PRD 21: Quantum LDPC Codes for Fault-Tolerant Quantum Computing

Pure Thought AI Challenges

January 19, 2026

Contents

Domain: Quantum Information Theory **Timeline**: 7-10 months **Difficulty**: High **Prerequisites**: Quantum error correction, coding theory, graph theory, linear algebra over GF(2), algebraic topology

—

0.1 1. Problem Statement

0.1.1 Scientific Context

Quantum Low-Density Parity-Check (QLDPC) codes represent a breakthrough in quantum error correction, offering the potential for **fault-tolerant quantum computation with constant overhead** — a Holy Grail problem in quantum information theory. Unlike surface codes (the current experimental standard), which require O(k log k) physical qubits to encode k logical qubits due to non-local syndrome extraction, QLDPC codes promise **O(k) scaling with constant-weight stabilizers**.

The foundational challenge in quantum error correction is the **no-cloning theorem**: you cannot copy quantum information to create redundancy. Instead, quantum codes encode k logical qubits into n ¿ k physical qubits using **stabilizer groups** — commuting Pauli operators whose +1 eigenspace defines the code subspace. The **stabilizer formalism** (Gottesman 1997) reduces quantum error correction to linear algebra over GF(2).

Classical LDPC codes (Gallager 1962) revolutionized classical error correction with: - **Sparse parity checks**: Each check involves O(1) bits (not all n bits) - **Near-capacity performance**: Achieve Shannon limit with efficient BP decoding - **Practical deployment**: Used in WiFi (802.11n), 5G, DVB-S2 satellite communication

Quantum LDPC codes inherit these properties but face additional constraints: - **CSS condition**: X-type and Z-type stabilizers must commute globally - **Locality vs distance tradeoff**: High distance requires long-range entanglement - **Decoding complexity**: Quantum errors (X, Y, Z) are continuous; syndrome measurement is noisy

Recent Breakthroughs (2020-2024): - **Asymptotically good QLDPC**: Codes with k = (n) logical qubits and d = (n) distance proven to exist (Panteleev-Kalachev 2022, Breuckmann-Eberhardt 2021) - **Fiber bundle codes**: Explicit construction using algebraic topology, achieving [[n, (n), (n)]] scaling - **Balanced product codes**: Generalization of hypergraph product with tunable parameters - **Linear-time decoders**: BP-based decoders with O(n) complexity (Roffe et al. 2020)

These codes are poised to **replace surface codes** in future quantum computers, reducing qubit count by orders of magnitude for fault-tolerant quantum algorithms (Shor's algorithm, quantum chemistry, optimization).

0.1.2 Core Question

Can we construct explicit families of quantum LDPC codes achieving: 1. **Constant rate**: k/n = (1) (not vanishing with n) 2. **Linear distance**: d = (n) (or at least d = (n)) 3. **Constant locality**: Each stabilizer has weight w = O(1) 4. **Efficient decoding**: Algorithm running in O(n polylog n) time with threshold p_th ¿ 0

Using ONLY: - Classical LDPC codes + CSS construction - Hypergraph product / balanced product / fiber bundle methods - Graph expansion theory (Ramanujan graphs, expanders) - Algebraic topology (chain complexes, homology) - NO empirical tuning or machine learning until validation phase

0.1.3 Why This Matters

- **Scalable quantum computing**: Reducing physical qubit overhead from 10-10 (surface codes) to 10^2-10^3 (QLDPC) makes large-scale QC feasible - **Distributed quantum networks**:

QLDPC enables efficient quantum error correction for long-distance entanglement distribution - **Topological quantum memory**: Understanding QLDPC connects to anyonic systems and topological order in condensed matter - **Coding theory**: Quantum codes reveal deep connections between graph theory, topology, and information theory

0.1.4 Pure Thought Advantages

1. **Exact symbolic methods**: Code distance and parameters are rigorously proven via linear algebra over GF(2) 2. **Certificates**: Stabilizer commutation, CSS condition, and distance bounds are machine-verifiable 3. **No hardware dependence**: Code constructions are abstract (graphs, chain complexes), independent of physical platform 4. **Scalability**: Hypergraph product constructions generate codes for arbitrary n programmatically

—

0.2 2. Mathematical Formulation

0.2.1 Stabilizer Formalism

A **quantum stabilizer code** is defined by a stabilizer group $S \subset \mathcal{P}_n$ where $\mathcal{P}_n$ is then $-qubit Pauli group (generated by X, Z, I)$.

Code subspace: $C = \{—\psi\rangle : S|\psi\rangle = |\psi\rangle \text{ for all } S \in \mathcal{S}\}$

Parameters: [[n, k, d]] where: - n = number of physical qubits - k = number of encoded logical qubits: $k = n - \log_2|\mathcal{S}|$ $(assuming independent stabilizers) - d = code distance(minimum weight of non-trivial logical operators)$

Stabilizer generators: $S = \langle S_1, \ldots, S_{n-k}\rangle$ $(abelian group with n-k independent generators)$

0.2.2 CSS Codes

CSS (Calderbank-Shor-Steane) construction from two classical binary linear codes $C_1, C_2 \subseteq \mathbb{F}_2^n$:

Condition: $C_2^{\perp} \subseteq C_1$ $(ensures X and Z stabilizers commute)$

Stabilizer generators: - X-type: For each row $h_x \in H_{C_1^{\perp}}$, $define$ $X_{h_x} = \bigotimes_{i:(h_x)_i=1} X_i$ $- Z - type : For each row $h_z \in H_{C_2^{\perp}}$, $define$ $Z_{h_z} = \bigotimes_{i:(h_z)_i=1} Z_i$

Parameters: $$k = \dim(C_1) - \dim(C_2^{\perp}) = \dim(C_1) + \dim(C_2) - n$$$$ $$d \geq \min(d(C_1^{\perp}), d(C_2^{\perp}))$$

where $d(C)$ is the minimum Hamming weight of non-zero codewords in C.

