Fully Parallelized BP Decoding for Quantum LDPC Codes Can Outperform BP-OSD

Ming Wang
Department of Computer Science
North Carolina State University
Raleigh, USA
mwang42@ncsu.edu

Ang Li
Pacific Northwest National Laboratory
Richland, Washington
ang.li@pnnl.gov

Frank Mueller
Department of Computer Science
North Carolina State University
Raleigh, USA
fmuelle@ncsu.edu

***Abstract*—This work presents a hardware-efficient and fully parallelizable decoder for quantum LDPC codes that leverages belief propagation (BP) with a speculative post-processing strategy inspired by classical Chase decoding algorithm. By monitoring bit-level oscillation patterns during BP, our method identifies unreliable bits and generates multiple candidate vectors to selectively flip syndromes. Each modified syndrome is then decoded independently using short-depth BP, a process we refer to as BP-SF (syndrome flip). This design eliminates the need for costly Gaussian elimination used in the current BP-OSD approaches. Our implementation achieves logical error rates comparable to or better than BP-OSD while offering significantly lower latency due to its high degree of parallelism for a variety of bivariate bicycle codes. Evaluation on the $[\![144, 12, 12]\!]$ bivariate bicycle code shows that the proposed decoder reduces average latency to approximately 70% of BP-OSD. When post-processing is parallelized the average latency is reduced by 55% compared to the single process implementation, with the maximum latency reaching as low as 18%. These advantages make it particularly well-suited for real-time and resource-constrained quantum error correction systems.**

I. Introduction

One of the central challenges in quantum computing is achieving fault-tolerant quantum computation (FTQC), which requires the use of quantum error correction (QEC) codes to correct errors arising from noisy quantum devices. Significant efforts [6], [7], [29]–[31] have been made toward designing decoders and hardware architectures for various QEC codes. Among the QEC codes, quantum low-density parity-check (qLDPC) codes have attracted significant attention in recent years due to their potential to encode more logical qubits than surface codes while maintaining a high threshold [2], [18], [27]. However, decoding qLDPC codes remains a major challenge. To ensure reliable operation, especially in fault-tolerant quantum memory and computation, errors must be corrected both quickly and accurately. In practice, decoders must keep pace with the rate of syndrome extraction to prevent data backlog [25], placing stringent demands on both the decoding algorithms and their implementations in terms of performance and computational efficiency.

Belief propagation (BP)-based decoders, widely used in classical LDPC codes, are appealing due to their low complexity, parallelizability, and near-optimal performance [8], [21], [24], [37]. However, their effectiveness diminishes significantly when applied to qLDPC codes. This degradation is primarily due to inherent properties of qLDPC codes, such as degeneracy, the presence of many low-weight stabilizers, and the prevalence of trapping sets, which hinder convergence and reliability of BP decoding [20].

Many works have sought to address the limitations of BP-based decoders on qLDPC codes. In [19], Poulin and Chung proposed several techniques to enhance convergence, including random freezing of variable nodes, perturbing prior information, and colliding unsatisfied check nodes. Similar approaches are also adopted in [35] to freeze a node based on posterior information. In [5], instead of performing static trapping set analysis prior to decoding, Chytas et al. identified oscillation bits affected by trapping sets dynamically during the decoding process. Once identified, the posterior information of these bits is modified, similar to the approach in [19], to help the decoder escape local minima and converge. Raveendran et al. [20] analyzed different types of trapping sets and proposed using a layered BP decoder to mitigate the effect of symmetric trapping sets. While layered decoder can reduce complexity, it often comes at the cost of increased decoding latency as layered decoders are serial. More recently, Yin et al. [36] leveraged the degeneracy property of quantum LDPC codes to enhance BP decoding. By analyzing bit-wise marginal probabilities from BP, they selectively split rows in the parity-check matrix and modify the corresponding Tanner graph. This symmetry-breaking technique helps the decoder avoid convergence stalls caused by structural degeneracies in the code. In general, these methods typically involve modifications to the graph structure, prior, or posterior information in BP decoding to constrain the decoder, which effectively reduce the search space to facilitate convergence. In [11], Gong et al. proposed guided decimation guessing (GDG), a method based on tracking the decoding history of BP to accelerate its convergence. They also employ a tree-search-like strategy that keeps multiple decimation paths open to correct errors as a decoding ensemble. However, the decision tree-like structure of the “guessing” phase restricts the algorithm’s potential for further parallelization. During the preparation of this work, Müller et al. [17] proposed Relay-BP, a hardware-friendly and real-time decoder constructed by chaining multiple Mem-BP [4] decoders with varying memory strengths. While this approach achieves a logical error rate significantly lower than the BP-OSD decoder, its sequential chaining structure imposes a latency overhead that prevents the different decoding stages from being parallelized.

Apart from efforts aimed at improving the BP decoder itself, several works focus on post-processing techniques to enhance decoding performance. A widely used approach is belief propagation combined with ordered statistics decoding (BP-OSD) [18], [23]. While BP-OSD significantly enhances error correction performance, its reliance on a Gaussian elimination step during the OSD phase introduces substantial computational overhead. Specifically, this step incurs a complexity of $O(N^3)$ [18], where N denotes either the code length (in the code-capacity error model) or the number of error mechanisms (in the circuit-level noise model). In contrast, each iteration of BP has a computational complexity of only $O(N)$ [21]. This difference makes BP-OSD computationally expensive and less suitable for large-scale or real-time decoding applications. To address this limitation, recent works such as [13], [34] proposed partitioning the Tanner graph into several clusters. This localized decoding approach reduces the size of the matrices involved in Gaussian elimination, thereby lowering the overall computational burden without substantially compromising decoding performance. These post-processing techniques provide a fallback when BP decoding fails, but they often involve complex data structures and control flows, making them difficult to implement efficiently in hardware.

In this work, rather than focusing on improving BP or OSD individually, we propose that we can achieve error rate performance comparable to BP-OSD while offering significantly lower latency. The key insights behind our approach, which we refer to as BP-SF, are twofold: (i) Most BP decoding failures are caused by a small subset of bits referred to as oscillating bits. By flipping some of these bits in the syndrome domain, BP can often converge rapidly. (ii) Due to the low complexity and inherently parallel nature of BP, we can speculatively perform multiple decoding attempts in parallel, incurring minimal additional latency. Our technique can also be viewed as analogous to providing different starting points to the optimizer, helping it escape local minima and find a global one.

We give a brief visualization in Figure 1 to illustrate our purely BP-based algorithm. We demonstrate that, with appropriate implementation and design, it can closely match the logical error rates achieved by BP-OSD. Through simulations on a variety of quantum LDPC codes, including the $[\![72,12,6]\!]$, $[\![144,12,12]\!]$, and $[\![288,12,18]\!]$ bivariate bicycle codes; the $[\![126,12,10]\!]$ and $[\![154,6,16]\!]$ coprime bivariate bicycle codes; the $[\![225,16,8]\!]$ subsystem hypergraph product simplex code and the $[\![254,28]\!]$ generalized bicycle code, we show that the proposed decoder performs similarly to BP-OSD with a combination-sweep of order 10. Additionally, the proposed BP decoder requires lower latency as it is fully parallelizable, which means it can outperform BP-OSD in execution efficiency, making it a promising candidate for scalable and efficient decoding of quantum LDPC codes in practical fault-tolerant quantum computing systems.

Contributions:

- **BP Decoding Analysis:** We analyze the behavior of the BP decoding algorithm using the $[\![144,12,12]\!]$ code as a case study, revealing a long-tail distribution in the number of iterations and a strong correlation between oscillating bits and actual error locations.
- **Oscillation-Guided Speculative Decoding:** We propose a fully parallelizable post-processing technique that leverages bit-level oscillation during BP decoding to identify unreliable bits and generate test vectors, enabling Chase-like decoding without requiring Gaussian elimination.
- **Efficient Implementation:** We implement the proposed decoder in both serial and multi-process CPU versions, achieving faster runtime than BP-OSD while maintaining comparable logical error rates across various qLDPC codes. We also explore GPU implementation potential, and provide a pessimistic upper bound for GPU decoding time.

II. Background

A. Quantum LDPC Codes

Stabilizer codes are among the most commonly used codes in quantum error correction. One can measure each stabilizer to infer both the type and location of errors in a multi-qubit system. To construct such a code, all stabilizers must commute with each other. Thus, they have a common eigenspace and form a stabilizer group $\mathcal{S}$. The code space $\mathcal{C}$ defined by such group is

$$\mathcal{C} = \{ |\psi\rangle \mid s\,|\psi\rangle = |\psi\rangle\,,\ \forall s \in \mathcal{S} \}. \tag{1}$$

An $[\![n,k,d]\!]$ stabilizer code can be defined by $n-k$ independent stabilizers, allowing us to encode k qubits of logical information into an n-qubit block tolerating up to $\lfloor (d-1)/2 \rfloor$ errors. CSS codes are an important class of stabilizer with two sets of stabilizers, X-type and Z-type, represented by parity-check matrices H_X and H_Z, respectively. Each row in a parity-check matrix corresponds to a stabilizer generator, and each column corresponds to a physical qubit. A "1" entry indicates an X or Z operator (depending on whether it is in H_X or H_Z), while a "0" indicates the identity. Consequently, an X-type stabilizer acts as X or the identity on each qubit, and a Z-type stabilizer acts as Z or the identity on each qubit. Errors can therefore be corrected by handling Z errors and X errors separately. Since all stabilizers must commute with each other, it follows directly that for a CSS code $H_X H_Z^T = 0$. If both H_X and H_Z are sparse matrices, the code is a CSS-type qLDPC code. The sparsity of these matrices offers a key advantage: syndrome extraction can be performed using fewer quantum gates, thereby reducing circuit depth and potential error accumulation. In this sense, the surface code can be regarded as a special case of qLDPC codes, characterized by strictly local, nearest-neighbor interactions on a 2D lattice. More general qLDPC codes, in contrast, typically exhibit higher connectivity and non-local parity-check relationships.

