Logical Computation on QLDPC Codes through Surgery

With Case Study on Bivariate Bicycle codes

Andrew Cross, **Zhiyang He (Sunny)**, Tomas Jochym-O'Connor, Patrick Rall, **Esha Swaroop**, Dominic Williamson, Theodore Yoder

Outline

- Background and Motivation
    - Quantum LDPC Codes
    - Code Surgery Methods
- Auxiliary Graph Surgery on QLDPC Codes
    - Graph Desiderata
    - Universal Adapter for Joint-measurements
- Case Study: [[144,12,12]] Bivariate bicyclic code

Basics of QEC

Quantum error correction is a fundamental building block of large-scale quantum computation.

Basics of QEC

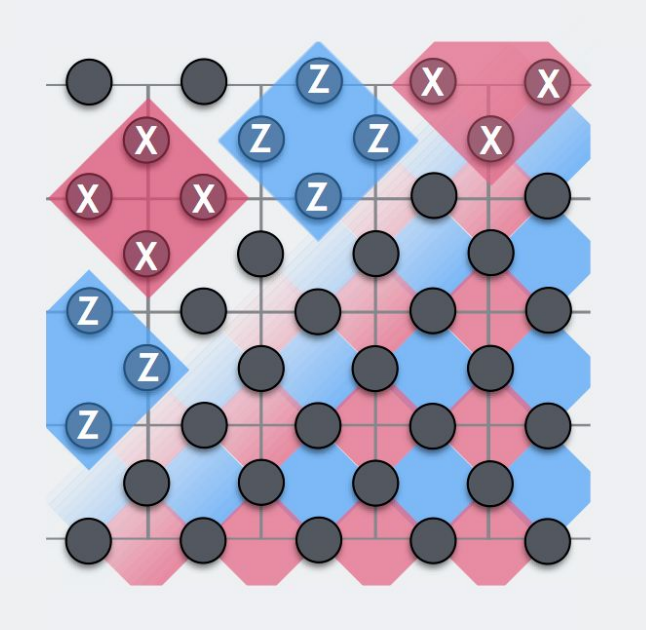
Quantum error correction is a fundamental building block of large-scale quantum computation.

One of the most promising codes: Surface Code.

1. Built on a 2D lattice of qubits.
2. Parameters [n= $2L^2$, k = 1, d = L].
3. Experimental demonstration of subthreshold scaling by Google*.

Challenge:

1. Significant space overhead (~1000x)!



* Quantum error correction below the surface code threshold [Google and collaborators (2024)]

** Surface code figure credit to Niel de Beaudrap.

Quantum LDPC Codes: the Tradeoff

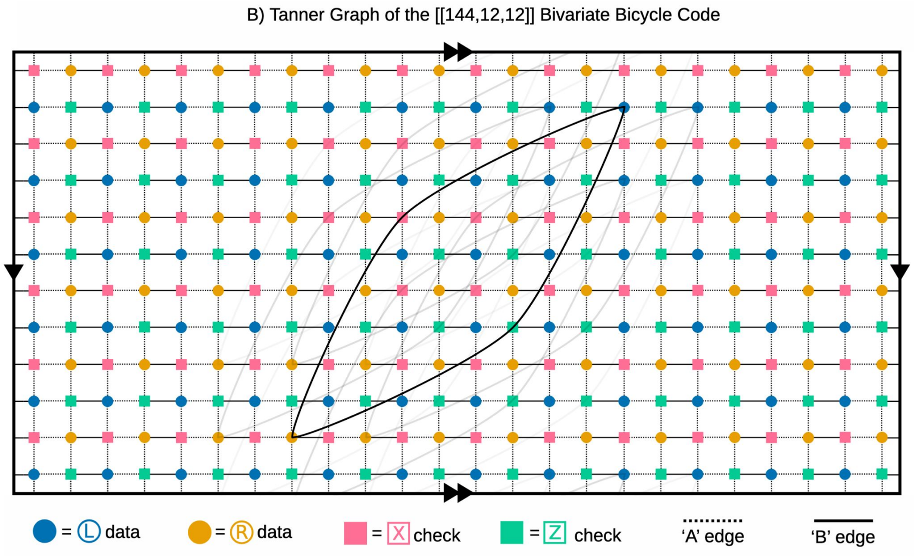
Quantum Low-Density Parity-Check (LDPC) Codes:
stabilizers of $O(1)$ weight, qubits in $O(1)$ stabilizers.
Better encoding rate than surface code!

Quantum LDPC Codes: the Tradeoff

Quantum Low-Density Parity-Check (LDPC) Codes: stabilizers of O(1) weight, qubits in O(1) stabilizers. Better encoding rate than surface code!

Practically: many codes with nice parameters, such as the quasi-cyclic lifted product codes and IBM's [n = 144, k = 12, d = 12] Bivariate Bicycle Code.*

Theoretically: Asymptotically good codes with k, d = O(n).**



* High-threshold and low-overhead fault-tolerant quantum memory [Bravyi et al 2023]
** Asymptotically Good Quantum and Locally Testable Classical LDPC Codes [Panteleev Kalachev 2021]

The Challenge: Logical Computation on Quantum LDPC Codes

QLDPC codes could be >10x more space-efficient than surface code. However,

how do we compute on the logical qubits?

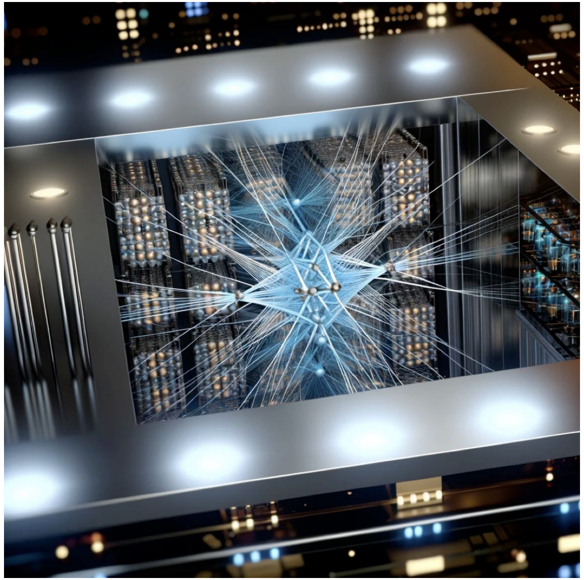
The Challenge: Logical Computation on Quantum LDPC Codes

QLDPC codes could be >10x more space-efficient than surface code. However,

how do we compute on the logical qubits?

For QLDPC codes, we need methods of performing logical computation that are:

1. Fault-tolerant;



GPT4 Impression of
"Quantum Hardware based on QLDPC codes".

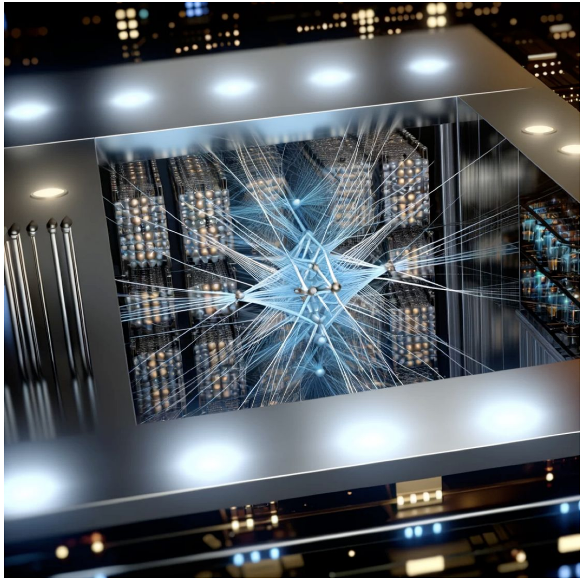
The Challenge: Logical Computation on Quantum LDPC Codes

QLDPC codes could be >10x more space-efficient than surface code. However,

how do we compute on the logical qubits?

For QLDPC codes, we need methods of performing logical computation that are:

1. Fault-tolerant;
2. Addressable: capable of addressing subsets of logical qubits in multi-qubit codeblocks;



GPT4 Impression of
“Quantum Hardware based on QLDPC codes”.

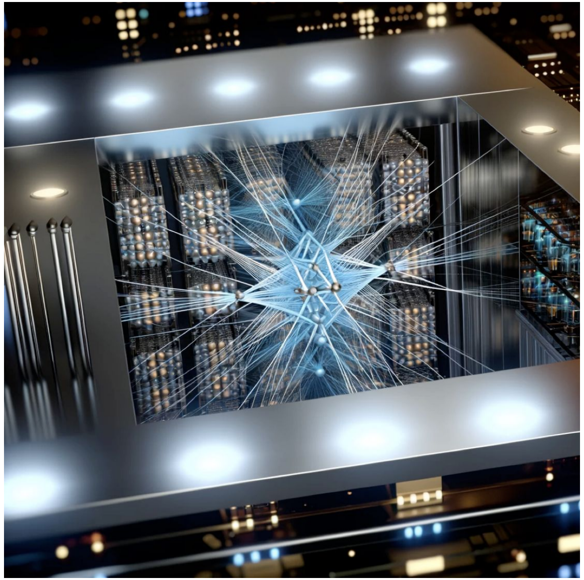
The Challenge: Logical Computation on Quantum LDPC Codes

QLDPC codes could be >10x more space-efficient than surface code. However,

how do we compute on the logical qubits?

For QLDPC codes, we need methods of performing logical computation that are:

1. Fault-tolerant;
2. Addressable: capable of addressing subsets of logical qubits in multi-qubit codeblocks;
3. Universal;



GPT4 Impression of
“Quantum Hardware based on QLDPC codes”.

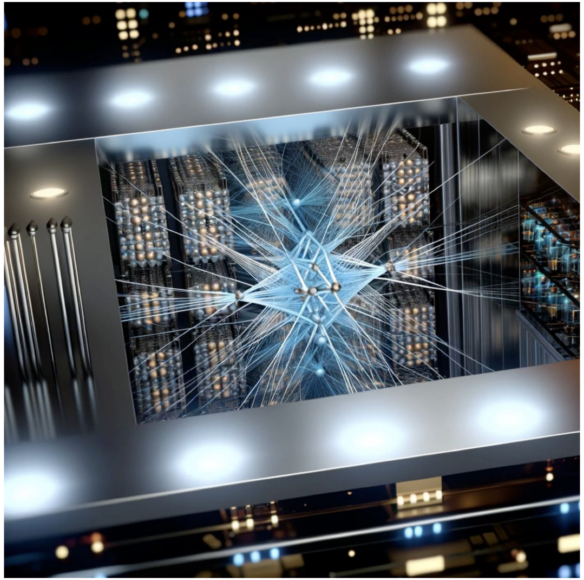
The Challenge: Logical Computation on Quantum LDPC Codes

QLDPC codes could be >10x more space-efficient than surface code. However,

how do we compute on the logical qubits?

For QLDPC codes, we need methods of performing logical computation that are:

1. Fault-tolerant;
2. Addressable: capable of addressing subsets of logical qubits in multi-qubit codeblocks;
3. Universal;
4. Low cost in space and time.



GPT4 Impression of
“Quantum Hardware based on QLDPC codes”.

The Challenge: Logical Computation on Quantum LDPC Codes

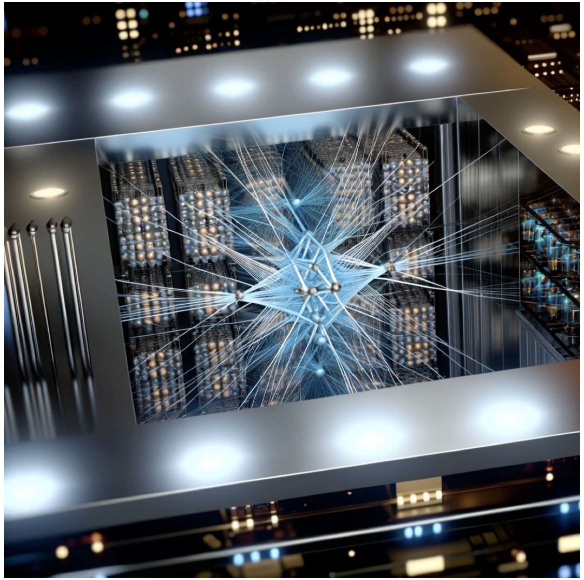
QLDPC codes could be >10x more space-efficient than surface code. However,

how do we compute on the logical qubits?

For QLDPC codes, we need methods of performing logical computation that are:

1. Fault-tolerant;
2. Addressable: capable of addressing subsets of logical qubits in multi-qubit codeblocks;
3. Universal;
4. Low cost in space and time.

This is the central problem in the study of QLDPC codes.



GPT4 Impression of
“Quantum Hardware based on QLDPC codes”.

Logical Computation on QLDPC Codes through Surgery

This talk: QLDPC surgery is a method of logical computation that is fault-tolerant, addressable, universal, and low-overhead. Work done collectively in three papers.

1. Improved QLDPC Surgery: Logical Measurements and Bridging Codes.
Andrew Cross, **Zhiyang He**, Patrick Rall, Theodore Yoder. 2407.18393

2. Low-overhead fault-tolerant quantum computation by gauging logical operators.
Dominic Williamson, Theodore Yoder. 2410.02213

3. Universal adapters between quantum LDPC codes.
Esha Swaroop, Tomas Jochym-O'Connor, Theodore Yoder. 2410.03628

Logical Computation on QLDPC Codes through Surgery

This talk: QLDPC surgery is a method of logical computation that is fault-tolerant, addressable, universal, and low-overhead. Work done collectively in three papers.

1. Improved QLDPC Surgery: Logical Measurements and Bridging Codes.
Andrew Cross, **Zhiyang He**, Patrick Rall, Theodore Yoder. 2407.18393

2. Low-overhead fault-tolerant quantum computation by gauging logical operators.
Dominic Williamson, Theodore Yoder. 2410.02213

3. Universal adapters between quantum LDPC codes.
Esha Swaroop, Tomas Jochym-O'Connor, Theodore Yoder. 2410.03628

Other related works: 2407.09423 (Cowtan), 2407.18490 (Xu et al.), 2408.01339 (Zhang, Li), 2410.02753 (Ide et al.).

Outline

- Background and Motivation
  - QLDPC Codes
  - Code Surgery Methods
- Auxiliary Graph Surgery on QLDPC Codes
  - Graph Desiderata
  - Universal Adapter for Joint-measurements
- Case Study: [[144,12,12]] Bivariate bicyclic code

Logical Measurement and Lattice Surgery

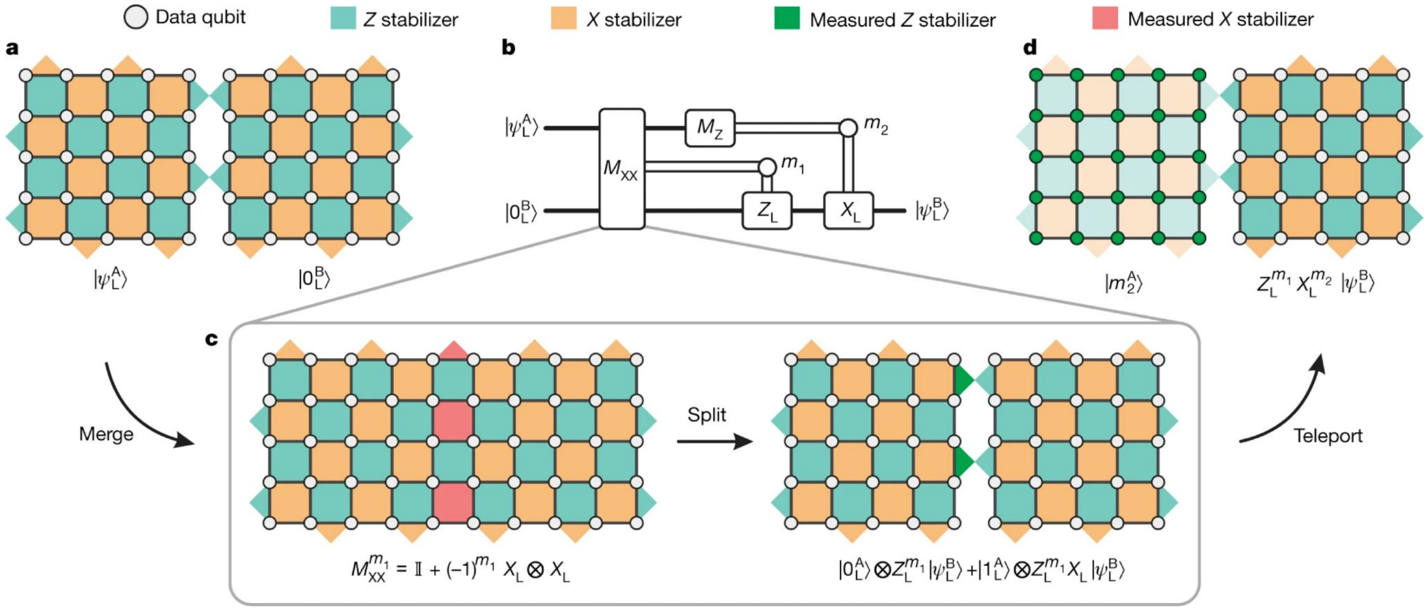
Pauli-based computation: Pauli measurements on logical qubits + magic states = universal computation.

Logical measurements on surface codes: lattice surgery.

Logical Measurement and Lattice Surgery

Pauli-based computation: Pauli measurements on logical qubits + magic states = universal computation.

Logical measurements on surface codes: lattice surgery.

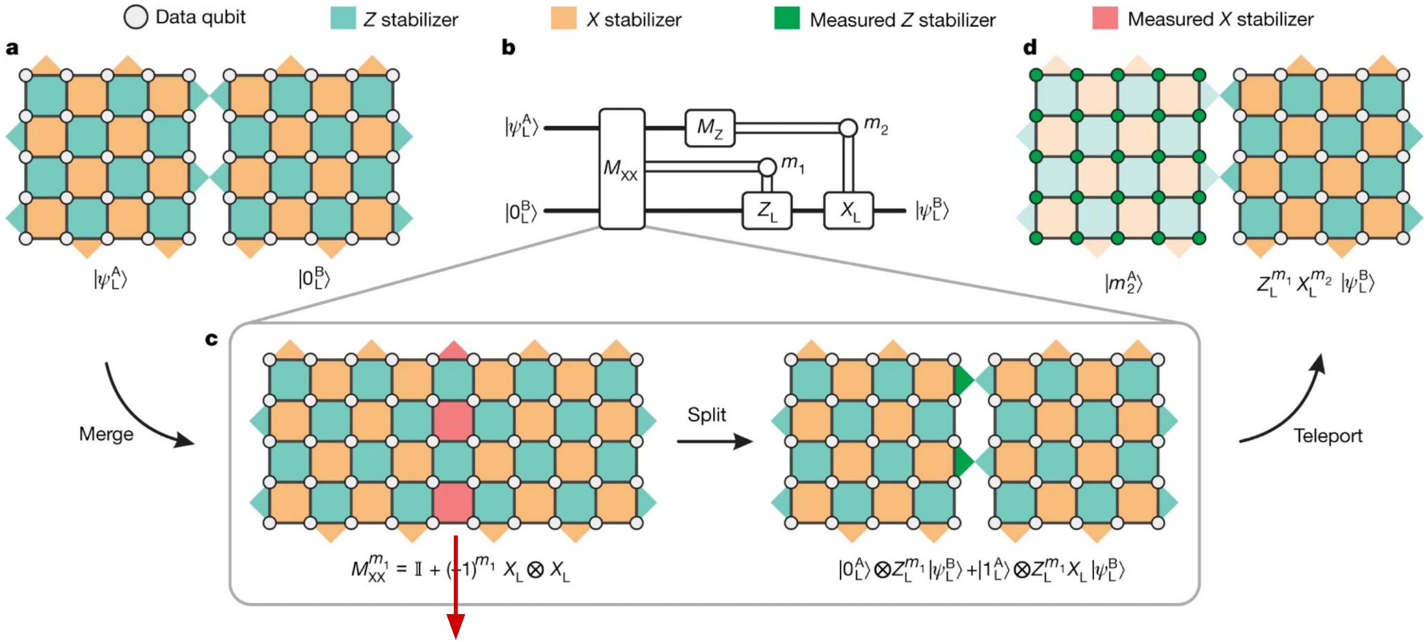


* Figure from Entangling logical qubits with lattice surgery [Erhard et al 2020].

