

Efficient and high-performance routing of lattice-surgery paths on three-dimensional lattice

Kou Hamada¹, Yasunari Suzuki^{2,3,4}, and Yuuki Tokunaga^{2,5}

¹Department of Mathematical Informatics, Graduate School of Information Science and Technology, The University of Tokyo, 7-3-1 Hongo, Bunkyo-ku, Tokyo 113-0033, Japan

²NTT Computer and Data Science Laboratories, NTT Corporation, Musashino 180-8585, Japan

³JST, PRESTO, 4-1-8 Honcho, Kawaguchi, Saitama, 332-0012, Japan

⁴Center for Quantum Computing, RIKEN, 2-1 Hirosawa, Wako, Saitama 351-0198, Japan

⁵Faculty of Information Science and Technology, Hokkaido University, Sapporo 060-0814, Japan

Encoding logical qubits with surface codes and performing multi-qubit logical operations with lattice surgery is one of the most promising approaches to demonstrate fault-tolerant quantum computing. Thus, a method to efficiently schedule a sequence of lattice-surgery operations is vital for high-performance fault-tolerant quantum computing. A possible strategy to improve the throughput of lattice-surgery operations is splitting a large instruction into several small instructions, such as Bell state preparation and measurements, and executing a part of them in advance. However, scheduling methods to fully utilize this idea have yet to be explored. In this paper, we propose a fast and high-performance scheduling algorithm for lattice-surgery instructions leveraging this strategy. We achieved this by converting the scheduling problem of lattice-surgery instructions to a graph problem of embedding 3D paths into a 3D lattice, which enables us to explore efficient scheduling by solving path search problems in the 3D lattice. Based on this reduction, we propose a method to solve the path-finding problems, the look-ahead Dijkstra projection. We numerically show that this method reduced the execution time of benchmark programs generated from quantum phase estimation algorithms by 3.8 times compared with a naive method based on greedy algorithms. Our study establishes the relation between the lattice-surgery scheduling and graph search problems, which leads to further theoretical analysis on compiler optimization of fault-tolerant quantum computing.

1 Introduction

Quantum computing can efficiently solve several vital scientific problems [1–3], but its high error rates prevent us from a practical demonstration of quantum advantage. We can overcome this problem with a quantum error-correction technology, i.e., by creating a logical qubit with a negligible error rate by using several noisy qubits. Surface codes [4–6]

Kou Hamada: zkouaaa@g.ecc.u-tokyo.ac.jp

Yasunari Suzuki: yasunari.suzuki@riken.jp

Yuuki Tokunaga: tokunaga@ist.hokudai.ac.jp

are known as promising candidates for quantum error-correcting codes since they show high error-correction performance and can be implemented with qubits having nearest-neighbor interactions on a two-dimensional (2D) grid lattice. Lattice surgery [7, 8] plays a key role in the fault-tolerant quantum computation with surface-code logical qubits: they allow multi-qubit Pauli measurements in a fault-tolerant manner by connecting target logical qubits through ancillary logical-qubit space, and they can be performed in parallel as long as their paths do not overlap. The current state-of-the-art resource estimation stands on architectures with surface codes and lattice surgery [9–11].

To run quantum algorithms with a short execution time, we should maximize the throughput of quantum instructions [12–15] by optimizing their scheduling. Unfortunately, finding the optimal scheduling of lattice-surgery instructions is known to be NP-hard [16, 17]. Thus, fast and near-optimal approximation strategies for scheduling lattice-surgery instructions are demanded. For frequently used modules such as magic-state distillation circuits, there are dedicated designs and logical-qubit mappings [8, 18, 19]. On the other hand, for general purposes, we need an algorithm that can accept an arbitrary lattice-surgery sequence. One possible strategy is to map problems into well-known NP-hard instances and find high-quality solutions with fast solvers. Lao *et al.* [20] map problems into the quadratic assignment problem, and Molavi *et al.* [17] into SAT problems. While this method would show near-optimal solutions, the target size of the compilation is limited by the capability of solvers. Thus, efficient and high-performance scheduling methods based on other concepts are still in demand.

One of the promising approaches for general and high-performance optimization is to split instructions that are difficult to parallelize into several small fragments and execute parts of them in advance. Beverland *et al.* [21] proposed a method based on this idea. They propose methods to find a way to split and schedule instructions by finding the edge-disjoint path on a 2D lattice and converting the solution to a lattice-surgery sequence for multiple execution cycles. While this strategy is shown to be effective, the proposed method sticks to specific logical-qubit allocation patterns while possible logical-qubit allocations can be arbitrary [9–11], which loses the versatility in compilation targets. Also, the range of optimization in this algorithm is limited to instructions within a narrow time window. Therefore, we need a strategy that can fully leverage the idea of instruction split and preceding execution of them.

Here, we propose an efficient and high-performance lattice-surgery scheduling algorithm that utilizes the strategy. The key step of our proposal is establishing the correspondence between a path in a 3D lattice, an executable sequence of lattice-surgery instructions, and achievable actions on target logical qubits. Thanks to this correspondence, we can rephrase the optimization of a lattice-surgery sequence as a problem to embed several 3D paths in a 3D grid lattice, in which we can utilize graph optimization methods. On the basis of this result, we show several polynomial-time approximation algorithms to solve this problem. We evaluated these algorithms with numerical simulations and found that one among them, named look-ahead Dijkstra projection, significantly improves throughput with a reasonable compilation time. We numerically showed that the look-ahead Dijkstra projection improves throughput by 3.8 times compared with a naive solution in a benchmark with realistic applications, SELECT modules in qubitization-based quantum phase estimation [10, 22].

Our results not only optimize the runtime of quantum algorithms but also show that the complicated lattice-surgery compilation can be reduced to simple graph problems, where we can naturally utilize the knowledge of path-search algorithms. Since our algorithm works for general logical-qubit mapping and connectivity, our methods can be extended to other fault-tolerant quantum computing (FTQC) designs based on lattice surgery or

distributed quantum computing. While our benchmark focuses on the case of two-body lattice-surgery instructions, we also showed that our method can be straightforwardly extended to multi-body lattice surgery in Appendix A. Thus, our results offer a base for versatile and high-performance compilation for fault-tolerant quantum computing.

Our contribution can be summarized as follows:

- We extend the lattice-surgery scheduling problem from a 2D routing problem to a 3D routing problem. Since our formulation strictly contains the original problem as a special case, it always improves the achievable solutions equal to or superior to the conventional 2D approaches.
- We propose an efficient and high-performance scheduling method for our 3D formulation, called look-ahead Dijkstra projection. This algorithm leverages the properties of the extended formalism with only a modest increase in time complexity compared to simple baseline algorithms.
- We numerically evaluate the performance of the proposed algorithm and demonstrate a 3.8-times speedup on benchmark programs derived from practical applications compared with a baseline method. Although the compilation time increases by a factor of 7.0 in the worst scenario, it still completes within a few tens of seconds even for practical-scale applications. We also show that other sources of overhead, such as increased logical error rates, remain modest or negligible.

2 Preliminary

2.1 Surface codes and logical operations

Since qubits suffer from large error rates, we need quantum error correction to suppress their error rate to an arbitrarily small value. Surface codes are known as the most promising quantum error-correcting codes, which encode the information of a qubit into a square cell of qubits. The width of the cell corresponds to a code distance, and by increasing the code distance, we can exponentially suppress the error rates if physical error rates are smaller than the value known as thresholds [4, 23, 24]. A cell of surface codes has two types of boundaries: X -type and Z -type, where Pauli- X operations along with X -type boundary constitute logical X operations, and the same for Pauli- Z .

We suppose that qubits are integrated on a large two-dimensional plane, which we call a qubit plane. We assume that the qubit plane is divided into several surface-code logical-qubit cells. During the computation, several cells are used for keeping single-qubit information, which we call *data cells*. The other cells are available as working space for logical operations on the data cells, which we call *ancillary cells*. Fig. 1 shows an example of a qubit plane, where dark cells are data cells and light cells are ancillary cells. As surface codes have the two types of boundaries, each cell also has two types of directions: top and bottom are X -boundaries, and the others are Z -boundaries; or top and bottom are Z -boundaries, and the others are X -boundaries.

To execute quantum algorithms, we need to perform universal quantum operations on encoded logical qubits fault-tolerantly. The most standard set of logical operations is as follows. We can initialize a cell in a Z -basis ($|0\rangle, |1\rangle$) or X -basis ($|+\rangle, |-\rangle$) and can measure them in X - and Z -bases destructively. These operations can be achieved with a single code cycle, i.e., with effectively negligible time compared with the other operations. Note that the direction of X/Z -type boundaries can be freely chosen at the initialization timing. We can also fault-tolerantly perform single-qubit logical operations, i.e., Hadamard gates,

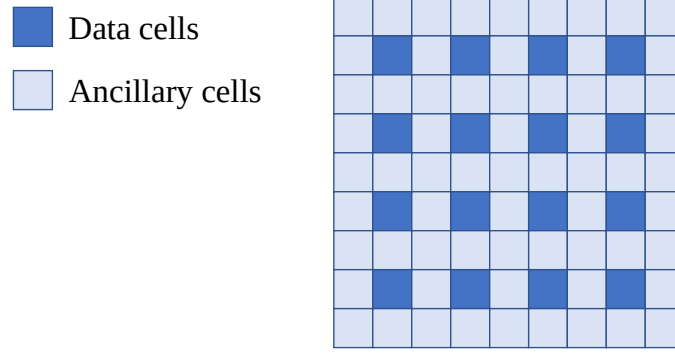


Figure 1: Data allocation in a qubit plane.

phase gates with a twist, T gates via magic-state injection, distillation, and teleportation [8, 23, 25]. Finally, we can achieve multi-qubit logical operations via lattice surgery [7], which acts on code space as multi-qubit Pauli measurements.

Hereafter, the discussion will focus exclusively on logical qubits. Thus, the term “qubit” refers to a logical qubit unless specified otherwise.

2.2 Scheduling of lattice surgery

Among the universal operations, the scheduling of lattice-surgery instructions tends to be a complicated task since multiple cells are relevant to them. Thus, in this paper, we focus on the scheduling of lattice-surgery instructions. A lattice-surgery instruction is achieved by connecting target logical-qubit cells. If the target cells are not neighboring, we can use ancillary cells to connect them. Once connected, we need to repeat syndrome measurements for d times, where d is a code distance, to guarantee reliable parity-check measurements. Thus, it is convenient to define this duration of d code cycles for lattice surgery as a time unit, which is called a *code beat*. The types of connected boundaries determine the basis of Pauli measurements. We can perform Pauli-Z measurements if a Z-boundary of a cell is connected and Pauli-X for X-boundary.

Throughout this paper, we assume the following for simplicity.

1. We ignore the time for executing instructions other than the lattice surgery. In other words, we assume all the instructions are lattice surgery.
2. While lattice surgery can act on an arbitrary number of cells, we only consider lattice-surgery instructions acting on two logical qubits with the same types of boundaries.

The extension to many-body lattice surgery is detailed in Appendix A. The extensions to general cases will be left as future work and discussed in Sec. 5.

To maximize the computational speed, we should parallelize the instructions as much as possible. Suppose that lattice-surgery instructions are provided as a sequence, each of which describes two target logical qubits and the type of boundaries. Fig. 2 shows the example of parallel execution of lattice surgeries. In this figure, three pairs of orange logical qubits are connected simultaneously by using ancillary cells, which means three lattice-surgery instructions are performed at the same time. On the other hand, we cannot perform lattice surgery on two hatched green cells in this code beat since there are no paths between them. After a code beat, the cells occupied by orange lattice-surgery paths become available, and we can connect hatched green cells.

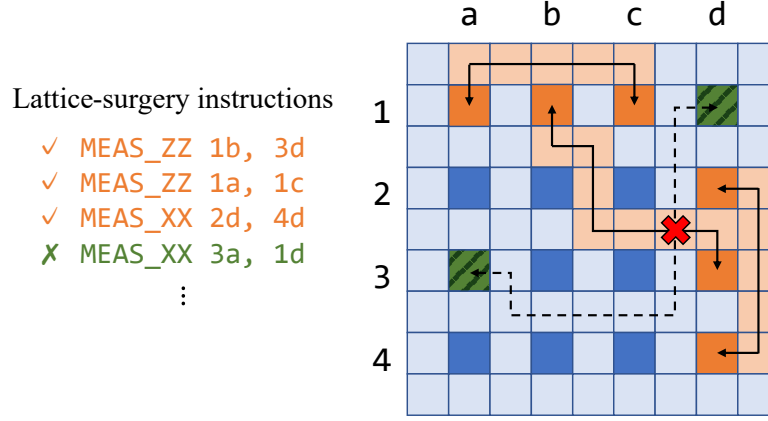


Figure 2: Example of not-parallelizable lattice-surgery instructions due to path conflict.

This example illustrates that optimizing the ordering and routing of lattice surgery is an important factor in maximizing the performance of quantum computing. Unfortunately, the optimal scheduling of the sequence of lattice surgery is known to be NP-hard, i.e., the optimal scheduling is computationally difficult to find within a reasonable time [16]. Thus, an efficient algorithm that can find the near-optimal paths of lattice-surgery instructions is strongly demanded.

2.3 Baseline scheduling method: BFS

As a baseline method, we introduce a simple procedure to schedule the sequence of lattice-surgery instructions. The most naive strategy for scheduling instructions is to use breadth-first search (BFS) as a subroutine to find one of the shortest paths. We call the procedure *BFS*, which is defined as follows.

1. Mark all the ancillary cells as available.
2. Pop an instruction from the instruction queue.
3. Find one of the shortest paths connecting two logical qubits with specified boundaries. If found, mark the cells on the path as unavailable, and then go to 2. If not, increase the code beat and go to 1.

The pseudo-code of this procedure is shown in Alg. 1. Here, we keep whether the ancillary cells are available or not in a map object A . `FINDSHORTESTPATH` will return one of the shortest paths from v_1 to v_2 on graph G for a lattice surgery instruction $o = (v_1, v_2, d)$ so that the path does not use surface-code cells marked 1 in A and v_1, v_2 are connected to the path from $d \in \{X, Z\}$ boundaries. If there is no path between them, the function returns an empty list. We can find the shortest paths in a graph G with V vertices and E edges in $O(V + E)$ time using the BFS algorithm. In this procedure, while choosing the shortest path will not necessarily lead to optimal results, we expect this choice to be reasonable since it will consume the fewest cells.

The routed lattice-surgery sequence can be visualized in the 3D lattice, where the XY-plane corresponds to the qubit plane and the Z-axis to the time flow. Fig. 3 shows an example converted from that of Fig. 2. A 3D cell, or a *voxel*, at the (x, y, t) position in the 3D lattice corresponds to the surface-code cell at the coordinate (x, y) at the t -th code beat. When the cell is used for encoding a logical qubit or temporally for logical

Algorithm 1 BFS scheduling

Input: Logical-qubit connectivity graph $G = (V, E)$.

Input: List of m instructions of two-body lattice surgery. Each instruction $o \in O$ consists of $o = (v_1, v_2, d)$ where $v_1, v_2 \in V$ and $d \in \{X, Z\}$.

Output: Scheduling paths and timings S and total runtime t .

```
1:  $S \leftarrow []$ 
2: Define  $A$  as a map from  $V$  to  $\{0, 1\}$  and initialize all the elements with 0
3:  $t \leftarrow 0$ 
4:  $i \leftarrow 0$ 
5: while  $i < m$  do
6:   while  $i < m$  do
7:      $o \leftarrow O[i]$ 
8:      $p \leftarrow \text{FINDSHORTESTPATH}(o, G, A)$ 
9:     if  $p$  is empty then
10:       break
11:     end if
12:     Append  $(p, t)$  to  $S$ 
13:     for  $v$  in  $p$  do
14:        $A[v] \leftarrow 1$ 
15:     end for
16:      $i \leftarrow i + 1$ 
17:   end while
18:   for  $v$  in  $V$  do
19:      $A[v] \leftarrow 0$ 
20:   end for
21:    $t \leftarrow t + 1$ 
22: end while
23: return  $(S, t)$ 
```

operations, it is drawn as a colored voxel in the 3D lattice. When we assume logical qubits are persistently allocated at the same position, the cell of encoded logical qubits is visualized as a pillar from the initialization to the destructive measurements. The lattice surgery can be visualized as a path connecting two target pillars at a certain time slice.

