

# Flag-Proxy Networks: Overcoming the Architectural, Scheduling and Decoding Obstacles of Quantum LDPC Codes

Suhas Vittal  
Georgia Institute of Technology  
Atlanta, GA, United States  
suhaskvittal@gatech.edu

Ali Javadi-Abhari  
IBM T.J. Watson Research Center  
Yorktown Heights, NY, United States  
ali.javadi@ibm.com

Andrew W. Cross  
IBM T.J. Watson Research Center  
Yorktown Heights, NY, United States  
awcross@us.ibm.com

Lev S. Bishop  
IBM T.J. Watson Research Center  
Yorktown Heights, NY, United States  
lsbishop@us.ibm.com

Moinuddin Qureshi  
Georgia Institute of Technology  
Atlanta, GA, United States  
moin@gatech.edu

**Abstract**—Quantum error correction is necessary for achieving exponential speedups on important applications. The planar surface code has remained the most studied error-correcting code for the last two decades because of its relative simplicity. However, encoding a singular logical qubit with the planar surface code requires physical qubits quadratic in the code distance ( $d$ ), making it space-inefficient for the large-distance codes necessary for promising applications. Thus, *Quantum Low-Density Parity-Check (QLDPC)* have emerged as an alternative to the planar surface code but require a higher degree of connectivity. Furthermore, the problems of fault-tolerant syndrome extraction and decoding are understudied for these codes and also remain obstacles to their usage.

In this paper, we consider two under-studied families of QLDPC codes: hyperbolic surface codes and hyperbolic color codes. We tackle the three challenges mentioned above as follows. *First*, we propose *Flag-Proxy Networks (FPNs)*, a generalizable architecture for quantum codes that achieves low connectivity through flag and proxy qubits. *Second*, we propose a *greedy syndrome extraction scheduling* algorithm for general quantum codes and further use this algorithm for fault-tolerant syndrome extraction on FPNs. *Third*, we present two decoders that leverage flag measurements to decode the hyperbolic codes accurately. Our work finds that degree-4 FPNs of the hyperbolic surface and color codes are respectively  $2.9\times$  and  $5.5\times$  more space-efficient than the  $d = 5$  planar surface code, and become even more space-efficient when considering higher distances. The hyperbolic codes also have error rates comparable to their planar counterparts.

**Index Terms**—Quantum Error Correction, Quantum Error Decoding, Syndrome Extraction

## I. INTRODUCTION

Quantum error correction is the most promising path towards realizing exponential speedups for applications in quantum chemistry and cryptanalysis [10, 14, 24, 29, 40, 42]. Error-corrected quantum computers leverage quantum error correction by encoding multiple noisy *physical* qubits into a *logical block* containing fewer error-resilient logical qubits. The logical qubits encoded by quantum error correcting codes have better fidelity than their constituent physical qubits, provided the physical error rate is low enough (e.g., 0.1%).

In this paper, we focus on superconducting quantum computers. Quantum error-correcting codes implemented on su-

perconducting quantum computers arrange physical qubits into *data qubits*, which maintain the logical state, and *parity qubits*, which detect  $X$  and  $Z$  errors. Leveraging these parity qubits requires executing a quantum *syndrome extraction circuit*, which entangles the parity qubits with neighboring data qubits. Subsequently, these parity qubits are measured, yielding a bitstring known as a *syndrome*. The syndrome is sent to a *decoder*, which identifies errors on the data qubits. However, since syndromes are unreliable as syndrome extraction itself is erroneous, the decoder must analyze  $d$  consecutive rounds of syndromes to identify errors accurately. Ideally, a distance  $d$  quantum error-correcting code should correct up to  $(d-1)/2$  errors within  $d$  syndrome extraction rounds; increasing  $d$  exponentially suppresses error. However, circuit errors can harm the *effective distance* of the code, preventing  $(d-1)/2$  errors from being corrected. A syndrome extraction circuit is *fault-tolerant* if the effective distance is sufficiently high such that errors remain exponentially suppressed with increasing  $d$ .

The *planar surface code* has remained the focus of error correction research for the last two decades. We attribute this dominance to three reasons:

- 1) The planar surface code requires grid connectivity, which is easy to fabricate using superconducting qubits. For this reason, the planar surface code has been realized on multiple quantum processors [1, 2, 30]. In contrast, more densely connected processors are hard to fabricate due to frequency crowding and crosstalk [35, 39].
- 2) Fault-tolerant syndrome extraction for the planar surface code is enabled by diligent CNOT scheduling [44]. In contrast, fault-tolerant syndrome extraction with other quantum codes requires additional physical overheads beyond data and parity qubits [11]–[13].
- 3) The planar surface code can be decoded with the *Minimum Weight Perfect Matching (MWPM)* algorithm, whose implementations are fast and readily available [25, 27].

However, the planar surface code is space-inefficient as any amount of redundancy encodes a singular logical qubit. For

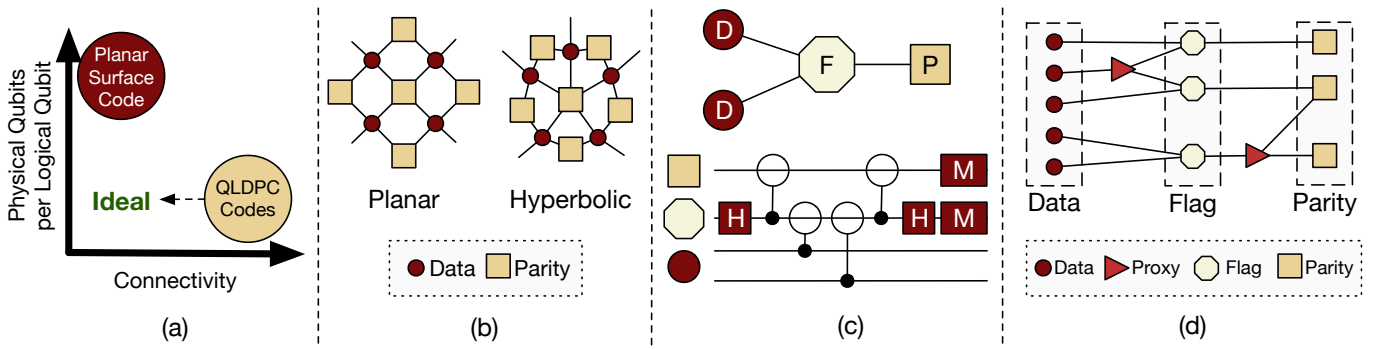


Fig. 1. (a) Tradeoff between efficiency and connectivity amongst quantum error correcting codes. (b) Local structures for the planar and hyperbolic surface codes. (c) Syndrome extraction with a flag qubit is measured to detect errors on the data qubits. (d) The proposed Flag-Proxy Network architecture.

any promising applications, implementing a single logical qubit at a near-term error rate of  $p = 10^{-3}$  consumes at least 1000 physical qubits [37]. Since many promising applications require 1000s of high-fidelity program qubits [21], error-corrected quantum computers exclusively using the planar surface code will require millions of physical qubits to provide quantum advantage. These overheads are undesirable given the engineering challenges of building very large quantum computers. Ideally, we desire alternative quantum codes with better efficiency and comparable error rates.

**Moving Away from the Planar Surface Code:** *Quantum Low-Density Parity Check (QLDPC)* codes have emerged as alternatives to the planar surface code. Unlike planar surface code, QLDPC codes are space-efficient, but this efficiency comes at the price of denser connectivity beyond the degree-4 connectivity required for the planar surface code [7, 9, 16, 26, 43]. Furthermore, fault-tolerant syndrome extraction and decoding are relatively unexplored, especially under realistic noise models. Figure 1(a) presents the two extremes of current error correction proposals. Planar surface codes have been demonstrated as they have simple connectivity requirements [1, 30]. However, it is hard to scale because of its poor rate. Contrarily, QLDPC codes are efficient but demand connectivity beyond degree-4. In this paper, we tackle the problems of *constructing architectures for QLDPC codes*, *fault-tolerant syndrome extraction*, and *decoding*. We focus on two understudied QLDPC code families: *hyperbolic surface codes* [8, 9], and *hyperbolic color codes* [16].

**Reducing Connectivity Requirements:** Figure 1(b) compares the connectivity demands of the planar surface code to that of a hyperbolic surface code. Locally, the planar surface code requires degree-4 connectivity, as each parity qubit is connected to four data qubits and vice versa. In contrast, the hyperbolic surface code requires degree-5 connectivity, where parity qubits are connected to five data qubits. This naïve architecture for the hyperbolic surface code would have two issues. The first issue is clear: the degree-5 connectivity would be hard to fabricate. The less obvious problem is that even if such an architecture could be fabricated, it might not support fault-tolerant syndrome extraction, potentially yielding poor

error rates in practice. Our goal is to construct sparse architectures that also support fault-tolerant syndrome extraction.

Prior work has proposed using *flag* qubits to reduce connectivity requirements [3, 11]–[13, 33, 36]. A flag qubit, shown in Figure 1(c), has two purposes during syndrome extraction. *First*, the flag qubit reduces connectivity: the circuit shown in Figure 1(c) entangles two data qubits with a parity qubit by using a flag qubit as an intermediary. *Second*, measuring the flag qubit can detect errors that harm the effective distance. These flag measurements form a *flag syndrome* that a decoder must correctly leverage to correct errors accurately. Consequently, *overusing* flag qubits will overwhelm the decoder with unnecessary information and increase decoding complexity. Ideally, an architecture should use as few flag qubits as possible while supporting fault-tolerant syndrome extraction.

To this end, we propose *Flag-Proxy Networks (FPNs)*. FPNs, whose high-level layout is shown in Figure 1(d), primarily use flag qubits to protect data qubits from errors that harm the effective distance while reducing connectivity demands. To avoid overusing flag qubits, further reductions in connectivity can be achieved by introducing “proxy qubits,” which need not be measured. To reduce the physical overheads of FPNs, we propose *flag sharing*, which merges flag qubits common to the same data qubits.

**Syndrome Extraction Scheduling:** While FPNs provide an architecture to realize an error-correcting code, we must execute a syndrome extraction circuit to retrieve a syndrome and detect errors. Constructing valid and low-depth syndrome extraction schedules is an NP-hard problem [4, 15, 19]. For codes with *translation invariance*, such as the planar surface code [4], a valid schedule for a single check can be reused for other checks in the code, thus significantly simplifying the problem. However, QLDPC codes do not necessarily have translation invariance. We present a *greedy scheduling algorithm* for syndrome extraction scheduling for such codes. Our greedy algorithm uses a solver to schedule checks in isolation and imposes constraints on each check’s schedule given already-scheduled checks.

**Decoding with Flag Qubits:** As FPNs leverage flag qubits to detect errors during syndrome extraction, a decoder must

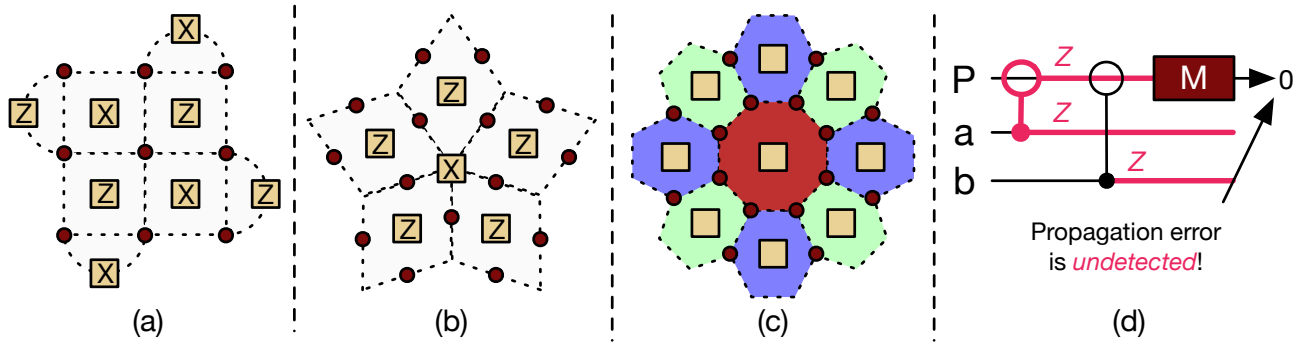


Fig. 2. (a) Example of planar surface code. (b) Example of the local structure of a  $\{4, 5\}$  hyperbolic surface code. Each edge corresponds to a data qubit, each face corresponds to an  $X$  check, and each vertex corresponds to a  $Z$  check. (c) Example of a  $\{4, 6\}$  hyperbolic color code. Each vertex corresponds to a data qubit, and each face (plaquette) corresponds to both a  $Z$  and  $X$  check. (d) Example of an undetected propagation error caused by a CNOT error.

leverage the additional *flag syndrome* bits to identify errors correctly. In our analysis of error patterns on the hyperbolic codes, we find that errors that flip the *same syndrome bits* but *different flag bits* may correspond to entirely different data qubit errors. To handle this problem, we propose assigning flagged and unflagged errors into *equivalence classes*. These equivalence classes contain error events that (1) flip the same syndrome bits but (2) flip different flag bits. During decoding, only one error event is considered from each equivalence class, given the flag syndrome. Finally, to evaluate our flag protocol, we propose two decoders that leverage this protocol to decode both flavors of hyperbolic codes accurately.

