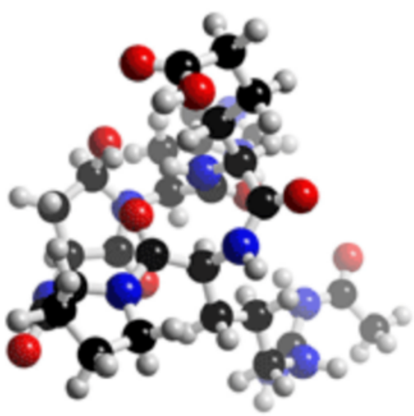
High Performance Graph Mining Systems

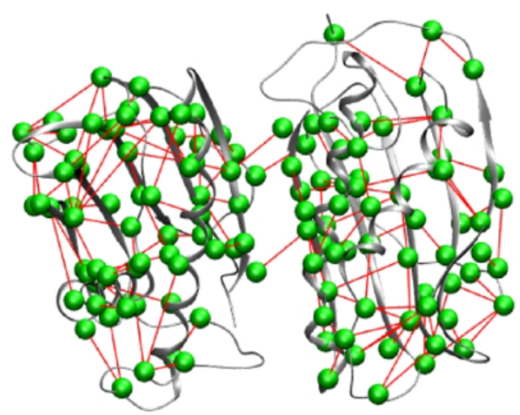
Xuehai Qian
University of Southern California

Graphs are Everywhere

- A **graph** $G = (V, E)$ is represented by its vertices set and edges set E:
  - E is a subset of $V \times V$, $(u, v) \in E$ iff u and v are connected by an edge
- Graphs naturally capture the relationship between entities in different applications

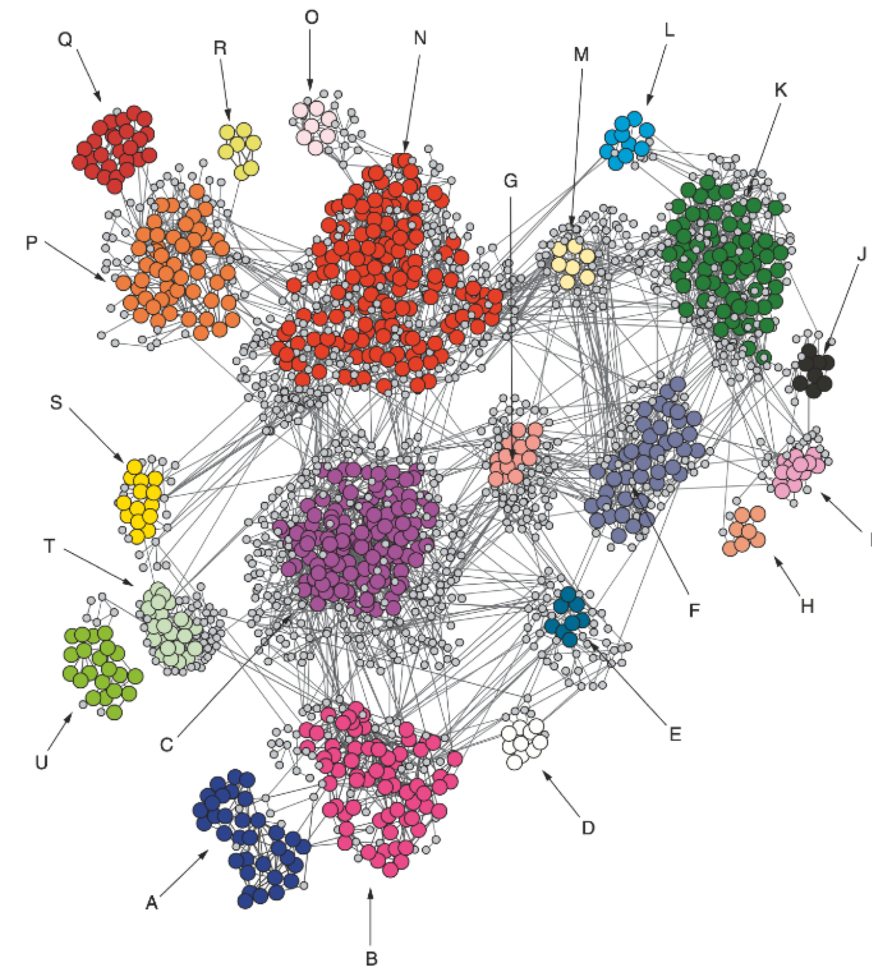


Chemical compound



Protein structure

Social networks
[Grandjean, M. (2016)]