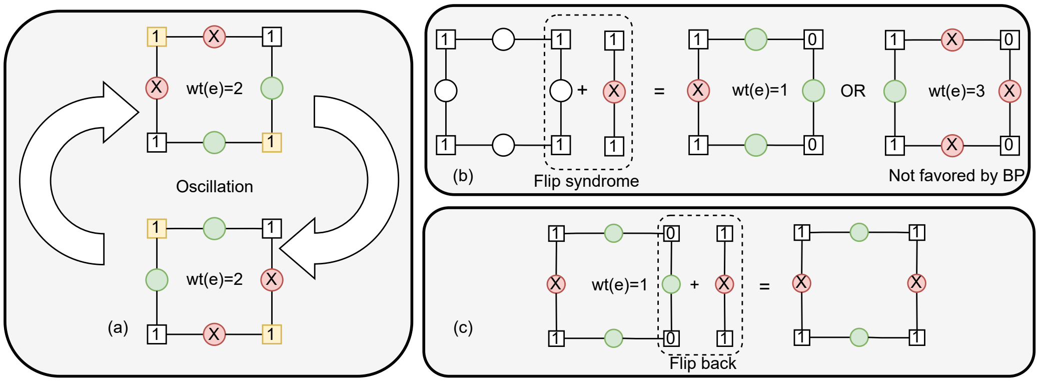


Fig. 1: (a) Example of BP failing to converge due to oscillations. Red Xs denote bits identified as erroneous by BP, and yellow squares represent unsatisfied syndrome checks. All four bits are oscillating. (b) One oscillating bit (e.g., the rightmost one) is selected, and its neighboring syndromes are flipped. BP then converges since the two competing error patterns now have different weights. (c) After convergence, the selected bit is flipped back to restore consistency with the original input syndrome.

This allows them to encode more logical qubits while maintaining a large code distance, but it also makes matching-based algorithms (commonly used for surface codes) less effective, since the increased connectivity introduces hyperedges into the decoding graph.

B. Decoding Problem

Assuming the noise is uniform on each bit, the optimal syndrome decoding problem for classical codes can be formalized as follows: Given a code with parity-check matrix $H \in \mathbb{F}_2^{M \times N}$ and syndrome $\boldsymbol{s} \in \mathbb{F}_2^M$, we want to find an error $\hat{\boldsymbol{e}} \in \mathbb{F}_2^N$ that satisfies the syndrome

$$\hat{\boldsymbol{e}} = \arg \min_{\hat{\boldsymbol{e}} H^T = \boldsymbol{s}} (\sum_{i}^{N} \hat{\boldsymbol{e}}_i). \tag{2}$$

Quantum stabilizer codes face a problem due to the phenomenon of *degeneracy*, where multiple errors have the same effect on the code space. As a result, the goal of decoding is not to identify the most likely error itself, but rather the most likely equivalence class of errors modulo stabilizers, since errors that differ by a stabilizer operation act identically on the code space. Given a syndrome, $\boldsymbol{s}$, the optimal decoding problem considering degeneracy becomes

$$[\hat{E}] = \arg \max_{[E]:\text{synd}(E)=\boldsymbol{s}} \Pr([E]), \tag{3}$$

where $[E]$ denotes the equivalence class of errors under the stabilizer group and $\Pr([E])$ is the total probability of all errors in that class. While the classical decoding problem in Eq. (2) is NP-hard [1], its quantum counterpart in Eq. (3) is even more complex, being #P-complete [14]. In practice, optimal decoding is computationally intractable, so efficient decoders typically aim to approximate the solution to Eq. (2) with good performance under realistic noise models.

As CSS codes can be decoded separately on the X- and Z-error bases, each decoding problem can be treated as a classical decoding task and addressed using classical decoding algorithms. For example, BP, the most commonly used decoder for classical LDPC codes, is also widely applied in qLDPC codes. This is because the sparsity of qLDPC and LDPC codes can be effectively exploited by BP-based decoders, which rely on the assumption of independent probability updates. But this assumption is only valid when the parity-check matrix is sparse. As the matrix becomes denser, the variable nodes exhibit stronger correlations, violating the independence assumption. Consequently, although the BP decoder can, in principle, be applied to denser codes, its empirical performance tends to degrade due to the increased dependency within the graph structure. Given an $M \times N$ parity-check matrix H, let $v_1, \ldots, v_N$ denote the variable nodes (corresponding to the columns of H) and $c_1, \ldots, c_M$ the check nodes (corresponding to the rows). The normalized min-sum algorithm, a widely used variant of BP can be described as follows:

1) **Initialization:** Given the prior error information, $\boldsymbol{p}$, of each variable node, v_i, the channel LLR is initialized to

$$l_{v_i}^{ch} = \log \frac{1 - p_{v_i}}{p_{v_i}}. \tag{4}$$

2) **Variable-to-Check (V2C) Message Update:** Each variable node, v_i, sends a message to its neighboring check node, c_j, based on the channel LLR and the incoming messages from all other neighboring check nodes, denoted as

$$l_{v_i \to c_j} = l_{v_i}^{ch} + \sum_{c_{j'} \in N(v_i) \backslash \{c_j\}} l_{c_{j'} \to v_i} \ , \tag{5}$$

where $l_{c_{j'} \to v_i}$ is set to 0 for the first iteration, "\" is set minus, and $N(v_i)$ is the set of all the check nodes connected with v_i.

3) **Check nodes to variable nodes (C2V) update:** Each check node, c_j, updates its message to a neighboring variable node, v_i, using the min-sum rule denoted as

$$\begin{aligned} l_{c_j \to v_i} =& (-1)^{s_j} \cdot \alpha \min_{v_{i'} \in N(c_j) \backslash \{v_i\}} |l_{v_{i'} \to c_j}| \cdot \\ & \prod_{v_{i'} \in N(c_j) \backslash \{v_i\}} \text{sign}(l_{v_{i'} \to c_j}), \end{aligned} \tag{6}$$

where α is the damping factor used to attenuate the c2v message.

4) **Hard decision**: After a fixed number of iterations or upon convergence, the final marginal LLR for each variable node is computed as

$$l_{v_i}^{out} = l_{v_i}^{ch} + \sum_{c_{j'} \in N(v_i)} l_{c_{j'} \to v_i} \ , \tag{7}$$

where the $\boldsymbol{l}^{out}$ is the marginalized LLR for each variable node. The estimated error is then obtained via hard decision as

$$\hat{e}_i = \begin{cases} 0 & \text{if} \quad l_{v_i}^{out} > 0 \\ 1 & \text{otherwise} \end{cases} , \tag{8}$$

In each iteration, steps 2, 3 and 4 are performed. The BP decoding algorithm proceeds until $\boldsymbol{e}H^T = \boldsymbol{s}$ is satisfied or the maximum number of iterations is reached.

III. BP Behavior Analysis: A Case Study

In this section, we analyze the behavior of BP-based decoders on qLDPC codes using the $[\![144, 12, 12]\!]$ "gross" code from [2] as a representative case study. While specific to this code, the analysis provides general insights into the behavior of BP decoding on qLDPC codes and offers guidance for improving decoder performance. The BP decoder used below is a min-sum decoder as described in Eq. (6) with an adaptive damping factor of $\alpha = 1 - 2^i$, where i is the current number of iterations.

A. Number of Iterations

Figure 2 shows the ratio of syndromes failed to converge under the circuit-level noise model, where p denotes the physical error rate. The curves are obtained by simulating 10,000 samples for each of $p = 0.001$ and $p = 0.002$, both representative values below the threshold. For each sample, we record the number of iterations required for convergence and compute the cumulative distribution. This non-convergence rate at iteration i is defined as the fraction of samples that have not converged within i iterations—that is, 1 minus the cumulative convergence rate.

As Fig. 2 indicates, in most cases, BP converges within a small number of iterations. For instance, at $p = 0.001$, the average number of iterations is merely 8.9 despite setting maximum number of iterations to 1,000. Even at higher error rates, such as $p = 0.002$, the average number of iterations remains low, although the tail becomes longer. Notably, cases that do not converge within the early iterations rarely benefit from increasing the iteration count further. This observation motivates an alternative strategy: Rather than extending the number of BP iterations, we can vary the inputs to the BP decoder while keeping the maximum number of iterations small. If these varied inputs have better independent chances of successful decoding, then running multiple instances in parallel allows us to exponentially suppress the logical error rate without incurring significant decoding latency.

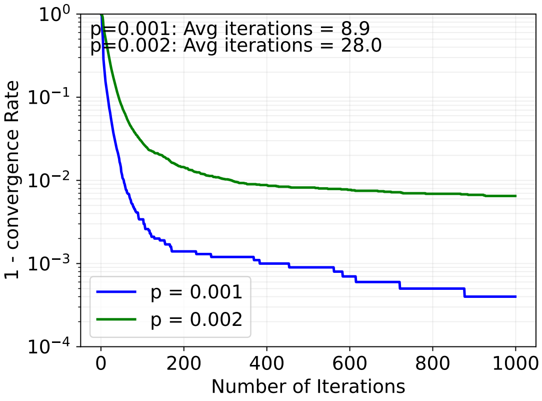


Fig. 2: Ratio of unsuccessful BP decoding (1−convergence rate) on the $[\![144, 12, 12]\!]$ code under the circuit-level noise model. The maximum number of decoder iterations is set to 1,000 and number of samples is 10,000.