In summary, our paper makes the following contributions:

- 1) We propose *Flag-Proxy Networks*, a generalized architecture for quantum error correcting codes. Flag-Proxy Networks reduce connectivity demands by using flag and proxy qubits while enabling fault-tolerant syndrome extraction.
- 2) We propose a *greedy scheduling algorithm* applicable to any quantum code.
- 3) We propose a flag syndrome protocol that organizes errors into equivalence classes. We further propose two decoders that leverage this protocol.

Our evaluations demonstrate that FPNs of the hyperbolic surface and color codes are, respectively,  $2.9\times$  and  $5.5\times$  more efficient than the  $d = 5$  planar surface code (49 physical qubits per logical qubit) while having comparable connectivity and error rates. This benefit only increases with larger distances.

## II. BACKGROUND

### A. Characterizing Quantum Codes

Quantum error-correcting codes are often characterized by three properties: (1) the number of data qubits in a logical block, (2) the number of logical qubits in a logical block, and (3) the code distance. The *parameters* of a code are written as  $[[n, k, d_X, d_Z]]$ , where  $n$  is the number of data qubits,  $k$  is the number of logical qubits,  $d_X$  is the code distance for  $X$  errors, and  $d_Z$  is the code distance for  $Z$  errors. If  $d_X = d_Z = d$ , then the code may be written as  $[[n, k, d]]$ . Furthermore,  $n - k$  checks must be measured to detect errors.

Two other characteristics of quantum codes cannot be characterized entirely by  $n$ ,  $k$ , and  $d$ . The first such characteristic is *check weight*, which is the number of data qubits involved in the check. If a check has weight  $\delta$ , then the check's corresponding parity qubit must be connected to  $\delta$  data qubits. The second characteristic is the total number of physical qubits  $N$  required to implement a *logical block*, which depends on the underlying hardware and code. Standard implementations of the planar surface code on superconducting qubits require  $N = 2n - 1$  physical qubits. As  $N \neq n$ , we use two metrics to quantify a code's overheads: the *ideal rate*, defined as  $R_{\text{ideal}} = k/n$ , and the *effective rate*, defined as  $R_{\text{eff}} = k/N$ .

### B. Planar Surface Codes

First, we review the basics of the planar surface code. Each data qubit on the planar surface code is protected by at most two  $Z$  checks and two  $X$  checks, which correct  $X$  and  $Z$  errors and are sufficient to correct an arbitrary error on the data qubit. The most common implementation of the planar surface code is the *rotated surface code* [28, 44], which is shown in Figure 2(a). This implementation of the planar surface code has parameters  $[[d^2, 1, d]]$  and requires degree-4 connectivity. Secondly, fault-tolerant syndrome extraction for the rotated surface code can be implemented through proper CNOT ordering [44]. Finally, the rotated surface code can be decoded through *Minimum-Weight Perfect Matching (MWPM)* decoding, which is fast and readily available. For these reasons, the planar surface code remains the predominant error-correcting code for superconducting systems.

### C. A Brief Primer on QLDPC Codes

The pitfall of the planar surface code is that  $R_{\text{ideal}} = 1/d^2$ , so error-corrected quantum computers using the planar surface code will require millions of physical qubits to support most applications [21, 37, 40]. Given the significant engineering hurdles necessary to support millions of qubits, the planar surface code is not an ideal candidate for error correction at scale.

Instead of encoding a single logical qubit with a code, we want to encode multiple logical qubits. To do so, we must

use codes with far fewer parity checks than data qubits: if a code has  $n$  data qubits and  $x$  parity checks, it will encode  $k = n - x$  logical qubits. Figure 3(a) shows the setup of the surface code, where parity checks detect errors on four data qubits. If checks were more complex and could detect errors across more data qubits, as in Figure 3(b), then we could use fewer parity checks. This is the fundamental idea behind *Quantum Low-Density Parity Check (QLDPC)* codes, which use far fewer checks than the surface code to detect errors on data qubits, thus yielding more logical qubits. Unfortunately, this comes at the cost of denser connectivity: *denser* codes generally encode *more* logical qubits.

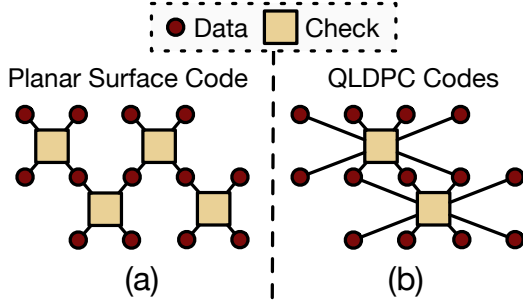


Fig. 3. Parity checks for (a) the planar surface code, and (b) QLDPC codes. For illustration purposes only.

#### D. Code Families for QLDPC Codes

We note two challenges with realizing QLDPC codes on superconducting quantum computers. *First*, QLDPC codes have high check-weights due to encoding multiple logical qubits. *Second*, most QLDPC codes are understudied under realistic noise models; thus, our understanding of fault-tolerant syndrome extraction and decoding is limited for most codes. This section discusses these problems for two examples of QLDPC codes: *hyperbolic surface codes* and *hyperbolic color codes*. In this paper, we use “hyperbolic codes” to collectively refer to hyperbolic surface and color codes.

1) *Hyperbolic Surface Codes*: Hyperbolic surface codes are created from geometric tilings parameterized by two integers  $r$  and  $s$  with the relationship in Equation (1) [6, 8, 9]. These  $\{r, s\}$  tilings have the property that  $s$   $r$ -gons meet at a point. Each face corresponds to a  $Z$  check, each point corresponds to an  $X$  check, and each edge between two faces corresponds to a data qubit. Figure 2(b) shows an example of a  $\{4, 5\}$  hyperbolic surface code.

$$\frac{1}{r} + \frac{1}{s} < \frac{1}{2} \quad (1)$$

2) *Hyperbolic Color Codes*: Hyperbolic color codes are constructed by selecting  $r$  and  $s$  with the same relationship as in Equation (1) with the additional constraint that  $s$  is even [16]. However, unlike hyperbolic surface codes, hyperbolic color codes are constructed by creating a tiling with three types of *plaquettes*: *red* plaquettes with  $2r$  vertices and *green* and *blue* plaquettes with  $s$  vertices. Each vertex in the tiling

corresponds to a data qubit and is incident to one plaquette of each color, and each plaquette corresponds to an  $X$  and  $Z$  check. Figure 2(c) shows an example of a  $\{4, 6\}$  hyperbolic color code.

3) *Rate of Hyperbolic Codes*: Each pair  $\{r, s\}$  corresponds to a different *subfamily* of hyperbolic codes, and each subfamily has a minimum ideal rate given by Equation (2). Consequently, larger hyperbolic codes encode more logical qubits. In contrast, the planar surface code always encodes a single logical qubit.

$$R_{\text{ideal}} \geq 1 - \frac{2}{r} - \frac{2}{s} \quad (2)$$

#### E. The Challenge of Connectivity

We note the following challenges with hyperbolic codes that are solved problems for the planar surface code:

- 1) Hyperbolic surface codes require degree- $r$  and degree- $s$  connectivity for  $Z$  and  $X$  checks, respectively, whereas hyperbolic color codes require degree- $2r$  and degree- $s$  connectivity for each plaquette. In contrast, the planar surface code requires degree-4 connectivity.
- 2) Hyperbolic codes have mostly not been studied under circuit-level noise beyond small examples [15]. Thus, fault-tolerant syndrome extraction and decoding remain open research areas for most of the hyperbolic codes. In contrast, all three areas are well-studied for the planar surface code [25, 27, 44].

Given these challenges, we believe that the hyperbolic codes are representative of some obstacles blocking the realization of QLDPC codes on superconducting systems. In this paper, we consider codes with  $n \leq 3000$  for four subfamilies of hyperbolic surface and color codes. Tables IV and V in the Appendix list the hyperbolic codes considered in this paper.

#### F. The Challenge of Syndrome Extraction

In principle, a distance  $d$  quantum code can correct  $\lfloor (d-1)/2 \rfloor$  errors. However, operation errors in a syndrome extraction circuit can limit the effective distance ( $d_{\text{eff}}$ ) of a quantum code, limiting the correction capability of the code to only  $\lfloor (d_{\text{eff}}-1)/2 \rfloor$  errors. Ideally, we want  $d_{\text{eff}} = d$ : such a syndrome extraction circuit is considered *fault-tolerant*.

Consider a syndrome extraction circuit where a parity qubit interacts directly with each data qubit, as in Figure 2(c) for a  $Z$  parity qubit  $P$  interacting with two data qubits,  $a$  and  $b$ . First,  $CNOT(a, P)$  fails and causes a  $Z$  error on both  $a$  and  $P$ . While the  $Z$  error on  $P$  does not affect the parity outcome during measurement, it will propagate to  $b$  through  $CNOT(b, P)$ ; note this occurs regardless of whether the  $CNOT$  fails. By the end of the syndrome extraction round, both  $a$  and  $b$  have  $Z$  errors. Note that this *two-qubit data error* has occurred from a *single fault* in the syndrome extraction circuit. We call such errors *propagation errors*.

**Impact of propagation errors:** We explain how propagation errors impact  $d_{\text{eff}}$  at a high level. First, consider a  $d = 3$  code, which is only guaranteed to protect against one data error. A

$d = 3$  code cannot handle a propagation error because a single propagation error affects multiple qubits. However, the error stemmed from a single CNOT error; hence, as the  $d = 3$  code cannot handle a single operation error, so  $d_{\text{eff}} = 2$ .

**Fault-Tolerance in the Planar Surface Code:** Note that syndrome extraction in standard implementations of the planar surface code can tolerate propagation errors by reordering CNOTs [44]. Such a property is due to the structure of the planar surface code. While other codes can achieve fault-tolerance by reordering CNOTs [13, 15, 38], we explore leveraging flag qubits as a general strategy for fault-tolerance.

### G. The Challenge of Decoding

As a decoder must correct errors encountered during program execution, its performance is closely tied to the syndrome extraction circuitry. If the syndrome extraction circuit is not fault-tolerant, the decoder cannot correct more than  $(d_{\text{eff}} - 1)/2$  errors. Similarly, an ineffective decoder may harm the effective distance even if the syndrome extraction circuit is fault-tolerant. For most QLDPC codes, this inter-relatedness between syndrome extraction and decoding has not been considered, as very little research considers the impact of circuit-level noise.

### H. Goal

QLDPC codes promise efficient error-corrected quantum computers but have challenges in (a) dense connectivity, (b) fault-tolerant syndrome extraction, and (c) accurate decoding. In this paper, we develop a general, low-connectivity architecture for QLDPC codes while tackling the interrelated problems of fault-tolerant syndrome extraction and decoding.

## III. EVALUATION METHODOLOGY

### A. Error Model

This paper considers a circuit-level error model, which best reflects errors found in real systems. Our error model contains the following errors for a physical error rate  $p$ .

- 1) Decoherence and dephasing errors at the beginning of a syndrome extraction round. We assign each qubit a  $T_1 = (1/p)\mu\text{s}$  and  $T_2 = 0.5T_1$  to model decoherence and idling error. Then, given a syndrome extraction latency  $t$ ,  $X$ ,  $Y$ , and  $Z$  errors are injected with probability  $p_X$ ,  $p_Y$ , and  $p_Z$  according to the Pauli twirling approximation as described in Equations (3) and (4) [44]. Specific operation latencies are listed below.

$$p_X = p_Y = \frac{1 - e^{-t/T_1}}{4} \quad (3)$$

$$p_Z = \frac{1 - 2e^{-t/T_2} + e^{-t/T_1}}{4} \quad (4)$$

- 2) Single-qubit gates cause random depolarizing errors at a rate of  $0.1p$  and have a latency of 30ns.
- 3) Two-qubit gates cause random two-qubit depolarizing errors at a rate of  $p$  and have a latency of 40ns.

