AutoBraid: A Framework for Enabling Efficient Surface Code Communication in Quantum Computing

Fei Hua
Rutgers University
huafei90@gmail.com

Yanhao Chen
Rutgers University
chenyh64@gmail.com

Yuwei Jin
Rutgers University
yj243@scarletmail.rutgers.edu

Chi Zhang
University of Pittsburgh
raymond.chizhang@gmail.com

Ari Hayes
Rutgers University
arihayes@gmail.com

Youtao Zhang
University of Pittsburgh
zhangyt@cs.pitt.edu

Eddy Z. Zhang
Rutgers University
eddy.zhengzhang@gmail.com

ABSTRACT

Quantum computers can solve problems that are intractable using the most powerful classical computer. However, qubits are fickle and error prone. It is necessary to actively correct errors in the execution of a quantum circuit. Quantum error correction (QEC) codes are developed to enable fault-tolerant quantum computing. With QEC, one logical circuit is converted into an encoded circuit.

Most studies on quantum circuit compilation focus on NISQ devices which have 10-100 qubits and are not fault-tolerant. In this paper, we focus on the compilation for fault-tolerant quantum hardware. In particular, we focus on optimizing *communication parallelism* for the *surface code* based QEC. The execution of surface code circuits involves non-trivial geometric manipulation of a large lattice of entangled physical qubits. A two-qubit gate in surface code is implemented as a virtual "pipe" in space-time called a *braiding* path. The braiding paths should be carefully routed to avoid congestion. Communication between qubits is considered the major bottleneck as it involves scheduling and searching for simultaneous paths between qubits. We provide a framework for efficiently scheduling braiding paths. We discover that for quantum programs with a *local parallelism* pattern, our framework guarantees an optimal solution, while the previous greedy-heuristic-based solution cannot. Moreover, we propose an extension to the local parallelism analysis framework to address the communication bottleneck. Our framework achieves *orders of magnitude* improvement after addressing the communication bottleneck.

CCS CONCEPTS

• **Software and its engineering → Compilers**; • **Hardware → Quantum error correction and fault tolerance**;

ACM Reference Format:
Fei Hua, Yanhao Chen, Yuwei Jin, Chi Zhang, Ari Hayes, Youtao Zhang, and Eddy Z. Zhang. 2021. AutoBraid: A Framework for Enabling Efficient Surface Code Communication in Quantum Computing. In *MICRO'21: 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '21), October 18–22, 2021, Virtual Event, Greece.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3466752.3480072

1 INTRODUCTION

Quantum computing has significant theoretical advantages over classical computing for applications in decryption, simulation, and optimizations. In some cases, it provides exponential speedup, for instance, for the quantum fourier transformation and quantum phase estimation applications. However, the major reason that prevents theoretical advantages from being realized is that quantum hardware is error prone. Qubits and gates are subject to decoherence and operation errors.

To run a quantum program reliably, it is necessary to detect and correct errors in the circuit. A prominent quantum error correction code (QEC) is *surface code*. Surface code yields one of the highest fault-tolerant threshold error rates. According to *Threshold Theorem*, given any QEC, as long as the physical error rate is lower than *a threshold* [3], a quantum circuit can run with an arbitrarily low logical error rate provided that there are enough physical qubits. Different QECs tolerate different threshold error rates. Surface code QEC can tolerate up to 1% physical error rate while most other QEC cannot. For instance, the *[7-1-3]* error correction code [3] has a threshold error rate of 10^{-6}. Current physical qubit implementation has a physical error rate of 0.1%-1%. Surface code is a QEC that makes near term fault-tolerant quantum computing possible.

A fault-tolerant quantum computer equipped with surface code performs computation in a software defined manner. The quantum hardware only needs to prepare a large lattice of entangled physical qubits. Computation is performed through geometric manipulation of measurement qubits. A logical qubit is encoded using a set of physical qubits. A two-qubit gate requires communication between two logical qubits, which is implemented as a virtual "pipe" in space-time and referred to as a *braiding path* [10]. At every time point, simultaneous braiding paths shall not cross. An example for non-intersecting braiding paths for concurrent two-qubit gates is shown in Fig. 1.

Surface code imposes challenges on both micro-architecture design and program compilation. As a logical qubit is usually encoded with a large number of physical qubits, tremendous number of instructions are generated during the execution of an encoded circuit. Tannu *et al.* [24] exploit the similarity of QEC instructions and significantly reduce the instruction footprint through micro-controller design. On the other hand, when compiling a quantum program into its encoded form, it must produce a schedule of braiding paths in the hardware lattice, such that simultaneous paths do not intersect. And we want to schedule as many braiding paths as possible in short time period, in order to minimize the overall circuit latency.

In this paper, we focus on the compilation of quantum circuits into their encoded form for the surface code mode. Most existing studies for quantum circuit compilation are for non-fault-tolerant hardware. A large body of studies [5, 14, 18, 23, 25–27, 29, 30] are on noisy intermediate-scale quantum (NISQ) devices with donzens of qubits. Even the current NISQ devices do not have sufficient qubits for a fault-tolerant execution, the machine with tremendous computing capability is emerging. IBM is projected to release a 1,121 qubit machine named Condor in the year of 2023.

The most relevant study to our work is the scheduling framework by Javadi-Abhari *et al.* [10]. While its main purpose is to study two different communication modes in surface code: braiding (double-defect code) and teleportation (planar code), it also proposes techniques that automatically schedule braiding paths. Its observation is that braiding-based communication may cause significant routing congestion and delay the circuit execution, making the double-defect code less desirable than the planar code (when physical error rate is low). In this paper, we discovered it is not the inherent nature of braiding-based communication that causes congestion, but the inefficient design of braiding algorithms that causes congestion. Double-defect code tolerates higher error rate with the same number of physical qubits. With double-defect mode, it can have the benefits of minimal physical resource usage and optimized circuit latency at the same time, given proper braiding methods. Our method significantly outperforms this work [10] as we achieve (near) critical path performance for small circuits and orders of magnitude speedup for large-scale circuits.

Another line of relevant research is the routing problem for VLSI circuit design [22]. The qubit braiding problem is similar but different in several ways. The traditional VLSI routing problem determines the paths that connects pins on circuit blocks. It typically optimizes the length of the routing paths. However, the surface code braiding problem is insensitive to the path length [10]. The latency of a braiding operation is constant regardless of the length of the braiding path. Moreover quantum program have different characteristics compared to the VLSI circuit.

In this paper, we perform a systematic exploration of communication scheduling in surface code mode. We develop a framework named *AutoBraid* for analyzing the characteristics of quantum programs and scheduling two-qubit gates in a scalable manner. Our framework mitigates the bottleneck in the scheduling of braiding paths in important quantum programs, and achieves up to orders of magnitude performance improvement. Our contributions are summarized as follows:

- We discover a *local parallelism pattern* for quantum programs, especially for the building block circuits of reversible logical functions. We show how to discover such a pattern and obtain an optimal schedule.
- We discover that *dynamic qubit placement* is important for communication scheduling. While the previous study [10] focuses on static qubit placement, our study for the first time proposes a placement scheme that allows the location of a logical qubit to be dynamically changed throughout the circuit execution, which can tackle the communication bottleneck problem.
- We design and implement a *stack-based path finder* that can efficiently find congestion-free braiding paths and maximize the resource utilization.
- With the *dynamic qubit placement* optimization and the *stack-based path finder*, our framework can efficiently schedule communication for an extensive set of quantum programs. Our framework outperforms the best known work [10], especially for large-scale real-world circuits with up to 5,000 logical qubits and 1,620,000 physical qubits.

The rest of the paper is organized as follows. We introduce the background of quantum error correction in Section 2. We describe our framework in Section 3. Section 4, 5, and 6 are respectively evaluation, related work, and conclusion.

2 BACKGROUND

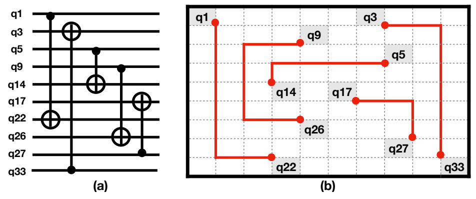


Figure 1: A braiding path must be established between a pair of qubits participating in a CX (two-qubit) gate: (a) A circuit of 5 CX gates that can run concurrently in theory; (b) One possible simultaneous schedule of five braiding paths. (A dot in the lattice represents a logical qubit)

Error correction. Quantum qubits are fickle and can lose their state information due to the interaction with the environment. This phenomenon is called *decoherence*. In addition to decoherence, quantum gates have low fidelity and can cause erroneous outcomes. Readout through the classical devices also has a failure rate. Experiments show an average error rate of 10^{-3} per 100 *ns* for a superconducting quantum device [19].

