Parallelizing halo2: Resolving data flow dependencies with graphs

Application of Graph Methods for Efficient Quotient Polynomial Evaluation in Halo2

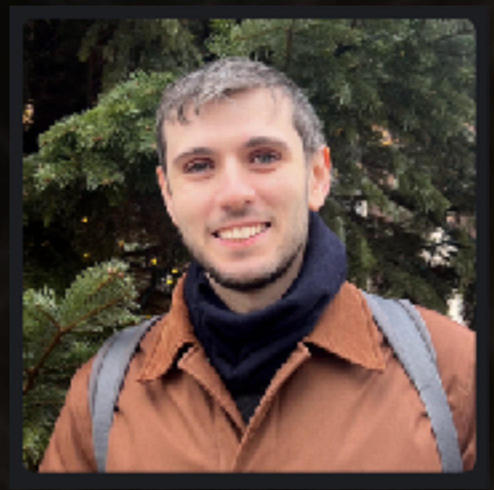
Karthik Inbasekar
Ingonyama
karthik@ingonyama.com

Roman Palkin
Ingonyama
roman@ingonyama.com

Guy Weissenberg
EPFL
guy.weissenberg@epfl.ch

INGONYAMA

ZKSummit 11



https://github.com/ingonyama-zk/papers/blob/main/halo2_community_detection_research.pdf

Outline

1. Introduction
2. Connected Components (CC) in a graph
3. Community detection
4. Applications
5. Summary

1. Introduction

Introduction

- Halo2 - ZKP system that uses Plonkish arithmetization
- Circuit data: (Public, Private) - encoded in a trace table
- Circuit constraints → polynomial identities and prove using quotient argument

Commit to the trace

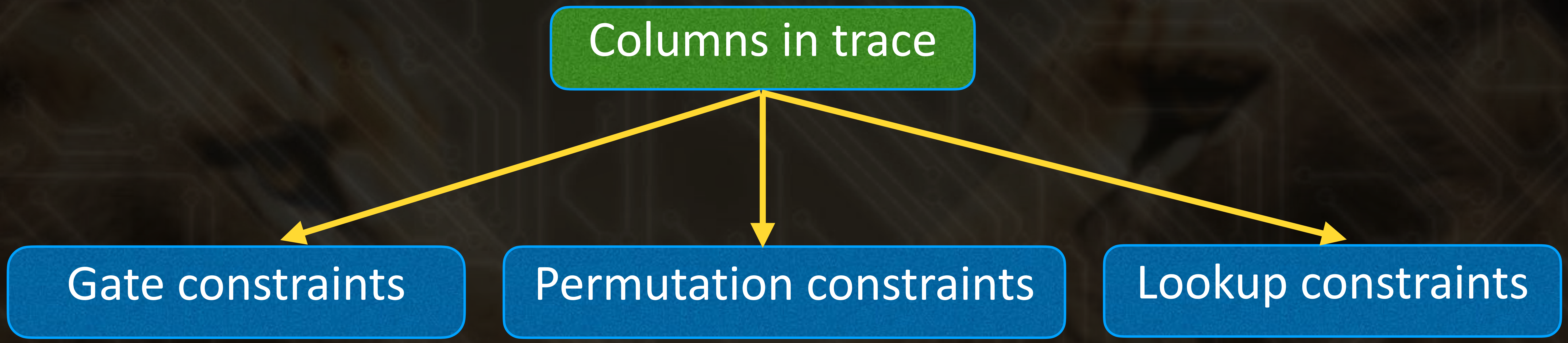
MSM, Elliptic curve, base field arithmetic.

H poly

NTT, scalar field arithmetic

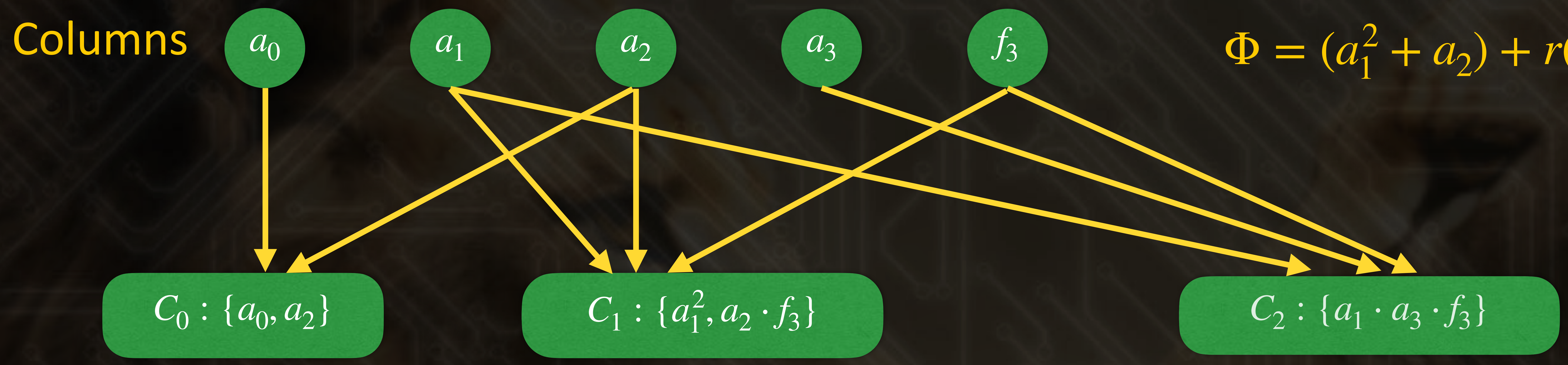
- MSM based compute primitives are highly parallelizable - universal
- Quotient (h poly): compute or memory bottlenecked - not universal
- Reason: Data flow dependency

Data flow dependency



- constraints - mathematically independent, but data flow dependent (shared data)
- Columns extended several times to different sizes for different constraints (many NTTs)
- hard to make circuit agnostic optimizations for constraint evaluation
- Quotient poly: takes 30-40% of proof time in several zkEVM/zkML circuits
- When, where and how to extend - key to minimize number/size of NTTs & memory

Evaluating h poly - eg: degree Clusters



$$\Phi = (a_1^2 + a_2) + r(a_1 . f_3 . a_3) + r^2(a_0 + a_2 . f_3)$$

Copy or recompute?