- 4) Measurements return incorrect outcomes at a rate of  $p$  and have a latency of 800ns.
- 5) Resets fail at a rate of  $0.1p$  and have a latency of 30ns.
- 6) Idling errors occur during each two-qubit gate on qubits unused during the gate at a rate of  $0.1p$ .

Unlike prior work, which fixes decoherence and dephasing errors to occur with probability  $p$ , using  $T_1$  and  $T_2$  times to model decoherence and dephasing errors penalizes longer syndrome extraction latencies. For instance,  $2\times$  higher syndrome extraction latency results in  $2\times$  higher  $T_1$  and  $T_2$  errors. Longer circuits also incur more idling errors. Finally, we perform our simulations using Google’s *Stim* simulator [20].

### B. Evaluating Architectural Overheads

We consider effective rate  $R_{\text{eff}} = k/N$ , where  $k$  is the number of logical qubits in a logical block, and  $N$  is the total number of physical qubits required to realize the block.

### C. Evaluating Block Error Rate

In this paper, we execute memory experiments to evaluate a code’s *block error rate* (*BER*). A single memory experiment tests the code’s capability to preserve an initial state (either  $|00\dots 0\rangle$  or  $|++\dots+\rangle$ ) in the presence of errors over  $d$  syndrome extraction rounds. Once the  $d$  rounds are finished, the resulting syndrome is given to a decoder, which tries to correct any logical qubits. If any logical qubits have an error after decoding, an error has occurred.

Thousands of memory experiment trials are executed to estimate the *BER*, defined in Equation (5). We use the *normalized block error rate*  $BER_{\text{norm}} = BER/k$  to compare codes of different block sizes.

$$BER = \frac{\text{number of errors on any logical qubit}}{\text{number of fault-injection trials}} \quad (5)$$

## IV. FLAG-PROXY NETWORKS

This section presents *Flag-Proxy Networks*, a microarchitectural paradigm that makes QLDPC codes amenable to superconducting architectures.

### A. Flag Qubits

Reducing connectivity requires introducing additional qubits to enable interactions between non-adjacent qubits. One such approach involves using *flag qubits*<sup>1</sup> [13, 33, 46, 47]. Flag qubits not only reduce connectivity, but with correct use, they can detect propagation errors. Figure 4 presents the example of a flag qubit  $F$  used to entangle data qubits  $a$  and  $b$  with a parity qubit  $P$ . Here, a propagation error on  $a$  and  $b$  affects the phase of  $F$ . At the end of the syndrome extraction circuit,  $F$  is measured, and this measurement yields one, indicating a propagation error has occurred.

<sup>1</sup>Flag qubits are also known as “(flag-)bridge” qubits in prior work [33, 46, 47] to emphasize that flag qubits reduce connectivity demands.

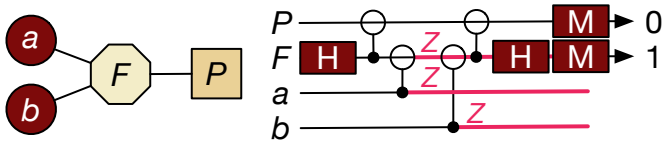


Fig. 4. Left: local connectivity required by a flag qubit. Right: corresponding syndrome extraction with a flag qubit  $F$ , which detects a propagation error.

**Fault-Tolerance with Flag Qubits:** While flag qubits can detect propagation errors, simply introducing flag qubits is insufficient for fault-tolerance. Indeed, the flag measurements form a secondary syndrome known as the *flag syndrome*. The decoder must leverage this flag syndrome to account for propagation errors during syndrome extraction. Furthermore, flag measurement errors cannot be detected, unlike parity measurements, as a propagation error will not repeat between rounds. Thus, an accurate decoder must account for flag measurement errors during decoding. We observe that while prior work has used flag qubits to reduce connectivity demands, they have not considered how to leverage flag qubits more generally during decoding [33, 46, 47].

**Flag Overuse:** As a decoder must use flag qubits, *overusing* flag qubits where they are unnecessary may overburden the decoder with useless information. An example of such a situation is shown in Figure 5, which shows a syndrome extraction circuit with three flags  $F$ ,  $G$ , and  $H$ . Here, while  $H$  detects a propagation error on  $F$  and  $G$ ,  $F$  and  $G$  both detect the same propagation error. Thus, measuring  $H$  is unnecessary as  $F$  and  $G$  detect the propagation error.

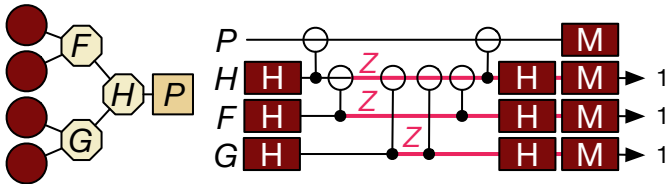


Fig. 5. An example of a flag  $H$  which provides redundant information about a propagation error detected by flags  $F$  and  $G$ .

### B. Proxy Qubits

As an alternative to flag qubits, we propose *proxy qubits* as a secondary mechanism for sparsifying connectivity. Unlike flag qubits, proxy qubits need not be measured nor entangled with parity qubits at the start of a syndrome extraction round. Thus, they avoid the same overuse problem seen with flag qubits. Figure 6 depicts the example of a proxy qubit  $x$  used to entangle data qubits  $a$  and  $b$  with parity qubit  $P$ . Here, the proxy qubit  $x$ , initialized in  $|0\rangle$ , is entangled with  $a$  to form a GHZ state after  $CNOT(a, x)$ . Then, operation  $CNOT(x, P)$  effectively performs the CNOT between  $a$  and  $P$ . Finally,  $CNOT(a, x)$  undoes the GHZ state, ideally returning  $x$  to  $|0\rangle$ . This same procedure is repeated for qubit  $b$ .

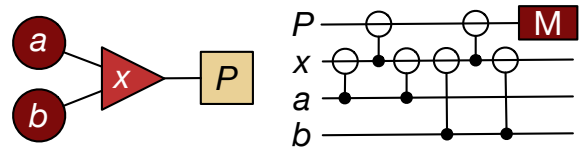


Fig. 6. Proxy  $x$  is used to entangle data qubits  $a$  and  $b$  with parity qubit  $P$ .

Nevertheless, proxy qubits are susceptible to certain errors during syndrome extraction. We discuss how three such errors can be managed to avoid harming the block error rate.

**Type 1 Errors:** These errors are  $X$  ( $Z$ ) errors when measuring a  $Z$  ( $X$ ) check. We find that these errors result in the proxy qubit returning to the  $|1\rangle$  state instead of the  $|0\rangle$  state, which results in measurement-like errors on the parity measurements. These errors do not reduce the effective distance.

**Type 2 Errors:** These errors result from improper CNOT orientation, namely  $Z$  errors while measuring an  $X$  check. Figure 7(a) and Figure 7(b) present two ways of entangling a data qubit  $a$  and a parity qubit  $P$  via a proxy  $x$ . Figure 7(a)'s method is more erroneous as it causes more measurement-like errors when measuring  $P$ . These errors stem from three sources:

- 1) A  $Y/Z$  error on  $x$  or  $P$  in the first CNOT (about  $p/2$ ).
- 2) A  $Y/Z$  error on  $x$  in the second CNOT ( $p/4$ ).
- 3) A  $Y/Z$  error on  $P$  in the last CNOT ( $p/4$ ).

In total, the error probability is  $p$ . In contrast, Figure 7(b)'s circuit only has an error probability  $p/2$ , as only  $Y/Z$  errors in the first two CNOTs can cause a measurement-like error. Our studies also confirm that Figure 7(b)'s circuit yields lower error rates.

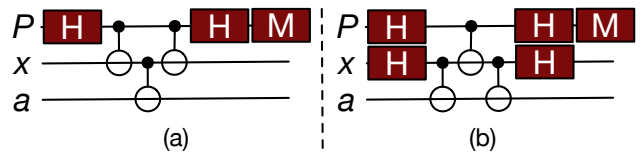


Fig. 7. Examples of two possible CNOT orders to entangle  $a$  and  $P$  through a proxy  $x$ .

**Type 3 Errors:** These are propagation errors that result from the misapplication of proxy qubits, as in Figure 9. Here, data qubits  $a$  and  $b$  are simultaneously entangled to parity qubit  $P$  via proxy qubit  $x$ . When a  $Z$  error occurs on  $x$ , it propagates to  $a$  and  $b$ . To avoid such errors,  $a$  and  $b$  must be entangled to  $P$  separately. Note that such errors only occur when  $a$  and  $b$  are both data qubits. If both qubits are flags, they can be entangled simultaneously with  $P$ , as any resulting propagation error is detectable by measuring the flag qubits.

**Measuring Proxy Qubits?** We found that measuring proxy qubits (like flag qubits) to detect errors results in rather complex syndromes. We leave this avenue for future research.

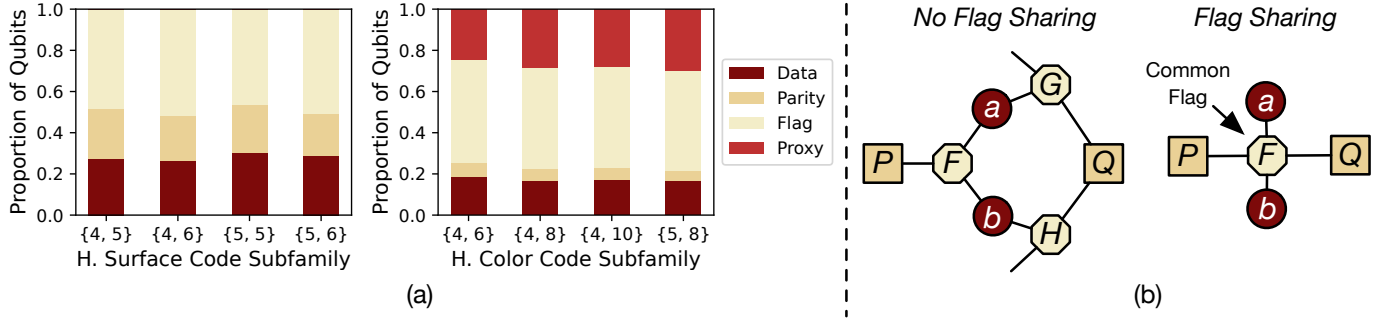


Fig. 8. (a) Qubit overheads by type (data, parity, etc.) for subfamilies of hyperbolic surface and color codes. (b) Example of flag sharing applied to data qubits  $a$  and  $b$ , which have common checks  $P$  and  $Q$ . Flag sharing reduces overheads and connectivity demands.

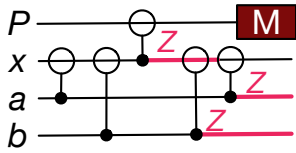


Fig. 9. Misusing proxy  $x$  results in a propagation error onto data qubits  $a$  and  $b$ .

### C. Flag-Proxy Networks

Flag and proxy qubits offer two methods of reducing connectivity requirements. While one or the other can be used indiscriminately, we note that both approaches have pitfalls. Only using flag qubits will result in flag overuse, which can harm the decoder, whereas only using proxy qubits will not guarantee fault-tolerant syndrome extraction. Ideally, we want an architecture that supports fault-tolerant syndrome extraction without overusing flag qubits.

Our key insight towards this goal is as follows: *given a minimal set of flag qubits that protect data qubits from propagation errors, proxy qubits can further reduce connectivity without harming the effective distance*. We present this argument formally in Theorem 1, whose proof is available in Appendix A. Nevertheless, we leverage this insight to design *Flag-Proxy Networks (FPNs)*. FPNs meet all our criteria for a good architecture: (1) its connectivity can be made arbitrarily low with flag and proxy qubits, (2) it supports fault-tolerant syndrome extraction by leveraging flag qubits, and (3) it avoids flag overuse by using proxy qubits.

In general, computing a minimal set of flag qubits is difficult and depends on syndrome extraction, a code's logical operators, and the decoder [11, 13]. For simplicity, we set up the flag layer as in Figure 10, where  $\delta/2$  flags detect propagation errors in a weight- $\delta$  check such that each flag is assigned to a pair of data qubits. This setup is fault-tolerant, as any propagation error can be detected by one or more flags and is amenable to the decoders we consider in this paper.

**Theorem 1.** *Suppose that a Flag-Proxy Network without proxies is fault-tolerant. The same network with proxies is also fault-tolerant.*

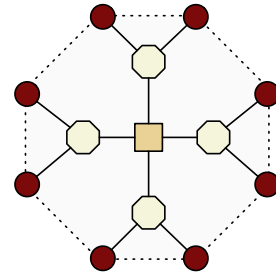


Fig. 10. Flag setup for a weight-8 check. Note that this setup may need proxies to meet connectivity constraints.