B. Oscillation

As suggested in previous works [5], [20], trapping sets and code degeneracy often result in ambiguous BP decoding, leading to convergence failures. A common symptom of such a failure is bit-level oscillation, where certain output bits repeatedly flip between 0 and 1 across iterations. To better understand the relationship between oscillating bits and decoding errors, we analyze the dynamics of bit oscillations during the BP process. During the decoding process, we track bit-level oscillations by comparing the output of each iteration with that of the previous one and counting how often each bit flips, which is similar to [5]. We then identify a set of oscillating bits, denoted by Φ, based on their flip frequency. Specifically, Φ is defined as the top $|\Phi|$ of the most frequently flipped bits. We denote $\mathrm{supp}(\boldsymbol{e})$ as the set of erroneous bits and define the hit precision and recall as

$$\text{Precision} = \frac{|\,\mathrm{supp}(\boldsymbol{e}) \cap \Phi|}{|\Phi|}, \tag{9}$$

$$\text{Recall} = \frac{|\,\mathrm{supp}(\boldsymbol{e}) \cap \Phi|}{|\,\mathrm{supp}(\boldsymbol{e})|}. \tag{10}$$

Fig. 3 shows the precision and recall rate when the min-sum decoder fails to decode a syndrome. We can see that even when the BP decoder fails to fully correct an error, the pattern of bit oscillations essentially reveals a meaningful subset of the actual error locations. In particular, at lower physical error rates, the set of oscillating bits nearly covers the entire true error positions. To confirm, we observe that the hit precision, i.e., the fraction of oscillating bits that are indeed erroneous, is substantially higher than the physical error rate. This suggests that bits in the oscillation set Φ are much more likely to be true errors than random guesses, making them valuable targets for post-processing. As the physical error rate increases, the recall decreases, primarily because the total number of errors grows while the candidate set size remains fixed.

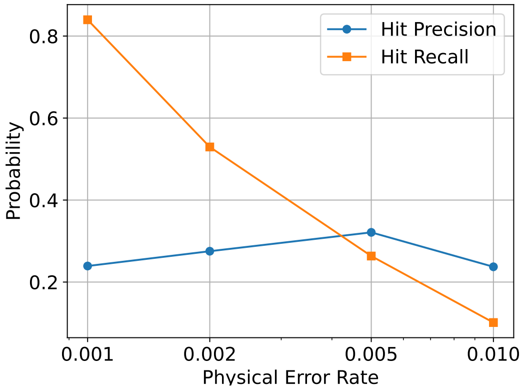


Fig. 3: Precision and recall probabilities of candidate bit selection on the $[\![144, 12, 12]\!]$ code. We evaluate the correlation between candidate bits and actual error locations by identifying the top 50 most frequently flipped bits among approximately 8,000 error mechanisms. The decoder is run with a maximum of 50 iterations, and statistics are collected over 1,000 decoding failures. This analysis reveals how well bit-level oscillation can serve as a heuristic for error localization.

IV. A New Speculative Decoding Method

To enhance the performance of BP decoding, we adopt a Chase-like post-processing technique [3] that we call BP-SF (Syndrome Flip). This approach generates a set of trial vectors by flipping candidate bits and attempts decoding on each of them, thereby increasing the likelihood of successful error correction. In the quantum decoding setting, we only have access to the syndrome and prior information of error mechanisms, rather than the prior information of each bit based on signal strength in classical soft decoding. Therefore, unlike the original Chase algorithm, which relies on prior channel information to select candidate bits, our BP-SF method identifies candidate bits based on BP flipping statistics. As shown in Figure 4, once these candidate bits Φ are identified, we generate diverse decoding attempts by flipping the input syndrome accordingly across multiple BP instances. This strategy increases the variety in the decoder's inputs and distinguishes our BP-SF approach from that in [15], which modifies the posterior information instead of the syndrome. If the decoder successfully converges on this modified input, we then flip these bits back in the output. This restoration ensures that the final output error matches the original syndrome and preserves the validity of the decoding result. Since these candidate bits are likely to correspond to actual error locations, flipping them can also effectively reduce the number of errors in the input and equivalently lowers the physical error rate. This reduction not only increases the likelihood of successful decoding but also reduces the number of iterations needed for BP to converge. The detailed pseudo code can be found in Algorithm 1.

We note that this concept is also widely adopted in classical decoding frameworks [3], [9], [12], [32]. In such approaches, multiple modified inputs are generated and decoded independently, and the best result is selected according to the maximum likelihood criterion, typically based on minimum weight or log-likelihood score. Since each decoding attempt is independent, they can be executed in parallel, introducing minimal latency overhead. However, in the quantum setting, the structure of qLDPC codes introduces unique advantages. Due to the degeneracy and high distance of qLDPC codes, and the tendency of BP decoders to favor minimum-weight solutions, successful BP convergence rarely results in a logical error. This is partly because the classical codes $\mathcal{C}_X$ and $\mathcal{C}_Z$, defined by the parity-check matrices H_X and H_Z, have low minimum distances (recall that the minimum distance of $\mathcal{C}_X$ is upper bounded by the row weight of H_Z, and vice versa). As a result, there are many low-weight codewords in $\mathcal{C}_X$ and $\mathcal{C}_Z$. Therefore, even if BP converges to a non-optimal codeword within $\mathcal{C}_X$ or $\mathcal{C}_Z$, the resulting error is likely to differ from the optimal solution by a low-weight codeword. Consequently, the probability that this low-weight codeword forms a logical operator is very low, as the weight of logical operator is much higher (at least d). This motivates our use of a speculative decoding strategy. In our BP-SF approach, we omit the maximum likelihood selection step: Because of code degeneracy, any solution that satisfies the syndrome is likely to belong to the correct coset, particularly in the low-error regime and for high-distance codes. As a result, we simply return the first valid codeword that satisfies the syndrome among the parallel decoding attempts. This strategy reduces latency while preserving decoding performance.

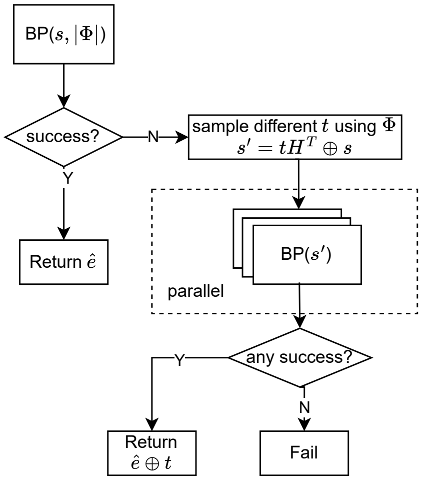


Fig. 4: A simplified flowchart of the proposed decoder. The full procedure is described in Algorithm 1.

V. Simulation Results

In this section, we evaluate the performance of our proposed BP-SF decoder and compare it against the BP-OSD decoder. All BP decoders use the min-sum algorithm with an adaptive damping factor $\alpha = 1 - 2^i$, where i is the current number

Algorithm 1: BP decoding with our Chase-like post-processing (BP-SF)

```
Input: Syndrome $\boldsymbol{s}$, max flip weight $w_{\max}$, number of
       uncertain bits $|\Phi|$
Result: Estimated error $\hat{\boldsymbol{e}}$
Function Main($\boldsymbol{s}, w_{\max}, |\Phi|$):
    /* Initial BP attempt with
       oscillation tracking                          */
    $s_{\text{ucc}}, \hat{\boldsymbol{e}}, \Phi \leftarrow$ BP_with_oscillation($\boldsymbol{s}, |\Phi|$)
    if $s_{ucc}$ then
        return $\hat{\boldsymbol{e}}$ /* Syndrome decoded
            successfully                             */
    else
        /* Speculative decoding using
            trial vectors based on $\Phi$           */
        parallel for $\boldsymbol{t} \in$ combinations($\Phi, w_{\max}$)
            $\boldsymbol{s}' = \boldsymbol{s} \oplus \boldsymbol{t}H^T$ /* Flip selected
                bits in syndrome domain              */
            $s_{\text{ucc}}, \hat{\boldsymbol{e}} \leftarrow$ BP($\boldsymbol{s}'$)
            if $s_{ucc}$ then
                return $\hat{\boldsymbol{e}} \oplus \boldsymbol{t}$ /* Undo flipped
                    bits in output                   */
        end
        return Decoding failure
    end
Function BP_with_oscillation($\boldsymbol{s}, |\Phi|$):
    flip_count $\leftarrow \mathbf{0}$
    $\hat{\boldsymbol{e}}_{\text{prev}} \leftarrow \mathbf{0}$
    for $i = 1$ to $i_{\max}$ do
        $\hat{e} \leftarrow$ BP_Update() /* Standard BP
            update                                   */
        flip_count $\leftarrow$ flip_count $+ (\hat{\boldsymbol{e}} \oplus \hat{\boldsymbol{e}}_{\text{prev}})$
            /* Track bit oscillations                */
        $\hat{\boldsymbol{e}}_{\text{prev}} \leftarrow \hat{\boldsymbol{e}}$
        if $\hat{\boldsymbol{e}}H^T = \boldsymbol{s}$ then
            return True, $\hat{\boldsymbol{e}}, \emptyset$
    end
    /* Select top $|\Phi|$ most frequently
       flipped bits                                  */
    $\Phi \leftarrow$ top(flip_count, $|\Phi|$)
    return False, $\hat{\boldsymbol{e}}, \Phi$
```

of iterations. The OSD method is OSD-CS in [23], and for briefness, we use labels such as "BP1000-OSD10" to denote a decoder using BP with a maximum of 1,000 iterations followed by OSD-CS of order 10. Each data point is obtained by collecting at least 100 logical errors unless specified otherwise. The statistical uncertainty is sufficiently small, and error bars are omitted for clarity.

