DYNAMIC PROGRAMMING AND CONSTRAINED OPTIMIZATION FOR IMPROVED PARALLEL QUANTUM CIRCUIT EXECUTION

by

AARON ALEXANDER ORENSTEIN

Submitted in partial fulfillment of the requirements for the degree of Master of Science

Department of Computer and Data Sciences

CASE WESTERN RESERVE UNIVERSITY

May, 2024

CASE WESTERN RESERVE UNIVERSITY

SCHOOL OF GRADUATE STUDIES

We hereby approve the thesis/dissertation of

Aaron Alexander Orenstein

candidate for the degree of

Master of Science[1].

Committee Chair

Vipin Chaudhary

Committee Member

Vipin Chaudhary

Committee Member

Shuai Xu

Committee Member

Mehmet Koyutürk

Date of Defense

March 29, 2024

[1]We also certify that written approval has been obtained for any proprietary material contained therein.

DEDICATION

Dedicated to my mom, Catherine, and dad, James, who are the biggest reason I made it here. I wish you could see the fruits of your labor.

TABLE OF CONTENTS

List of Tables vi

List of Figures vii

Preface viii

Acknowledgements ix

List of Abbreviations x

Abstract xi

Chapter I: Introduction 1

1.1 Qubit Mapping 1

1.2 Job Scheduling 3

1.3 Proposed Methods 3

Chapter II: Background 5

2.1 Topology and Gates Errors 5

2.2 Circuit Execution 6

Chapter III: Related Work 7

3.1 Greedy Methods for Mapping 7

3.2 Optimizers for Mapping 7

3.3 Crosstalk 7

3.4 Measurement Crosstalk 8

3.5 Classical Job Scheduling 8

3.6 Quantum Job Scheduling 9

Chapter IV: Qubit Mapping with Binary Integer Nonlinear Programming . . . 10

4.1 Solver 10

4.1.1 Clusterer 10

4.1.2 Placer 18

4.2 Meta-Optimization 22

4.2.1 Equalities > Inequalities 22

4.2.2 Soft Constraints 22
Chapter V: QGroup - Parallel Job Scheduler 23
5.1 Job Grouping with Rod Cutting 26
5.2 Parallel Scheduling with BILP 27
5.2.1 Representation 27
5.2.2 Objectives 29
5.3 Machine Selection 30
5.4 Performance 31
5.4.1 Runtime 31
5.4.2 Computer Groups 32
Chapter VI: Results 34
6.1 Qubit Mapping 34
6.1.1 Runtime 34
6.1.2 Evaluation on IBMQ Machines 37
6.2 Job Scheduling 41
6.2.1 Real Machine Evaluation 43
6.2.2 High Throughput Simulation 45
Chapter VII: Discussion and Conclusion 47
7.1 Qubit Mapping 47
7.2 Job Scheduling 48
Chapter VIII: Questionnaire 49
Bibliography 50

LIST OF TABLES

Number *Page*

5.1 The runtimes of the scheduling algorithms. N is the number of jobs and C is the number of computers. C' is the number of computer types. ω is the runtime of the mapping function used in Algorithms 4,5. Runtime for BILP solvers is not specified in general, so we provide the scaling of the number of variables in the model. 32

6.1 Experiment data comparing circuit cutting vs. parallel execution vs. single execution. Executed on IBM Brisbane. O_{cl} and O_{pl} are shown pre-transpilation (left subcolumn) and post-transpilation (right subcolumn). 39

6.2 Results for QAOA experiments. The top table shows machine information and parameter values for each experiment. The lower table shows experiment results. 40

6.3 RevLib circuits used in the throughput stress test. We used sys6-v0_111 twice. 42

6.4 Results for the density test. The top table shows experiment setup information. The bottom table shows expermient results. 42

6.5 Revlib circuits used in the short queue with the number of shots. . . . 44

6.6 Results for the small queue evaluation. Best values are bolded. For PST, we ignore the serial algorithm because we expect the serial PST to be higher than the parallel case. The same holds for TiIF. 45

6.7 Results for the large queue simulation. 46

LIST OF FIGURES

Number *Page*

4.1 Swap error matrix value between q_0 and q_{113} for varying values of p. q_0 and q_{113} have multiple geodesic paths. Calculations used calibration data for IBM Brisbane on 2024-01-31 13:13:52 16

6.1 Plot of clusterer objective value for different maximum time constraints. 35

6.2 Plot of placer objective value for different maximum time constraints. The placer was initialized with the final assignment produced by the clusterer in Fig. 6.1. 36

6.3 Plot of placer objective value for different maximum clusterer time constraints. We fixed the placer time constraint at 2s. 37

6.4 Input graph for the MaxCut problem targeted by the QAOA ansatz benchmark. 40

6.5 Subtopologies of IBM Brisbane used to generate circuits using QUEKO for the throughput stress test. Used TFL density vectors and depths of 100 and 200. 42

6.6 Execution schedule for each scheduling method. Black horizontal lines separate execution on each computer. The numbering and coloring of jobs is consistent across each chart. Groups are shown by the shaded backgrounds for QGroup. 44

PREFACE

The qubit mapping algorithms discussed in this thesis come from a paper title "Quantum Circuit Mapping Using Binary Integer Nonlinear Programming" accepted for publishing as part of the 2024 IPDPS Workshop on Quantum Computing Algorithms, Systems, and Applications (Q-CASA) [20]. I am first author on this paper and all the ideas and methods described within are my own work.

ACKNOWLEDGEMENTS

A million thanks my friends and family for supporting me through this journey and keeping my life interesting throughout the process. I would like to thank Vinooth Rao Kulkarni, Xinpeng Li, Betis Baheri, and the Kent-Case Quantum Group for help with brainstorming, finding data, and refining my ideas. Additional thanks to Hanrui Wang and Song Han from MIT for providing the queue data used in Section 6.2.2. Special thanks to my advisor, Vipin Chaudhary, for all his guidance with writing, researching, and learning.

This work was supported in part by NSF award CCF-2216923.

LIST OF ABBREVIATIONS

BILP Binary Integer Linear Programming. v, vi, 23, 27, 31, 32

BINLP Binary Integer NonLinear Programming. xi, 3, 39

EPST Estimated PST. 25, 31–33

NISQ Near-term Intermediate-Scale Quantum (Devices). 1, 2, 36, 37, 47

PST Probability of a Successful Trial. vi, x, 32, 43, 45

QAOA Quantum Approximate Optmization Algorithm. vi, vii, xi, 1, 4, 34, 39, 40, 47, 48

TiIF Time Inflation Factor. vi, 43, 45, 46, 48

TiRF Time Reduction Factor. 43, 45, 46, 48

TRF Trial Reduction Factor. 43, 45, 46

Dynamic Programming and Constrained Optimization for Improved Parallel Quantum Circuit Execution

Abstract

by

AARON ALEXANDER ORENSTEIN

As Quantum Computers continue to increase in size, throughput has not increased proportionally [22]. Researchers have begun exploring ways of parallelizing circuit execution [6, 14–19, 21]. Due to the noisiness of quantum computers, this requires new algorithms for efficient resource allocation, qubit mapping, and scheduling. We improve on existing greedy algorithms by formulating the mapping search as a Binary Integer Non-Linear Programming (BINLP) problem. We model practical constraints and propose new heuristics for determining the goodness of a mapping. We observe similar fidelity compared to Qiskit's transpiler for circuit cutting and throughput benchmarks. We observe greater fidelity over Qiskit's transpiler for dense QAOA ansatzes. We find that parallel circuit cutting provides greater fidelity than full-circuit execution. We also propose a scheduling algorithm for parallelizing circuits of different lengths and shot counts. Our algorithm achieves a lower makespan for time and number-of-shots for diverse workloads as well as lower per-job runtimes in all cases.

Chapter 1

INTRODUCTION

Quantum computers provide a new method of computation, which comes with new noise models as well. With Quantum computers, each qubit contributes differently to the overall noise and limitations of the system [10]. Errors can occur in initializing, operating, and measuring qubits and from exceeding the decoherence time of qubits [10]. In the NISQ era, where such errors are significant, it is important to plan the execution of circuits to mitigate noise. As quantum computers continue to increase in size, this planning is crucial to achieve accurate results with larger circuits, even within the bounds of the hardware. Error correction provides a promising avenue for fault-tolerant circuit execution, but qubit demands are too large even as today's machines become larger [11]. Quantum computers are often underutilized, where 100+ qubit machines are used to run small circuits.

Recent research has focused on optimizing execution of multiple circuits simultaneously on the same machine [6, 14–19, 21]. Such work benefits throughput both for the user and machine managers. For users, parallelism promises to run more circuits in fewer jobs, execute shots in parallel, and improve parameter exploration in iterative algorithms like QAOA. For machine managers like IBM, parallelism maximizes machine throughput, allowing more users to be served and reducing queue wait times.

1.1 Qubit Mapping

Parallel circuit optimization provides unique challenges over its serial counterpart. The first challenge is the multiple local topologies. Single circuit mapping benefits from the contiguity heuristic where qubits are assigned to a connected subgraph of the topology. Contiguity reduces the need for SWAP gates and shrinks the search

space when finding the optimal mapping. In the parallel case, we must separate the local topologies of each circuit and choose multiple connected subgraphs in an optimal configuration.

The second challenge is gate crosstalk. When gates are applied between qubit pairs, they induce noise on the states of all neighboring qubits. Serial mappings do not seek to avoid this as separating qubits to mitigate crosstalk incurs heavy SWAP costs when those qubits must be operated on. In the parallel case, qubit usage is independent across circuits so we can and must mitigate crosstalk by separating the qubit clusters used for different circuits. IBM has recently announced near-total mitigation of crosstalk errors for new systems, which may remove this constraint in the future [8]. Crosstalk continues to be a challenge for current NISQ systems.

Finally, quantum computers often contain non-operable qubits or nodes/edges with high error rates. Given this, running multiple smaller circuits provides an advantage over running a single large circuit as the smaller circuits can be mapped to avoid areas of low fidelity. However, this advantage is only realized if mapping algorithms are able to find a placement around these areas. Since finding the optimal placement is NP-Hard, approaches find solutions by limiting the search space. Search-space reduction must be performed carefully as to not remove viable placements [23].

Current approaches utilize greedy methods to find a good mapping [6, 14, 15, 17–19, 21]. Greedy methods have good runtime, often scaling linearly or quadratically with the number of circuits, qubits, and gates. However, each circuit and qubit is assigned with little regard for the needs of future circuits. Given the unevenness of errors and topologies, this can lead to cases where initial circuit placements split the remaining qubits so that future circuit placements must choose low fidelity qubits. By optimizing placement for all circuits simultaneously, we search a greater portion of the search space and are able to reduce the fidelity of initial assignments in return for a greater increase to the fidelity of future assignments.

1.2 Job Scheduling

Parallelism is also needed to improve throughput when processing jobs from different sources. Providers such as IBM provide access to multiple computers through job queue. Jobs are processed one at a time, regardless of their circuits' resource usage. This is particularly sub-optimal for small circuits. Due to the high errors on current devices, benchmarks and circuits are small (<20 qubits) relative to the size of the computers (100+ qubits). For example, the largest circuit width used in [6, 14, 15, 17, 21] is 16 qubits. Thus there are a large number of qubits sitting idle at each execution.

We look to classical scheduling for inspiration. The field of classical scheduling has produced numerous algorithms for scheduling jobs on machines of different or similar properties [5, 11]. However, our use case is more similar to the problem scheduling multiple jobs on one machine [25, 27]. Additional constraints regarding qubits, measurement synchronicity, and runtime similarity do not have analogies in existing classical solutions.

Designing a parallel scheduling algorithm for quantum computing requires careful consideration of the makespan-fidelity tradeoff. Increased parallelism means quicker job completion but lower circuit fidelity. The only prior proposal for such an algorithm is QuCloud+ [14] which uses a simple greedy method to incrementally pair circuits. This method does not consider difference in job shot specifications and only partially considers circuit runtime differences.

1.3 Proposed Methods

We propose a new method for parallel circuit mapping using BINLP to optimize qubit placement for all circuits simultaneously. Our method uses the Gurobi optimizer [10] to traverse the search space with state-of-the-art optimization techniques. We represent the problem in a matrix-vector format and present new heuristics for

modelling circuit error rates incurred by measurement and SWAP operations.

We evaluate our method on applications of QAOAs and synthetic deep circuits. The ability to generate deep circuits is especially useful as they can be designed to have optimal gate counts [28]. This means no gates can be removed during compilation, which is the worst-case scenario for compiler optimization.

Additionally, we evaluate our mapper for parallelizing circuit cutting algorithms. This method [24] reduces error by splitting circuits into smaller subcircuits. Executing smaller circuits, even at larger quantities, may result in better fidelity as smaller circuits can be placed with greater flexibility.