Logical Measurement and Lattice Surgery

Pauli-based computation: Pauli measurements on logical qubits + magic states = universal computation.

Logical measurements on surface codes: lattice surgery.



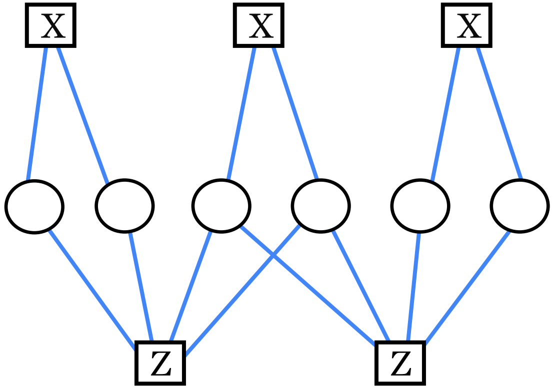
Product of red X-checks = $X_L \otimes X_L$ – obtain logical measurement result by measuring new stabilizers.

* Figure from Entangling logical qubits with lattice surgery [Erhard et al 2020].

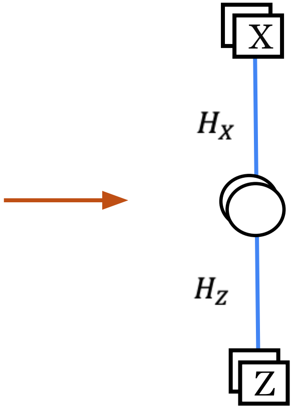
Quantum LDPC Surgery

Quantum LDPC Surgery

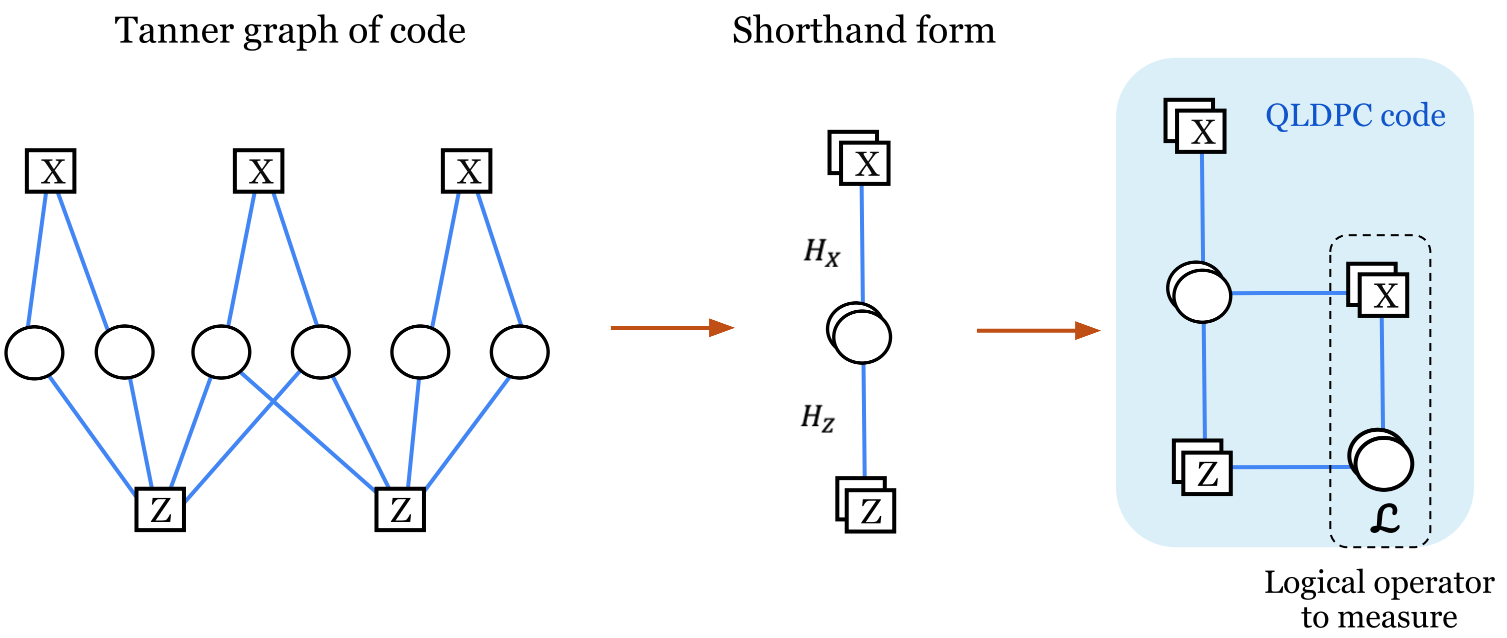
Tanner graph of code



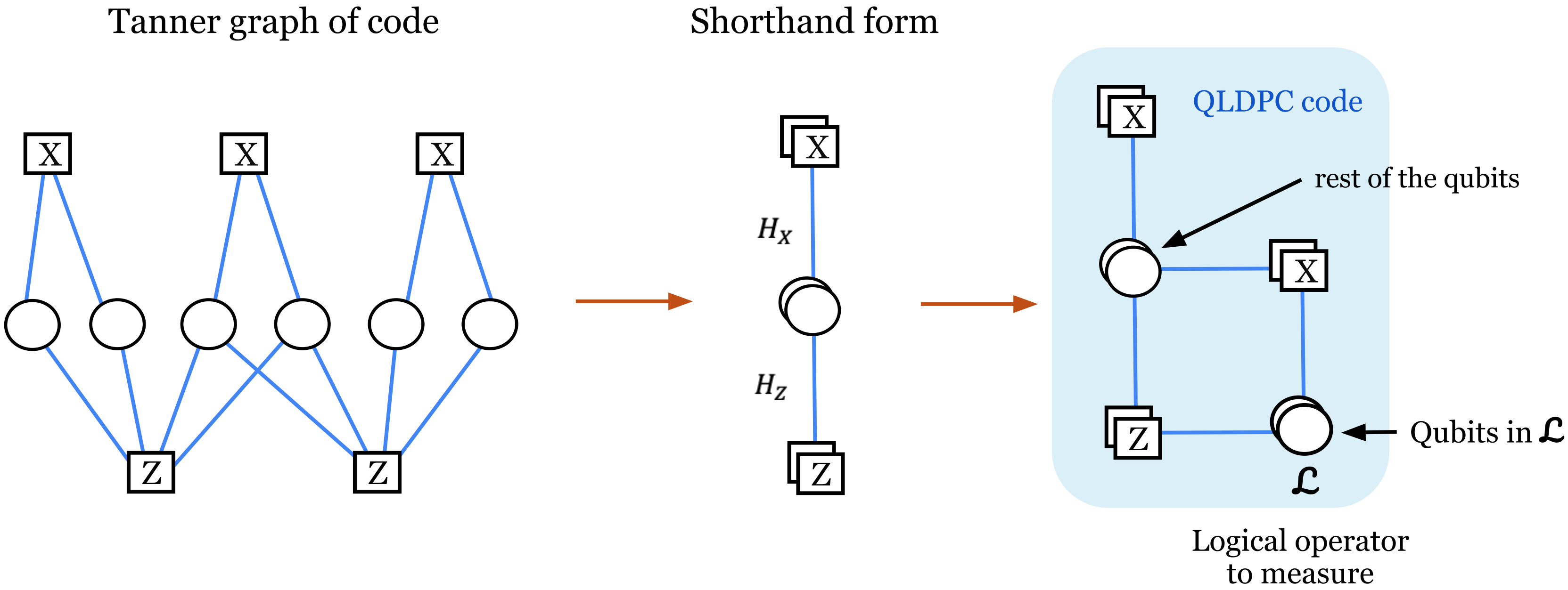
Shorthand form



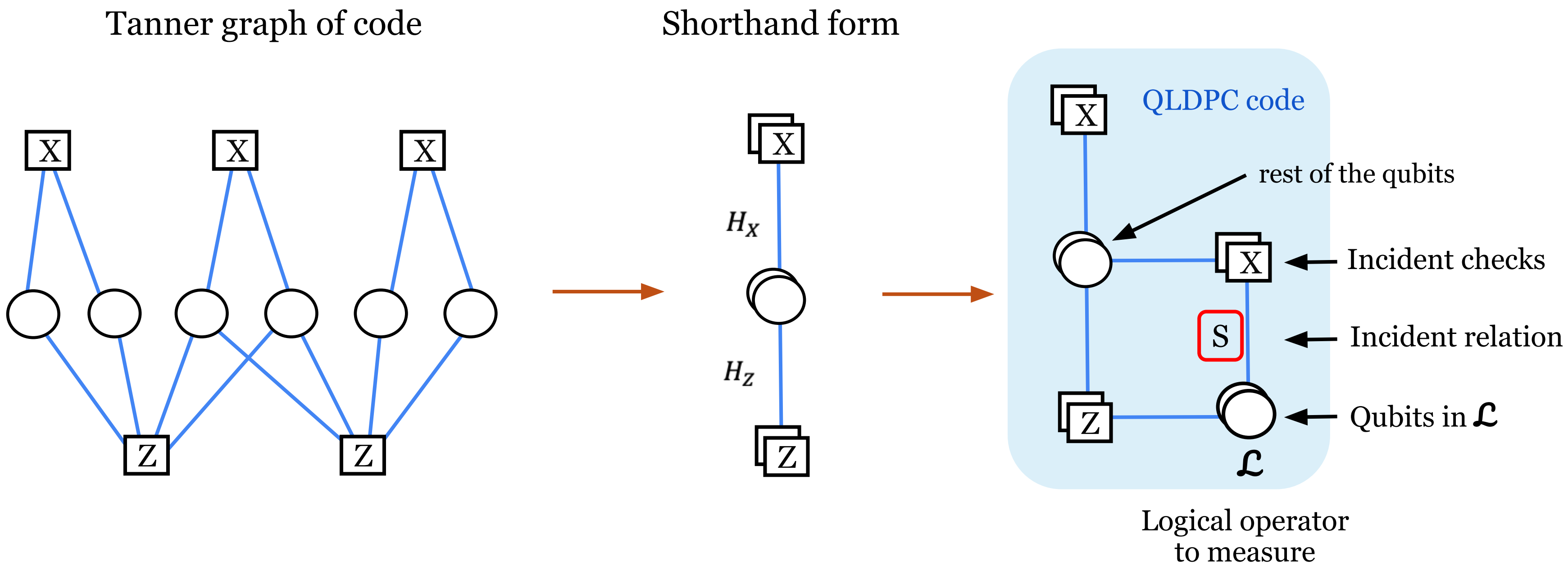
Quantum LDPC Surgery



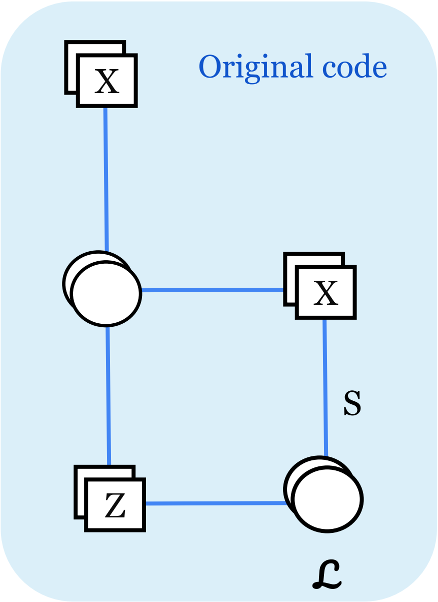
Quantum LDPC Surgery



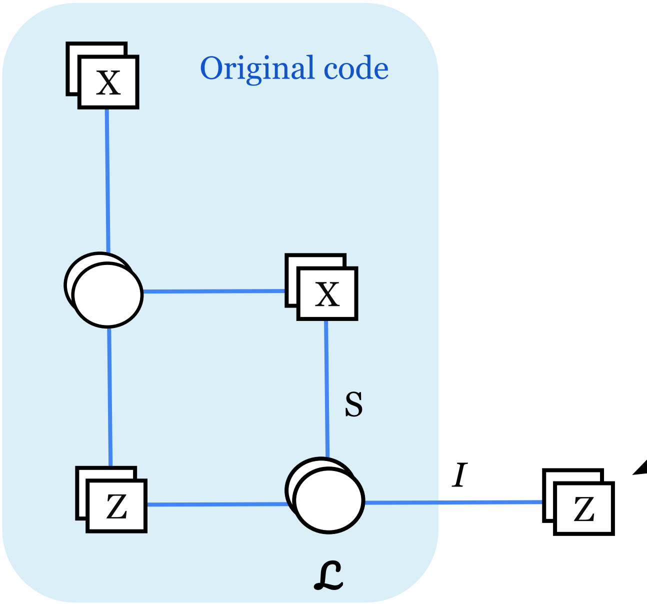
Quantum LDPC Surgery



CKBB Surgery Method [2110.10794]

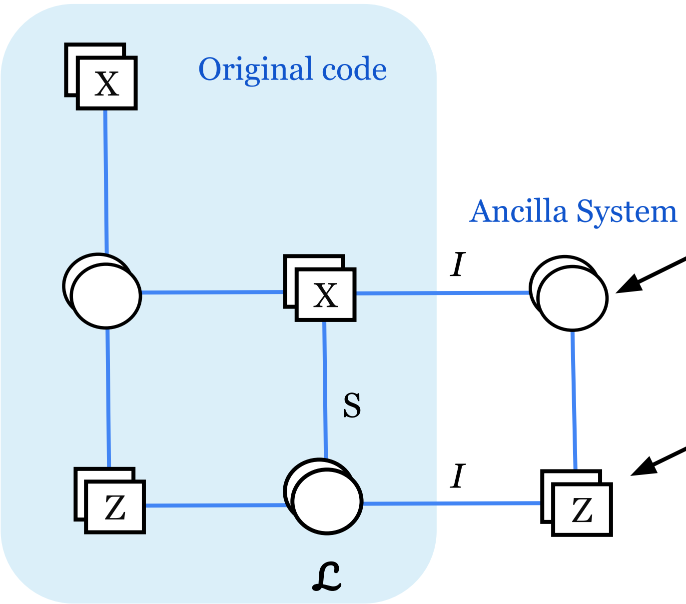


CKBB Surgery Method [2110.10794]



Create one ancilla $\boxed{Z}$ check for each qubit in $\mathcal{L}$.

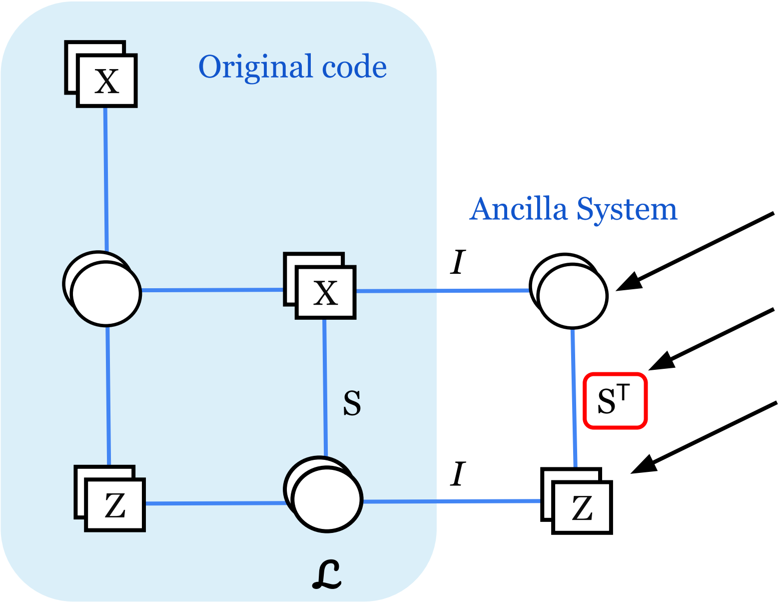
CKBB Surgery Method [2110.10794]



Add one ancilla qubit for each X check.

Create one ancilla Z check for each qubit in $\mathcal{L}$.

CKBB Surgery Method [2110.10794]



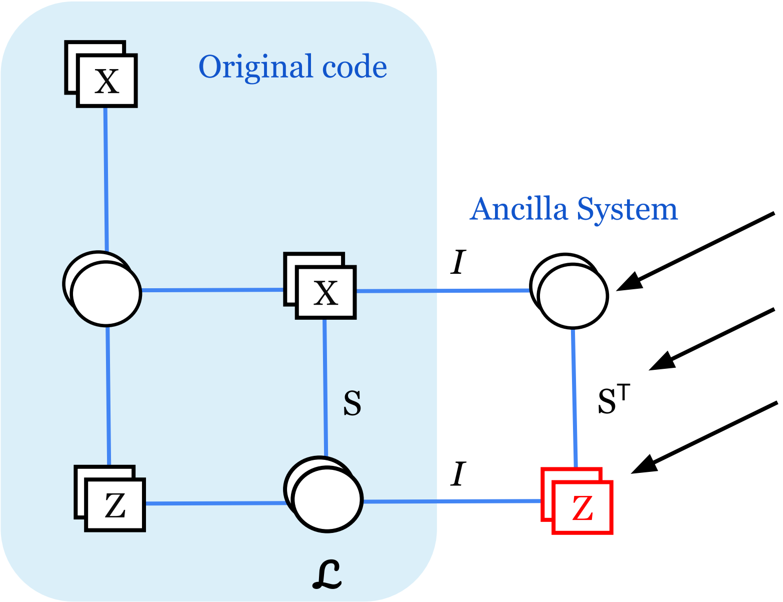
Add one ancilla qubit for each X check.

Same incidence relation as in $\mathcal{L}$.

Create one ancilla Z check for each qubit in $\mathcal{L}$.

Low-overhead fault-tolerant quantum computing using long-range connectivity; [Cohen, Kim, Bartlett, Brown 2021].

CKBB Surgery Method [2110.10794]



Add one ancilla qubit for each X check.

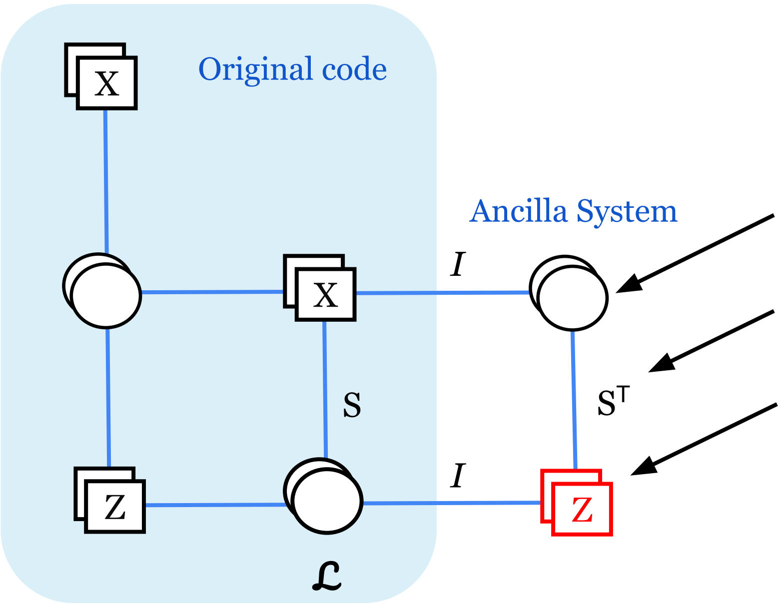
Same incidence relation as in $\mathcal{L}$.

Create one ancilla Z check for each qubit in $\mathcal{L}$.

Once we connect ancilla system to the code,

Product of ancilla Z checks = Z logical operator!

CKBB Surgery Method [2110.10794]



Are we done?

Add one ancilla qubit for each X check.

Same incidence relation as in $\mathcal{L}$.

Create one ancilla Z check for each qubit in $\mathcal{L}$.

Once we connect ancilla system to the code,

Product of ancilla Z checks = Z logical operator!

