random quantum circuit composed of N interconnected components, referred to as modules. The relationships between these modules are represented by a structure known as the dependency graph. These modules need to be mapped onto a quantum computer with a regular grid topology of m qubits.

In the following sections, we will provide a comprehensive overview of the entire workflow. We will start by discussing elements closely related to defining problem requirements, such as circuit generation (including the dependency graph) and backend generation. After this foundational exploration, we will focus on more problem-related aspects, including module scheduling to create the set $C_t = \{C_{1t}, C_{2t}, \dots, C_{kt}\}$ of quantum circuits that can be mapped at time step t, the development of the compatibility graph, and the identification of the maximal clique.

3.1. Circuit generation

To design circuits, we developed a class that randomly generates them based on three key parameters: the total number of modules, the maximum number of qubits that each module can utilize, and the maximum number of gates that can be incorporated into each module.
To manage dependencies effectively, we leverage the **NetworkX** library to create a dependency graph structured as a random tree. We maintain a list called *current_nodes*, which contains modules ready for creation—meaning all their dependencies have already been generated.
At each step, we select the first module from the list and generate it using a specific set of gates. To maintain connectivity within the modules, we carefully choose the gates for each new module to avoid creating isolated sections within them.
When a module is created, each qubit is assigned an identification value. If a qubit connects to an existing module, referred to as a dependency, it receives an identification value that matches one from that module. Any newly added qubits are given new identification values. This system allows us to track which modules share qubits and how these qubits are used throughout the circuit. When multiple modules utilize the same qubit, they will have at least one qubit with a matching identification value, facilitating the monitoring of qubit usage across the entire circuit.

3.2. Backend generation

We developed a new simulated backend because none of the existing options offered a structure resembling a regular grid. The available backends had sparse connection patterns instead. Our implementation accepts the desired number of rows and columns for the topology as input and generates a backend where each qubit connects to the qubit on its right (except for those at the end of each row), to the qubit below it (except for those at the bottom of each column), to the qubit on its left (except for those at the start of each row), and to the qubit above it (except for those at the top of each column).

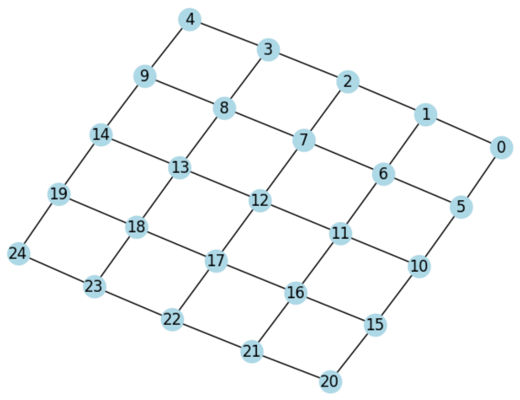


Figure 1: Visual representation of a 5x5 qubit grid, showcasing its connection pattern.

3.3. Module Scheduling

Regarding our implementation, we opted for an As Soon As Possible (ASAP) scheduling approach to determine which modules should be mapped.

As outlined in Section 3, we use a dependency graph to represent dependencies between modules, allowing us to identify the set C_t of modules available for mapping at each timestep t.

Since the dependency graph is structured as a tree, the nodes eligible for mapping at any timestep are those without incoming edges, except for edges originating from modules mapped in the previous timestep. Thus, at the initial timestep, nodes without incoming edges are mapped first. In the following iterations, nodes with incoming edges from previously mapped modules become available, and this continues iteratively.

This schedule is initially established and then adjusted whenever some modules from C_t remain unmapped. During rescheduling, the next nodes available for mapping will be those with dependencies fulfilled by the already mapped modules.

An example of ASAP scheduling is illustrated in Figure 2.