0.2.3 QLDPC Definition

A CSS code is **QLDPC** if the parity check matrices H_X (for $C_1^{\perp}$) and H_Z $(for $C_2^{\perp}$) are * *sparse * * : - * *Row weight * * : $w_r = O(1)$ $(each stabilizer acts on O(1) qubits) - * * Column weight * * : $w_c = O(1)$ $(each qubit participates in O(1) stabilizers)$

Tanner graph: Bipartite graph $G = (Q \cup C, E)$ where : $-Q = qubit nodes (n vertices) - C = check nodes (n - k vertices) - Edge$ (q, c) if qubit q appears in stabilizer c

Girth: Shortest cycle length in Tanner graph. Higher girth $\rightarrow$ better BP decoding performance.

0.2.4 Hypergraph Product

Construction: Given two classical codes with parity matrices $H_1 \in \mathbb{F}_2^{m_1 \times n_1}$ and $H_2 \in \mathbb{F}_2^{m_2 \times n_2}$, $define$:

$$H_X = \left[H_1 \otimes I_{n_2} \& I_{m_1} \otimes H_2^T\right]$$ $$H_Z = \left[I_{n_1} \otimes H_2 \& H_1^T \otimes I_{m_2}\right]$$

Parameters: - $n = n_1 n_2 + m_1 m_2$ (physical qubits) - $k \geq k_1 k_2$ $(logical qubits, where $k_i = n_i - m_i$) - $d \geq \min(d_1, d_2)$ $(distance) - If H_1, H_2 are LDPC, then H_X, H_Z are QLDPC with same u

Key Insight: Hypergraph product **preserves sparsity** and **boosts parameters** — starting from good classical LDPC codes yields good QLDPC codes.

0.2.5 Good QLDPC Codes

A family ${C_n}$ of codes with $C_n = [[n_i, k_i, d_i]]$ is **asymptotically good** if: $$\liminf_{i\to\infty} \frac{k_i}{n_i} > 0 \quad \text{(constant rate)}$$$$\liminf_{i\to\infty} \frac{d_i}{n_i} > 0 \quad \text{(constant relative distance)}$$

Theorem (Panteleev-Kalachev 2022): There exist families of $[[n, \Theta(n), \Theta(n)]]$$QLDPCcodeswithconst

Explicit constructions (Breuckmann-Eberhardt 2021, Hastings-Haah-O'Donnell 2021): Fiber bundle codes achieve $[[n, \Theta(n), \Theta(\sqrt{n})]]$$withexplicitconstruction.

0.2.6 Decoding Problem

Input: Syndrome $s \in \mathbb{F}_2^{n-k}$$frommeasuringstabilizers(some$S_i$give-1eigenvalueßbit"1"insyndrom

Output: Error estimate $ \in \{X, Z, Y\}^n$$suchthat$\hat{e}$hassamesyndromeastrueerrore

Complexity: NP-hard in general (syndrome decoding problem). For QLDPC, hope for efficient approximate decoders.

Belief Propagation (BP): Iterative message-passing on Tanner graph. Complexity O(n) per iteration. Works well for codes with high girth.

Threshold: Maximum error rate $p_{th}$$belowwhichdecodingsucceedswithhighprobabilityas$n \to \infty$.

0.2.7 Certificate Specification

A **valid QLDPC code certificate** must include:

1. **Stabilizer Matrices** $H_X, H_Z \in \mathbb{F}_2^{(n-k)\times n}$: $-Verify : rowrank = n-k(independentstabilizers)-Verify : $H_X H_Z^T = 0 \mod 2$(CSScommutationcondition)

2. **QLDPC Property**: - Row weights: $\max_i \|H_X[i,:]\|_0 \le w_r$$and$\max_i \|H_Z[i,:]\|_0 \le w_r$ $-Columnweights : $\max_j \|H_X[:,j]\|_0 \le w_c$$and$\max_j \|H_Z[:,j]\|_0 \le w_c$

3. **Code Distance**: - Lower bound $d \ge d_{\min}$$via : $-Exhaustivesearch(forsmalln < 20) - Linearprogrammingbound - Algebraicbounds(forproductcodes)$

4. **Logical Operators**: - X-logical: $ \in \mathbb{F}_2^{k\times n}$$with$\bar{X} H_Z^T = 0$, $\bar{X} H_X^T \neq 0$ $-Z - logical : $\bar{Z} \in \mathbb{F}_2^{k\times n}$$with$\bar{Z} H_X^T = 0$, $\bar{Z} H_Z^T \neq 0$

5. **Decoding Threshold** (via simulation): - Physical error rate p sampled from $[10^{-5}, 10^{-1}]$$-Logicalerrorrate$p_L(p)$$measuredover10trials - Threshold : $p_{th} = \sup\{p : p_L(p) < p\}$

Export Format: JSON + HDF5 with: - Matrices H_X, H_Z (sparse CSR format) - Parameters [[n, k, d]] - Logical operators (sparse) - Decoding simulation data (threshold curves)