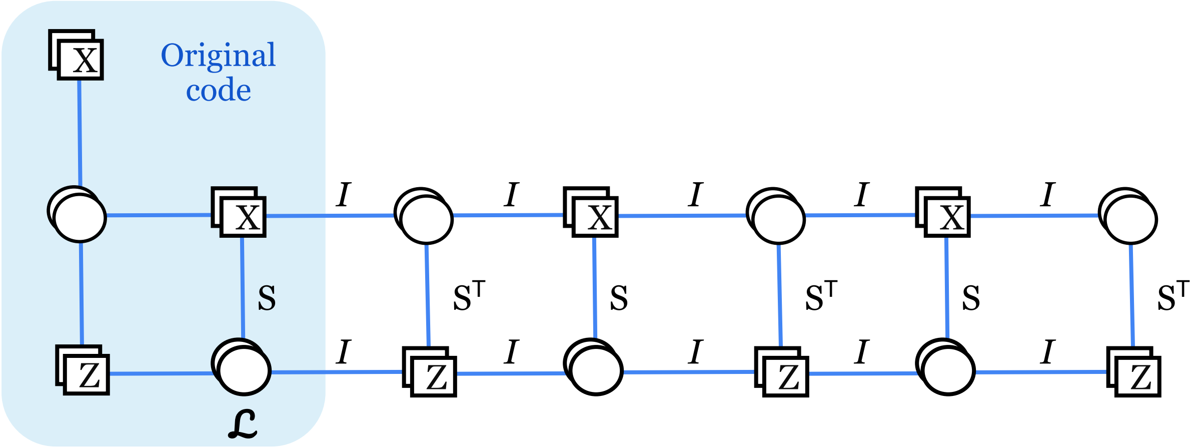
CKBB Surgery Method [2110.10794]

Problem: merged code has lower distance, less fault-tolerant.

CKBB Surgery Method [2110.10794]

Problem: merged code has lower distance, less fault-tolerant.

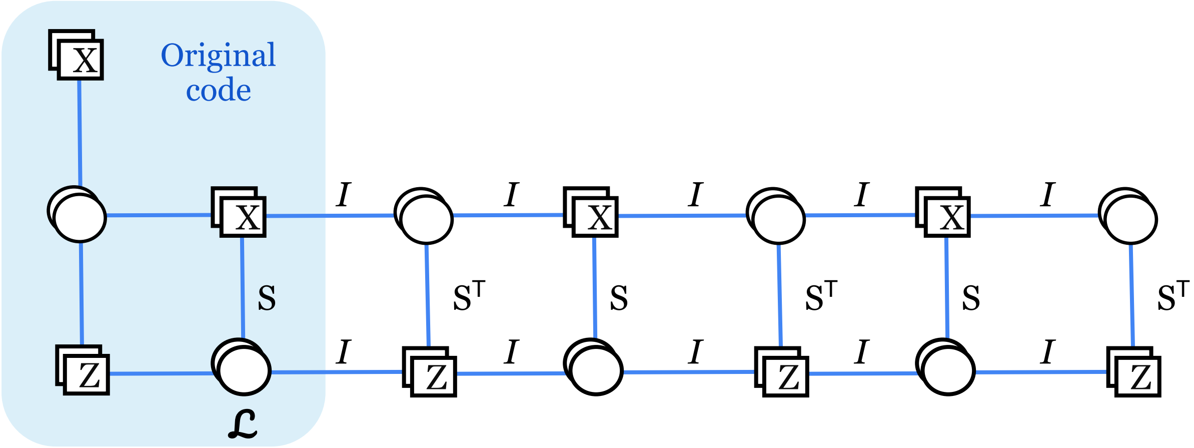
Solution: Repeat for O(d) layers.



CKBB Surgery Method [2110.10794]

Problem: merged code has lower distance, less fault-tolerant.

Solution: Repeat for O(d) layers.



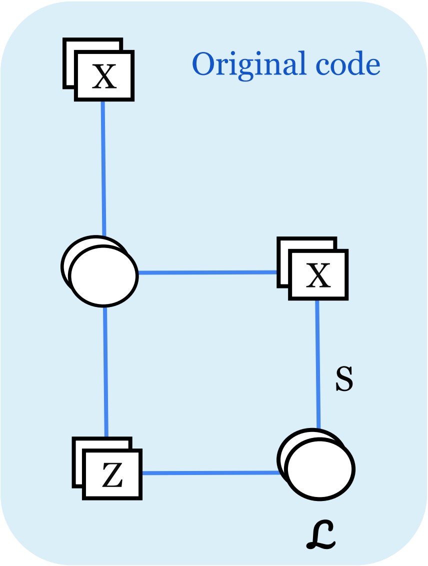
Advantage: applicable to any QLDPC code,

Issue: space overhead ~ $O(d^2)$, similar to surface code!

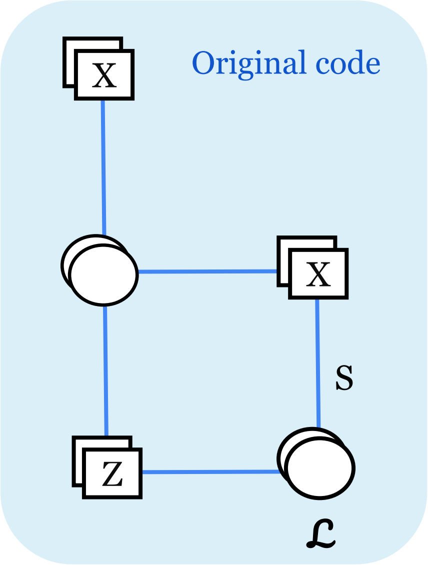
Outline

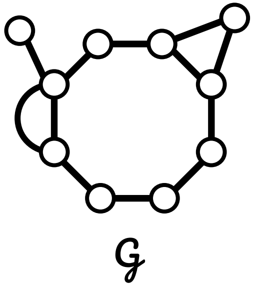
- Background and Motivation
    - Quantum LDPC Codes
    - Code Surgery Methods
- Auxiliary Graph Surgery on QLDPC Codes
    - Graph Desiderata
    - Universal Adapter for Joint-measurements
- Case Study: [[144,12,12]] Bivariate bicyclic code

New Approach: Auxiliary Graph Surgery



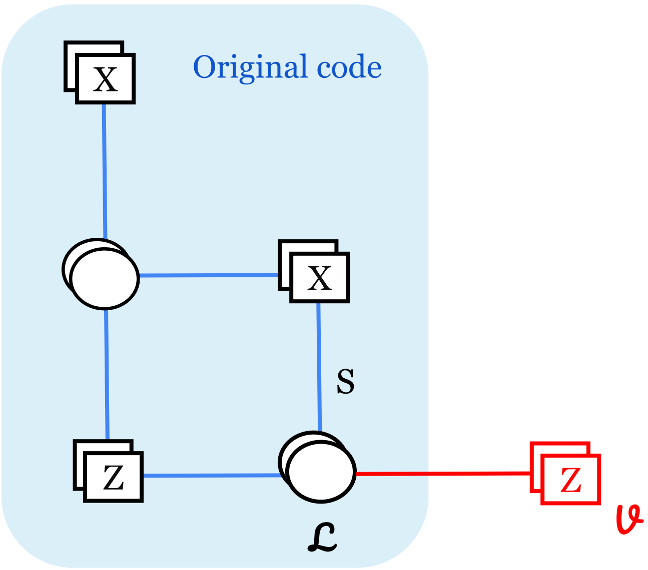
New Approach: Auxiliary Graph Surgery

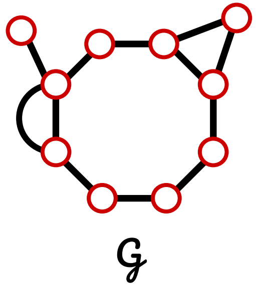




1. Build a customized graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, with one vertex per qubit in $\mathcal{L}$.

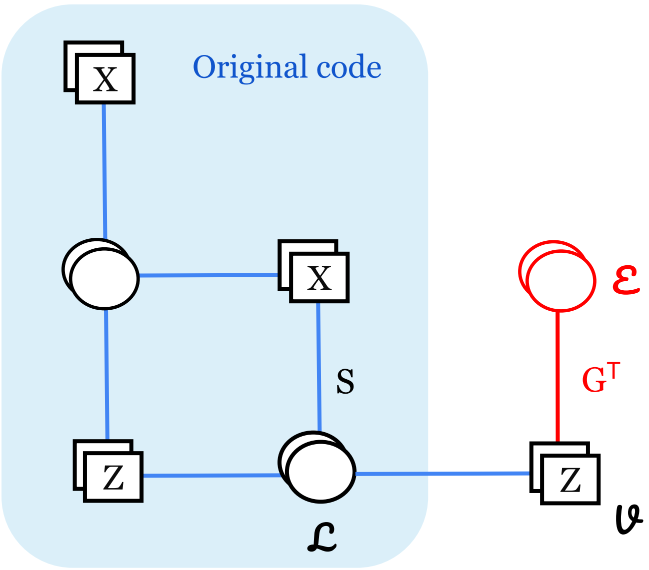
New Approach: Auxiliary Graph Surgery

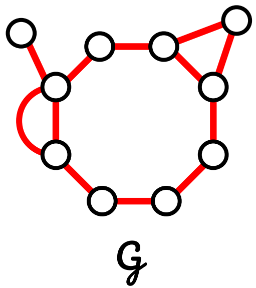




1. Build a customized graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, with one vertex per qubit in $\mathcal{L}$.
2. One new Z check for every vertex in $\mathcal{V}$.

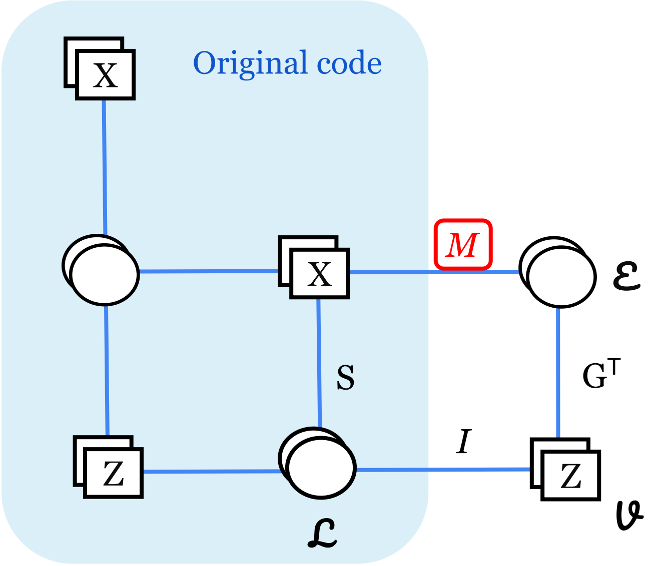
New Approach: Auxiliary Graph Surgery

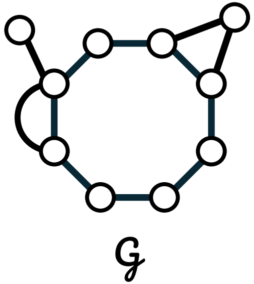




1. Build a customized graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, with one vertex per qubit in $\mathcal{L}$.
2. One new Z check for every vertex in $\mathcal{V}$.
3. One new ancilla qubit for every edge in $\mathcal{E}$.

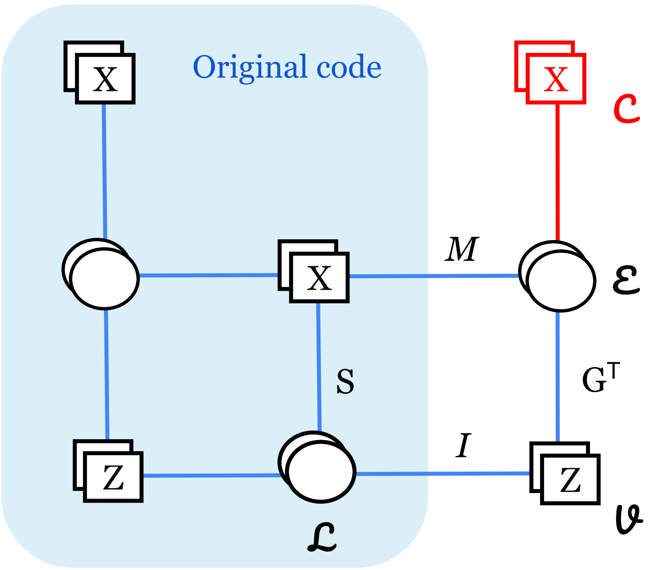
New Approach: Auxiliary Graph Surgery

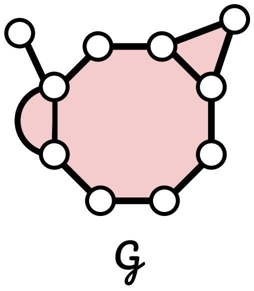




1. Build a customized graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, with one vertex per qubit in $\mathcal{L}$.
2. One new Z check for every vertex in $\mathcal{V}$.
3. One new ancilla qubit for every edge in $\mathcal{E}$.
4. Connect $\mathcal{E}$ qubits to X checks on $\mathcal{L}$.

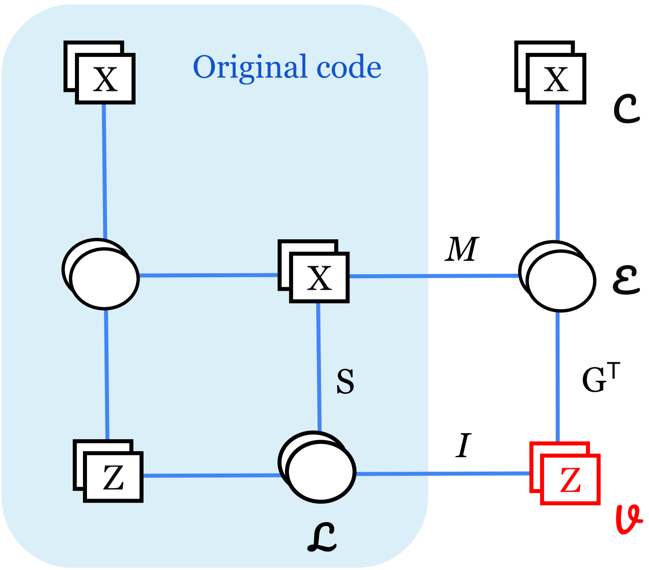
New Approach: Auxiliary Graph Surgery

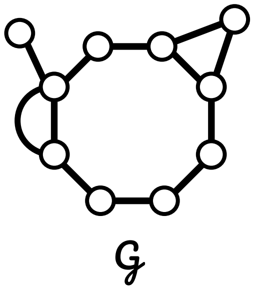




1. Build a customized graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, with one vertex per qubit in $\mathcal{L}$.
2. One new $\boxed{Z}$ check for every vertex in $\mathcal{V}$.
3. One new ancilla qubit for every edge in $\mathcal{E}$.
4. Connect $\mathcal{E}$ qubits to $\boxed{X}$ checks on $\mathcal{L}$.
5. Pick a cycle basis $\mathcal{C}$ of G. For each cycle basis element, introduce an $\boxed{X}$ check.

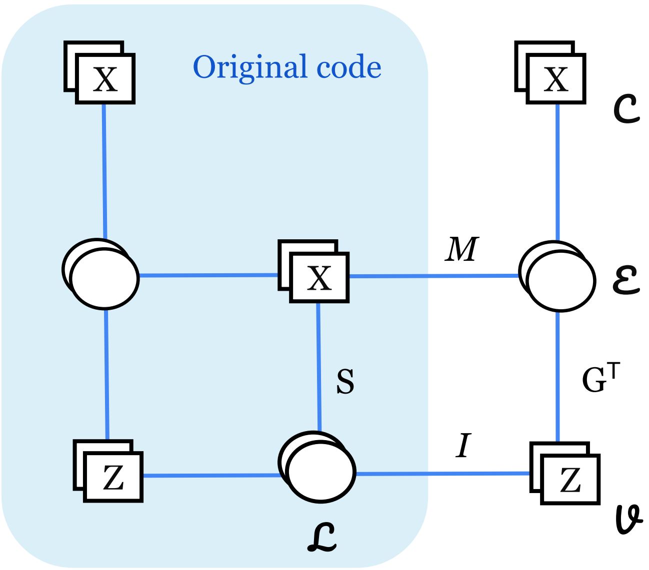
New Approach: Auxiliary Graph Surgery

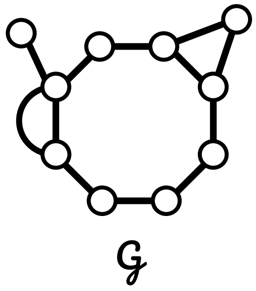




1. Build a customized graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, with one vertex per qubit in $\mathcal{L}$.
2. One new Z check for every vertex in $\mathcal{V}$.
3. One new ancilla qubit for every edge in $\mathcal{E}$.
4. Connect $\mathcal{E}$ qubits to X checks on $\mathcal{L}$.
5. Pick a cycle basis $\mathcal{C}$ of G. For each cycle basis element, introduce an X check.
6. Product of new Z checks = Z logical operator

New Approach: Auxiliary Graph Surgery

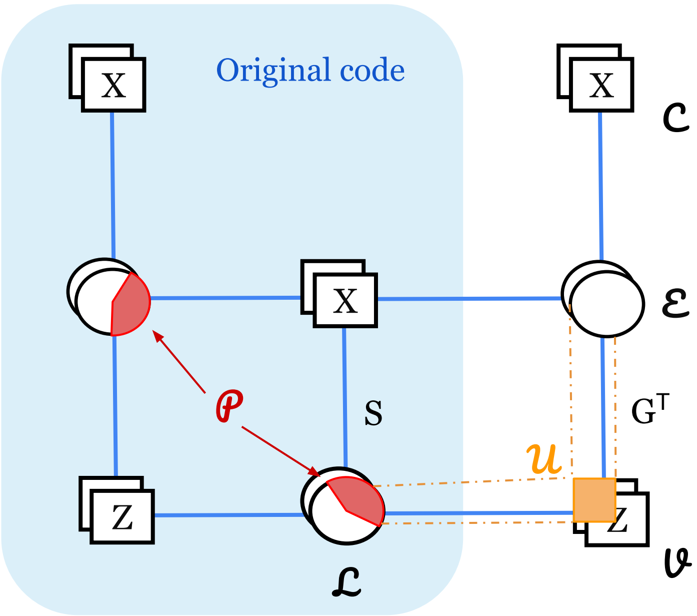




1. Build a customized graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, with one vertex per qubit in $\mathcal{L}$.
2. One new $\boxed{\text{Z}}$ check for every vertex in $\mathcal{V}$.
3. One new ancilla qubit for every edge in $\mathcal{E}$.
4. Connect $\mathcal{E}$ qubits to $\boxed{\text{X}}$ checks on $\mathcal{L}$.
5. Pick a cycle basis $\mathcal{C}$ of G. For each cycle basis element, introduce an $\boxed{\text{X}}$ check.
6. Product of new Z checks = Z logical operator

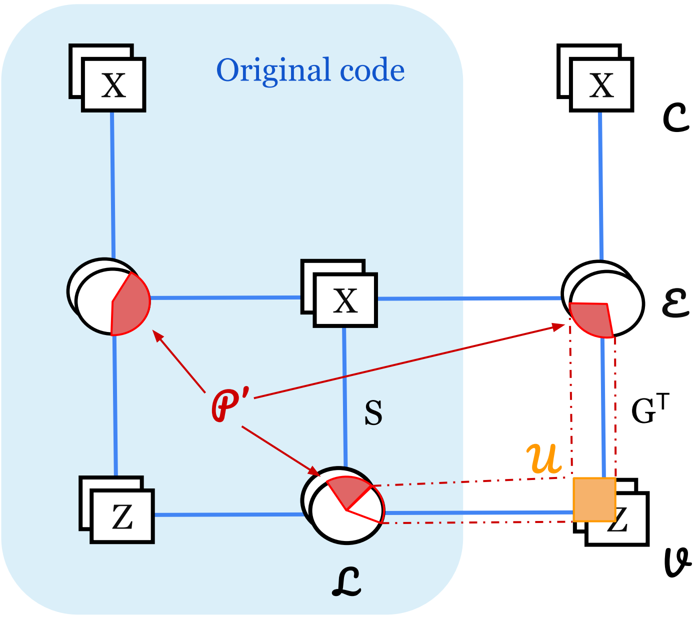
OK… what about code distance?