### D. Constructing Flag-Proxy Networks

We briefly discuss how to construct FPNs given a quantum code. To begin with, start with a naïve architecture that connects data qubits to parity qubits. While this architecture may violate connectivity constraints, we can introduce flag and proxy qubits to alleviate the connectivity demands. Flag qubits should be introduced according to a fault-tolerant flag protocol. For simplicity, we use the flag protocol in Figure 10, which uses many flags but is guaranteed to be fault-tolerant. After introducing flags, any high-degree qubits can be reduced to lower-degree qubits through proxy qubits, as in Figure 11.

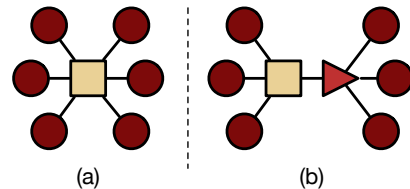


Fig. 11. (a) A degree-6 qubit transformed into (b) a degree-4 qubit by adding a proxy qubit.

**How many flag and proxy qubits?** The number of flag and proxy qubits depends on the underlying quantum code and flag protocol. If a dense code is used with a flag protocol that uses few flags, then many proxies may be required to maintain connectivity constraints. With the flag protocol used in this paper, the hyperbolic surface codes do not need proxy qubits as they have, at worst, degree-6 connectivity. In contrast, the hyperbolic color codes are very dense and need a few (at most three) proxy qubits to achieve degree-4 connectivity.

### E. Reducing Overheads with Flag Sharing

Figure 8(a) shows the average qubit composition within an FPN for different subfamilies of hyperbolic codes. Flag qubits make up *almost half* of all qubits and are thus the most significant contributor to physical overheads. Ideally, we want to support fault-tolerant syndrome extraction without having such exorbitant overheads.

To this end, we propose *flag sharing* within FPNs, as shown in Figure 8(b). Our key insight here is that checks often share at least two data qubits; note that this is true for practically all error-correcting codes. Thus, we merge flag qubits between checks with common data qubits to reduce the number of flag qubits. To optimize overheads across the entire code, we pair data qubits together using *maximum weight matching*, where the weight between a pair of qubits is the number of common checks. With this strategy, flag sharing reduces flag overheads by 10%. Furthermore, flag sharing also removes the need for proxy qubits outside the 4,10 and 5,8 hyperbolic color codes subfamilies.

**Results:** Figure 12 further compares the effective rate of FPNs for hyperbolic codes with and without flag sharing. The effective rate for the standard implementation of a  $d = 5$  planar surface code is marked for reference. Flag sharing improves the effective rate by  $1.2\times$  and  $2.4\times$  for hyperbolic surface and color codes, respectively. We also find that FPNs for the hyperbolic codes strictly outperform the  $d = 5$  planar surface code, which has an effective rate of  $1/49$ . FPNs of the hyperbolic surface codes outperform the  $d = 5$  planar surface code by  $2.9\times$  on average and up to  $4.6\times$ . Concurrently, FPNs of the hyperbolic color codes outperform the  $d = 5$  planar surface code by  $5.5\times$  on average and up to  $6.8\times$ . These FPNs will only further outperform the planar surface code when considering larger code distances.

Table I compares the highest mean degree of an FPN (with flag sharing) in each subfamily to the standard implementations of the  $d = 3, 5, 7$  planar surface codes. Note that the the maximum degree of each FPN is four, the same as the planar surface code. The lower connectivity of the hyperbolic codes is because flag sharing causes each data qubit to be connected to two flag qubits. In contrast, most data qubits in the surface code are connected to three or four parity qubits.

TABLE I  
HIGHEST MEAN DEGREE BY SUBFAMILY

Family	Subfamily	Highest Mean Conn.
H. Surface Code	{4, 5}	2.98
	{4, 6}	2.94
	{5, 5}	3.12
	{5, 6}	3.11
H. Color Code	{4, 6}	2.80
	{4, 8}	2.94
	{4, 10}	2.90
	{5, 8}	2.93
P. Surface Code	$d = 3$	2.82
	$d = 5$	3.26
	$d = 7$	3.46

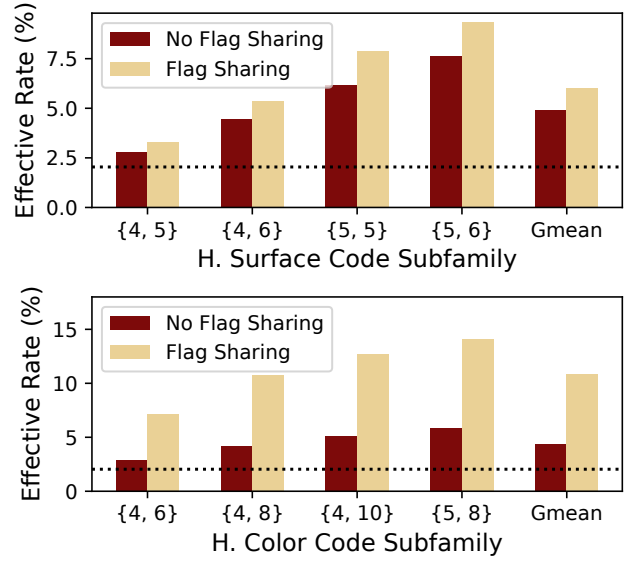


Fig. 12. Effective rates for FPNs with and without flag sharing for hyperbolic codes. The effective rate of a  $d = 5$  planar surface code ( $1/49$ ) is marked for reference.

## V. SYNDROME EXTRACTION

While FPNs provide an architecture for a quantum code, detecting errors on the code requires executing a syndrome extraction circuit. In this section, we detail the challenges of syndrome extraction scheduling and present a general algorithm for syndrome extraction scheduling.

### A. Constraints of Syndrome Extraction

Functionally correct syndrome extraction schedules must abide by two constraints [4, 19]. In this section, we use the notation  $t_K(q)$  to refer to the timestep a qubit  $q$  has a CNOT when measuring check  $K$ .

**Uniqueness:** A data qubit  $q$  can only perform one CNOT at a time. That is,  $t_{K_i}(q) \neq t_{K_j}(q)$  where  $K_i$  and  $K_j$  are checks. Similarly, a parity qubit for  $K_i$  can only perform one CNOT at a time:  $t_{K_i}(q_1) \neq t_{K_i}(q_2)$  for  $q_1, q_2 \in K_i$ .

**Commutation:** Let  $K_X$  be an  $X$  check and  $K_Z$  be some  $Z$  check. If  $K_X$  and  $K_Z$  have common qubits  $\text{Comm}$ , then the relationship in Equation (6) must hold.

$$\prod_{q \in \text{Comm}} (t_{K_X}(q) - t_{K_Z}(q)) > 0 \quad (6)$$

Due to these two constraints, the worst-case latency for a syndrome extraction schedule is one where  $X$  checks and  $Z$  checks are measured disjointly; such a schedule has depth  $\max(\delta_X) + \max(\delta_Z)$  where  $\delta_X$  and  $\delta_Z$  are the  $X$  and  $Z$  check weights of the code. To the best of our knowledge, the only prior work on syndrome extraction scheduling is a coloring algorithm that guarantees this worst-case depth [45]. We aim to achieve better-than-worst-case syndrome extraction depth to minimize decoherence, dephasing, and idling errors.



## B. Difficulty of Scheduling

Optimal low-depth scheduling is NP-Hard, and state-of-the-art solvers cannot compute optimal schedules beyond the smallest quantum codes. Indeed, QLDPC codes are rather large, and thus, solvers cannot be used to schedule these codes. However, scheduling is relatively straightforward when considering planar codes, such as the planar surface code, as such codes are *translation invariant*. Translation invariance implies that checks on the code are “locally identical”. For codes with translation invariance, optimal schedules can be obtained by computing a low-depth schedule for a handful of checks and then reusing these schedules for the entire code, as shown in Figure 13 for the planar surface code. Unfortunately, QLDPC codes are not necessarily translation invariant. For codes without translation invariance, we need an alternative method of computing syndrome extraction schedules.

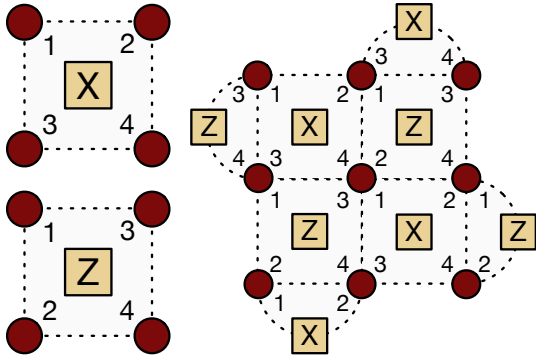


Fig. 13. Translation invariance on the planar surface code, where the same X and Z schedules are reused for every such check.

## C. Reducing the Complexity of the Problem

Computing syndrome extraction schedules is not tractable for solvers for two reasons. *First*, a solver will need at most  $(n-k)\delta_{\max}$  “time” variables ( $t_K(q)$ ) to encode the scheduling problem, as  $\delta$  variables are required for each weight- $\delta$  check. Thus, a code with about 100 qubits will require thousands of time variables. *Second*, uniqueness and commutation constraints require additional “auxiliary” variables and constraints to implement in practice. Equation (7) presents the encoding of a uniqueness condition, where each condition requires two constraints and an additional auxiliary variable  $x$ . Commutation constraints, as in Equation (8), require  $2|\text{Comm}| + 1$  constraints and  $|\text{Comm}| + 1$  auxiliary variables.

$$t_{K_i}(q) \neq t_{K_j}(q) \rightarrow \begin{cases} t_{K_i}(q) - Mx \leq t_{K_j}(q) - 1 \\ t_{K_i}(q) - M(1-x) \geq t_{K_j}(q) + 1 \\ x \in \{0, 1\}, M \gg 0 \end{cases} \quad (7)$$

$$\prod_{q \in \text{Comm}} (t_{K_X}(q) - t_{K_Z}(q)) > 0 \rightarrow \begin{cases} \sum_{q \in \text{Comm}} x_q = 2y \\ t_{K_X}(q) - t_{K_Z}(q) \leq Mx_q \\ t_{K_Z}(q) - t_{K_X}(q) \leq M(1-x_q) \\ x_q \in \{0, 1\}, y > 0, M \gg 0 \end{cases} \quad (8)$$

Leveraging a solver to compute a schedule requires significantly reducing these variable and constraint overheads. Our insight towards this goal is to *compute locally optimal schedules* instead of globally optimal ones. Locally optimal scheduling reduces variable and constraint overheads in two ways. *First*, by scheduling a single weight- $\delta$  check, we only require  $\delta$  time variables. *Second*, uniqueness and commutation constraints only need to be considered for adjacent checks that share data qubits. Thus, the runtime of the solver instead depends on *the size of the check* instead of the size of the code. We denote the runtime for a weight- $\delta$  check as  $T(\delta)$ .

## D. Greedy Scheduling Algorithm

We present a greedy algorithm for syndrome extraction scheduling, which leverages our prior insight, shown in Algorithm 1. The complexity of this algorithm, which schedules checks sequentially, is  $O((n-k)T(\delta_{\max}))$  for an  $[[n, k, d]]$  code. In practice, our algorithm is fast as  $T(\delta_{\max}) \approx O(100\text{ms})$  for commercial solvers in the worst case.

---

### Algorithm 1: Greedy Scheduling Algorithm

---

**Input:** Checks  $K_1, \dots, K_{n-k}$

**Output:** A CNOT Schedule

---

Let  $\delta_{\max}$  be the maximum check weight.

Create a table of scheduled CNOT times  $T(K, q)$ .

**for**  $1 \leq i \leq n-k$  **do**

Create the following program for check  $K_i$ :

**min.**  $t_{\max}$

**s.t.**  $t_{K_i}(q_1) \neq t_{K_i}(q_2)$  where  $q_1, q_2 \in K_i$

$t_{K_i}(q) \neq T(K_j, q)$  where  $q \in \text{Comm}(K_i, K_j)$

$\prod_{q \in \text{Comm}(K_i, K_j)} (t_{K_i}(q) - T(K_j, q)) > 0$

$t_{\max} \geq t_{K_i}(q)$

$1 \leq t_{K_i}(q) \leq 2\delta_{\max}$

where  $1 \leq j < i$ .

Use a solver to compute a solution to the program.

Assign  $T(K_i, q) := \text{Result}(t_{K_i}(q))$  for all  $q \in K_i$ .