We propose a new algorithm based on classical scheduling techniques that better encodes the constraints of quantum computing systems. This algorithm uses dynamic programming to reduce the slowdown from scheduling uneven circuits simultaneously. We use a binary linear programming formulation based on the identical-machines and bin-packing problem formulations to balance makespan and fidelity while reducing overheads incurred from workload changes. We include support for jobs with different shot lengths.

Chapter 2

BACKGROUND

2.1 Topology and Gates Errors

Quantum circuits evolve individual qubit states as well as qubit pair states, creating entanglement. Due to the limitations of the hardware, not all pairs of qubits can be operated on jointly. This is captured in the qubit connectivity of a hardware, represented as a graph. Qubits are shown as nodes, with edges indicating allowed operations between qubits. Pairwise gates are often implemented as either CNOT or ECR gates, which evolve the state of a target qubit based on the state of a control qubit. For some hardware types, edges are directed, indicating which qubits within each pair may be the target or control.

Physical limitations on gate operations creates the need for a gate scheduler which designates the order of gates between qubits. Qubits involved in a gate must be connected prior to the gate's execution. When this is not the case, we must identify a path between the qubits and apply the SWAP operator between qubits along the path. Each SWAP operator decomposes to 3 CNOT gates, so reducing SWAP paths through efficient scheduling is important to reduce circuit depth and runtime.

Circuit compilation is separated into mapping, gate scheduling, and gate optimization. Mapping selects which hardware qubits are used for each qubit in the quantum circuit. Maximum efficiency of gate scheduling depends strongly on the hardware qubits chosen to execute the circuit. The subtopology used for a circuit determines how often SWAPs must be insert and how well they can be avoided. Thus achieving greater optimality in circuit mapping is important for improving scheduling techniques.

Because of the optimality dependence of scheduling on mapping, we seek a new approach to solving the mapping problem.

2.2 Circuit Execution

The goal of executing a quantum circuit is to measure the resulting probability distribution. Because we cannot do this with a single sample, we re-run each circuit several times. Each run is a shot.

When circuits are parallelized, multiple circuits are executed simultaneously during each shot. Due to crosstalk from measurement operations, all circuits must be measured simultaneously at the end of the shot. This means that the runtime of the shot is the runtime of the slowest circuit. When scheduling parallelism, we must take care not to schedule together circuits with different runtimes. This will slow down the execution of the shorter circuit.

There is significant overhead with executing a circuit. We must compile the circuit, which involves several NP hard problems including qubit mapping and gate/SWAP scheduling. Additionally, error mitigation techniques may be applied on the circuit results after execution. Both of these incur significant time costs. Every time we change the workload on a computer e.g. because a circuit finishes or we add a new circuit, we incur the compilation and error mitigation overheads. It is important to reduce the number of times we change the set of parallel circuits to reduce this overhead.

Finally, scheduling more jobs in parallel increases competition for high-fidelity qubits and gates, decreasing the fidelity of execution. This means we must be conservative when introducing parallelism as to maintain the usefulness of circuit results.

Chapter 3

RELATED WORK

3.1 Greedy Methods for Mapping

Many previous approaches [6, 14, 15, 17–19, 21] are greedy. Generally, these methods present heuristics for modeling qubit and gate fidelity, crosstalk, and other sources of error. They select qubits and gates sequentially based on their heuristic value. In the case of parallel mapping, multiple distinct circuits are represented through *partitions*, or sets of qubits allocated for each circuit. These partitions are also formed greedily.

3.2 Optimizers for Mapping

Nannicini et al. [16] employs a binary integer linear programming approach for qubit mapping and scheduling. However, the restriction to linear programming reduces the representability of hardware information, especially for 2-qubit gates. We devise a nonlinear programming approach focused on more accurate hardware error representation.

3.3 Crosstalk

Previous work [17, 19] determined that crosstalk can be almost fully mitigated with a 1-qubit gap between circuits. We follow these results by mitigating the number of *inter-circuit connections*, or edges between qubits of different circuits. This is necessary to minimize noise on current devices. IBM recently announced that new devices will be largely immune to crosstalk [8]. However, including separation between circuits provides more flexibility for scheduling SWAP operations and allocating ancilla qubits in later stages of the optimization pipeline. In our approach,

we consider the usefulness of crosstalk mitigation both for its intended purpose and for scheduling improvement.

3.4 Measurement Crosstalk

Das et al. [6] showed that measurement operations introduced errors on the states of qubits that had no entanglement with the measured qubits. In single-circuit applications, qubits are often measured all together at the end and thus measurement crosstalk is a non-issue. For parallel circuit execution, circuits have different runtimes and end at different times. To avoid circuits from idling, it is important to schedule circuit execution as late as possible to finish at the same time for synchronous measurement. While this constraint does not affect mapping, it is crucially important for parallel job scheduling and we use this scheduling policy in all of our experiments.

3.5 Classical Job Scheduling

In classical scheduling, the identical-machines formulation is the simplest case where all machines are assumed to be identical. This problem has several polynomial-time algorithms that can get arbitrarily close to the optimal solution [5, 11].

The problem can be generalized to the job-shop scheduling problem which considers machines with different properties [4, 7, 9]. Since the fidelity and runtime of circuits varies between computers, this is better suited for our situation. However, we need to schedule multiple jobs to one device, rather than the other way around.

Assigning multiple jobs to one machine is an alternate formulation for e.g. scheduling containers to nodes in an HPC environment [25, 27]. This is closer to the problem we are trying to solve as we need to schedule multiple circuits on one computer, but is missing objectives analogous to fidelity maximization. While multiple scheduling models resource constraints and runtime costs for increasing the job count, classical errors are low enough that they are not considered.

3.6 Quantum Job Scheduling

Liu et al. [14] propose a parallel scheduling algorithm specific for quantum computing. They consider a single seed job and then sort the remaining jobs by similarity based on circuit depth. The jobs are considered one-by-one. If adding the job to the workload does not decrease the expected fidelity by more than a factor of ϵ, then the job is added.

QuCloud+ assumes all jobs require the same number of shots. Their depth similarity metric seeks to reduce the runtime disparity during simultaneous execution. However, the depth is a soft constraint so circuits with significant dissimilarity can still be scheduled together. This can cause significant slowdown for certain jobs in the workload.

Chapter 4

QUBIT MAPPING WITH BINARY INTEGER NONLINEAR PROGRAMMING

4.1 Solver

We separate the mapping optimization problem into two parts, outlined in Algorithm 1. The triangle symbol on the right side of certain lines indicates a corresponding section or equation that explains that line.

The first step (`Cluster()`) assigns each circuit a subset of qubits to use. This is equivalent to the partitioning phase in greedy algorithms, but extends it to optimize all qubit subsets simultaneously. Since subsets tend to be contiguous, we call this stage clustering. Clustering uses only information from the hardware (`f,A,S`) to produce the mapping (`X`).

The second step (`Place()`) places each circuit on a cluster and assigns each virtual qubit to a qubit within the cluster. Circuits and clusters are split into groups based on length as circuits cannot be assigned to clusters of different length. The optimization produces a permutation (`Y,Z`) which we apply to X to derive the final mapping. Placement incorporates information about qubit and circuit usage (`W,G`).

The following section describes the technical details and definitions of this algorithm.

4.1.1 Clusterer

The goal of the clusterer is to find sets of qubits for each circuit that optimize measurement and gate fidelity. To do so, we must create a representation for the machine calibration data and qubit assignments so that we can easily represent constraints and objective terms.

Algorithm 1 Optimization Algorithm

function CLUSTER(circuits, backend)
    $f, A, S \leftarrow$ Model noise ▷ 4.1.1, 4.1.1
    $X, U \leftarrow$ Create assignment variables ▷ 4.1.1
    $C(X), C(U)$ Apply constraints ▷ (4.5)-(4.7)
    $O(X, f, A, S)$ Define objective ▷ (4.8)-(4.12)
    Optimize X w.r.t (4.12)
    return X
end function

function PLACECOHORT(Q, circuits, backend)
    $F, \mathcal{S} \leftarrow$ Model noise ▷ 4.1.2
    $W, G \leftarrow$ Model circuit gates ▷ 4.1.2
    $Z_{ij} = I$ Initial mapping
    for all circuit$_i, Q_j \in$ circuit-cluster pairs **do**
        $Z \leftarrow$ Create permutation variables ▷ 4.1.2
        $C(Z)$ Apply constraints ▷ (4.13)
        $O(Z, W_i, G_i, F_j, \mathcal{S}_j)$ Define objective ▷ (4.14)-(4.16)
        Optimize Z w.r.t (4.16)
    end for
    $Y \leftarrow$ Create permutation variables ▷ 4.1.2
    $C(Y)$ Apply constraints ▷ (4.13)
    $V_{ij} \leftarrow O(Z_{ij}, W_i, G_i, F_j, \mathcal{S}_j)$ Compute pair values
    $O(Y, V)$ Define objective ▷ (4.17)
    Optimize Y w.r.t (4.17)
    $Z'_i \leftarrow Z_j \Leftrightarrow Y_{ij} = 1$ Choose best pairs
    $Q'_i \leftarrow Q_j \Leftrightarrow Y_{ij} = 1$ Permute circuits
    $Q'_{ij} \leftarrow Q'_{ik} \Leftrightarrow (Z'_i)_{kj} = 1$ Permute qubits
    return Q'
end function

function PLACE(X, circuits, backend)
    Split circuits/clusters into cohorts ▷ 4.1.2
    cohorts$_i \leftarrow \{X_j \in X, circ_j \in circuits | N_j = i\}$
    for all Q_i, circs$_i \in$ cohorts **do**
        $Q_i \leftarrow$ PLACECOHORT(Q_i, circs$_i$, backend)
    end for
    $Q \leftarrow$ Combine Q_i into original circuit order
    return Q
end function

function OPTIMIZE(circuits, backend)
    $X \leftarrow$ CLUSTER(circuits, backend)
    $X \leftarrow$ PLACE(X, circuits, backend)
    return X
end function

Machine Information

The variables and expressions used to represent the machine calibration data are summarised below. We use $\mathbb{Z}_2 = \{0, 1\}$ to represent binary matrices and variables and $\mathbb{I} = [0, 1]$ to represent values in the unit interval such as probabilities.

- N: The number of qubits on the target machine.
- C: The number circuits to be executed in parallel.
- $A \in \mathbb{Z}_2^{N \times N}$: Machine topography represented as an adjacency matrix. IBM's machine topographies are directed graphs where direction indicates the allowed control/target qubits. The control/target qubits can be reversed by applying Hadamard gates so for simplicity, we ignore direction. Then A is symmetric. Formally, $A_{ij} = 1 \Leftrightarrow$ a 2-qubit gate can be applied between q_i and q_j.
- $E \in \mathbb{I}^{N \times N}$: Error adjacency matrix where E_{ij} is the probability of an error when executing a 2-qubit gate on q_i and q_j. The 2-qubit gate considered here depends on which basis gates are available in the quantum computer, but is often the CNOT or ECR gate.
- $f \in \mathbb{I}^N$: Readout error vector where f_i is the probability of an error when measuring q_i.

Swap Error Matrix

Prior approaches generate objective functions by selecting qubits which have minimal values in E and f [15, 17]. Such qubits are likely to be non-adjacent, incurring error and time costs from the necessary SWAP gates. Approaches address this by adding separate contiguity constraints, selecting qubits that are adjacent to a previously selected qubit. Rather than artificially restricting the search space with

contiguity and closeness constraints, we propose a novel heuristic that more explicitly models the cost of introducing SWAP gates.

A SWAP insertion between two qubits is represented by a path in the topology graph. We model the cost of a SWAP insertion as the probability of any error when applying gates along the path. For an unweighted adjacency matrix, A, the expression A^n computes the number of n-length paths between all pairs of nodes in the graph. For a weighted adjacency matrix, E, this computes the sum of the product of path weights over all n-length paths. This expression is formalized in (4.1), using $P_{ij}^{(n)}$ as the set of n-length paths between q_i and q_j. Because A_{ij}^n is the number of paths, $\frac{E_{ij}^n}{A_{ij}^n}$ is the average product of weights across all n-length paths between q_i and q_j.

$$E_{ij}^n = \sum_{p \in P_{ij}^{(n)}} \left(\prod_{q_k \to q_l \in p} \left(E_{kl} \right. \right. \tag{4.1}$$

Now we consider $(1-E)^N$. For each n-length path p, the product becomes $\prod_{q_k \to q_l \in p} (1 - E_{kl})$. Since E_{kl} is the probability of an error when applying a gate between q_k and q_l, $1 - E_{kl}$ is the probability of a success. Gate errors are assumed to be independent and taking the product over all edges in the path gives the probability of success when applying all gates along the path. SWAP operations can be decomposed into three CNOT gates. Thus for CNOT-based machines, the SWAP success chance is exactly calculated by cubing the success probability for all edges except the one which applies the final post-swap gate. During the mapping phase, it is unknown at which edge the final gate will be applied so we ignore this technicality. Since x^3 increases monotonically for $x \in \mathbb{I}$, we do not need to cube the success probability. The heuristic is directly proportional to the SWAP success chance. Finally, we take the complement of the result to obtain the probability of at least one error during the SWAP, which we later minimize.