$Coset_0NTT(a_0)$

$Coset_0NTT(a_2)$ → $Coset_0NTT(a_2)$ $Coset_1NTT(a_2)$

$Coset_0NTT(a_1)$ $Coset_1NTT(a_1)$ → $Coset_0NTT(a_1)$ $Coset_1NTT(a_1)$ $Coset_2NTT(a_1)$ $Coset_3NTT(a_1)$

$Coset_0NTT(a_3)$ $Coset_1NTT(a_3)$ $Coset_2NTT(a_3)$ $Coset_3NTT(a_3)$

$Coset_0NTT(f_3)$ $Coset_1NTT(f_3)$ → $Coset_0NTT(f_3)$ $Coset_1NTT(f_3)$ $Coset_2NTT(f_3)$ $Coset_3NTT(f_3)$

$P_o^n = r^2 \cdot a_0 + a_2$ → Extend to 4n using CosetINTT and CosetNTT

$\oplus$

$P_1^{2n} = a_1^2 + r^2 \cdot a_2 \cdot f_3$ → Extend to 4n using CosetINTT and CosetNTT

$\oplus$

$P_2^{4n} = r \cdot a_1 \cdot a_3 \cdot f_3$

Cluster interdependency

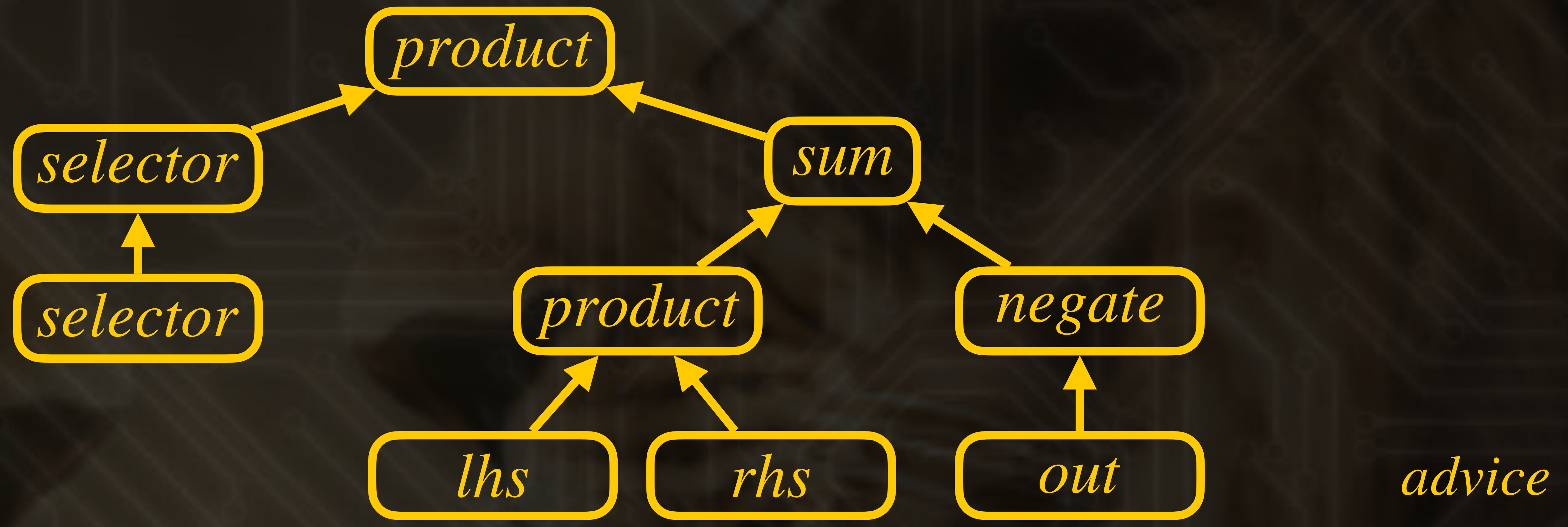
Difficult to parallelize

How to parallelize?

- Relationship between columns & constraints → graph relation of nodes & edges
- Halo2: already uses symbolic representation of constraints in AST

$$q\_M * (X_L * X_R - X_O)$$

Expression



Abstract Syntax Tree

- Extend idea to graph partition methods to analyze parallelizability of circuits!
- <u>Parallelizabilty criteria:</u>
    - independent *connected components* in the graph ?
    - communities - *weakly connected groups* of connected components ?