**end**

**return**  $T(K, q)$  as a schedule of CNOTs.

---

## E. Validity of the Greedy Algorithm

To output a valid schedule, the greedy algorithm (Algorithm 1) must abide by global uniqueness and commutativity constraints. We discuss how the greedy algorithm does so while scheduling checks sequentially. Without loss of generality, suppose the greedy algorithm is currently scheduling a check  $K_i$  and has already scheduled check  $K_{i-1}$ , and suppose  $K_i$  and  $K_{i-1}$  share two qubits,  $a$  and  $b$ . Note that the steps below are repeated for any scheduled check  $K_j$  that shares qubits with  $K_i$ .

1) *Uniqueness*: As  $K_{i-1}$  has already been scheduled, we know that qubits  $a$  and  $b$  are already scheduled to perform a CNOT. Say these times are  $t_a = T(K_{i-1}, a)$  and  $t_b = T(K_{i-1}, b)$ . Then, when scheduling check  $K_i$ , the greedy algorithm requires that qubits  $a$  and  $b$  are not scheduled at times  $t_a$  and  $t_b$ , respectively. This ensures that qubits  $a$  and  $b$  perform only one CNOT at any given time, ensuring global uniqueness constraints are maintained.

2) *Commutativity*: As commutativity constraints need not be applied if  $K_i$  and  $K_{i-1}$  are both  $X$  or both  $Z$  checks, without loss of generality, assume  $K_i$  is a  $Z$  check and  $K_{i-1}$  is an  $X$  check. Like before, we know that qubits  $a$  and  $b$  are already scheduled for a CNOT for check  $K_{i-1}$  at times  $t_a$  and  $t_b$ . To ensure global commutativity constraints are met, the greedy algorithm directs the solver to enforce the constraint  $(t'_a - t_a)(t'_b - t_b) > 0$ , where  $t'_a, t'_b$  are solver variables.

### E. Performance of Greedy Algorithm

To evaluate the performance of the greedy algorithm, we first discuss the *theoretically shortest circuit* for each code. For a code with a maximum check weight of  $\delta$ , the theoretically shortest circuit will have  $2H$  gates,  $\delta$  CNOT gates, and a measurement+reset gate. On the other hand, the *longest possible circuit* will forego commutation constraints and schedule  $X$  and  $Z$  checks separately and thus will have  $2H$  gates,  $\delta_X + \delta_Z$  CNOT gates, and a measurement+reset gate. Under our timing model from Section III-A, the shortest possible circuit and longest possible circuit will have latencies of  $(890 + 40\delta)$ ns and  $(890 + 40\delta_X + 40\delta_Z)$ ns.

Figure 14 compares the output syndrome extraction circuits from the greedy algorithm to that of the theoretically shortest and longest circuits. In all cases but for the  $\{4, 5\}$  hyperbolic surface codes, the mean latency observed is less than that of the theoretical longest latency. Furthermore, we find the greedy algorithm performs better for denser codes, as the difference between the theoretical shortest and longest latency is much larger for these codes. Nevertheless, our algorithm is the first to offer better-than-worst-case syndrome extraction latency, and we expect future work to improve upon our results.

### G. Scheduling for FPNs

Algorithm 1 schedules CNOTs between data and parity qubits and can be used outside of FPNs. However, by itself, Algorithm 1 does not (1) consider the layout dictated by an FPN and (2) support fault-tolerant syndrome extraction with flag and proxy qubits. In this section, we discuss minor modifications to the algorithm to operate with FPNs.

1) *Flag Qubit Modifications*: Flag qubits must be (1) initialized, (2) perform CNOTs with data qubits, and (3) measured. During initialization and measurement, flag qubits perform CNOTs with the parity qubits. These CNOTs have no constraints beyond uniqueness and can be done greedily. When performing CNOTs with data qubits, flag qubits present an opportunity for parallelism during syndrome extraction. To leverage this parallelism, uniqueness constraints must be modified by replacing all parity qubit constraints with flag

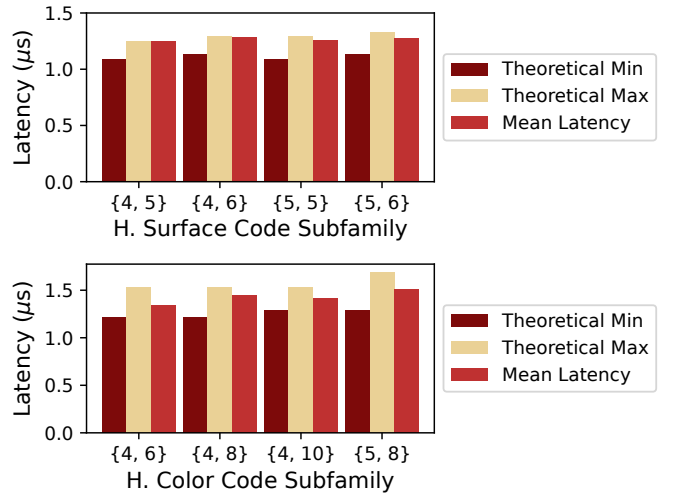


Fig. 14. Syndrome extraction latencies for schedules computed by the greedy algorithm (Algorithm 1).

qubit constraints. If a flag qubit  $F$  operates on qubits  $q_1$  and  $q_2$ , then  $t_F(q_1) \neq t_F(q_2)$  must hold. Note that if  $q_1, q_2 \in K$  and  $F$  is used in the syndrome extraction of  $K$ , then  $t_F(q_1) = t_K(q_1)$  and  $t_F(q_2) = t_K(q_2)$ . Furthermore, if checks  $K_i$  and  $K_j$  share flag  $F$ , then the constraint  $t_{K_i}(q) \neq T(K_j, q)$  must be replaced with  $t_{K_i}(q) = T(K_j, q)$  as  $q$  has already been entangled with  $K_i$  through  $F$ .

2) *Proxy Qubit Modifications*: Proxy qubits do not require any modifications in Algorithm 1 beyond handling CNOTs between two non-adjacent qubits. Here, we compute the shortest path between two non-adjacent qubits such that the interior edges of this path do not pass over data, parity, or flag qubits. Then, we perform CNOTs along this path.

3) *Results*: We briefly compare the output syndrome extraction latencies for FPNs of the hyperbolic codes to that of a standard implementation of the planar surface code. Under our timing model in Section III-A, the planar surface code has a syndrome extraction latency of about  $1\mu$ s. In comparison, the hyperbolic surface and color codes have worst-case latencies of  $2.3\mu$ s and  $3.4\mu$ s. The longer latency of the hyperbolic surface code results from flag sharing, and the longer latency of the hyperbolic color codes result from  $X$  and  $Z$  checks being measured separately. Nevertheless, these latencies are comparable to that of the planar surface code.

## VI. DECODING WITH FLAG QUBITS

While FPNs realize error-correcting codes and provide fault-tolerant syndrome extraction circuits, correcting errors requires using a decoder to identify data qubit errors. In this section, we present a generalizable flag protocol applicable to FPNs. We further present modifications of *Minimum-Weight Perfect Matching (MWPM)* [17, 25, 27] and *Restriction* [11, 31] decoders which leverage this flag protocol to decode the hyperbolic codes.

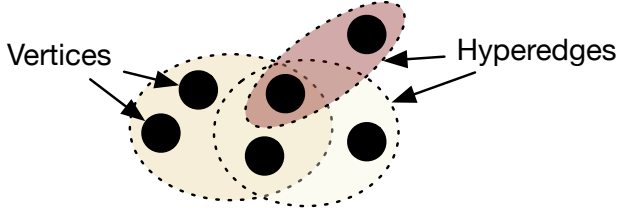


Fig. 15. A hypergraph with three hyperedges. The primary difference between a graph and a hypergraph is that a hyperedge can connect  $\geq 2$  vertices, whereas an edge can only connect 2 vertices.

### A. Decoding Hypergraph

We represent all possible errors during syndrome extraction in a *decoding hypergraph*. Hypergraphs, shown in Figure 15, extend graphs by using *hyperedges*, which connect arbitrarily many vertices. The vertices of a decoding hypergraph correspond to syndrome bits, and the hyperedges correspond to error events. A hyperedge has the following properties:

- 1) A set of syndrome bits  $\sigma(e)$  flipped by the corresponding error event.
- 2) A set of flag bits  $f(e)$  flipped by the corresponding error event.
- 3) An error probability  $\pi(e)$ .
- 4) A set of affected Pauli frames  $\lambda(e)$ . Each Pauli frame corresponds to either an  $X$  or  $Z$  error on a logical qubit. These errors are tracked by the decoder in the software.

Finally, we say a hyperedge is a *flag hyperedge* if  $|f(e)| > 0$  and otherwise call it a *normal hyperedge*.

### B. Error Equivalence Classes

Given the flag syndrome from syndrome extraction, we must determine whether or not to use flag hyperedges. For hyperbolic codes, we observe the following patterns in flag hyperedges:

- 1) A flag hyperedge  $e_f$  may have the same syndrome bits as a normal hyperedge  $e_n$ , but may affect different Pauli frames. That is  $\sigma(e_f) = \sigma(e_n)$ , yet  $\lambda(e_f) \neq \lambda(e_n)$ .
- 2) Two flag error events  $e_f$  and  $e_g$  may flip the same syndrome bits but may flip different flag bits. That is  $\sigma(e_f) = \sigma(e_g)$  but  $f(e_f) \neq f(e_g)$ .

To handle these situations, we propose categorizing hyperedges into *equivalence classes*. Two edges  $e_i$  and  $e_j$  reside in the same equivalence class  $C$  if  $\sigma(e_i) = \sigma(e_j)$ . Furthermore, during decoding, given a set of flag bits  $F$ , a single representative  $\bar{e}$  is chosen from  $C$  such that  $|f(\bar{e}) \oplus F|$  is minimized. Furthermore, if  $|F| > 0$ ,  $\pi(\bar{e})$  is renormalized as in Equation 9, where  $p_M$  is the measurement error probability. Thus, we select the most probable error event from each equivalence class given a set of flag bits.

$$\pi(\bar{e}) \rightarrow p_M^{|f(\bar{e}) \oplus F|} \pi(\bar{e})^{|\sigma(\bar{e})| - 1} \quad (9)$$

We briefly provide an example of how to leverage error equivalence classes. Consider the three error classes,  $C_1$ ,  $C_2$ , and  $C_3$ , shown in Table II. We run through three possible syndromes we might obtain during syndrome extraction.

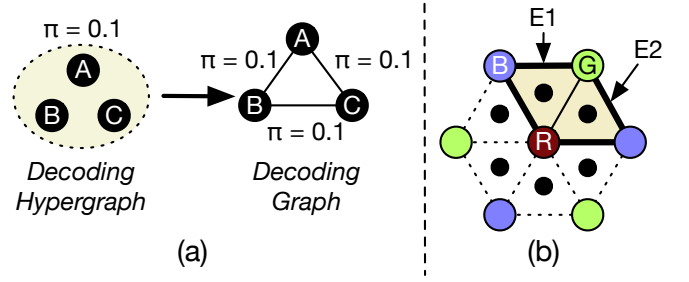


Fig. 16. (a) Example of a hyperedge between vertices  $A$ ,  $B$ , and  $C$  being translated to three edges. (b) The edges from matching (bolded) are used to identify hyperedges  $E1$  and  $E2$  by lifting.

**1: Syndrome** =  $\{\sigma_0, \sigma_3\}$ , **no flags**. As there are no flags, we only use the error events from  $C_1$  and  $C_2$  that have no flags:  $e_1 = \{\sigma_0, \sigma_1, \sigma_2\}$  from  $C_1$  and  $e_2 = \{\sigma_1, \sigma_2, \sigma_3\}$ . A decoder should identify that  $e_1$  and  $e_2$  have occurred, as  $\sigma(e_1) \oplus \sigma(e_2) = \{\sigma_0, \sigma_3\}$ .

**2: Syndrome** =  $\{\sigma_0, \sigma_3\}$ , **Flags** =  $\{f_1\}$ . Now that we have measured a flag bit, we must select events from each class whose flag bits are closest in similarity to  $f_1$ . From  $C_1$ , we select  $e_1 = \{\sigma_0, \sigma_1, \sigma_2\}, \{f_1\}$  as it is only one flag off. From  $C_2$ , we select  $e_2 = \{\sigma_1, \sigma_2, \sigma_3\}, \{f_1\}$  as it is the only option. From  $C_3$ , we select the flag edge  $e_3 = \{\sigma_0, \sigma_3\}, \{f_1\}$ . A decoder should identify that  $e_3$  has occurred as  $\sigma(e_3) = \{\sigma_0, \sigma_3\} \neq$ .