However, this method only computes success chances for a fixed path length n.

Since qubit pairs vary in topological distance, we must determine n for each pair. Since increasing the length of a SWAP path will increase its error, we make the assumption that the SWAP scheduler will choose from the shortest SWAP paths. Then for each qubit pair, we set n as their geodesic distance. This is computed by applying Seidel's algorithm to A to derive the distance matrix $D \in \mathbb{Z}^{N\times N}$ where D_{ij} = geodesic path length between q_i and q_j [26]. The definition for the swap error matrix, $S \in \mathbb{Z}_2^{N\times N}$, is given in (4.2).

$$S_{ij} = 1 - \frac{(1-E)_{ij}^{D_{ij}}}{A_{ij}^{D_{ij}}} \tag{4.2}$$

This expression models the average probability of a SWAP error over all geodesic paths between two qubits. However, an intelligent SWAP scheduler will not choose a random SWAP path, but instead choose SWAP paths with higher success chances. It is more useful for our optimization to find the minimum probability of a SWAP error. In order to compute this without losing the polynomial formulation, we incorporate an approximation technique based on the p-norm. The p-norm can be used to approximate the maximum element of a vector by taking $\lim_{p\to\infty}$. Likewise, taking $\lim_{p\to-\infty}$ approaches the minimum element. For practical purposes, we use p with a large, finite magnitude to approximate the minimum or maximum (see Fig. 4.1).

Considering a vector which contains the SWAP success probability of every n-length path between q_i and q_j, the p-norm equation becomes (4.3). We do not need to use absolute value since probabilities are nonnegative. The p exponent is distributed into the product term and the resulting expression is analogous to (4.1) with $(1-E)^{\circ p}$ instead of E. We use $\circ p$ to represent element-wise exponentiation (also called Hadamard exponentiation). We combine (4.2) and (4.3) to produce the final formulation of S in (4.4). We reintroduce the expression A into the denominator so that at $p = 1$, S approaches the average SWAP error probability. As $p \to \infty$, the denominator becomes 1 and S approaches the minimum SWAP

error probability. Using this metric indicates full trust for the scheduling phase to choose good SWAP paths. As $p \to -\infty$, the denominator again becomes 1 and S approaches the maximum SWAP error probability. This models the case of a malicious scheduler, which is interesting but outside the scope of this paper. Fig. 4.1 shows the relationship between S and p.

$$\begin{aligned} d_{ij}^{(n)}(p) &= \left(\sum_{p \in P_{ij}^{(n)}} \left(\prod_{q_k \to q_l \in p} (1 - E_{kl}) \right)^p \right)^{\frac{1}{p}} \\ &= \left(\sum_{p \in P_{ij}^{(n)}} \left(\prod_{q_k \to q_l \in p} (1 - E_{kl})^p \right)^{\frac{1}{p}} \right) \\ &= (((1 - E)^{\circ p})_{ij}^n)^{\frac{1}{p}} \end{aligned} \tag{4.3}$$

$$S_{ij} = 1 - \left(\frac{((1 - E)^{\circ p})_{ij}^{D_{ij}}}{A_{ij}^{D_{ij}}} \right)^{\frac{1}{p}} \tag{4.4}$$

Qubit Assignment

We use the matrix and vector encoding of the machine data to constrain and optimize the selection of qubits using matrix and vector multiplication. We encode the assignment as a matrix $X \in \mathbb{Z}_2^{C \times N}$ such that $X_{ij} = 1 \Leftrightarrow q_j$ is assigned to cluster i. We defined an additional vector $U \in \mathbb{Z}_2^N$ such that $U_j = 1 \Leftrightarrow q_j$ is unassigned. We use the notation X_i and X_{*j} to refer to the i^{th} row and j^{th} column vectors of X, respectively.

From this formulation, we derive two constraints. Constraint C_1 in (4.5) requires the number of assigned qubits to equal the number of qubits in the corresponding circuit. This ensures our cluster sizes match our circuit sizes. Constraint C_{1b} in (4.6) requires that all remaining qubits be unassigned. Constraint C_2 in (4.7) ensures each qubit is either unassigned or assigned to exactly one cluster. C_1 and C_2 imply

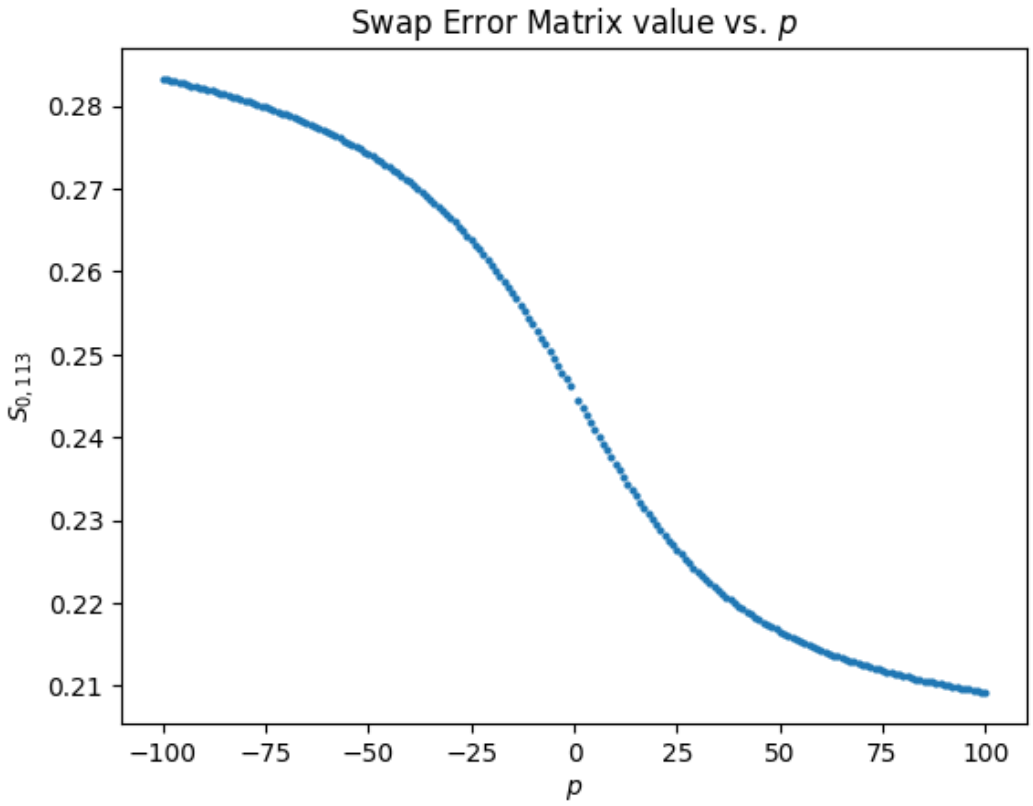


Figure 4.1: Swap error matrix value between q_0 and q_{113} for varying values of p. q_0 and q_{113} have multiple geodesic paths. Calculations used calibration data for IBM Brisbane on 2024-01-31 13:13:52

C_{1b}, however we explicitly encode C_{1b} for optimization purposes as described in Section 4.2.

$$C_1: \quad \|X_i\|_1 = N_i \qquad 1 \le i \le C \tag{4.5}$$

$$C_{1b}: \quad \|U\|_1 = 1 - \sum_{i=1}^{C} \left(N_i \right. \tag{4.6}$$

$$C_2: \quad U_j + \|X_{*j}\|_1 = 1 \qquad 1 \le j \le N \tag{4.7}$$

These constraints ensure that a solution can be turned into a valid mapping. To find a quality solution, we define several objective terms and minimize their weighted sum.

$$O_1(X) = \|Xf\|_1 \tag{4.8}$$

(4.8) sums the readout error of all used qubits. We want to minimize this.

$$O_2(X) = tr(XSX^T) \tag{4.9}$$

For (4.9), $(XSX^T)_{ij}$ sums the SWAP error approximation for all edges between cluster i and cluster j. By taking the trace, we only consider when $i = j$. Thus this expression sums the SWAP error for all qubit pairs that are in the same cluster. We want to minimize the result.

$$O_3(X) = \sum_{i \neq j} \left(X_i A X_j^T \right. \tag{4.10}$$

For (4.10), $X_i A X_j^T$ computes double the number of edges between qubits of cluster i and cluster j. We are iterating over $i \neq j$ so this expression is the number of inter-cluster edges. When this value is 0, we have at least a one-qubit gap between all clusters. By minimizing this expression, we protect against cross-talk error.

$$O_4(X) = tr(XAX^T) \tag{4.11}$$

Equation (4.11) uses the same expression as O_3, but iterates over $i = j$. This produces the number of intra-cluster edges. Maximizing these edges acts as a soft contiguity constraint. While O_2 prefers contiguous assignments, varying gate and qubit fidelities means a non-contiguous assignment can be more optimal. The mapping stage does not consider circuit properties. We use this objective term to indicate preference towards contiguous assignments for e.g. circuits with a large number of 2-qubit gates. To maximize O_4, we set the weight $\alpha_4 < 0$.

$$X^* = argmin_X \sum_{i=1}^{4} \left(\alpha_i O_i(X) \right. \tag{4.12}$$

X^* in (4.12) optimizes our assignment matrix by minimizing the weighted sum of all the objective terms. The weighting allows us to fine-tune the impact of each term and disable terms.

4.1.2 Placer

The goal of the circuit placement problem is to decide which circuits will be assigned to which clusters and which virtual qubits will be assigned to which physical qubits. Circuits can only be assigned to clusters with the same number of qubits. Thus, we split our circuits and clusters into cohorts based on their width. We solve the placement problem for each cohort independently.

Machine Information

We reuse the machine information calculated for the clustering problem by extracting only the values relevant to each cluster.

- D: The number of circuits and clusters in the cohort.
- M: The number of qubits in each circuit/cluster.
- $Q \in [[1, N]]^{D \times M}$: This matrix encodes the qubits selected for each cluster by the mapper. $Q_{ij} = k \Leftrightarrow q_k$ is the j^{th} qubit in the i^{th} cluster.
- $F \in \mathbb{I}^{D \times M}$: The fidelity matrix derived from f. Each row is a cluster and each column is a qubit. $F_{ij} = f_{Q_{ij}}$.
- $\mathcal{S} \in \mathbb{I}^{D \times M \times M}$: The swap matrix for each cluster derived from S. $(\mathcal{S}_i)_{jk} = S_{Q_{ij}Q_{ik}}$.

Circuit Information

If we only use machine information, then every circuit is identical and all solutions have the same optimality. To evaluate the goodness of solutions, we construct

metrics to encode circuit information that mirror the hardware information of F and $\mathcal{S}$.

- $W \in \mathbb{Z}^{D\times M}$: The weight matrix describes how often we use each qubits. We use this to apply weights to F. Qubits that are used more often accumulate more errors and should be placed on higher fidelity qubits. W_{ij} is the number of gates that use q_j in circuit i.

- $G \in \mathbb{Z}^{D\times M\times M}$: The gate matrix is used to weight $\mathcal{S}$ and describes how often qubits are operated on in pairs. $(G_i)_{jk}$ is the number of 2-qubits gates that use q_j and q_k in circuit i. The goal is that qubits that are used more often together should be placed on qubits with paths that are shorter and have higher fidelity edges.

Permutation Matrices

To encode the mappings for the placement problem, we use several permutation matrices. This ensures the mapping represented by each matrix is bijective.

- $Y \in \mathbb{Z}_2^{D\times D}$: This is the permutation matrix for mapping circuits to clusters. Specifically, $Y_{ij} = 1 \Leftrightarrow$ circuit i is assigned to cluster j.

- $Z_{ij} \in \mathbb{Z}_2^{M\times M}$: This is the qubit permutation matrix for circuit i and cluster j. $(Z_{ij})_{kl} = 1 \Leftrightarrow q_l$ of circuit i maps to q_k of cluster j.

To fully optimize the placement, we must optimize Z_{ij} for all circuit-cluster pairs (i.e. $1 \le i, j \le D$). Using the results, we optimize Y to decide which circuit-cluster pairs will actually be used. Since this is $O(D^2)$ optimization problems, we explore alternate techniques in Section 4.1.2 that reduce the number of pairs we consider.