2. Connected Components (CC) in a graph

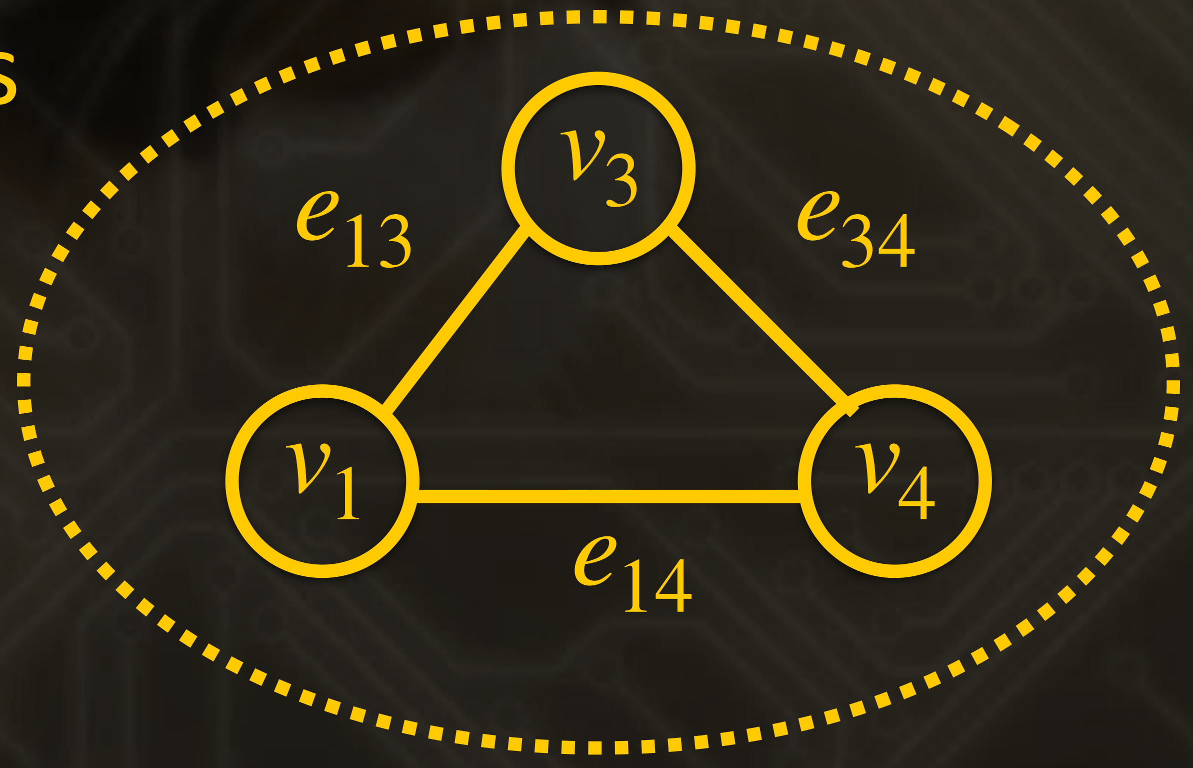
Connected components in a graph

- $G : (V, E)$ for $v_i \in V$ and $e_i \in E$

Simple graph ← Multigraph

- CC in a graph

Connected subgraphs



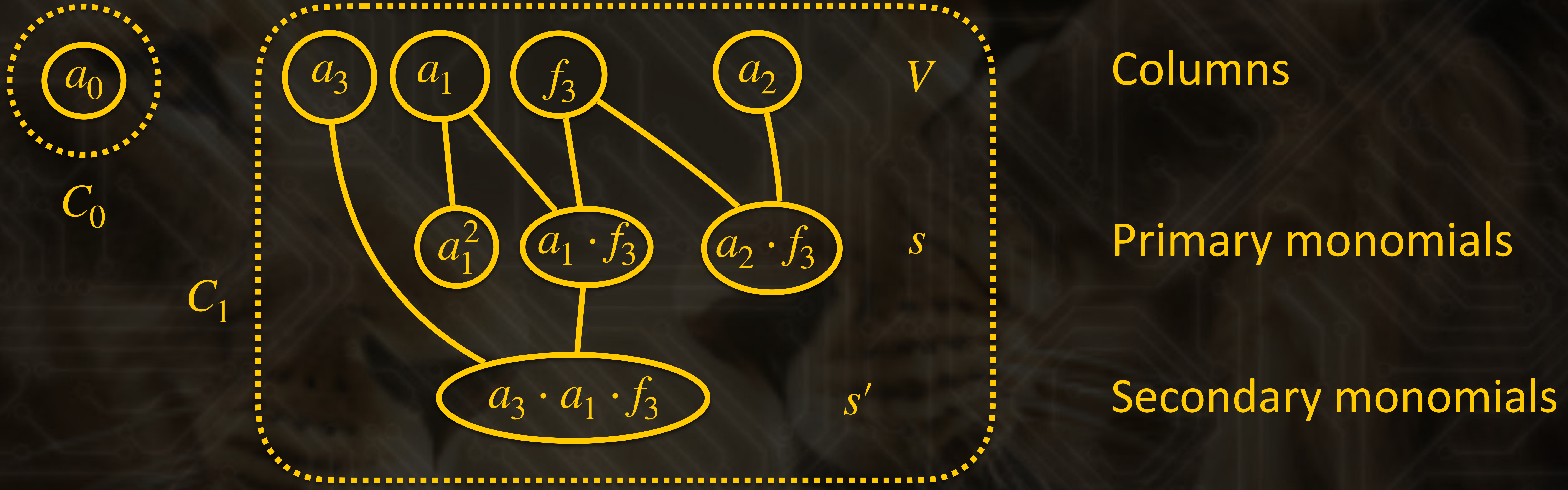
Disjoint pieces in a large graph

Building the graph - need a heuristic

- $G = (V, E)$
- $V = \{v_1, v_2, \ldots v_n, s\}$, v_i are columns in the trace.
- $s = \{s_1, s_2, \ldots, s'\}$ - primary monomial set; $s_i = F_i(v_1, v_2, \ldots)$
- $s' = \{s'_1, s'_2, \ldots, \}$ - secondary monomial set; $s'_i = \tilde{F}_i(s_1, s_2, \ldots, v_1, v_2, \ldots)$
- $E = \{e_{v_i - v_j}, e_{v_i - s_j}, e_{v_i - s'_j}, e_{s_i - s'_j}\}$ whenever there is a connection
- Breadth First Search (BFS) to compute CC