**3: Syndrome** =  $\{\sigma_0, \sigma_1, \sigma_2\}$ , **Flags** =  $\{f_2, f_3\}$ . From  $C_1$ , we select  $\{\sigma_0, \sigma_1, \sigma_2\}, \{f_1, f_2, f_3\}$ . From  $C_2$ , we select  $\{\sigma_1, \sigma_2, \sigma_3\}, \{f_1\}$ . From  $C_3$ , we select  $\{\sigma_0, \sigma_3\}, \{f_2, f_3\}$ . A decoder should identify that  $e_1$  has occurred.

TABLE II  
EXAMPLE OF ERROR EQUIVALENCE CLASSES

Class $C_1$	Class $C_2$	Class $C_3$
$\{\sigma_0, \sigma_1, \sigma_2\}, \{f_1, f_2, f_3\}$	$\{\sigma_1, \sigma_2, \sigma_3\}, \{f_1\}$	$\{\sigma_0, \sigma_3\}, \{f_2, f_3\}$

### C. Decoding Hyperbolic Surface Codes

Our MWPM decoder for hyperbolic surface codes operates as follows. First, the decoding hypergraph must be translated into a decoding graph. To do so, we collect representatives  $\bar{e}$  from each error equivalence class. Then, for each  $\sigma_i, \sigma_j \in \sigma(\bar{e})$ , an edge  $(\sigma_i, \sigma_j)$  is created in the decoding graph and is assigned a weight  $w_{ij} = -\log \pi(\bar{e})$ . Figure 16(a) shows an example of this translation.

Given the decoding graph, we must form a fully connected graph  $G$ , where vertices in  $G$  are flipped syndrome bits. Each pair of flipped syndrome bits  $(\sigma_i, \sigma_j)$  is assigned an edge whose weight  $w_{ij}$  is equal to the weight of the shortest path between  $\sigma_i$  and  $\sigma_j$  in the decoding graph. Conceptually, this path corresponds to the most probable set of errors causing  $\sigma_i$  and  $\sigma_j$  to flip. Next, we pair all flipped syndrome bits to minimize the sum  $\sum_{\sigma_i \leftrightarrow \sigma_j} w_{ij}$  to form a minimum-weight perfect matching.

To identify errors, if  $\sigma_i$  and  $\sigma_j$  are matched, we retrieve all edges in the path between  $\sigma_i$  and  $\sigma_j$  in the decoding graph. Usually, an edge  $x$  in the path is present in the decoding hypergraph: in this situation, we update all Pauli frames  $\lambda(x)$ . However, when  $x$  does not correspond to any edge, it is a flag edge, so we select the most similar flag edge  $x_f$  and update all Pauli frames  $\lambda(x_f)$ .

#### D. Decoding Hyperbolic Color Codes

Our Restriction decoder for hyperbolic color codes operates as follows. For the color codes, each syndrome bit  $\sigma_i$  is associated with some color  $C(\sigma_i) \in \{R, G, B\}$ . Using these colors, we define three decoding graphs called *restricted lattices*:  $L_{RG}$ ,  $L_{RB}$  and  $L_{GB}$ , which only contain syndrome bits of the specified colors. Then, we compute a minimum-weight perfect matching on these restricted lattices.

To identify errors, we must translate the matchings into errors on the color code to identify errors. First, we collect all edges present in paths for each matching: we call this set  $E_M$ . If any flag edge  $e_f$  appears twice in  $E_M$ , we immediately correct all Pauli frames  $\lambda(e_f)$  and remove  $e_f$  from  $E_M$ . This occurs if two syndrome bits are matched in different restricted lattices, and both matchings' paths contain  $e_f$ .

All remaining edges are used in a lifting procedure; an example of lifting is shown in Figure 16(b). First, all edges in  $E_M$  are flattened such that their endpoints correspond to syndrome bits in the first round of syndrome extraction. This step handles measurement errors, which do not affect the logical state. Next, we identify all syndrome bits colored  $R$  that are also incident on some edge in  $E_M$ . Then, for each incident syndrome bit, we select a maximal subset of incident edges from  $E_M$ : for each hyperedge  $e$  outlined by this subset, we correct all Pauli frames  $\lambda(e)$ . We repeat the lifting procedure until  $E_M$  is empty.

#### E. Performance of Flagged Decoders

TABLE III  
EVALUATED HYPERBOLIC CODES

Family	Subfamily	$n$	$k$	$d_X$	$d_Z$
H. Surface Code	{4, 5}	160	18	8	6
	{5, 5}	150	32	6	6
H. Color Code	{4, 6}	216	40	8	8
	{5, 8}	360	130	6	6

Figure 17 and Figure 18 compare  $BER_{\text{norm}}$  for the hyperbolic surface and color codes in Table III to the  $d = 5$  and  $d = 7$  planar surface and (6.6.6) color codes, respectively. Our setup for the planar color code uses the flag protocol proposed by Chamberland et al. [11].

We find that the hyperbolic codes have competitive error rates with their planar counterparts while having higher  $R_{\text{eff}}$ . In particular, the  $[[150, 32, 6, 6]]$  hyperbolic surface code requires 424 physical qubits to encode 32 logical qubits while having comparable error rates to the  $d = 5$  planar surface code, which would require 1568 physical qubits. Similarly, the  $[[216, 40, 8, 8]]$  hyperbolic color code requires 512 physical

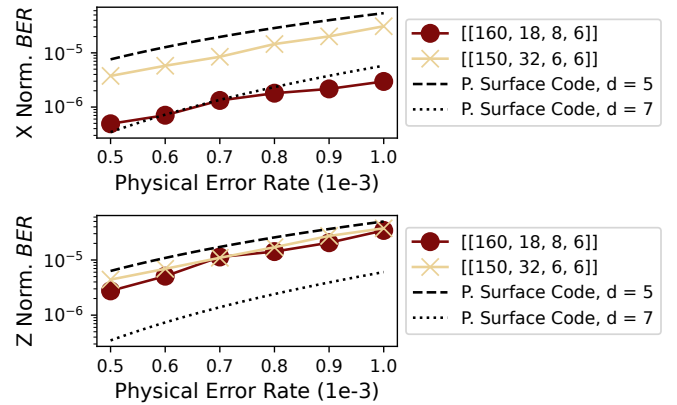


Fig. 17.  $BER_{\text{norm}}$  for surface codes.

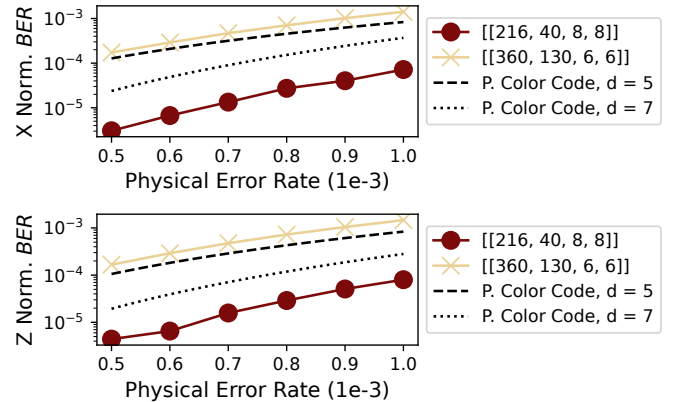


Fig. 18.  $BER_{\text{norm}}$  for color codes.

qubits to encode 40 logical qubits while having comparable error rates to the  $d = 7$  planar color code, which would require from 2200 to 4000 physical qubits, depending on implementation [11, 32].

#### F. Comparison with Prior Decoders

We briefly compare the proposed flagged decoders with prior work for planar codes. For hyperbolic surface codes, we compare against PyMatching [27], an open source MWPM decoder for planar surface codes that has been used in much prior work [23, 26, 46, 47]. For hyperbolic color codes, we compare against Chromobius [22], a recent open-source planar color code decoder, and Chamberland et al.'s Restriction decoder [11], a planar color code decoder that uses flag qubits. As PyMatching and Chromobius do not work with flag qubits, we test them on architectures where parity qubits are directly connected to data qubits. Furthermore, as Chamberland et al.'s original work did not consider a generalized flag protocol, we modified the decoder to use our flag protocol during decoding.

1) *Surface Codes*: In this section, we consider a  $[[30, 8, 3, 3]]$  hyperbolic surface code from the  $\{5, 5\}$  subfamily. Figure 19 shows the  $X$  and  $Z$   $BER$  for PyMatching and our flagged MWPM decoder. When handling  $Z$  errors, PyMatching only achieves  $d_{\text{eff}} = 2$ , whereas our flagged MWPM decoder achieves the full code distance ( $d_{\text{eff}} = 3$ ) by leveraging flag measurements.

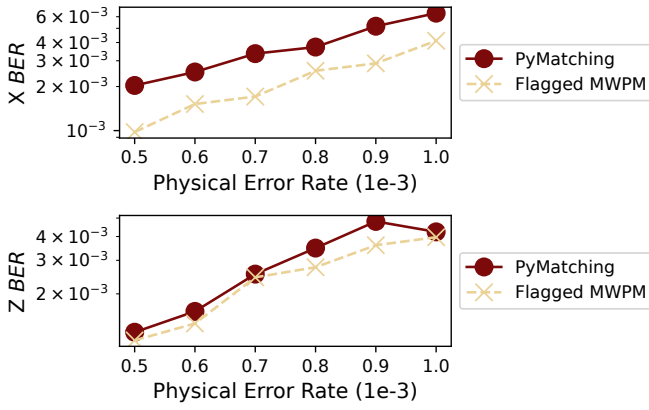


Fig. 19.  $BER$  of a  $[[30, 8, 3, 3]]$  hyperbolic surface code using PyMatching and the flagged MWPM decoder.

2) *Color Codes*: In this section, we consider a  $[[24, 8, 4, 4]]$  hyperbolic color code from the  $\{4, 6\}$  subfamily. Chromobius, unfortunately, cannot decode the syndrome extraction circuit as it cannot handle error events where two parity qubits of the same color are flipped due to a CNOT propagation error. We also find that Chromobius cannot accurately decode when CNOT errors are disabled. Thus, Figure 20 shows the  $X$  and  $Z$   $BER$  only for Chamberland et al.’s decoder and our flagged Restriction decoder. Chamberland et al.’s decoder only achieves  $d_{\text{eff}} = 2$ , whereas the flagged Restriction decoder achieves the full code distance ( $d_{\text{eff}} = 4$ ).

Given that the flag protocol used by both decoders is the same, we find that the flagged Restriction decoder outperforms Chamberland et al.’s decoder because Chamberland et al.’s decoder only handles flag edges in the MWPM stage of the decoder. In contrast, we handle flag edges outside the MWPM stage. Specifically, much of the improvement comes from where we handle flag edges  $e_f$  that appear twice in  $E_M$ , as stated in Section VI-D.

## VII. RELATED WORK

### A. Architectural Construction

To the best of our knowledge, the only prior work in this area is by Tremblay et al. concerning architectures for *Hypergraph Product (HGP)* codes [43, 45]. We note that the architecture considered in their paper is limited to HGP codes and has dense connectivity (at most degree-8 for the codes considered in their paper). Furthermore, using the authors’ proposal, it is unclear how to support fault-tolerant syndrome extraction with flag qubits.

### B. Code Mapping

Code mapping compilers, namely *Surf-Stitch* [46] and *Code-Stitch* [47], seek to map quantum codes to pre-existing quantum processors. The primary limitation of both these works concerns their support for fault-tolerant syndrome extraction. **Surf-Stitch** exclusively uses flag qubits to meet connectivity requirements, but this approach falls victim to the flag overuse problem. It also does not offer a flag protocol for handling flag syndromes, and thus, the results observed in the paper indicate

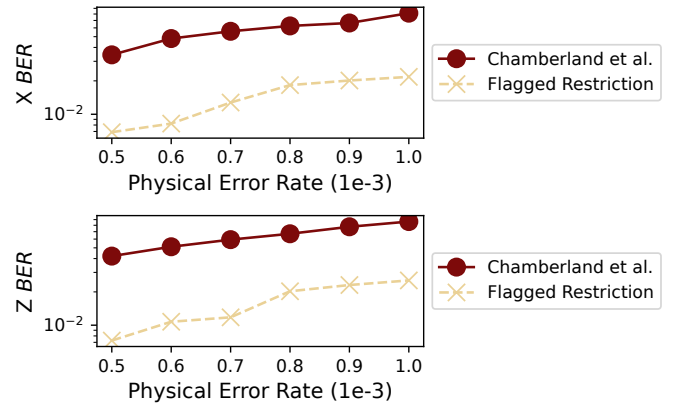


Fig. 20.  $BER$  of a  $[[24, 8, 4, 4]]$  hyperbolic color code using Chamberland et al.’s decoder and the flagged Restriction Decoder.

that the mapped codes are not fault-tolerant. **Code-Stitch** has similar limitations regarding flag qubits but can also use Shor-style syndrome extraction to avoid using flag qubits. Shor-style syndrome extraction is fault-tolerant but has high overheads as measuring a weight- $\delta$  check requires  $\delta$  parity qubits.