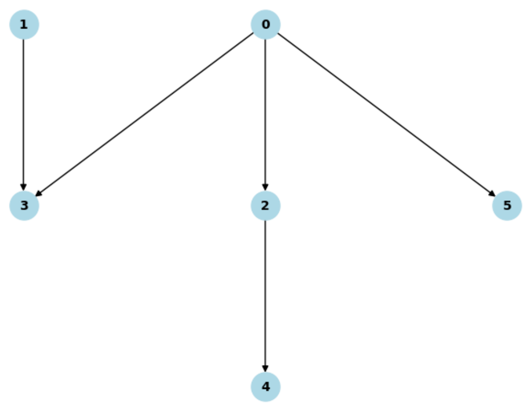


Figure 2: In this dependency graph, the initial ASAP scheduling maps modules 0 and 1 first, followed by modules 2, 3, and 5, and finally, module 4.

3.4. Compatibility Graph generation

The compatibility graph used in our algorithm is structured similarly to the one introduced in Section 2. In this graph, each vertex represents a module and one of its possible layouts (a specific placement on the hardware) among those available on the topology. Two vertices are connected by an edge if they meet the following constraints:

- The modules corresponding to the two vertices are distinct.
- The layouts of the vertices do not overlap on the topology.

The edge weight is determined by two components: *Common Dependence Distance* and *Input SWAPs Distance*.

The *Common Dependence Distance* measures the physical distance on the topology between the output qubits of the two layouts when they share a dependency (i.e. when the output qubits are used by a module in a subsequent timestep). This distance is zero if the modules are not connected to the same module in the dependency graph. The purpose of this weight component is to prioritize the proximity of modules with a shared dependency, reducing the number of SWAP operations required to transfer the output qubits to the input qubits of subsequent modules.

In contrast, the *Input SWAPs Distance* is calculated as the physical distance between the input qubits of the modules represented by the two vertices connected and the output qubits of their preceding dependent module. This distance guides the algorithm to prioritize layouts that minimize the SWAP operations to route inputs to the edge modules.

To improve efficiency when working with the compatibility graph, we imposed an additional constraint. If the combined weight of the two components exceeds a certain threshold, we do not insert the edge into the graph.

After calculating all the edge weights, we apply the following transformation to each weight:

$$w'_{i,j} = w_{max} - w_{i,j} \tag{1}$$

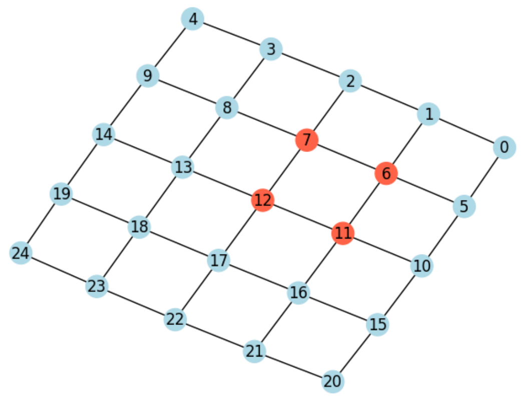
Here $w'_{i,j}$ is the new edge weight, $w_{i,j}$ is the previously calculated weight, and w_{max} is the highest weight among those computed for building the compatibility graph. This transformation ensures that identifying the maximum clique in the compatibility graph produces the optimal qubit mapping for each module at a certain timestep.

For a more detailed discussion on optimizing layout searches, see Section 3.6.