Examples of CC

$$\Phi = (a_1^2 + a_2) + r(a_1 . f_3 . a_3) + r^2(a_0 + a_2 . f_3)$$



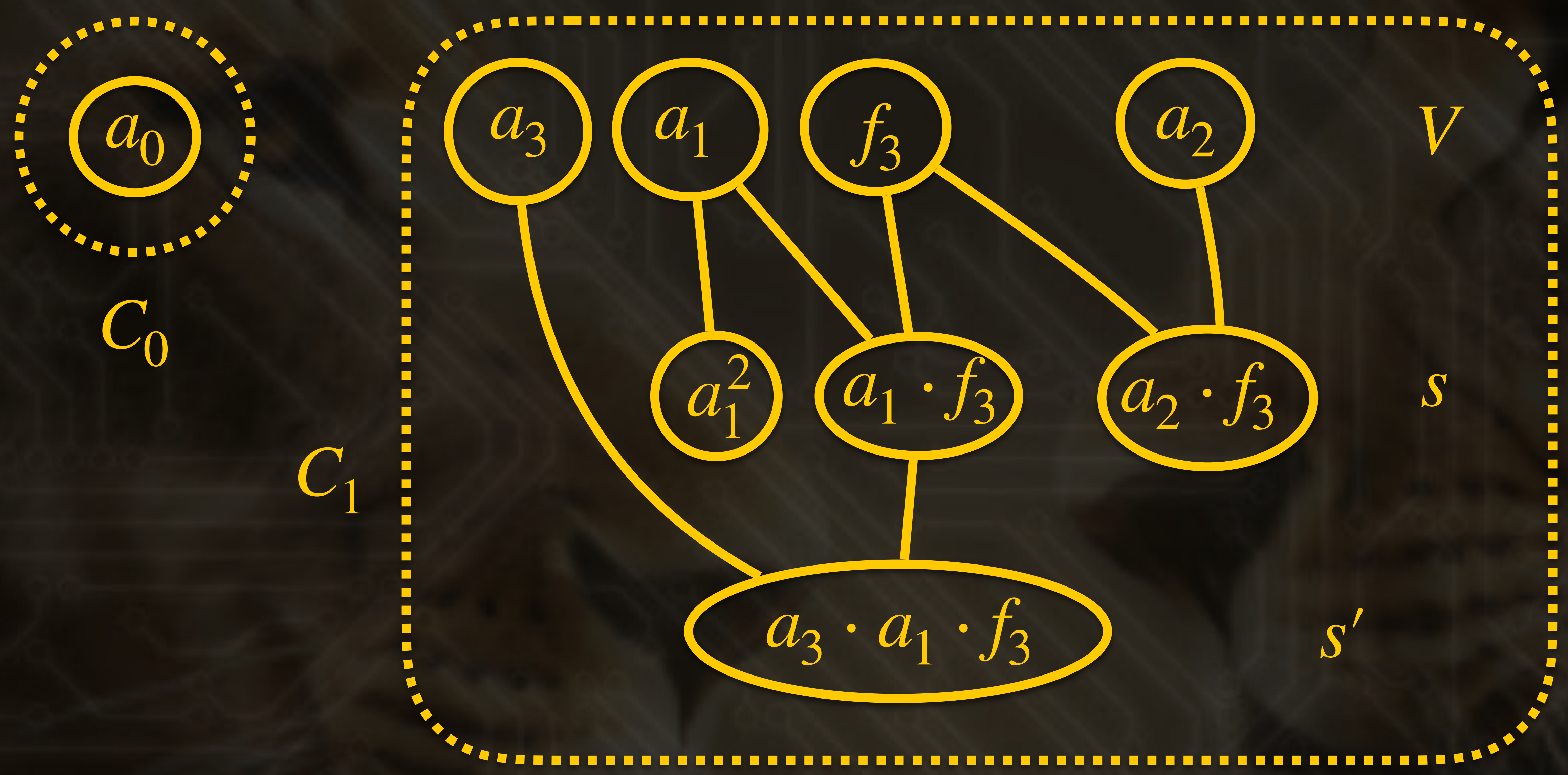
$$|C_0| = 1 \ , \ |C_1| = 8 \ , \ deg(C_0) = 1 \ , \ deg(C_1) = 4$$

- For each $c_i \in C$, extend to $deg(C)$, and evaluate
- If there are disjoint CC, parallel compute!

Example: comparison with clusters

$$\Phi = (a_1^2 + a_2) + r(a_1 . f_3 . a_3) + r^2(a_0 + a_2 . f_3)$$

$Coset_0NTT(a_0)$



$Coset_0NTT(a_1)$ $Coset_1NTT(a_1)$ → $Coset_0NTT(a_1)$ $Coset_1NTT(a_1)$ $Coset_2NTT(a_1)$ $Coset_3NTT(a_1)$

$Coset_0NTT(a_2)$ → $Coset_0NTT(a_2)$ $Coset_1NTT(a_2)$ | $Coset_0NTT(a_3)$ $Coset_1NTT(a_3)$ $Coset_2NTT(a_3)$ $Coset_3NTT(a_3)$

$Coset_0NTT(f_3)$ $Coset_1NTT(f_3)$ → $Coset_0NTT(f_3)$ $Coset_1NTT(f_3)$ $Coset_2NTT(f_3)$ $Coset_3NTT(f_3)$

- CC in a graph: Identify independently computable expressions/sub-expressions

Applications: CC

- reduce unnecessary NTT's: elements in a CC set - extend only to max degree of the CC set