A. Code Capacity Model

In the code capacity error model, X, Y, and Z errors are applied independently for each data qubit with probability of $p/3$. All other operations are perfect, where p is the physical error rate. In this model, we decode all codes using all trial vectors of weight only up to 1, which proves sufficient to achieve satisfactory logical error rates. In many of the codes we tested, the BP decoder already performs well under the code capacity model, leaving limited room for improvement as shown in Appendix B. However, there exist some exceptions. One such case is the $[\![154, 6, 16]\!]$ coprime BB code introduced in [33], where the min-sum decoder performs poorly despite the code's high distance.

Fig. 5 shows the logical error rates of different decoders on the $[\![154, 6, 16]\!]$ coprime-BB code under the code capacity noise model. For our BP-SF decoder, the candidate set size is set to $|\Phi| = 8$, resulting in a maximum of $50 \times (8 + 1) = 450$ BP iterations per decoding attempt. Considering that BP decoders can be run in parallel after the initial run, the latency can be optimized to 100 BP iterations. As shown, our BP-SF decoder significantly outperforms both the baseline BP and BP-OSD decoders, achieving lower logical error rates with fewer iterations and without requiring costly OSD post-processing. Additionally, both BP and BP-OSD exhibit an error floor at low physical error rates. Upon examining the decoding failures, we find that many are due to low-weight (e.g., weight-3) errors that fall into trapping sets. Our BP-SF algorithm circumvents this issue by decoding flipped syndromes, which effectively reduces the number of errors in decoding attempts. Another example is the $[\![288, 12, 18]\!]$ BB code from [2]. As shown in Fig. 6, our BP-SF decoder performs similar to the BP-OSD decoder while using fewer than $50 \times (20 + 1) = 1050$ iterations per decoding attempt and the latency is still 100 iterations considering full parallelization, as we analyzed above.

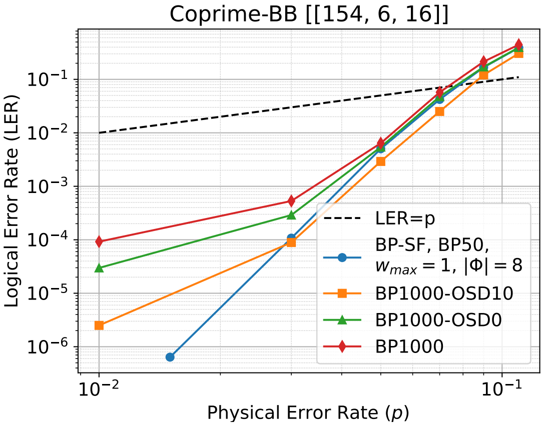


Fig. 5: Error rates of the $[\![154, 6, 16]\!]$ coprime-BB code under the code capacity model.

B. Circuit-Level Noise Model

Under the circuit-level noise model, errors are injected uniformly across gates and measurements. They can propagate through the circuit as it runs. We use Stim [10] to generate the syndrome extraction circuit, the detector error model, and the corresponding parity-check matrix. In this matrix, each row represents a detector event, and each column corresponds to

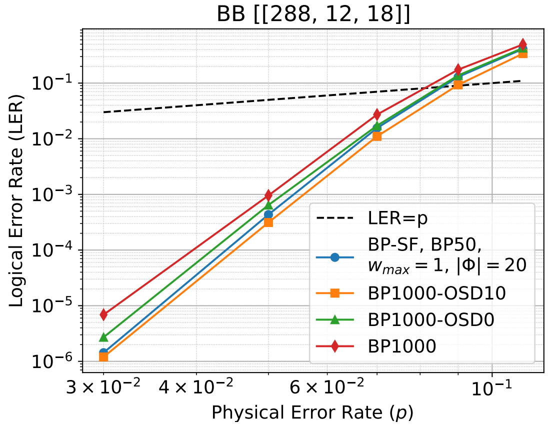


Fig. 6: Error rates of the $[[288, 12, 18]]$ BB code under the code capacity model.

a specific error mechanism. Following the convention in prior literature, we perform d rounds of syndrome extraction and define the logical error rate per round as

$$\text{LER Per Round} = 1 - (1 - \text{LER})^{\frac{1}{d}}, \tag{11}$$

where LER is the logical error rate after d rounds.

In this model, the number of error mechanisms is typically much larger than the number of qubits, resulting in very large parity-check matrices. Consequently, flipping a single bit as in the code capacity model is often insufficient for the BP decoder to converge. On the other hand, exhaustively decoding all trial vectors with weight up to a threshold is computationally expensive. To address this, we adopt a sampling-based approach. Given a maximum trial vector weight w_{max}, we randomly sample n_s trial vectors for each weight in $\{1, \ldots, w_{max}\}$, resulting in a total of $n_s \times w_{max}$ trial vectors per failed BP decoding attempt.

Figure 7 shows the logical error rates of different decoders on the $[[144, 12, 12]]$ BB code under the circuit-level noise model. In this setting, the number of error mechanisms is approximately 8,000, which is significantly larger than the code length used in the code capacity model. Therefore, we expand the candidate set size in our BP-SF decoder. Specifically, our decoder uses up to 3,100 BP iterations (based on $w_{max} = 6, n_s = 5$) and achieves logical error rates that are slightly higher but still comparable to that of the BP-OSD decoder, which uses a maximum of 1,000 iterations and OSD order 10.

Fig. 8 shows the logical error rates (LER) of different decoders on the $[[288, 12, 18]]$ BB code under the circuit-level noise model. The proposed BP-SF decoder exhibits a slightly higher LER than the BP1000–OSD10 decoder. We note that all decoders for this code employ the layered BP variant, as the regular BP shows significantly worse LER performance. This behavior is likely caused by symmetric trapping sets, as suggested in [20], which can lead to large variations in LER depending on the BP scheduling strategy.

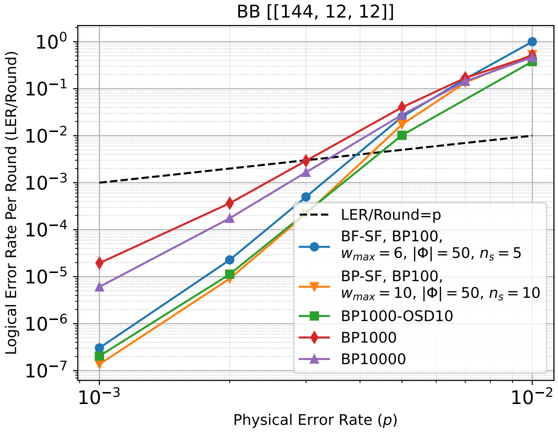


Fig. 7: Error rates of the $[[144, 12, 12]]$ BB code under the circuit-level noise model.

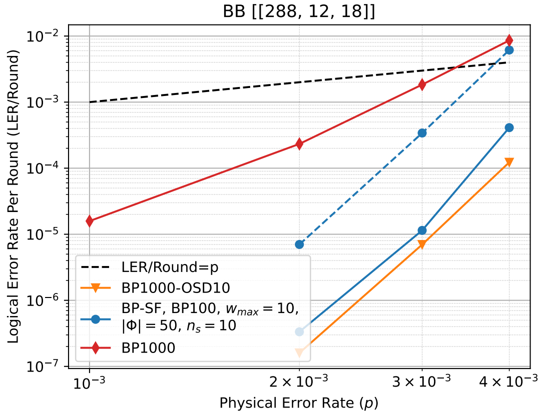


Fig. 8: Error rates of the $[[288, 12, 18]]$ BB code under the circuit-level noise model. Layered BP decoding is used for all decoders except the one shown with a dashed line, which is BP-SF using flooding BP.

Fig. 9 shows the logical error rates of different decoders on the $[[154, 6, 16]]$ BB code under the circuit-level noise model. Our BP-SF decoder uses up to 6,000 iterations (based on $w_{max} = 6, n_s = 10$. However, since all trial syndromes are decoded in parallel, the effective decoding latency corresponds to only 200 iterations if the attempts are run in parallel. Our BP-SF decoder achieves logical error rates that are slightly higher but still comparable to that of the BP-OSD decoder at lower physical error rate, which uses a maximum of 1,000 iterations and OSD order 10. In the higher physical error rate regime, our BP-SF decoder exhibits logical error rates that are higher than those of BP-OSD but still consistently lower than baseline BP decoding.

Fig. 10 shows the logical error rates of different decoders on the $[[126, 12, 10]]$ coprime-BB code under the circuit-level noise model. Our BP-SF decoder uses up to approximately 3,000 BP iterations to achieve a logical error rate comparable

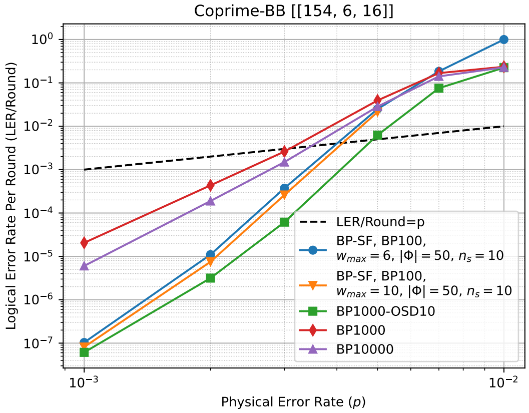


Fig. 9: Error rates of the $[\![154, 6, 16]\!]$ coprime-BB code under the circuit-level noise model.

to that of the BP1000-OSD10 decoder. By increasing both n_s and $w_{\max}$, we are able to further reduce the logical error rate to slightly below that of the BP-OSD decoder. However, this improvement comes at the cost of increased complexity, requiring up to 10,000 BP iterations.