Quantum error correction codes (QEC) are necessary to ensure reliable execution of quantum programs. QEC detects error syndromes and correct them correspondingly. With a proper single qubit physical error rate, a logical qubit's error rate can be arbitrarily small if encoded using enough physical qubits, due to Threshold Theorem [8].

Why surface code. There are different types of quantum error correction codes. We choose the *surface code* model as our underlying

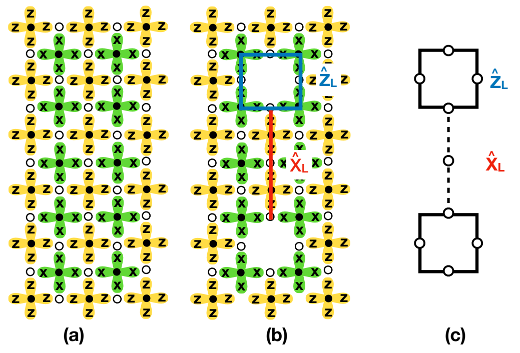


Figure 2: (a) Surface code lattice, (b) a double-defect Z-cut logical qubit, (c) only the nine data qubits of the Z-cut logical in (b) are shown. Data qubits are represented using empty circles, and measurement qubits as dark circles.

platform since it yields one of the highest fault-tolerant threshold of any QEC. It is also one of the widely used error-correction codes [16].

The logical qubit error rate of surface code model can be calculated as follows:

$$P_L = 0.03\left(\frac{p}{p_{th}}\right)^{(d+1)/2} \tag{1}$$

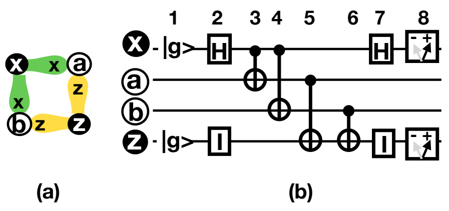
where p_{th} represents the threshold error rate, d represents the strength of error correction (the minimal number of qubits bit flipped or phase flipped in order to define a logical X_L or Z_L operation), p is the physical error rate.

If the physical qubit error rate is < threshold error p_{th}, the logical error rate decreases exponentially with the number of qubits added (d is highly correlated with the total number of physical qubits). Assume we have physical qubit error rate as 0.1% (this is what today's best superconducting quantum devices can achieve), a typical threshold error rate p_{th} as 0.57% (we use the same as that by Fowler *et al.* [6]), a distance d of 55, we can have logical qubit error as low as $P_L = 9.334 \cdot e^{-23}$.

Surface code model. In the surface code model, quantum hardware prepares a large lattice of entangled qubits. An example of a two-dimension lattice of entangled physical qubits is shown in Fig. 2 (a). There are two types of qubits: data qubits and measurement qubits. Measurement qubits are ancilla qubits which can be measured for detecting error syndromes. Data qubits are used for encoding logical qubits.

A logical qubit is created by disabling two same-type measurement qubits, as if creating two defects in the lattice. The data qubits on the boundary of and the link between these two defects represent one logical qubit. If disabling X measurement qubits in the defects, it is called X-cut logical qubit. If disabling Z measurement qubit, it is Z-cut logical qubit. An example of Z-cut qubits is shown in Fig. 2 (b) and (c).

Measuring the ancilla qubits project the data qubits into a simultaneous eigenstate of a set of corresponding stabilizers imposed by the circuit. A basic building block circuit of surface code is shown in Fig. 3 (b). As shown in Fig. 3 (c), depending on the measurement



Z ab	X ab	State of qubits a, b
+1	+1	$(\|00\rangle + \|11\rangle)/\sqrt{2}$
+1	- 1	$(\|00\rangle - \|11\rangle)/\sqrt{2}$
-1	+1	$(\|01\rangle + \|10\rangle)/\sqrt{2}$
-1	- 1	$(\|01\rangle - \|10\rangle)/\sqrt{2}$

(c)

Figure 3: (a) A building block of the surface code lattice, (b) the quantum circuit corresponding to the basic block, and (c) the state of qubits a and b uniquely determined by the measurement outcome X_{ab} and Z_{ab}. The measurement gates corresponding to the 8-th cycle in (b), X_{ab} and Z_{ab} respectively denoted by dark circle with text X and Z inside.

outcome of two measurement qubits X_{ab} and Z_{ab}, the state of two data qubits a and b is uniquely determined. It can be verified by circuit in Fig. 3 (b).

After the first measurement, if there is no error, continuous measurement outcomes of the ancilla qubits X_{ab} and Z_{ab} will be the same. The state of data qubits a and b also remain unchanged. It is as if *stabilizing* the state of the data qubits. There are 8 cycles for running the circuit in Fig. 3 (b), as denoted by the numbers on top of the circuit. The 8 physical cycles denote a surface code cycle. The circuit is repeatedly executed. If there is error, the measurement outcome changes compared with that in previous surface code cycle. We will not discuss the details of error detection and correction as it is not related to our paper. We refer interested readers to the work by Fowler *et al.* [16].

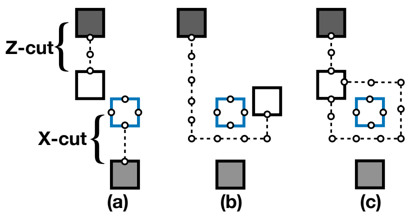


Figure 4: Braiding operation that represents a CNOT between a Z-cut and X-cut logical qubit

Encoded gate operations. In surface code, logical qubit gates such as X and Z can be implemented by applying X and Z gates to a subset of data qubits qubits for the logical qubit, as shown in Fig. 2 (b). Hadamard gate is more complicated, but it can still be applied locally to the logical qubit itself and its surrounding physical qubits.

Two-qubit gate is more complicated. Controlled NOT - CNOT gate is one of the most commonly used two-qubit gates. It is also the only two-qubit gate in an universal gate set. In this paper, we also use CX to denote the CNOT gate. For a CX gate, if the control qubit is 0, the target qubit does not change. If the control qubit is 1, it flips the target qubit. CX between two different types of logical qubits, a Z-cut and X-cut is implemented as a process of "dragging" one defect of a logical qubit around one defect of the other logical qubit. It is called ***braiding***. An example is shown in Fig. 4.

Moving a defect is implemented as turning off and on relevant measurement qubits and their involved circuit operations [16] at different time coordinates. It is not really moving physical qubits. The path of the defect movements over time for one pair of qubits should not intersect that of another pair of qubits. That is, a path is exclusive to one CX operation at one time. Thus the braiding path scheduling problem essentially reduces to a routing problem, as shown in our example in Fig. 1.

Auto-braiding as a routing problem. A lattice is prepared with logical qubits of the same type [10]. It is partitioned into tiles and channels. The partition divides the lattice by function [10]. No circuit or qubit operation is involved in the partition. An example of eight tiles is shown in Fig. 5 (a).

Each tile encloses a logical qubit as well as necessary surrounding physical qubits. For instance, the encoded hadamard gate on a logical qubit q requires the help of the other physical qubits surrounding q. A tile confines an encoded hadamard gate to only physical qubits within the tile.

For implementing a CX gate on two same-type qubits, a dynamically initialized logical qubit is necessary. This dynamically allocated logical qubit will interact with the two original logical qubits in the CX gate. It is initialized within the tile of one logical qubit; One of its defects goes around one defect of this logical qubit tile it is created, then goes around one defect of the other logical qubit, and comes back. An example is shown in Fig. 5 (b). Once the CX gate is completed, the dynamically allocated logical qubit is reclaimed. It is done by turning on corresponding measurement qubits as if erasing the defects. Each tile in the lattice contains enough physical qubits such that an ancilla logical qubit can be created on the fly within it.

As can be seen, to perform a CX gate, a routing path must be established between two tiles and be kept for a certain amount of time. We call the pathway between the tiles as *channels*. The channels are used for "moving" the defects in the braiding process. CX gate is the only gate that requires to use the channels in the lattice in the universal gate set we discuss here.

One important feature of braiding path is that it is latency insensitive. As aforementioned, braiding does not really move qubits. It is essentially turning on/off measurement qubits and their circuit operations. These operations can happen in parallel for different measurement qubits. Thus, regardless how long the braiding path is, the time to perform braiding is always the same [16].