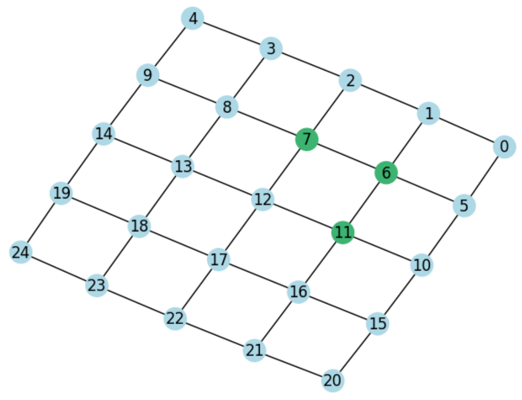
3.5. Maximum Clique Search

Finding the maximum clique on the compatibility graph is a critical step. It enables us to determine optimal layouts for the modules at each timestep, which helps reduce the number of SWAP gates required during circuit execution.
To identify the maximum clique, we used functions available in the **networkx** library and pursued two approaches: one heuristic and one exact.
The exact approach uses the `find_cliques` function, based on the Bron and Kerbosch algorithm (1973) [3], which aims to identify all possible cliques in the compatibility graph. To select the optimal layout, we first consider the sum of edge weights, choosing the layout with the highest total value. If multiple layouts have the same weight sum, we select the layout that maximizes the number of qubits used on the topology.
Alternatively, the heuristic approach employs the `max_clique` function, based on the algorithm by Boppana and Halldórsson (1992) [2].

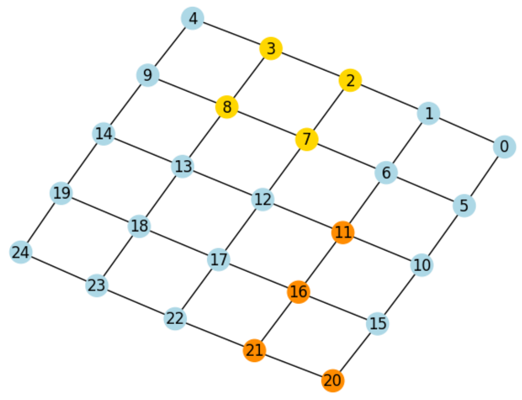
An example of the identified layouts, after determining the maximum clique in a simple case, is presented in Figure 3.



(a) First timestep



(b) Second timestep



(c) Third timestep

Figure 3: Layouts obtained by identifying the maximum clique in a simple random case involving four modules on a 5x5 regular grid.

3.6. Further Optimizations

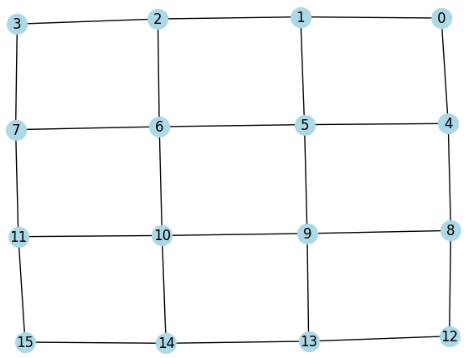
The approach outlined in previous sections proved effective in identifying the optimal layout for each module, but it demanded considerable computation time. This was particularly true for regular topologies with a large number of qubits and for compatibility graphs across a lot of modules. As the number of available qubits increases, so does the number of potential layouts per module, leading to a larger compatibility graph with more nodes. Likewise, having more modules at each timestep increases the graph's size, further extending the computation time.

The first implemented optimization aims to reduce the topology size where we search for all possible layouts. To address this, we introduced the parameter *reduced_distance*, which effec-

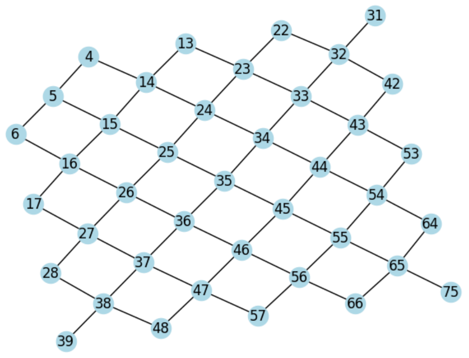
tively restricts the quantum computer's topology to a relevant subset. This parameter is user-defined; however, if not specified, it defaults to the number of qubits in the module under consideration.
An exception applies to the initial set of modules we map, where *reduced_ distance*, if not user-defined, is set to the largest module qubit count at that timestep. Before moving to the other optimizations it is essential to evaluate how *reduced_ distance* influences the topology used for layout searches.
Specifically, for the first set of modules, the layout search is conducted within a regular grid topology of size $reduced\_distance \times reduced\_distance$. For all subsequent module sets, the topology includes all qubits within a distance equal to *reduced_ distance* from the output qubits of the previous steps, which serve as inputs for the current module.
The adjustment made to the *reduced_ distance* for the initial modules in the mapping process is necessary because, without previous dependencies, there is no reference point to guide the topology reduction. Consequently, since we are only searching for a mapping within a smaller topology, we aim to increase the likelihood of fitting all the modules by choosing the largest module qubit count at that timestep.
Figure 4 shows examples of reduced topologies.