—

0.3 3. Implementation Approach

0.3.1 Phase 1: Classical LDPC Codes and GF(2) Algebra (Months 1-2)

Goal: Implement classical LDPC codes, verify properties, test BP decoding.

```
import numpy as np
from scipy.sparse import csr_matrix, lil_matrix
import galois  # GF(2) linear algebra
from typing import Tuple, List

# GF(2) field
GF2 = galois.GF(2)

def generate_random_regular_ldpc(n: int, w_col: int, w_row: int) -> np.
    ndarray:
```

```
    """
    Generate random (w_row, w_col)-regular LDPC parity check matrix.

    Parameters:
        n: code length
        w_col: column weight (each bit in w_col checks)
        w_row: row weight (each check involves w_row bits)

    Returns:
        H: parity check matrix (m x n) over GF(2)
    """
    # Number of checks: m * w_row = n * w_col (degree sum)
    m = (n * w_col) // w_row

    # Edge list construction (random permutation)
    edges = []
    for col in range(n):
        edges.extend([(col, -1)] * w_col)  # -1 is placeholder

    # Shuffle and assign to rows
    np.random.shuffle(edges)

    H = lil_matrix((m, n), dtype=int)

    edge_idx = 0
    for row in range(m):
        for _ in range(w_row):
            col = edges[edge_idx][0]
            H[row, col] = 1
            edge_idx += 1

    return GF2(H.toarray())


def compute_dual_code(G: np.ndarray) -> np.ndarray:
    """
    Compute parity check matrix H for dual code C^perp.

    If G is generator for C, then H generates C^perp with G H^T = 0.
    """
    G_gf2 = GF2(G)

    # Systematic form: G = [I_k | P]
    # Then H = [-P^T | I_{n-k}] = [P^T | I_{n-k}] in GF(2)

    k, n = G_gf2.shape

    # Row echelon form
    G_rref, pivots = G_gf2.row_reduce(True)

    # Extract P (non-pivot columns)
    P = G_rref[:, k:]

    # Dual: H = [P^T | I]
    H = np.hstack([P.T, GF2.Identity(n - k)])

    return H
```

```
def check_ldpc_sparsity(H: np.ndarray, w_row_max: int = 10, w_col_max:
    int = 10) -> bool:
    """
    Verify H is LDPC (sparse).
    """
    row_weights = np.sum(H, axis=1)
    col_weights = np.sum(H, axis=0)

    max_row = int(np.max(row_weights))
    max_col = int(np.max(col_weights))

    print(f"Max row weight: {max_row}, max col weight: {max_col}")

    return max_row <= w_row_max and max_col <= w_col_max


def compute_tanner_graph_girth(H: np.ndarray) -> int:
    """
    Compute girth of Tanner graph (shortest cycle).

    High girth     better BP performance.
    """
    import networkx as nx

    m, n = H.shape

    # Bipartite graph: qubit nodes (0..n-1), check nodes (n..n+m-1)
    G = nx.Graph()

    for i in range(m):
        for j in range(n):
            if H[i, j] == 1:
                G.add_edge(j, n + i)  # Qubit j to check i

    try:
        girth = nx.girth(G)
    except:
        girth = float('inf')  # No cycles (tree)

    return girth


# Test: Generate (3, 6)-regular LDPC
def test_ldpc_generation():
    n = 120
    w_col = 3
    w_row = 6

    H = generate_random_regular_ldpc(n, w_col, w_row)

    print(f"Generated LDPC: {H.shape}")
    assert check_ldpc_sparsity(H, w_row_max=10, w_col_max=10)

    girth = compute_tanner_graph_girth(H)
    print(f"Tanner graph girth: {girth}")

    assert girth >= 4, "Girth too small!"
```

```
    print("    LDPC code generated successfully")
```

Output: Random (3,6)-regular LDPC code with girth 6.