Another important feature of braiding is that it follows topological rules. The braiding can take any path as long as the chosen paths are topologically the same. In surface code braiding, as long as the braiding paths go around the same defects, the effects are equivalent on the circuit. Fig. 5 (b), (c), (d), and (e) represent four topologically equivalent braiding paths for performing CX gate between a pair of logical qubits in P1 and P2.

3 DESIGN OVERVIEW

3.1 Problem Setting

We define a two-dimensional grid where the braiding paths are scheduled. We let each cell in the grid represent a logical qubit tile. We let each vertex represent an intersection of two *channels*. Each edge represents a segment of a channel between two vertices. A braiding path consists of a set of vertices and edges in the grid. Examples are shown in Fig. 5 (f)-(i).

A braiding path is established from any vertex of a cell to any vertex of another cell. There are 16 possible path configurations between two cells with respect to starting and ending vertices. Fig. 5 (f)-(i) represent 4 out of 16 possible combinations between the two cells P1 and P2.

A single-qubit gate applies locally to a cell and does not use any routing vertex/edge. A two-qubit gate requires to establish a path between two operand qubits. Two simultaneous two-qubit gates must have non-intersecting braiding paths. Our goal is to schedule braiding paths and minimize the latency of the encoded circuit.

3.2 Design Considerations

Efficient communication scheduling depends on three factors: the parallelism of CX gates in the program, the placement of the qubits, and the choice of braiding paths.

Inherent Communication Parallelism. The inherent communication parallelism in the circuit matters. In certain circuits, even if there are CX gates, the communication parallelism may be low. An example is shown in Fig. 6 for the Bernstein Varizani (BV) algorithm, where there is no CX parallelism due to gate dependence. When communication parallelism is low, braiding paths can be easily scheduled. In some other circuits, the communication parallelism is high. An example is shown in Fig. 7 for the Ising model circuit where there are $n/2$ simultaneous CX gates and n is the qubit number. For these cases, braiding paths must be scheduled properly to mitigate congestion.

Path Finding. The choice of braiding paths also matters. The greedy algorithm by Javadi Abhari *et al.* [10] finds a shortest path for each pair of qubits, as shortest paths consume minimal routing resources. However, just finding shortest paths is not enough. The scheduler must take a global view, as the path placement for one pair of qubits might affect that for another pair of qubits. Once a path is obtained for one pair of qubits, the vertices used by this path cannot be used by other braiding paths. Even using the same shortest path method for all pairs of qubits, the set of vertices that can be used for routing may vary if the order of path search varies.

We show an example in Fig. 8 (a), if the scheduler finds shortest paths in the order of $\{A_1, A_2\}$, $\{B_1, B_2\}$, and $\{E_1, E_2\}$, then neither $\{C_1, C_2\}$ or $\{D_1, D_2\}$ can find a braiding path as the lattice has been divided into two disconnected components due to the placement of A, B, and E's paths. However, as shown in Fig. 8 (b), if the scheduler finds shortest paths in the order of $\{B_1, B_2\}$, $\{C_1, C_2\}$, $\{D_1, D_2\}$, $\{E_1, E_2\}$, and $\{A_1, A_2\}$, all CX gates can run simultaneously.

Qubit Layout. The layout of the logical qubits on the physical lattice matters. An efficient scheduler must find out a proper layout of the qubits in order to maximally exploit the communication

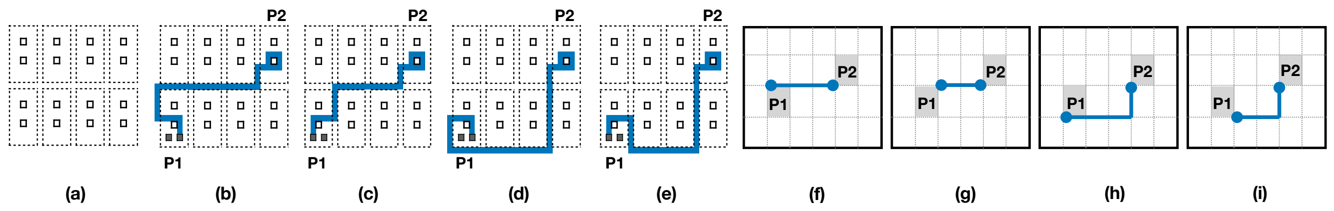


Figure 5: The braiding problem modeled as a routing problem: (a) eight tiles in the original lattice; (b), (c), (d), and (e) show that a braiding path from P1 can start from any corner of P1; (f), (g), (h), and (i) stand for the routing paths in the two-dimensional grid representation we define, respectively for (b), (c), (d), and (e).

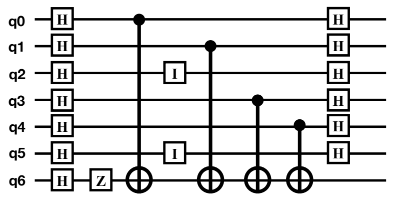


Figure 6: 7-qubit bernstein varizani circuit

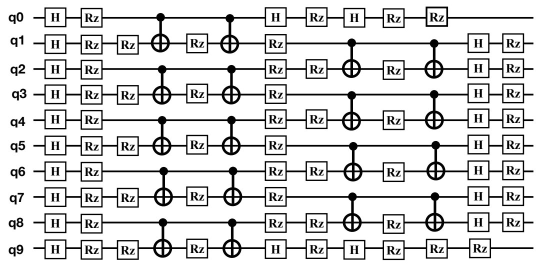


Figure 7: A part of the 10-qubit Ising model circuit

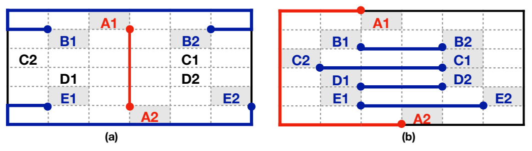


Figure 8: Path search for five CX gates: A, B, C, D, and E. A_1 and A_2 denote two qubits in CX gate A. We use similar notation for other gates; (a) shows that if shortest paths are found in the order of A, B, and E, then two CX gates C and D cannot run; (b) shows that if processing in the order of B, C, D, E, and A, it can find simultaneous paths for all CX gates.

parallelism in the program. We can construct a case where even if there are only four pairs of qubits, no matter how large the lattice is, these four CX gates cannot run simultaneously, unless the qubit layout is changed.

We construct the case as follows. For each CX gate, we draw a straight line between two closest vertices from its two qubits and we denote it as *straight line path*. In the constructed case, for any CX gate, its straight line path separates any other CX gate's two qubits into two sides of the straight line. Further, each qubit is on the boundary of the lattice. With four such pairs of qubits, we construct an example in Fig. 9. We can rigorously prove that four simultaneous braiding paths cannot be scheduled in this case, but we sketch the proof due to space limit.

If we look at the two CX gates $\{A_{0,0}, A_{0,1}\}$ and $\{A_{1,0}, A_{1,1}\}$, at least one CX's braiding path must go around one qubit of the other CX gate. By *going around a qubit q*, we mean a braiding path uses the boundary of the grid, and uses at least one edge of the qubit q on the boundary. This holds because if neither CX's braiding path go around one qubit of the other CX gate, the two braiding paths will cross. Without loss of generality, we assume such a braiding path is for $\{A_{1,0}, A_{1,1}\}$ which goes around the qubit $A_{0,1}$, as shown in blue in Fig. 9 (a). This leaves $A_{0,0}$'s edge on the boundary open. Then the braiding path for $\{A_{0,0}, A_{0,1}\}$ shall not go around any qubit in $A_{1,0}$ or $A_{1,1}$ otherwise it will cross the first braiding path. We draw the braiding path for $\{A_{0,0}, A_{0,1}\}$ in red in Fig. 9 (a). If we try to find the path for $\{A_{2,0}, A_{2,1}\}$, it must go around $A_{0,0}$ otherwise it will cross the first two established paths (the red and blue). Now it is impossible to find path for $\{A_{3,0}, A_{3,1}\}$ since the first three paths have placed $A_{3,0}$ and $A_{3,1}$ into two disconnected partitions of the lattice. And it is proved that, these four CX gates cannot run in parallel regardless how large the lattice is.

However, if we change the qubit layout by switching $A_{3,0}$ with $A_{0,0}$, and $A_{2,0}$ with $A_{1,0}$ as shown in Fig. 9 (b), a congestion-free routing schedule exists for all four CX gates.