2.4 Baseline scheduling method: Look-ahead BFS

When the instructions are provided as a 1D sequence, we can change the order of the instructions so that the actions of sequences do not change and achieve a more efficient scheduling. A method to find unexecuted instructions is to iterate over all the unexecuted instructions and check if each unexecuted instruction can be executed beforehand, that is, maintain a flag to indicate whether each cell has any previous unexecuted instructions to be executed during the iteration and check if either of the two logical qubits has any previous instructions. The modified algorithms can be found as follows.

1. Mark all the instructions unexecuted.
2. Mark all the ancillary cells as available.
3. Mark all the data cells as ready.
4. Perform Steps 5 and 6 for all the unexecuted instructions.

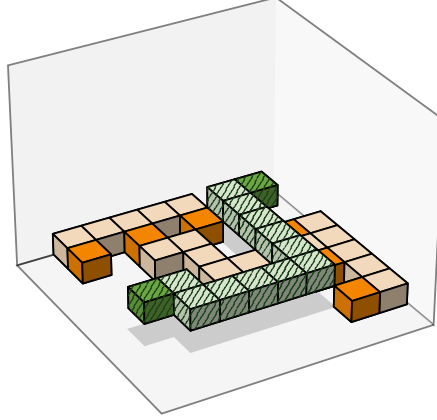


Figure 3: 3D lattice representation of lattice-surgery instructions. This example corresponds to the sequence in Fig. 2.

5. If the two target logical qubits are both marked as ready, find one of the shortest paths connecting two logical qubits with specified boundaries. If found, mark the cells on the path as unavailable, and mark the instruction executed. If not, do nothing.
6. Mark the two target logical qubits as unready.
7. Increase the code beat and go to 2.

The pseudo-code of this procedure is shown in Alg. 2.

In this algorithm, we use an object D with the following data structure for time-efficient scheduling. This object manages an instruction dependency graph as a directed acyclic graph, where each node has pointers to child nodes and has a counter of the unexecuted parent nodes. The function `GETEXECUTABLEINDEXLIST` returns all the indices of instructions that have no unexecuted parent nodes in ascending order, and `UPDATEDependencyGraph` marks the i -th instruction as executed, i.e., decrements the count of unexecuted parent nodes for each child node. Assuming each instruction acts on a constant number of logical qubits, we can create this data structure with $O(m)$, and these functions have $O(\log m)$ running time per instruction by using a priority queue: `GETEXECUTABLEINDEXLIST` runs in $O(|L| \log m)$ time and `UPDATEDependencyGraph` runs in $O(\log m)$ time, where $|L|$ denotes the number of executable instructions. Thus, the runtime to maintain the instruction dependency is negligible compared with the runtime of searching for a path.

3 Scheduling of lattice surgery using entanglement

3.1 Overview

In this paper, we propose a polynomial-time algorithm for efficiently scheduling lattice surgery. The outline of our idea is as follows. First, we split a lattice-surgery instruction on two data cells into a sequence of lattice-surgery instructions on the data and working cells. Then, we can execute a part of the separated instructions in advance to improve the throughput. A difficult point of this approach is finding an efficient splitting and scheduling method for instructions. Here, we found that the optimization of lattice-surgery scheduling

Algorithm 2 Look-ahead BFS scheduling

Input: Logical-qubit connectivity graph $G = (V, E)$.

Input: List of m instructions of two-body lattice surgery. Each instruction $o \in O$ consists of $o = (v_1, v_2, d)$ where $v_1, v_2 \in V$ and $d \in \{X, Z\}$.

Input: Instruction dependency graph D . See main text for details.

Output: Scheduling paths and timings S and total runtime t .

```
1: Define  $S$  as a length- $m$  list to store the paths and timings.
2: Define  $A$  as a map from  $V$  to  $\{0, 1\}$  and initialize all the elements with 0.
3:  $t \leftarrow 0$ 
4: while not IsEMPTY( $D$ ) do
5:    $L \leftarrow \text{GETEXECUTABLEINDEXLIST}(D)$ 
6:   for  $i$  in  $L$  do
7:      $o \leftarrow O[i]$ 
8:      $p \leftarrow \text{FINDSHORTESTPATH}(o, G, A)$ 
9:     if  $p$  is not empty then
10:       $S[i] \leftarrow (p, t)$ 
11:       $\text{UPDATEDEPENDENCYGRAPH}(D, i)$ 
12:      for  $v$  in  $p$  do
13:         $A[v] \leftarrow 1$ 
14:      end for
15:    end if
16:  end for
17:  for  $v$  in  $V$  do
18:     $A[v] \leftarrow 0$ 
19:  end for
20:   $t \leftarrow t + 1$ 
21: end while
22: return  $(S, t)$ 
```

using this mechanism can be converted into the path search problem in the 3D lattice satisfying several conditions. Thus, we can propose an efficient lattice-surgery scheduling algorithm by proposing path-packing algorithms on the 3D lattice. In this section, we explain this conversion and propose four path-finding algorithms.

This section is constructed as follows. In Sec. 3.2, we show a simple example to motivate readers and clarify our idea using a specific situation. In Sec. 3.3, we theoretically show that a 3D path satisfying several conditions can be systematically translated to a sequence of lattice-surgery instructions that result in a desired logical operation, such as long-range lattice-surgery or CNOT gates. Sec. 3.4 evaluates the effectiveness of 3D routing by establishing the upper and lower bounds for performance improvement. Based on the obtained results, in Sec. 3.5, we will propose a method to search for 3D paths and pack them. The time complexities of the baseline methods and our proposed methods are analyzed in Sec. 3.6. Their performances are numerically benchmarked in Sec. 4.

3.2 Motivating examples

Here, with a simple example, we show that splitting of lattice-surgery instructions enables efficient use of resources, i.e., voxels in the 3D lattice. An overview of our example is shown in Fig. 4. We consider the same situation as Fig. 2. In this situation, the BFS or look-ahead BFS algorithm cannot schedule all the routing at the same code beat, and a hatched path

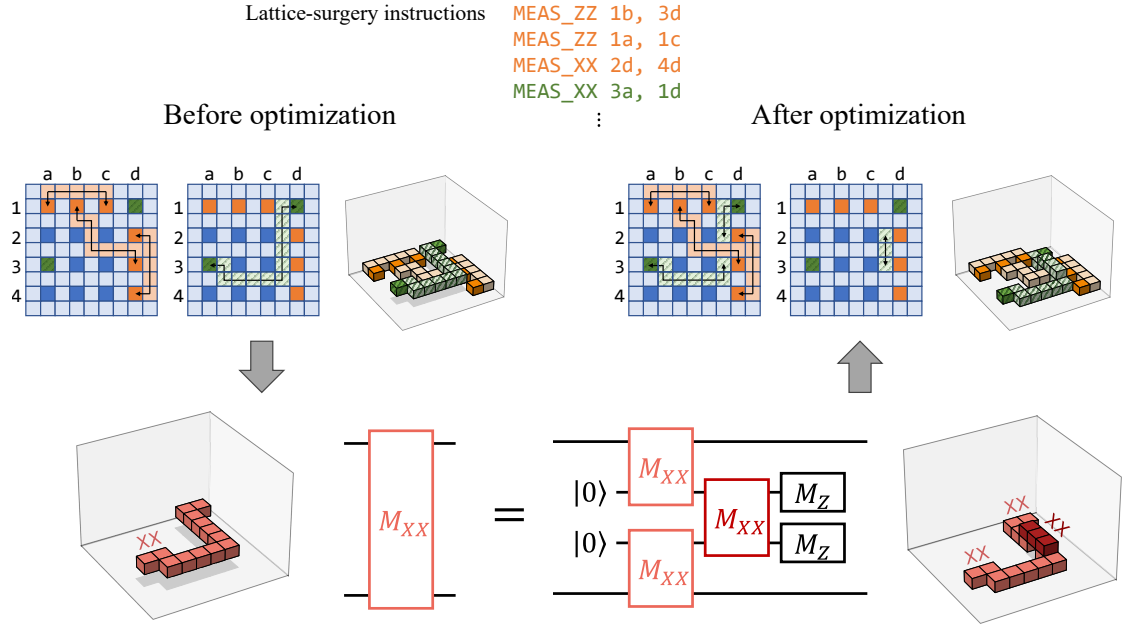


Figure 4: Overview of the optimization by routing lattice-surgery paths more flexibly in a 3D lattice. The equality of quantum circuits holds up to appropriate feedback of Pauli operations.

between green cells must wait for the next code beat. The 3D-lattice representation of this scheduling is shown in the left half of the figure. In this case, voxels below green paths are left unused and lose the parallelism of scheduling.

To mitigate this degradation, we can divide a not-executable instruction into multiple instructions and execute parts of them. As shown in the bottom of Fig. 4, the action of Pauli- XX measurement instruction is equal to that of a chain of three Pauli- XX measurements. This instruction can be achieved with three horizontal routings in the 3D lattice shown in the right-bottom figure. After this modification, voxels unused in the original scheduling are now utilized for instructions, and we can create free voxels for future instructions. Thus, with this technique, we can route a lattice-surgery path more flexibly in the 3D lattices. In other words, if we find a lattice-surgery instruction that cannot be efficiently routed, we can convert the instruction into a chain of them so that a 3D path achieves the chain of instructions and compactly fits into the currently available voxels. This modification enables more efficient use of voxels and allows us to leave more free voxels for future instructions. To utilize this mechanism, we show a general relationship between 3D paths and separated lattice-surgery instructions in the next subsection.

3.3 3D routing for lattice surgery

In this section, we show that a 3D path satisfying several conditions can be converted to a sequence of lattice-surgery instructions having the desired action on the target data cells. To describe the conditions, we introduce the concept of *kink*, which refers to a bending point in the path whose direction is rotated by 90 degrees before and after it moves in the time direction (i.e., Z-direction). For example, the bending point shown in Fig. 5(a) is a kink while bending points shown in Figs. 5(b) and 5(c) are not kinks. Using the concept of kink, we finally show the following theorem.

Theorem 1. *If there is a 3D path that connects X -(Z -)boundaries of cells and has an*

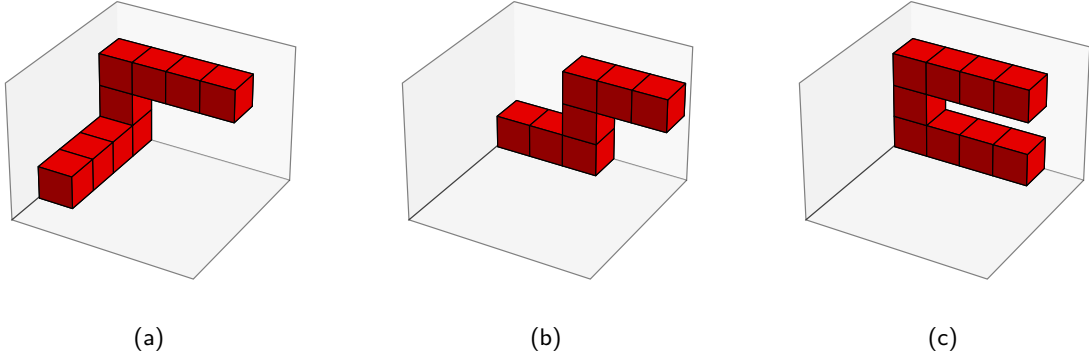


Figure 5: Kink and not-kink examples. In these examples, (a) is an example of a kink, while (b) and (c) are not kinks.

even number of kinks, then there is a sequence of lattice-surgery instructions that consumes the resource corresponding to voxels in the path and results in logical Pauli- XX ($-ZZ$) measurements. Also, if the path connects different types of boundaries and has an odd number of kinks, there is a sequence that results in a logical $CNOT$.

The above theorem implies that we can convert the scheduling optimization into the problem of packing several 3D paths connecting specified cells into a 3D lattice while minimizing its height.

3.3.1 Conversion of 3D paths to quantum circuits

First, we show a procedure for converting a 3D path connecting two logical qubits into a quantum circuit composed of lattice-surgery instructions. Suppose that a 3D path is given connecting two logical qubits with the same types of boundaries. Any 3D path can be split into two types of segments: horizontal and vertical, which are regions moving in the XY -plane and along the Z -axis, respectively.

Each vertical segment is converted to the allocation and deallocation of an ancillary surface-code cell at the position of the segment at the lowest and highest timing of the segment. We note that this corresponds to allocating an intermediate logical qubit to the existing ancillary cell and does not necessitate any additional physical qubits. The ancillary cell is initialized with the state $|0\rangle$ or $|+\rangle$ and eventually measured in the Z or X basis. The basis for the initialization and measurement is determined from the two horizontal segments connected to the vertical segment, which is discussed in the next paragraph. These operations can be executed on a shorter time scale than lattice-surgery operations. Thus, they do not occupy voxels in the 3D lattice.

For each horizontal segment, we assign the XX or ZZ measurement. The basis is determined to be consistent with the direction of cells at the endpoint of the 3D path as follows. Suppose that the 3D path is divided into a sequence of horizontal and vertical segments. Note that the first and last segments are horizontal. We denote the boundary type of the starting point of the 3D path as $A \in \{X, Z\}$. Then, we assign the horizontal path connected to the starting point as the Pauli- AA measurement. This assignment determines the rotating directions of the ancillary cell corresponding to the first vertical segment. Then, whether this vertical segment is a kink or not determines the type of connected boundary of the second horizontal segment. If a vertical segment is a kink, the type of measurements assigned to horizontal segments before and after the vertical segment will be different. Otherwise, i.e., if the direction changes by 0 or 180 degrees

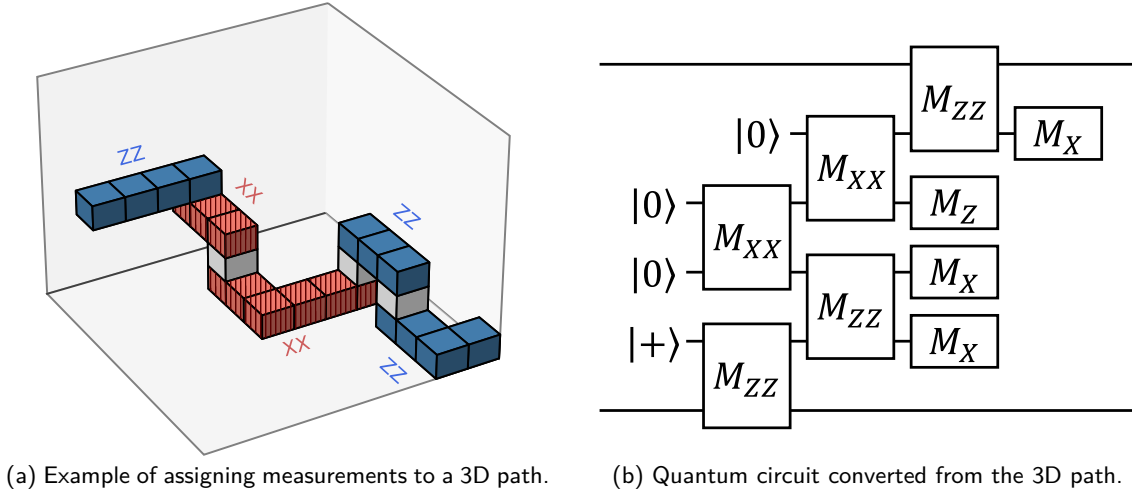


Figure 6: Conversion of a 3D path to a quantum circuit.

before and after the vertical segment, the type of measurements assigned to horizontal segments before and after the vertical segment will be the same. For the next horizontal segment, we assume that it connects the same type of boundary. The measurement types of subsequent horizontal segments and the rotations of vertical-segment cells are assigned by repetitively applying the above rule. This rule will be consistent with the rotation of the other endpoint of the 3D path if the 3D path connects the same type of boundaries and has an even number of kinks. It also becomes consistent if the path connects different types of boundaries and has an odd number of kinks. A concrete example is shown in Fig. 6(a). In this example, there are two kinks.