The only constraint we need is that Y, Z_{ij} be permutation matrices. C_1 in (4.13) shows the formulation for the permutation constraint on Y and Z_{ij}.

$$C_1: \quad Y\mathbf{1} = Y^T\mathbf{1} = Z_{ij}\mathbf{1} = Z_{ij}^T\mathbf{1} = \mathbf{1} \tag{4.13}$$

We use the permutation matrices to construct objective terms. Our general strategy is to take the circuit information, and multiply it with the permutation matrix. This orders the circuit metrics to match which physical qubits are used by each virtual qubit. We multiply the permuted circuit weights with the hardware information and optimize the result.

$$O_1(Z_{ij}) = F_j Z_{ij} W_i \tag{4.14}$$

O_1 in (4.14) sums the qubit errors weighted by how much each qubit is used. $(Z_{ij}W_i)_k$ is the weight for q_k of the cluster and indicates how much q_k would be used if Z_{ij} is the mapping. We want to minimize the result.

$$O_2(Z_{ij}) = \sum_{kl} \Big((\mathcal{S}_j)_{kl} (Z_{ij} G_i Z_{ij}^T)_{kl} \tag{4.15}$$

O_2 in (4.15) sums the SWAP errors weighted by how much each qubit pair is used. $Z_{ij}G_iZ_{ij}^T$ permutes the rows and columns of G_i so that index k,l is the number of times q_k, q_l of the cluster are used together under the mapping Z_{ij}. We mutliply these weights element-wise with $\mathcal{S}_j$ and sum every entry. We want to minimize the result.

$$Z_{ij}^* = \arg\min_{Z_{ij}} \sum_{k=1}^{2} \Big(\alpha_k O_k(Z_{ij}) \tag{4.16}$$

Z_{ij}^* in (4.16) minimizes the weighted sum of each objective term. Once we've optimized the qubit mapping for each circuit-cluster pair, we decide which circuit-cluster pairs we will actually use. (4.17) formulates the subsequent minimization problem. Y_{ij} selects the objective values for the circuit-cluster pairs that are used. Y^* selects the pairs that provide the minimum errors.

$$Y^* = \arg\min_{Y} \sum_{ij} \left(Y_{ij} \sum_{k=1}^{2} \left(\alpha_k O_k(Z_{ij}) \right. \right. \tag{4.17}$$

Optimization

It is possible to combine (4.16) and (4.17) and produce a singular optimization problem over Y and Z. This produces a problem with cubic objectives which is difficult to solve. Solvers such as Gurobi do not allow cubic constraints, which must be decomposed into quadratic constraints, introducing a large number of intermediate variables. We observed that splitting the formulation into multiple subproblems drastically reduces the runtime needed to optimize the problem.

The number of qubit optimization problems posed here grows quadratically with the number of circuits in the cohort. We propose modified algorithms to reduce the number of pairs that are optimized. We still optimize (4.17) normally, but for every pair that is not optimized, we replace $O_k(Z_{ij})$ with $O_k(I)$.

- Greedy: We continually remove the circuit or cluster that performs the best as we optimize mappings. This can also be performed over each circuit, removing the best cluster and vice versa. The rationale is that a circuit-cluster pair with a good objective value will likely be used so either the circuit or cluster do not need to be optimized for other combinations.

- Linear: We pick a random mapping of circuits to clusters and optimize the qubit mappings for only those pairs. The rationale for this approach is that qubit mapping is more important than the circuit-cluster mapping. As long as we optimize the qubit mapping, we will achieve reasonably good results. This approach requires solving a linear number of qubit-mapping optimizations.

4.2 Meta-Optimization

The method we use to encode constraints and objectives is very important to how well the optimizer performs. There are several 'meta'-optimizations that we applied to improve the efficacy of the solver.

4.2.1 Equalities > Inequalities

In the given formulation, we include an additional term U for unassigned qubits. This allows us to formulate (4.7) as an equality. Using an inequality such as $\|X_{*j}\| \leq 1$ has the same effect, but is harder for Gurobi to optimize.

4.2.2 Soft Constraints

We first formulated O_3 as a hard constraint enforcing a 1-qubit gap between circuits. However, we found that expressing this term in the objective function improved the optimality of solutions, especially with large numbers of circuits. By using a large α weight, we can still force the optimizer to reduce the number of inter-cluster edges to 0. For extreme cases of high throughput, relaxing this constraint allows Gurobi to search for solutions more freely.

Chapter 5

QGROUP - PARALLEL JOB SCHEDULER

To ensure efficient scheduling of jobs without sacrificing accuracy, we devise a three-part algorithm outlined in Algorithms 2-5. We call this algorithm *QGroup* due to the emphasis on grouping jobs prior to scheduling.

Due to measurement synchronization, each shot has the runtime of the longest circuit in the group. For example, if circuit i takes t time per shot and is scheduled with circuit j which takes $2t$ time per shot, then circuit i will take twice as long to complete. We can only parallelize circuits with similar per-shot runtimes. Algorithm 2 reduces this problem to a modified instance of rod cutting, allowing us to find the optimal grouping of circuits with respect to a objective function in $O(N^3)$ runtime.

Once we group the circuits by runtime, we must determine their order of execution (Algorithm 3). This problem is a tradeoff between maximizing the fidelity and minimizing the makespan. The fidelity is trivially optimized by running all circuits sequentially, which also maximizes the makespan. This is undesirable, so we formulate the makespan minimization problem using binary integer linear programming (BILP). We then partially encode the fidelity maximization using constraints. Fidelity is further maximized in the final assignment stage.

At this point, we have methods for grouping circuits and scheduling parallelism with minimal makespan. We repeat this process for all available computers so that we may select the best computer for each group. To create a "goodness" score for each machine, we combine an extended makespan metric with an estimate of fidelity, as shown in Algorithm 4. Groups are assigned according to the order of jobs in the queue.

Algorithm 5 shows how these sub-algorithms are orchestrated. This chapter provides greater detail of the formulation and mathematics of the algorithms.

Algorithm 2 Job Partitioning Algorithm

function RECONSTRUCT(T, I, s, l)
  $i \leftarrow I_{s,l}$
  if i == 0 **then**
    return $[T_{s\ldots s+l}]$
  else
    return RECONSTRUCT(T, I, s, i) $\|$ RECONSTRUCT($T, I, s+i, l-i$)
  end if
end function

function PARTITION($T, \alpha, \beta, \gamma, d$)
  sort(T)
  $D_{s,l} \leftarrow d(T_{s\ldots s+l})$ ▷ (5.4)
  $D'_{s,1} \leftarrow \alpha + \gamma T_s$ ▷ (5.2)
  for all l in $2..n$ **do**
    for all s in $1..n-l$ **do**
      $F \leftarrow \{D_{s,l}\} \cup \{(D'_{s,i} + D'_{s+i,l-i} + \beta) | 1 \leq i < l\}$ ▷ (5.1)
      $D'_{s,l} \leftarrow \min F$ ▷ (5.1)
      $I_{s,l} \leftarrow \arg\min F$
    end for
  end for
  return RECONSTRUCT($T, I, 1, n$)
end function

Algorithm 3 Parallelism Scheduling Algorithm

function SCHEDULE(Q, g, δ, η)
  $Q \leftarrow \eta Q$
  $s_i \leftarrow g_i$.shots
  $q_i \leftarrow g_i$.qubits
  $\delta \leftarrow \max\{\delta, \gcd(s)\}$ ▷ 5.2.1
  $b_i = \lceil \frac{s_i}{d} \rceil$ ▷ 5.2.1
  $N \leftarrow |s|$
  $B \leftarrow \sum_{i=1}^{N} b_i$
  $X, Y, Z \leftarrow$ Create model variables ▷ (5.9),(5.10)
  $C(X, Y, X)$ Apply constraints ▷ (5.5),(5.6),(5.8)
  $O(X, Y, Z)$ Apply objective ▷ (5.7),(5.11)
  Optimize X, Y, Z ▷ (5.12)
  return X, Y, Z
end function

Algorithm 4 Machine Selection Algorithm

function ASSIGN(C, T, queue) ▷ 5.3
  $G_c \leftarrow$ PARTITION$(T, \alpha, \beta, \gamma, d)$
  $X_{cj}, Y_{cj}, Z_{cj} \leftarrow$ SCHEDULE$(Q_c, G_{cj}, \delta, \eta)$
  for all i in queue **do**
    if i is assigned **then**
      continue
    end if
    // Iterate computer types
    for all c in C **do**
      $g \leftarrow g \in G_c | i \in g$
      $m \leftarrow \text{makespan}(X_{cg})$
      // Iterate computers
      for all c_j in c **do**
        $t_f \leftarrow t(c_j) + m$ ▷ (5.13)
        // Apply any mapping algorithm
        $q \leftarrow \text{map}(g, c_j)$
        $p \leftarrow \min_k \log \text{EPST}_k(c_j, g, q)$
        $s(g, c_j) \leftarrow \log t_f - p$ ▷ (5.14)
      end for
    end for
    $g^*, c^* \leftarrow \arg\min s(g, c_j)$ ▷ (5.15)
    for all j in g^* **do**
      assign j to c^* according to X_{g^*}
    end for
    // Recompute and schedule groups
    $T' \leftarrow T_{cj} | j$ is not assigned
    for all c in $C - \{c^*\}$ **do**
      $G_c \leftarrow$ PARTITION$(T', \alpha, \beta, \gamma, d)$
      $X_{cj}, Y_{cj}, Z_{cj} \leftarrow$ SCHEDULE$(Q_c, G_{cj}, \delta, \eta)$
    end for
  end for
end function

Algorithm 5 Full QGroup Algorithm

function QGROUP(C, queue)
  $h \leftarrow \{\text{hash}(c) | c \in C\}$
  $C'_j \leftarrow [c \in C | \text{hash}(c) = h_j]$
  $T_{ji} \leftarrow t(\text{queue}_i, C'_{j0})$
  ASSIGN$(C', T$, queue$)$
end function

5.1 Job Grouping with Rod Cutting

The number of possible partitions of jobs is exponential and thus searching for the optimal partition is intractable. We make the important observation that each job will be best scheduled either alone or in a group with at least one of its runtime neighbors. Then we can sort the list of jobs by runtime and only consider partitions of contiguous groups. This setup is a variant of the rod-cutting problem, where a rod must be cut and sold for the highest price with each segment-length having a fixed price. We define the "price" to be an arbitrary measure of dissimilarity $d(T)$ between the runtimes of the group, and we minimize the total dissimilarity.

The rod-cutting problem defines a price function that is independent of cut locations. However, job runtimes can vary and group dissimilarity changes based on both the location and length of the group. We need a dissimilarity matrix D where $D_{sl} = d(T_{s...s+l})$. We reflect this change in the recursive formulation shown in (5.1). This is identical to the original rod-cutting formulation with the added location index. We discuss the β term later.

$$D'_{s,l} = \min\{D_{s,l}, (D'_{s,i} + D'_{s+i,l-i} + \beta) | 1 \leq i < l\} \tag{5.1}$$

Any measure of dissimilarity is trivially minimized by grouping jobs by equal runtime. Since runtimes are expected to be different between jobs, this leads to zero parallelism. To prevent this, we add an initial cost to placing a job in its own group in (5.2). The α term is a flat cost for every group. γ defines a relative cost that scales with the runtime of each job. $D'_{s,1}$ increases the similarity required of $D'_{s+1,l-1}$ in order to separate T_s into its own group. (5.3) shows this fact. A similar property holds when T_s is at the end of the group. When dissimilarity grows with T, using γ ensures the similarity bound scales with T as well.

While α, γ limit the number of single-job groups, β in (5.1) limits the overall number of groups. This prevents the group formation from creating groups of size

2 to mitigate the α, γ constraints.

$$D'_{s,1} = \alpha + \gamma T_s \tag{5.2}$$

$$\begin{aligned} \text{Split } T_s &\Leftrightarrow D'_{s,1} + D'_{s+1,l-1} < D'_{s,i} + D'_{s+i,l-i} \text{ for } 2 \leq i \leq l-2 \\ &\Rightarrow D'_{s+1,j-1} < D'_{s,j} + D'_{s+i,l-i} - D'_{s,1} < D'_{s,j} + D'_{s+i,l-i} \end{aligned} \tag{5.3}$$

Finally, we need an objective function for the dissimilarity of a group. (5.4) uses the ratio of maximum to minimum runtime as the objective. This value is the greatest slowdown factor that any job in the group will experience when being scheduled in parallel.

$$d(T) = \frac{\max T}{\min T} - 1 \tag{5.4}$$

5.2 Parallel Scheduling with BILP

The result of Algorithm 2 is groups of jobs with similar runtimes. Let $g = g_{1\ldots N}$ be one such group. Similar runtime indicates the these jobs *may* be scheduled in parallel. We must decide which jobs are actually scheduled while respecting qubit constraints, maximizing fidelity, and minimizing makespan. As discussed, the latter two objectives constitute a tradeoff.

There has been much research into minimal-makespan scheduling for classical workloads [5, 7, 9, 11]. However, little research considers the case of assigning multiple jobs to one machine [25] and we have found none that have analogous constraints to qubit count and parallel fidelity. Instead, we model these constraints by adapting linear programming techniques for classical scheduling.

5.2.1 Representation

- $Q \in \mathbb{Z}$: The number of qubits on the machine.

- $N \in \mathbb{Z}$: The number of circuits in g.
- $\underline{s} \in \mathbb{Z}^N$: The number of shots in each circuit of g.
- $\underline{q} \in \mathbb{Z}^N$: The number of qubits required for each circuit of g.