Expansion brings fault-tolerance



Let $\mathcal{P}$ be another Z operator, $\mathcal{U}$ be a set of ancilla checks,

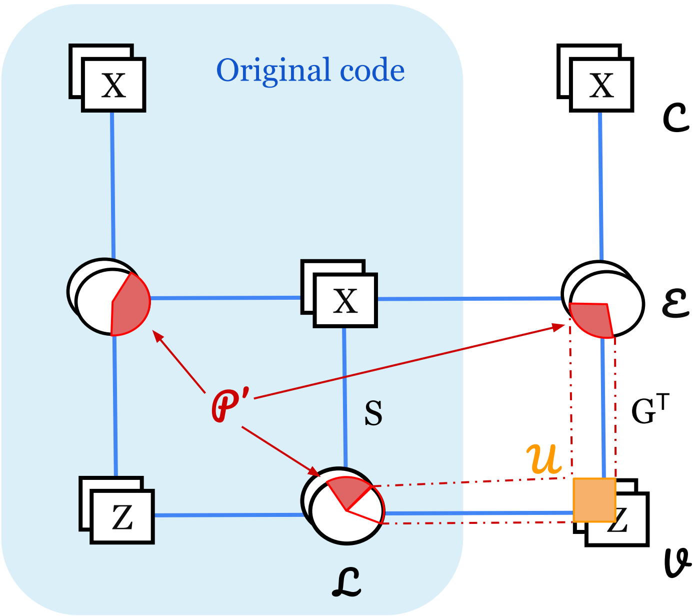
Expansion brings fault-tolerance



Let $\mathcal{P}$ be another Z operator, $\mathcal{U}$ be a set of ancilla checks,

$$\mathcal{P}' = \mathcal{P} \times \mathcal{U}G^T$$

Expansion brings fault-tolerance

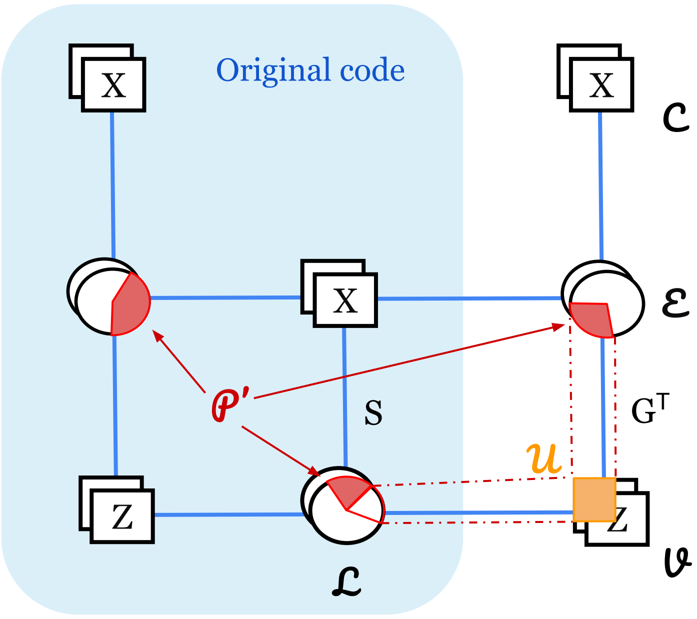


Let $\mathcal{P}$ be another Z operator, $\mathcal{U}$ be a set of ancilla checks,

$$\mathcal{P}' = \mathcal{P} \times \mathcal{U}G^T$$

Issue: $\mathcal{P}'$ may have lower weight than $\mathcal{P}$ → merged code has lower distance!

Expansion brings fault-tolerance



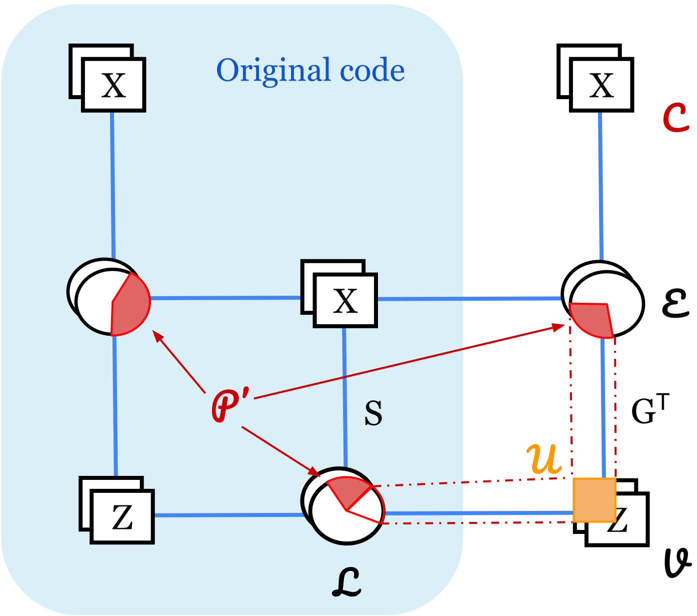
Let $\mathcal{P}$ be another Z operator, $\mathcal{U}$ be a set of ancilla checks,

$$\mathcal{P}' = \mathcal{P} \times \mathcal{U}G^T$$

Issue: $\mathcal{P}'$ may have lower weight than $\mathcal{P}$ → merged code has lower distance!

Solution: if $\mathcal{G}$ is expanding (large Cheeger constant), then $\mathcal{U}$ has large support $\mathcal{U}G^T$ in $\mathcal{E}$.

Expansion brings fault-tolerance



Let $\mathcal{P}$ be another Z operator, $\mathcal{U}$ be a set of ancilla checks,

$$\mathcal{P}' = \mathcal{P} \times \mathcal{U}G^T$$

Issue: $\mathcal{P}'$ may have lower weight than $\mathcal{P}$ → merged code has lower distance!

Solution: if $\mathcal{G}$ is expanding (large Cheeger constant), then $\mathcal{U}$ has large support $\mathcal{U}G^T$ in $\mathcal{E}$.

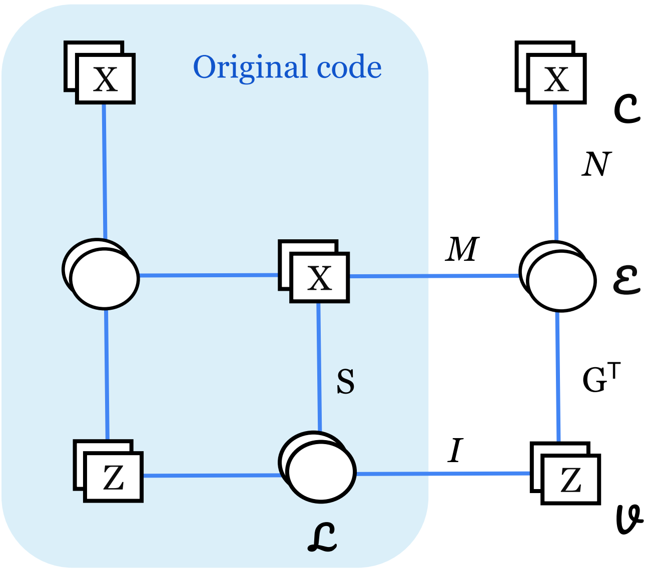
Other ways to reduce distance?

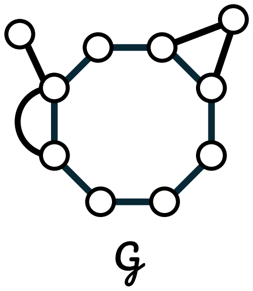
Not if we measure the cycles $\mathcal{C}$.

Outline

- Background and Motivation
  - Quantum LDPC Codes
  - Code Surgery Methods
- Auxiliary Graph Surgery on QLDPC Codes
  - Graph Desiderata
  - Universal Adapter for Joint-measurements
- Case Study: [[144,12,12]] Bivariate bicyclic code

Graph Desiderata





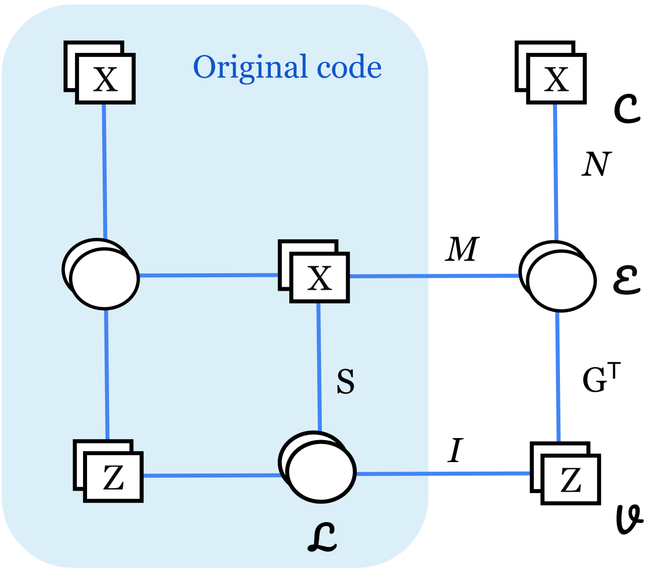
We want:

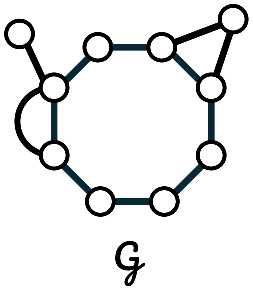
1. Code should preserve distance of original code.

Graph 𝒢 should have the following properties:

1. 𝒢 is expanding;

Graph Desiderata





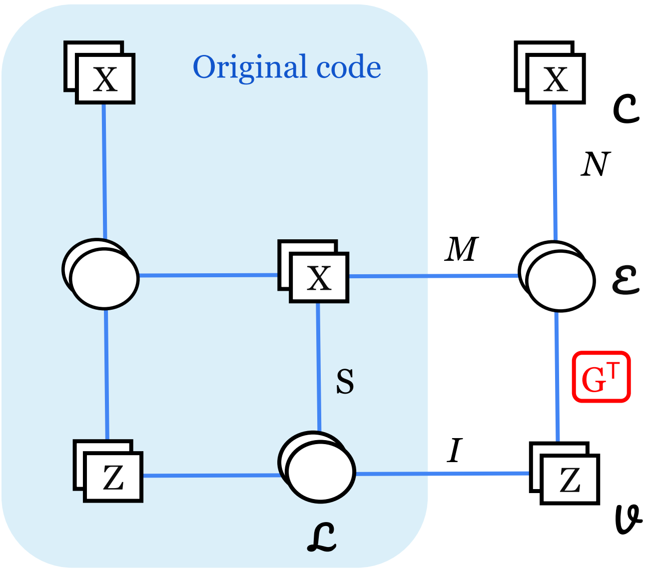
We want:

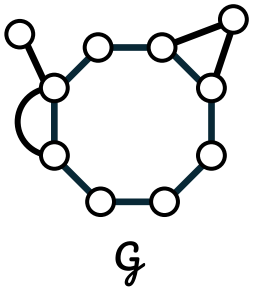
1. Code should preserve distance of original code.
2. Merged code should be LDPC at all stages

Graph $\mathcal{G}$ should have the following properties:

1. $\mathcal{G}$ is expanding;

Graph Desiderata





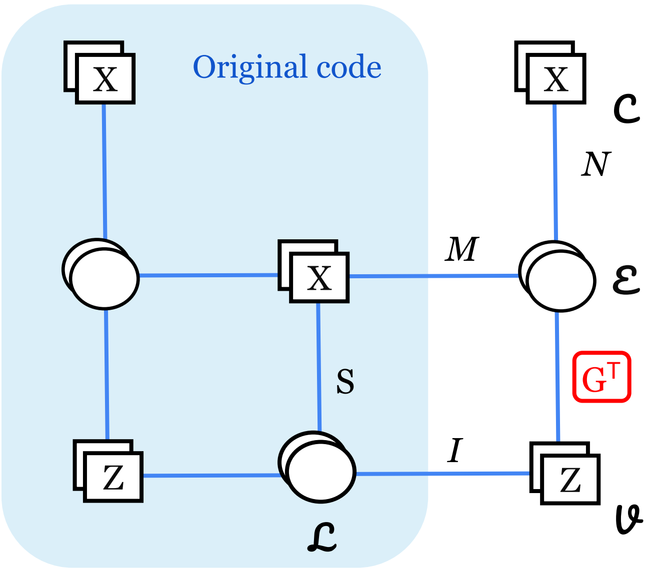
We want:

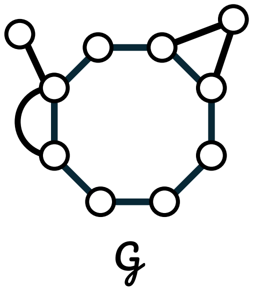
1. Code should preserve distance of original code.
2. Merged code should be LDPC at all stages

Graph $\mathcal{G}$ should have the following properties:

1. $\mathcal{G}$ is expanding;

Graph Desiderata





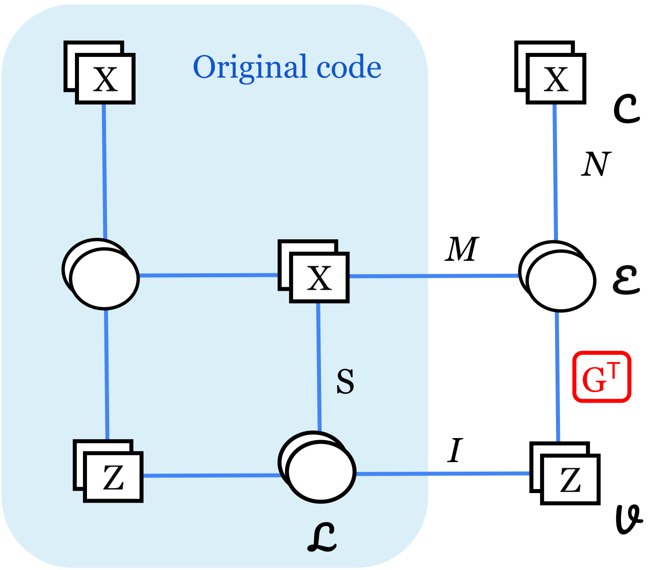
We want:

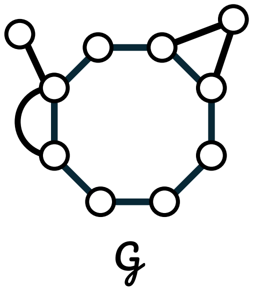
1. Code should preserve distance of original code.
2. Merged code should be LDPC at all stages

Graph $\mathcal{G}$ should have the following properties:

1. $\mathcal{G}$ is expanding;
2. All vertices have O(1) degree.

Graph Desiderata





We want:

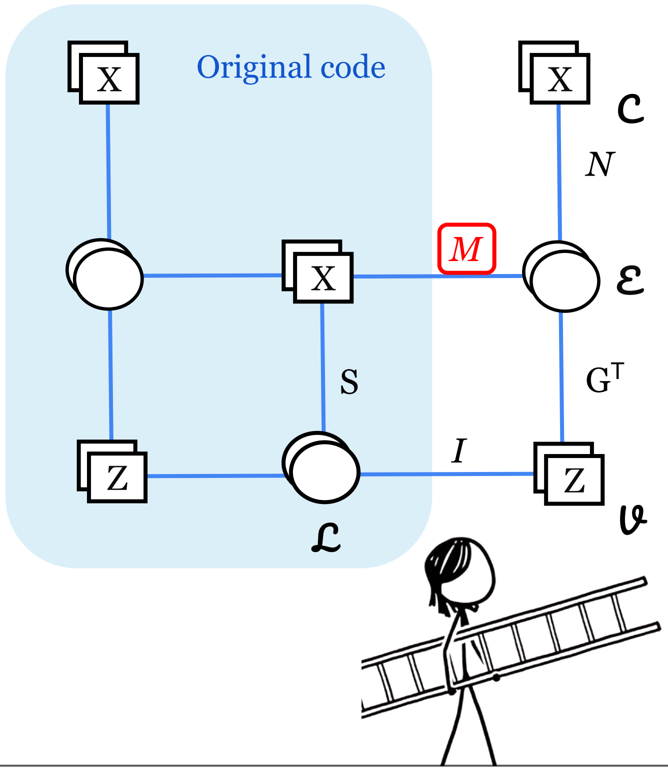
1. Code should preserve distance of original code.
2. Merged code should be LDPC at all stages

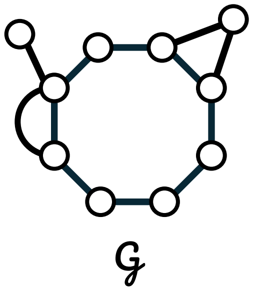
Graph 𝓖 should have the following properties:

1. 𝓖 is expanding;
2. All vertices have O(1) degree.

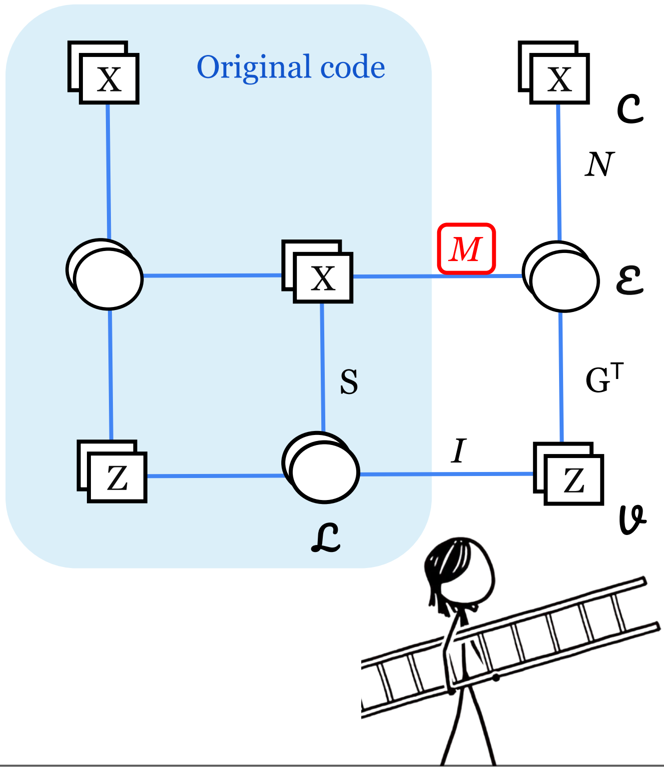
Ingredient #1: choose a randomly constructed constant-degree expander graph.

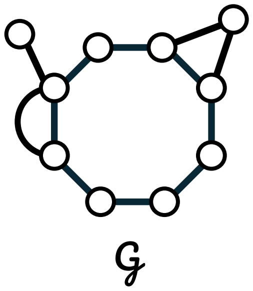
Desiderata 3: perfect matchings on graph $\mathcal{G}$





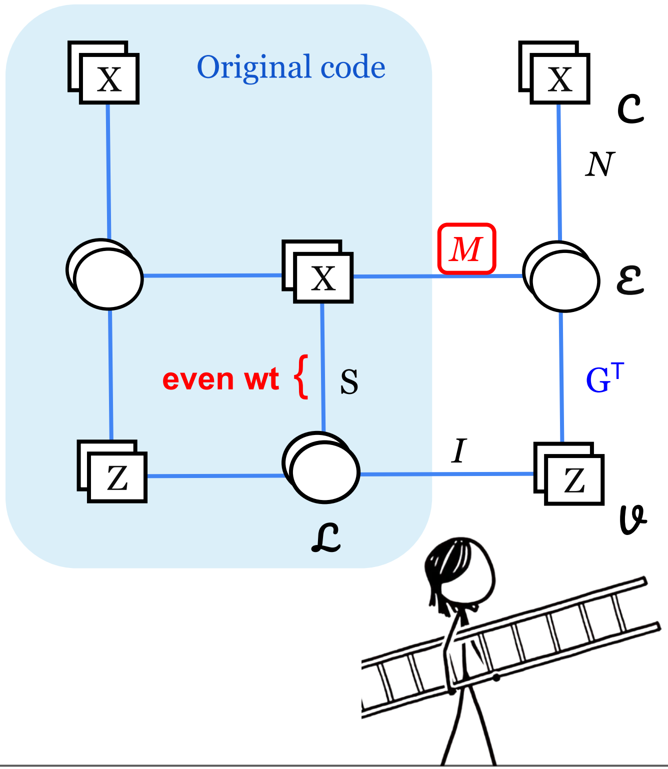
Desiderata 3: perfect matchings on graph $\mathcal{G}$

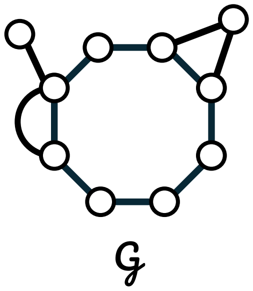




M deforms the original X stabilizers to edge qubits $\mathcal{E}$.