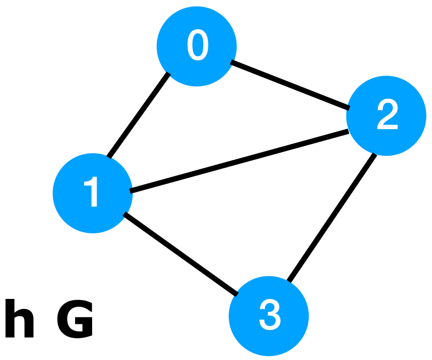
Coexpression networks
[Magwene et al. (2004)]

Graph Mining

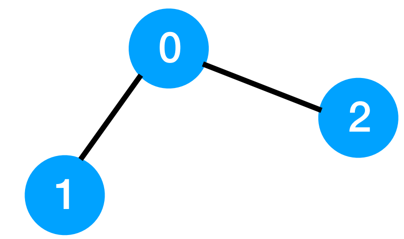
- Two graphs $G_0 = (V_0, E_0)$ and $G_1 = (V_1, E_1)$ are **isomorphic** iff there exists an one-to-one mapping $f : V_0 \to V_1$ such that $(u, v) \in E_0 \Leftrightarrow (f(u), f(v)) \in E_1$
  - An equivalent relation
- **Graph mining**: find patterns from a graph
  - Input: a large input graph; a pattern graph
  - Compute: enumerate all the subgraphs isomorphic to the given pattern—embeddings
  - Process: gather some information, depending on the application
- We consider two main types:
  - Counting: simply return the count of embeddings
  - Frequent subgraph mining (FSM)

Edge/Vertex-induced Subgraphs

- Edge-induced subgraph
  - For two graphs $g = (V_g, E_g)$, $G = (V, E)$ such that $V_g \subseteq V, E_g \subseteq E$
  - Consider edges: $u, v \in g$, $(u, v) \in G$, but $(u, v) \notin g$—u and v are in g due to other edges
- Vertex-induced subgraph
  - For two graphs $g = (V_g, E_g)$, $G = (V, E)$ such that $\{(u, v) \mid u, v \in V_g, (u, v) \in E\} = E_g$
  - Consider vertices: if two vertices u and v are in g, and there is an edge between then in G, the edge must be also in g