Next, we assign the initialization and measurement basis for each vertical segment. Each ancillary cell is initialized in a basis different from the basis of the first two-body measurement: an ancillary cell whose first two-body measurement is a Pauli- XX ($-ZZ$) measurement should be initialized to $|0\rangle$ ($|+\rangle$), respectively. Similarly, each ancillary cell is measured in a basis different from the basis of the last two-body measurement: an ancillary cell whose last two-body measurement is a Pauli- XX ($-ZZ$) measurement should be measured in the Z (X) basis, respectively. Consequently, the pre-processing and post-processing of the ancillary cells in a given 3D path can be uniquely determined. The quantum circuit corresponding to the 3D path in Fig. 6(a) is shown in Fig. 6(b).

3.3.2 Simplifying Quantum Circuits

We show that the quantum circuit converted from a 3D path with the presented rule can be simplified to a two-body measurement or a CNOT operation on the data cells at the path endpoints in accordance with whether the number of kinks is even or odd. The quantum circuit converted from a 3D path is a chain of Pauli- XX measurements (M_{XX}) and Pauli- ZZ measurements (M_{ZZ}), as shown in Fig. 6(b). We use the following rules to simplify such a circuit with the feedback of appropriate logical Pauli operations.

1. As shown in Fig. 7(a), a chain of measurements of the same type can be simplified to a single measurement of the same type.
2. As shown in Fig. 7(b), a chain of two measurements of the same type with another in-between measurement of the other type can be simplified to a single measurement of the same type as the two measurements.

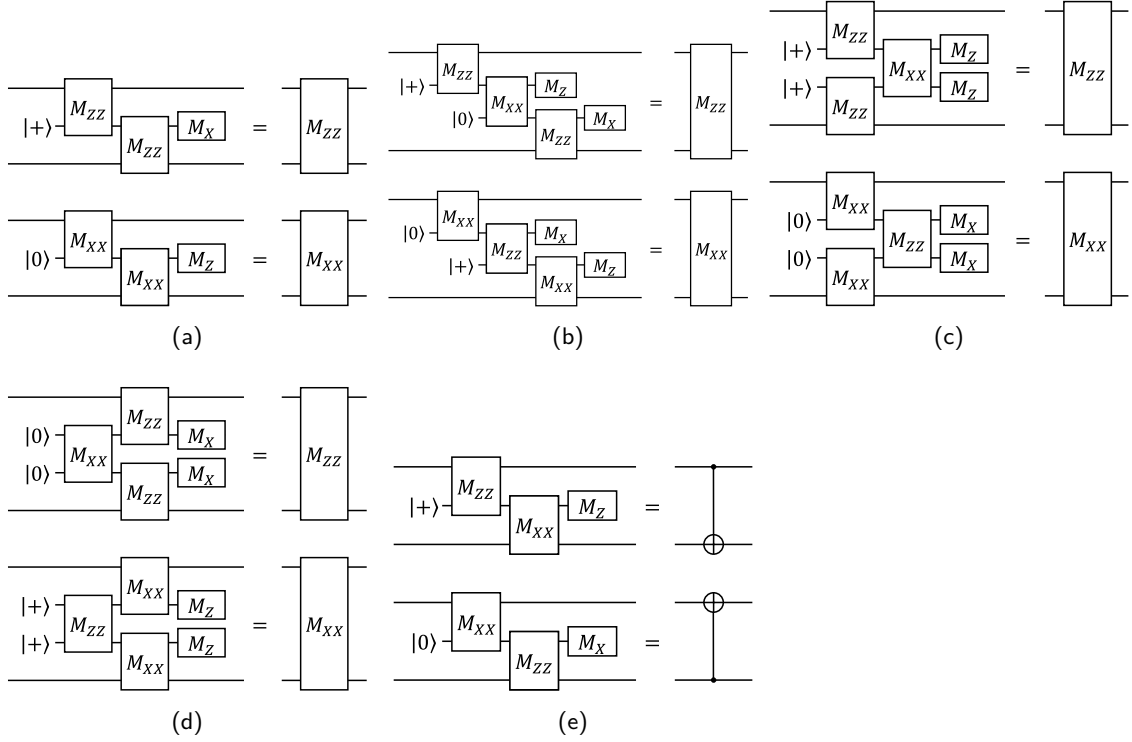


Figure 7: Equivalent processes of two-qubit measurements. (a) Two two-body measurements with the same basis are equal to a single two-body measurement. (b,c,d) Three alternating two-body measurements are equal to a single two-body measurement. (e) Two two-body measurements with different bases are equal to a CNOT gate.

3. As shown in Fig. 7(c), a chain of two measurements of the same type followed by another measurement of the other type can be simplified to a single measurement of the same type as the two measurements.
4. As shown in Fig. 7(d), a chain of two measurements of the same type following another measurement of the other type can be simplified to a single measurement of the same type as the two measurements.
5. As shown in Fig. 7(e), a chain of two measurements of the different types can be simplified to a CNOT operation.

These rules can be verified by simple calculations. Note that while the final state must be corrected by the Pauli operation in accordance with the measurement results, the time required for this correction is negligible since this is achieved via the Pauli frame [8, 23].

Putting these rules together, we can show the proposed theorem by simplifying the circuit with a chain of M_{XX} and M_{ZZ} as follows.

1. By repeatedly applying rule 1, all the sub-sequences consisting of the same type of measurements can be merged into one, so that the different types of measurements alternate in the chain. Note that kinks correspond to changes in measurements. Hence, after the simplification, if the number of kinks is even, the number of measurements is odd; if the number of kinks is odd, the number of measurements is even.
2. Using the rules of 2, 3, and 4, we can reduce the number of measurements by two if the length of a chain is longer than two. Therefore, a circuit composed of an

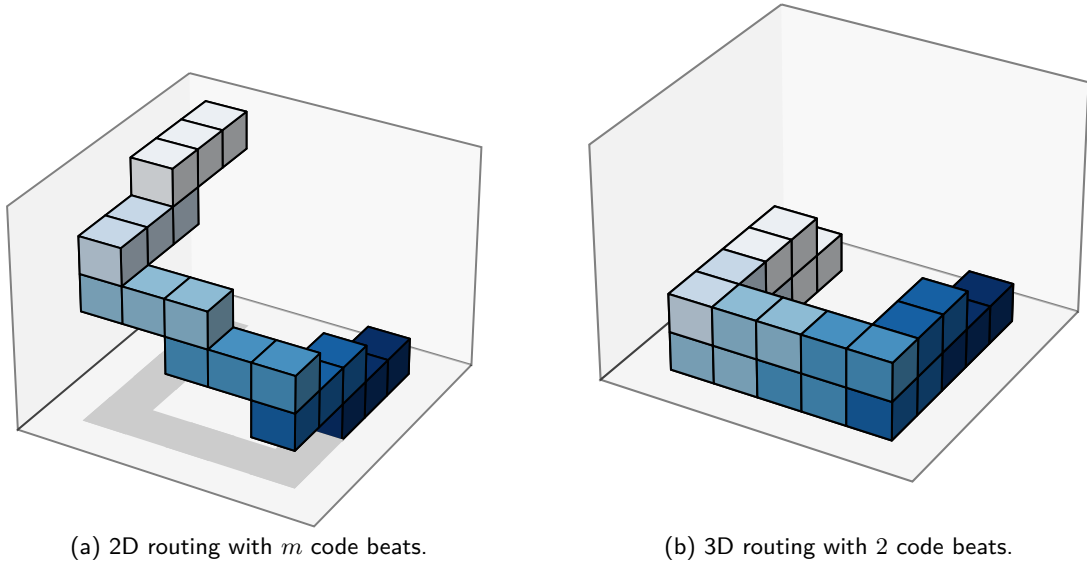


Figure 8: Instruction sequence with a stair-like structure, to which 3D routing is most effective.

even number of measurements can be reduced to two two-body measurements with alternating bases, and a circuit of an odd number of measurements to a single two-body measurement.

3. If a circuit of two measurements is obtained, it is equivalent to a CNOT operation by rule 5, and the direction of the CNOT is determined from the boundary types of the endpoints. If a circuit of a single two-body measurement is obtained, it is equivalent to an XX or ZZ measurement, and the type of measurement is the same as the boundary types of the endpoints.

As a consequence, the circuit can be simplified to a two-body measurement if the path has an even number of kinks, and to a CNOT operation if the path has an odd number of kinks, which leads to the theorem.

3.4 Solution improvement by 3D routing

This section demonstrates the capabilities and limitations of 3D routing through two contrasting examples. The maximum benefits of 3D routing are observed when the input instruction sequence has a stair-like structure (Fig. 8). For instance, consider an instruction sequence of length m where the i -th instruction operates on logical qubits i and $i + 1$. Then, 3D routing achieves a $\Theta(m)$ -fold improvement, as it completes the process in 2 code beats while 2D routing requires m code beats. We emphasize that the subsequent proposed methods aim to replicate this improvement.

Conversely, the benefits vanish when the instruction sequence exhibits a hub-and-spoke structure. Specifically, when the i -th instruction connects logical qubits 1 and $i + 1$, the solution obtained by 2D routing is already optimal, meaning 3D routing offers no additional acceleration.

These examples establish the upper and lower bounds for the performance improvement provided by 3D routing. They also clearly demonstrate that the achievable gain is highly dependent on the input structure. Our numerical experiments in Sec. 4 will quantify the performance improvement by the proposed 3D routing methods on two realistic inputs with distinct structural characteristics.

Table 1: Comparison of path search subroutines between the scheduling methods. These properties remain the same when instruction look-ahead is integrated. As for the time complexity, n denotes the number of cells in the qubit plane, and h denotes the height of the 3D lattice.

Method	Path search space	Path search time complexity		Utilize 3D path
BFS	2D lattice	$O(n)$	(Fast)	No
3D BFS	3D lattice	$O(nh)$	(Slow)	Yes
3D Dijkstra	3D lattice	$O(nh \log nh)$	(Slow)	Yes
Dijkstra projection	2D lattice	$O(n \log n)$	(Fast)	Yes

Table 2: Comparison of the baseline and proposed scheduling methods. As for the time complexity, n denotes the number of cells in the qubit plane, and m denotes the number of instructions.

Method	Time complexity	Execution time	Solution quality	Kink condition
BFS	$O(nm)$	Shortest	Low	Not required
LA BFS	$O(n^2m + m \log m)$	Very Short	Moderate	Not required
3D BFS	$O(nm^2)$	Long	High	Not satisfied
3D Dijkstra	$O(nm^2 \log nm)$	Long	Very High	Not satisfied
Dijkstra proj.	$O(nm \log n)$	Short	Very High	Satisfied
LA Dijkstra proj.	$O(nm \log n + m \log m)$	Short	Highest	Satisfied

3.5 Scheduling algorithms

In this section, we propose methods to schedule a sequence of lattice-surgery instructions using the proved theorem. First, we ignore the kink condition for simplicity and discuss four scheduling methods that achieve the 3D routing of lattice-surgery operations. The first two methods, *3D BFS* and *3D Dijkstra*, are simple extensions of the baseline methods. While they provide scheduling with higher throughput than the baseline method, they need 3D path-finding algorithms and consume longer compilation time. To mitigate this drawback without losing the throughput of instructions, we propose a method named *Dijkstra projection*. This algorithm provides high-throughput scheduling comparable to the 3D variants and achieves reduced time complexity. We also introduce the *look-ahead Dijkstra projection*, which integrates the instruction look-ahead technique to further optimize its scheduling performance. Then, we consider the kink condition and show modifications to the Dijkstra projection algorithms by which the paths satisfy the kink condition. The baseline and proposed scheduling methods are listed and compared in Tabs. 1 and 2. The time complexities in these tables are analyzed in Sec. 3.6.

3.5.1 Breadth-first search extended in the time direction (3D BFS)

We propose a method called 3D BFS, which utilizes time direction for scheduling with a small modification to the BFS algorithm. It searches for a path in a 3D lattice, including the time axis, and connects the target data cells by one of the shortest paths. The BFS algorithm can be used to find the shortest path in the 3D lattice, as in the case of the 2D lattice. When we choose the endpoint voxels of the lattice-surgery path in the 3D lattice, the choice of their Z coordinates is arbitrary. The 3D BFS method adopts the lowest possible voxels as endpoints and then searches for the shortest path connecting them. If there is no path connecting them, the height of the endpoint is raised, and the shortest path connecting them is searched for until it is found. For example, by using the 3D BFS, the scheduling for the case shown in Fig. 2 is improved to a more efficient result shown in Fig. 9. Note that instruction look-ahead is no longer required to mitigate scheduling delays caused by routing failures, since we can sequentially schedule all the instructions.

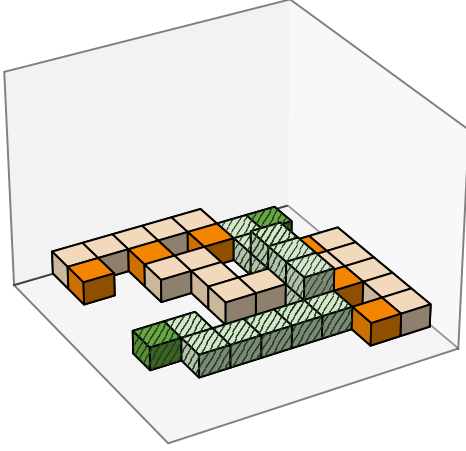


Figure 9: Example of the 3D BFS.

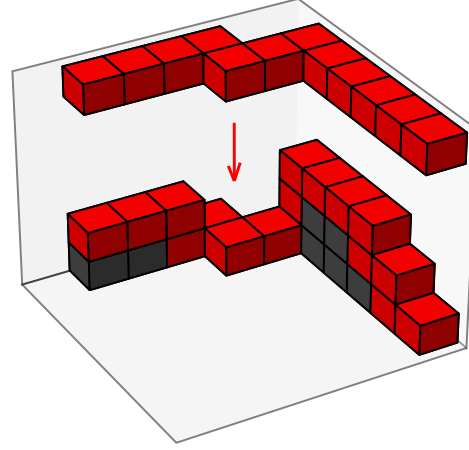


Figure 10: Example of stacking a path in the Dijkstra projection.

The pseudo-code of this procedure is shown in Alg. 3. The dynamic array A maintains whether voxels in the 3D lattice are occupied. The function `FIND3DSHORTESTPATH` returns one of the shortest and lowest paths connecting the target data cells, each of which corresponds to v_1 and v_2 . We can find the path with polynomial time using the BFS.

3.5.2 Dijkstra's algorithm extended in the time direction (3D Dijkstra)

As another method utilizing time direction, we propose the 3D Dijkstra. The basic scheme of the 3D Dijkstra is the same as that of the 3D BFS. The difference is that the 3D Dijkstra assigns the weight to the voxels and finds a path that minimizes the sum of weights. We assign large weights as the height of voxels increases. This weighting prioritizes paths with lower voxels over those with higher voxels, and we can expect that the 3D lattice can be used more densely by packing paths from the bottom. To find the shortest path on a weighted lattice, it is efficient to use a variant of Dijkstra's algorithm for sparse graphs [26]: it runs in $O(V \log V + E)$ time for a general graph with V vertices and E edges.

We heuristically choose the weights as 2^h , where h is the height of the voxel. The reason for using exponential weights is that the ratio of the costs of adjacent layers is constant, regardless of the current height. The reasons for adopting 2 as the base are twofold: the scheduling result was found to be efficient when measured, and the algorithm runs fast because it can be implemented with integer bit shifts.

The pseudo-code of this procedure is almost identical to that shown in Alg. 3. The only difference is that the function `FIND3DSHORTESTPATH` should be replaced with a weighted version that uses Dijkstra's algorithm.

3.5.3 Dijkstra projection

As shown later in the numerical evaluation parts, the 3D BFS and 3D Dijkstra methods are high-performance but time-consuming for practical problem instances, while they are polynomial-time algorithms. They are especially time-consuming for deep circuits because their time complexity of path search depends on the height of the 3D lattice h , as shown in Tab. 1. This motivates us to propose a more time-efficient algorithm, the Dijkstra

Algorithm 3 3D BFS scheduling

Input: Logical-qubit connectivity graph $G = (V, E)$.