- several disjoint CC sets enhances parallelizability

Eg: PSE Tx circuit $|C_0| = 30 \ , \ |C_1| = 68$

- For large circuits: one big connected component :(

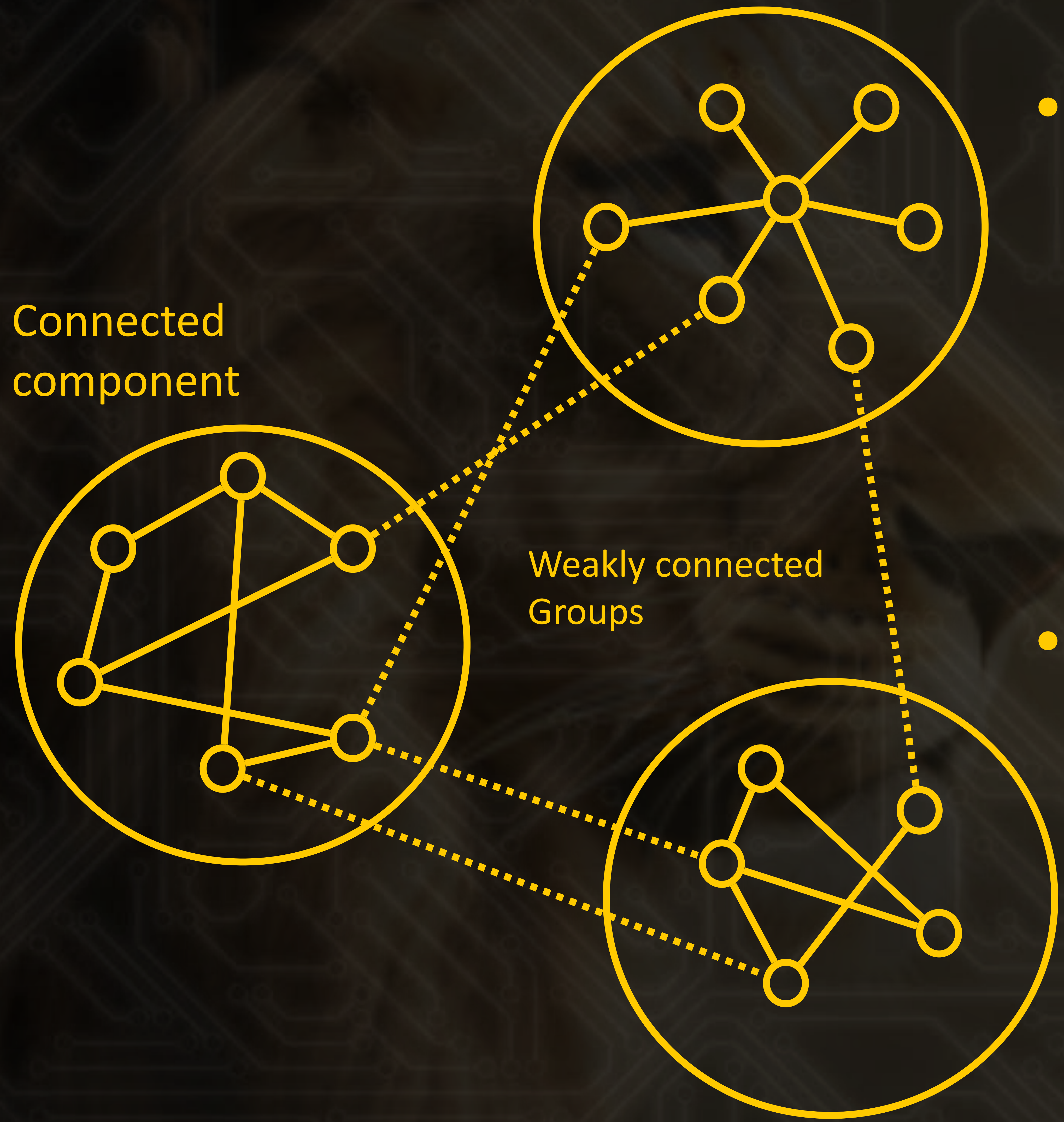
Eg: PSE super-circuit $|C_0| = 499 \ , \ |C_1| = 1 \ , \ |C_2| = 52 \ , \ |C_3| = 3$

- Why? permutations and lookups increase connectivity of the graph - data flow dependency

- In these cases: Can we find weakly connected clusters of CC? - community detection

3. Community detection

Community detection methods



- Communities: a graph G, split into k groups
  - ⋆ Intra-group edges are dense (CC)
  - ⋆ Inter-group edges are sparse
- Heuristic solutions by optimizing
  - ⋆ modularity of partition
  - ⋆ Edge-betweenness (Girvan-Newmann)
  - ⋆ Max flow - Min cut etc

Girvan-Newman algorithm

- Edge Betweenness centrality: Edges that connect communities have high “betweenness”

$$EB(e \in E(G)) = \sum_{v_1, v_2 \in V} \frac{\sigma_{v_1,v_2}(e)}{\sigma_{v_1,v_2}}$$

# of shortest paths between v_1, v_2 that include edge e

# of shortest paths between v_1, v_2

- Successively peeling off “high traffic” edges, reveals community structure

- Recursive application, results in communities, sub-communities, sub-sub communities etc

https://github.com/ChickenLover/petgraph/blob/master/src/algo/community.rs

https://arxiv.org/abs/cond-mat/0308217

Building the graph - heuristics once again

- $G = (V, E)$
- $V = \{v_1, v_2, \ldots v_n\}$, v_i are columns , $E = \{e_{v_i - v_j}\}$ are the edges
- $E \leftarrow e_{v_i - v_j}$ if v_i, v_j belong to
    - same custom gate
    - same permutation set
    - same lookup argument
- $GN(G) \rightarrow \{C_0, C_1, \ldots,\}$, $E_{high\ traffic} = \{e_{v_i - v_j}\}$ such that $|C_0| \geq |C_1| \geq \ldots$
- C_i contain subsets of V (columns)
- Optimization: Minimize number of high traffic edges