In contrast, this paper proposes using proxy qubits to avoid the flag overuse problem and presents a flag protocol to recover the code distance. Nevertheless, note that the goal of code mapping compilers is moreso experimental, as they enable testing quantum codes on systems that do not support their connectivity requirements. In contrast, our goal focuses on relaxing the connectivity demands of quantum codes to facilitate the production of new processors.

## VIII. CONCLUSION

QLDPC codes are a scalable alternative to planar surface code. However, the biggest obstacles towards realizing these codes are practical, namely (1) dense connectivity requirements greater than degree-4, (2) fault-tolerant syndrome extraction circuits, and (3) accurate decoding under circuit-level noise. For the first problem, we propose *Flag-Proxy Networks (FPNs)* as a general architecture that uses flag and proxy qubits to achieve low connectivity while supporting fault-tolerant syndrome extraction. For the second, we propose a greedy scheduling algorithm generalizable to any quantum code. For the third, we propose a flag protocol to correct “propagation errors” during syndrome extraction. Our evaluations on hyperbolic surface and color codes indicate that FPNs of these codes are respectively  $2.9\times$  and  $5.5\times$  more space-efficient than the  $d = 5$  planar surface code. The error rates of these codes are also comparable to their planar counterparts.

## ACKNOWLEDGEMENT

This research was conducted using the *Partnership for an Advanced Computing Environment (PACE)* cluster at Georgia Tech.

## A. Proof of Theorem 1

**Theorem 1.** *Suppose that a Flag-Proxy Network without proxies is fault-tolerant. The same network with proxies is also fault-tolerant.*

*Proof.* There are two locations where proxy qubits may be added into an FPN: between the data and flag qubits and between the flag and proxy qubits. For brevity, we consider the first location, as our argument for the second is similar.

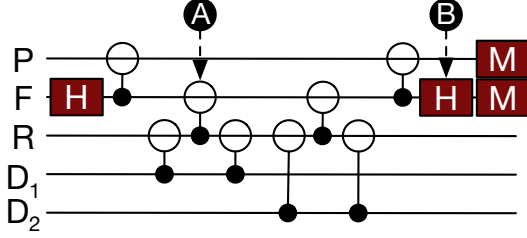


Fig. 21. The circuit used in the proof of Theorem 1.

Consider the  $Z$  syndrome extraction circuit shown in Figure 21, where data qubits  $D_1$  and  $D_2$  are in the joint state  $|D_1, D_2\rangle = \alpha_{00}|00\rangle + \alpha_{01}|01\rangle + \alpha_{10}|10\rangle + \alpha_{11}|11\rangle$ .  $P$ ,  $F$ , and  $R$  are parity, flag, and proxy qubits, respectively. At the start of the circuit, the state is as in Equation (10). The ideal evolution until **A** yields the state in Equation (11). Now, without loss of generality, suppose that  $CNOT(R, F)$  causes  $Z$  errors on both  $R$  and  $F$ . Then, we obtain the state in Equation (12).

$$|D_1, D_2\rangle |R\rangle |F, P\rangle = (\alpha_{00}|00\rangle + \alpha_{01}|01\rangle + \alpha_{10}|10\rangle + \alpha_{11}|11\rangle) \otimes |0\rangle \otimes \frac{|00\rangle + |11\rangle}{\sqrt{2}} \quad (10)$$

$$|D_1, D_2, R, F, P\rangle = \frac{\alpha_{00}|00000\rangle + \alpha_{00}|00011\rangle + \dots + \alpha_{11}|11110\rangle + \alpha_{11}|11101\rangle}{\sqrt{2}} \quad (11)$$

$$|D_1, D_2, R, F, P\rangle = \frac{\alpha_{00}|00000\rangle - \alpha_{00}|00011\rangle + \dots + \alpha_{11}|11110\rangle - \alpha_{11}|11101\rangle}{\sqrt{2}} \quad (12)$$

The subsequent evolution until **B** yields the state in Equation (13). Although a  $Z$  propagation error has occurred on  $D_1$  and  $D_2$ , it is detectable as measuring  $F$  will always yield 1.

$$|D_1, D_2, F, P\rangle = \alpha_{00}|0010\rangle - \alpha_{01}|0111\rangle - \alpha_{10}|1011\rangle + \alpha_{11}|1111\rangle \quad (13)$$

Note that the propagation error resulting from the CNOT error on  $R$  would have occurred even if it did not exist. Syndrome extraction with  $R$  is functionally equivalent to syndrome extraction without  $R$ ; including  $R$  only increases the “effective CNOT error” in the circuit. Hence, our argument above will also extend to fault-tolerant syndrome extraction circuits without flags, as the propagation error would not have harmed the effective distance. Therefore, adding proxies into an already fault-tolerant FPN does not reduce the effective distance.  $\square$

## B. List of Codes

Tables IV and V list all hyperbolic codes evaluated in this paper. These codes were generated with code written using GAP [18], a computer-algebra system, and the code distances were computed through brute-force search in Stim [20]. Codes were generated according to the procedures described by Breuckmann et al. for hyperbolic surface codes [8] and by Higgott and Breuckmann [26] for hyperbolic color codes.

TABLE IV  
LIST OF HYPERBOLIC SURFACE CODES

Subfamily	$R_{\text{ideal}}$	$n$	$k$	$d_X$	$d_Z$	$R_{\text{eff}}(\%)$
{4, 5}	$1/10 = 0.1$	60	8	6	4	4.3
		160	18	8	6	3.6
		360	38	8	8	3.3
		660	68	10	8	3.3
		1800	182	10	10	3.2
		1920	194	12	10	3.2
{4, 6}	$1/6 \approx 0.17$	36	8	4	4	7.2
		336	58	8	6	5.5
		864	146	10	8	5.3
{5, 5}	$1/5 = 0.2$	30	8	3	3	9.4
		40	10	4	4	9.3
		80	18	5	5	8.0
		150	32	6	6	7.5
		900	182	8	8	7.2
{5, 6}	$4/15 \approx 0.27$	60	18	4	3	10.6
		120	34	6	5	10.0
		2520	674	8	6	9.2

TABLE V  
LIST OF HYPERBOLIC COLOR CODES

Subfamily	Asymptotic Rate	$n$	$k$	$d$	$R_{\text{eff}}(\%)$
{4, 6}	$1/6 \approx 0.17$	24	8	4	15.1
		120	24	6	8.3
		216	40	8	7.8
		1320	224	10	7.0
		1440	244	12	6.8
{4, 8}	$1/4 = 0.25$	32	12	4	17.1
		400	104	8	11.1
		2688	676	12	10.6
{4, 10}	$3/10 = 0.3$	40	16	4	18.2
		1000	304	8	12.5
{5, 8}	$7/20 = 0.35$	320	116	4	14.6
		360	130	6	14.3
		1920	676	8	14.0

**Note:** We found the Restriction decoder (and variants like the Möbius decoder [22, 41]) cannot accurately decode several hyperbolic color codes under code capacity noise (no operation error). We often found that such codes often have a lower distance counterpart with the same  $n$  and  $k$ . However, a concurrent work, the *Concatenated MWPM* decoder [34], can accurately decode most hyperbolic color codes (under code capacity noise). However, the decoder needs our flag protocol to achieve the full distance of the hyperbolic color codes under circuit-level noise. As a result of this phenomenon, Table V only contains hyperbolic color codes that the Restriction decoder can accurately decode under code capacity noise.

**Planarity:** All FPNs listed above are *biplanar*, much like bi-variate bicycle codes [5] and the hyperbolic floquet codes [26].

## APPENDIX (ARTIFACT)

### A. Abstract

Our artifacts are the codebase which is used to construct and evaluate FPNs. We have also provided the codes that are used in our evaluations.

Our artifact allows the user to reproduce the following data: Figure 10(a), Figure 12, Table I, Figure 14, Figure 17, Figure 18, Figure 19, and Figure 20, which are all the quantitative results of our paper. After running the requisite experiments, all data is viewable in a provided iPython notebook.

### B. Artifact check-list (meta-information)

- **Program:** `protean`, `pr_base_memory`, `pr_planar_memory`
- **Compilation:** `gcc`, at least version 12
- **Data set:** Hyperbolic quantum codes provided by the authors.
- **Hardware:** FPN generation requires MacOS. BER evaluations require a computing cluster.
- **Execution:** Python and bash scripts which automate the experiments.
- **How much disk space required (approximately)?:** At most 4GB
- **How much time is needed to prepare workflow (approximately)?:** At most 30 minutes
- **How much time is needed to complete experiments (approximately)?:** At most 3 days
- **Publicly available?:** Yes
- **Code licenses (if publicly available)?:** MIT
- **Data licenses (if publicly available)?:** MIT
- **Workflow automation framework used?:** CMake, version 3.20.2 or higher
- **Archived (provide DOI)?:** 10.5281/zenodo.13325358

### C. Description

1) *How to access:* Available on Zenodo [here](#).

2) *Hardware dependencies:* To replicate the construction of the *Flag-Proxy Networks (FPNs)*, a laptop with MacOS is needed: such evaluations are expected to take no more than six hours. To replicate *Block Error Rate (BER)* evaluations, a computing cluster will be necessary to handle large codes. Small codes can be evaluated on any laptop.

Note that while FPN generation can be "run" on Linux, we found that there is a bug on Linux that prevents syndrome extraction circuits from containing CNOTs: it is unclear why this bug occurs (culprit is likely in the scheduling code).

3) *Software dependencies:* Our code compiles with `gcc-12` through `gcc-14`. It has not been tested with `clang`. Furthermore, our codebase uses the CMake build tool (version 3.20.2 or higher) to automate compilation.

Constructing FPNs requires installing CPLEX, which is free for academics (see [here](#)). If it is not possible to install CPLEX, we can provide the FPNs for all evaluated codes. On the other hand, evaluating BERs requires MPI to parallelize these evaluations across many cores on large clusters. The MPI version must match the corresponding `gcc` version. When building our code with `gcc-14`, we were able to build our code using `openmpi v5.0.3`. All other dependencies, such as Stim, are packaged with our codebase.

Finally, to execute the experiments, we have provided Python scripts which execute the scripts on the requisite quantum codes and also set the appropriate flags for the program. To run these scripts, we require Python 3.10 or higher. To create the plots used in our paper, we require `matplotlib v3.8.3`, `numpy v1.26.3`, `scipy v1.11.4`, and a method of opening an iPython notebook (i.e. JupyterLab).

4) *Data sets:* We have provided representations of the evaluated quantum codes in the folder `data/tanner`. Hyperbolic surface codes are listed under `hysc` and hyperbolic color codes are listed under `hycc`. Furthermore, both code folders are organized by sub-family.

These codes are constructed using *GAP*, a free and commonly-used computer-algebra system. We do not include the corresponding GAP scripts in the artifact to reproduce constructing the codes, as the code construction is rather tedious. If you require these scripts, please reach out to the corresponding author.

### D. Installation

1) *CPLEX Configuration:* Making FPNs relies on linking IBM's CPLEX solver to some of the built executables. Currently, we have an automated method of finding CPLEX on MacOS devices in the CMake file `cmake/FindCPLEX.cmake`. This method may not work on a Windows or Linux machine, but the corresponding CMake variables, `CPLEX_LIBRARY_DIR` and `CPLEX_INCLUDE_DIR`, may be set manually by modifying their definition in this file, or providing the paths when running CMake (see below). However, we do note that our FPN generation program only works on MacOS. Please reach out to the authors for the FPNs produced in the paper.

If you are using different machines to make FPNs and run memory experiments (as we did since our cluster did not support CPLEX), then on the machine that does not support CPLEX, set the variables `COMPILE_PROTEAN_LIB` and `COMPILE_PROTEAN_MAIN` to `OFF` in the file `cmake/UserConfig.cmake`.

2) *Building Executables:* Run the following commands to build the necessary executables:

```
$ mkdir Release && cd Release
$ cmake .. -DCMAKE_BUILD_TYPE=Release
[-DCPLEX_LIBRARY_DIR=... -DCPLEX_INCLUDE_DIR=...]
$ make -j4
```

### E. Experiment workflow

1) *Figure 10, Table I, Figure 12, Figure 14:* Building the FPNs can be done by running the bash script `scripts/protean/make_all_arch.sh`. This must be done in the base directory (same level as `Release`). This script will call the Python file `scripts/protean/evals.py`, which will build FPNs for all families and sub-families.