Input: List of m instructions of two-body lattice surgery. Each instruction $o \in O$ consists of $o = (v_1, v_2, d)$ where $v_1, v_2 \in V$ and $d \in \{X, Z\}$.

Output: Scheduling paths S and total runtime t .

```
1:  $S \leftarrow []$ 
2:  $A \leftarrow []$ 
3: Append to  $A$  a map with all values set to 0.
4:  $t \leftarrow 0$ 
5: for  $i = 0, \dots, m - 1$  do
6:    $o \leftarrow O[i]$ 
7:    $P \leftarrow \text{FIND3DSHORTESTPATH}(o, G, A)$ 
8:   Append  $P$  to  $S$ 
9:   for  $(v, \tau)$  in  $P$  do
10:     $A[\tau][v] \leftarrow 1$ 
11:   end for
12:   for  $(v, \tau)$  in  $P$  do
13:     if  $\tau = t$  then
14:       Append to  $A$  a map with all values set to 0.
15:        $t \leftarrow t + 1$ 
16:       break
17:     end if
18:   end for
19: end for
20: return  $(S, t)$ 
```

projection. It searches for a 2D path on a 2D lattice and then obtains a 3D path by stacking the 2D path in the time direction, like a projection. While the 3D BFS and the 3D Dijkstra search for a path in a 3D lattice, the Dijkstra projection searches for a path only in a 2D lattice, which is likely to speed up the scheduling process.

The Dijkstra projection determines a pre-projection path in a similar way to the 3D Dijkstra. First, each cell of the qubit plane is weighted exponentially with respect to the heights of the paths already stacked. On the weighted 2D lattice, Dijkstra's algorithm searches for the path with the minimum sum of the weights. The path is then stacked in the 3D lattice along the time axis, preserving its 2D shape while ensuring that adjacent voxels of the path share a face. Fig. 10 shows an example of stacking a path.

The pseudo-code of this procedure is shown in Alg. 4. The map named H corresponds to the height of 2D cells. The function `FINDSHORTESTPATHFROMHEIGHT` weights the graph G with the weight obtained from the height H and returns one of the shortest paths on the weighted graph. The function `LIFTPATH` takes a 2D path and returns a stacked 3D path as shown in Fig. 10.

3.5.4 Look-ahead Dijkstra projection

Instruction look-ahead, introduced in Sec. 2.4, is a technique employed to efficiently manage the set of currently executable instructions. While this technique was initially utilized to mitigate scheduling delays caused by routing failures, we can also leverage it to provide the proposed algorithms with greater flexibility in selecting the next instruction for execution. Specifically, we integrate it with the Dijkstra projection and call it look-ahead Dijkstra

Algorithm 4 Dijkstra projection scheduling

Input: Logical-qubit connectivity graph $G = (V, E)$.

Input: List of m instructions of two-body lattice surgery. Each instruction $o \in O$ consists of $o = (v_1, v_2, d)$ where $v_1, v_2 \in V$ and $d \in \{X, Z\}$.

Output: Scheduling paths S and total runtime t .

```
1:  $S \leftarrow []$ 
2: Define  $H$  as a map from  $V$  to nonnegative integers and initialize all the elements with 0.
3:  $t \leftarrow 0$ 
4: for  $i = 0, \dots, m - 1$  do
5:    $o \leftarrow O[i]$ 
6:    $p \leftarrow \text{FINDSHORTESTPATHFROMHEIGHT}(o, G, H)$ 
7:    $P \leftarrow \text{LIFTPATH}(p, H)$ 
8:   Append  $P$  to  $S$ 
9:   for  $(v, \tau)$  in  $P$  do
10:     $H[v] \leftarrow \max\{H[v], \tau + 1\}$ 
11:     $t \leftarrow \max\{t, \tau + 1\}$ 
12:   end for
13: end for
14: return  $(S, t)$ 
```

projection.

Empirically, it is effective to choose an executable instruction with minimum height, where the height of an executable instruction is defined as the maximum height among its operand cells. We note that this value is static; once an instruction becomes executable, its height will not change until it is executed, because no other instructions will be executed on its operand cells.

Finally, we describe the modifications required to integrate this optimization into the Dijkstra projection. A similar approach applies to our other two algorithms. The integration involves three primary changes:

1. The instruction height serves as the key for the priority queue.
2. Since our 3D routing subroutine never fails, we only need to fetch a single executable instruction. Thus, we replace the function `GETEXECUTABLEINDEXLIST` with a new function, `GETEXECUTABLEINDEX`, which retrieves the highest-priority instruction from the priority queue in $O(\log m)$ time.
3. Then, we modify Alg. 4 by replacing Line 5 with `GETEXECUTABLEINDEX` and inserting `UPDATEDDEPENDENCYGRAPH` immediately after Line 12. It is noteworthy that these changes barely affect the total time complexity, since both functions run in $O(\log m)$ time, which is negligible compared to the $O(n)$ time required for path searching.

3.5.5 Modifications to meet the kink condition

We now discuss how we can let algorithms find paths satisfying kink conditions. Here, we focus on the Dijkstra projection algorithm and its look-ahead variant, and show how to adjust a 3D path of lattice surgery so that the parity of kinks becomes even. Our approach is to add a process to modify 3D paths after projection. We propose a procedure that

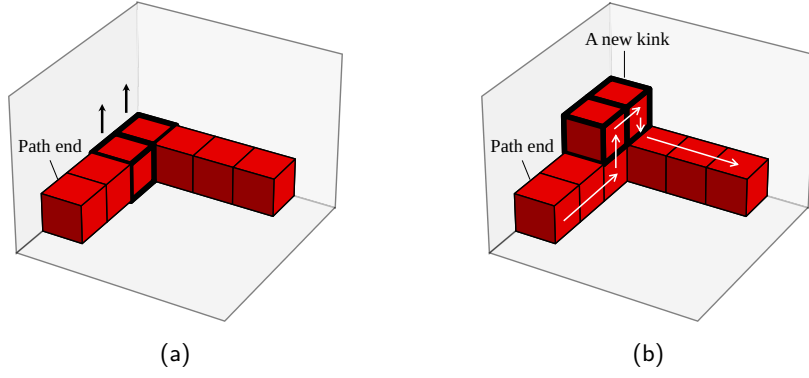


Figure 11: Example of creating a kink provided that the first 90-degree corner in the path is not a kink. (a) Lifting two cells of the non-kink corner creates a kink without affecting the other corners. (b) After the adjustment, a new kink is created without affecting the other corners.

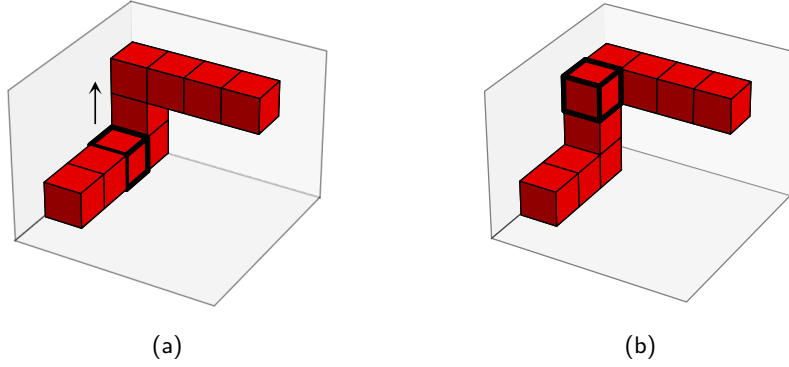


Figure 12: Example of vertically aligning two cells adjacent to a kink. (a) To erase the kink, the lower cell should be raised and aligned with the higher cell. (b) After the alignment, the corner is no longer a kink. However, it may create or erase another adjacent kink, in which case it fails to flip the parity.

flips the kink parity of a projected path, provided that it possesses at least one 90-degree corner. Note that this condition is satisfied when modifying a projected path for an XX or ZZ measurement that has an odd number of kinks. This procedure is employed in the numerical evaluation because it runs in time linear in the path length, which makes it highly efficient.

The core concept for path modification involves lifting corner voxels, as illustrated in Figs. 11 and 12. However, the challenge lies in the fact that this modification might inadvertently create or erase another kink, thereby failing to flip the parity. To circumvent this challenge, we propose the following procedure that works on the first corner in the path.

1. Scan the path, starting from either end, to identify the first 90-degree corner.
2. If the corner is not a kink, lift the corner voxels to create a new kink, as shown in Fig. 11. Since the first corner is chosen, this action does not affect the other corners and thus always flips the kink parity.
3. If the corner is a kink, attempt to flip the parity by aligning it, as shown in Fig. 12. If this adjustment corrects the parity, terminate the procedure.
4. If the parity remains odd after the attempt, apply Step 2 to the non-kink corner that

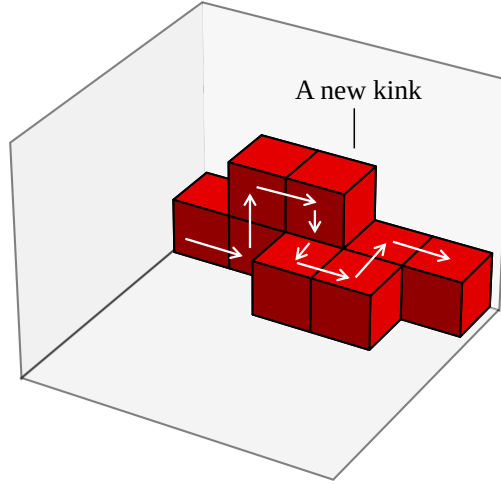


Figure 13: Twisting a straight path to create an extra kink.

results from the previous step. As Step 2 is guaranteed to flip the parity, this action ensures that the parity is flipped.

As this procedure only assumes that the projected path has a corner, it similarly applies to routing logical CNOT gates in most cases. This assumption is fulfilled when, for instance, the data cells are assumed to have the same boundary orientations. We adopted this configuration for our numerical experiment, as described in Appendix B.4.

Conversely, addressing a projected path for a CNOT gate without corners requires a totally different approach, as lifting voxels of such a path will not create any kinks. A viable alternative in this scenario is to twist a path to create a kink, as depicted in Fig. 13. While this modification is minimal, requiring only four extra voxels, it involves nearby cells and demands finding an available space to insert a kink.

There is ample room for further proposals of kink-modification strategies, and comparing these proposals is left for future work.

3.6 Time complexity analysis

This section derives the time complexities shown in Tabs. 1 and 2. We first analyze the time complexity of the path finding subroutines shown in Tab. 1. We begin by recalling that the BFS algorithm and Dijkstra’s algorithm [26] find a shortest path in an unweighted and weighted graph, respectively. Their respective running times for a graph with V vertices and E edges are $O(V+E)$ and $O(V \log V + E)$. Since a 2D lattice with n cells has $V = O(n)$ vertices and $E = O(n)$ edges, the path search subroutines of the BFS and the Dijkstra projection run in $O(n)$ and $O(n \log n)$ time, respectively. Similarly, since a 3D lattice with height h has $V = O(nh)$ vertices and $E = O(nh)$ edges, the path search subroutines of the 3D BFS and the 3D Dijkstra are $O(nh)$ and $O(nh \log nh)$, respectively.

We then analyze the overall time complexity of our scheduling methods, as shown in Tab. 2. Before examining each method, we initially observe that the additional time complexity required for instruction look-ahead is $O(m \log m)$, since both fetching the next executable instruction and updating the dependency graph upon execution require $O(\log m)$ time.

Regarding the BFS, the bottleneck of the iterative procedure is FINDSHORTESTPATH, which runs in $O(n)$ time. Since the failure in routing resets the qubit plane, the number

of calls to `FINDSHORTESTPATH` can be bounded above by $2m$. Thus, the total runtime complexity is $O(nm)$.

We then analyze the runtime of the look-ahead BFS, which does not admit a favorable bound. Unlike in the BFS, the number of times `FINDSHORTESTPATH` is called can only be bounded as $O(nm)$. This bound is obtained because the outer loop iterates $O(m)$ times and the inner loop iterates $|L| = O(n)$ times. Furthermore, this bound may be tight in a pathological case where a single path partitions the qubit plane and subsequent executable instructions attempt to connect two logical qubits lying on different components. Thus, by incorporating the runtime of instruction look-ahead, the total runtime complexity can be bounded by $O(n^2m + m \log m)$. Despite this poor theoretical bound, the algorithm runs almost as fast as the BFS in the numerical experiments.

Next, we analyze the 3D BFS and the 3D Dijkstra simultaneously as they share the algorithm except for the path search subroutine. Since the height h of the 3D lattice can be evaluated as $O(m)$, the path search subroutine runs in $O(nm)$ for the 3D BFS and $O(nm \log nm)$ for the 3D Dijkstra. Since the routing subroutine never fails to find a path, the number of iterations is m , resulting in the total runtime of $O(nm^2)$ for the 3D BFS and $O(nm^2 \log nm)$ for the 3D Dijkstra.

Lastly, we analyze the time complexity of the Dijkstra projection and its look-ahead variant. As shown in Alg. 4, the overall time complexity of the Dijkstra projection is dominated by the function `FINDSHORTESTPATHFROMHEIGHT`, which executes in $O(n \log n)$ time. We note that the function `LIFTPATH` will not be a bottleneck, since it runs in time linear to the path length $|p|$, even when modifying the kink parity to satisfy the condition. Given that the path length $|p|$ can be bounded above by n , this procedure runs in $O(n)$ time. Consequently, we have a total time complexity of $O(nm \log n)$ for the original algorithm and $O(nm \log n + m \log m)$ when combined with instruction look-ahead.

4 Performance evaluation

In this section, we evaluate the proposed methods (3D BFS, 3D Dijkstra, Dijkstra projection, and look-ahead Dijkstra projection) and compare them with the baseline methods (BFS and look-ahead BFS). We used random circuits and circuits compiled from quantum phase estimation algorithms for benchmarking. For quantum phase estimation, we first converted the circuit into an instruction sequence consisting of Pauli- XX and ZZ measurements and compared the performance of our methods in Sec. 4.2. We also compared our methods with the existing algorithm for large-scale compilation, the Edge-Disjoint Paths Compilation (EDPC) algorithm [21], in Sec. 4.3. To satisfy their requirements and reproduce their experimental settings, we converted the circuit into another instruction sequence consisting of CNOT operations and T -gates and computed the scheduling results on their qubit plane pattern. See each section and Appendix B.4 for further details of these conversions and settings.

We compare these methods using two metrics. For a performance measure of the scheduling results, we used *throughput*, the average number of instructions processed in parallel per code beat. The other one is execution time, which is the required time for executing scheduling algorithms on classical computers. We evaluated the execution times on a computer equipped with an 11th Gen Intel(R) Core(TM) i5-1135G7 @ 2.40 GHz with 16 GB memory. The scheduling program was implemented in C++ and optimized with the `O3` option using `g++` (Ubuntu 13.3.0-6ubuntu2~24.04) 13.3.0. For comparison with the EDPC algorithm, we also benchmarked the existing Julia code of the EDPC algorithm. We ran it on Julia Version 1.12.1 (2025-10-17) and measured the execution time with the

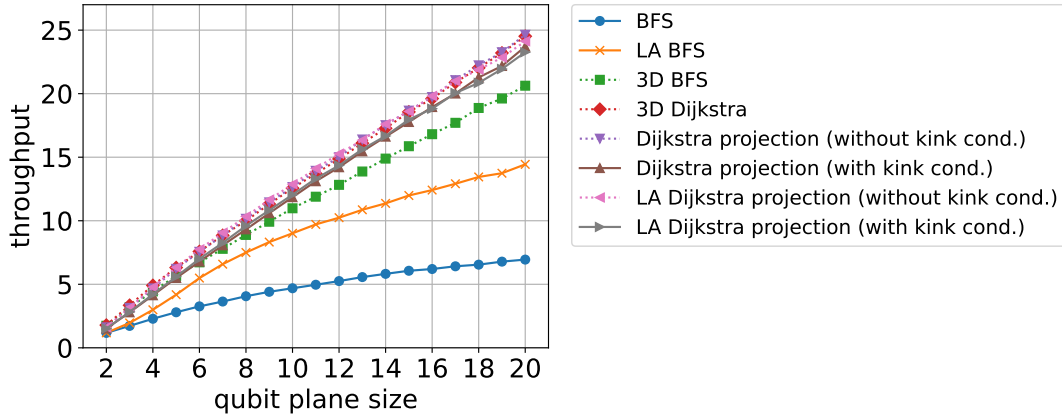


Figure 14: Throughput evaluation on random circuits.