The unit of time for quantum circuit execution is a single shot. Because this is a discrete measure, we can represent the execution of quantum circuits exactly. We define $X \in \mathbb{Z}_2^{N \times S}$ where $S = \sum_{i=1}^{N} s_i$ such that $X_{ij} = 1 \Leftrightarrow$ circuit i is executing during shot j. S is the maximum number of shots that could execute, which is the case of zero parallelism.

Using maximum granularity is expensive as the number of shots can be arbitrarily high and is often in the range of $[1,000..10,000]$. To reduce the number of variables in the model, we introduce a granularity parameter δ. We resize our problem so that the unit of time is bins of δ shots. δ also helps us to align job executions, which we discussed later. Now we have $X \in \mathbb{Z}_2^{N \times B}$.

The best value for δ is $\gcd(s_{1..N})$ as this reduces the problem size while allowing us to schedule the exact correct number of shots for each job. In the case that the gcd is too low, we use a minimum δ which represents the maximum number of extra shots we are willing to run when scheduling a circuit. In most real-world cases, we are able to use the gcd with $\delta = 500, 1000$, or a multiple thereof.

- $\delta \in \mathbb{Z}^+$: The shot granularity.
- $\underline{b} \in \mathbb{Z}^N$: The number of shot "bins" calculated as $b_i = \lceil \frac{s_i}{\delta} \rceil$.
- $B \in \mathbb{Z} = \sum_{i=1}^{N} b_i$: The maximum number of bins that we may need to execute.

From here, we can define constraints to make X meaningful. The number of qubits required to run all concurrent circuits cannot exceed Q, which is captured in (5.5). Additionally, circuits must be scheduled for the correct number of bins, shown in (5.6). We introduce $\eta \in [0, 1]$ to reduce the maximum number of qubits

we are allows to use concurrently. Using all or nearly all qubits on a computer is detrimental to fidelity due to crosstalk and low-fidelity gates/qubits.

$$C_1: \quad \sum_{i=1}^{N} q_i Z_{ij} \leq \eta Q \qquad 1 \leq j \leq S \tag{5.5}$$

$$C_2: \quad \sum_{j=1}^{B} z_{ij} = b_i \qquad 1 \leq i \leq N \tag{5.6}$$

5.2.2 Objectives

The simplest objective is to minimize the makespan of the schedule. This is the highest value j where $\exists i$ s.t. $X_{ij} = 1$. While the maximum could be optimized by introducing temporary variables with constraints, we instead weight each X_{ij} by $\frac{j}{B}$ for a simpler expression. We sum every $\frac{j}{B} X_{ij}$ (5.7). This is minimized by executing circuits in lower bins. This minimizes not only the group makespan, but the makespan of each individual job within the group.

$$O_1 = \sum_{j=1}^{B} \frac{j}{B} \sum_{i=1}^{N} X_{ij} \tag{5.7}$$

We must also minimize the synchronicity of workload changes. These changes occur whenever a job begins or finishes execution. We then have a new set of circuits which incurs an overhead cost for compiling and applying error mitigation techniques. To limit this, we use indicator variables and conditional constraints.

Gurobi allows constraints of the form $p \Rightarrow q$ where $p = 1 \Leftrightarrow q$ is true. We define several $p_i = 1 \Leftrightarrow q_i$ constraints and then maximize/constrain $\sum_i p_i$. This maximizes/constrains the number of q_i that are true.

$$C_3: \quad \sum_{j=1}^{B-1} Y_{ij} + (1 - X_{i0}) + (1 - X_{jN}) == B - 1 \tag{5.8}$$

$$Y_{ij} = 1 \Rightarrow X_{i,j+1} == X_{i,j} \tag{5.9}$$

$$Z_j = 1 \Rightarrow \begin{cases} \sum_{i=1}^{N} Y_{ij} == N & 1 \leq j < B \\ \sum_{i=1}^{N} X_{i0} == 0 & j == B \\ \sum_{i=1}^{N} X_{iB} == 0 & j == B+1 \end{cases} \tag{5.10}$$

- $Y \in \mathbb{Z}_2^{N \times B-1}$ (5.9): $Y_{ij} = 1$ indicates that circuit i maintained its execution status from j to $j+1$. (5.8) ensures that the number of times the execution status does change is ≤ 2. (5.6) prevents the number of changes from being 0. Having only 1 change is not possible. Then (5.8) requires that the job changes status exactly twice. This is equivalent to contiguous execution.

- $Z \in \mathbb{Z}_2^{B+1}$ (5.10): $Z_j = 1$ indicates *no* job changes when finishing bin j. $Z_B, Z_{B+1} = 1$ state that no jobs start/end in the first and last bins, respectively. Minimizing (5.11) seeks to maximize the number of such bins. This minimizes the number of times we have to change the workload.

$$O_2 = -\sum_{j=1}^{B+1} Z_j \tag{5.11}$$

$$X^* = \arg\min_X (\alpha_1 O_1 + \alpha_2 O_2) \tag{5.12}$$

5.3 Machine Selection

The job grouping and parallel scheduling algorithms provide us the optimal schedule for each group on each computer. The final task remains to assign groups to computers. Note that the group partitions may not be the same across different computer types. Algorithm 4 presents a greedy algorithm to make this selection.

The goal of the algorithm is to minimize the makespan and maximize the fidelity for each job. We give preference to minimizing the makespan of the earlier jobs

in the queue as earlier queue status warrants earlier completion. We consider all jobs in the group when maximizing fidelity, as all jobs should be afforded similar execution fidelities.

To evaluate fidelity, we use the expected probability of a successful trial (EPST) as defined by Liu et al. in [14]. Higher EPST indicates greater fidelity, which is desirable.

We iterate through each job in the queue. If it has not been assigned, we rank each computer according to (5.14). $g_c(i)$ is the group for machine c which contains job i. (5.13) Is the expected finish time of $g_c(i)$ if it were run on c. $t(c)$ is the number of shots left to complete the current workload on c. In (5.15), we minimize the expected finish time and maximize the *minimum* EPST of each job in $g_c(i)$.

$$t_f(c,g) = t(c) + \text{makespan}(g) \tag{5.13}$$

$$s(c,i) = \log(t_f(c, g_c(i))) - \log(\min_j \text{EPST}_j(c, g_c(i))) \tag{5.14}$$

$$c^* = \arg\min_c s(c,i) \tag{5.15}$$

Because groups are not the same across computers, assigning jobs from $g_{c^*}(i)$ will invalidate any other groups where $g_{c\neq c^*}(j) \cap g_{c^*}(i) \neq \emptyset$. Then the final step is to recompute the group partitions and schedules for all computers except c^*.

5.4 Performance

5.4.1 Runtime

Runtimes for the proposed algorithms are summarized in Table 5.1. Algorithm 2 has runtime of $O(N^3)$ consistent with the rod cutting problem. Algorithm 3 is difficult to analyze since the runtime of BILP solvers is not well defined. Instead, we evaluate the scaling of the number of variables in the model, $O(NB)$. Since the number of shots per job is limited by providers, $B = \kappa N$ for an upper bound κ. Then

Algorithm	Runtime
2	$O(N^3)$
3	# Vars: $O(NB) = O(N^2)$
4,5	$O(NC(\omega + N^3))$
4,5 optimized	$O(WC\omega + W^4C') = O(C\omega + C')$

Table 5.1: The runtimes of the scheduling algorithms. N is the number of jobs and C is the number of computers. C' is the number of computer types. ω is the runtime of the mapping function used in Algorithms 4,5. Runtime for BILP solvers is not specified in general, so we provide the scaling of the number of variables in the model.

the number of variables is $O(N^2)$. Algorithms 4,5 run the previous algorithms for each computer and (in the worst case) each job in the queue. Then the runtime is $O(NC(\omega + N^3))$. We add ω to be the runtime of the mapper used for computing EPST. The purpose of the mapper is to estimate the PST so we can reduce ω by using a mapper that uses a high degree of approximation.

The strongest dependence here is on N^4. However, this is easily mitigated by defining a window size W and processing batches of W jobs from the queue. Then the runtime is bounded by $O(CW^4)$ where W^4 is a constant. This replaces the quartic scaling with a configurable constant.

5.4.2 Computer Groups

While we can eliminate N from the runtime, the dependence on C is still problematic for performance, especially as providers release more and more quantum computers. As the number of computers goes up, the viable windows size goes down.

To address this, we group quantum computers by similar properties. The purpose of computer information in the job grouping and scheduling algorithms is to estimate the runtime of each circuit. The runtime is dependent on the computer gate speed and circuit depth. Since gate speed affects all circuit evenly and we only care about relative runtimes, we can ignore differences across computers.

Circuit depth is associated with the basis gate set and topology of the computer. The former determines how many gates are produced by decomposition. The latter

determines how many SWAP gates will need to be inserted. Then we expect that quantum computers with identical basis gates and topologies will produce equal (or similar) optimal job partitions. We relax the topology similarity to ignore gate direction as the connectivity has a greater effect on SWAP insertion than direction.

Now we can compute T based on the properties of each computer *group* rather than each singular computer. Similarly, algorithms 2, 3 can be applied per computer group. Only EPST is computed for every computer. The last row of Table 5.1 shows the runtime with all optimizations.

Chapter 6

RESULTS

6.1 Qubit Mapping

We ran several experiments to characterize the runtime and fidelity of our algorithm. For each experiment we explain our selections for $\alpha, t^{cl}_{max}, t^{pl}_{max}$, and the number of shots we use. t_{max} is the maximum time allocated for the clusterer (cl) and placer (pl), respectively. To quantify the results, we show values for $O_{cl}, O_{pl}, \epsilon_{min}, \epsilon_{max}$, and ϵ_{avg}. O is the objective value of the assignment, which we report for each step. ϵ is the mean squared error (MSE) metric. We use mean squared error for ϵ to quantify the difference between our experiments and the ground truth obtained from a statevector simulator. We compute and aggregate ϵ for all circuits in the experiment. We used Qiskit's transpiler with optimization level 3 to simplify gates and schedule SWAP gates. Qiskit's routing pass may adjust qubit locations slightly from the initial mapping. To show this, we calculate the objective values for our mapping pre- and post-transpilation. When using Qiskit's mapper as a baseline, we only provide the post-transpilation mapping.

6.1.1 Runtime

Gurobi's optimization algorithms use an iterative approach, which allows us to specify iteration and time limits and receive solutions after those constraints are reached. Thus, the runtime of the optimization step is largely irrelevant as we can specify it ourselves. Instead, we characterize the runtime-optimality trade-off by observing the change in objective as we increase the time constraints. We evaluate runtime on the QAOA examples (Section 6.1.2). For each set of time parameters, we run 10 trials to account for stochasticity in the Gurobi optimizer.

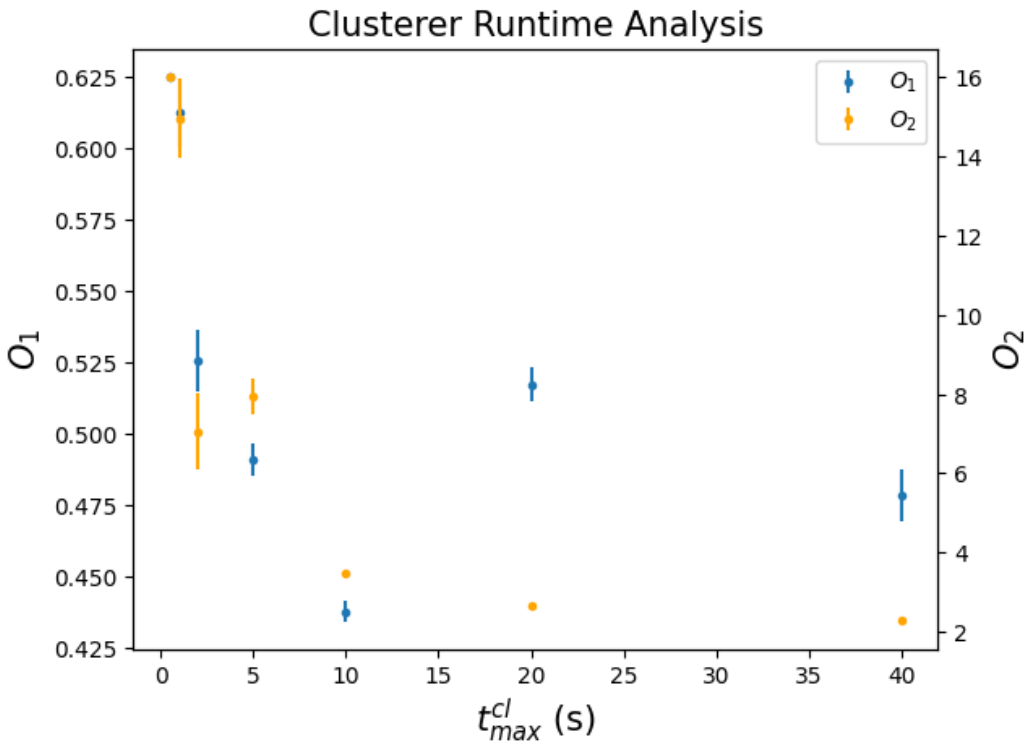


Figure 6.1: Plot of clusterer objective value for different maximum time constraints.