—

0.3.2 Phase 2: CSS Code Construction (Months 2-4)

Goal: Construct CSS quantum codes from classical codes, verify stabilizer commutation.

```
def css_code_from_classical(C1_generator: np.ndarray, C2_generator: np.
    ndarray) -> dict:
    """
    Construct CSS code from two classical codes  C  ,  C  .

    Condition:  C  ^perp      C   (ensures commutation).

    Returns:
        H_X, H_Z: stabilizer matrices
        n, k, d: code parameters
    """
    C1 = GF2(C1_generator)
    C2 = GF2(C2_generator)

    # Dual codes
    H1 = compute_dual_code(C1)  #  C  ^perp
    H2 = compute_dual_code(C2)  #  C  ^perp

    # Check CSS condition: H2 (rows of  C  ^perp) must be in  C   (span
        of C1)
    # i.e., H2 * C1.T = 0

    if not np.all((H2 @ C1.T) == 0):
        raise ValueError("CSS condition violated:  C  ^perp not subset
            of  C  !")

    # X-stabilizers from  C  ^perp
    H_X = H1

    # Z-stabilizers from  C  ^perp
    H_Z = H2

    # Code parameters
    n = C1.shape[1]
    k = compute_logical_dimension(H_X, H_Z, n)
    d_lower_bound = estimate_code_distance(H_X, H_Z)

    return {
        'H_X': H_X,
        'H_Z': H_Z,
        'parameters': [n, k, d_lower_bound],
        'n': n,
        'k': k,
        'd': d_lower_bound
    }


def compute_logical_dimension(H_X: np.ndarray, H_Z: np.ndarray, n: int)
    -> int:
    """
```

```
    k = n - rank(H_X) - rank(H_Z)
    """
    rank_X = np.linalg.matrix_rank(GF2(H_X))
    rank_Z = np.linalg.matrix_rank(GF2(H_Z))

    return n - rank_X - rank_Z


def verify_css_commutation(H_X: np.ndarray, H_Z: np.ndarray) -> bool:
    """
    Verify [X_i, Z_j] = 0 for all stabilizers.

    Equivalent to: H_X * H_Z^T = 0 (mod 2)
    """
    commutator = GF2(H_X) @ GF2(H_Z).T

    return np.all(commutator == 0)


def estimate_code_distance(H_X: np.ndarray, H_Z: np.ndarray, max_weight:
    int = 20) -> int:
    """
    Estimate code distance via exhaustive search (small codes only).

    Distance = min weight of logical operators (not in stabilizer group)
       .
    """
    n = H_X.shape[1]

    # Simplified: check weight-1, weight-2, ... errors
    for w in range(1, min(max_weight, n) + 1):
        # Check if weight-w X error is detectable
        for error in itertools.combinations(range(n), w):
            e_vec = np.zeros(n, dtype=int)
            e_vec[list(error)] = 1

            syndrome_X = GF2(H_X) @ GF2(e_vec)
            syndrome_Z = GF2(H_Z) @ GF2(e_vec)

            # If both syndromes are zero, it's a logical operator
            if np.all(syndrome_X == 0) and np.all(syndrome_Z == 0):
                # Check if it's non-trivial (not in stabilizer)
                # (Simplified check)
                return w

    return max_weight  # Lower bound


# Test: Construct small CSS code
def test_css_code():
    # Example: Steane [[7,1,3]] code
    #  C   =  C   = Hamming [7,4,3] code

    # Hamming [7,4,3] generator matrix
    G_hamming = GF2([[1, 0, 0, 0, 0, 1, 1],
                     [0, 1, 0, 0, 1, 0, 1],
                     [0, 0, 1, 0, 1, 1, 0],
                     [0, 0, 0, 1, 1, 1, 1]])
```

```
    css = css_code_from_classical(G_hamming, G_hamming)

    print(f"CSS code parameters: [[{css['n']}, {css['k']}, {css['d
        ']}]]")

    # Verify commutation
    assert verify_css_commutation(css['H_X'], css['H_Z'])
    print("    CSS stabilizers commute")

    # Expected: [[7, 1, 3]]
    assert css['n'] == 7 and css['k'] == 1 and css['d'] >= 3
    print("    Steane code parameters verified")
```

—

0.3.3 Phase 3: Hypergraph Product Construction (Months 4-5)

Goal: Implement hypergraph product to generate large QLDPC codes.

```
def hypergraph_product(H1: np.ndarray, H2: np.ndarray) -> Tuple[np.
    ndarray, np.ndarray]:
    """
    Hypergraph product: quantum code from two classical codes.

    H_X = [H1     I_{n2} | I_{m1}     H2^T]
    H_Z = [I_{n1}     H2 | H1^T     I_{m2}]

    Parameters:
        H1: (m1, n1) parity matrix for  C
        H2: (m2, n2) parity matrix for  C

    Returns:
        H_X, H_Z: quantum stabilizer matrices
    """
    H1_gf2 = GF2(H1)
    H2_gf2 = GF2(H2)

    m1, n1 = H1.shape
    m2, n2 = H2.shape

    # X-check matrix: [H1     I_{n2} | I_{m1}     H2^T]
    block1 = np.kron(H1_gf2, GF2.Identity(n2))
    block2 = np.kron(GF2.Identity(m1), H2_gf2.T)

    H_X = np.hstack([block1, block2])

    # Z-check matrix: [I_{n1}     H2 | H1^T     I_{m2}]
    block3 = np.kron(GF2.Identity(n1), H2_gf2)
    block4 = np.kron(H1_gf2.T, GF2.Identity(m2))

    H_Z = np.hstack([block3, block4])

    return H_X, H_Z


def hypergraph_product_parameters(H1: np.ndarray, H2: np.ndarray, d1:
    int, d2: int) -> dict:
    """
```

```
    Compute parameters of HP code.

    n = n1*n2 + m1*m2
    k >= k1*k2 (where k_i = n_i - rank(H_i))
    d >= min(d1, d2)
    """
    m1, n1 = H1.shape
    m2, n2 = H2.shape

    rank1 = np.linalg.matrix_rank(GF2(H1))
    rank2 = np.linalg.matrix_rank(GF2(H2))

    k1 = n1 - rank1
    k2 = n2 - rank2

    n_quantum = n1 * n2 + m1 * m2
    k_quantum_lower = k1 * k2
    d_quantum_lower = min(d1, d2)

    return {
        'n': n_quantum,
        'k_lower': k_quantum_lower,
        'd_lower': d_quantum_lower
    }


# Test: Hypergraph product of two small LDPC codes
def test_hypergraph_product():
    # Two small [n, k, d] = [7, 4, 3] Hamming codes
    H1 = GF2([[1, 1, 1, 0, 1, 0, 0],
              [1, 0, 0, 1, 0, 1, 0],
              [0, 1, 0, 1, 0, 0, 1]])

    H2 = H1  # Same code

    H_X, H_Z = hypergraph_product(H1, H2)

    print(f"HP code stabilizers: H_X shape {H_X.shape}, H_Z shape {H_Z.
       shape}")

    # Verify commutation
    assert verify_css_commutation(H_X, H_Z)
    print("    HP code stabilizers commute")

    # Compute parameters
    params = hypergraph_product_parameters(H1, H2, d1=3, d2=3)
    print(f"HP code parameters (lower bounds): [[{params['n']}, >={
       params['k_lower']}, >={params['d_lower']}]]")

    # Verify LDPC
    assert check_ldpc_sparsity(H_X, w_row_max=10, w_col_max=10)
    assert check_ldpc_sparsity(H_Z, w_row_max=10, w_col_max=10)
    print("    HP code is QLDPC")
```

Expected Output: [[90, ¿=16, ¿=3]] QLDPC code from [7,4,3] Hamming codes.