@btime macro.

4.1 Random circuits

For the random circuit benchmark, we set the number of total instructions to 1000, and the two logical qubits connected by each instruction were chosen uniformly at random. The data cells were placed in locations where the rows and columns are both even-numbered, as shown in Fig. 1. The number of logical qubits with data lined up on one side of the square qubit plane was defined as *qubit plane size*. For example, Fig. 1 corresponds to the example of a qubit plane size equal to four. The performance was evaluated by changing the qubit plane size from 2 to 20. For evaluating the throughput and the execution time, we evaluated methods with ten random circuits generated with different seed values and averaged the results.

We first evaluate the throughput of the baseline and proposed methods. The results are shown in Fig. 14. In this figure, each color corresponds to the performance of the proposed and baseline methods. Here, we emphasize that the performance of the four methods shown as dotted lines in the figure, 3D BFS, 3D Dijkstra, Dijkstra projection (without kink condition), and look-ahead Dijkstra projection (without kink condition), do not satisfy the kink conditions. They are shown as a performance reference, and only the plots of the baseline methods, the Dijkstra projection (with kink condition), and the look-ahead Dijkstra projection (with kink condition) are valid. Since the optimization of the Dijkstra projection by instruction look-ahead hardly impacted throughput or execution time when evaluated on random circuits, we refer to both the original and optimized versions simply as Dijkstra projection throughout this setting.

The methods 3D Dijkstra and Dijkstra projection had the best throughput, followed in order by 3D BFS, look-ahead BFS, and BFS. The small discrepancy between the Dijkstra projection and the 3D Dijkstra suggests that the limited search space of the Dijkstra projection does not significantly affect the quality of the obtained solution. We find that the penalty of considering the kink condition is negligible since the performance difference of the Dijkstra projection with and without considering kink conditions is small. In terms of the increase in throughput as the qubit plane size increased, the best baseline method, look-ahead BFS, showed a little plateauing, while the Dijkstra projection showed no such trend and increased at a similar rate. The throughput of the Dijkstra projection was on average 1.5 times higher than that of the look-ahead BFS, but the improvement is likely

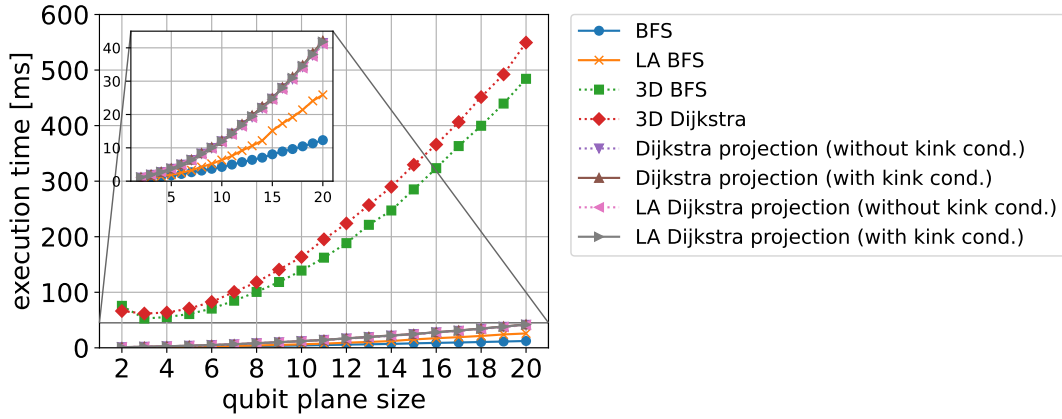


Figure 15: Evaluation of execution time on random circuits.

to be even greater as the qubit plane size increases based on the above observation.

Next, we evaluated the execution times, which are shown in Fig. 15. We observed that the Dijkstra projection and two baseline methods show better scaling compared to the 3D BFS and the 3D Dijkstra. This is because the first three methods search on the 2D lattice, while the other two search on the 3D lattice. These results are consistent with the expected properties of the algorithm design. We also observed that there is almost no difference between the execution times of the Dijkstra projection with and without considering kink conditions. Thus, the time for adjusting kink parity is negligible compared to the path-finding part.

4.2 SELECT circuits

We then evaluated the performance for practical applications. Here, we chose the SELECT circuit as a benchmark target. The SELECT operation is one of the bottleneck components in quantum phase estimation algorithms using qubitization [10, 11, 22]. Its action is defined from Hamiltonian $H = \sum_{i=0}^{L-1} \alpha_i P_i$ as follows:

$$\sum_{i=0}^{L-1} |i\rangle \langle i| \otimes P_i, \quad (1)$$

where L is an integer, $|i\rangle$ is a computational basis of an L -dimensional Hilbert space, and P_i is a Pauli operator. In this benchmark, a SELECT circuit is synthesized for a Hamiltonian of a 16×16 2D Heisenberg model (i.e., 256 spins) with the nearest-neighbor interaction. There are several ways to translate SELECT circuits into a sequence of basic operations. We used the methods proposed by Yoshioka *et al.* [11] and generated SELECT circuits with two configurations, which we named SELECT 0 and SELECT 5. These suffixes indicate the parallelism parameter, where SELECT 0 is a serial implementation of the SELECT circuit and SELECT 5 is parallelized into $2^5 = 32$ threads. SELECT 5 requires more logical qubits and instructions than SELECT 0, but ideally has a larger throughput due to parallelization. SELECT 0 required 25 qubit plane size and 59,344 instructions, and SELECT 5 required 55 qubit plane size and 61,492 instructions. We note that these instances are solved by quantum computers sufficiently faster than the state-of-the-art algorithm for classical computing [11]. Thus, the performance for this instance reflects the realistic scenario of practical use of FTQCs.

Table 3: Throughput evaluation on SELECT circuits.

	throughput	
	SELECT 0	SELECT 5
BFS	1.59	1.76
look-ahead BFS	1.77	20.74
Dijkstra projection (without kink condition)	2.90	60.58
Dijkstra projection (with kink condition)	2.90	57.68
look-ahead Dijkstra projection (without kink condition)	2.90	80.28
look-ahead Dijkstra projection (with kink condition)	2.90	78.43

Table 4: Evaluation of execution time on SELECT circuits.

	execution time [s]	
	SELECT 0	SELECT 5
BFS	0.58	2.78
look-ahead BFS	0.65	6.84
Dijkstra projection (without kink condition)	3.55	26.07
Dijkstra projection (with kink condition)	3.57	26.56
look-ahead Dijkstra projection (without kink condition)	3.60	19.55
look-ahead Dijkstra projection (with kink condition)	3.59	19.75

In the following evaluation, we compare four methods: BFS, look-ahead BFS, Dijkstra projection, and look-ahead Dijkstra projection. We omitted the results of the 3D BFS and the 3D Dijkstra since they cannot finish the compilation within 1 hour.

The throughput of the four methods is shown in Tab. 3. The Dijkstra projection and its optimized variant show better throughput than the two baseline methods. Compared to the look-ahead BFS, the look-ahead Dijkstra projection shows improvement by a factor of 1.6 for SELECT 0 and 3.8 for SELECT 5. We observed higher throughput improvement for SELECT 5, which is an expected result considering that SELECT 5 has a structure like a parallel circuit and leaves much room for parallelization. We can see that the degradation by considering kink conditions is at most 5%, which is not negligible, but does not lose the advantage of the Dijkstra projection.

The execution times of the methods are shown in Tab. 4. The order of execution times of the four methods for the SELECT circuits was basically the same as for the random circuit. On the other hand, the look-ahead Dijkstra projection demonstrated a significant speedup compared to the original Dijkstra projection when applied to SELECT 5. Further profiling indicated that these execution times were dominated by path searching, which strongly suggests that the instruction look-ahead technique results in substantially earlier termination of the path search subroutine. This is likely because fetching the instruction with the lowest height by instruction look-ahead flattens the current scheduling outcome with less biased height distribution, thereby shortening the subsequent lattice-surgery paths and the time required to find them.

All four methods finished in practical execution time. The additional time for considering kink conditions is at most 2%. Thus, the time for considering kink conditions is also negligible in this case. Also, when applied to the SELECT 0 circuit, the 3D BFS and the 3D Dijkstra cannot finish within an hour while the Dijkstra projection finishes within a few seconds, again indicating that the Dijkstra projection excels in execution time among

Table 5: Throughput evaluation on SELECT circuits. The grayed rows annotated with an asterisk represent reference values that disregard the kink condition, optimistically assuming that CNOT operations complete via a 2D path in one code beat. Although these raw infeasible values are used to compare net routing performance, halving these grayed values will provide the corresponding feasible values, reflecting that CNOT operations realistically require two code beats. The remaining methods adhere to the kink condition and yield a feasible solution.

	throughput	
	SELECT 0	SELECT 5
EDPC*	1.09	9.06
EDPC	0.66	4.74
BFS*	1.06	1.26
look-ahead BFS*	1.12	17.26
Dijkstra projection (without kink condition)*	1.85	17.34
Dijkstra projection (with kink condition)	1.85	15.48
look-ahead Dijkstra projection (without kink condition)*	1.85	36.29
look-ahead Dijkstra projection (with kink condition)	1.85	31.39

the methods that utilize the time direction.

4.3 Comparison with the existing methods

In the following evaluation, four of our methods—BFS, look-ahead BFS, Dijkstra projection, and look-ahead Dijkstra projection—were compared with the Edge-Disjoint Paths Compilation (EDPC) algorithm [21]. We adopted a problem setting that satisfies the requirements of the EDPC algorithm. Specifically, the instruction sequence consists solely of CNOT gates and T -gates. Each T -gate is realized via T -gate teleportation between the target cell and a magic state factory cell aligned at the perimeter of the qubit plane. (For further details, see Appendix B.4.) Under this setting, SELECT 0 required 17 qubit plane size and 37,856 instructions, and SELECT 5 required 30 qubit plane size and 39,050 instructions.

The throughput of these methods is shown in Tab. 5. The grayed results represent configurations that violate the kink condition, rendering them infeasible. Since a CNOT gate can be implemented via a 2D path using two code beats [21], these entries can be converted to feasible counterparts by halving the values. However, the subsequent discussion compares the raw values to evaluate routing efficiency without this overhead. Notably, since our proposed methods satisfy the kink condition with modest overheads, accounting for the feasibility would effectively double the performance advantage, making our results even more favorable compared to the baseline methods.

For the performance of the EDPC algorithm, it lies between that of the BFS and the look-ahead BFS. This is due to their different strategy of managing instruction dependency. Their algorithm initially applies topological sorting to the instruction dependency graph and repeatedly removes the instructions that have no dependent instructions, thereby decomposing the instructions into a sequence of instruction sets consisting of mutually independent instructions. This partitioning enables a parallel compilation of the instruction sets and facilitates designing a scheduler that handles many types of instructions, whereas it also worsens the scheduling performance by effectively adding “barriers” in the circuit. We consider this to be one of the reasons why the look-ahead BFS outperforms EDPC, although their routing strategy is more sophisticated than our baseline methods.

Although the overall trends of our methods are consistent with the results in Sec. 4.2,

Table 6: Evaluation of execution time on SELECT circuits. The grayed row annotated with an asterisk presents reference values, as this algorithm was implemented in Julia, whereas our methods are implemented in C++.

	execution time [s]	
	SELECT 0	SELECT 5
EDPC*	22.19	207.52
BFS	0.31	0.90
look-ahead BFS	0.32	1.47
Dijkstra projection (without kink condition)	1.27	4.81
Dijkstra projection (with kink condition)	1.28	4.83
look-ahead Dijkstra projection (without kink condition)	1.28	3.71
look-ahead Dijkstra projection (with kink condition)	1.29	3.74

it is worth mentioning that the instruction look-ahead technique proved exceptionally effective in this instance. This almost closed the gap between the look-ahead BFS and the Dijkstra projection, and resulted in an outstanding performance by the look-ahead Dijkstra projection.

This gap reduction can also be attributed to the difference in CNOT gate handling. The former experiment decomposes CNOT gates into ZZ and XX measurements, which partially yields stair-like structures and is likely advantageous for 3D routing. However, we emphasize that the same benefits can also be observed in this experiment when we compare feasible throughputs by halving the grayed ones: the Dijkstra projection showed 1.8-fold improvement over the look-ahead BFS, and the look-ahead Dijkstra projection demonstrated 3.6-fold improvement.

The execution times of the methods are shown in Tab. 6. The substantial execution time of EDPC may stem from several reasons. We first note that their algorithm was implemented in Julia, whereas ours was implemented in C++. Although we used `@btime` to warm up the program and exclude just-in-time compilation time, this difference likely introduced overhead to their execution time. Another factor is their use of maximum flow subroutines in T -gate teleportation, which may have incurred high computational cost.

4.4 Discussion

First, we discuss characteristics seen both in the random circuits and the SELECT circuits. For throughput, the BFS, the look-ahead BFS, the 3D BFS, and the 3D Dijkstra performed well in that order, with the Dijkstra projection performing as well as the 3D Dijkstra. The look-ahead optimization in the Dijkstra projection had no impact on the random circuits, while it proved effective in SELECT circuits. It is natural that the look-ahead BFS outperforms the BFS, considering that the look-ahead BFS executes future operations and produces more spare space that would be blocked in the BFS. The primary reason the 3D BFS outperforms the look-ahead BFS is its ability to pack paths more densely into the lattice by increasing the dimension of the paths by one. Another reason is that a simple look-ahead can be reproduced by searching in 3D. The reason the 3D Dijkstra outperforms the 3D BFS is that the paths can be packed more densely into the lattice in the end by keeping the overall path low, even if it is a bit roundabout, rather than unnecessarily shortening the path by going through an excessively high layer. The reason the Dijkstra projection performs comparably to the 3D Dijkstra is likely that the paths adopted by the 3D Dijkstra tend to be along the bottom surface, as in the Dijkstra projection.

In terms of execution time, the BFS was the shortest, the look-ahead BFS was slightly longer than the BFS, the Dijkstra projection was a few times longer, and the 3D BFS and the 3D Dijkstra were much longer than the others. From these results, execution time is significantly affected by whether the search space is a 2D or 3D lattice. The fact that the look-ahead BFS is slightly slower than the BFS is a natural consequence of the additional cost of instruction look-ahead. The reason the Dijkstra projection is several times slower than the BFS and the look-ahead BFS is that, as a comparison of search algorithms, Dijkstra’s algorithm is more computationally expensive than the BFS algorithm. The same argument holds for the 3D BFS and the 3D Dijkstra.

4.5 Resource overhead analysis

This section analyzes the overhead associated with introducing the concept of 3D routing and scheduling algorithms based on this idea, such as the Dijkstra projection. We categorize the potential penalties into four types: compilation-time overhead, error-rate overhead, qubit-count overhead, and architectural overhead.

The quantitative compilation-time overhead was shown in the previous section as execution time. The results indicate that employing the Dijkstra projection with kink constraints increases the compilation time per instruction by a factor of a few. For the SELECT instances, which are large enough to demonstrate quantum advantage [11], the compilation completes in approximately half a minute. This duration is negligible compared to the total execution time, as the quantum phase estimation with the corresponding SELECT circuit on a real FTQC would take over an hour [11]. Even for larger instances, both the compilation time with the Dijkstra projection and the execution time of an actual machine scale almost linearly with the number of instructions, as summarized in Tab. 2. Therefore, the use of the Dijkstra projection introduces no practical overhead in regimes where quantum computation is meaningful.