Desiderata 3: perfect matchings on graph $\mathcal{G}$

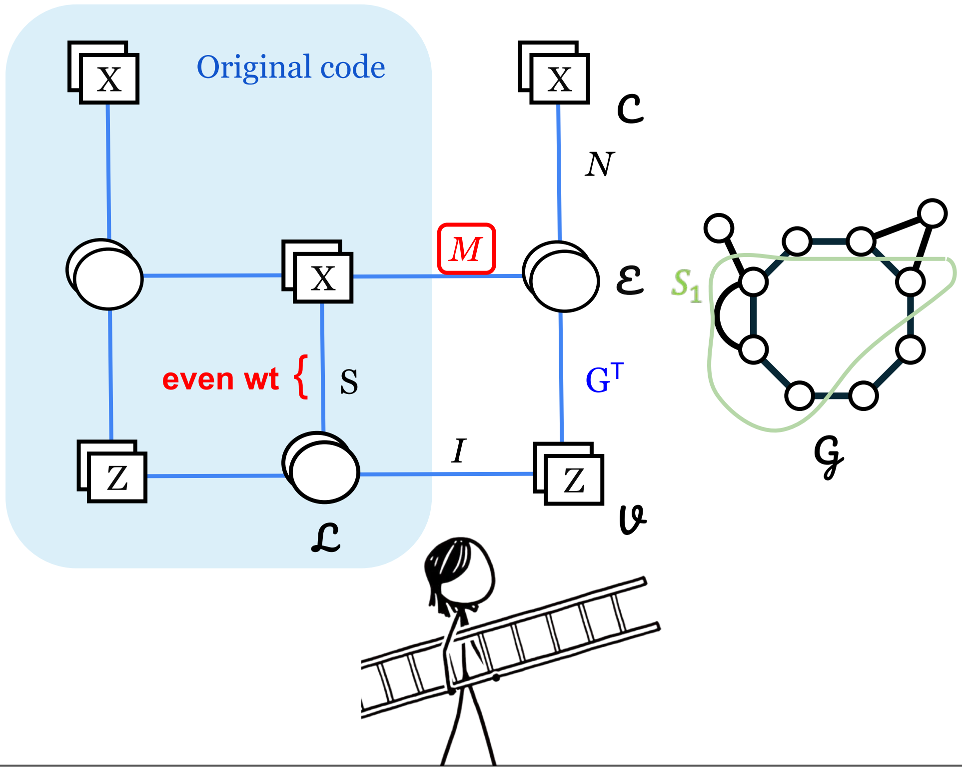




M deforms the original X stabilizers to edge qubits $\mathcal{E}$.

The original X stabilizers overlap $\mathcal{L}$ on an even number of qubits.

Desiderata 3: perfect matchings on graph $\mathcal{G}$

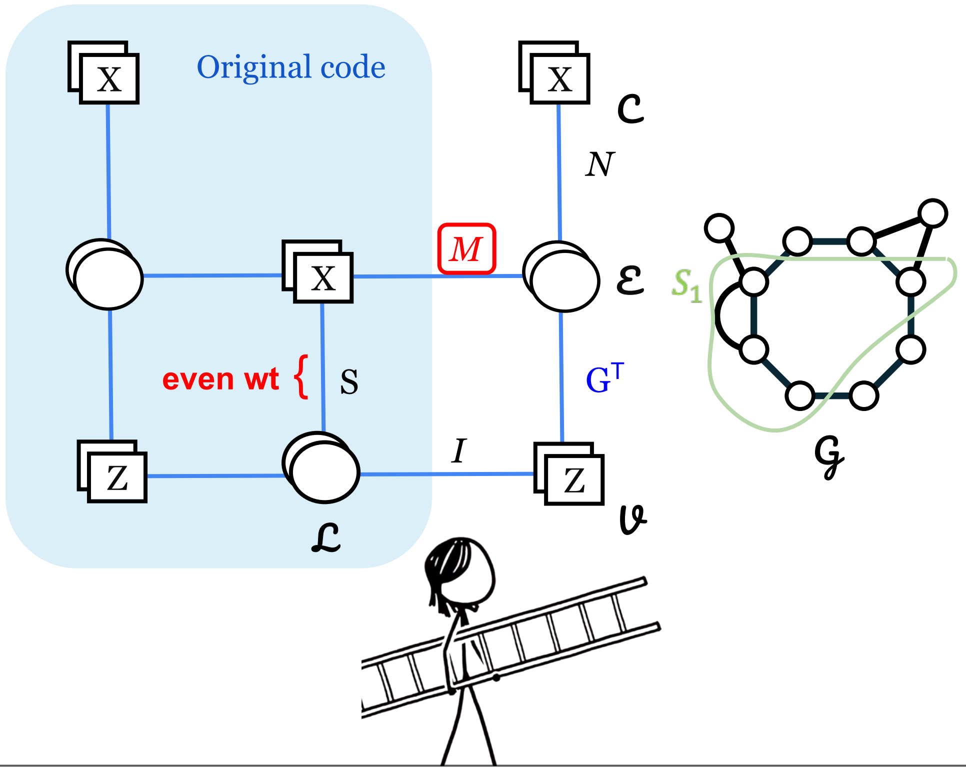


M deforms the original X stabilizers to edge qubits $\mathcal{E}$.

The original X stabilizers overlap $\mathcal{L}$ on an even number of qubits.

⟺ original X stabilizers anti-commute with an even number of new vertex Z stabilizers $\mathcal{V}$.

Desiderata 3: perfect matchings on graph $\mathcal{G}$

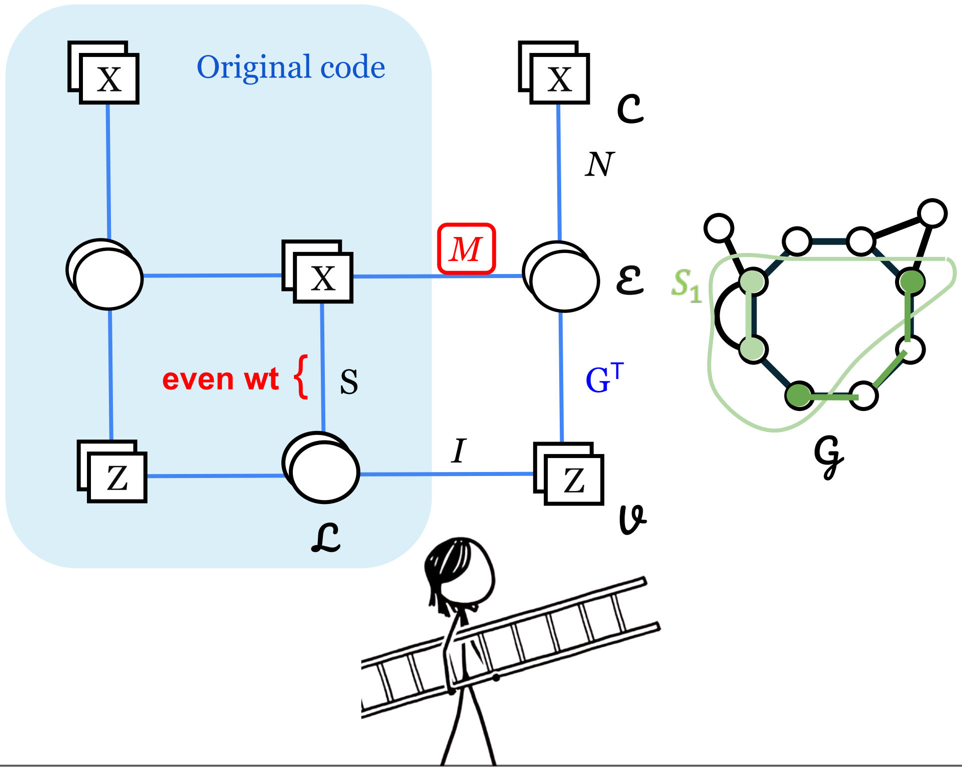


M deforms the original X stabilizers to edge qubits $\mathcal{E}$.

The original X stabilizers overlap $\mathcal{L}$ on an even number of qubits.

M is just a way to break up the even sized set of vertices → pairs of vertices i.e. a *matching* on this graph

Desiderata 3: perfect matchings on graph $\mathcal{G}$



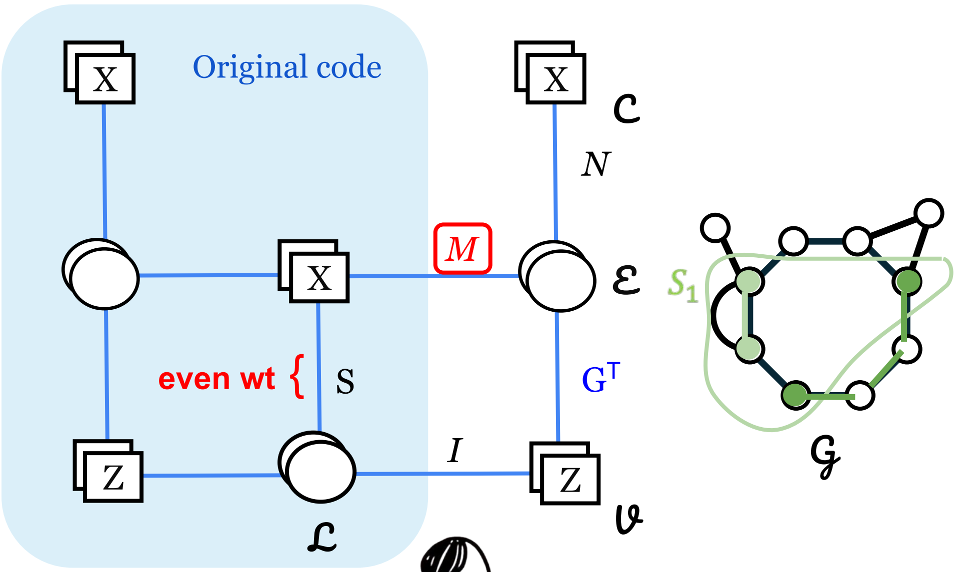
M deforms the original X stabilizers to edge qubits $\mathcal{E}$.

The original X stabilizers overlap $\mathcal{L}$ on an even number of qubits.

M is just a way to break up the
even sized set of vertices → pairs of vertices
i.e. a *matching* on this graph

length of matching
= # of edges in this path b/w paired vertices
= # of extra qubits supp. deformed X stabilizers

Desiderata 3: perfect matchings on graph $\mathcal{G}$

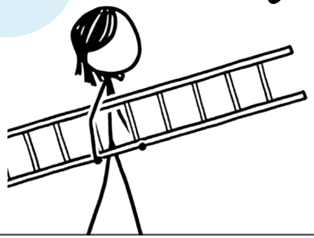


M deforms the original X stabilizers to edge qubits $\mathcal{E}$.

The original X stabilizers overlap $\mathcal{L}$ on an even number of qubits.

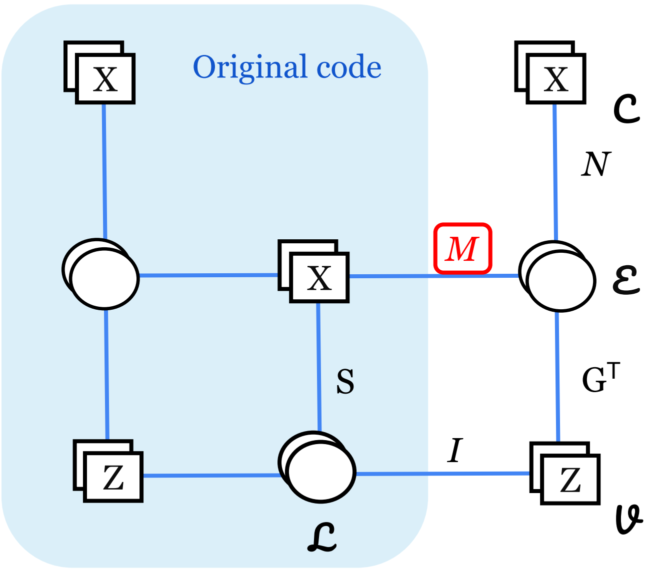
M is just a way to break up the
even sized set of vertices → pairs of vertices
i.e. a *matching* on this graph

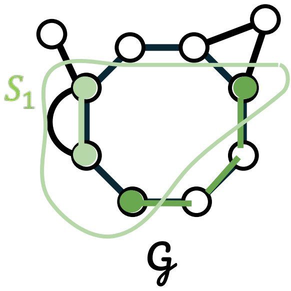
length of matching
= # of edges in this path b/w paired vertices
= # of extra qubits supp. deformed X stabilizers



Ingredient #2: come up with a (short) perfect matching on $\mathcal{G}$

Graph Desiderata





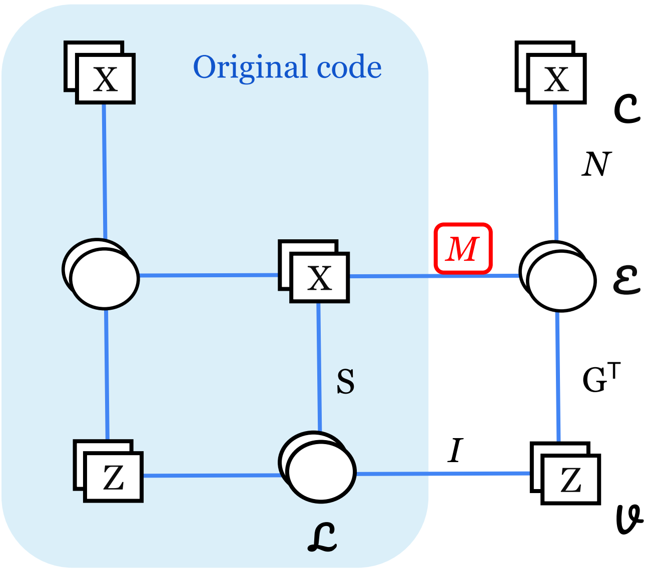
We want:

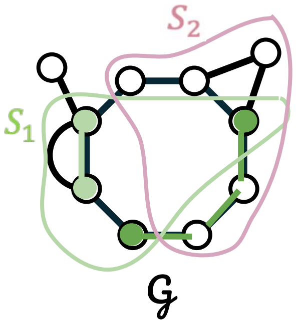
1. Code should preserve distance of original code.
2. Merged code should be LDPC at all stages

Graph $\mathcal{G}$ should have the following properties:

1. $\mathcal{G}$ is expanding;
2. All vertices have O(1) degree
3. Short perfect matchings on $\mathcal{G}$ (For original X checks)

Graph Desiderata





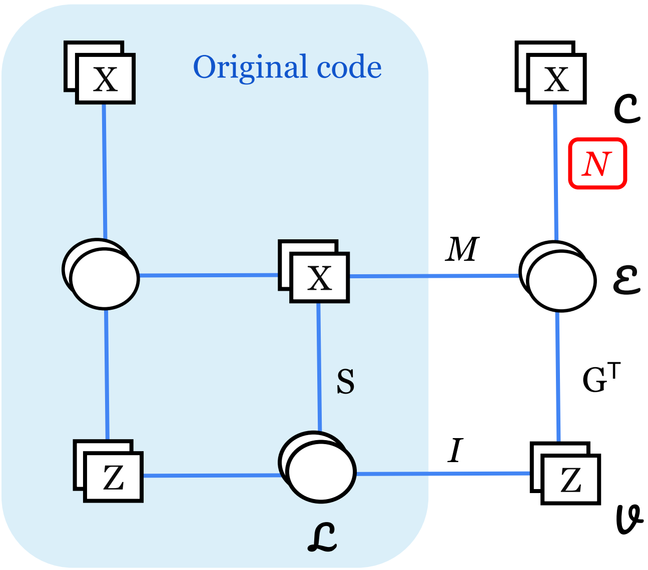
We want:

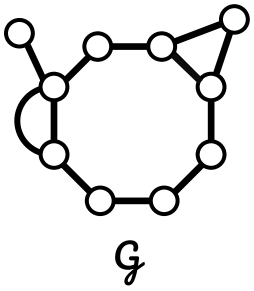
1. Code should preserve distance of original code.
2. Merged code should be LDPC at all stages

Graph $\mathcal{G}$ should have the following properties:

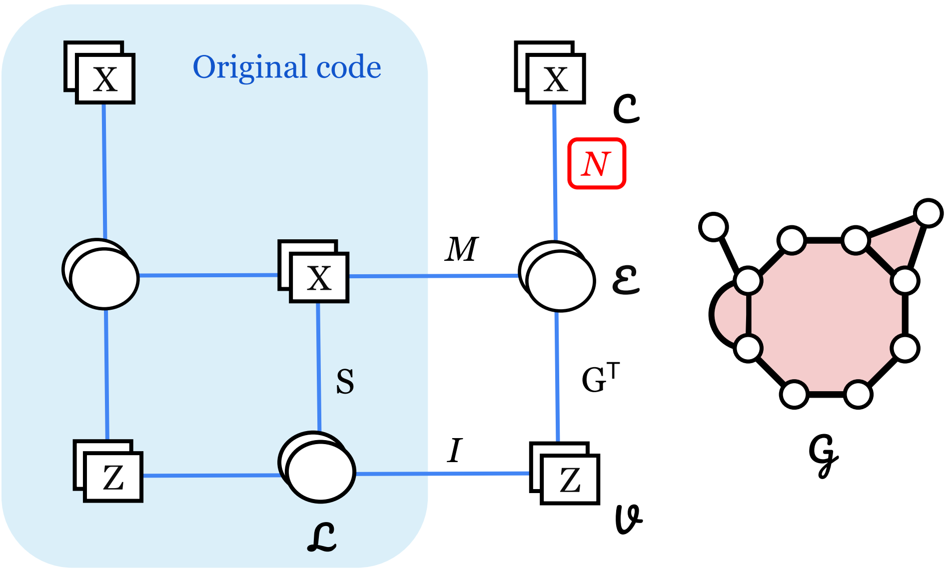
1. $\mathcal{G}$ is expanding;
2. All vertices have O(1) degree
3. Short perfect matchings on $\mathcal{G}$ (For original X checks)

Desiderata 4: Sparse cycle basis for graph $\mathcal{G}$



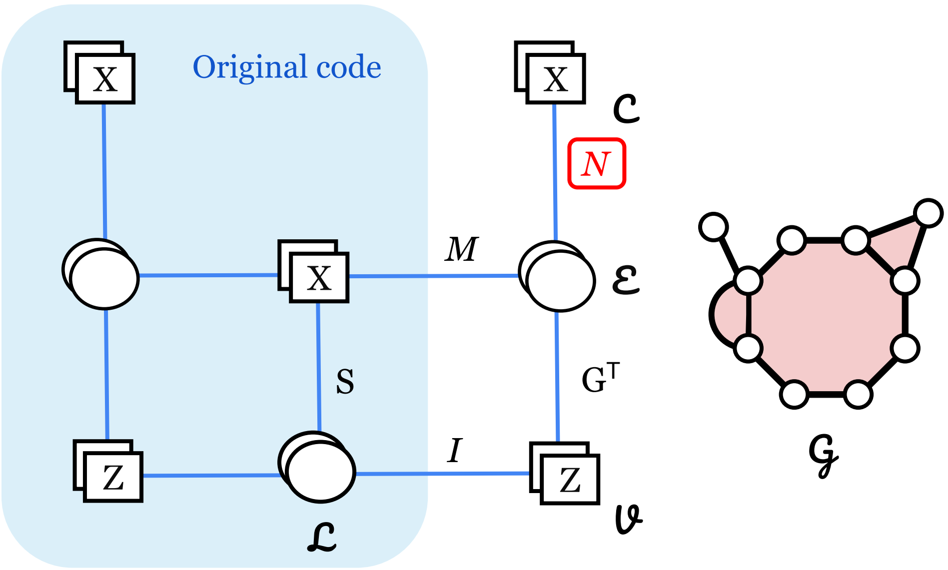


Desiderata 4: Sparse cycle basis for graph $\mathcal{G}$



N corresponds to cycles in the graph G.