The challenge of qubit layout optimization is that there is no one-fit-all qubit placement which satisfies all theoretically concurrent CX gates. At different execution points, different combinations of CX gates are allowed to run concurrently. The study by Javadi-Abhari *et al.* [10] uses a graph partition method to determine an initial qubit placement that places the frequently interacting qubits into as compact regions as possible, but the qubit layout is still fixed throughout the entire circuit. *In our design, we allow the qubit placement to dynamically change throughout the scheduling process, and hence achieves significant flexibility and up to orders of magnitude performance improvement in some cases.*

3.3 AutoBraid Framework

We propose the *AutoBraid* framework as shown in Fig. 10. It performs communication scheduling in three stages.

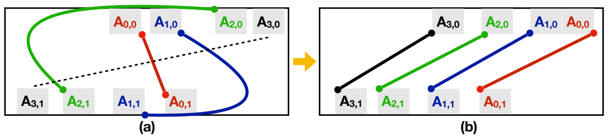


Figure 9: Impact of qubit layout: Four CX gates can run simultaneously in theory; They are $\{A_{i,0}, A_{i,1}\}$ $(i = 0, 1, 2, 3)$; (a) shows a qubit-layout which does not allow concurrent execution of four CX gates no matter how smart the path finder is; (b) shows another qubit-layout which allows all four CX gates to run in parallel.

First, it analyzes communication parallelism. It obtains the information of the number of CX gates run concurrently in the ideal logical circuit, if there is any.

Second, it performs *initial placement.* It analyzes the qubit coupling graph and use iterative graph partitioner *metis* [12] to find an initial qubit placement. In a qubit coupling graph, two qubits have an edge if there is a CX gate between them. We fine tune this initial mapping returned by *metis* by two different methods: (1) simulated annealing based on LLG characteristics (described in Section 3.3.1), and (2) optimizing for special graphs with maximal degree of two.

Third, it repeats the following two steps until all gates are scheduled: (1) a *stack-based* path finder assigns an order to the CX gates and then performs path search in this order, and (2) if necessary, a qubit placement optimizer change the layout. To dynamically remap qubits, we use the *swap-insertion* strategy. A swap gate consists of three CX gates, as shown in Fig. 11. Therefore the *layout optimization* problem becomes yet another braiding problem.

The key insight of our design is a *local parallelism pattern* we discovered in this paper. We characterize the communication parallelism using local parallel groups (LLG). LLGs are amenable to program analysis and optimization, and is highly correlated with the braiding performance.

We first introduce *LLG* definition and characterization, then introduce our detailed framework based on LLG.

3.3.1 LLG Characterization. A LLG is a minimal set of CX gates whose joint bounding box does not overlap with any other LLG's bounding box. The joint bounding box of a set of CX gates is the minimal bounding box that encloses the bounding boxes of all these individual CX gates. The size of a LLG is the number of CX gates in it. A LLG's size is at minimal 1 and at maximum the total number of CX gates in the lattice at a concurrent time step. An example of LLG is shown in Fig. 12.

We discover two properties. We exploit these two properties to find maximal number of simultaneous paths. We describe them below (rigorous proof in the Appendix).

Theorem 1. *For a LLG with ≤ 3 CX gates, given an arbitrary placement of the operand qubits, there exist a simultaneous braiding schedule for all CX gates in the LLG, and the schedule is confined within or on the boundary of the LLG's bounding box.*

Note that the theoretically concurrent CX gates at one time step can be divided into a set of LLG(s). The implication of Theorem 1 is that if all LLGs have size of 3 or smaller, there exists a congestion-free routing schedule for all CX gates. It is because different LLGs do not intersect and each LLG can find their braiding paths locally in their bounding boxes.

However, it does not mean when all LLGs' size > 3 there does not exist a congestion-free schedule for all CX gates.

Theorem 2. *For a LLG with strictly nested CX gates, there exist a simultaneous braiding schedule for all CX gates in the LLG, and such a schedule confined within or on the boundary of the LLG's bounding box. A CX gate A is strictly nested within another CX gate B iff B's bounding box encloses A's bounding box and they do not overlap.*

An example of strictly nested bounding boxes is shown in Fig. 12 denoted as *LLG1*, where each CX gate can schedule its braiding paths on the boundary of its bounding box or inside its bounding box such that these paths do not intersect.

With Theorem 1 and Theorem 2, we set a foundation for the framework of schedule braiding paths. First, the initial placement of the qubits can be optimized to minimize the number of LLGs that have size > 3 and that are not nested LLGs. We can use simulated annealing on top of the graph partition result. We can keep swapping qubits until the number of k-LLG $(k > 3)$ cannot be reduced anymore.

This already significantly reduces the circuit execution time. We show real experiment results in Table 1, after we apply the aforementioned initial layout optimization, the performance is already much improved.

Next we can partition a set of concurrent CX gates into multiple layers such that each layer runs at one time, and each layer contains maximal number of LLGs that satisfy Theorem 1 and 2. However, the conditions of Theorem 1 and 2 are overly strict. There still exist LLGs that do not satisfy either condition, but still can be scheduled without any congestion. Moreover, we need to come up with an *order for path search* as well as to enable *dynamic qubit placement.*

	After LLG Optimization		Before LLG Optimization		
Benchmark	# of LLG's (size > 3)	execution time(us)	# of LLG's (size > 3)	execution time(us)	Speed up
qft16	19	1.28K	29	1.84K	1.44
qft50	160	8.97K	176	19.2K	2.14
urf2	268	149K	2515	154K	1.03
IM16	21	745	31	1161	1.55
IM10	11	673	24	950	1.41
Shors	2010	135K	2116	283K	2.09
BTW	20	950	38	1056	1.11
Sqrt8	1	21K	6	21.1K	1.05

Table 1: Impact of LLGs' sizes

3.3.2 Path Finder and Layout Optimizer.

Path Finder. The path finder is a critical component in our AutoBraid framework shown in Fig. 10. It determines of the order of path finding for different CX gates. It first constructs a CX interference graph. In the CX interference graph, each node represents a CX gate, and each edge represents that two CX gates' bounding box intersect.

We keep removing the largest degree node from the CX interference graph. Each removed node is pushed into a stack. If there is a tie, the CX gate whose bounding box has the largest area is chosen.

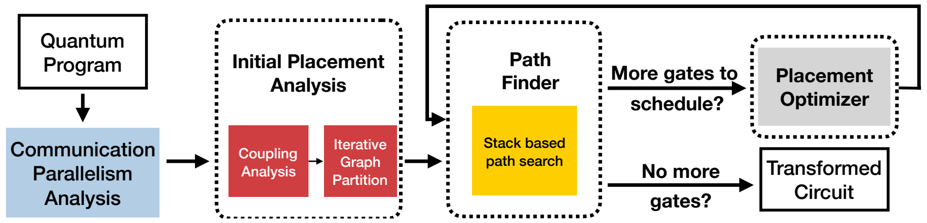


Figure 10: AutoBraid: A Framework for Scheduling Braiding Paths

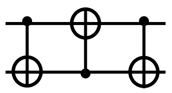


Figure 11: A SWAP gate implemented as as 3 CX gates

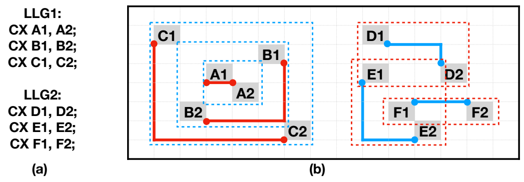


Figure 12: An example of LLGs: There are two LLGs in this lattice. Each has three CX gates. Out of these two, *LLG1* is a nested LLG, where the bounding box of C encloses that of B, and the bounding box of B encloses that of A.

The node removal process terminates until the maximal degree of the CX interference graph is ≤ 2. It is a relaxation of the condition in Theorem 1. Because when all LLGs have size ≤ 3, the degree of the CX interference graph is ≤ 2. But if the maximal degree of the CX interference graph is 2, it does not necessarily imply that all LLGs have size ≤ 3, for instance, when there is a chain of more than 3 nodes in the CX interference graph. However, this greatly reduces the complexity of path search and in practice usually leads to fully concurrent braiding paths too.

After the stack is constructed and the CX interference graph is updated, we use A* to find actual paths. It first schedules braiding paths for all CX gates in the updated interference graph. Next it schedules CX gates in the stack. It pops off a CX gate from the stack, and uses A* to find a shortest path for this CX gate. The stack ensures a last in first out (LIFO) order. The algorithm is shown in Fig. 13. This method can also handle the nested CX gate case in Theorem 2 since the CX gate that encloses other CX gates and have largest-area bounding box is handled last.