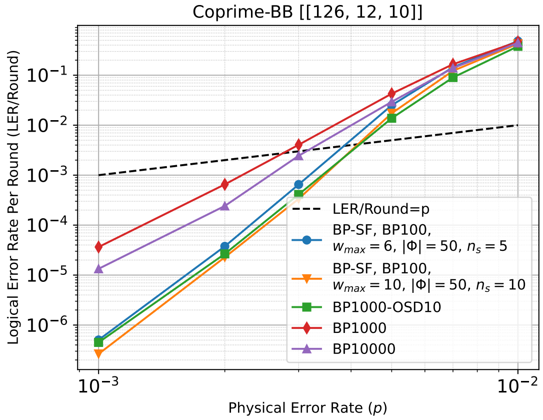


Fig. 10: Error rates of the $[\![126, 12, 10]\!]$ coprime-BB code under the circuit-level noise model.

To demonstrate that our decoder generalizes across different classes of qLDPC codes, we evaluated BP-SF on the $[\![225, 16, 8]\!]$ subsystem hypergraph product simplex (SHYPS) code [16], as shown in Fig. 11. The proposed BP-SF achieves nearly an identical logical error rate (LER) performance to the BP1000–OSD10 decoder, using $w_{\max}$ and only $n_s = 5$, i.e., with fewer parallel trials than required for other codes.

C. Complexity and Parameter Selection

Next, we analyze the computational complexity of our BP-SF decoder. To ensure a fair comparison with baseline methods, we measure the average number of BP iterations under a serial execution model. Specifically, when the initial BP decoding fails, each trial syndrome is decoded sequentially, and the total number of iterations is defined as the cumulative number of BP iterations required until the first successful decoding. This approach provides a conservative estimate of decoding cost, as it does not account for the inherent parallelism of our method, but allows for a meaningful comparison with standard decoders.

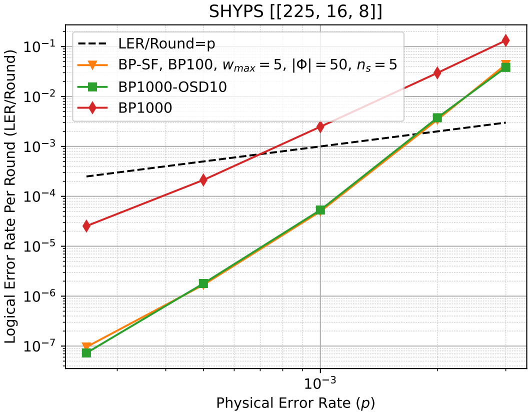


Fig. 11: Error rates of the $[\![225, 16, 8]\!]$ SHYPS code under the circuit-level noise model.

Fig. 12 shows the growth in decoding complexity for the $[\![144, 12, 12]\!]$ code as we target progressively lower logical error rates. For the BP decoder, we vary the maximum number of iterations to control decoding complexity. For our BP-SF decoder, we fix the maximum number of iterations per BP instance to 100 and vary n_s (the number of trial vectors sampled per weight), while keeping $w_{\max}$ constant. This allows us to explore the trade-off between complexity and performance. The physical error rate is fixed at 3×10^{-3} under the circuit-level noise model. All data points are collected by simulating 10,000 shots for logical error rates above 10^{-3} and 100,000 shots for those below.

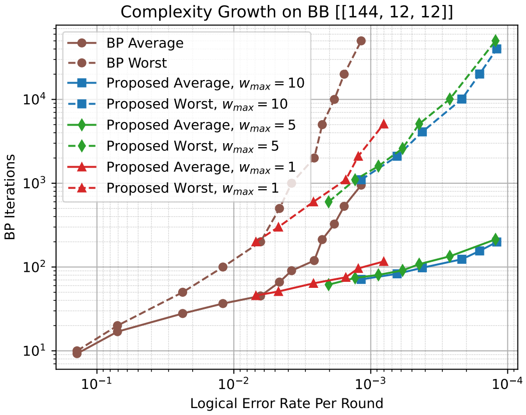


Fig. 12: Complexity growth of different decoders. The parameter $|\Phi|$ is set to 50 for all BP-SF decoders. The number of iterations is calculated assuming serial execution.

Across all decoders, we observe a linear region in which the number of BP iterations increases approximately linearly as the logical error rate decreases, up to a point where each curve drops off sharply, forming what we refer to as a cliff. This cliff marks a regime where the decoder can no longer reliably suppress logical errors within reasonable iteration limits. Our BP-SF decoder consistently postpones this cliff compared to baseline BP, maintaining a lower iteration count at comparable logical error rates. Moreover, increasing w_{max} increases the complexity but extends the linear region further and delays the start of the cliff, providing a tunable trade-off between decoding complexity and error suppression.

Fig. 13 shows how the decoding latency scales as the code size increases. Here, "number of error mechanisms" refers to the total number of error sources in a memory experiment circuit, which grows much faster than the number of qubits. The proposed BP-SF consistently achieves lower average decoding latency. Since most errors are corrected by the initial BP stage, the asymptotic advantage of BP-SF is not evident from the average decoding time alone, which is about $0.63\times$ that of BP-OSD for the $[\![288, 12, 18]\!]$ code. However, when considering only cases where the initial BP fails, BP-SF requires merely $0.1\times$ the latency of BP-OSD, an order of magnitude improvement. Both the overall average latency and the post-processing latency of BP-SF can be further reduced (discussed in Section VI).

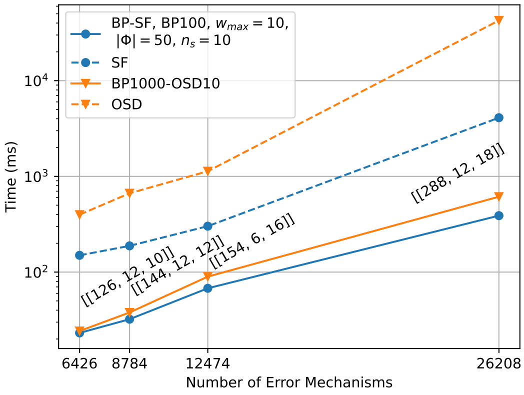


Fig. 13: Latency scaling of BP-SF and BP-OSD at a physical error rate of 3×10^{-3}. Dashed lines indicate the average latency of the post-processing stage, measured only for cases where the initial BP decoding fails.

VI. Parallel Implementation and Performance

In this section, we describe the implementation of our proposed algorithm and compare it with the BP-OSD decoder in the LDPC library [22] and Nvidia's CUDA-Q library [26]. For a fair comparison, we implemented three versions of our algorithm:

Serial CPU version: We modify the BP decoder from [22], which is also a serial CPU version implemented in C++ and wrapped with Cython, to track bit-level oscillations. If the first decoding attempt fails, trial vectors and corresponding syndromes will be prepared in Python, and each trial syndrome will be decoded using the C++ BP decoder. We return the first successful decoding result and skip the rest.

Parallel CPU version (P=# of worker processes): This version maintains a persistent worker process pool along with input and output queues to handle the remaining $n_s \times w_{\max}$ decoding trials after the initial BP attempt fails. The manager process selects candidate positions, generates trial vectors, and computes the corresponding trial syndromes. These trial syndromes are then split into small batches and placed in the input queue. Worker processes continuously retrieve batches from the queue and attempt to decode them until one finds a valid solution, which it places in the output queue. The main process monitors the output queue and, once a valid result is found, signals all workers to stop. The workers then wait for the next input from input queue, and the main process returns the successful result. To avoid accepting stale results, each syndrome is tagged with a serial number. The main process compares the serial number of each fetched result with that of the currently processed syndrome to ensure correctness.

Estimated GPU version (GPU_Est): Since the CUDA-Q library does not provide access to oscillation statistics during decoding, we implemented this version to estimate the decoding speed. We first precompute the syndromes and their corresponding oscillation bits using a CPU-based BP decoder. These syndromes are then decoded using CUDA-Q. If decoding fails, the precomputed oscillation bits are used in Python to generate trial syndromes, which are then decoded one-by-one using CUDA-Q until a successful decoding is found or the maximum number of trials is reached. Due to numerical instability and implementation differences, the CUDA-Q decoder may report a different success status than the CPU BP decoder. In such cases, the pre-computation is invalid and we discard those results. This approach results in a pessimistic estimate of GPU performance, as trial syndrome generation is performed on the CPU after initial failure, and transferring them to the GPU introduces additional memory copy overhead. A more efficient strategy would submit all trial syndromes to the GPU decoder simultaneously and return upon the first success. However, the current CUDA-Q library only supports the `decode_batch` method, which waits for all syndromes in the batch to complete, effectively blocking on the slowest one.

For the sake of fairness, we select the number of BP iterations to make BP-OSD as fast as possible. At first glance, one might expect that reducing the number of BP iterations would lower the overall decoding latency. However, beyond some point, reducing the number of BP iterations not only degrades the logical error rate but can also *increase* the total latency as shown in Table I. This is caused by the BP stage being relatively inexpensive compared to the OSD stage, where higher number of BP iterations allows BP to correct more errors reduces the frequency with which the costly OSD procedure must be invoked. We listed the BP iterations of experiments with the $[\![144, 12, 12]\!]$ code under circuit-level noise of $p = 3 \times 10^{-3}$.