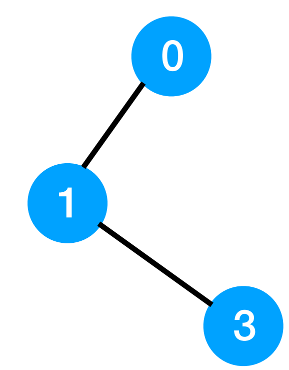


Graph G



Edge-induced subgraph

***Not* vertex-induced subgraph**

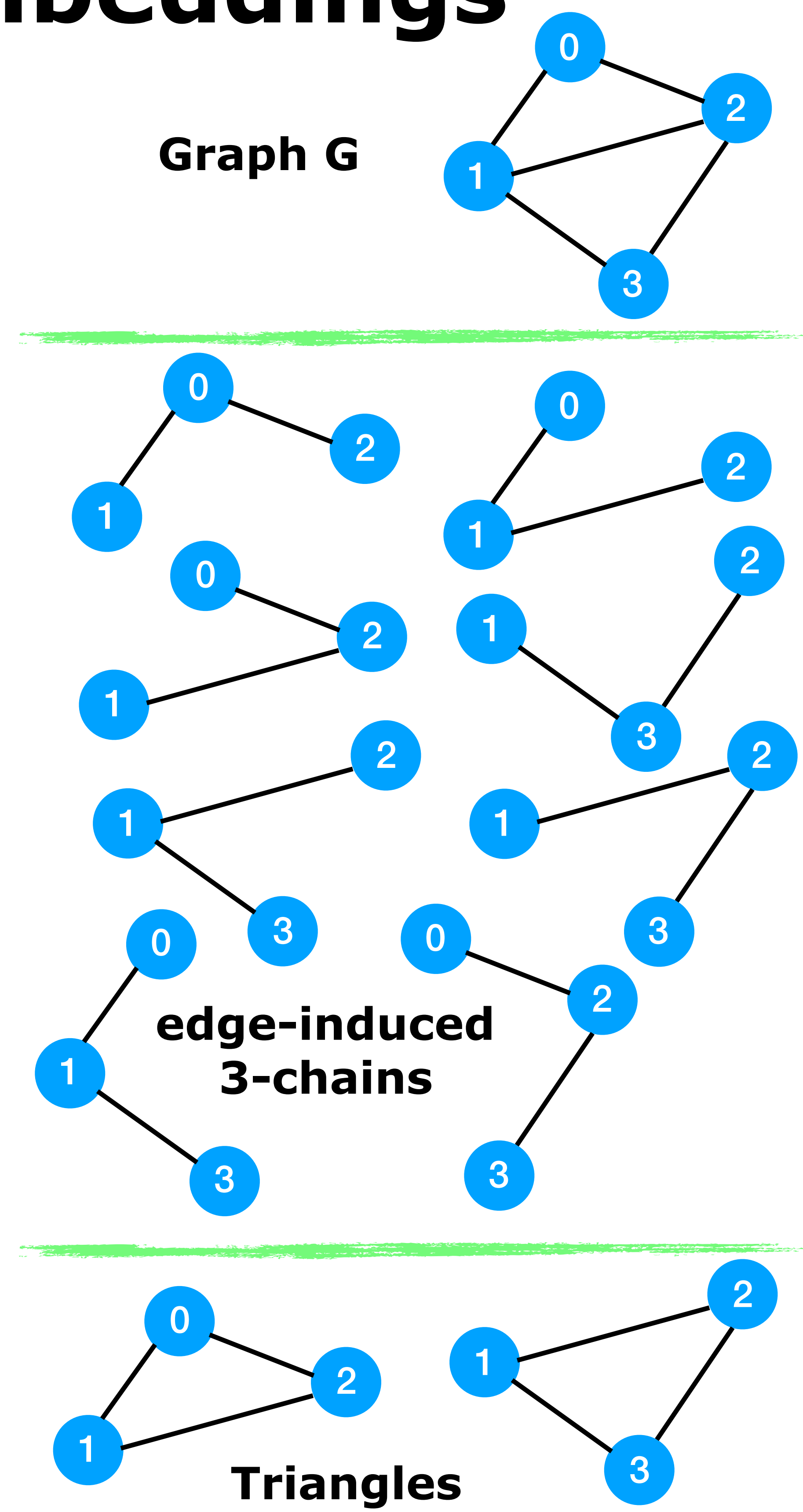


Edge-induced subgraph

Vertex-induced subgraph

Edge/Vertex-induced Embeddings

- Vertex-induced embedding
  - The subgraph that is isomorphic to a pattern should be a valid vertex-induced subgraph
  - The count be calculated from edge-induced embedding count
  - Example: C(vertex-induced 3-chain) =C(edge-induced 3-chain)-3C(edge-induced triangle). C=count
  - The #of vertex-induced count of 3-chain of G (on the right)=8-3×2=2

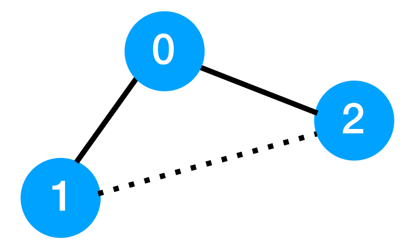


Edge/Vertex-induced Embeddings

- For the vertex-set-based method (used in AutoMine*), the edge-induced embedding can be calculated with minor code modification

for $v_0 \in V$ **do**
  for $v_1 \in N(v_0)$ **do**
    for $v_2 \in N(v_0) - N(v_1)$ **do**
      $count_{vertex-induced} \leftarrow count_{vertex-induced} + 1$
    end for
  end for
end for

Vertex-induced 3-chain

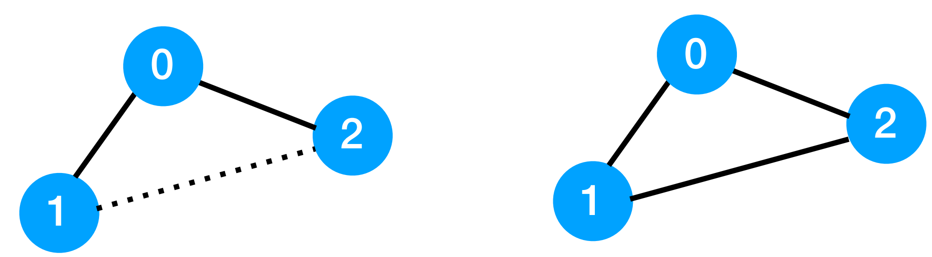


N(v): the vertex set containing all neighbors of v

-N(v_1): v_2 should not connect to v_1, otherwise it is triangle in the original graph—the 3-chain is not a valid vertex-induced subgraph

for $v_0 \in V$ **do**
  for $v_1 \in N(v_0)$ **do**
    for $v_2 \in N(v_0)$ **do**
      $count_{edge-induced} \leftarrow count_{edge-induced} + 1$
    end for
  end for
end for

Edge-induced 3-chain



Remove -N(v_1): cover both cases

- In the talk, we consider edge-induced embeddings

* Daniel Manwhirter, et al. AutoMine: harmonizing high-level abstraction and high performance for graph mining. SOSP'19