With this algorithm, we can handle path search in a hierarchical and distributive manner. It is important to avoid the scenario that a set of CX gate(s) use very long braiding paths and occupy most of the routing vertices. Our goal is to schedule as many paths as possible, and the routing resources are limited. Handling the small LLGs locally will ensure that those short distanced qubit pairs are handled first. It is also important to ensure that certain paths that may divide the lattice into disconnected component are scheduled with lowest priority. The stack ensures that.

```
Input: IG //CX interference graph
Output: paths //Routing paths for CX gates in LLG without cross
        ratio // #scheduled gates over #total gates in a LLG
    ———————————————————————
    paths = {(key, value )} // key: CX gate; value: path for this CX
    S = stack
    degreeGT2 = true
    While(degreeGT2){
        s_tie = ∅
        s_tie.push( IG.get_max_degree_nodes() )
        if( s_tie.size > 1){
            cx_d = get_largest_area(s_tie)
        }else{ cx_d = s_tie[0] }
        if(cx_ d.degree > 2){
            S.push(cx_d)
            IG.remove(cx_d) //remove node; update degree
        }else{ degreeGT2 = false }
    }
    for cx_ s in IG{
        p = get_path(cx_ s)  //p = An array stores CX's path
        paths.insert( (cx_s, p) )
    }
    while(! S.empty()){
        cx_ l = S.pop()
        p = get_path(cx_ l)
        if(! p.empty() ){
            paths.insert( (cx_l, p) ) }
    }
    ratio = paths.size()/IG.size()
```

Figure 13: Algorithm for Path Finding

We show an example in Fig. 14 where it has one large LLG of size 7, while using our algorithm all CX gates can be scheduled simultaneously.

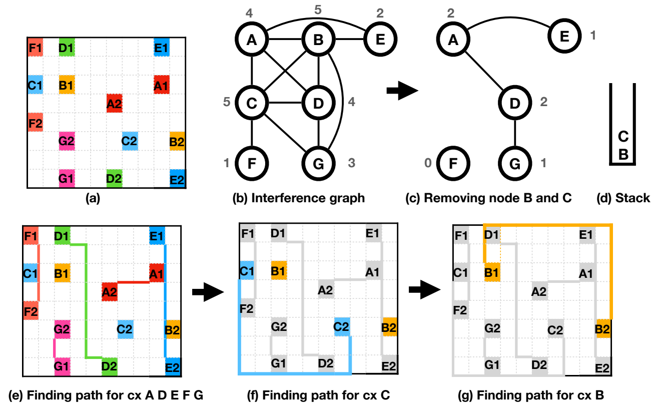

F1
D1
E1
C1
B1
A1
A2
F2
G2
C2
B2
G1
D2
E2
(a)
A
B
E
C
D
F
G
4
5
2
5
4
1
3
(b) Interference graph
A
E
D
F
G
2
1
2
0
1
(c) Removing node B and C
C
B
(d) Stack
(e) Finding path for cx A D E F G
(f) Finding path for cx C
(g) Finding path for cx B

Figure 14: An example for the stack-based path finder.

Layout Optimizer. In some cases, even the best path finder cannot find fully concurrent braiding paths due to the qubit layout. As discussed in *design considerations* in Section 3.2, even there are a lot of routing resources, they cannot be exploited. In this case, the qubit layout must be changed. We insert swap gates to dynamically change the qubit placement. A swap operation exchanges the locations of two logical qubits. Each swap incurs a cost of 3 CX gates. It needs to be performed only when it is worthwhile. We use swap gates to tackle the communication bottleneck cases.

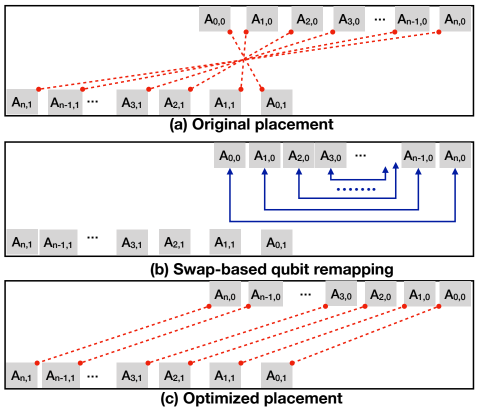

$A_{0,0}$ $A_{1,0}$ $A_{2,0}$ $A_{3,0}$ ··· $A_{n-1,0}$ $A_{n,0}$
$A_{n,1}$ $A_{n-1,1}$ ··· $A_{3,1}$ $A_{2,1}$ $A_{1,1}$ $A_{0,1}$
(a) Original placement
(b) Swap-based qubit remapping
$A_{n,0}$ $A_{n-1,0}$ ··· $A_{3,0}$ $A_{2,0}$ $A_{1,0}$ $A_{0,0}$
(c) Optimized placement

Figure 15: Swap-based qubit placement optimization

We show an example of the communication bottleneck cases. Assuming there are m CX gates that can be scheduled. However, at most 3 CX gates can be scheduled at one time. If the qubit layout can be changed, it takes at most one parallel swap step to make all CX gates executable.

In this example, the original qubit placement is shown in Fig. 15 (a). It is an extension of the case discussed in Fig. 9. We draw a straight line between two qubits in every CX gate. Every straight line separates every other pair of qubits into two different regions of the lattice. We proved that if there are four such pairs of qubits, at most three CX gates can run simultaneously. It is the same for m pairs of such qubits. At most three 3 CX gates can run at the same time. After we remove 3 CX gates from the circuit, the rest $m-3$ CX gates still have the same pattern. Therefore, in total it needs $m/3$ parallel steps, no matter how much routing resource there is.

However, we can use SWAPs to tackle this communication bottleneck. As shown in Fig. 15 (b) and (c), one parallel layer of swaps makes all CX gates executable. In this case, even one swap costs 3 CX gates, it takes 4 parallel time steps to finish all the CX gates, which is significant better than $m/3$.

In our placement optimizer, we find pairs of qubits to swap to change the qubit layout. The swaps must be schedulable with respect to braiding constraints. We first choose the CX gate that interferes with most other CX gates in the interference graph. If there is a tie, we choose the one with the largest bounding box. Then we choose a second CX gate that interferes with the first picked one and with the most rest of the gates.

For the two pairs of qubits, we choose two out of four qubits to swap. Now this swap pair is pushed into a stack. We test the swap and determine to keep it or not. We repeat this process, until no new swaps can be added due to braiding constraints. To check whether a newly inserted swap can run simultaneously with existing swaps in the stack, we take advantage of Theorem 1 and Theorem 2. If it immediately satisfies these two theorems, we keep this swap in the stack. For the cases that do not immediately satisfy Theorem 1 and 2, we try to pop out all swaps from the stack, and find paths in that order. If simultaneous paths can be found, we keep this swap, and then push all swaps back to the stack. Otherwise, we do not keep the new swap in the stack and we restore the stack to its previous state. We consider the specialization of swap insertion for all-to-all communication pattern proposed by Maslov [17] which is originally used for nearest neighbor architectures, i.e., two-qubit gates can only be enabled between neighboring qubits, as it guarantees linear depth in this congested case. It is trivial to see that for disjoint pairs of neighboring qubits, simultaneous routing paths exist. For applications with all-to-all communication pattern, we apply both greedy layout optimizer and Maslov's method, then choose the better one.

The placement optimizer is only triggered if most of the theoretically concurrent CX gates cannot be scheduled. We try different threshold ratios $p\%$. If after using the path finder, less than $p\%$ CX gates can be scheduled, we run the placement optimizer. Otherwise, we skip the placement optimization step. We try 0% to 90% threshold values on a 10% step and choose the best one among all.

4 EVALUATION

4.1 Experiment Setup

Metric Using surface code, the unit of time is the surface code cycle. We refer to *a surface code cycle* as *a cycle* for simplicity of notation.

Phase gates require magic state ancilla qubits. We use the same assumption by Javadi-Abhari *et al.* [10] such that there is a steady supply of magic state qubits at the location of the data.

We evaluate two versions of our framework. One version does not optimize the qubit placement. We refer to this version as *autobraid-sp*, where *sp* refers to stack-based path finder. The other version includes both path finding and layout optimization. We refer to it as *autobraid-full*.

Baseline We use the best known approach [10] as our baseline. It implements seven policies and we choose the best policy as the baseline. As it is a greedy policy, which prioritize routing paths with respect to shortest distance, we refer to it as the GP method. The baseline uses the graph partitioner *metis* [12] to optimize initial mapping, We denote it as GP w. initM.

Platform We simulate a surface code lattice as a structured two-dimensional L by L grid, where L is the number of unit cells at each dimension of the grid. For an input circuit with N qubits, we use the smallest square grid which provide enough qubits, such that $L = \lceil \sqrt{N} \rceil$. Using the smallest possible grid can test the effectiveness of different braiding methods. The simulation runs on a machine with Intel Xeon CPU E5-1607 which has 4 cores at 3 GHz.