The second optimization focused on the initial set of modules for mapping. In the first optimization, we identified all possible layouts on a simplified generic topology with dimensions $reduced\_distance \times reduced\_distance$. Building on these layouts, the second optimization sought to prevent solutions from being too close to the edges of the original topology. To achieve this, we centralized the layouts of the initial module set, moving them away from the boundaries and positioning them toward the center. This adjustment minimizes the likelihood of encountering edge qubits in future expansions while exploring new reduced topologies.

The latest optimization distinguishes between cases that require mapping a single module and those involving multiple modules. When mapping only one module, building a compatibility graph is unnecessary. Instead, the main concern is the total distance between the current layout and outputs from previous timesteps that serve as inputs for the current module. The layout with the smallest distance is chosen as the optimal mapping for that module at this timestep. Consequently, there is no need to construct a compatibility graph or find the maximum clique in this scenario. Additionally, if this single module is the first to be mapped, its layout can be selected randomly, as it has no dependencies on other qubits.



(a) Reduced topology for a module during the initial timestep.



(b) Reduced topology for a module at a subsequent timestep.

Figure 4: Examples of reduced topologies obtained from mapping a random circuit onto a 10x10 regular qubit grid.

4. Results

To assess the performance of our implementation, we conducted tests under various conditions, with the following adjustable parameters:

- *num_ modules*: The number of modules created in the circuit.

- *module_max_qubits*: The maximum number of qubits per module.
- *module_max_gates*: The maximum number of gates per module.
- *reduced_distance*: The distance for the reduced topology, as discussed in Section 3.6.
- *max_allowed_weight*: The maximum allowable weight when summing the *Common Dependence Distance* and *Input SWAPs Distance* components, as described in Section 3.4.
- *num_qubits_x*: The number of qubits along the x-axis of the generated topology.
- *num_qubits_y*: The number of qubits along the y-axis of the generated topology.
- *heuristic*: Set to `True` to enable the heuristic algorithm for the maximum clique, as explained in subsection 3.5.
- *optimization_level*: The optimization level used by Qiskit during module transpiling.

For our experiments, we set the following parameters: each module was limited to a maximum of 4 qubits, and the number of gates per module was capped at 6. We left the reduced distance parameter set to `None` to use the default values, and fixed the maximum allowed weight at 5. The grid topology followed a 100 by 100 layout, with both *num_qubits_x* and *num_qubits_y* set to 100. Additionally, we disabled the heuristic by setting it to False. During the testing process, we adjusted several parameters, experimenting with the number of modules ranging from 4 to 7, and examined Qiskit's optimization levels from 1 to 3.

For each combination of these settings, we conducted approximately 100 experiments with different random circuits.

4.1. SWAP Count

The first metric we analyzed is the number of SWAP gates introduced by our algorithm.

Figure 5 shows the average number of SWAP gates added for each module count, organized by different optimization levels. As the plot demonstrates, our approach maps the circuits with a minimal number of SWAP gates, averaging approximately one SWAP per circuit. Although the number of SWAP gates tends to increase with the number of modules in the circuit, it remains unaffected by the optimization level chosen in the Qiskit transpiler.