These experiments should take at most six hours on a laptop.

2) *Figure 17, Figure 18*: To run the main memory experiments (for the codes in Table III), run the `scripts/protean/compute_ber_5e-4_1e-3.sh` script as follows:

```
$ ./scripts/protean/compute_ber_5e-4_1e-3.sh
hysc/5_5/150_32_6_6 mwpm -mx
$ ./scripts/protean/compute_ber_5e-4_1e-3.sh
hysc/5_5/150_32_6_6 mwpm -mz
```

Repeat the same commands for `hysc/4_5/160_18_8_6` as well. These commands will execute memory experiments for the hyperbolic surface codes in Table III. For the hyperbolic color codes in Table III, run the above commands for `hycc/4_6/216_40_8_8` and `hycc/5_8/360_130_6_6` but also replace `mwpm` with `restriction`.

We have provided our version of the `scripts/protean/compute_ber_5e-4_1e-3.sh`, which is designed to work with our compute cluster. We have commented out lines that correspond to module imports (i.e. for OpenMPI). Note that our version uses `srunc`, which implicitly calls `mpirun`, as our cluster uses SLURM. Hence, if the user's cluster does not use SLURM, it will need to be switched out (i.e. to `mpirun <X>`, where `X` is the number of cores used).

For each code, we used 512 cores, 8GB of memory per core, and a 12 hour wall-time.

3) *Figure 19, Figure 20*: Run the following bash script: `./scripts/protean/eval_decoders.sh <X>`, where `X` is the number of cores used. This can be done on a laptop within a few minutes.

## F. Evaluation and expected results

All results can be found by examining their respective cells in the iPython notebook `scripts/protean/plots.ipynb`. Furthermore, all figures can be found in `scripts/protean/plots` after running the respective cells in the notebook.

Generated plots and data should roughly match what is reported in the main text, with some possible variance due to randomness.

## G. Methodology

Submission, reviewing and badging methodology:

- <https://www.acm.org/publications/policies/artifact-review-and-badging-current>
- <https://cTuning.org/ae>

## REFERENCES

- [1] "Suppressing quantum errors by scaling a surface code logical qubit," *Nature*, vol. 614, no. 7949, pp. 676–681, 2023.
- [2] R. Acharya, L. Aghababaie-Beni, I. Aleiner, T. I. Andersen, M. Anshmann, F. Arute, K. Arya, A. Asfaw, N. Astrakhantsev, J. Atalaya *et al.*, "Quantum error correction below the surface code threshold," *arXiv preprint arXiv:2408.13687*, 2024.
- [3] P. Baireuther, M. D. Caio, B. Criger, C. W. Beenakker, and T. E. O'Brien, "Neural network decoder for topological color codes with circuit level noise," *New Journal of Physics*, vol. 21, no. 1, p. 013003, 2019.
- [4] M. E. Beverland, A. Kubica, and K. M. Svore, "Cost of universality: A comparative study of the overhead of state distillation and code switching with color codes," *PRX Quantum*, vol. 2, no. 2, p. 020341, 2021.
- [5] S. Bravyi, A. W. Cross, J. M. Gambetta, D. Maslov, P. Rall, and T. J. Yoder, "High-threshold and low-overhead fault-tolerant quantum memory," *arXiv preprint arXiv:2308.07915*, 2023.
- [6] N. P. Breuckmann and S. Burton, "Fold-transversal clifford gates for quantum codes," *arXiv preprint arXiv:2202.06647*, 2022.
- [7] N. P. Breuckmann and J. N. Eberhardt, "Balanced product quantum codes," *IEEE Transactions on Information Theory*, vol. 67, no. 10, pp. 6653–6674, 2021.
- [8] N. P. Breuckmann and B. M. Terhal, "Constructions and noise threshold of hyperbolic surface codes," *IEEE transactions on Information Theory*, vol. 62, no. 6, pp. 3731–3744, 2016.
- [9] N. P. Breuckmann, C. Vuillot, E. Campbell, A. Krishna, and B. M. Terhal, "Hyperbolic and semi-hyperbolic surface codes for quantum storage," *Quantum Science and Technology*, vol. 2, no. 3, p. 035007, 2017.
- [10] E. Campbell, A. Khurana, and A. Montanaro, "Applying quantum algorithms to constraint satisfaction problems," *Quantum*, vol. 3, p. 167, jul 2019. [Online]. Available: <https://doi.org/10.22331/qf-2019-07-18-167>
- [11] C. Chamberland, A. Kubica, T. J. Yoder, and G. Zhu, "Triangular color codes on trivalent graphs with flag qubits," *New Journal of Physics*, vol. 22, no. 2, p. 023019, 2020.
- [12] C. Chamberland, G. Zhu, T. J. Yoder, J. B. Hertzberg, and A. W. Cross, "Topological and subsystem codes on low-degree graphs with flag qubits," *Physical Review X*, vol. 10, no. 1, p. 011022, 2020.
- [13] R. Chao and B. W. Reichardt, "Quantum error correction with only two extra qubits," *Physical review letters*, vol. 121, no. 5, p. 050502, 2018.
- [14] A. M. Childs, D. Maslov, Y. Nam, N. J. Ross, and Y. Su, "Toward the first quantum simulation with quantum speedup," *Proceedings of the National Academy of Sciences*, vol. 115, no. 38, pp. 9456–9461, sep 2018. [Online]. Available: <https://doi.org/10.1073/pnas.1801723115>
- [15] J. Conrad, C. Chamberland, N. P. Breuckmann, and B. M. Terhal, "The small stellated dodecahedron code and friends," *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 376, no. 2123, p. 20170323, 2018.
- [16] N. Delfosse, "Tradeoffs for reliable quantum information storage in surface codes and color codes," in *2013 IEEE International Symposium on Information Theory*. IEEE, 2013, pp. 917–921.
- [17] J. Edmonds, "Maximum matching and a polyhedron with 0,1-vertices," *Journal of Research of the National Bureau of Standards Section B Mathematics and Mathematical Physics*, p. 125, 1965.
- [18] "Gap – groups, algorithms, and programming, version 4.13.1," The Gap Group, 2024. [Online]. Available: <https://www.gap-system.org>
- [19] G. P. Gehér, O. Crawford, and E. T. Campbell, "Tangling schedules eases hardware connectivity requirements for quantum error correction," *PRX Quantum*, vol. 5, no. 1, p. 010348, 2024.
- [20] C. Gidney, "Stim: a fast stabilizer circuit simulator," *Quantum*, vol. 5, p. 497, 2021.
- [21] C. Gidney and M. Ekerå, "How to factor 2048 bit RSA integers in 8 hours using 20 million noisy qubits," *Quantum*, vol. 5, p. 433, apr 2021. [Online]. Available: <https://doi.org/10.22331/qf-2021-04-15-433>
- [22] C. Gidney and C. Jones, "New circuits and an open source decoder for the color code," *arXiv preprint arXiv:2312.08813*, 2023.
- [23] C. Gidney, M. Newman, P. Brooks, and C. Jones, "Yoked surface codes," *arXiv preprint arXiv:2312.04522*, 2023.
- [24] A. W. Harrow, A. Hassidim, and S. Lloyd, "Quantum algorithm for linear systems of equations," *Physical Review Letters*, vol. 103, no. 15, oct 2009. [Online]. Available: <https://doi.org/10.1103/PhysRevlett.103.150502>
- [25] O. Higgott, "Pymatching: A python package for decoding quantum codes with minimum-weight perfect matching," *ACM Transactions on Quantum Computing*, vol. 3, no. 3, pp. 1–16, 2022.
- [26] O. Higgott and N. P. Breuckmann, "Constructions and performance of hyperbolic and semi-hyperbolic floquet codes," *arXiv preprint arXiv:2308.03750*, 2023.
- [27] O. Higgott and C. Gidney, "Sparse blossom: correcting a million errors per core second with minimum-weight matching," *arXiv preprint arXiv:2303.15933*, 2023.



- [28] D. Horsman, A. G. Fowler, S. Devitt, and R. Van Meter, "Surface code quantum computing by lattice surgery," *New Journal of Physics*, vol. 14, no. 12, p. 123011, 2012.
- [29] I. D. Kivlichan, C. Gidney, D. W. Berry, N. Wiebe, J. McClean, W. Sun, Z. Jiang, N. Rubin, A. Fowler, A. Aspuru-Guzik, H. Neven, and R. Babbush, "Improved fault-tolerant quantum simulation of condensed-phase correlated electrons via trotterization," *Quantum*, vol. 4, p. 296, jul 2020. [Online]. Available: <https://doi.org/10.22331/q-2020-07-16-296>
- [30] S. Krinner, N. Lacroix, A. Remm, A. Di Paolo, E. Genois, C. Leroux, C. Hellings, S. Lazar, F. Swiadek, J. Herrmann *et al.*, "Realizing repeated quantum error correction in a distance-three surface code," *Nature*, vol. 605, no. 7911, pp. 669–674, 2022.
- [31] A. Kubica and N. Delfosse, "Efficient color code decoders in  $d \geq 2$  dimensions from toric code decoders," *Quantum*, vol. 7, p. 929, 2023.
- [32] A. J. Landahl, J. T. Anderson, and P. R. Rice, "Fault-tolerant quantum computing with color codes," 2011. [Online]. Available: <https://arxiv.org/abs/1108.5738>
- [33] L. Lao and C. G. Almudever, "Fault-tolerant quantum error correction on near-term quantum processors using flag and bridge qubits," *Physical Review A*, vol. 101, no. 3, p. 032333, 2020.
- [34] S.-H. Lee, A. Li, and S. D. Bartlett, "Color code decoder with improved scaling for correcting circuit-level noise," *arXiv preprint arXiv:2404.07482*, 2024.
- [35] G. Li, A. Wu, Y. Shi, A. Javadi-Abhari, Y. Ding, and Y. Xie, "On the co-design of quantum software and hardware," in *Proceedings of the Eight Annual ACM International Conference on Nanoscale Computing and Communication*, 2021, pp. 1–7.
- [36] P.-H. Liou and C.-Y. Lai, "Parallel syndrome extraction with shared flag qubits for calderbank-shor-steane codes of distance three," *Physical Review A*, vol. 107, no. 2, p. 022614, 2023.
- [37] D. Litinski, "A game of surface codes: Large-scale quantum computing with lattice surgery," *Quantum*, vol. 3, p. 128, 2019.
- [38] A. G. Manes and J. Claes, "Distance-preserving stabilizer measurements in hypergraph product codes," *arXiv preprint arXiv:2308.15520*, 2023.
- [39] P. Murali, D. C. McKay, M. Martonosi, and A. Javadi-Abhari, "Software mitigation of crosstalk on noisy intermediate-scale quantum computers," in *ASPLOS*, 2020.
- [40] M. Reiher, N. Wiebe, K. M. Svore, D. Wecker, and M. Troyer, "Elucidating reaction mechanisms on quantum computers," *Proceedings of the National Academy of Sciences*, vol. 114, no. 29, pp. 7555–7560, 2017. [Online]. Available: <https://www.pnas.org/doi/abs/10.1073/pnas.1619152114>
- [41] K. Sahay and B. J. Brown, "Decoder for the triangular color code by matching on a möbius strip," *PRX Quantum*, vol. 3, no. 1, p. 010310, 2022.
- [42] P. W. Shor, "Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer," *SIAM review*, 1999.
- [43] J.-P. Tillich and G. Zémor, "Quantum ldpc codes with positive rate and minimum distance proportional to the square root of the blocklength," *IEEE Transactions on Information Theory*, vol. 60, no. 2, pp. 1193–1202, 2013.
- [44] Y. Tomita and K. M. Svore, "Low-distance surface codes under realistic quantum noise," *Physical Review A*, vol. 90, no. 6, p. 062320, 2014.
- [45] M. A. Tremblay, N. Delfosse, and M. E. Beverland, "Constant-overhead quantum error correction with thin planar connectivity," *Physical Review Letters*, vol. 129, no. 5, p. 050504, 2022.
- [46] A. Wu, G. Li, H. Zhang, G. G. Guerreschi, Y. Ding, and Y. Xie, "A synthesis framework for stitching surface code with superconducting quantum devices," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, 2022, pp. 337–350.
- [47] K. Yin, H. Zhang, Y. Shi, T. Humble, A. Li, and Y. Ding, "Optimal synthesis of stabilizer codes via maxsat," *arXiv preprint arXiv:2308.06428*, 2023.