Decoder	LER/d @ 3e-3	Avg Time @ 3e-3
BP100-OSD10	2.89 $\times 10^{-4}$	56.13 ms
BP400-OSD10	2.23 $\times 10^{-4}$	37.69 ms
BP1000-OSD10	2.11 $\times 10^{-4}$	36.44 ms
BP2000-OSD10	2.00 $\times 10^{-4}$	44.01 ms
BP10000-OSD10	1.84 $\times 10^{-4}$	94.94 ms

TABLE I: Logical error rate per round and average decoding time for BP-OSD with different iterations.

We note that for CPU-based decoders, it is possible to parallelize decoding using OpenMP within each BP iteration or by decoding a batch of input syndromes in parallel. However, we did not pursue these options because the BP/BP-OSD decoder provided by the LDPC library [22] does not currently support OpenMP acceleration. For fairness and consistency, we instead implement parallelization at a coarser granularity, i.e., by decoding trial syndromes in parallel. This approach may be less efficient than parallelizing BP iterations directly, as it only accelerates cases where the initial decoding attempt fails. However, it helps mitigate the long-tail latency caused by the large number of trials required in such cases. We also do not parallelize across input syndromes, as decoding them sequentially is more aligned with real-world use cases, where syndrome extraction is performed sequentially and syndromes arrive in a streaming fashion.

Benchmarks were conducted on an NVIDIA Tesla V100-SXM2-16GB GPU and an Intel Xeon E5-2698 v4 @ 2.20GHz CPU. For each physical error rate, 20,000 syndromes were tested. Our newly proposed decoder with 100 BP iterations, $w_{\max} = 10$, and $n_s = 10$ achieves nearly identical logical error rates to BP1000-OSD10 ($1,000$ BP and OSD-CS of order 10) below threshold for the $[\![144, 12, 12]\!]$ code, which allows us to adopt these settings for all evaluations in this section.

Figure 14 shows the average decoding time per syndrome under varying physical error rates. The BP-OSD implementations for CPU and GPU from the LDPC python library and NVIDIA CUDA-Q. The GPU versions consistently achieve the lowest average runtime due to their high degree of parallelism. At low physical error rates (e.g., 0.001), the BP-SF decoders and BP1000-OSD10 exhibit similar performance, as most syndromes are successfully decoded during the initial BP attempt without invoking post-processing. However, as the error rate increases, the BP-SF decoder outperforms BP1000-OSD10 thanks to its more efficient post-processing strategy. The parallel CPU version (CPU, P=8) achieves about 1.8× speedup over the serial version and approaches the performance of BP100, which is included as a lower bound since it performs no post-processing. The GPU runtime of decoders increases only slightly with the error rate, due to the ability to perform computations in parallel. We can also see that the BP1000-OSD10 from CUDA-Q is slightly slower than our GPU_Sim on the same platform.

Figure 15 shows the distribution of decoding times for different decoders at a physical error rate of 0.003. All versions of our BP-SF decoder exhibit lower average decoding times compared to BP1000-OSD10 (Avg: 38.61 ms). Notably, the BP1000-OSD10 curve shows a distinct gap, corresponding to cases where OSD post-processing is triggered. In contrast, the serial CPU version of our algorithm displays a long-tailed but more compact distribution, due to most syndromes that fail the initial decoding being resolved within a few additional trials, with only a small number of outliers requiring longer decoding times. As the number of parallel processes increases, this long tail becomes increasingly compressed, resulting in even lower average decoding times, e.g., 21.00 ms with 2 processes, 17.80 ms with 4, and 15.73 ms with 8. And the speedup in worst case for P=8 is 5.6× compared with the serial version. This demonstrate efficiency of our approach.

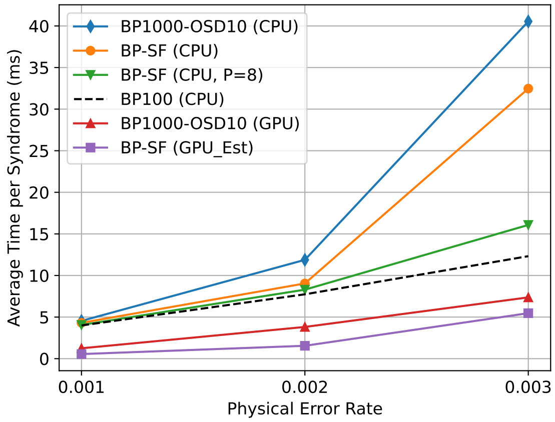


Fig. 14: Average decoding time per syndrome for the BP-SF decoders compared with BP1000-OSD10 and BP100, under varying physical error rates. BP100 is performed on the CPU serially as a reference lower bound, as it performs no post-processing.

Figure 16 shows the runtime distribution for the GPU decoders. We observe a similar pattern: while the BP-SF method achieves a lower average runtime (5.47 ms vs. 7.37 ms), its maximum runtime (73.74 ms) is higher than that of the BP-OSD decoder (39.76 ms), due to the serial decoding of trial syndromes. This long-tail latency can be effectively mitigated by enabling a GPU function that accepts a batch of trial syndromes and returns as soon as any one is successfully decoded as GPU can naturally decode multiple syndromes in parallel. We also observe that the BP-SF decoder exhibits a slightly higher minimum runtime (approximately 0.1 ms), which may be attributed to the use of a wrapper function around the CUDA-Q decoder, which introduces additional parameter I/O and initialization overhead.

Discussion: In addition to CPU and GPU implementations, our BP-SF decoder is also well-suited for deployment on FPGAs and ASICs due to its reliance on components that are already widely adopted in hardware. The core of the decoder is based on BP, which has mature hardware implementations

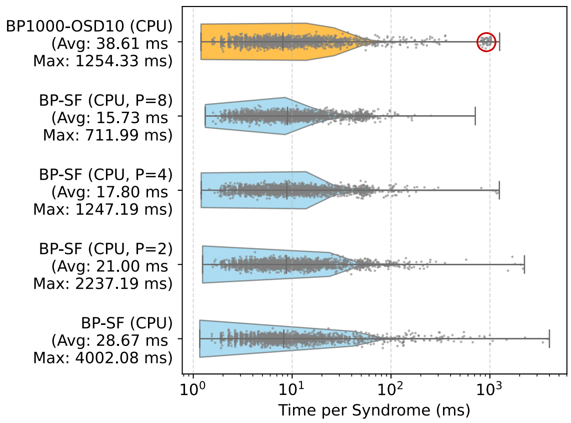


Fig. 15: Distribution of single-syndrome decoding times for different decoder configurations when physical error rate is 0.003. The horizontal violin plots show the probability density of the time taken for each decoding attempt, plotted on a logarithmic scale. Overlaid gray points represent individual decoding events, with vertical jitter added for clarity. The vertical lines mark the min, median, and max decoding time. Decoding events invoking OSD stage are circled in red.

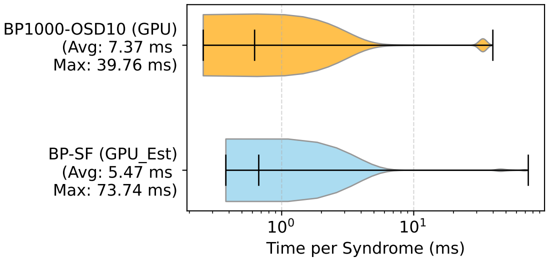


Fig. 16: Distribution of single-syndrome decoding times for GPU decoders when physical error rate is 0.003.

in commercial systems such as 5G mobile devices [24], 10GBASE-T Ethernet [37], digital video broadcasting [8], etc. The remaining components, such as candidate selection and trial syndrome generation, are also lightweight and hardware-friendly. Candidate selection involves identifying the top $|\Phi|$ most oscillating bits, which can be efficiently implemented using partial sorting algorithms such as incomplete selection sort, or full quicksort for larger values of $|\Phi|$. Trial syndrome generation can be formulated as a sparse matrix-sparse vector (SpMSpV) multiplication, another operation that maps well to hardware accelerators. Together, these features make our decoder highly amenable to low-latency, energy-efficient hardware realization. In a superconducting based quantum computer, assuming a typical syndrome extraction round time on the order of 1 μs and a BP iteration latency of approximately 20 ns [28], our decoder can achieve a worst-case latency of about 4 μs when fully parallelized (corresponding to 200 BP iterations, 100 for the initial BP stage and 100 for the parallelized trials). Considering that a full syndrome extraction circuit typically requires d rounds of syndrome extraction, our decoder is fast enough to perform real-time decoding.

VII. Conclusion and Future Work

We introduced a fully parallelizable decoder based on belief propagation (BP). By leveraging speculative decoding and bit-flipping strategies guided by BP oscillation statistics, our proposed BP-SF method achieves logical error rates comparable to BP-OSD, while significantly reducing computational complexity and avoiding costly Gaussian elimination. Extensive simulations show that our decoder performs exceptionally well under the code-capacity noise model across a range of bivariate bicycle codes. Under the more realistic circuit-level noise model, the decoder still delivers reasonable performance, though it requires a larger number of decoding trials to achieve comparable accuracy. We also want to notice that throughout the paper, we adopt the widely used min-sum variant of BP because of its simplicity and computational efficiency. Nonetheless, our approach could potentially benefit from incorporating more advanced BP-based techniques as long as their convergence is also affected by oscillating bits.