We plot the relationship between the objective value and run time for the clusterer and placer in Fig. 6.1 and Fig. 6.2. O_1 is the objective term for the qubit readout error defined in (4.8) and (4.14). O_2 is the objective term for the SWAP error defined in (4.9) and (4.15). We are interested in the trends of these terms, rather than their exact values. As we increase the run time, the objective value of the clusterer decreases. This pattern is more evident for the SWAP fidelity metric, likely because this term is quadratic and requires more time to optimize effectively. The qubit fidelity metric is linear and thus easier to optimize. The qubit error initially increases with time, due to the balance of optimizing the SWAP errors. With more iterations, the optimizer finds high fidelity qubits while retaining a sub-topology with lower SWAP errors.

When evaluating the placer, the time limit is applied globally so that each individual optimization problem is allocated an equal fraction of the overall maximum time.

In Fig. 6.3, we plot the dependence of the placer on the assignment produced by

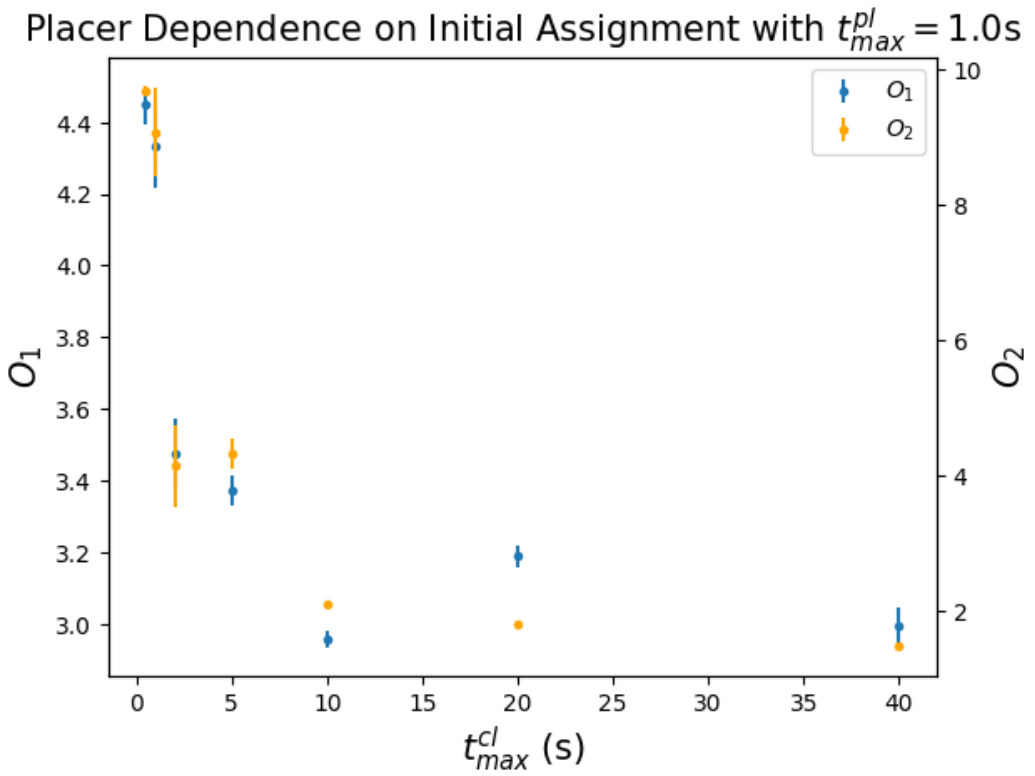


Figure 6.2: Plot of placer objective value for different maximum time constraints. The placer was initialized with the final assignment produced by the clusterer in Fig. 6.1.

the clusterer by varying the time allowed for the clusterer to produce an assignment while using a fixed time limit for the placer. The graph exhibits a strong dependence. Combined with the weak trend in Fig. 6.2, this shows that the placer is best optimized by allocating more time to the clusterer for small circuits. As the number and size of circuits on a fixed machine increases, so does the complexity of the placing problem. In this case, more time should be allocated to the placer.

The benefit from increasing clusterer runtime drops off around 10-20s when compiling for a 127-qubit machine. Despite seeking greater optimality, using integer programming for mapping is viable for current NISQ devices. Additionally, our approach provides flexibility to users. Circuits whose results are mission-critical can be compiled for longer time to achieve greater optimality. Circuits that must be executed in real time (e.g. iterative quantum programs), can balance optimality and

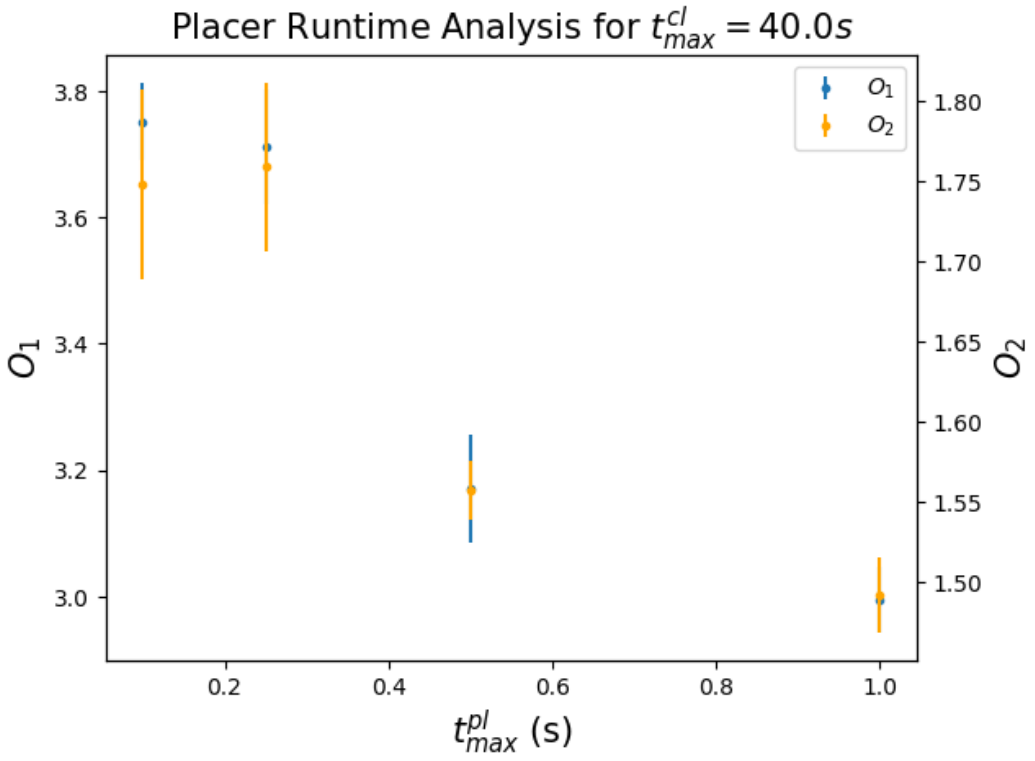


Figure 6.3: Plot of placer objective value for different maximum clusterer time constraints. We fixed the placer time constraint at 2s.

timeliness by stopping optimization before the level-off threshold.

6.1.2 Evaluation on IBMQ Machines

Circuit Cutting

Circuit cutting is an error mitigation technique for the NISQ-era introduced by [24] and implemented as the CutQC library by [29]. The method splits a circuit into two subcircuits. The upstream circuit must be measured in 3 different bases (Z,X,Y) and the downstream circuit must be initialized in 4 different states ($|0\rangle, |1\rangle, |+\rangle, |i\rangle$). This results in 7 subcircuits that must be run. The motivation is to allow circuits to be run on smaller machines. We use this technique as a benchmark for our mapping algorithm and to explore the optimization possibilities of placing multiple smaller circuits over one large circuit.

Peham et al. [23] showed that search space selection is very important for circuit mapping as it bounds the optimality of mappings. We explore the possibility that mapping several smaller circuits can achieve better fidelity than mapping a single larger circuit.

To evaluate the use of parallelism with circuit cutting, we used the example from Tang et al. [29] which has a width of 5 qubits and is split into two 3-qubit circuits after cutting. We designed and ran three experiments using our mapping algorithm and Qiskit's algorithm.

- Full Circuit: We placed a single instance of the full circuit and ran it for 4000 shots. This is our baseline for evaluation.
- Parallel Subcircuits: We placed seven 3-qubit circuits corresponding to the different measurement and initialization bases and ran for 4000 shots. This experiment evaluates parallelism for circuit cutting.
- Parallel Full Circuit: We placed four instance of the full circuit an ran it for 1000 shots. This simulates the same throughput as the subcircuit experiment, but requires fewer shots. We use this to evaluate the use of parallelism for reducing the number of shots needed to obtain same-fidelity results.

The parameters and results are shown in Table 6.1. Qiskit identified better mappings in all experiments. The circuits used here have few qubits and gates, allowing greedy gate scheduling methods to find near-optimal mappings. Between experiments, the single circuit case had less error than the parallel circuit case. Adding more circuits introduces more competition for high-fidelity qubits and gates. Thus the parallel case is more error-prone.

However, both experiments were outperformed by the parallelized subcircuits. By splitting the circuit, we reduce the gate and qubit count which increases fidelity [24, 29]. By mapping the subcircuits in parallel, we reduce the fidelity as seen with the full-circuit experiments. The parallel subcircuit experiment shows that the

Machine	$\alpha_{\mathbf{cl}}$				$\alpha_{\mathbf{pl}}$		$\mathbf{t^{cl}_{max}}(s)$	$\mathbf{t^{pl}_{max}}(s)$
Brisbane	2	1	1	-0.1	1	1	20	1

Name	**Method**	$\mathbf{O_{cl}}$		$\mathbf{O_{pl}}$		ϵ
Single Circuit	BINLP	-0.5	-0.5	0.10	0.11	0.04575
	Qiskit	n/a	-0.48	n/a	0.10	0.02349
Parallel Circuit	BINLP	-1.86	-1.86	0.45	0.47	0.42079
	Qiskit	n/a	0.46	n/a	0.51	0.37069
Subcircuits	BINLP	-2.09	-2.09	0.34	0.36	0.00357
	Qiskit	n/a	2.17	n/a	0.48	**0.00325**

Table 6.1: Experiment data comparing circuit cutting vs. parallel execution vs. single execution. Executed on IBM Brisbane. O_{cl} and O_{pl} are shown pre-transpilation (left subcolumn) and post-transpilation (right subcolumn).

fidelity gain from cutting outweighs the loss from parallelization. This has important ramifications for cicuit scheduling as quantum computers become larger.

QAOA Ansatz

Quantum Approximate Approximation Algorithms (QAOAs) construct a parameterized *ansatz* circuit which is executed iteratively. At each iteration, the parameters are updated based on the results of the ansatz. The ansatz is created so that optimizing the output of the ansatz solves a desired problem, such as MaxCut [2].

Much work has been done to parallelize gates within QAOA ansatz circuits [1, 12, 13]. However parallelization by running multiple ansatzes simultaneously has not been explored. We use an ansatz to solve MaxCut for the graph in Fig. 6.4.

QAOAs use especially dense circuits, which makes the mapping generation crucial to successful execution. We initialized the ansatz five times with random parameters and mapped the instances on a parallel circuit. We evaluated on IBM Brisbane and Kyoto and tested α's of -0.1 and -1 for the contiguity objective. Due to the high connectivity of the QAOA ansatz, greater contiguity is crucial to reduce the number of SWAPs that must be inserted. We optimized one iteration of the ansatzes with $t^{cl}_{max} = 10, 20, 30$ and $t^{pl}_{max} = 2s$. We summarize the results in table 6.2.

We observe that our algorithm outperforms Qiskit's transpiler for ansatz exe-

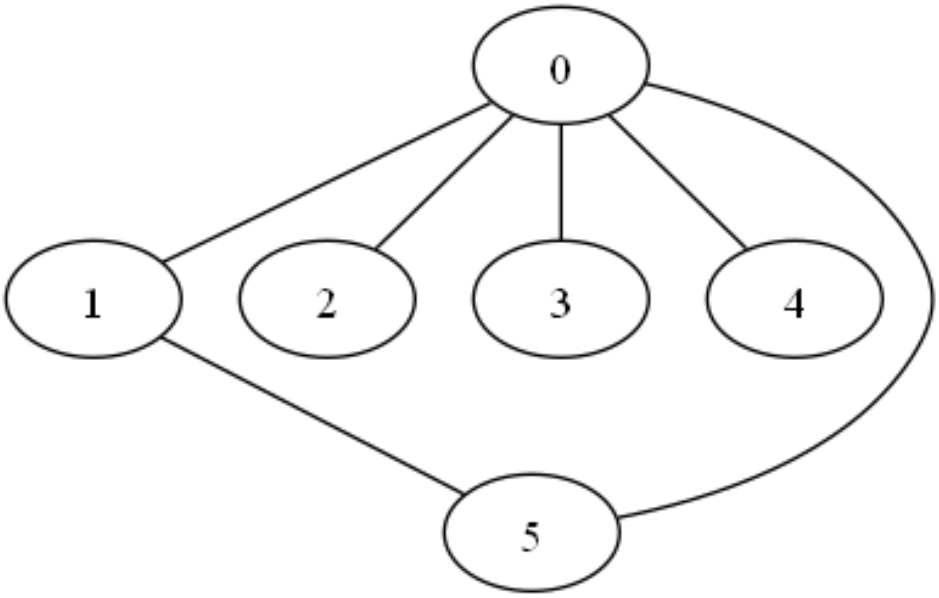


Figure 6.4: Input graph for the MaxCut problem targeted by the QAOA ansatz benchmark.