Evaluating h poly using the graph data

- Merge smaller communities (greedy) to get them to be of similar size
- A good 2 way split : largest community is 50-60% of total circuit size
- Largest: $bin_1 \leftarrow C_0$, $bin_2 \leftarrow merge\{C_1, C_2, \ldots\}$, $E_{copy} \leftarrow merge(E_{high\ traffic})$
- Assign columns to bins: $\forall\ e_{v_i - v_j} \in E$
  - ★ If $v_i \in bin_1$ & $v_j \in bin_2$, copy v_i to bin_2 — Inter-group edges
  - ★ # of copied columns = $|E_{copy}|$ (computational overhead)
  - ★ Idea is to have minimal $|E_{copy}|$ for a given split — Optimization criteria

Evaluating h poly using the graph data

- Assign constraints data (or part of constraints) to $bin_i \ \forall i$
  - ★ $\{v_k^{(i)}\}$ columns
  - ★ $\{G_k^{(i)}\}$ Gates
  - ★ $\{P_k^{(i)}\}$ Permutations
  - ★ $\{L_k^{(i)}\}$ Lookups

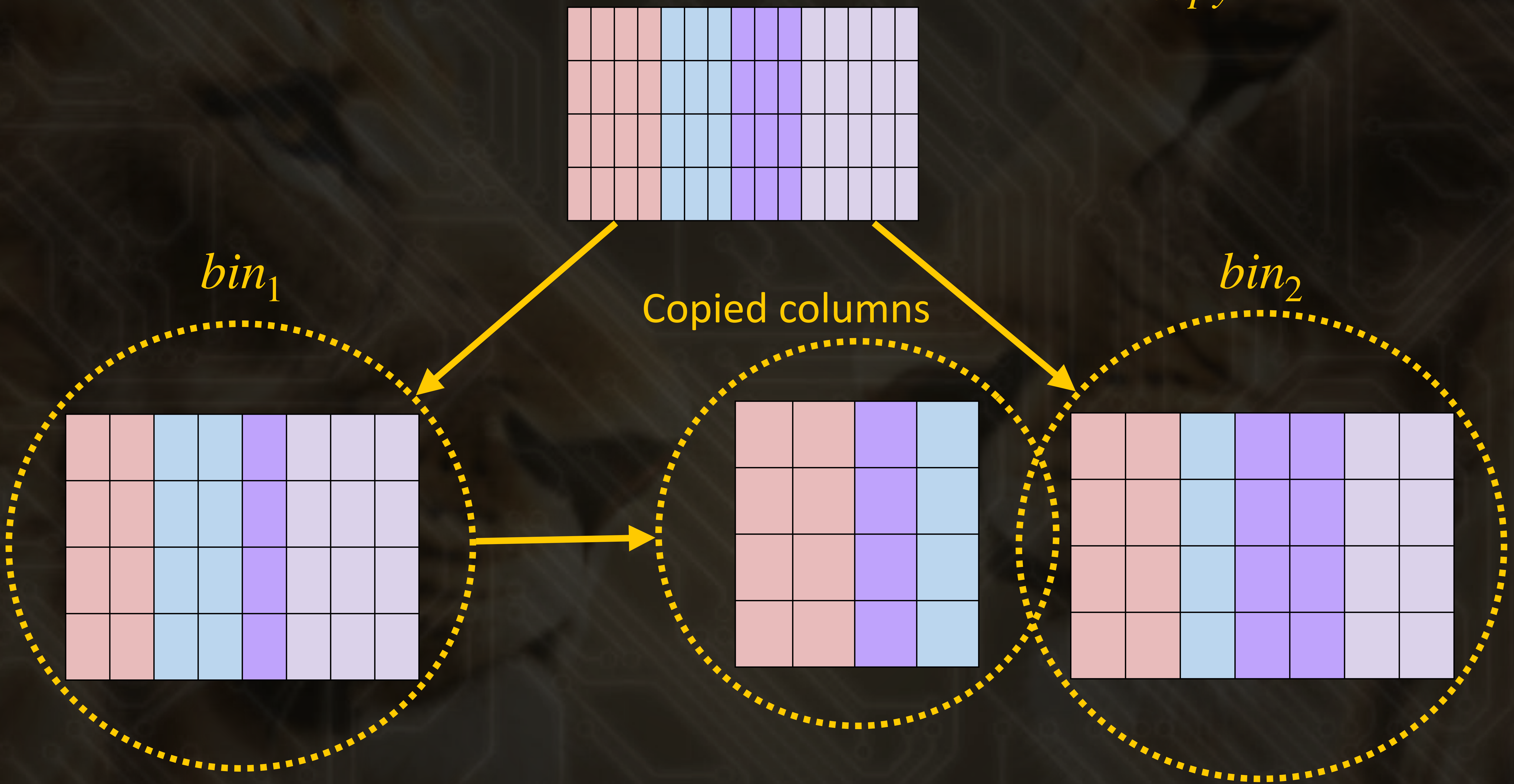
- wrap inside h poly evaluation code

- <u>Caution</u>: take care of powers of “y” and combine the results $bin_1 + bin_2$ to get final h poly

```rust
    /// Evaluate h poly
    pub(in crate::plonk) fn evaluate_h(
        &self,
        pk: &ProvingKey<C>,
        advice_polys: &[&[Polynomial<C::ScalarExt, Coeff>]],
        instance_polys: &[&[Polynomial<C::ScalarExt, Coeff>]],
        challenges: &[C::ScalarExt],
        y: C::ScalarExt,
        beta: C::ScalarExt,
        gamma: C::ScalarExt,
        theta: C::ScalarExt,
        lookups: &[Vec<lookup::prover::Committed<C>>],
        permutations: &[permutation::prover::Committed<C>],
    ) -> Polynomial<C::ScalarExt, ExtendedLagrangeCoeff> {
        let domain = &pk.vk.domain;
        let mut values = domain.empty_extended();
        for (b, bin) in self.bins.iter().enumerate() {
            let start = start_measure(format!("BIN {}", b), false);
            println!("Processing {} bin", b);
            let bin_advice_polys = advice_polys
                .iter()
                .map(|advice_polys| {
                    advice_polys
                        .iter()
                        .enumerate()
```