BenchmarkWe use two categories of benchmarks. The benchmarks are shown in Table 2. The first category benchmarks are building block circuits [30], which are elementary functions that can be used to construct large and complex applications. The second category contains real-world applications. They include *Bernstein-Vazirani* (BV) algorithm [15], *counterfeit-coin* (CC) finding algorithm [2], *quantum fourier transformation* (QFT), *Shor's* algorithm, *quantum optimization algorithm*(QAOA) algorithm, and *Binary-Welded-Tree* (BWT) algorithm [7]. We obtain these real world applications from IBM Qiskit [2], ScaffCC [11], RevLib [28], and Cirq [1]. Qiskit can generate arbitrary-size circuits with respect to the qubit number, for certain applications, including QFT, BV, and CC. We evaluate these benchmarks with different qubit numbers.

4.2 Experiment Results

Overview. We first present an overview of the experiment results for small and medium scale inputs, for all the benchmarks in Table 2. We use a fixed surface code distance d = 33 (which results in a reasonable logical error rate P_L). For each benchmark, we show its type, qubit number, gate number, physical and computation time in microseconds. We show our *autobraid-full* version which includes both path-find and placement optimizations. We let one surface code cycle take 2.2 microseconds, based on the parameters in [10] which are faithful to most recent superconducting implementation parameters. The experiments show that our approach significantly reduces the execution time.

Our approach outperforms the baseline method with minimal 1.07X speedup to maximum 30X speedup. In Table 2, we can see for the category of algorithm building blocks, our approach have achieved the same result as the ideal case (the critical path without worrying about braiding constraints) for most cases. The *GP w. initM* method also performs relatively well. It did not achieve critical path length, but for most cases, it has 10% to 30% more execution time compared with the critical path. The building block benchmarks are relatively small. For instance, the one that has the largest number of qubits is 15, which makes it easier for the GP scheduler to find a reasonably good routing schedule. Our approach still performs better than *GP w. initM*, because it systematically decompose the concurrent CX gates into LLGs, and we find that in most cases, there are less than 3 CX gates in each LLG. Thus it is guaranteed to have the same execution time as the critical path length.

For the category of real-world applications, our approach performs better than *GP w. initM.* And we achieve larger speedup with larger qubit number for most cases. For the CC benchmark with 100 qubits, our speedup is 1.12X, it is because the GP baseline already achieved near-optimal result and there is not much room for us to improve. For the QFT benchmark, our speedup increases as the number of qubits increases. When the qubit number is 400, we achieve 30x speedup. In Table 2, OM refers to out-of-memory.

Scalability analysis. We choose three applications QFT, Ising Model (IM), and QAOA to evaluate scalability, as they represent widely-used real-world applications. QFT is also significantly used in Shor's algorithm in Table 2. For the baseline, since *GP w. initM* provided in ScaffCC [11] runs out of memory when qubit number is > 400, we implement a scheduler that is as close to *GP w. initM* described in [10], and refer to it as *baseline.*

Fig. 16 shows the circuit execution time with respect to different logical error rates P_L. The logical error rate P_L indirectly determines the size of computation, as the circuit size is inversely proportional to P_L. The code distance d is also related to P_L, as d increases when

Table 2: Overview of Experiment Results

Benchmark						Baseline (μs)	Ours (μs)	
Type	**Name**	**Description**	**# qubit**	**# gate**	**CP**	**GP w initM**	**Auto ~braid**	**Speed up**
Build-ing Blocks	4gt11_8	Compare Input	5	20	1313	3049	1313	2.32
	4gt5_75		5	48	5387	6655	5387	1.23
	alu-v0_26	ALU by Gupta	5	48	5596	6800	5596	1.21
	rd32-v0	Bit Adder	4	34	2437	5577	2437	2.2
	sqrt8_260	Square Root	12	3.09K	186K	211K	186K	1.12
	squar5_261		13	1.11K	118K	132K	118K	1.11
	squar7		15	4.07K	426K	492K	426K	1.15
	urf1_278	Unstructured Reversible Function	9	54.8K	3.63M	5.56M	3.63M	1.52
	urf2_277		8	20.1K	1.34M	3.6M	1.34M	2.66
	urf5_158		9	0.16M	10.3M	14M	10.3M	1.35
	urf5_280		9	49.8K	3.2M	3.5M	3.2M	1.07
Real World appli-cations	QFT	Quantum Fourier Transform	200	20.1K	122K	1.47M	0.63M	2.31
			400	80.2K	0.24M	70.4M	2.1M	30
			500	0.12M	0.38M	OM	3.1M	N/A
	BV	Bernstein Vezirani	100	299	15.2K	17.2K	15.2K	1.13
			150	449	22.8K	25.5K	22.8K	1.11
			200	599	30.3K	33.8K	30.3K	1.11
	CC	Counterfeit -Coin Finding	100	198	15.1K	17.0K	15.1K	1.12
			200	398	30.3K	35.4K	30.3K	1.16
			300	598	45.4K	53.1K	45.4K	1.16
	IM	Ising Model	10	480	4162	12K	4162	2.88
			500	5494	908	2900	908	2.09
			1000	10.9K	908	2100	908	2.31
	BWT	Binary Welded Tree	179	260	7433	10.2K	7433	1.37
			240	365	7585	10.3K	7585	1.36
	QAOA	QAOA	100	4.5K	10.9K	20.8K	13K	1.59
			200	9K	12.3K	30.5K	13.4K	2.19
			300	13.5K	13.6K	40.7K	15.5K	2.64
	Shor's	Shor's algorithm	471	36.5K	0.53M	1.78M	0.54M	3.29

P_L decreases. We can see that for all benchmarks, our methods show significant improvement compared with the baseline. By applying the layout optimization, the result of "autobraid-full" is closer to the critical path performance. For IM, the results of "autobraid-full" exactly match the critical path lengths so the two curves overlap.

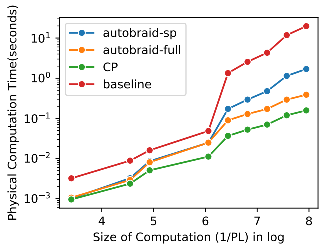
Resource utilization analysis. We also analyze the routing resource usage. The resource usage ratio is defined as the number of occupied vertices divided by the total number of available vertices. Higher ratio typically implies better usage of the routing resources. We scale the problem size of computation $(1/P_L)$. The result is shown in Fig. 17. We can see that our *autobraid* methods have relatively good resource usage. It uses up to 70% resource while the baseline achieves up to 37% resource.

P-sensitivity analysis. The layout optimizer is only triggered if most of the theoretically concurrent CX gates cannot be scheduled. We set a threshold percentage p%, such that if 1-p% CX gates cannot be scheduled by the path finder, the layout optimizer is triggered. The performance is sensitive to the p value for different benchmarks and hence we choose to test a range of p values to choose the best one. We show the p-sensitivity experiment results in Fig. 18 for QFT-1000 and QAOA-1000.

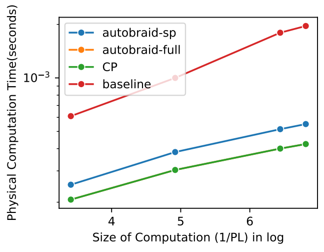
Compilation time analysis. We evaluate compilation time overhead. We compare it with the physical circuit execution time. We find for most of benchmarks, compilation time takes only around 1-2% percentage of physical computation time.

5 RELATED WORK

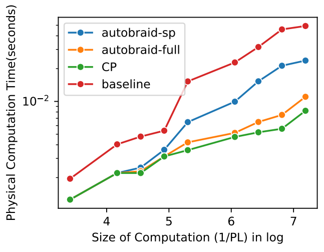
Compilation for quantum programs has been mostly focused on non-fault-tolerant hardware. Recently studies [14, 18, 23, 25–27, 30] focus on mapping logical circuits to superconducting quantum hardware with constrained physical connectivity. Heckey *et al.* [9]



(a) QFT

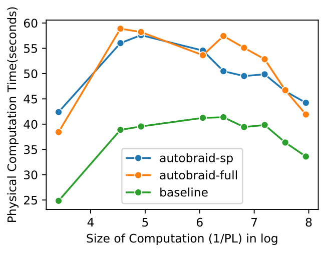


(b) IM

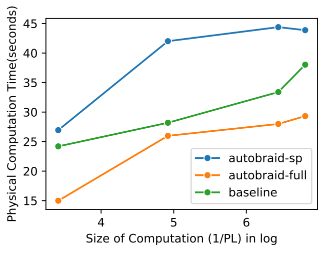


(c) QAOA

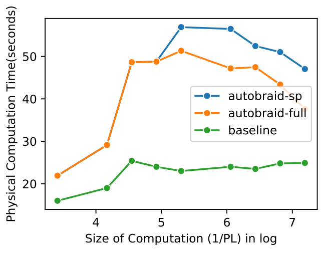
Figure 16: Physical circuit execution time (seconds) with respect to different computation sizes ($1/P_L$). "autobraid-full" applies path finder together with layout optimization, "autobraid-sp" applies only stack-based path finder, and "critical path (CP)" is the ideal execution time.