Machine	$\alpha_{\mathbf{cl}}$				$\alpha_{\mathbf{pl}}$		$\mathbf{t}^{\mathbf{pl}}_{\mathbf{max}}(s)$
IBM Brisbane	5	1	1	-0.1	2	1	2
IBM Brisbane	5	1	1	-1	2	1	2
IBM Kyoto	5	1	1	-0.1	2	1	2

$\mathbf{t}^{\mathbf{cl}}_{\mathbf{max}}(s)$	$\mathbf{t}(s)$	$\mathbf{O_{cl}}$		$\mathbf{O_{pl}}$		$\epsilon_{\mathbf{min}}$	$\epsilon_{\mathbf{max}}$	$\epsilon_{\mathbf{avg}}$
10	**33.8**	1.6	1.4	6.9	9.2	0.012	0.084	0.039
20	42.0	-0.1	-0.5	6.5	7.4	0.008	0.074	**0.035**
30	52.7	-1.0	-1.1	5.9	**7.2**	0.007	0.095	0.038
Qiskit	106.9	n/a	22.8	n/a	22.5	0.012	0.152	0.079
10	**32.7**	-44.2	-44.2	7.2	8.9	0.006	0.077	**0.035**
20	45.1	-44.2	-44.2	7.2	8.8	0.014	0.082	0.040
30	56.1	-46.1	-46.1	6.2	**7.1**	0.006	0.091	0.039
Qiskit	79.4	n/a	-8.3	n/a	18.8	0.069	0.249	0.162
10	**9.7**	-1.2	-1.2	5.8	6.1	0.004	0.101	0.042
20	37.2	-1.6	-1.6	5.2	5.5	0.008	0.089	0.041
30	43.3	-1.9	-1.9	4.8	**5.0**	0.009	0.087	**0.040**
Qiskit	91.0	n/a	14.0	n/a	18.8	0.024	0.129	0.079

Table 6.2: Results for QAOA experiments. The top table shows machine information and parameter values for each experiment. The lower table shows experiment results.

cution and requires less time to run. Results are consistent across α parameters, contrary to our initial expectations. Since the ansatzes were only 6 qubits, contiguity is easier to optimize. This term may be crucial for larger circuits.

Deep Circuit Evaluation

In Section 6.1.1, we evaluated the runtime-optimality trade-off for the circuit cutting example from Tang et al. [29]. Those circuits were shallow with only 3 and 5 qubits, and Qiskit was able to identify a better mapping. In order to compare our algorithms on more complex circuits, we designed two stress tests using deep synthetic and benchmark circuits.

- RevLib [30]: RevLib is library of reversible benchmarks for quantum computing. We used three circuits described in Table 6.3. Since RevLib circuits are not in the QASM format, we used Real2QASM [3] to convert to QASM.
- Queko [28]: Queko generates circuits with a specified gate density, depth, and topology. We generated 4 circuits with Queko using the TFL preset density, depths of 100 and 200, and two subtopologies derived from IBM Brisbane shown in Fig. 6.5.

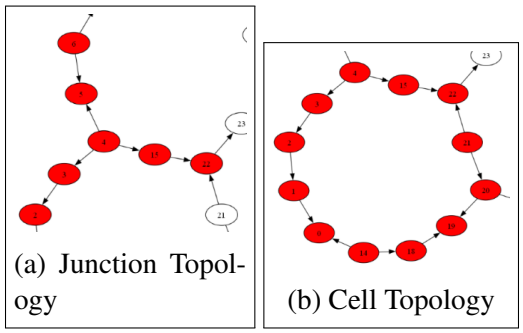
Table 6.4 summarizes the results for the depth test. Our mapping algorithm matches and exceeds the fidelity of Qiskit's algorithm. The fidelity does not have a clear trend with respect to the maximum optimizer runtime. The separation of the clustering and placing steps means that a clustering with a slightly better O_{cl} score may not lead to a better O_{pl} score.

6.2 Job Scheduling

To evaluate our job schedules, we use three metrics:

Name	Width	Depth
aj-e11_165	4	83
4gt4-v0_72	5	103
sys6-v0_111	10	82
sys6-v0_111	10	82

Table 6.3: RevLib circuits used in the throughput stress test. We used sys6-v0_111 twice.



(a) Junction Topology

(b) Cell Topology

Figure 6.5: Subtopologies of IBM Brisbane used to generate circuits using QUEKO for the throughput stress test. Used TFL density vectors and depths of 100 and 200.

Source	Machine	α_{cl}				α_{pl}		$t^{pl}_{max}(s)$
RevLib	Kyoto	10	1	1	-0.5	3	1	2
Queko	Brisbane	10	3	10	-0.1	1	3	2

$t^{cl}_{max}(s)$	$t(s)$	O_{cl}		O_{pl}		ϵ_{min}	ϵ_{max}	ϵ_{avg}
10	39.6	-15.6	-14.5	27.7	25.3	0.963	0.997	0.988
20	57.8	-15.6	-14.5	27.7	27.2	0.694	1.001	**0.912**
40	75.4	-15.6	-14.5	27.7	28.3	0.976	1.001	0.992
80	108	-18.5	-18.5	19.3	**19.5**	0.958	1.001	0.988
Qiskit	**32.0**	n/a	-10.9	n/a	34.9	0.946	1.001	0.982
10	49.4	24.6	24.5	85.1	61.4	0.972	1.000	0.990
20	49.0	21.8	35.0	81.0	67.4	0.993	1.001	0.997
40	75.5	21.1	37.9	80.4	72.8	0.982	1.001	0.994
80	112	21.1	27.4	80.4	71.6	0.978	1.000	0.993
Qiskit	**27.3**	n/a	23.2	n/a	**52.9**	0.973	1.001	**0.989**

Table 6.4: Results for the density test. The top table shows experiment setup information. The bottom table shows expermient results.

- Probability of a Successful Trial (PST): We used benchmark circuits from Revlib [30] which ideally measure to a fixed bit string b with probability 1. For a noisy machine, $PST = \frac{\#\text{ b occurrences}}{s}$. We want to maximize PST.
- Trial Reduction Factor (TRF) [14]: $TRF = \frac{\sum s_i}{\#\text{ shots ran}}$. The numerator is the number of shots needed for a sequential workload. TRF measures the factor by which we reduced the total shot count. We want to maximize TRF.
- Time Reduction Factor (TiRF): $TiRF = \frac{\sum t_i}{\sum t_c}$. Similar to TRF, this measures the time needed to run each job sequentially divided by the total time spent on all computers. We can estimate this from the circuits and computer calibration data. We want to maximize the TiRF.
- Time Inflation Factor (TiIF): $TiIF = \frac{1}{N}\sum_{i=1}^{N} \frac{t_i^{(0)}}{t_i}$ where t_i is the runtime of job i in the schedule and $t_i^{(0)}$ is the runtime of job i in a serial schedule. We want to maximize the TiIf.

We evaluated our machines using small and large queues. The small queue allows us to run our circuits on real machines to evaluate the PST, TiRF, and TRF. The large queue allows us to test the scalability of our system on complex workloads and evaluate the TRF and TiRF.

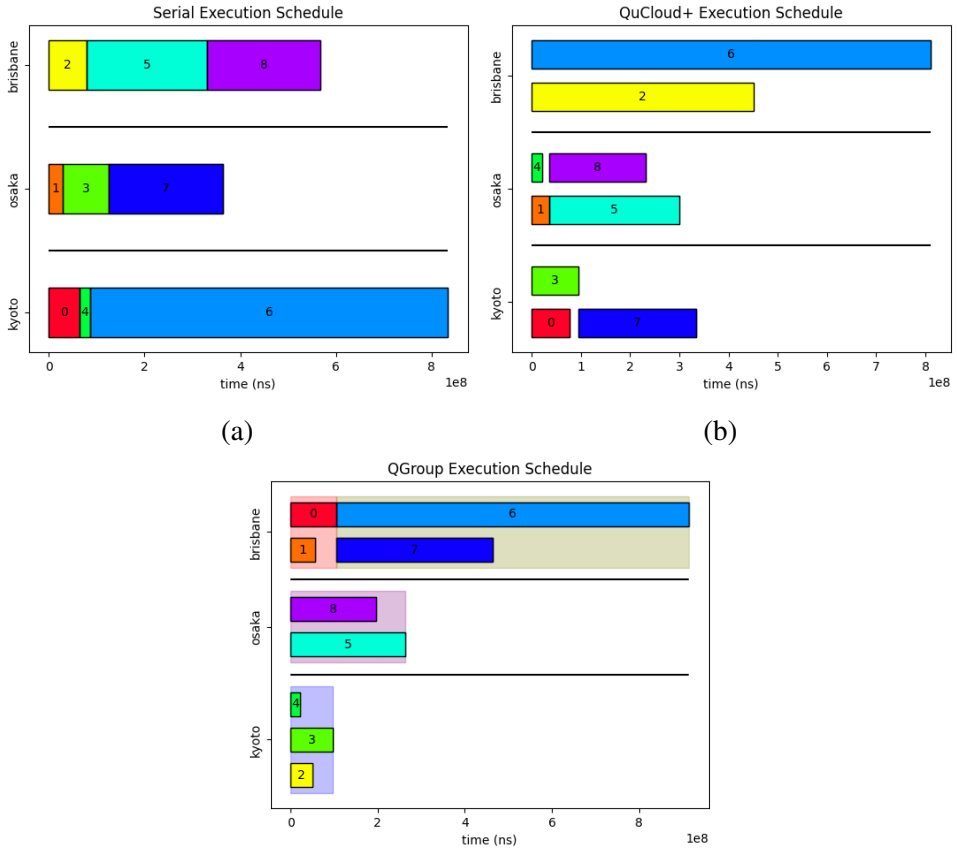
For Algorithm 2, we used (5.4) for d with $\gamma = 0.1$. For Algorithm 3, we used $\eta = 1$ since none of the schedules came close to using all qubits. We implemented the scheduler from QuCloud+ with $\epsilon = 0.15$

6.2.1 Real Machine Evaluation

We evaluated the scheduling algorithms using the Brisbane, Kyoto, and Osaka IBM computers. The circuits and shot counts used in the short queue are shown in Table 6.5. We chose circuits with a variety of depths to test the efficacy of our grouping algorithm.

Num	Name	Width	Depth	Shots
0	3_17_13	3	25	4000
1	4mod5-v1_22	5	13	2000
2	mod5mils_65	5	22	2500
3	alu-v0_27	5	24	5000
4	decod24-v2_43	4	34	1000
5	aj-e11_165	4	83	4500
6	sf_276	5	367	4500
7	sym9_146	12	140	2000
8	4gt4-v0_72	5	130	3000

Table 6.5: Revlib circuits used in the short queue with the number of shots.



(a)

(b)

(c) b

Figure 6.6: Execution schedule for each scheduling method. Black horizontal lines separate execution on each computer. The numbering and coloring of jobs is consistent across each chart. Groups are shown by the shaded backgrounds for QGroup.

Algorithm	**TRF**	**TiRF**	**TiIF**	$\mu_{\mathbf{PST}}$
Serial	1.0	1.0	1.0	0.291
QuCloud+	**1.583**	1.22	0.848	**0.224**
QGroup	**1.583**	**1.38**	**0.937**	0.210

Table 6.6: Results for the small queue evaluation. Best values are bolded. For PST, we ignore the serial algorithm because we expect the serial PST to be higher than the parallel case. The same holds for TiIF.

Fig. 6.6 shows the execution schedules generated by each algorithm with evaluation results summarized in Table 6.6. Our method matched the TRF with QuCloud+. Both our algorithms attempt to maximize parallelism, thus maximizing the TRF. However, our TiRF was higher than QuCloud+. Our algorithm was able to better able to use the depths and runtime estimates of the circuits to minimize the makespan. The PST for QGroup with lower than that of QCloud+, though still competitive.

Our algorithm does far better in reducing TiIF than QuCloud+. This is most easily seen in job 2 in Fig. 6.6b. This job takes far longer to run than in either other schedules. This is because job 6 takes a lot longer to run per shot than job 2. Then is slows down job 2. QGroup directly minimizes this induced slowdown to improve per-job runtime.