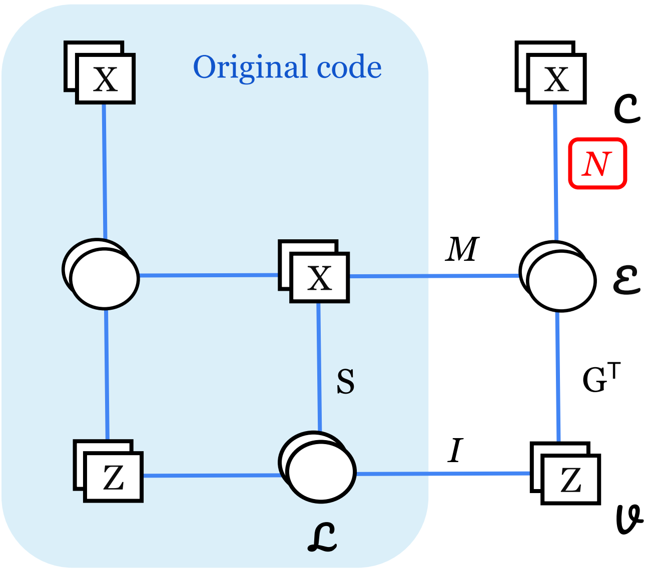
Desiderata 4: Sparse cycle basis for graph $\mathcal{G}$

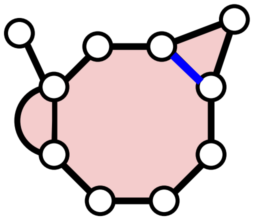


N corresponds to cycles in the graph G.

To ensure N is sparse,

Desiderata 4: Sparse cycle basis for graph $\mathcal{G}$

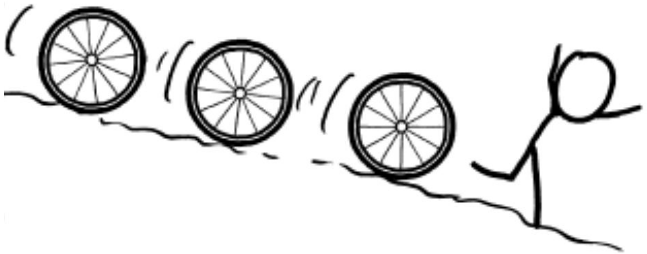




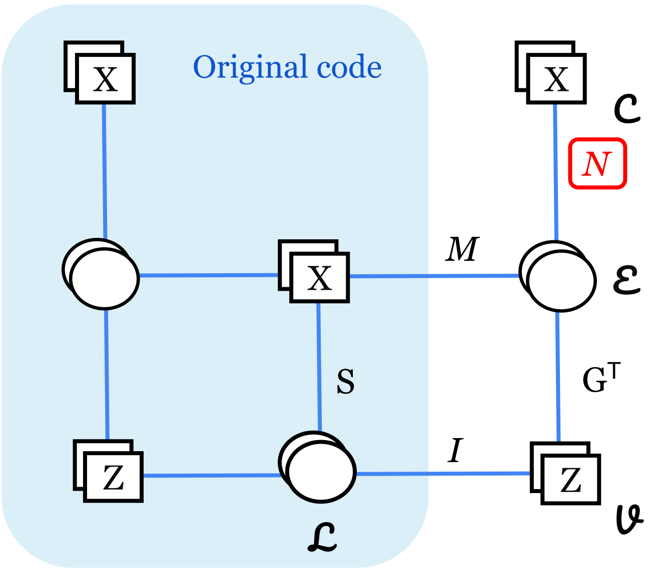
$\mathcal{G}$

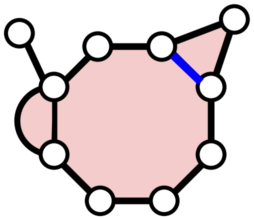
N corresponds to cycles in the graph G.

To ensure N is sparse,



Desiderata 4: Sparse cycle basis for graph $\mathcal{G}$

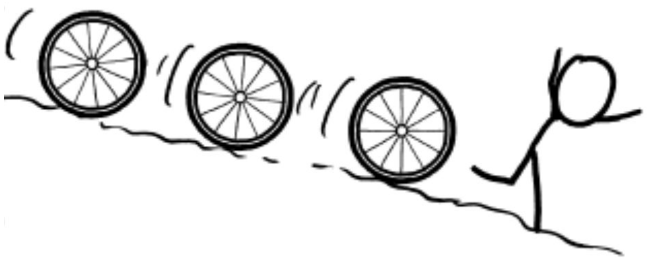




$\mathcal{G}$

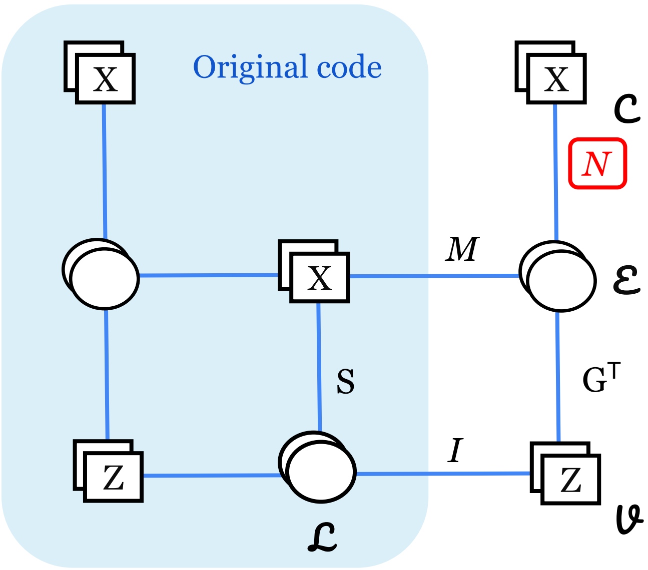
N corresponds to cycles in the graph G.

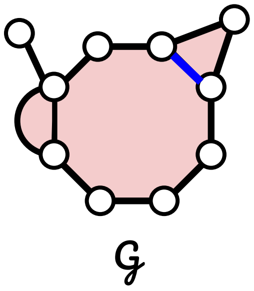
To ensure N is sparse,



- Each edge isn't in too many cycles

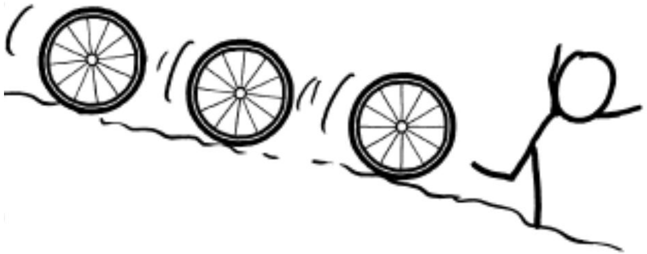
Desiderata 4: Sparse cycle basis for graph $\mathcal{G}$





N corresponds to cycles in the graph G.

To ensure N is sparse,



- Each edge isn't in too many cycles
- Each cycle* isn't too long.

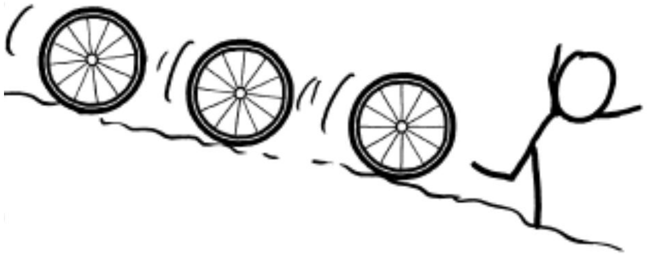
*element of the cycle basis

Desiderata 4: Sparse cycle basis for graph $\mathcal{G}$

Ensuring the cycle basis of graph is sparse :

- Each edge appears only in O(1) cycle basis elements

Ingredient #3: a notion of "decongesting" cycles in a graph



Desiderata 4: Sparse cycle basis for graph $\mathcal{G}$

Ensuring the cycle basis of graph is sparse :

- Each edge appears only in O(1) cycle basis elements

Ingredient #3: a notion of "decongesting" cycles in a graph

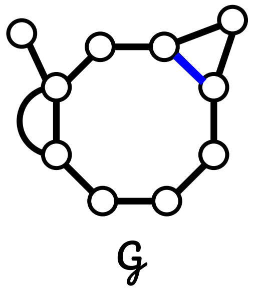
A cycle basis to begin with [Freedman Hastings 2020]

Desiderata 4: Sparse cycle basis for graph $\mathcal{G}$

Ensuring the cycle basis of graph is sparse :

- Each edge appears only in O(1) cycle basis elements

Ingredient #3: a notion of "decongesting" cycles in a graph



Cycles = X-checks

Edges = qubits

A cycle basis to begin with [Freedman Hastings 2020]

Input: graph $\mathcal{G}$ with O(1) vertex degree

Output: a cycle basis s.t.

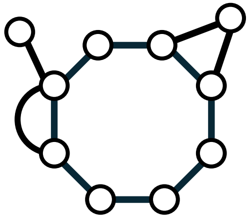
each cycle overlaps with at most **$O(\log^3(|V|))$** cycles.

Desiderata 4: Sparse cycle basis for graph $\mathcal{G}$

Ensuring the cycle basis of graph is sparse :

- Each edge appears only in O(1) cycle basis elements

Ingredient #3: a notion of "decongesting" cycles in a graph



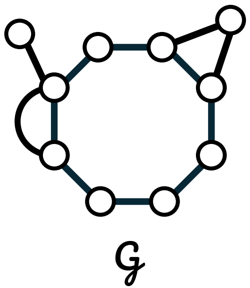
$\mathcal{G}$

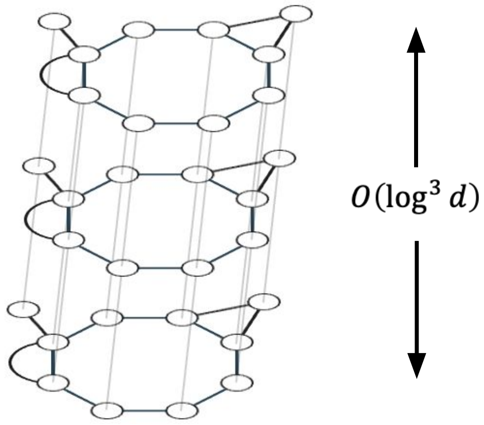
Desiderata 4: Sparse cycle basis for graph $\mathcal{G}$

Ensuring the cycle basis of graph is sparse :

- Each edge appears only in O(1) cycle basis elements

Ingredient #3: a notion of "decongesting" cycles in a graph



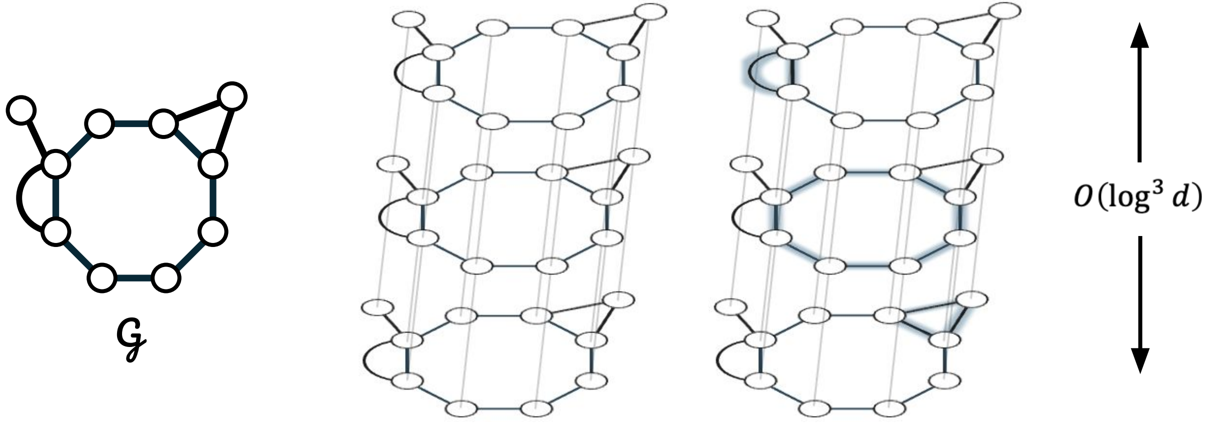


Desiderata 4: Sparse cycle basis for graph $\mathcal{G}$

Ensuring the cycle basis of graph is sparse :

- Each edge appears only in O(1) cycle basis elements

Ingredient #3: a notion of "decongesting" cycles in a graph

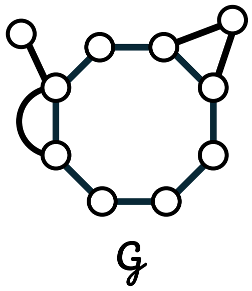


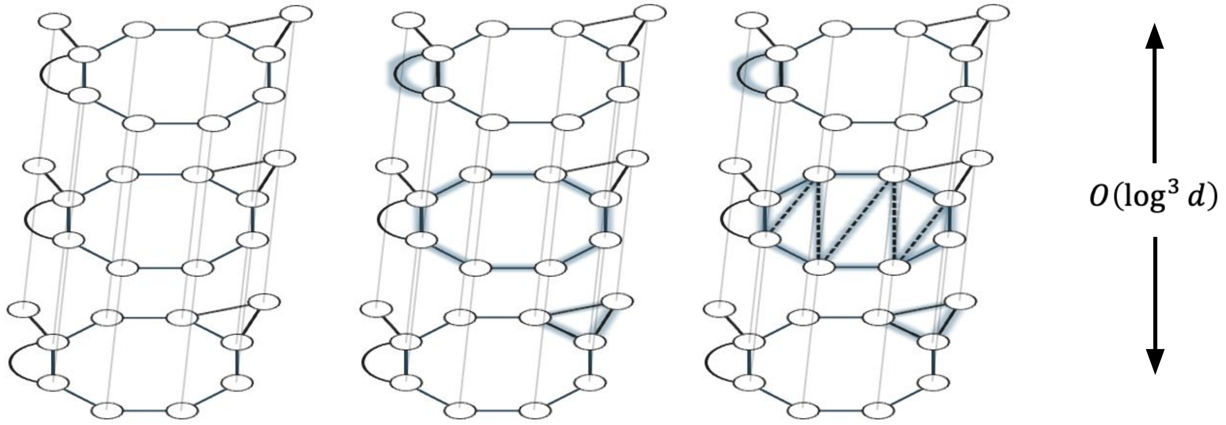
Desiderata 4: Sparse cycle basis for graph $\mathcal{G}$

Ensuring the cycle basis of graph is sparse :

- Each edge appears only in O(1) cycle basis elements

Ingredient #3: a notion of "decongesting" cycles in a graph

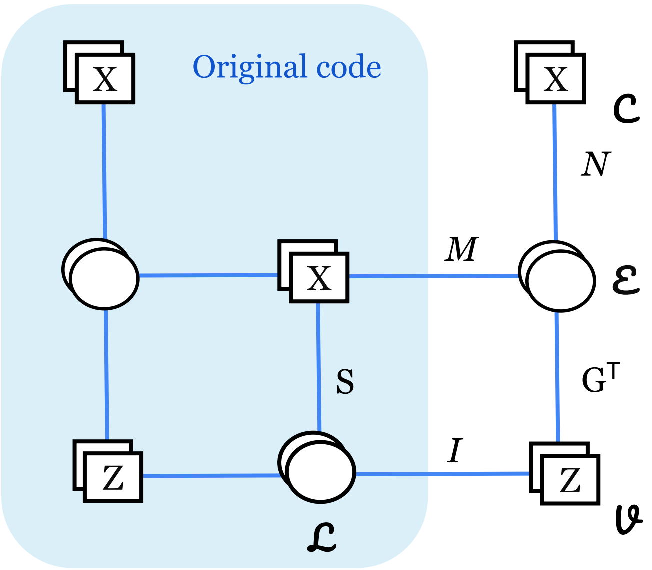


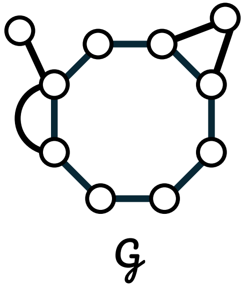


Also "cellulate" long cycles into smaller cycles

- Each cycle basis element has O(1) edges

Graph Desiderata





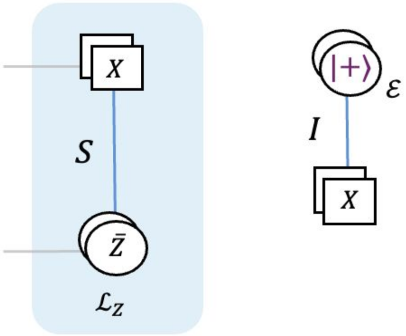
We want:

1. Code should preserve distance of original code.
2. Merged code should be LDPC at all stages

Graph $\mathcal{G}$ should have the following properties:

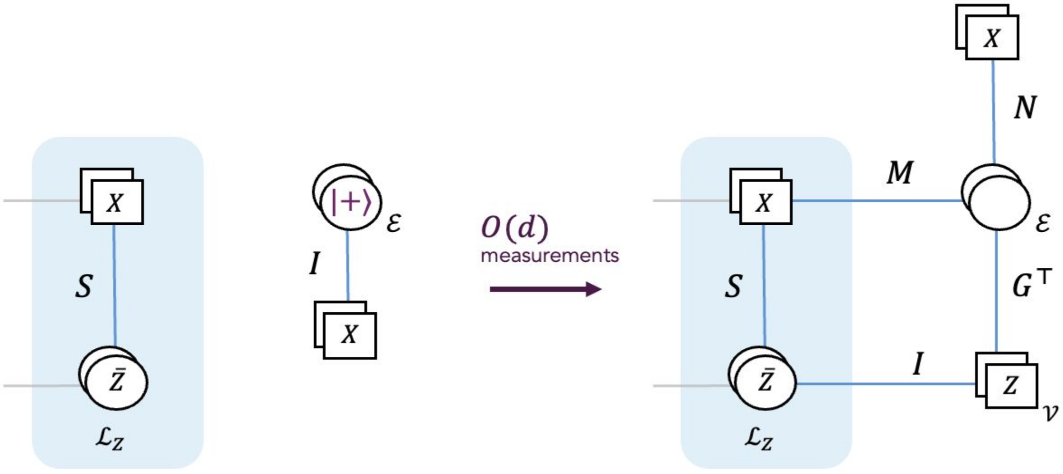
1. $\mathcal{G}$ is expanding; ✓
2. All vertices have O(1) degree ✓
3. Short perfect matchings on $\mathcal{G}$ (For original X checks) ✓ & each edge is in O(1) matchings. ✓
4. $\mathcal{G}$ has a sparse cycle basis. ✓