A potential overhead in using the Dijkstra projection is an increase in the error rate. Let the number of voxels in the 3D representation be denoted as the *active volume*. Because 3D routing might entail longer routing paths than 2D routing, the active volume can be larger. We denote the probability of maintaining a single logical qubit fault-tolerantly for a single code beat as p_{voxel} and the active volume as A . Then, under the assumption that p_{voxel} is sufficiently small and that errors are largely independent, the overall failure probability can be approximated as $p_{\text{voxel}}A$. Hence, the active volume serves as a semi-quantitative proxy of the overall failure probability, provided that the errors are approximately independent and that p_{voxel} is low enough to keep the overall failure risk within a practical range. We evaluated the increase in active volume using 3D-routing-based methods for the SELECT circuits, as shown in Tab. 7. As a conservative analysis, our proposed method satisfying the kink condition is compared with the baseline method disregarding the kink condition. The results show that, for the SELECT 0 instance, the look-ahead Dijkstra projection with kink condition increases the active volume by about 30% compared to the BFS without kink conditions. Conversely, the volume decreases for the SELECT 5 instance. We expect this is because improved instruction throughput due to 3D routing reduces the idle time of logical qubits. As discussed below, the total program failure rate can be kept constant by increasing the qubit count, and a 30% increase in active volume can be compensated with a modest cost.

Next, we examine the qubit-count penalty. The total number of physical qubits in an FTQC system is the product of the number of logical-qubit cells in the qubit plane and the number of physical qubits per logical-qubit cell. In surface codes, the latter is $2d^2 - 1$, where d is the code distance. The introduction of 3D routing does not increase the

Table 7: Active volume evaluation on SELECT circuits. The grayed rows annotated with an asterisk represent reference values that disregard the kink condition, optimistically assuming that CNOT operations complete via a 2D path in one code beat. Although these raw infeasible values are presented to show their net routing performance, doubling these values will provide the corresponding feasible values, reflecting that CNOT operations realistically require two code beats. The remaining methods adhere to the kink condition and yield a feasible solution.

	active volume ($\times 10^6$)	
	SELECT 0	SELECT 5
EDPC*	10.44	5.46
EDPC	17.25	9.89
BFS*	13.25	32.54
look-ahead BFS*	12.51	3.42
Dijkstra projection (without kink condition)*	17.47	7.35
Dijkstra projection (with kink condition)	16.43	8.21
look-ahead Dijkstra projection (without kink condition)*	17.45	3.45
look-ahead Dijkstra projection (with kink condition)	16.58	3.99

number of logical-qubit cells. Although 3D routing temporarily allocates logical qubits to separate logical instructions, they are assigned to existing cells. In contrast, the number of physical qubits per logical-qubit cell may increase. If the active volume increases, the error-correction failure rate per voxel p_{voxel} must be reduced to maintain a constant overall program failure probability, which requires increasing the code distance. The error-estimation failure probability per syndrome-measurement cycle on a single cell is approximated as $0.1(100p)^{(d+1)/2}$ [18], where p is the physical error rate per cycle. Consequently, the failure probability per voxel (per d cycles) is $p_{\text{voxel}} = 0.1d(100p)^{(d+1)/2}$. Assuming $p = 10^{-3}$ and d is sufficiently large, increasing (decreasing) the code distance by two allows for roughly 10 times larger (smaller) active volume. In practical regimes, the code distance d is typically 23 or higher [11]. Increasing d from 23 to 25 corresponds to approximately an 18% increase in total physical qubits. As discussed earlier, 3D routing can either increase or decrease the active volume by a few times, which corresponds to at most an 18% variation in total physical qubits.

Finally, we consider architectural overheads when implementing 3D routing in FTQC architectures. We emphasize again that the number of logical-qubit cells remains unchanged by 3D routing. Therefore, if an FTQC machine is capable of handling all logical-qubit cells simultaneously, our method introduces no architectural overhead. In extreme cases where the total number of logical qubits n is very large, it might be possible that FTQC machines cannot store n logical qubits due to limitations in symbol handling, error-estimation throughput, or synchronization. However, such situations are unlikely, at least in near- and mid-term FTQCs, as achieving quantum advantage typically requires at most 10^3 logical qubits. Assuming a code distance of $d = 23$, one code beat, i.e., the latency of a lattice-surgery instruction, corresponds to $23 \mu\text{s}$ or longer if we assume the code cycle is $1 \mu\text{s}$ [11]. Managing 10^3 symbols and logical qubits, as well as loading pre-compiled paths from memory every $23 \mu\text{s}$, is a feasible requirement for classical control hardware.

4.6 Robustness under Resource Pressure

To evaluate the robustness of our proposed methods under high resource contention, we examined their performance across circuits with varying gate densities. We define the *average gate density* as the number of instructions divided by the circuit depth, which

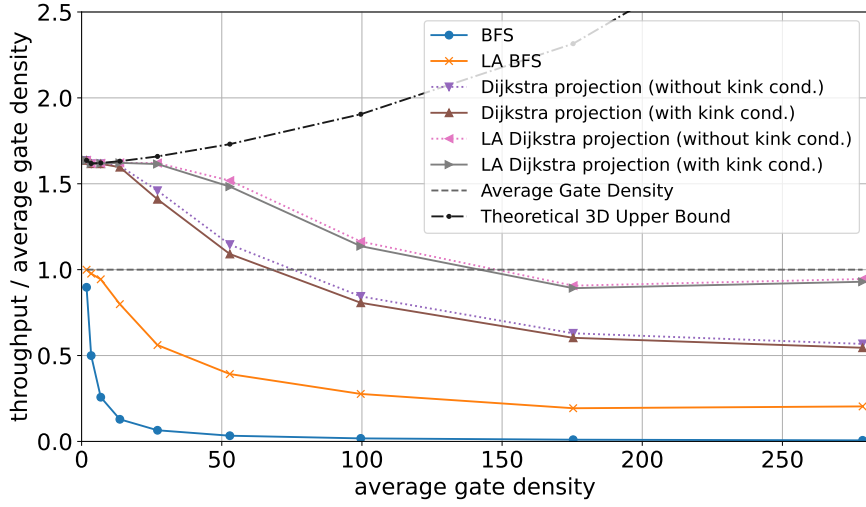


Figure 16: Degradation curves of throughput as a function of gate density. The performance is measured by the throughput normalized by the average gate density, which corresponds to an ideal throughput of conflict-free 2D routing.

quantifies the congestion level of the problem instance. This metric also serves as a theoretical upper bound for the throughput of the 2D routing methods, BFS and look-ahead BFS. As discussed in Sec. 4.2, we utilized the SELECT circuits to generate different levels of congestion by varying the parallelism parameter from 0 to 8: SELECT 0 corresponds to a serial implementation, and SELECT i is parallelized into 2^i threads.

Fig. 16 illustrates the performance degradation of the scheduling methods as resource pressure intensifies. The performance is quantified by the throughput normalized by the average gate density, which is equivalent to the ratio of the circuit depth to the total execution time in code beats. To assess the impact of routing conflicts in the proposed methods, we also included the normalized performance of the conflict-free 3D routing, that reserves only path endpoints and ignores all intermediate segments. This benchmark serves as a theoretical upper bound on the performance of 3D routing methods.

As expected, all methods exhibit a decline in efficiency as gate density increases. Notably, at low gate densities, the proposed methods achieve near-optimal performance, outperforming the circuit depth. This can be attributed to their advantages in stair-like structures, as discussed in Sec. 3.4. Increasing gate density leads to performance degradation across all routing methods, resulting in a growing discrepancy compared to the theoretical 3D upper bound due to intensified path conflicts. However, it is noteworthy that the proposed methods remain comparable to the circuit depth, demonstrating their resilience to high resource pressure.

Fig. 17 shows the active volume of the routing methods in the same numerical experiment. While all methods experience degradation, the proposed methods maintain an active volume close to that of the look-ahead BFS across different gate densities. Furthermore, they outperform the look-ahead BFS in higher densities due to their shorter execution time in code beat.

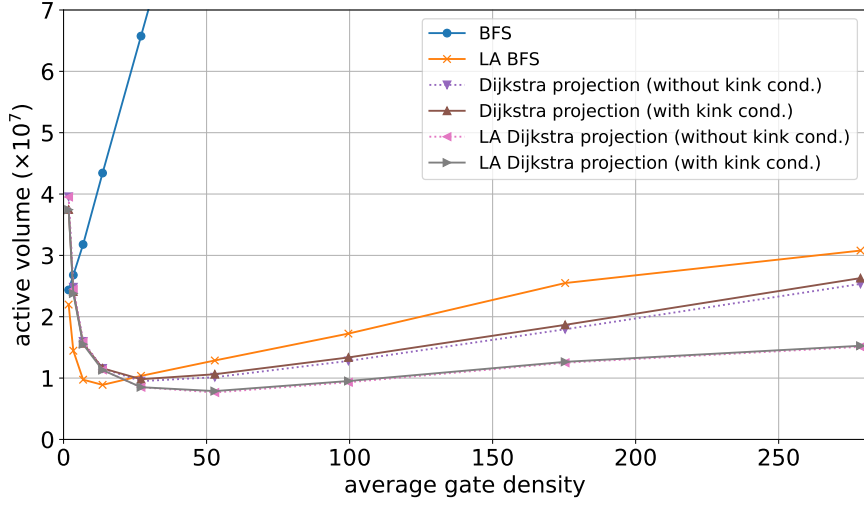


Figure 17: Degradation curves of active volume as a function of gate density.

5 Conclusion and future outlook

In this paper, we proposed efficient and high-performance algorithms for lattice-surgery instructions. We showed that the mapping of lattice-surgery instructions can be reduced to 3D path search problems. Based on this mapping, we proposed the Dijkstra projection and its look-ahead variant to perform efficient scheduling of lattice-surgery instructions. We evaluated their performance on several settings, and they showed significant improvements in throughput compared to the baseline, while the increase in execution time is modest.

In our numerical experiment, only the Pauli- XX and ZZ measurements were considered, and other logical operations were ignored for simplicity. However, we need to consider other types of instructions in practice, such as single-qubit Clifford gates, lattice-surgery instructions acting on many logical-qubit cells, and magic-state teleportations. In particular, many-body lattice-surgery operations are vital since they are used to remove Clifford unitary operations from the circuits [18] and used in several existing works [18, 27]. In Appendix A, we provide the extension of our techniques to many-body lattice surgery.

One may also be concerned that logical-qubit cells are blocked during the reaction time in magic-state teleportation. Typical T -gate teleportation circuits require S -gate feedback according to the measurement results, and we cannot perform operations on cells that may be targets of the feedback during the latency. Nevertheless, we expect this is not a problem since we can relax this blocking by using auto-corrected T -gate teleportation [18, 28], which converts feedback operations from S -gate on the computing space to the choice of Pauli measurement basis on the ancillary space.

Combining these theoretical ideas and providing consistent and full-stack compilation frameworks, such as Refs. [9, 15, 29, 30], would be the next direction of this topic. To illustrate the contexts intended by our method, we describe an example full-stack compilation workflow in Appendix B. As explained in this appendix, our method is compatible with lattice-surgery-based FTQC pipelines under the following four assumptions. (1) The positions and connectivity of physical qubits are fixed and cannot be changed during computation. (2) Logical qubits are encoded using surface codes, and logical operations are implemented via code deformation and lattice surgery. (3) All required surface-code logical qubits are allocated at the beginning of execution and are not deallocated during program

execution. (4) Magic states are generated and managed within abstracted modules placed at fixed locations, and we ignore probabilistic behavior at runtime for simplicity. We believe that these assumptions are common when evaluating architectures with superconducting qubits [9, 31, 32], and the third and fourth assumptions would be removed as they are assumed to simplify the compilation and evaluation process. On the other hand, the other two assumptions do not necessarily hold when integrating recent technologies such as qubit shuttling in neutral-atom platforms [33], non-topological or concatenated quantum error-correcting codes [34–36], and generalized lattice-surgery operations for such codes [37]. Extending our method to a wider range of compilation frameworks and evaluating the performance of our methods in broader situations are expected, which are left for future work.

There are several other directions for future work. For the voxel weighting in the 3D Dijkstra and the Dijkstra projection, we proposed the weighting that worked best among those we tried, but we expect that there is room for further improvement. Also, the theoretical upper and lower bounds of the idea, separating large instructions into small fragments, may be calculated in more detail in the future. By comparing these bounds with our numerical results, we can measure the room for improvement. While we introduced a method to satisfy the restriction of kink parity for lattice-surgery instructions, there may be a more efficient way to control the number of kinks. Integrating kink restrictions in the path-finding algorithms more naturally would be another future topic.

Integrating our method into distributed FTQC would also be essential. The integration method strongly depends on the paradigm of distributed FTQC. If the number of qubits per node is sufficiently small, logical qubits are constructed through communication among several nodes [38] or realized via measurement-based FTQC [39, 40]. In such cases, the routing problem is equivalent to that in the single-node scenario, and our method can be applied without modification. The situation becomes more complex when the number of qubits per node is moderate. In this regime, each node can host several logical qubits, while two-qubit logical operations are performed via logical entanglement distribution and teleportation [41, 42]. Since this inter-node teleportation can be regarded as a sparse connection between 3D voxels, we believe our method can be extended to such cases. Nevertheless, the detailed implementation of distributed FTQC architectures remains immature and will be an important subject for future work.

The results of this research extend the lattice-surgery scheduling problem, which has been addressed as a graph problem in 2D space, to a 3D path search by adding a time direction axis. Therefore, the present results not only enable computation on fault-tolerant quantum computation with higher efficiency than previously possible but also suggest that graph algorithms can be applied to optimize the compilation of programs for fault-tolerant quantum computation. Combining the optimization methods obtained in the present study with the optimization of quantum circuits and the optimization of the placement of logical qubits, and appropriately formulating these schemes into graph problems, we expect to be able to accelerate the quantum algorithms that will be practical in fault-tolerant quantum computers in the future.

Acknowledgements

We are grateful to Hiroki Hamaguchi and Nobuyuki Yoshioka for their valuable assistance in executing the existing EDPC algorithm code. This work is supported by PRESTO JST Grant No. JPMJPR1916, MEXT Q-LEAP Grant No. JPMXS0120319794 and JPMXS0118068682, JST Moonshot R&D Grant No. JPMJMS2061, and JST CREST Grant

A Extension to many-body lattice surgery

The proof and algorithm presented in the main text focused on 3D routing for two-body lattice surgery. Two-body lattice surgery is a crucial case because quantum circuits are typically expressed in the Clifford+T form, where S and H gates are executed via code deformation [8, 25], and CNOT and T gates are translated into two-body lattice surgeries [43]. On the other hand, all Clifford gates can be omitted by converting two-body lattice surgeries into many-body ones, provided there is no intermediate measurement [18]. It is therefore important to apply the key idea of this paper, decomposing a large lattice surgery into smaller fragments, to such cases. In this section, we show that the concept of this paper can be straightforwardly extended to many-body cases.

A.1 Conditions for 3D routing for many-body lattice surgery

In this section, we formulate a many-body variant of the 3D routing conditions, where routing is represented as a network of voxels in three dimensions connecting more than two data cells. There are four conditions in total. Since the latter conditions depend on properties derived from the earlier ones, we explain them sequentially.

Condition 1 (Tree-shaped): *The 3D routing must not contain any loop. Each voxel at the end of 3D routing (i.e., voxels with a single connected voxel) is called a leaf. Within the routing voxels, each leaf must be a unique voxel located at a target cell, and every target cell must contain one leaf.*

For the second condition, we introduce the concept of *fork*, a voxel connected to more than two voxels. Figure 18 illustrates examples of a fork in the 3D routing, where a fork voxel is aligned with a black solid line. Then, the second condition is stated as follows.

Condition 2 (No-temporal-fork): *No voxel is connected to the top or bottom of a fork voxel.* Thus, Fig. 18(a) is a fork satisfying this condition while Figs. 18(b) and 18(c) do not.