6.2.2 High Throughput Simulation

The large queue consists of 2,750 circuits with an average width of 2.34 and average depth of 32.76. These circuits are very small and thus often have identical runtimes due to using the exact same gates albeit with different parameters.

We set $\beta = -0.01$ to prevent massive groups of jobs from forming. We used a window of $W = 25$. We used all 17 IBM backends to simulate the scheduling process, though we did not have access to run on the machines. The results from scheduling and running the jobs on simulators are presented in Table 6.7.

QuCloud+ achieves a higher TRF and TiRF, but QGroup again has a higher TiIF

Algorithm	**TRF**	**TiRF**	**TiIF**
Serial	1.0	1.0	1.0
QuCloud+	**1.59**	**5.65**	0.964
QGroup	1.33	4.42	**0.99**

Table 6.7: Results for the large queue simulation.

score. The workload contains many small circuits. Since QuCloud+ determines its groups based on EPST rather than runtime, it is better at scheduling many small circuits in parallel. However, this leads to increased runtimes for some circuit as shown by the worse TiIF metric.

Chapter 7

DISCUSSION AND CONCLUSION

7.1 Qubit Mapping

We present a novel SWAP matrix heuristic to better model error probabilities of SWAP gates. We incorporate this with qubit fidelity information to construct a binary integer non-linear programming problem that optimizes circuit mapping and qubit placement. We use the Gurobi optimizer [**Gurobi**] and evaluate our approach vs. the Qiskit compiler using examples from circuit cutting [29], QAOAs [2], RevLib [3, 30], and Queko [28].

For the throughput benchmarks (RevLib and Queko), we observe that our algorithm is able to match and, for the RevLib experiment, exceed Qiskit's performance. For the QAOA benchmark, our algorithm achieves lower MSE than Qiskit by factor of $\times 1.88$ and a runtime improvement of $\times 1.42$.

For the case of circuit cutting, we found that executing subcircuits in parallel maintains improved fidelity compared to single or parallel execution of the full circuit. Our results extend the improvements of circuit cutting [24, 29] into parallel execution. This also confirms the analysis of Peham et al. [23] that choosing good search spaces is important for optimization purposes. We can place the smaller subcircuits with more flexibility when compared to the full circuit. This result indicates that throughput maximization in the NISQ era will depend on splitting circuits and allocating more qubits to individual jobs, rather than executing multiple distinct circuits in parallel.

Finally, our algorithm exhibits the early-termination property and allows flexibility in objective terms so that optimization can be tailored to specific circuits. Early-termination allows time resources to be explicitly allocated when balancing

fidelity and runtime. The α weights can be used to assign more emphasis to e.g. contiguity of circuit assignments for dense circuits such as QAOA.

7.2 Job Scheduling

We present a parallel job scheduling algorithm, QGroup, focused on optimizing the execution makespan. We use dynamic programming to group circuits by similar run time. This ensures there is no runtime performance loss when running circuits in parallel. We use linear programming to optimize the schedule of each job group. This process minimizes the makespan and reduce the number of changes to the workload. We use this to reduce the overhead induced by compilation and error mitigation. Finally, we use these algorithms to assign job groups to the machines with the best fidelity. QGroup is parameterized to provide flexibility for different circuit loads and sizes.

We evaluated our algorithm against QuCloud+ and a naive serial scheduler. Our method succeeds in reducing the time-makespan and runtime of individual jobs, achieving higher TiRF and TiIF values. Our method sacrifices some fidelity to increase the parallelism.

We evaluate our algorithm on a benchmark of 2,750 small circuits. Our method is able to achieve near-perfect TiIF values, ensuring that no jobs are significantly slowed down by parallelism.

Chapter 8

QUESTIONNAIRE

BIBLIOGRAPHY

[1] Mahabubul Alam, Abdullah Ash-Saki, and Swaroop Ghosh. "Circuit Compilation Methodologies for Quantum Approximate Optimization Algorithm". In: *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2020, pp. 215–228. DOI: 10.1109/MICRO50266.2020.00029.

[2] Kostas Blekos, Dean Brand, Andrea Ceschini, Chiao-Hui Chou, Rui-Hao Li, Komal Pandya, and Alessandro Summer. *A Review on Quantum Approximate Optimization Algorithm and its Variants*. 2023. arXiv: 2306.09198 [quant-ph].

[3] Kaun-Yu Chang and Chun-Yi Lee. "Mapping Nearest Neighbor Compliant Quantum Circuits onto a 2-D Hexagonal Architecture". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2021), pp. 1–1. DOI: 10.1109/TCAD.2021.3127868.

[4] V. Chaudhary and J.K. Aggarwal. "A generalized scheme for mapping parallel algorithms". In: *IEEE Transactions on Parallel and Distributed Systems* 4.3 (1993), pp. 328–346. DOI: 10.1109/71.210815.

[5] E. G. Coffman Jr., M. R. Garey, and D. S. Johnson. "An Application of Bin-Packing to Multiprocessor Scheduling". In: *SIAM Journal on Computing* 7.1 (1978), pp. 1–17. DOI: 10.1137/0207001. eprint: https://doi.org/10.1137/0207001. URL: https://doi.org/10.1137/0207001.

[6] Poulami Das, Swamit S. Tannu, Prashant J. Nair, and Moinuddin Qureshi. "A Case for Multi-Programming Quantum Computers". In: *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO '52. Columbus, OH, USA: Association for Computing Machinery, 2019, pp. 291–303. ISBN: 9781450369381. DOI: 10.1145/3352460.3358287. URL: https://doi.org/10.1145/3352460.3358287.

[7] Rudolf Fleischer and Michaela Wahl. “Online Scheduling Revisited”. In: *Algorithms - ESA 2000*. Ed. by Mike S. Paterson. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 202–210. ISBN: 978-3-540-45253-9.

[8] Gambetta Jay, IBM Quantum. *The hardware and software for the era of quantum utility is here*. 2023. URL: https://research.ibm.com/blog/quantum-roadmap-2033.

[9] R. L. Graham. “Bounds for certain multiprocessing anomalies”. In: *The Bell System Technical Journal* 45.9 (1966), pp. 1563–1581. DOI: 10.1002/j.1538-7305.1966.tb01709.x.

[10] Gurobi Optimization, LLC. *Gurobi Optimizer Reference Manual*. 2023. URL: https://www.gurobi.com.

[11] Xin Huang and Pinyan Lu. “An Algorithmic Framework for Approximating Maximin Share Allocation of Chores”. In: *Proceedings of the 22nd ACM Conference on Economics and Computation*. EC ’21. Budapest, Hungary: Association for Computing Machinery, 2021, pp. 630–631. ISBN: 9781450385541. DOI: 10.1145/3465456.3467555. URL: https://doi.org/10.1145/3465456.3467555.

[12] Ayse Kotil, Fedor Simkovic, and Martin Leib. *Improved Qubit Routing for QAOA Circuits*. 2023. arXiv: 2312.15982 [quant-ph].

[13] Wolfgang Lechner. “Quantum Approximate Optimization With Parallelizable Gates”. In: *IEEE Transactions on Quantum Engineering* 1 (2020), pp. 1–6. DOI: 10.1109/TQE.2020.3034798.

[14] Lei Liu and Xinglei Dou. “QuCloud+: A Holistic Qubit Mapping Scheme for Single/Multi-programming on 2D/3D NISQ Quantum Computers”. In: 21.1 (Jan. 2024). ISSN: 1544-3566. DOI: 10.1145/3631525. URL: https://doi.org/10.1145/3631525.

[15] Prakash Murali, Jonathan M. Baker, Ali Javadi-Abhari, Frederic T. Chong, and Margaret Martonosi. “Noise-Adaptive Compiler Mappings for Noisy Intermediate-Scale Quantum Computers”. In: ASPLOS ’19. Providence, RI, USA: Association for Computing Machinery, 2019, pp. 1015–1029. ISBN: 9781450362405. DOI: 10.1145/3297858.3304075. URL: https://doi.org/10.1145/3297858.3304075.

[16] Giacomo Nannicini, Lev S. Bishop, Oktay Günlük, and Petar Jurcevic. “Optimal Qubit Assignment and Routing via Integer Programming”. In: *ACM Transactions on Quantum Computing* 4.1 (Oct. 2022). DOI: 10.1145/3544563. URL: https://doi.org/10.1145/3544563.

[17] Siyuan Niu and Aida Todri-Sanial. “Enabling Multi-programming Mechanism for Quantum Computing in the NISQ Era”. In: *Quantum* 7 (Feb. 2023), p. 925. ISSN: 2521-327X. DOI: 10.22331/q-2023-02-16-925. URL: https://doi.org/10.22331/q-2023-02-16-925.

[18] Siyuan Niu and Aida Todri-Sanial. “How parallel circuit execution can be useful for NISQ computing?” In: *Proceedings of the 2022 Conference & Exhibition on Design, Automation & Test in Europe*. DATE ’22. Antwerp, Belgium: European Design and Automation Association, 2022, pp. 1065–1070. ISBN: 9783981926361.

[19] Yasuhiro Ohkura, Takahiko Satoh, and Rodney Van Meter. “Simultaneous Execution of Quantum Circuits on Current and Near-Future NISQ Systems”. In: *IEEE Transactions on Quantum Engineering* 3 (2022), pp. 1–10. DOI: 10.1109/TQE.2022.3164716.

[20] Aaron Orenstein and Vipin Chaudhary. “Quantum Circuit Mapping Using Binary Integer Nonlinear Programming”. In: *International Parallel and Distributed Processing Symposium Workshops* (2024).

[21] Anabel Ovide, Santiago Rodrigo, Medina Bandic, Hans Van Someren, Sebastian Feld, Sergi Abadal, Eduard Alarcon, and Carmen G. Almudever. "Mapping quantum algorithms to multi-core quantum computing architectures". In: (Mar. 2023). DOI: 10.1109/ISCAS46773.2023.10181589. arXiv: 2303.16125 [quant-ph].

[22] P. Jurcevic and D. Zajac and J. Stehlik and I. Lauer and R. Mandelbaum. *Pushing quantum performance forward with our highest Quantum Volume yet*. 2022. URL: https://research.ibm.com/blog/quantum-volume-256.

[23] Tom Peham, Lukas Burgholzer, and Robert Wille. "On Optimal Subarchitectures for Quantum Circuit Mapping". In: *ACM Transactions on Quantum Computing* 4.4 (July 2023). DOI: 10.1145/3593594. URL: https://doi.org/10.1145/3593594.

[24] Tianyi Peng, Aram W. Harrow, Maris Ozols, and Xiaodi Wu. "Simulating Large Quantum Circuits on a Small Quantum Computer". In: *Phys. Rev. Lett.* 125 (15 Oct. 2020), p. 150504. DOI: 10.1103/PhysRevLett.125.150504. URL: https://link.aps.org/doi/10.1103/PhysRevLett.125.150504.

[25] Quazzafi Rabbani, Aamir Khan, and Abdul Quddoos. "Assignment of multiple jobs scheduling to a single machine". In: *Advances in Mathematics Scientific Journal* 10 (Feb. 2021), pp. 1003–1011. DOI: 10.37418/amsj.10.2.29.

[26] R. Seidel. "On the All-Pairs-Shortest-Path Problem in Unweighted Undirected Graphs". In: *Journal of Computer and System Sciences* 51.3 (1995), pp. 400–403. ISSN: 0022-0000. DOI: https://doi.org/10.1006/jcss.1995.1078. URL: https://www.sciencedirect.com/science/article/pii/S0022000085710781.

[27] Khaldoun Senjab, Sohail Abbas, Naveed Ahmed, and Atta ur Rehman Khan. "A survey of Kubernetes scheduling algorithms". In: *Journal of Cloud Computing* 12.1 (June 2023), p. 87. ISSN: 2192-113X. DOI: 10.1186/s13677-023-00471-1. URL: https://doi.org/10.1186/s13677-023-00471-1.

[28] Bochen Tan and Jason Cong. “Optimality Study of Existing Quantum Computing Layout Synthesis Tools”. en. In: *IEEE Transactions on Computers* (July 2020). DOI: 10.1109/TC.2020.3009140. arXiv: 2002.09783 [quant-ph].

[29] Wei Tang, Teague Tomesh, Martin Suchara, Jeffrey Larson, and Margaret 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*. ASPLOS ’21. Virtual, USA: Association for Computing Machinery, 2021, pp. 473–486. ISBN: 9781450383172. DOI: 10.1145/3445814.3446758. URL: https://doi.org/10.1145/3445814.3446758.

[30] Robert Wille, Daniel Große, Lisa Teuber, Gerhard W Dueck, and Rolf Drechsler. “RevLib: An online resource for reversible functions and reversible circuits”. In: *38th International Symposium on Multiple Valued Logic (ismvl 2008)*. IEEE. 2008, pp. 220–225.