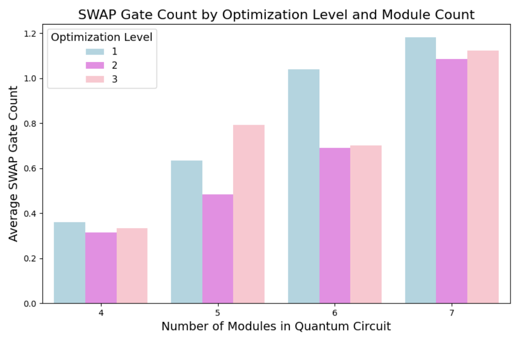


Figure 5: Comparative Analysis of SWAP Gate Count Across Optimization Levels and Quantum Circuit Module Numbers

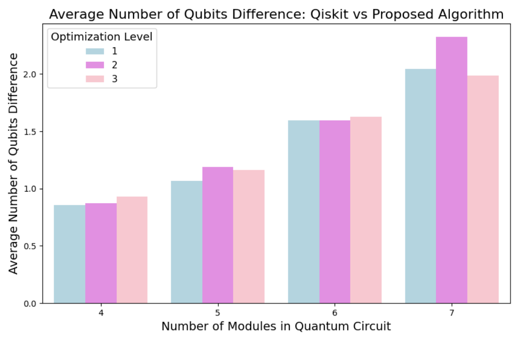


Figure 6: Average difference in the number of qubits used by Qiskit and our approach as the number of modules and optimization level increase.

When comparing our results to those from the standard Qiskit transpiler, we observe that the latter introduces no SWAP gates, largely due to the high connectivity of the topology used compared to current topologies. However, while Qiskit minimizes SWAP gate usage better, this choice has a significant drawback regarding the following metric.

4.2. Total number of qubits

The second metric we examined is the number of qubits used by the compiled circuit. We evaluated this metric for both our approach and the Qiskit compiler. From this analysis, we created a key plot that illustrates the average difference between the number of qubits used by Qiskit and our method. This plot is shown in Figure 6. As the plot demonstrates, Qiskit requires an increasing number of qubits as the number of modules grows. Notably, due to the high connectivity of the topology used in our approach compared to current topologies, we observe no significant change in our metric as the optimization level increases. However, we expect to see a decreasing trend with a more sparse topology as the optimization level rises.

4.3. Additional Metrics

Other metrics assessed include circuit depth, as well as the total number of gates and T gates. Generally, the circuit depth with our approach tends to be higher than that produced by Qiskit. This difference is fully justified, as we use Qiskit for internal module transpiling. Figure 7 clearly demonstrates this trend. However, when it comes to the other two metrics, our method yields results that are not significantly different from those of Qiskit.

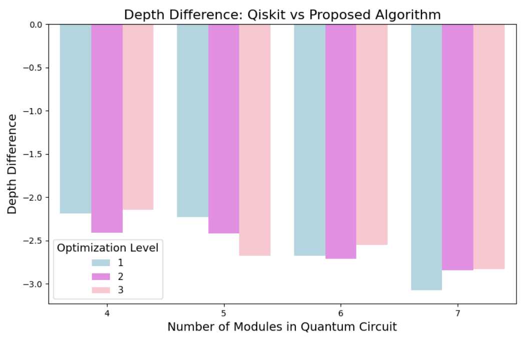


Figure 7: Average difference in circuit depth between Qiskit and our approach as the number of modules and the optimization level increase.

5. Conclusions

In conclusion, the approach implemented in this project proves to be an effective strategy for reducing the number of SWAP gates when mapping multiple dependent modules. It ensures a minimal introduction of SWAP gates while maintaining a lower qubit count compared to Qiskit. The results are comparable to Qiskit in terms of gate counts and T gates, although the circuit depth is slightly higher. This is entirely understandable, given that Qiskit is used for the internal module transpiling process. Future work could explore more efficient methods to accelerate computation, as the maximum clique problem is computationally expensive to solve.

References

[1] Debasmita Bhoumik, Ritajit Majumdar, and Susmita Sur-Kolay. Resource-aware scheduling of multiple quantum circuits on a hardware device, 2024.

[2] Ravi Boppana and Magnús M Halldórsson. Approximating maximum independent sets by excluding subgraphs. *BIT*, 32(2):180–196, June 1992.

[3] Coen Bron and Joep Kerbosch. Algorithm 457: finding all cliques of an undirected graph. *Commun. ACM*, 16(9):575–577, September 1973.