Overall protocol for auxiliary graph qLDPC surgery



(i) initialize

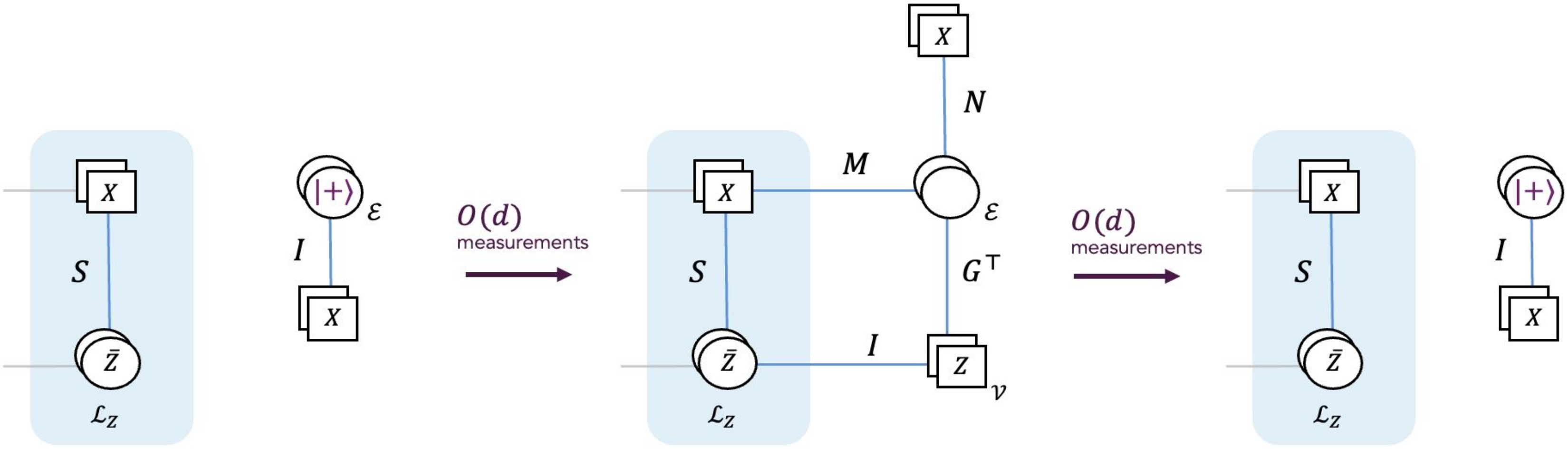
Overall protocol for auxiliary graph qLDPC surgery



(i) initialize

(ii) merge step

Overall protocol for auxiliary graph qLDPC surgery

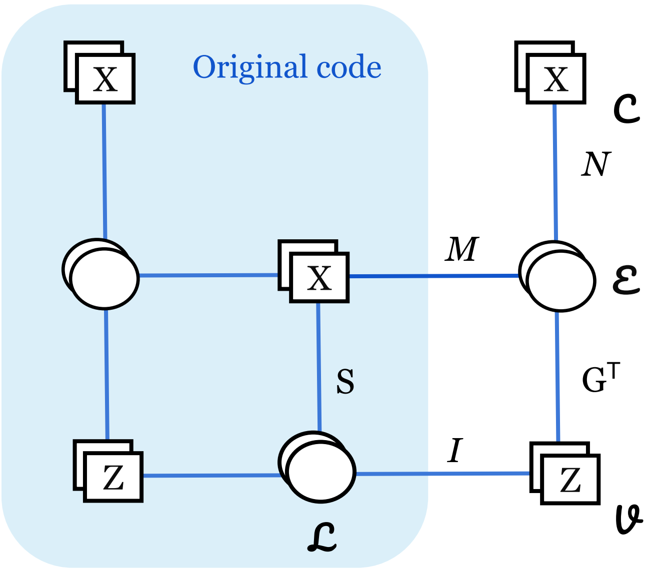


(i) initialize

(ii) merge step

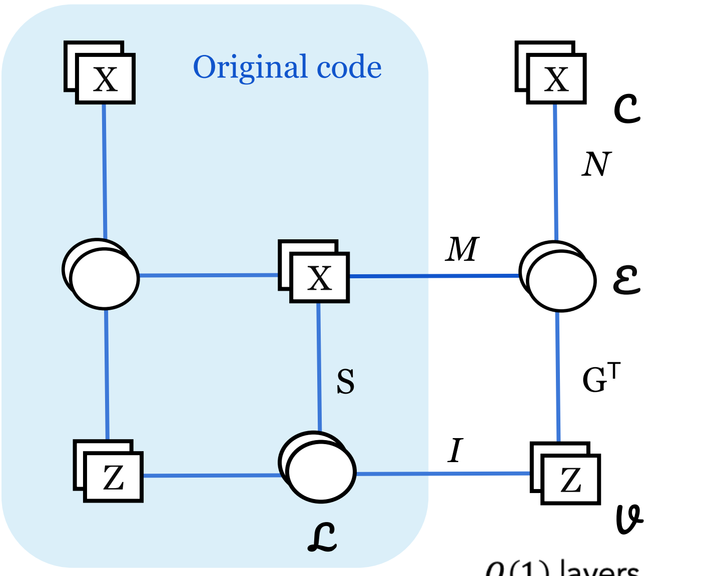
(iii) split step

In summary: auxiliary graph qLDPC surgery



✔ **Applicable to any quantum LDPC code!**

In summary: auxiliary graph qLDPC surgery



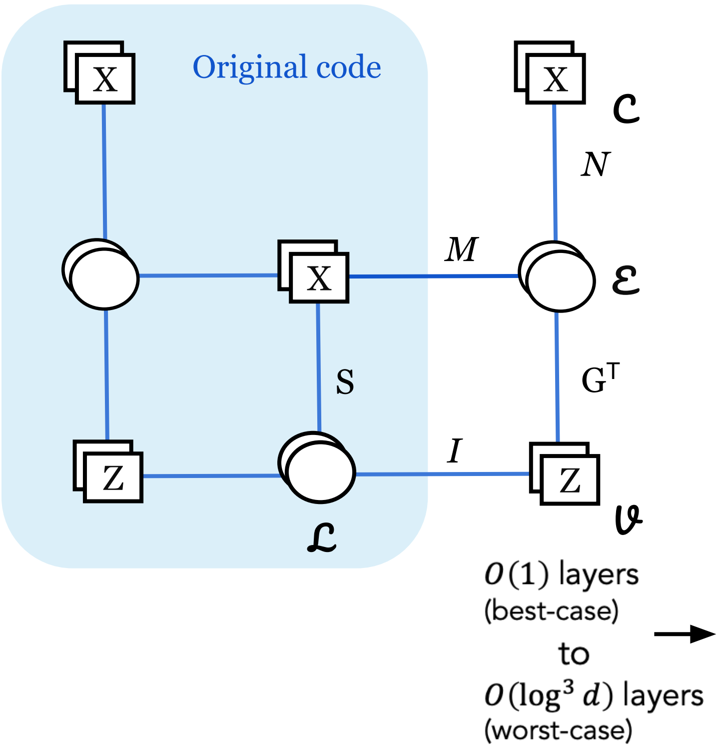
$O(1)$ layers
(best-case)
to
$O(\log^3 d)$ layers
(worst-case)

✔ **Applicable to any quantum LDPC code!**

Qubit overhead of scheme: **$O(d \log^3 (d))$**

~ **O(d) in distance d, upto polylog**

In summary: auxiliary graph qLDPC surgery



✔ **Applicable to any quantum LDPC code!**

Qubit overhead of scheme: **$O(d \log^3 (d))$**

~ **O(d) in distance d, upto polylog**

⇒ Significant improvement in overhead from previous scheme for arbitrary quantum LDPC codes, $O(d^2)$

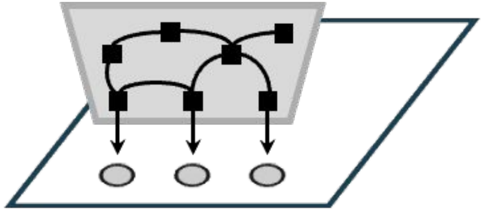
What about Pauli products?

Joint measurements of **products of logical Pauli operators, say $X_1Z_2Z_3$**, etc.. (without simultaneously measuring individual Paulis)

What about Pauli products?

Joint measurements of **products of logical Pauli operators, say $X_1Z_2Z_3$**, etc.. (without simultaneously measuring individual Paulis)

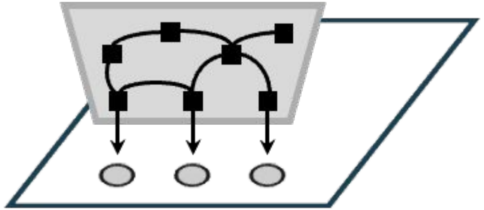
Construct a large **auxiliary graph** on the **entire logical**.



What about Pauli products?

Joint measurements of **products of logical Pauli operators, say $X_1Z_2Z_3$**, etc.. (without simultaneously measuring individual Paulis)

Construct a large **auxiliary graph** on the **entire logical**.

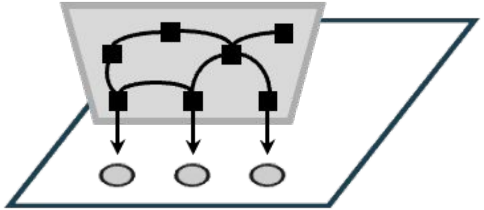


This would mean exponentially many auxiliary graphs for each of the ~ 4^k logical operators!

What about Pauli products?

Joint measurements of **products of logical Pauli operators, say $X_1Z_2Z_3$**, etc.. (without simultaneously measuring individual Paulis)

Construct a large **auxiliary graph** on the **entire logical**.



This would mean exponentially many auxiliary graphs for each of the ~ 4^k logical operators!

Can we break up the problem?

Outline

- Background and Motivation
  - Quantum LDPC Codes
  - Code Surgery Methods
- Auxiliary Graph Surgery on QLDPC Codes
  - Graph Desiderata
  - Universal Adapter for Joint-measurements
- Case Study: [[144,12,12]] Bivariate bicyclic code

Joint-measurements: a modular approach

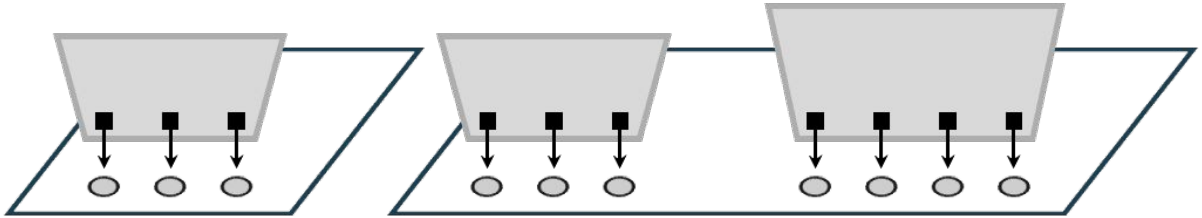
Is there a way to **connect** graphs constructed to measure **individual** Pauli logicals ?

Joint-measurements: a modular approach

Is there a way to **connect** graphs constructed to measure **individual** Pauli logicals (**efficiently**)?

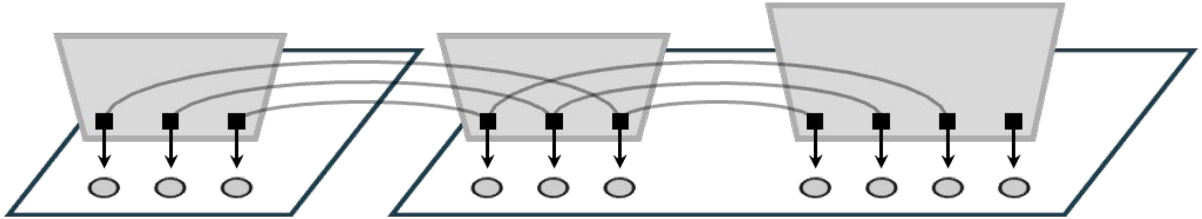
Joint-measurements: a modular approach

Is there a way to **connect** graphs constructed to measure **individual** Pauli logicals (**efficiently**)?



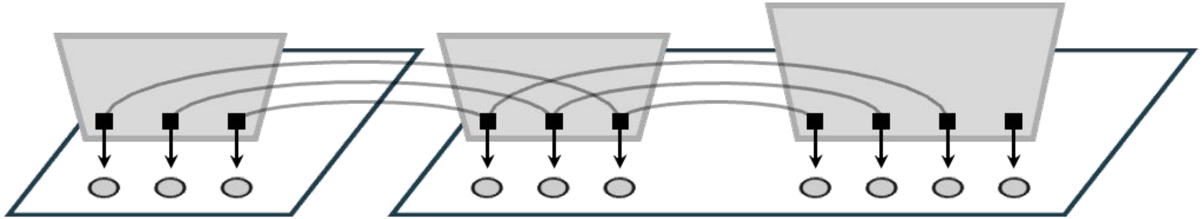
Joint-measurements: a modular approach

Is there a way to **connect** graphs constructed to measure **individual** Pauli logicals (**efficiently**)?



Joint-measurements: a modular approach

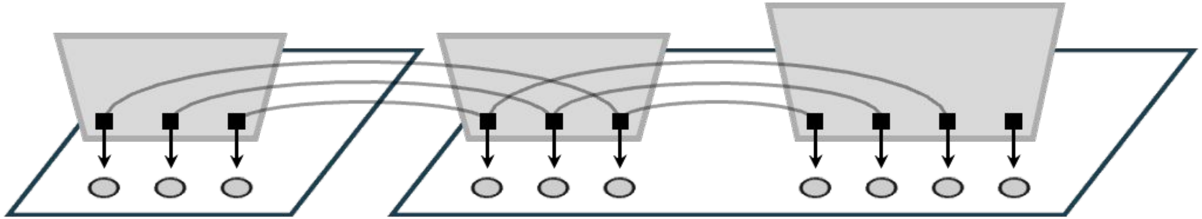
Is there a way to **connect** graphs constructed to measure **individual** Pauli logicals (**efficiently**)?



~ needs only d extra edges

Joint-measurements: a modular approach

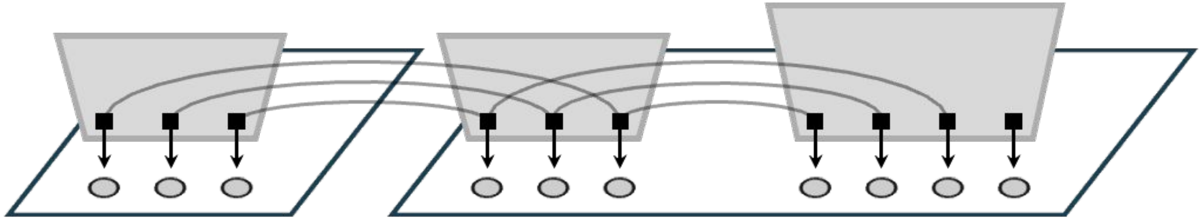
Is there a way to **connect** graphs constructed to measure **individual** Pauli logicals (**efficiently**)?



~ needs only d extra edges ≡ d extra qubits

Joint-measurements: a modular approach

Is there a way to **connect** graphs constructed to measure **individual** Pauli logicals (**efficiently**)?

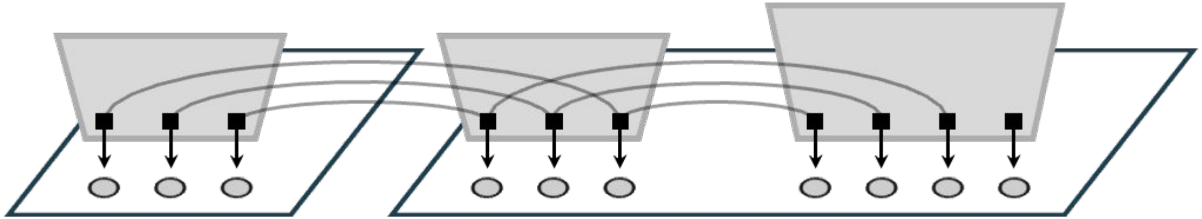


~ needs only d extra edges ≡ d extra qubits

New cycles are at most length 8

Joint-measurements: a modular approach

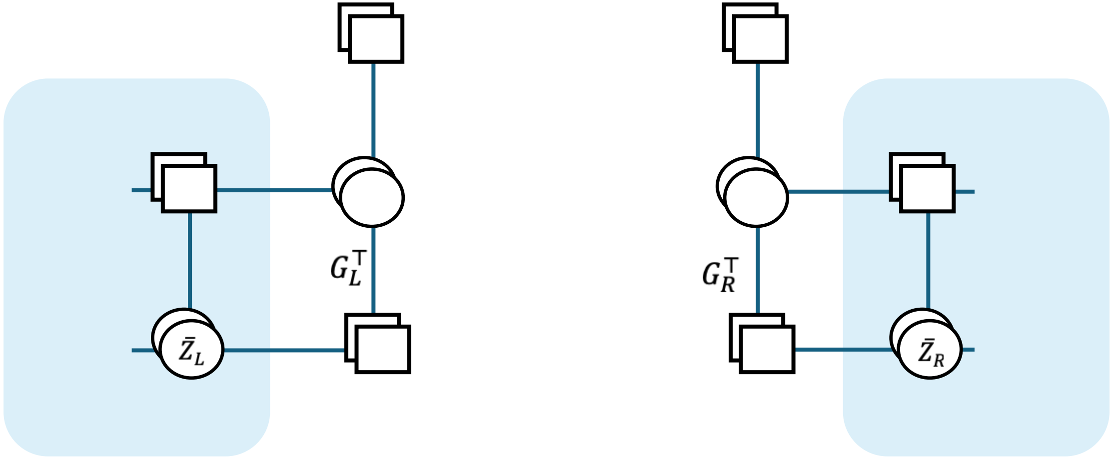
Is there a way to **connect** graphs constructed to measure **individual** Pauli logicals (**efficiently**)?