—

0.3.4 Phase 4: Belief Propagation Decoding (Months 5-7)

Goal: Implement BP decoder for QLDPC, measure decoding threshold.

```
def belief_propagation_decoder(syndrome: np.ndarray, H: np.ndarray,
                               p_error: float = 0.01, max_iters: int =
                                  100) -> np.ndarray:
    """
    BP decoder for QLDPC codes.

    Args:
        syndrome: (m,) binary syndrome vector
        H: (m, n) parity check matrix
        p_error: physical error rate (prior)
        max_iters: maximum BP iterations

    Returns:
        error_estimate: (n,) binary error vector
    """
    m, n = H.shape

    # Log-likelihood ratios
    # LLR(x) = log(P(x=0) / P(x=1))
    llr_prior = np.log((1 - p_error) / p_error) * np.ones(n)

    # Messages: q->c and c->q
    llr_q_to_c = np.tile(llr_prior[:, None], (1, m)).T  # (m, n)
    llr_c_to_q = np.zeros((m, n))

    for iteration in range(max_iters):
        # Update check-to-qubit messages
        for check in range(m):
            neighbors = np.where(H[check] == 1)[0]

            for qubit in neighbors:
                # Product of tanh(LLR/2) from other qubits
                other_qubits = neighbors[neighbors != qubit]

                if len(other_qubits) == 0:
                    llr_c_to_q[check, qubit] = 0
                    continue

                prod_tanh = np.prod(np.tanh(llr_q_to_c[check,
                   other_qubits] / 2))

                # Syndrome flip: if syndrome[check] = 1, negate product
                if syndrome[check] == 1:
                    prod_tanh *= -1

                # Convert back to LLR
                llr_c_to_q[check, qubit] = 2 * np.arctanh(np.clip(
                   prod_tanh, -0.999, 0.999))

        # Update qubit-to-check messages
        for qubit in range(n):
            neighbors = np.where(H[:, qubit] == 1)[0]

            for check in neighbors:
                other_checks = neighbors[neighbors != check]
```

```
                llr_q_to_c[check, qubit] = llr_prior[qubit] + np.sum(
                    llr_c_to_q[other_checks, qubit])

        # Decision: posterior LLR
        llr_posterior = llr_prior + np.sum(llr_c_to_q, axis=0)

        error_estimate = (llr_posterior < 0).astype(int)

        # Check if syndrome matches
        computed_syndrome = (H @ error_estimate) % 2

        if np.all(computed_syndrome == syndrome):
            print(f"BP converged at iteration {iteration + 1}")
            return error_estimate

    print("BP did not converge")
    return error_estimate


def simulate_decoding_threshold(H_X: np.ndarray, H_Z: np.ndarray,
                                p_range: np.ndarray, num_trials: int =
                                    1000) -> dict:
    """
    Measure decoding threshold via Monte Carlo simulation.

    Args:
        H_X, H_Z: stabilizer matrices
        p_range: array of physical error rates to test
        num_trials: number of trials per p

    Returns:
        results: dict with p_values, logical_error_rates
    """
    m_X, n = H_X.shape
    m_Z, _ = H_Z.shape

    logical_error_rates = []

    for p in p_range:
        print(f"\nTesting p = {p:.4f}...")

        failures = 0

        for trial in range(num_trials):
            # Sample random X errors
            error_X = (np.random.rand(n) < p).astype(int)

            # Syndrome
            syndrome_X = (H_X @ error_X) % 2

            # Decode
            decoded_X = belief_propagation_decoder(syndrome_X, H_X,
                p_error=p, max_iters=50)

            # Residual error
            residual_X = (error_X + decoded_X) % 2
```

```
            # Check if residual is a logical error (not in stabilizer,
               commutes with Z-stabs)
            syndrome_residual = (H_X @ residual_X) % 2

            if not np.all(syndrome_residual == 0):
                # Decoding failed to find valid stabilizer error
                failures += 1
                continue

            # Check if it's a non-trivial logical (simplified: weight >
               0 and commutes)
            if np.sum(residual_X) > 0:
                # Logical error
                failures += 1

        logical_error_rate = failures / num_trials
        logical_error_rates.append(logical_error_rate)

        print(f"Logical error rate: {logical_error_rate:.4f}")

    return {
        'p_values': p_range,
        'logical_error_rates': np.array(logical_error_rates)
    }


# Test: Threshold simulation for small code
def test_bp_threshold():
    # Use HP code from previous test
    H1 = GF2([[1, 1, 1, 0, 1, 0, 0],
              [1, 0, 0, 1, 0, 1, 0],
              [0, 1, 0, 1, 0, 0, 1]])
    H_X, H_Z = hypergraph_product(H1, H1)

    p_range = np.linspace(0.001, 0.05, 10)

    results = simulate_decoding_threshold(H_X, H_Z, p_range, num_trials
       =100)

    # Threshold: p where logical error rate crosses physical error rate
    # (Simplified check)
    print(f"\ n   BP decoding threshold simulation complete")
    print(f"Threshold estimate: p_th     {results['p_values'][np.argmax(
       results['logical_error_rates'] > results['p_values'])]:.4f}")
```