(a) QFT

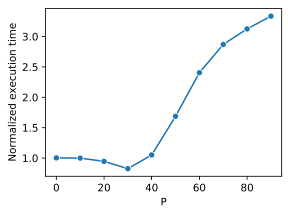


(b) IM

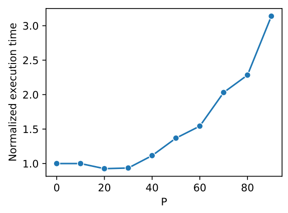


(c) QAOA

Figure 17: Resource utilization ratio (%) with respect to computation size ($1/P_L$).



(a) QFT-500



(b) QAOA-500

Figure 18: P-sensitivity analysis (time is normalized to that when $p = 0$ on y-axis). X-axis corresponds to p value.

improve instruction level parallelism for ion-trap quantum computers. The work by Kudrow and others [13] improves dynamic compilation time. These studies target hardware without QEC.

There are two categories of studies that focus on program compilation on fault-tolerant quantum hardware. The first category of studies [4, 6, 21] focus on topological transformations in a three dimensional lattice, where the physical qubits form a two-dimension layout, and the third dimension is time. These studies usually focus on a CX network which consists of a sequence of CNOT gates that have dependence relationships. The braiding paths can be deformed with respect to topological transformation rules to minimize circuit time. This line of work is orthogonal to ours, since we focus on scheduling parallel CX gates and can use their results as templates for dependent CX gates. The second category of studies [5, 20] focus on magic state distillation circuits and how to insert them into the time-space dimensions during the circuit synthesis phase, for supporting Clifford+T gates. The magic distillation circuits are important, but they present a different type of problems as qubits need to be re-arranged for purifying qubits that have high fidelity and their work is complementary to ours.

As far as we can tell, the study by Javadi-Abhari *et al.* [10] is the most relevant. It provides a comprehensive evaluation of two types of communication modes in surface codes: the planar code (through teleportation) and the double-defect code (through braiding). They provide greedy scheduling policies for braiding operations. Our study has been shown to outperform their methods for both small-scale and large-scale circuits. Javadi-Abhari *et al.* [10] discovered in certain scenarios, the planar code might be more favorable than the double-defect code due to braiding congestion. However, it might not be that the nature of braiding-based two-qubit gate causes this problem. With a proper design of the braiding scheduler that addresses the congestion bottleneck, the double-defect code might be more favorable than the planar code, as it uses fewer physical qubits than the planar code.

6 CONCLUSION

We propose a framework for analyzing quantum programs in surface code and scheduling braiding paths that correspond to two-qubit gates. We develop an analytical model for the programs with local parallelism pattern. Our framework achieves (near-)critical path performance for small and medium scale quantum programs, and up to orders of magnitude improvement for large-scale quantum programs, compared with the best known approach. Our effort is an important step for building the required synthesis tool for future large-scale fault-tolerant quantum computers.

ACKNOWLEDGMENTS

We thank Ali Javadi-Abhari for providing invaluable comments on the baseline implementation and the anonymous reviewers for their constructive feedback. This work is supported by grants from Rutgers Research Council and NSF-FET-2129872. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of our sponsors.

APPENDIX

We use the notation E(α, β, γ, ...) to denote the outer bounding box for a set of CX gates α, β, γ, ... We use the notation I(α) to represent the inner bounding box of CX gate α. It is the minimal bounding box that encloses at least one vertex of A_1 and one vertex of A_2. The outer bounding box of a CX gate does not intersect the inner bounding box of itself as long as this CX gate's bounding box is not one dimensional. Examples of outer and inner bounding boxes are given in Fig. 19.

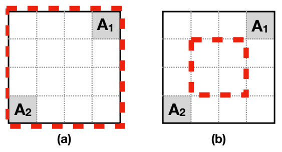


Figure 19: (a) outer bounding box, (b) inner bounding box.

A 4-CX LLG

If a LLG has $\geq$ 4 CX gates, it is not guaranteed to have simultaneous braiding paths within the LLG's bounding box.

Theorem 3. *For a LLG with four CX gates A, B, C and D on a two-dimensional grid, it is not guaranteed that there exist simultaneous braiding paths within E(A, B, C, D), given arbitrary placement of the four pairs of qubits.*

Proof. We have provided a case of 4 CX gates in Fig. 9 where simultaneous braiding paths cannot be found in Section 3.2. Hence it is proved. □

Next we show the proof for the existence of simultaneous braiding paths in 1-CX, 2-CX, and 3-CX LLGs.

B 1-CX LLG

Theorem 4. *For a LLG with just one CX A, there exists a braiding path for A within the bounding box E(A).*

Proof. The proof is trivial. We find the shortest path between A_1 and A_2 (the two operand qubits in the CX gate A) on the inner bounding box I(A). Since the inner bounding box I(A) is confined to the outer bounding box E(A), it is proved. □

C 2-CX LLG

Theorem 5. *For a LLG with two CX gates A and B, there exists simultaneous braiding paths for both A and B within or on the boundary of the LLG's bounding box E(A, B).*

Proof. To prove the existence, we construct such two braiding paths. We let A's braiding path be one shortest path on the inner bounding box I(A). Let A_1 and A_2 be the two qubits of the CX gate A. Either the braiding path does not reach the boundary of E(A, B) or the entire braiding path is on the boundary of E(A,B). Neither case will divide E(A, B) into two disjoint connected components. Since I(A) is the inner bounding box of A, it is either completely disjoint from the boundary of E(A, B) as shown in Fig. 19, or is completely on one border of E(A, B) when both A_1 and A_2 are on the boundary of E(A, B).

Furthermore, A's braiding path itself does not form a cycle – meaning its starting vertex does not overlap with its ending vertex, hence A's braiding path will not occupy any qubit cell's four vertices. So the four vertices of either of B's two qubits are not fully utilized by A. Since the rest un-utilized vertices and edges form a connected component, the CX gate B can find a braiding path within E(A, B). □

D 3-CX LLG

Now we prove that there exist simultaneous braiding paths for all CX gates in a 3-CX LLG when the LLG's bounding box is at least 2 on each dimension.

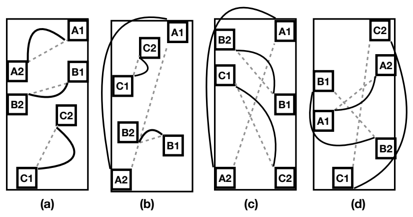


Figure 20: Four cases for the proof of 3-CX LLG.

Theorem 6. *For a LLG with 3 CX gates, and the LLG's bounding box has at least 2 by 3 or 3 by 2 cells, given arbitrary placement of the six qubits, there exist simultaneous braiding path confined to the LLG's bounding box E(A,B,C).*

Proof. For each CX gate, we draw a straight line path between the two qubits in its inner bounding box. If one CX gate's straight line path intersects another CX gate's straight line path or the vertices of another CX's qubits, we say that these two CX gates strictly interference as opposed to CX interference by bounding box intersection in Section 3.3.2.

Based on the strict interference relationship of the three CX gates in the LLG, we prove Theorem 6 case by case.

The first case is that no CX gate strictly interfere with any other CX gate. An example is shown in Fig. 20 (a), we find paths by routing through the cells each straight line path crosses. There exist such three paths since three sets of cells that are covered by three straight line paths are disjoint. The three paths can pick disjoint set of vertices on each set of qubit cells and cause no congestion.

The second case is that two CX gates strictly interfere, while the third CX gate does not interfere with either. An example is shown in Fig. 20 (b). Without loss of generality, assuming A is the CX gate that strictly interferes with B, and C is the CX gate that does not interfere with either A or B. We first construct braiding paths for B and C along their straight line paths inside the E(A, B). Then for A, we find a braiding path that go around the paths of both B and C, and along the boundary of E(A, B, C).