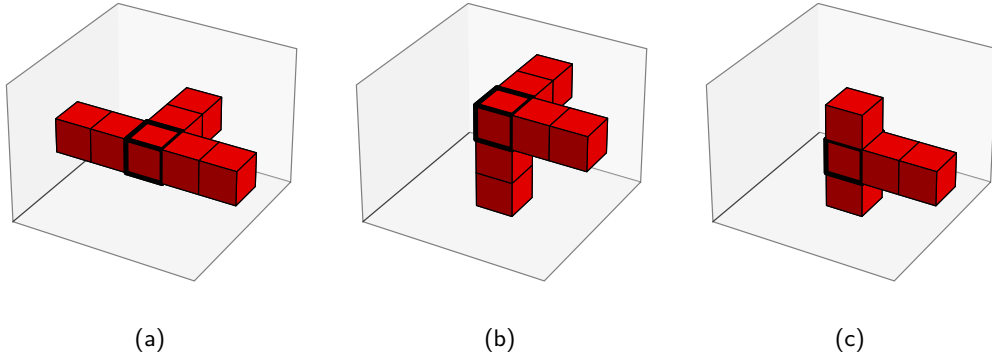


Figure 18: Examples of forks. (a) valid fork structure. (b,c) fork structures prohibited by Condition 2.

For the third condition, we need to introduce the concept of *even and odd segment*. A segment is a set of voxels connected within the same XY plane in the 3D routing. We classify all segments into even or odd as follows. First, we select a specific leaf as a pivot leaf, and the segment connected to it is designated as an even segment. If Condition 2 is satisfied, each ancillary cell connects exactly two segments. If segments are connected to an even (odd) segment without a kink via an ancillary cell, they are labeled as an even

(odd) segment. If segments are connected to an even (odd) segment with a kink via an ancillary cell, they are labeled as an odd (even) segment. Repeating this classification, all segments are labeled as either even or odd.

Then, the third and fourth conditions can be stated as follows.

Condition 3 (No-leaf-in-odd-segment): *No leaf is included in odd-segment.*

Condition 4 (No-fork-in-odd-segment): *No fork voxel is included in odd-segment.*

Note that one might worry that these conditions depend on the choice of the pivot leaf. However, if Condition 3 is satisfied for one choice of pivot, it remains satisfied for any other pivot choice. Therefore, the pivot leaf can be chosen arbitrarily.

Figure 19(a) shows an example of how a 3D routing that satisfies the above four conditions can be decomposed into even and odd segments. Here, diagonally hatched green voxels correspond to leaves (i.e., data cells), blue voxels are even segments, vertically hatched red voxels are odd segments, and white voxels correspond to the idling of ancillary cells. There are four even segments and three odd segments. Note that leaves are a part of even segments.

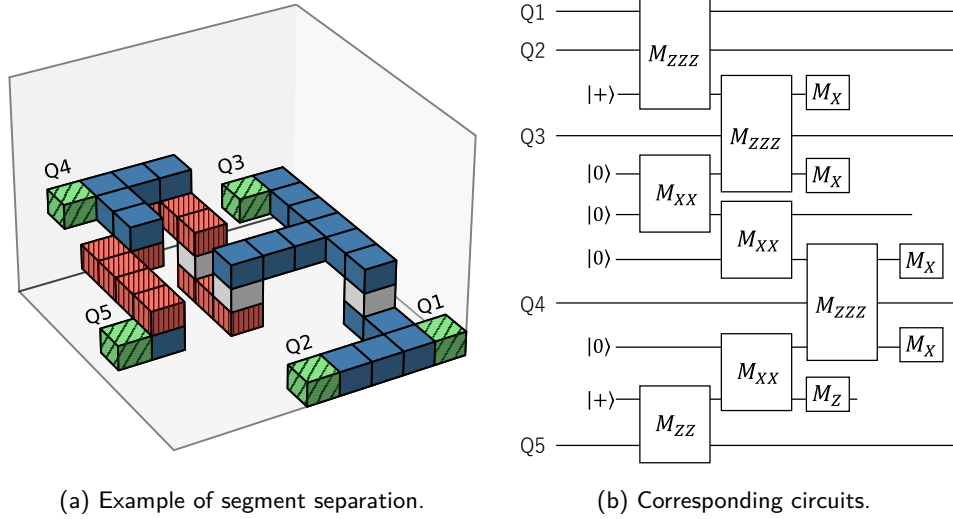


Figure 19: Example of segment separation and corresponding circuits.

Finally, we can state the following theorem, which provides a sufficient condition for converting a 3D routing into many-body lattice surgery.

Theorem 2. *Suppose there are m target data cells. A 3D routing satisfying the following conditions can be converted into m -body Pauli- $Z(X)$ measurements on m target cells.*

1. *The 3D routing satisfies the above four conditions.*
2. *The 3D routing connects $Z(X)$ -boundaries of each target data cell.*

Thus, if all the data cells (i.e., leaves) in Fig. 19(a) are connected via Z -boundaries, this 3D routing can be mapped to a five-body Pauli- Z measurement. The following subsections will show the proof of the above theorem.

Note that if we want to perform lattice surgery in which the Pauli operator contains both X and Z , we can perform it by adjusting the position of the surface-code lattice in advance. This requires the ability to shift or rotate the shape of surface codes and to perform stabilizer Pauli measurements that involve both X and Z [8, 18, 44], which would incur additional latency in scheduling.

A.2 Proof of theorem

This section provides the proof for many-body Pauli- Z measurements. The proof for Pauli- X measurement can be obtained by swapping the X and Z in the following explanation. First, we assign the direction of ancillary cells so that even segments are connected to a Z -boundary of them, and odd segments to an X -boundary. Given the four conditions, the 3D routing after this assignment satisfies the following properties:

- Every odd segment forms a path between the X -boundaries of two ancillary cells.
- Each end of an even-segment is connected to a Z -boundary of an ancillary cell or target cell.
- Every ancillary cell bridges exactly two segments.

We reduce an ancillary cell connected by two even segments into a single even segment, using the circuit equivalence shown in Fig. 20(a). Similarly, an ancillary cell connected by two odd segments is reduced to a single odd segment. This procedure is repeated until every ancillary cell bridges two segments of different parities.

Next, we reduce a sequence consisting of an even segment, an ancillary cell, an odd segment, another ancillary cell, and an even segment into a single even segment using the circuit equivalence shown in Figs. 20(b), 20(c), 20(d). This procedure is repeated until no ancillary cells or odd segments remain. Eventually, a single even segment connects all the leaves of the original tree, which corresponds to the target many-body Pauli measurement.

Figure 19(b) shows the corresponding quantum circuits to Fig. 19(a). By repeating the replacement according to Fig. 20, we can convert it to five-body Z measurements.

A.3 Extension of Dijkstra projection to many-body cases

We can also define a many-body variant of the Dijkstra projection for many-body lattice surgery on logical qubits $(Q_1, \dots, Q_m)$ by using the two-body Dijkstra projection as a subroutine. The procedure can be summarized as follows:

1. Perform the two-body Dijkstra projection for logical qubits Q_1 and Q_2 .
2. For each $i = 3, 4, \dots, m$, perform the two-body Dijkstra projection between logical qubit Q_i and the nearest Z -boundary of the existing even segment, and adjust the kink between them to be even.

This procedure ensures that the final routing is tree-shaped, has no temporal forks, and that no odd segment contains any fork or leaf. Other possible implementations of the Dijkstra projection for many-body cases may exist, but their exploration is left for future work.

B Compilation framework and architecture of FTQC

This section provides a brief outline of the expected compilation procedure and the architecture of FTQC machines. Note that the explanation below serves as an example of a compilation flow; our technique can be applied to any form of compilation and any FTQC machine, as long as the underlying architecture relies on lattice-surgery techniques.

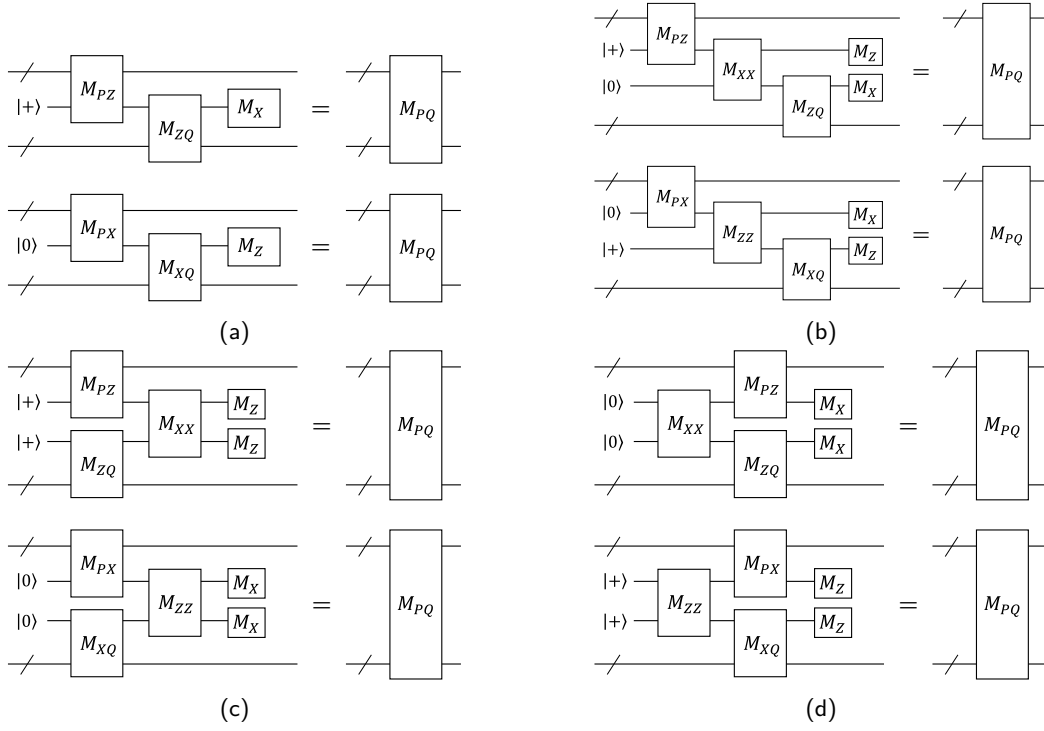


Figure 20: Equivalent processes of many-body measurements. Here, P and Q are an arbitrary many-body Pauli operator, and M_{AB} is a Pauli measurement with projectors $(I \pm A \otimes B)/2$ on two registers. (a) Two many-body measurements that act on an ancillary logical qubit with the same basis are equal to a single many-body measurement. (b,c,d) Three alternating many-body measurements are equal to a single many-body measurement. These are many-body variants of Fig. 7.

B.1 Quantum circuit representation

The first step is to synthesize a quantum circuit with a sequence of quantum gates defined in the form listed in Table 8. In this table, the data types **Q** and **R** represent qubit symbols and rotation angles, respectively. Although control instructions such as **BRANCH**, **RETURN**, and **CALL** can be included, most quantum algorithms can be written without them. We therefore restrict ourselves to a form that omits such instructions to simplify the overall implementation.

B.2 FTQC machine instructions

Instructions that can be directly executed on actual FTQC machines are limited, since they must be implemented in a fault-tolerant manner. While the details of the available gate set depend on the quantum hardware, the error-correcting code, and the control strategy, a representative instruction set is shown in Table 9. In this format, in addition to the qubit symbol **Q**, we use the magic-state-factory symbol **F**, the location symbol **L**, and the classical-bit symbol **C**. Qubits are allocated to specific cell locations using **ALLOC_QUBIT**. The allocated logical qubits can be initialized to $|0\rangle$ or $|+\rangle$ using the **INIT_*** instructions. A typical FTQC requires a dedicated region for magic-state generation, which is allocated using **ALLOC_FACTORY**.

The unitary instructions **UNIT_*** perform logical Hadamard and phase gates acting on target cells [8, 25]. These operations require a neighboring cell: the Hadamard gate requires any adjacent cell, while the phase gate requires one attached to the Z-boundary [9]. A previously stored measurement value in **C** can be used to condition these operations.

Table 8: Quantum circuit representation

Instruction	Operand	Description
X	Q	Pauli-X gate
Y	Q	Pauli-Y gate
Z	Q	Pauli-Z gate
H	Q	Hadamard gate
S	Q	Pauli-Z $\pi/4$ rotation
T	Q	Pauli-Z $\pi/8$ rotation
CX	Q1 Q1	CNOT gate
CCX	Q1 Q2 Q3	Toffoli gate
RX	Q R	Pauli-X rotation gate
RY	Q R	Pauli-Y rotation gate
RZ	Q R	Pauli-Z rotation gate

Table 9: Gate representation in quantum algorithms

Instruction	Operand	Description
ALLOC_QUBIT	L Q	Allocate qubits
ALLOC_FACTORY	L F	Allocate magic state factory
INIT_Z	Q	Initialize qubits to $ 0\rangle$
INIT_X	Q	Initialize qubits to $ +\rangle$
UNIT_H	Q L C	Hadamard gate with rotations
UNIT_S	Q L C	Phase gate with twist
MEAS_Z	Q C	Z-Pauli measurement
MEAS_X	Q C	X-Pauli measurement
MEAS_ZZ	Q1 Q2 list[L] C	ZZ-Pauli measurement
MEAS_XX	Q1 Q2 list[L] C	XX-Pauli measurement
MEAS_MAGIC	Q F list[L] C	ZZ-Pauli measurement on target and magic
XOR	C1 C2	XOR calculation

The subsequent measurement instructions (**MEAS_***) are used to perform logical Pauli measurements on the target qubits and to store the outcomes in **C**. A single-qubit measurement does not require additional space, whereas multi-qubit measurements temporarily occupy the path between relevant cells. This path is specified as a sequence of cell locations **list[L]**. The **MEAS_MAGIC** instruction is similar to **MEAS_ZZ** but acts on a target logical qubit and a magic state generated by the factory.

The final instruction, **XOR**, performs classical-bit arithmetic on measurement outcomes, typically to compute conditions from multiple measurement results.

It is also possible to omit the path information from the operand and instead determine paths between logical qubits at runtime. The drawback of this approach is that the path-finding process must be sufficiently fast, which can become challenging for large connectivity graphs or when the stabilizer-measurement cycle is short. On the other hand, this approach offers several advantages: (1) reduced object code size, (2) reusable functions independent of allocation status, and (3) dynamic path selection according to the runtime information, such as success timing of magic-state or entanglement generation. We believe the optimal design choice depends on the machine architecture, and we leave this aspect as future work.

B.3 Translation from circuit forms to instructions

This section describes how to translate a quantum circuit representation into FTQC instructions without location symbols. Quantum gates **H** and **S** are directly replaced with **UNIT_H** and **UNIT_S**, respectively. We decompose **CCX** into **(CX, T, UNIT_H)**, following known constructions [45]. Next, we decompose **CX** into **(INIT_X, MEAS_ZZ, MEAS_XX, MEAS_Z)** and conditional **(X, Z)** operations using the method in Ref. [8]. This decomposition requires an ancillary logical qubit, so we assign one ancillary qubit symbol for each data logical qubit. We then decompose the Pauli rotations **(RX, RY, RZ)** into **(UNIT_S, UNIT_H, T)** using the Ross–Selinger decomposition [46]. The **T** gate is further decomposed via magic-state consumption using **(MEAS_MAGIC, MEAS_X)** and conditional **(UNIT_S, X)** gates, following Ref. [8]. Finally, we remove Pauli unitary operations **X**, **Y**, and **Z** and replace them with **XOR** operations using the Pauli frame technique. After these steps, the quantum circuit is fully decomposed into **INIT_***, **MEAS_***, **UNIT_***, and **XOR** instructions without location operands.

Next, we perform placement and routing [11], which fills in the location operands (**L**) to make the executable compatible with a specific machine layout. For simplicity, we assume that all required logical qubits are allocated initially and never deallocated. In our benchmark, logical qubits are arranged on the pattern shown in Fig. 1. The allocation rule for each evaluation is detailed in the next section. Finally, we assign ancillary locations and paths for **UNIT_*** and **MEAS_*** instructions. While this process can be performed greedily, our method can be used to improve throughput.

B.4 Instance synthesis and simplification used in this paper

For the numerical experiments in the main text, we directly synthesized the **SELECT** circuits provided in Refs. [10, 11] according to the target Hamiltonian of the 2D Heisenberg model. The **SELECT** circuits consist only of **(S, H, CX, CCX)**, thus we do not need to mind the other gates, such as rotation gates. It is known that **CCX** in these circuits can be decomposed with four **T** gates if we know the target logical qubit is given as or returned to $|0\rangle$ [10, 47].