—

0.3.5 Phase 5: Good QLDPC Constructions (Months 7-9)

Goal: Implement advanced constructions (fiber bundle, balanced product) achieving constant rate.

```
def balanced_product_code(H1: np.ndarray, H2: np.ndarray, G1: np.ndarray
    , G2: np.ndarray) -> dict:
    """
    Balanced product construction (Breuckmann-Eberhardt 2021).

    Generalizes hypergraph product with tunable parameters.
```

```
    Requires both generator G and parity H for input codes.
    """
    # Simplified implementation (full version requires chain complex
       formalism)

    m1, n1 = H1.shape
    m2, n2 = H2.shape

    k1, _ = G1.shape
    k2, _ = G2.shape

    # Balanced product uses both G and H to construct stabilizers
    # (Details omitted for brevity)

    # Placeholder: use HP for now
    H_X, H_Z = hypergraph_product(H1, H2)

    return {
        'H_X': H_X,
        'H_Z': H_Z,
        'note': 'Balanced product (simplified to HP)'
    }


def construct_expander_qldpc(n_target: int, degree: int = 3) -> dict:
    """
    Construct QLDPC from expander graphs.

    Good expanders     good codes (via Sipser-Spielman).
    """
    import networkx as nx

    # Generate Ramanujan graph (optimal expander)
    # (Simplified: use random regular graph as proxy)

    G = nx.random_regular_graph(degree, n_target)

    # Adjacency matrix as parity check
    A = nx.adjacency_matrix(G).toarray()

    # Apply hypergraph product
    H_X, H_Z = hypergraph_product(A, A)

    return {
        'H_X': H_X,
        'H_Z': H_Z,
        'expansion': 'random_regular'  # Placeholder
    }


# Comparison: Surface code vs QLDPC
def compare_overhead(k_logical: int):
    """
    Compare qubit overhead for surface code vs QLDPC.

    Surface: n = O(k log k)
    QLDPC: n = O(k)
    """
```

```
    # Surface code: [[d^2, 1, d]] per logical qubit
    # For distance d~100: n_surf = 100^2 = 10,000 physical per logical

    d_surface = 100
    n_surface_per_logical = d_surface ** 2

    n_surface_total = k_logical * n_surface_per_logical

    # QLDPC: [[n, n/4, sqrt(n)]] typical
    # For k_logical: need n_qldpc = 4 * k_logical

    rate_qldpc = 0.25
    n_qldpc_total = int(k_logical / rate_qldpc)

    print(f"Overhead comparison for k={k_logical} logical qubits:")
    print(f"  Surface code: n = {n_surface_total} physical qubits")
    print(f"  QLDPC code: n = {n_qldpc_total} physical qubits")
    print(f"  Reduction factor: {n_surface_total / n_qldpc_total:.1f}x")


# Test: Overhead comparison
if __name__ == "__main__":
    compare_overhead(k_logical=100)
```

—

0.3.6 Phase 6: Certificate Generation (Months 9-10)

Goal: Export QLDPC codes with complete certificates for verification.

```
import json
import h5py
from scipy.sparse import save_npz, load_npz

def export_qldpc_certificate(H_X: np.ndarray, H_Z: np.ndarray,
   code_params: dict,
                             threshold_data: dict, output_file: str):
    """
    Export QLDPC code certificate.
    """
    # Verify commutation
    assert verify_css_commutation(H_X, H_Z), "Stabilizers do not commute
       !"

    # Sparsity
    ldpc_X = check_ldpc_sparsity(H_X, w_row_max=20, w_col_max=20)
    ldpc_Z = check_ldpc_sparsity(H_Z, w_row_max=20, w_col_max=20)

    certificate = {
        'code_parameters': code_params,  # [[n, k, d]]
        'stabilizer_dimensions': {
            'H_X_shape': H_X.shape,
            'H_Z_shape': H_Z.shape
        },
        'ldpc_properties': {
            'is_ldpc': ldpc_X and ldpc_Z,
            'max_row_weight_X': int(np.max(np.sum(H_X, axis=1))),
            'max_row_weight_Z': int(np.max(np.sum(H_Z, axis=1))),
            'max_col_weight_X': int(np.max(np.sum(H_X, axis=0))),
```

```
            'max_col_weight_Z': int(np.max(np.sum(H_Z, axis=0)))
        },
        'commutation_verified': True,
        'threshold': {
            'p_threshold_estimate': float(threshold_data.get('p_th', 0))
                ,
            'simulation_trials': threshold_data.get('num_trials', 0)
        },
        'certificate_version': '1.0'
    }

    # JSON certificate
    with open(output_file + '.json', 'w') as f:
        json.dump(certificate, f, indent=2)

    # HDF5 for matrices
    with h5py.File(output_file + '.h5', 'w') as f:
        f.create_dataset('H_X', data=H_X, compression='gzip')
        f.create_dataset('H_Z', data=H_Z, compression='gzip')

        if 'p_values' in threshold_data:
            f.create_dataset('threshold/p_values', data=threshold_data['
                p_values'])
            f.create_dataset('threshold/logical_error_rates', data=
                threshold_data['logical_error_rates'])

    print(f"    Certificate exported to {output_file}.json and {
        output_file}.h5")


def verify_qldpc_certificate(cert_file: str):
    """
    Independent verification of QLDPC certificate.
    """
    with open(cert_file + '.json', 'r') as f:
        cert = json.load(f)

    print("=== QLDPC Certificate Verification ===\n")

    # Load matrices
    with h5py.File(cert_file + '.h5', 'r') as f:
        H_X = GF2(f['H_X'][:])
        H_Z = GF2(f['H_Z'][:])

    # 1. Verify commutation
    commutes = verify_css_commutation(H_X, H_Z)
    assert commutes, "Stabilizers do not commute!"
    print("    CSS commutation verified")

    # 2. Verify QLDPC
    is_ldpc = cert['ldpc_properties']['is_ldpc']
    assert is_ldpc, "Code is not QLDPC!"
    print(f"    QLDPC verified (max row weight: {cert['ldpc_properties
        ']['max_row_weight_X']})")

    # 3. Verify parameters
    n_claimed = cert['code_parameters']['n']
    k_claimed = cert['code_parameters']['k']
```

```
    n_actual = H_X.shape[1]
    k_actual = compute_logical_dimension(H_X, H_Z, n_actual)

    assert n_claimed == n_actual, "Code length mismatch!"
    assert k_claimed == k_actual, "Logical dimension mismatch!"
    print(f"    Code parameters verified: [[{n_actual}, {k_actual},  d
       {cert['code_parameters']['d']}]]")

    print("\n=== ALL VERIFICATIONS PASSED ===")
    return True
```