The third case is that one CX gate strictly interferes with two other CX gates, while the other two CX gates do not interfere with each other. An example is shown in Fig. 20 (c). Assuming A is the CX gate that strictly interferes with B and C, we find braiding paths for B and C first along B's and C's straight line paths, then there exists at least one path of A along the boundary of the LLG's bounding box E(A, B, C) and also around B's and C's paths.

The fourth case is that all CX gates strictly interfere with each other. That means one CX gate's two qubits are on two sides of any other CX gate's straight line path. There is only one way to construct such a case, as shown in Fig. 20 (d). To handle this case, we pick any CX gate, and we denote it as A. We first construct a braiding path along the straight line path of A, which is inside the LLG's bounding box. Next we pick any other CX gate, and we call it B. For B, we let it go around one qubit of A, and call it A_1. It means that we leave the other qubit of A's untouched – the border of E(A, B, C) closest to this qubit is open too. Next, for C, it is clear that C can go around A_2 (the other qubit of A) to find a path.

There are only four possible cases with respect to the relationships of strict interference. Hence it is proved. □

REFERENCES

[1] [n. d.]. Cirq, a python framework for creating, editing, and invoking Noisy Intermediate Scale Quantum (NISQ) circuits. ([n. d.]). url-https://github.com/quantumlib/Cirq.

[2] Andrew Cross. 2018. The IBM Q experience and QISKit open-source quantum computing software. *Bulletin of the American Physical Society* 63 (2018).

[3] Simon J Devitt, William J Munro, and Kae Nemoto. 2013. Quantum error correction for beginners. *Reports on Progress in Physics* 76, 7 (2013), 076001.

[4] Simon J Devitt, Ashley M Stephens, William J Munro, and Kae Nemoto. 2013. Requirements for fault-tolerant factoring on an atom-optics quantum computer. *Nature communications* 4 (2013), 2524.

[5] Yongshan Ding, Adam Holmes, Ali Javadi-Abhari, Diana Franklin, Margaret Martonosi, and Frederic Chong. 2018. Magic-state functional units: Mapping and scheduling multi-level distillation circuits for fault-tolerant quantum architectures. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 828–840.

[6] Austin G Fowler and Simon J Devitt. 2012. A bridge to lower overhead quantum computation. *arXiv preprint arXiv:1209.0510* (2012).

[7] Mrityunjay Ghosh, Amlan Chakrabarti, and Niraj K Jha. 2017. Automated quantum circuit synthesis and cost estimation for the binary welded tree oracle. *ACM Journal on Emerging Technologies in Computing Systems (JETC)* 13, 4 (2017), 1–14.

[8] Daniel Gottesman. 2010. An introduction to quantum error correction and fault-tolerant quantum computation. In *Quantum information science and its contributions to mathematics, Proceedings of Symposia in Applied Mathematics*, Vol. 68. 13–58.

[9] Jeff Heckey, Shruti Patil, Ali JavadiAbhari, Adam Holmes, Daniel Kudrow, Kenneth R Brown, Diana Franklin, Frederic T Chong, and Margaret Martonosi. 2015. Compiler management of communication and parallelism for quantum computation. In *ACM SIGARCH Computer Architecture News*, Vol. 43. ACM, 445–456.

[10] Ali Javadi-Abhari, Pranav Gokhale, Adam Holmes, Diana Franklin, Kenneth R. Brown, Margaret Martonosi, and Frederic T. Chong. 2017. Optimized surface code communication in superconducting quantum computers. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2017, Cambridge, MA, USA, October 14-18n.*

[11] Ali JavadiAbhari, Shruti Patil, Daniel Kudrow, Jeff Heckey, Alexey Lvov, Frederic T Chong, and Margaret Martonosi. 2014. ScaffCC: a framework for compilation and analysis of quantum computing programs. In *Proceedings of the 11th ACM Conference on Computing Frontiers*. 1–10.

[12] George Karypis and Vipin Kumar. 1995. Metis-unstructured graph partitioning and sparse matrix ordering system, version 2.0. (1995).

[13] Daniel Kudrow, Kenneth Bier, Zhaoxia Deng, Diana Franklin, Yu Tomita, Kenneth R. Brown, and Frederic T. Chong. 2013. Quantum Rotations: A Case Study in Static and Dynamic Machine-Code Generation for Quantum Computers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA '13)*. Association for Computing Machinery, New York, NY, USA, 166–176. https://doi.org/10.1145/2485922.2485937

[14] Gushu Li, Yufei Ding, and Yuan Xie. 2019. Tackling the qubit mapping problem for NISQ-era quantum devices. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 1001–1014.

[15] Hongwei Li and Li Yang. 2015. A quantum algorithm for approximating the influences of Boolean functions and its applications. *Quantum Information Processing* 14, 6 (2015), 1787–1797.

[16] Austin G. Fowler Matteo Mariantoni, John M. Martinis, and Andrew N. Cleland. 2012. Surface codes: Towards practical large-scale quantum computation. In *PHYSICAL REVIEW A covering atomic, molecular, and optical physics and quantum information.*

[17] Dmitri Maslov. 2007. Linear depth stabilizer and quantum Fourier transformation circuits with no auxiliary qubits in finite-neighbor quantum architectures. *Physical Review A* 76, 5 (Nov 2007). https://doi.org/10.1103/physreva.76.052310

[18] Prakash Murali, Jonathan M Baker, Ali Javadi-Abhari, Frederic T Chong, and Margaret Martonosi. 2019. Noise-adaptive compiler mappings for noisy intermediate-scale quantum computers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 1015–1029.

[19] P. J. J. O'Malley, J. Kelly, R. Barends, B. Campbell, Y. Chen, Z. Chen, B. Chiaro, A. Dunsworth, A. G. Fowler, I.-C. Hoi, and et al. 2015. Qubit Metrology of Ultralow Phase Noise Using Randomized Benchmarking. *Physical Review Applied* 3, 4 (Apr 2015). https://doi.org/10.1103/physrevapplied.3.044009

[20] Alexandru Paler. 2019. SurfBraid: A concept tool for preparing and resource estimating quantum circuits protected by the surface code. arXiv:quant-ph/1902.02417

[21] R Raussendorf, J Harrington, and K Goyal. 2007. Topological fault-tolerance in cluster state quantum computation. *New Journal of Physics* 9, 6 (Jun 2007), 199–199. https://doi.org/10.1088/1367-2630/9/6/199

[22] Michael A Riepe and Karem A Sakallah. 2003. Transistor placement for noncomplementary digital VLSI cell synthesis. *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 8, 1 (2003), 81–107.

[23] Marcos Yukio Siraichi, Vinícius Fernandes dos Santos, Sylvain Collange, and Fernando Magno Quintão Pereira. 2018. Qubit allocation. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*. ACM, 113–125.

[24] Swamit S. Tannu, Zachary A. Myers, Prashant J. Nair, Douglas M. Carmean, and Moinuddin K. Qureshi. 2017. Taming the Instruction Bandwidth of Quantum Computers via Hardware-Managed Error Correction. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-50 '17)*. Association for Computing Machinery, New York, NY, USA, 679–691. https://doi.org/10.1145/3123939.3123940

[25] Swamit S. Tannu and Moinuddin Qureshi. 2019. Ensemble of Diverse Mappings: Improving Reliability of Quantum Computers by Orchestrating Dissimilar Mistakes. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '52)*. Association for Computing Machinery, New York, NY, USA, 253–265. https://doi.org/10.1145/3352460.3358257

[26] Swamit S Tannu and Moinuddin K Qureshi. 2019. Not all qubits are created equal: a case for variability-aware policies for NISQ-era quantum computers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 987–999.

[27] Robert Wille, Lukas Burgholzer, and Alwin Zulehner. 2019. Mapping quantum circuits to IBM QX architectures using the minimal number of SWAP and H operations. In *Proceedings of the 56th Annual Design Automation Conference 2019*. ACM, 142.

[28] R. Wille, D. Große, L. Teuber, G. W. Dueck, and R. Drechsler. 2008. RevLib: An Online Resource for Reversible Functions and Reversible Circuits. In *Int'l Symp. on Multi-Valued Logic*. 220–225. RevLib is available at http://www.revlib.org.

[29] Chi Zhang, Ari Hayes, Longfei Qiu, Yuwei Jin, Yanhao Chen, and Edd Z. Zhang. 2021. Time-Optimal Qubit Mapping. In *Proceedings of the Twenty-Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21)*. ACM, Virtual.

[30] Alwin Zulehner, Alexandru Paler, and Robert Wille. 2018. Efficient mapping of quantum circuits to the IBM QX architectures. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 1135–1138.