For numerical evaluation in Sec. 4.2, we performed further decompositions and removed all instructions irrelevant to lattice surgery. We also assigned the magic operand of **MEAS_MAGIC** to one of the magic state factories in a cyclic manner, thereby reducing **MEAS_MAGIC** to **MEAS_ZZ**. After this conversion, we obtained an instruction sequence consisting of **MEAS_XX** and **MEAS_ZZ**. Using this sequence, we assigned the logical qubits and magic state factories to the data cells from the top left to the bottom right. We assume that these data cells have identical boundary orientations. To rule out the influence of magic state factories and quantify the lattice-surgery routing cost, we assumed that magic state factories produce a magic state every code beat and that the number of factories equals the number of logical qubit operands.

Regarding the comparison in Sec. 4.3, we constructed an instruction sequence consisting of **CX** and **MEAS_MAGIC** for compatibility with EDPC. Also, we adopted the settings of the EDPC algorithm in Ref. [21], which are outlined below. Logical qubits are assigned from top left to bottom right. It is assumed that the data cells have identical boundary orientations. Magic state factories are placed at the position (i, j) such that i and j are even and satisfying any of the following: $i = 0$, $i = W - 1$, $j = 0$, or $j = W - 1$, where W is the width of the qubit-plane cells. See Fig. 10 of Ref. [21] for illustrations. It is also assumed that a magic state factory produces a single magic state every code beat.

References

- [1] Peter W. Shor. “Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer”. SIAM Review **41**, 303–332 (1999). doi: [10.1137/S0036144598347011](https://doi.org/10.1137/S0036144598347011).
- [2] A. Yu. Kitaev. “Quantum measurements and the abelian stabilizer problem” (1995). doi: [10.48550/arXiv.quant-ph/9511026](https://doi.org/10.48550/arXiv.quant-ph/9511026). [arXiv:quant-ph/9511026](https://arxiv.org/abs/quant-ph/9511026).
- [3] Seth Lloyd. “Universal quantum simulators”. Science **273**, 1073–1078 (1996). doi: [10.1126/science.273.5278.1073](https://doi.org/10.1126/science.273.5278.1073).
- [4] A. Yu. Kitaev. “Quantum computations: algorithms and error correction”. Russian Mathematical Surveys **52**, 1191–1249 (1997). doi: [10.1070/RM1997v052n06ABEH002155](https://doi.org/10.1070/RM1997v052n06ABEH002155).
- [5] Sergey B. Bravyi and Alexei Yu. Kitaev. “Quantum codes on a lattice with boundary” (1998). doi: [10.48550/arXiv.quant-ph/9811052](https://doi.org/10.48550/arXiv.quant-ph/9811052). [arXiv:quant-ph/9811052](https://arxiv.org/abs/quant-ph/9811052).
- [6] Austin G. Fowler, Matteo Mariantoni, John M. Martinis, and Andrew N. Cleland. “Surface codes: Towards practical large-scale quantum computation”. Physical Review A **86**, 032324 (2012). doi: [10.1103/PhysRevA.86.032324](https://doi.org/10.1103/PhysRevA.86.032324).
- [7] Clare Horsman, Austin G. Fowler, Simon Devitt, and Rodney Van Meter. “Surface code quantum computing by lattice surgery”. New Journal of Physics **14**, 123011 (2012). doi: [10.1088/1367-2630/14/12/123011](https://doi.org/10.1088/1367-2630/14/12/123011).
- [8] Austin G. Fowler and Craig Gidney. “Low overhead quantum computation using lattice surgery” (2018). doi: doi.org/10.48550/arXiv.1808.06709. [arXiv:1808.06709](https://arxiv.org/abs/1808.06709).
- [9] Michael E. Beverland, Prakash Murali, Matthias Troyer, Krysta M. Svore, Torsten Hoeffler, Vadym Kliuchnikov, Guang Hao Low, Mathias Soeken, Aarthi Sundaram, and Alexander Vaschillo. “Assessing requirements to scale to practical quantum advantage” (2022). doi: [10.48550/arXiv.2211.07629](https://doi.org/10.48550/arXiv.2211.07629). [arXiv:2211.07629](https://arxiv.org/abs/2211.07629).
- [10] Ryan Babbush, Craig Gidney, Dominic W. Berry, Nathan Wiebe, Jarrod McClean, Alexandru Paler, Austin Fowler, and Hartmut Neven. “Encoding electronic spectra in quantum circuits with linear t complexity”. Physical Review X **8**, 041015 (2018). doi: [10.1103/PhysRevX.8.041015](https://doi.org/10.1103/PhysRevX.8.041015).
- [11] Nobuyuki Yoshioka, Tsuyoshi Okubo, Yasunari Suzuki, Yuki Koizumi, and Wataru Mizukami. “Hunting for quantum-classical crossover in condensed matter problems”. npj Quantum Information **10**, 45 (2024). doi: [10.1038/s41534-024-00839-4](https://doi.org/10.1038/s41534-024-00839-4).
- [12] Ali JavadiAbhari, Shruti Patil, Daniel Kudrow, Jeff Heckey, Alexey Lvov, Frederic T. Chong, and Margaret Martonosi. “ScaffCC: A framework for compilation and analysis of quantum computing programs”. In Proceedings of the 11th ACM Conference on Computing Frontiers. CF ’14. Association for Computing Machinery (2014). doi: [10.1145/2597917.2597939](https://doi.org/10.1145/2597917.2597939).
- [13] Qiskit contributors. “Qiskit: An open-source framework for quantum computing”. url: doi.org/10.5281/zenodo.2573505.
- [14] Seyon Sivarajah, Silas Dilkes, Alexander Cowtan, Will Simmons, Alec Edgington, and Ross Duncan. “t|ket>: a retargetable compiler for NISQ devices”. Quantum Science and Technology **6**, 014003 (2020). doi: [10.1088/2058-9565/ab8e92](https://doi.org/10.1088/2058-9565/ab8e92).
- [15] George Watkins, Hoang Minh Nguyen, Keelan Watkins, Steven Pearce, Hoi-Kwan Lau, and Alexandru Paler. “A high performance compiler for very large scale surface code computations”. Quantum **8**, 1354 (2024). doi: [10.22331/q-2024-05-22-1354](https://doi.org/10.22331/q-2024-05-22-1354).
- [16] Daniel Herr, Franco Nori, and Simon J. Devitt. “Optimization of lattice surgery is np-hard”. Npj quantum information **3**, 35 (2017). doi: [10.1038/s41534-017-0035-1](https://doi.org/10.1038/s41534-017-0035-1).

- [17] Abtin Molavi, Amanda Xu, Swamit Tannu, and Aws Albarghouthi. “Compilation for surface code quantum computers” (2023). doi: [10.48550/arXiv.2311.18042](https://doi.org/10.48550/arXiv.2311.18042). [arXiv:2311.18042](https://arxiv.org/abs/2311.18042).
- [18] Daniel Litinski. “A game of surface codes: Large-scale quantum computing with lattice surgery”. *Quantum* **3**, 128 (2019). doi: [10.22331/q-2019-03-05-128](https://doi.org/10.22331/q-2019-03-05-128).
- [19] Daniel Litinski. “Magic state distillation: Not as costly as you think”. *Quantum* **3**, 205 (2019). doi: [10.22331/q-2019-12-02-205](https://doi.org/10.22331/q-2019-12-02-205).
- [20] Lingling Lao, Bas van Wee, Imran Ashraf, J. van Someren, Nader Khammassi, Koen Bertels, and Carmen G. Almudever. “Mapping of lattice surgery-based quantum circuits on surface code architectures”. *Quantum Science and Technology* **4**, 015005 (2018). doi: [10.1088/2058-9565/aadd1a](https://doi.org/10.1088/2058-9565/aadd1a).
- [21] Michael Beverland, Vadym Kliuchnikov, and Eddie Schoute. “Surface code compilation via edge-disjoint paths”. *PRX Quantum* **3**, 020342 (2022). doi: [10.1103/PRXQuantum.3.020342](https://doi.org/10.1103/PRXQuantum.3.020342).
- [22] Guang Hao Low and Isaac L. Chuang. “Hamiltonian simulation by qubitization”. *Quantum* **3**, 163 (2019). doi: [10.22331/q-2019-07-12-163](https://doi.org/10.22331/q-2019-07-12-163).
- [23] Austin G. Fowler, Adam C. Whiteside, and Lloyd C. L. Hollenberg. “Towards practical classical processing for the surface code”. *Physical review letters* **108**, 180501 (2012). doi: [10.1103/PhysRevLett.108.180501](https://doi.org/10.1103/PhysRevLett.108.180501).
- [24] Rajeev Acharya, Igor Aleiner, Richard Allen, Trond I. Andersen, Markus Ansmann, Frank Arute, Kunal Arya, Abraham Asfaw, Juan Atalaya, Ryan Babbush, et al. “Suppressing quantum errors by scaling a surface code logical qubit”. *Nature* **614**, 676–681 (2023). doi: [10.1038/s41586-022-05434-1](https://doi.org/10.1038/s41586-022-05434-1).
- [25] Benjamin J. Brown, Katharina Laubscher, Markus S. Kesselring, and James R. Wootton. “Poking holes and cutting corners to achieve clifford gates with the surface code”. *Physical Review X* **7**, 021029 (2017). doi: [10.1103/PhysRevX.7.021029](https://doi.org/10.1103/PhysRevX.7.021029).
- [26] Michael L. Fredman and Robert Endre Tarjan. “Fibonacci heaps and their uses in improved network optimization algorithms”. *J. ACM* **34**, 596–615 (1987). doi: [10.1145/28869.28874](https://doi.org/10.1145/28869.28874).
- [27] Allyson Silva, Xiangyi Zhang, Zak Webb, Mia Kramer, Chan-Woo Yang, Xiao Liu, Jessica Lemieux, Ka-Wai Chen, Artur Scherer, and Pooya Ronagh. “Multi-qubit lattice surgery scheduling”. In *19th Conference on the Theory of Quantum Computation, Communication and Cryptography (TQC 2024)*. Pages 1–1. Schloss Dagstuhl–Leibniz-Zentrum für Informatik (2024). url: doi.org/10.4230/lipics.tqc.2024.1.
- [28] Austin G. Fowler. “Time-optimal quantum computation” (2012). doi: [10.48550/arXiv.1210.4626](https://doi.org/10.48550/arXiv.1210.4626). [arXiv:1210.4626](https://arxiv.org/abs/1210.4626).
- [29] Mingzheng Zhu, Hao Fu, Jun Wu, Chi Zhang, Wei Xie, and Xiang-Yang Li. “Ecmass: Efficient circuit mapping and scheduling for surface code”. In *2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. Page 158–169. IEEE (2024). doi: [10.1109/cgo57630.2024.10444874](https://doi.org/10.1109/cgo57630.2024.10444874).
- [30] Mingzheng Zhu, Hao Fu, Haishan Song, Jun Wu, Chi Zhang, Wei Xie, and Xiangyang Li. “Ecmass+: Efficient circuit mapping and scheduling for surface code encoded circuit on quantum cloud platform”. *ACM Transactions on Architecture and Code Optimization* **22**, 1–25 (2025). doi: [10.1145/3760783](https://doi.org/10.1145/3760783).
- [31] Google Quantum AI. “Suppressing quantum errors by scaling a surface code logical qubit”. *Nature* **614**, 676–681 (2023). doi: [10.1038/s41586-022-05434-1](https://doi.org/10.1038/s41586-022-05434-1).
- [32] Google Quantum AI and Collaborators. “Quantum error correction below the surface code threshold”. *Nature* **638**, 920–926 (2024). doi: [10.1038/s41586-024-08449-y](https://doi.org/10.1038/s41586-024-08449-y).

- [33] Hengyun Zhou, Chen Zhao, Madelyn Cain, Dolev Bluvstein, Nishad Maskara, Casey Duckering, Hong-Ye Hu, Sheng-Tao Wang, Aleksander Kubica, and Mikhail D. Lukin. “Low-overhead transversal fault tolerance for universal quantum computation”. *Nature* **646**, 303–308 (2025). doi: [10.1038/s41586-025-09543-5](https://doi.org/10.1038/s41586-025-09543-5).
- [34] Craig Gidney, Michael Newman, Peter Brooks, and Cody Jones. “Yoked surface codes”. *Nature Communications* **16** (2025). doi: [10.1038/s41467-025-59714-1](https://doi.org/10.1038/s41467-025-59714-1).
- [35] Theodore J. Yoder, Eddie Schoute, Patrick Rall, Emily Pritchett, Jay M. Gambetta, Andrew W. Cross, Malcolm Carroll, and Michael E. Beverland. “Tour de gross: A modular quantum computer based on bivariate bicycle codes” (2025). doi: [10.48550/ARXIV.2506.03094](https://doi.org/10.48550/ARXIV.2506.03094).
- [36] Qian Xu, J. Pablo Bonilla Ataides, Christopher A. Pattison, Nithin Raveendran, Dolev Bluvstein, Jonathan Wurtz, Bane Vasić, Mikhail D. Lukin, Liang Jiang, and Hengyun Zhou. “Constant-overhead fault-tolerant quantum computation with reconfigurable atom arrays”. *Nature Physics* **20**, 1084–1090 (2024). doi: [10.1038/s41567-024-02479-z](https://doi.org/10.1038/s41567-024-02479-z).
- [37] Lawrence Z. Cohen, Isaac H. Kim, Stephen D. Bartlett, and Benjamin J. Brown. “Low-overhead fault-tolerant quantum computing using long-range connectivity”. *Science Advances* **8** (2022). doi: [10.1126/sciadv.abn1717](https://doi.org/10.1126/sciadv.abn1717).
- [38] Naomi H Nickerson, Ying Li, and Simon C Benjamin. “Topological quantum computing with a very noisy network and local error rates approaching one percent”. *Nature communications* **4**, 1756 (2013). url: doi.org/10.1038/ncomms2773.
- [39] Hector Bombin, Isaac H Kim, Daniel Litinski, Naomi Nickerson, Mihir Pant, Fernando Pastawski, Sam Roberts, and Terry Rudolph. “Interleaving: Modular architectures for fault-tolerant photonic quantum computing” (2021). url: doi.org/10.48550/arXiv.2103.08612.
- [40] J Eli Bourassa, Rafael N Alexander, Michael Vasmer, Ashlesha Patil, Ilan Tzitrin, Takaya Matsuura, Daiqin Su, Ben Q Baragiola, Saikat Guha, Guillaume Dauphinais, et al. “Blueprint for a scalable photonic fault-tolerant quantum computer”. *Quantum* **5**, 392 (2021). url: doi.org/10.22331/q-2021-02-04-392.
- [41] Christopher Pattison, Gefen Baranes, Juan Pablo Bonilla Ataides, Mikhail D Lukin, and Hengyun Zhou. “Constant-rate entanglement distillation for fast quantum interconnects”. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture*. Pages 257–270. (2025). url: doi.org/10.1145/3695053.373106.
- [42] Yuya Maeda, Yasunari Suzuki, Toshiki Kobayashi, Takashi Yamamoto, Yuuki Tokunaga, and Keisuke Fujii. “Logical entanglement distribution between distant 2d array qubits” (2025). url: doi.org/10.48550/arXiv.2503.14894.
- [43] Christopher Chamberland and Earl T Campbell. “Universal quantum computing with twist-free and temporally encoded lattice surgery”. *PRX Quantum* **3**, 010331 (2022). url: doi.org/10.1103/PRXQuantum.3.010331.
- [44] Matt McEwen, Dave Bacon, and Craig Gidney. “Relaxing hardware requirements for surface code circuits using time-dynamics”. *Quantum* **7**, 1172 (2023). url: doi.org/10.22331/q-2023-11-07-1172.
- [45] Michael A Nielsen and Isaac L Chuang. “Quantum computation and quantum information”. Cambridge university press. (2010). url: doi.org/10.1017/CBO9780511976667.
- [46] Neil J Ross and Peter Selinger. “Optimal ancilla-free clifford+ t approximation of z-rotations” (2014). url: doi.org/10.48550/arXiv.1403.2975.
- [47] David P DiVincenzo. “Quantum gates and circuits”. *Proceedings of the Royal Society of London. Series A: Mathematical, Physical and Engineering Sciences* **454**, 261–276 (1998). url: doi.org/10.1098/rspa.1998.0159.