Bottomline

$$GN(G(E, V)) \rightarrow bin_1 \oplus (bin_2 + E_{copy})$$



H poly in bin_1 and bin_2 can run independently in parallel: For eg in 2 GPU's

Efficiently solve memory bottlenecks by "splitting" the trace

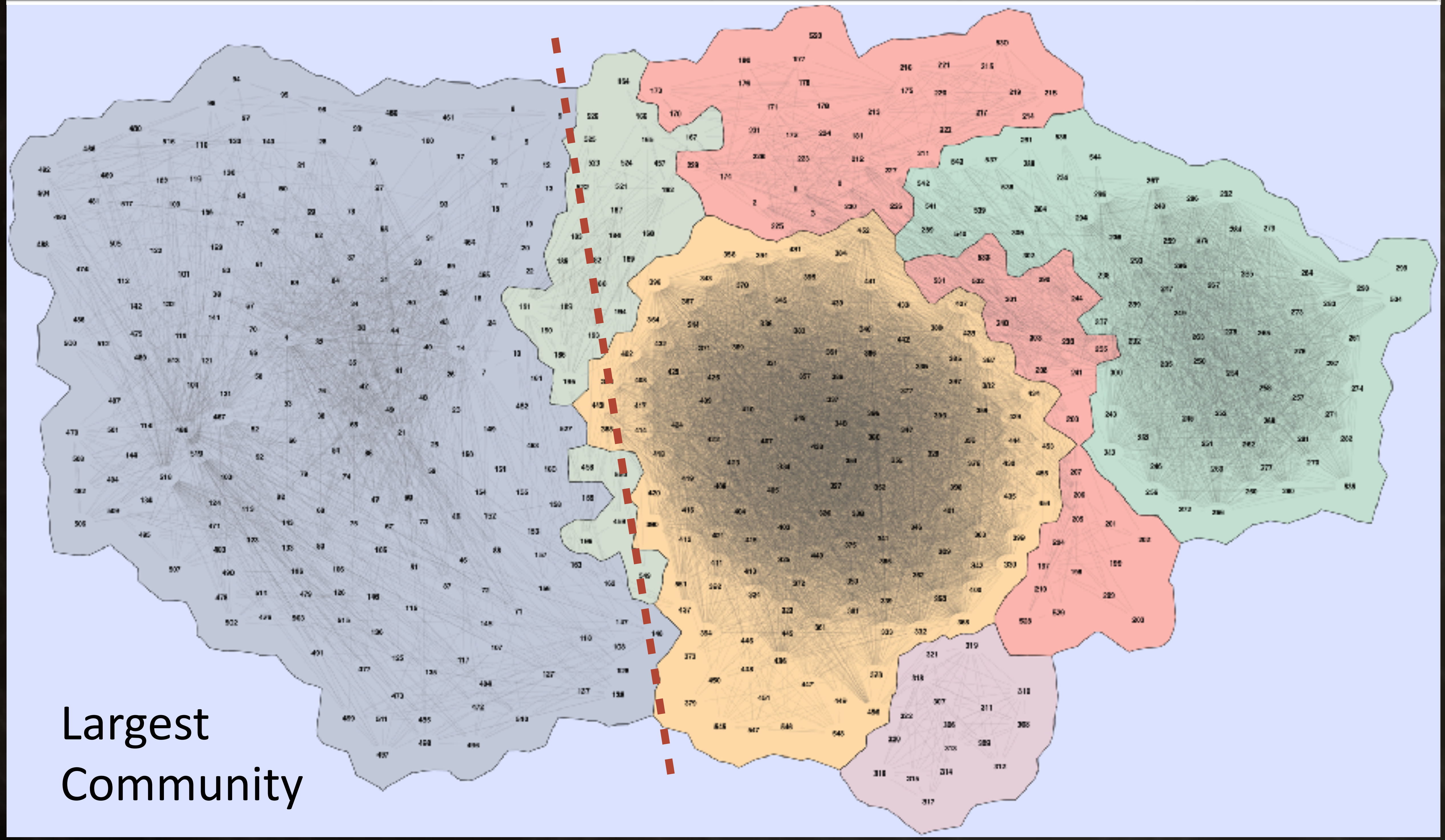
Trade off space: recompute NTTs for the copied columns

Applications: community detection

Community detection in Taiko zkEVM

i9-12900K 16 core CPU with 24 threads

Data	Base level	Net:ours	Bin 1	Bin 2
Columns	552	557	303	254
Instance	2	2	0	2
Advice	484	489	274	215
Fixed	66	66	29	37
Permutation sets	3	3	1	2
Lookups	244	244	114	130
Total Instance size	20.15 GB	20.51 GB	10.20 GB	10.31 GB
h-poly time	623 secs	826 secs	552 secs	274 secs