—

0.4 4. Example Starting Prompt

```
You are a quantum information theorist designing QLDPC codes for fault-
   tolerant quantum computing.
Your task is to construct explicit codes with constant rate and distance
    using ONLY graph theory,
coding theory, and linear algebra over GF(2)     NO empirical tuning or
   machine learning.

OBJECTIVE: Construct a family of [[n,   (n),   ( n )]] QLDPC codes
   using hypergraph product,
achieving constant rate k/n     0.1 and scalable distance d      n .

PHASE 1 (Months 1-2): Classical LDPC Foundation
- Implement (w_row, w_col)-regular LDPC code generator
- Test (3,6)-regular code for n=120: verify sparsity, girth     6
- Implement BP decoder, measure threshold p_th     0.03 for classical
   channel
- Verify dual code construction: G H^T = 0

PHASE 2 (Months 2-4): CSS Code Construction
- Implement CSS construction from two classical codes  C  ,  C
- Verify CSS condition:  C  ^perp      C   (check  H    C  ^T = 0)
- Test on Steane [[7,1,3]] code: reproduce known parameters
- Verify stabilizer commutation: H_X H_Z^T = 0 (GF(2))

PHASE 3 (Months 4-5): Hypergraph Product
- Implement HP construction: H_X = [  H  I   |   I  H  ^T], H_Z = [
    I  H   |  H  ^ T I ]
- Test on two [7,4,3] Hamming codes     [[90,  k 16 ,  d 3 ]] QLDPC
- Verify LDPC: row weights     10, column weights     10
- Scale to n=1000: use (3,6)-regular LDPC inputs, measure k/n ratio

PHASE 4 (Months 5-7): BP Decoding for QLDPC
- Adapt classical BP to quantum setting: separate X and Z syndromes
- Implement LLR message passing on Tanner graph
- Simulate X-errors at p = 0.01, 0.02, ..., 0.10
- Measure logical error rate p_L(p) over 1000 trials per p
- Estimate threshold: p_th where p_L crosses p (expect p_th     1-2% for
    HP codes)

PHASE 5 (Months 7-9): Asymptotically Good Codes
- Implement balanced product (if generators available) or fiber bundle
   construction
```

```
- Target: [[n, 0.1n, 0.01n]] for n=1000, 5000, 10000
- Compute rate k/n and relative distance d/n
- Compare to surface codes: overhead reduction factor

PHASE 6 (Months 9-10): Certificate and Validation
- Export H_X, H_Z as sparse matrices (CSR format)
- Generate certificate: commutation check, LDPC verification, distance
    bounds
- Independent verification script: reload matrices, recompute parameters
- Comparison to literature: [[90,8,6]] HP code, [[900,64,12]] balanced
    product
- Threshold plot: p_L vs p for n=100, 500, 1000 (scaling analysis)

SUCCESS CRITERIA:
- **MVR (Months 4-5)**: [[90,  k 16 ,  d 3 ]] HP code verified, BP
    decoder functional
- **Strong (Months 7-8)**: [[1000,   100  ,   10  ]] family constructed,
    threshold p_th > 1%,
   certificate exported and independently verified
- **Publication (Months 9-10)**: [[10000, 1000, 100]] code achieved,
    overhead vs surface
   code: 100x reduction demonstrated, novel balanced product variant

VERIFICATION PROTOCOL:
1. Commutation: H_X H_Z^T = 0 for all codes (GF(2))
2. LDPC: max row/col weight     20 for all codes
3. Distance: exhaust search for n<20, LP bound for  n 20 , compare to
     d min ( d   , d   ) (HP)
4. Threshold: p_th > 0.01 (better than repetition code threshold ~0.003)

5. Literature: [[90,8,6]] matches Tillich-Z mor Table II

EXPORT:
- `qldpc_code_n1000.json`: Certificate with [[n, k, d]] and LDPC
    properties
- `qldpc_code_n1000.h5`: Sparse H_X, H_Z matrices, threshold data
- `qldpc_decoder.py`: Reusable BP decoder module

This is a PURE THOUGHT challenge: use ONLY coding theory and graph
    combinatorics.
NO empirical neural decoders, NO hardware-specific optimizations until
    validation.
```

—

0.5 5. Success Criteria

0.5.1 Minimum Viable Result (MVR) — Months 4-5

Deliverable: Working hypergraph product code with BP decoder.

Specific Metrics: 1. **Code Construction**: - [[90, k16, d3]] HP code from two [7,4,3] Hamming codes - Commutation verified: H_X H_$Z^T = 0 - QLDPCverified : maxrowweight$10

2. **Decoding**: - BP decoder converges for p ¡ 0.05 - Logical error rate measured over 100 trials

3. **Certificate**: - H_X, H_Z exported as sparse matrices - Parameters [[n, k, d]] verified independently

Output: JSON certificate + HDF5 matrices for [[90, ¿=16, ¿=3]] code.

—

0.5.2 Strong Result — Months 7-8

Deliverable: Scalable QLDPC family with measured threshold.