In future work, we aim to better understand the challenges posed by circuit-level noise and explore targeted improvements, such as more effective candidate selection, improved trial vector sampling strategies, efficient decoder implementation, and enhancements to the inner BP decoder, in order to further improve decoding performance in practical fault-tolerant quantum computing systems.

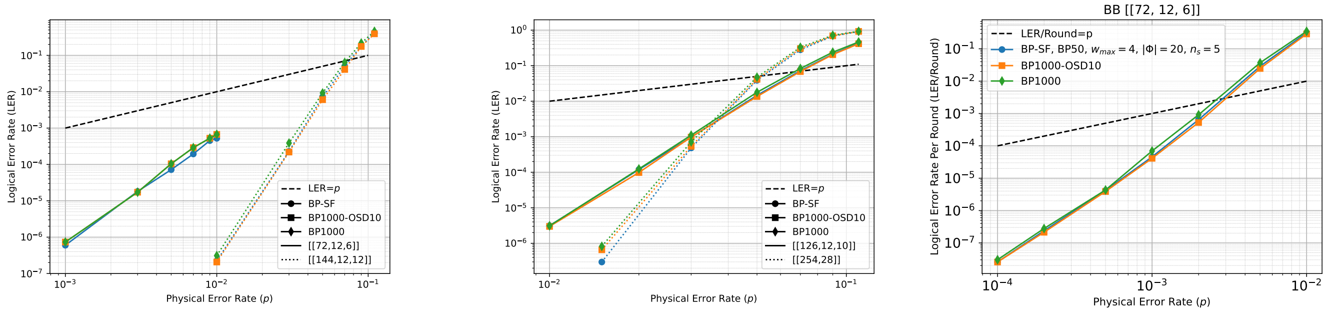
Acknowledgment

The authors would like to thank John Stack for providing source code and for discussions on circuit-level noise simulation, and Timo Hillmann for valuable discussions regarding the BP-OSD decoder. This material is based upon work supported by the U.S. Department of Energy, Office of Science, National Quantum Information Science Research Centers, Codesign Center for Quantum Advantage (C2QA) under contract number DE-SC0012704, (Basic Energy Sciences, PNNL FWP 76274). This research used resources of the Oak Ridge Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC05-00OR22725. This research used resources of the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility located at Lawrence Berkeley National Laboratory, operated under Contract No. DE-AC02-05CH11231. The Pacific Northwest National Laboratory is operated by Battelle for the U.S. Department of Energy under Contract DE-AC05-76RL01830. This work was also supported in part by NSF awards OSI-2410675, PHY-1818914, PHY-2325080, MPS-2120757, CISE-2217020, and CISE-2316201 as well as DOE DE-SC0025384.

Appendix

A. Code Construction

In this section, we provide the construction parameters for the qLDPC codes referenced throughout the paper. The



(a)

(b)

(c)

Fig. 17: (a-b): The logical error rates of different codes under code capacity error model. For the $[\![72, 12, 6]\!]$ BB code, BP and BP-OSD have the same error when we set the seed to the same. BP-SF have $w_{max} = 1$, and $|\Phi| = 4, 7, 6, 13$ for $[\![72, 12, 6]\!]$, $[\![144, 12, 12]\!]$, $[\![126, 12, 10]\!]$, $[\![254, 28]\!]$ codes, respectively. (c): Error rate comparison on the $[\![72, 12, 6]\!]$ under circuit-level noise model.

construction and circuit-level implementation of SHYPS code can be found in [16].

1) GB codes: Let S_l be the shift matrix of size l, defined as

$$S_l = I_l >> 1, \tag{12}$$

where ">>" denotes the right cyclic shift for each row in the matrix, and I_l be the identity matrix of size l. For example,

$$S_3 = I_3 >> 1 = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}. \tag{13}$$

Let $x = S_l$ here. The GB code can be defined by two polynomials, $a(x)$ and $b(x)$. These two polynomials can be represented by two matrices A, B naturally, and the parity check matrices for the GB code are defined as

$$\begin{aligned} H_X &= [A|B] \\ H_Z &= [B^T|A^T]. \end{aligned} \tag{14}$$

The $[\![254, 28]\!]$ GB code used in this paper can be constructed by $l = 127$, $a(x) = 1 + x^{15} + x^{20} + x^{28} + x^{66}$, $b(x) = 1 + x^{58} + x^{59} + x^{100} + x^{121}$ as proposed in [18].

2) BB Codes: BB codes are constructed similarly to GB codes but with two variables. Let $x = S_l \otimes I_m$ and $y = I_l \otimes S_m$, where $\otimes$ denote Kronecker product, the BB codes can also be defined by two polynomials, $A = a(x, y)$ and $B = b(x, y)$. The parity check matrices for BB codes are defined similarly to those in GB codes. The BB codes we used in the paper were proposed in [2]. And the polynomials are shown in Table II.

TABLE II: BB Codes Used in Simulations

l	m	$a(x, y)$	$b(x, y)$	$[\![n, k, d]\!]$
6	6	$x^3 + y + y^2$	$y^3 + x + x^2$	$[\![72, 12, 6]\!]$
12	6	$x^3 + y + y^2$	$y^3 + x + x^2$	$[\![144, 12, 12]\!]$
12	12	$x^3 + y^2 + y^7$	$y^3 + x + x^2$	$[\![288, 12, 18]\!]$

3) Coprime-BB Codes: The coprime-BB code we used originates from [33]. Let $\pi = xy$, where x and y are defined the same as in BB codes. The coprime-BB code we tested can be constructed as shown in Table III.

TABLE III: Coprime-BB Codes Used in Simulations

l	m	$a(\pi)$	$b(\pi)$	$[\![n, k, d]\!]$
7	9	$1 + \pi + \pi^{58}$	$1 + \pi^{13} + \pi^{41}$	$[\![126, 12, 10]\!]$
7	11	$1 + \pi + \pi^{31}$	$1 + \pi^{19} + \pi^{53}$	$[\![154, 6, 16]\!]$

B. "Good" Codes for BP

This subsection presents several codes that demonstrate good performance under BP decoding, along with their corresponding logical error rates. Since both BP-OSD and the proposed BP-SF decoder act as post-processing techniques, they are only invoked when the BP decoder fails to converge. As a result, for the codes shown below, where BP alone achieves high success rates, we observe similar overall performance across all decoding strategies.

Fig. 17(a-c) shows the performance of various decoders on several codes under the code-capacity and circuit level noise model. In these cases, the baseline BP decoder already achieves logical error rates comparable to those of the BP-OSD decoder, and post-processing yields only marginal improvements.

C. Artifact Evaluation

This artifact contains the source code for the BP-SF decoder, a parallelized belief-propagation decoder for quantum LDPC codes, as described in the paper "Fully Parallelized BP Decoding for Quantum LDPC Codes Can Outperform BP-OSD". It includes a custom Cython-based implementation of the Belief Propagation (BP) decoder, scripts for running circuit-level memory experiments using `stim`, and benchmarking tools to compare performance against standard BP, BP-OSD, and CUDA-Q qLDPC decoders. The artifact allows researchers to reproduce the Logical Error Rate (LER) and decoding speed results presented in the paper.

1) Artifact Check-List (Meta-information):

- **Algorithm:** BP-SF, BP-OSD, Standard BP (Min-sum).
- **Program:** Python 3.11 scripts with Cython extensions.
- **Compilation:** C/C++ compiler required for Cython extension.
- **Data set:** Synthetic data generated on-the-fly using `stim` circuit simulations.
- **Run-time environment:** Linux; Python 3.11-13 (newer versions may work but are untested).
- **Hardware:** Standard CPU for functional tests. NVIDIA GPU with CUDA support recommended for `cudaq-qec` comparisons and GPU estimation experiments.
- **Metrics:** Logical Error Rate per Round (LER/Round), average decoding time per sample (ms).
- **Output:** Text logs in `data/` directory and Jupyter Notebook for plotting.
- **How much time is needed to prepare?:** ∼10 minutes (install dependencies).
- **How much time is needed to complete experiments?:** Days on a 16-core machine.
- **Publicly available?:** Yes. https://github.com/Dies-Irae/BP-SF

2) Hardware dependencies: The core BP-SF decoder runs on standard CPUs. To reproduce the GPU comparison baselines (using `cudaq-qec`), an NVIDIA GPU with CUDA support is required.

3) Software dependencies:

- OS: Linux.
- Python: Version 3.11.
- Libraries: `numpy`, `scipy`, `stim`, `cython`, `setuptools`, `matplotlib`.
- Optional (for baselines): `ldpc`, `cudaq-qec`.

4) Installation: 1. Clone the repository:

```
git clone git@github.com:Dies-Irae/BP-SF.git
cd BP-SF
```

2. Create and activate a virtual environment:

```
python –m venv .venv
source .venv/bin/activate
```

3. Install Python dependencies:

```
pip install cython numpy scipy
pip install stim setuptools matplotlib
# Optional: For baselines
pip install cudaq-qec ldpc
```

4. Build the local Cython extension:

```
export PYTHONPATH=./src_python:$PYTHONPATH
cd minimal_bp_decoder
python setup.py build_ext --inplace
cd ..
```

The artifact provides shell scripts to automate the experiments. Results are saved to the `data/` directory.