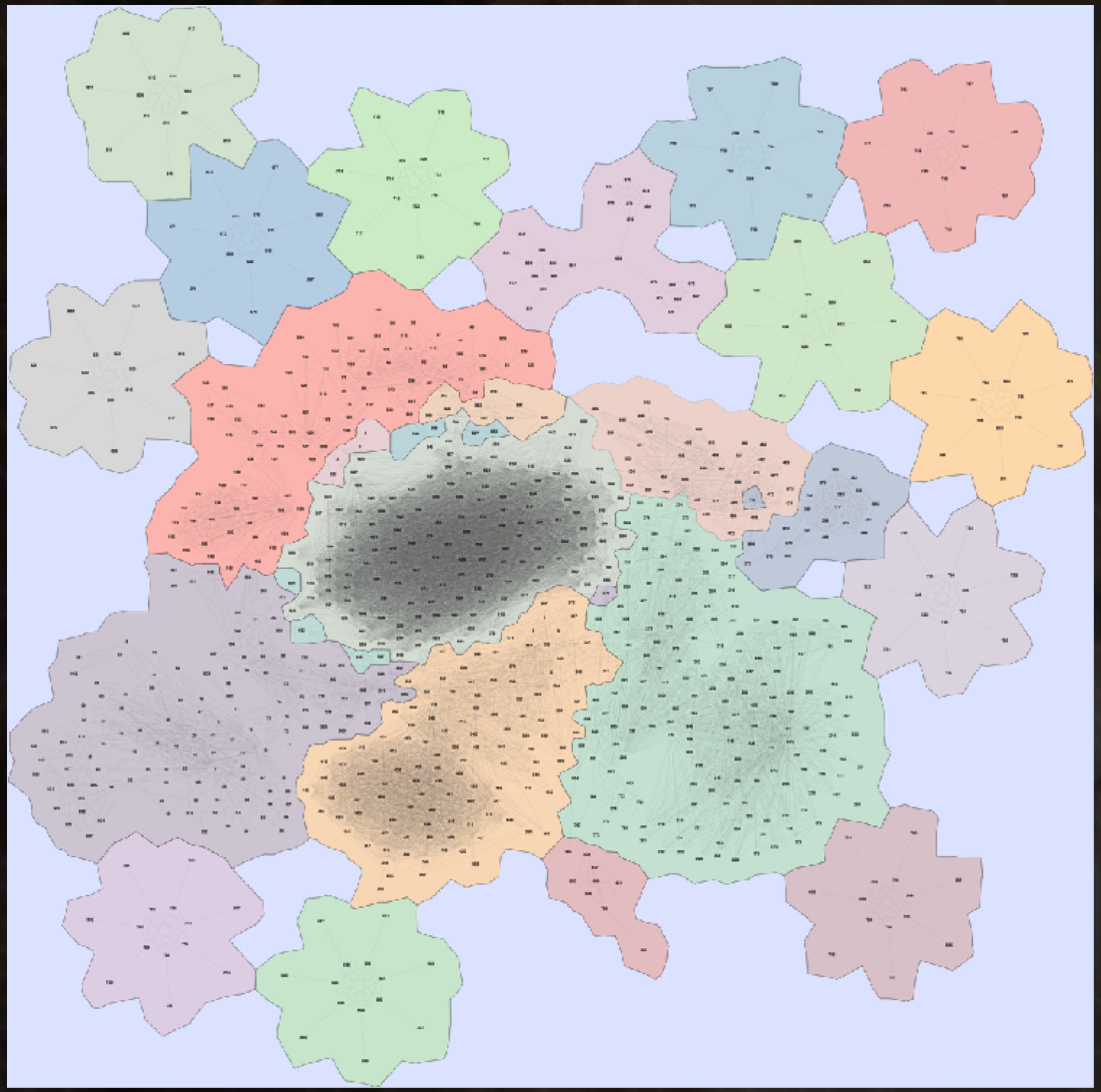
- Girvan-Newman partition on Super circuit

$$|C_1| = 303 \ , \ |C_2| = 254 \ , \ |E_{copy}| = 23$$

- largest community is 54 % of total circuit size
- Bin1 and Bin2 run independently in parallel!
- Solves poly memory bottleneck!
- Reduces latency to run time of largest community
- Reproduces correct h poly result

Rendering with gvmap (on graph data) highlighting different sub-communities as countries

Community detection in Scroll zkEVM



Rendering with gvmap (on graph data) highlighting different sub-communities as countries

- Large community is 68% of total circuit size

Circuit	Connected components	Merged communities	Edges to copy
Super circuit $k = 2$	$\lvert C_1\rvert = 658, \lvert C_2\rvert = 88, \lvert C_3\rvert = 26,$ $\lvert C_5\rvert = \ldots = \lvert C_{16}\rvert = 14$	$\lvert C_1\rvert = 658$ $\lvert C_{2-16}\rvert = 309$	$\lvert E_{copy}\rvert = 19$
Super circuit $k = 3$	$\lvert C_{1,1}\rvert = 564, \lvert C_{1,2}\rvert = 94, \lvert C_2\rvert = 88, \lvert C_3\rvert = 26$ $\lvert C_4\rvert = 20, \lvert C_5\rvert = \ldots = \lvert C_{16}\rvert = 14$	$\lvert C_{11}\rvert = 564$ $\lvert C_{1,2-16}\rvert = 411$	$\lvert E_{copy}\rvert = 35$

- Recursive application - finds sub communities

- Large # of permutation columns = long connected components

Data	Super ciruit	EVM	State	keccak	byte code	pi	tx	exp
Columns	971	236	91	112	26	35	211	43
Advice	736	211	84	94	20	20	139	39
Instance	1	0	0	0	0	1	0	0
Fixed	234	25	7	18	6	14	72	4
Lookups	113	20	14	27	2	1	17	14
Permutation columns	190	8	1	0	0	12	75	0

Summary

Summary

- graph methods to analyze parallelizability of circuits
- Connected components (CC) - organize constraint expressions
  - largest NTT in a CC set is only the highest degree in the CC set
  - If many disjoint CC - parallelizability
- Community detection - Weakly connected CC groups
  - GN algorithm - High traffic edges connect communities
  - Large number of permutations - bigger CC and fewer communities
- Application - Parallelizing zkEVM circuit (Taiko)
  - Use GN to identify communities in super circuit
  - Solve memory bottleneck and latency by parallelizing computation

Thank you!