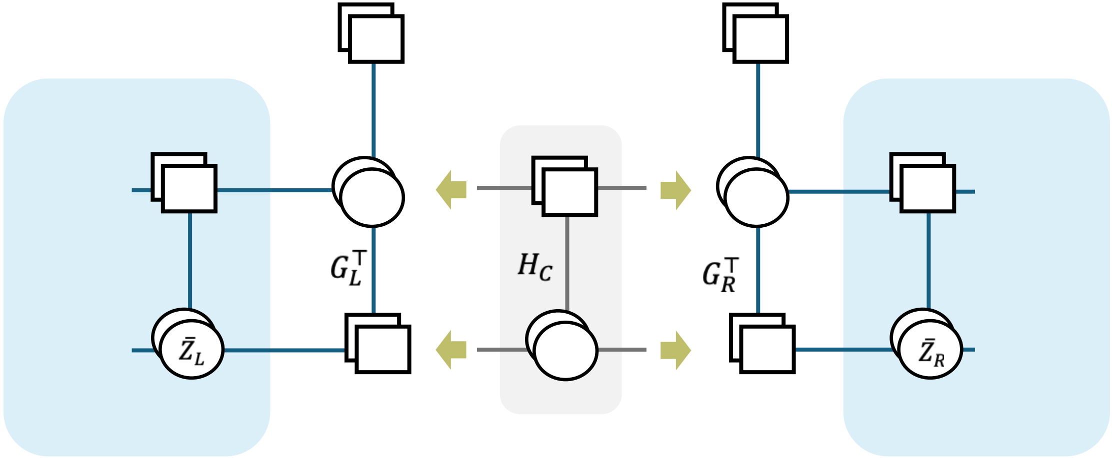
~ needs only d extra edges ≡ d extra qubits

New cycles are at most length 8 ≡ new X stabilizers are at most weight 8

Universal adapter: a new primitive

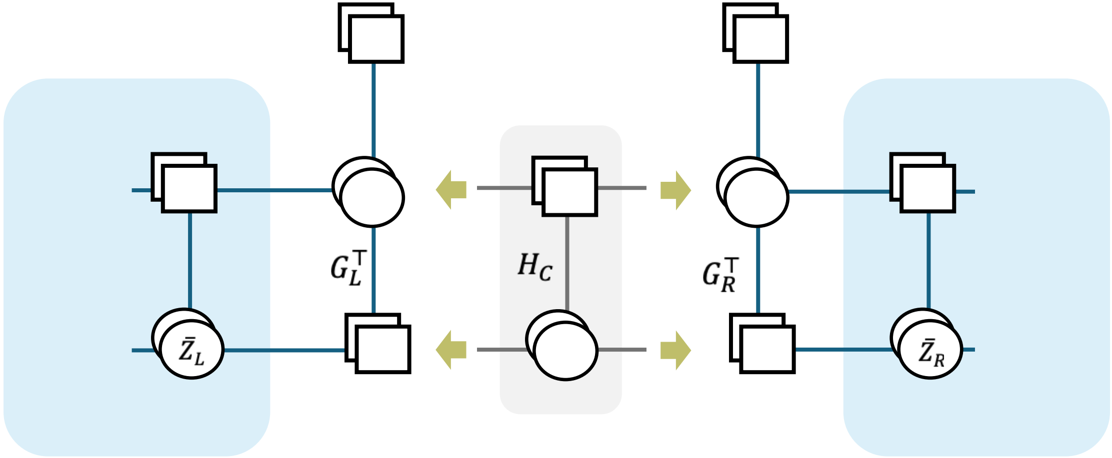


Universal adapter: a new primitive



Key insight: any tree-like graph is equivalent to a repetition code up to a transformation

Universal adapter: a new primitive

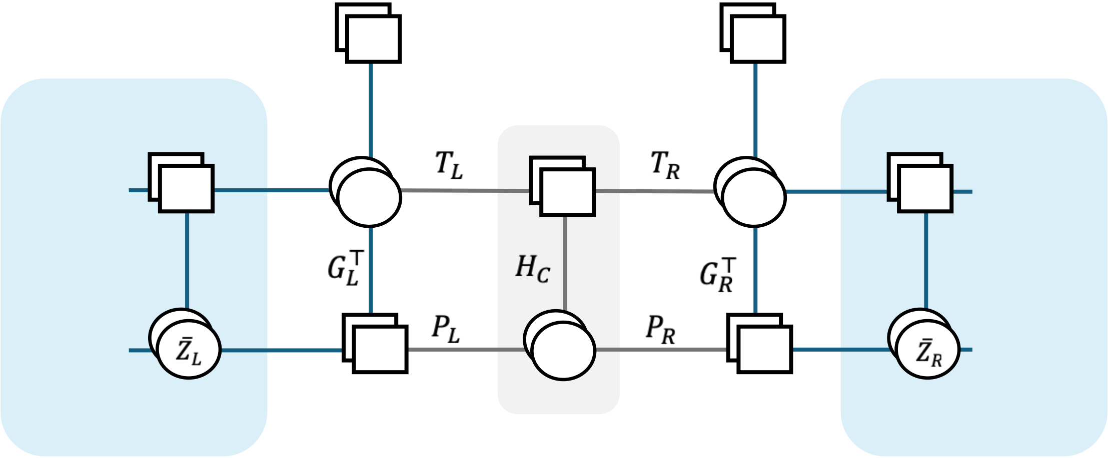


SkipTree(G) = T, P

Key insight: any tree-like graph is equivalent to a repetition code up to a transformation

[Universal Adapters between quantum LDPC codes], 2410.03628

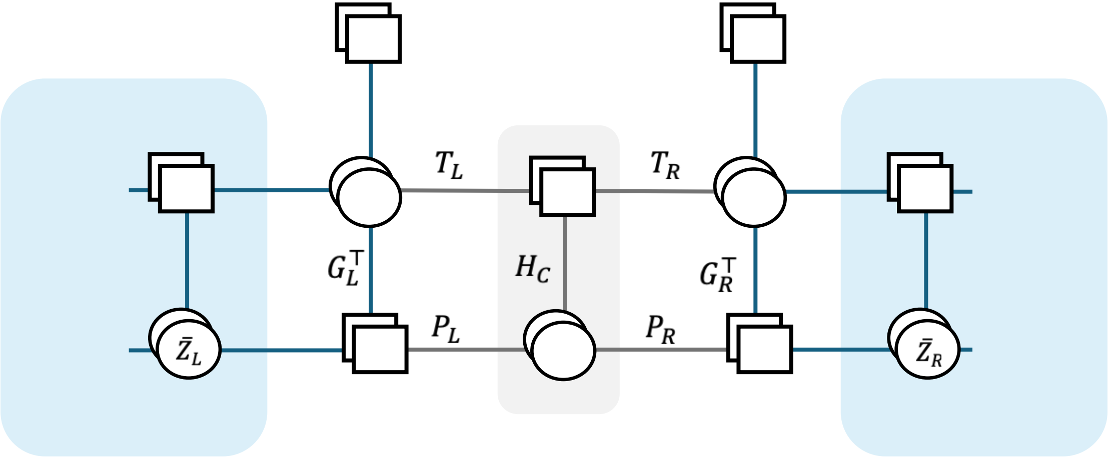
Universal adapter: a new primitive



SkipTree(G) = T, P

Key insight: any tree-like graph is equivalent to a repetition code up to a transformation

Universal adapter: a new primitive



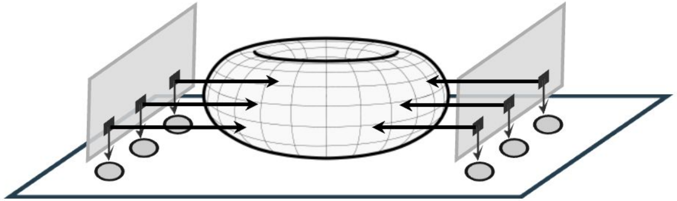
Significance : These can be arbitrary* logical operators in the LDPC code, they could belong to the same or different codeblocks, or even different quantum codes

A "Universal" way to connect between any two logical operators in quantum LDPC codes

Universal adapter: Connecting different qLDPC codes

- Arbitrary joint-measurements in the same or different codeblocks
- **multi-code architectures**…

Can be useful for teleportation, transversal gates, magic state factory, code-switching…



Can leverage known symmetries in codes, and implement these gates on logical qubits of other codes.

In Summary

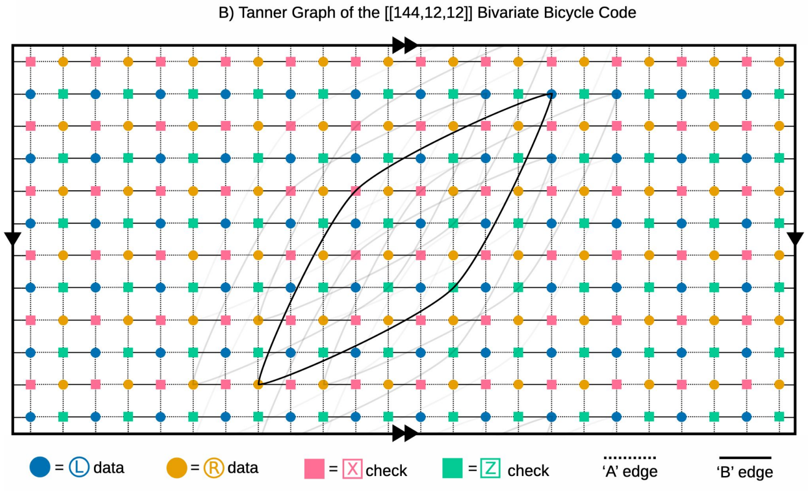
- **Addressable** gates for multi-qubit code blocks
- A "**universal**" scheme applicable to **arbitrary** quantum LDPC codes
- Space overhead is only ~**linear** - which is a quadratic improvement over previous schemes; the LDPC overhead advantage is now beginning to show even for logical gate schemes.

- Can connect arbitrary codes for **multi-code architectures** with just ~**d extra qubits** allowing one to leverage unique advantages of different codes, while keeping architecture LDPC.

Outline

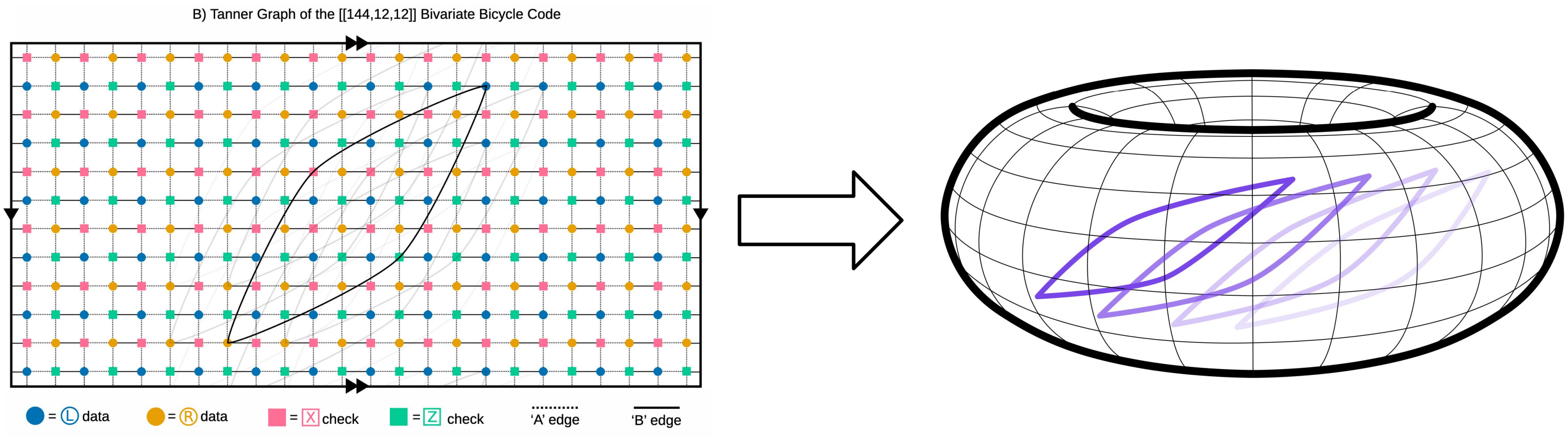
- Background and Motivation
  - Quantum LDPC Codes
  - Code Surgery Methods
- Auxiliary Graph Surgery on QLDPC Codes
  - Graph Desiderata
  - Universal Adapter for Joint-measurements
- Case Study: [[144,12,12]] Bivariate bicyclic code

Case Study: [[144, 12, 12]] Bivariate Bicycle Code



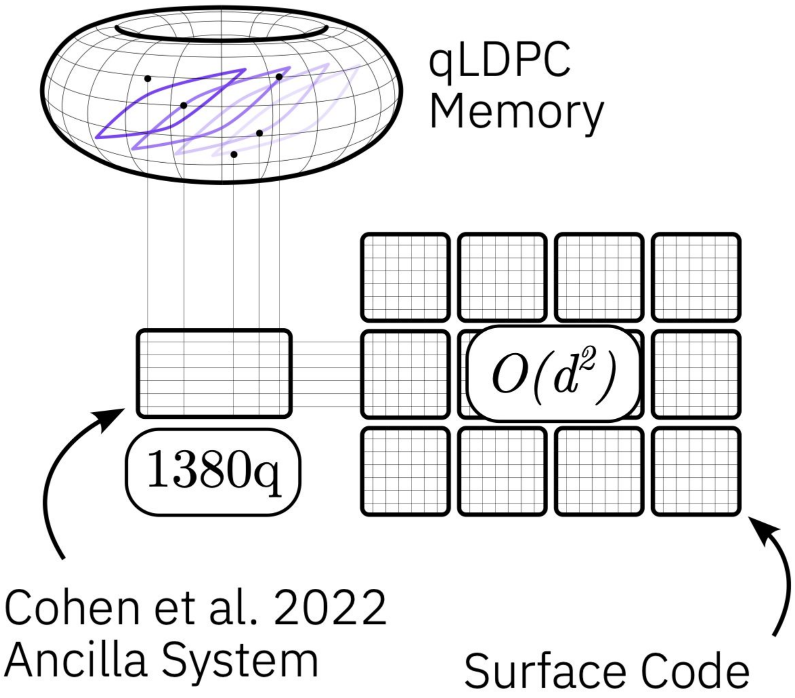
* High-threshold and low-overhead fault-tolerant quantum memory [Bravyi et al 2023]

Case Study: [[144, 12, 12]] Bivariate Bicycle Code



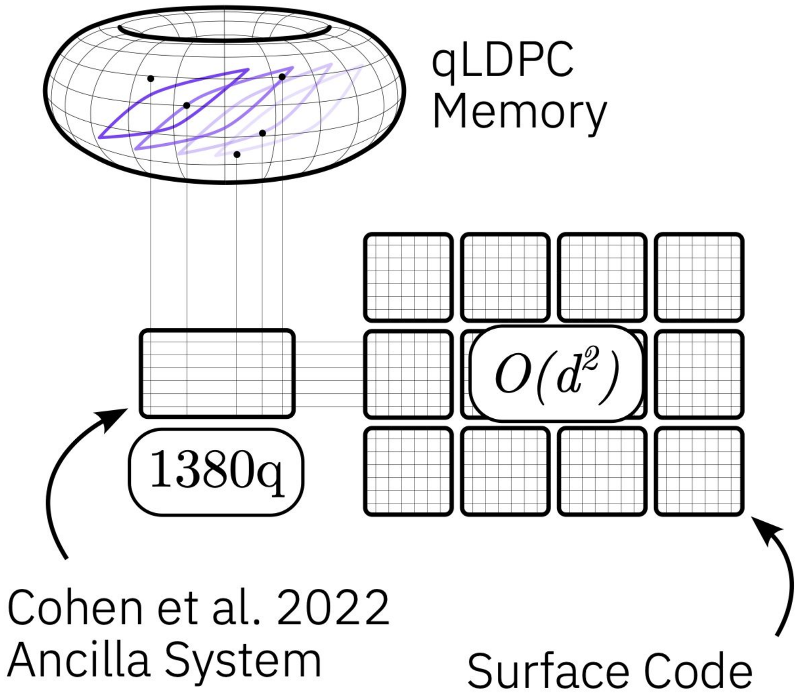
* High-threshold and low-overhead fault-tolerant quantum memory [Bravyi et al 2023]

Previous Proposal for Logical Computation



* High-threshold and low-overhead fault-tolerant quantum memory [Bravyi et al 2023]

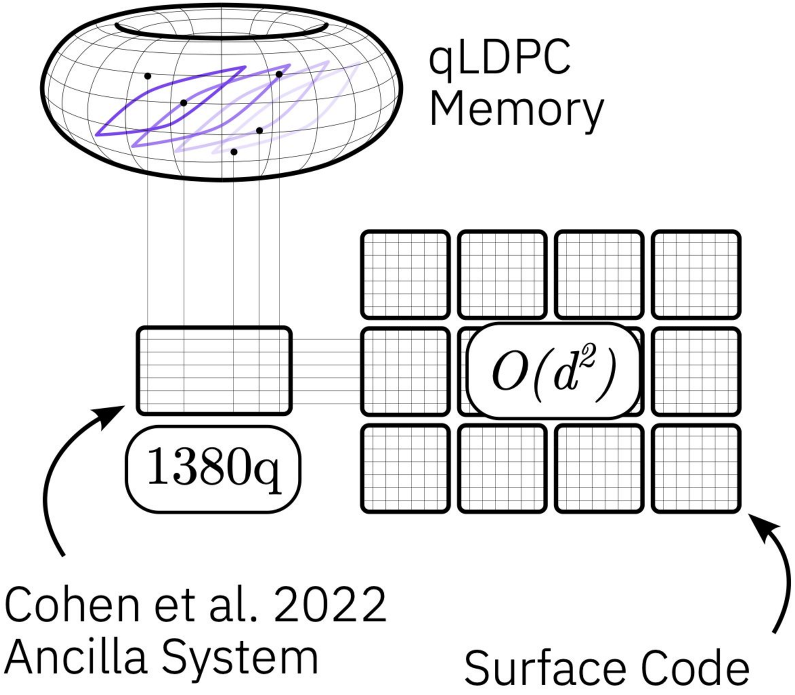
Previous Proposal for Logical Computation



Uses 1380 ancilla qubits.

* High-threshold and low-overhead fault-tolerant quantum memory [Bravyi et al 2023]

Previous Proposal for Logical Computation

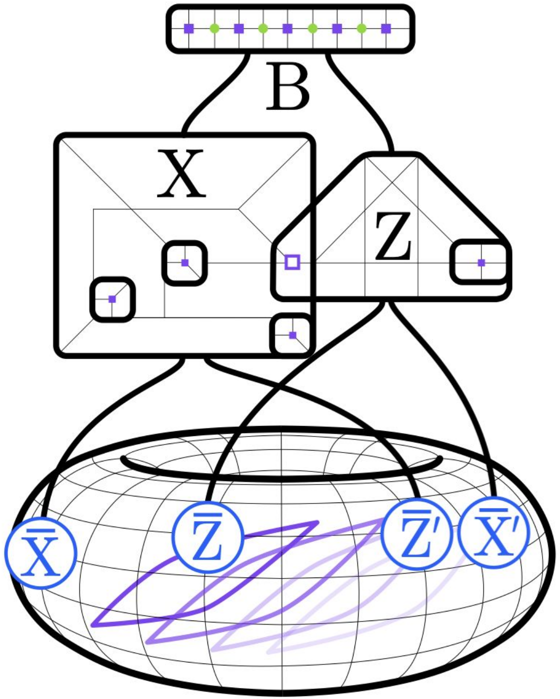


Uses 1380 ancilla qubits.

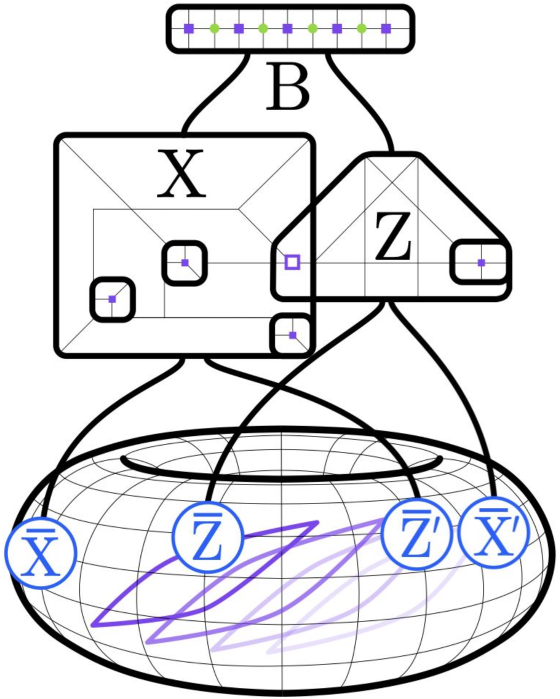
Teleport logical information into a surface code patch to do computation.

* High-threshold and low-overhead fault-tolerant quantum memory [Bravyi et al 2023]

New Proposal for Logical Computation



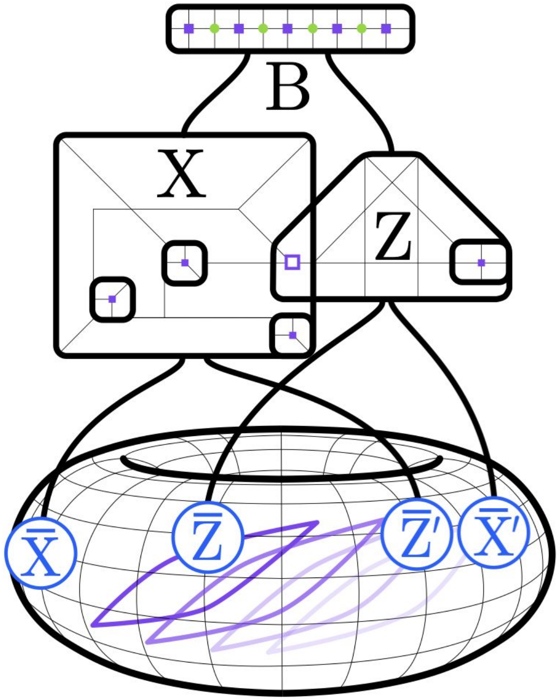
New Proposal for Logical Computation



Uses 103 ancilla qubits, 13x savings.

Use a bridge to perform joint measurement.

New Proposal for Logical Computation



Uses 103 ancilla qubits, 13x savings.

Use a bridge to perform joint measurement.

Automorphism gates + logical measurement =

Full logical Clifford group!

Promising numerical benchmarking.

Logical Computation on QLDPC Codes through Surgery

With Case Study on Bivariate Bicycle codes

Andrew Cross, **Zhiyang He (Sunny)**, Tomas Jochym-O'Connor, Patrick Rall, **Esha Swaroop**, Dominic Williamson, Theodore Yoder

The End