5) Functional Tests (LER): To evaluate the logical error rate of the BP-SF decoder compared to BP-OSD:

```
mkdir data
# Run BP-SF circuit-level simulation
sh bpsf_circ.sh
# Run BP-OSD circuit-level simulation
# (requires ldpc)
sh bposd_circ.sh
```

6) Performance Benchmarks (Timing): To measure and compare decoding speeds between BP-SF, BP, BP-OSD, and CUDAQ:

```
# Run speed benchmarks
sh time.sh
```

Note: The `time.sh` *script runs benchmarks for varying physical error rates and decoders. Ensure* `cudaq-qec` *is installed for the GPU baseline.*

After running the experiments, the results will be stored as text files in the `data/` folder (e.g., `time_cudaq.txt`, `⋆bb_test_⋆.txt`).

7) Plotting: Open the Jupyter notebook `plots.ipynb` and execute the cells to generate to visualize the results.

- **LER Plots:** Comparison of Logical Error Rate vs. Physical Error Rate for BP-SF and BP-OSD (corresponding to Fig. 7-10 in the paper).
- **Timing Plots:** Comparison of average decoding time per sample (corresponding to Fig. 14 in the paper).

References

[1] E. Berlekamp, R. McEliece, and H. van Tilborg, “On the inherent intractability of certain coding problems (corresp.),” *IEEE Transactions on Information Theory*, vol. 24, no. 3, pp. 384–386, 1978.

[2] 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,” *Nature*, vol. 627, no. 8005, p. 778–782, Mar. 2024.

[3] D. Chase, “Class of algorithms for decoding block codes with channel measurement information,” *IEEE Transactions on Information Theory*, vol. 18, no. 1, pp. 170–182, 1972.

[4] J. Chen, Z. Yi, Z. Liang, and X. Wang, “ Improved Belief Propagation Decoding Algorithms for Surface Codes ,” *IEEE Transactions on Quantum Engineering*, vol. 6, no. 01, pp. 1–16, Jan. 2025. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/TQE.2025.3577769

[5] D. Chytas, M. Pacenti, N. Raveendran, M. F. Flanagan, and B. Vasić, “Enhanced message-passing decoding of degenerate quantum codes utilizing trapping set dynamics,” *IEEE Communications Letters*, vol. 28, no. 3, pp. 444–448, 2024.

[6] P. Das, A. Locharla, and C. Jones, “Lilliput: a lightweight low-latency lookup-table decoder for near-term quantum error correction,” in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS ’22. New York, NY, USA: Association for Computing Machinery, 2022, p. 541–553. [Online]. Available: https://doi.org/10.1145/3503222.3507707

[7] P. Das, C. A. Pattison, S. Manne, D. M. Carmean, K. M. Svore, M. Qureshi, and N. Delfosse, “Afs: Accurate, fast, and scalable error-decoding for fault-tolerant quantum computers,” in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2022, pp. 259–273.

[8] J. Dielissen, A. Hekstra, and V. Berg, “Low cost ldpc decoder for dvb-s2,” in *Proceedings of the Design Automation & Test in Europe Conference*, vol. 2, 2006, pp. 1–6.

[9] I. Dimnik and Y. Be’ery, “Improved random redundant iterative hdpc decoding,” *Trans. Comm.*, vol. 57, no. 7, p. 1982–1985, Jul. 2009. [Online]. Available: https://doi.org/10.1109/TCOMM.2009.07.070621

[10] C. Gidney, “Stim: a fast stabilizer circuit simulator,” *Quantum*, vol. 5, p. 497, 2021.

[11] A. Gong, S. Cammerer, and J. M. Renes, “Toward low-latency iterative decoding of qldpc codes under circuit-level noise,” *arXiv preprint arXiv:2403.18901*, 2024.

[12] T. R. Halford and K. M. Chugg, "Random redundant soft-in soft-out decoding of linear block codes," in *2006 IEEE International Symposium on Information Theory*, 2006, pp. 2230–2234.

[13] T. Hillmann, L. Berent, A. O. Quintavalle, J. Eisert, R. Wille, and J. Roffe, "Localized statistics decoding: A parallel decoding algorithm for quantum low-density parity-check codes," *arXiv preprint arXiv:2406.18655*, 2024.

[14] P. Iyer and D. Poulin, "Hardness of decoding quantum stabilizer codes," *IEEE Trans. Inf. Theor.*, vol. 61, no. 9, p. 5209–5223, Sep. 2015. [Online]. Available: https://doi.org/10.1109/TIT.2015.2422294

[15] S. Koutsioumpas, H. Sayginel, M. Webster, and D. E. Browne, "Automorphism ensemble decoding of quantum ldpc codes," *arXiv preprint arXiv:2503.01738*, 2025.

[16] A. J. Malcolm, A. N. Glaudell, P. Fuentes, D. Chandra, A. Schotte, C. DeLisle, R. Haenel, A. Ebrahimi, J. Roffe, A. O. Quintavalle *et al.*, "Computing efficiently in qldpc codes," *arXiv preprint arXiv:2502.07150*, 2025.

[17] T. Müller, T. Alexander, M. E. Beverland, M. Bühler, B. R. Johnson, T. Maurer, and D. Vandeth, "Improved belief propagation is sufficient for real-time decoding of quantum memory," *arXiv preprint arXiv:2506.01779*, 2025.

[18] P. Panteleev and G. Kalachev, "Degenerate quantum LDPC codes with good finite length performance," *Quantum*, vol. 5, p. 585, 2021.

[19] D. Poulin and Y. Chung, "On the iterative decoding of sparse quantum codes," *arXiv preprint arXiv:0801.1241*, 2008.

[20] N. Raveendran and B. Vasić, "Trapping sets of quantum ldpc codes," *Quantum*, vol. 5, p. 562, 2021.

[21] T. Richardson, M. Shokrollahi, and R. Urbanke, "Design of capacity-approaching irregular low-density parity-check codes," *IEEE Transactions on Information Theory*, vol. 47, no. 2, pp. 619–637, 2001.

[22] J. Roffe, "LDPC: Python tools for low density parity check codes," 2022. [Online]. Available: https://pypi.org/project/ldpc/

[23] J. Roffe, D. R. White, S. Burton, and E. Campbell, "Decoding across the quantum low-density parity-check code landscape," *Phys. Rev. Res.*, vol. 2, p. 043423, Dec 2020.

[24] B.-S. Su, C.-H. Lee, and T.-D. Chiueh, "A 58.6/91.3 pj/b dual-mode belief-propagation decoder for ldpc and polar codes in the 5g communications standard," *IEEE Solid-State Circuits Letters*, vol. 5, pp. 98–101, 2022.

[25] B. M. Terhal, "Quantum error correction for quantum memories," *Rev. Mod. Phys.*, vol. 87, pp. 307–346, Apr 2015. [Online]. Available: https://link.aps.org/doi/10.1103/RevModPhys.87.307

[26] The CUDA-Q development team, "CUDA-Q," https://github.com/NVIDIA/cuda-quantum, NVIDIA, 2024. [Online]. Available: https://github.com/NVIDIA/cuda-quantum

[27] 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, 2014.

[28] J. Valls, F. Garcia-Herrero, N. Raveendran, and B. Vasić, "Syndrome-based min-sum vs osd-0 decoders: Fpga implementation and analysis for quantum ldpc codes," *IEEE Access*, vol. 9, pp. 138 734–138 743, 2021.

[29] J. Viszlai, J. D. Chadwick, S. Joshi, G. S. Ravi, Y. Li, and F. T. Chong, "Swiper: Minimizing fault-tolerant quantum program latency via speculative window decoding," in *Proceedings of the 52nd Annual International Symposium on Computer Architecture*, ser. ISCA '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 1386–1401. [Online]. Available: https://doi.org/10.1145/3695053.3731022

[30] S. Vittal, P. Das, and M. Qureshi, "Astrea: Accurate quantum error-decoding via practical minimum-weight perfect-matching," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ser. ISCA '23. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3579371.3589037

[31] S. Vittal, A. Javadi-Abhari, A. W. Cross, L. S. Bishop, and M. Qureshi, " Flag-Proxy Networks: Overcoming the Architectural, Scheduling and Decoding Obstacles of Quantum LDPC Codes ," in *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Los Alamitos, CA, USA: IEEE Computer Society, Nov. 2024, pp. 718–734. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/MICRO61859.2024.00059

[32] M. Wang, Y. Li, J. Liu, T. Guo, H. Wu, and F. C. Lau, "Neural layered min-sum decoders for cyclic codes," *Physical Communication*, vol. 61, p. 102194, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1874490723001970

[33] M. Wang and F. Mueller, "Coprime bivariate bicycle codes and their properties," *arXiv preprint arXiv:2408.10001*, 2024.

[34] S. Wolanski and B. Barber, "Ambiguity clustering: an accurate and efficient decoder for qldpc codes," *arXiv preprint arXiv:2406.14527*, 2024.

[35] H. Yao, W. A. Laban, C. Häger, A. G. i. Amat, and H. D. Pfister, "Belief propagation decoding of quantum ldpc codes with guided decimation," in *2024 IEEE International Symposium on Information Theory (ISIT)*, 2024, pp. 2478–2483.

[36] K. Yin, X. Fang, J. Ruan, H. Zhang, D. Tullsen, A. Sornborger, C. Liu, A. Li, T. Humble, and Y. Ding, "Symbreak: Mitigating quantum degeneracy issues in qldpc code decoders by breaking symmetry," *arXiv preprint arXiv:2412.02885*, 2024.

[37] Z. Zhang, V. Anantharam, M. J. Wainwright, and B. Nikolic, "An efficient 10gbase-t ethernet ldpc decoder design with low error floors," *IEEE Journal of Solid-State Circuits*, vol. 45, no. 4, pp. 843–855, 2010.