Specific Metrics: 1. **Code Family**: - [[1000, 100, 10]] code constructed via HP - Rate k/n 0.1, relative distance d/n 0.01

2. **Threshold**: - p_th ¿ 1% for X-errors (measured via 1000 trials) - Scaling: threshold improves with n

3. **Comparison**: - Surface code: $[[100^2, 1, 100]] \rightarrow$ 10,000 qubits for 1 logical - QLDPC: [[1000, 100, 30]] $\rightarrow$ 1000 qubits for 100 logical (100x improvement!)

Certificate: Full database of codes for n=100, 500, 1000 with threshold curves.

—

0.5.3 Publication-Quality Result — Months 9-10

Deliverable: Novel QLDPC constructions, comprehensive benchmarking, formal verification.

Specific Metrics: 1. **Novel Contribution**: - [[10000, 1000, 100]] code via balanced product - Rate k/n = 0.1, distance d/n = 0.01 (approaching asymptotic bounds)

2. **Threshold Analysis**: - p_th ¿ 2% for optimized BP decoder - Comparison to surface code: 2× higher threshold

3. **Validation**: - Distance verified via LP bound + random sampling - Independent code from literature reproduced (Panteleev-Kalachev [[900,64,12]]) - Formal certificate: commutation, LDPC, distance all machine-checked

4. **Publication Targets**: - *Quantum* (open access, quantum information) - *IEEE Transactions on Information Theory* (coding theory) - *Physical Review X Quantum* (quantum computing)

Certificate: Database of 10+ codes with [[n, k, d]] ranging from [[90, 16, 3]] to [[10000, 1000, 100]].

—

0.6 6. Verification Protocol

0.6.1 Automated Checks (Run After Each Phase)

```
def verify_qldpc_implementation():
    """
    Comprehensive QLDPC verification suite.
    """
    print("=== QLDPC Verification Suite ===\n")

    # 1. Classical LDPC
    print("1. Testing Classical LDPC")
    test_ldpc_generation()

    # 2. CSS construction
    print("\n2. Testing CSS Codes")
    test_css_code()

    # 3. Hypergraph product
    print("\n3. Testing Hypergraph Product")
    test_hypergraph_product()

    # 4. BP decoding
    print("\n4. Testing BP Decoder")
    # (Run small threshold sim)
```

```
    print("\n=== ALL TESTS PASSED ===")
```

Manual Checks: 1. Compare to literature codes (Tillich-Zémor, Breuckmann-Eberhardt tables) 2. Plot Tanner graph for small codes — verify bipartite structure 3. Threshold curves: p_L vs p should cross at p_th

—

0.7 7. Resources and Milestones

0.7.1 Essential References

Classical LDPC: - Gallager, R. G. (1962). "Low-density parity-check codes." *IRE Transactions on Information Theory* 8(1): 21-28. - MacKay, D. J. C. (1999). "Good error-correcting codes based on very sparse matrices." *IEEE Trans. Inform. Theory* 45(2): 399-431.

Quantum LDPC: - Tillich, J.-P., & Zémor, G. (2014). "Quantum LDPC codes with positive rate and minimum distance proportional to the square root of the blocklength." *IEEE Trans. Inform. Theory* 60(2): 1193-1202. - Panteleev, P., & Kalachev, G. (2022). "Asymptotically good quantum and locally testable classical LDPC codes." *STOC 2022*.

Good QLDPC Constructions: - Breuckmann, N. P., & Eberhardt, J. N. (2021). "Balanced product quantum codes." *IEEE Trans. Inform. Theory* 67(10): 6653-6674. - Hastings, M. B., Haah, J., & O'Donnell, R. (2021). "Fiber bundle codes: Breaking the $n^{1/2}$ $lognbarrierforquantumLDPC$ $STOC2021*$.

Decoding: - Roffe, J., et al. (2020). "Decoding across the quantum LDPC code landscape." *Physical Review Research* 2(4): 043423.

Software: - qLDPC (Python library), LDPC (MATLAB), Stim (fast stabilizer simulator)

0.7.2 Milestone Checklist

Month 1-2: - [ ] Classical (3,6)-regular LDPC code generator working - [ ] Girth computation via NetworkX - [ ] Dual code construction: $C^perpfromG$

Month 3-4: - [ ] CSS construction verified on Steane [[7,1,3]] - [ ] Commutation check: $H_X H_Z^T = 0automated$

Month 5-6: - [ ] Hypergraph product: [[90, 16, 3]] reproduced - [ ] Scaling to n=1000 successful - [ ] QLDPC verification: sparsity confirmed

Month 7-8: - [ ] BP decoder implemented - [ ] Threshold simulation: p_th measured for n=100, 500, 1000 - [ ] Threshold ¿ 1% achieved

Month 9-10: - [ ] Balanced product / fiber bundle code constructed - [ ] [[10000, 1000, 100]] code achieved - [ ] Certificate exported and verified independently - [ ] Comparison to surface codes documented - [ ] Draft paper prepared

0.7.3 Common Pitfalls

1. **CSS Condition**: Forgetting to check $C^perpCleadstonon-commutingstabilizers|codeisinvalid$!

2. **GF(2) Arithmetic**: Using regular integer arithmetic instead of modulo-2 causes incorrect rank/dual computations

3. **BP Convergence**: Poor girth (cycles of length 4) causes BP to oscillate; ensure girth 6

4. **Distance Overestimation**: Hypergraph product gives d min(d, d) as **lower bound**, not exact value

5. **Threshold Definition**: Logical error rate crossing physical rate is necessary but not sufficient — must verify scaling with n

—

End of PRD 21