Graph Mining Applications

- Mining biochemical structures
- Finding biological conserved subnetworks
- Finding functional modules
- Program control flow analysis
- Intrusion network analysis
- Mining communication networks
- Anomaly detection
- Mining XML structures
- Building blocks for graph classification, clustering, compression, comparison, correlation analysis, and indexing
- …

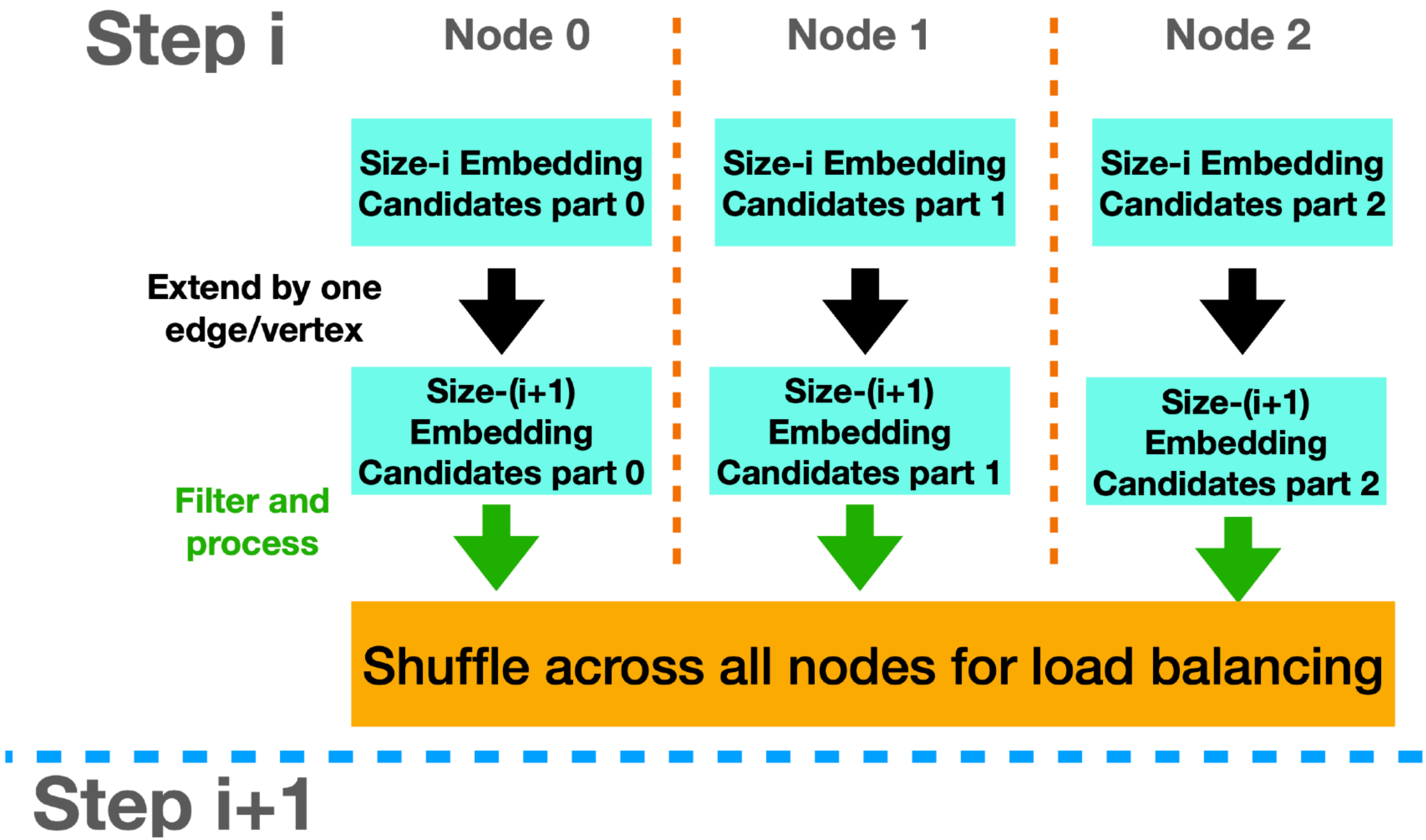
Graph Mining Systems

- While it is important, it is hard to write graph mining codes for various applications
  - Different patterns leads to different algorithms
  - The same algorithm can be implemented in various ways with different performance
- A general graph mining system can offer better programmability and high performance
  - Users simply specify the patterns
  - The system chooses the best implementations
  - A typical domain-specific system with similar motivation as graph processing systems
- Graph mining vs. graph computation
  - Graph computation: simple computation, memory bound
  - Graph mining: **computational intensive**

Existing Graph Mining Systems

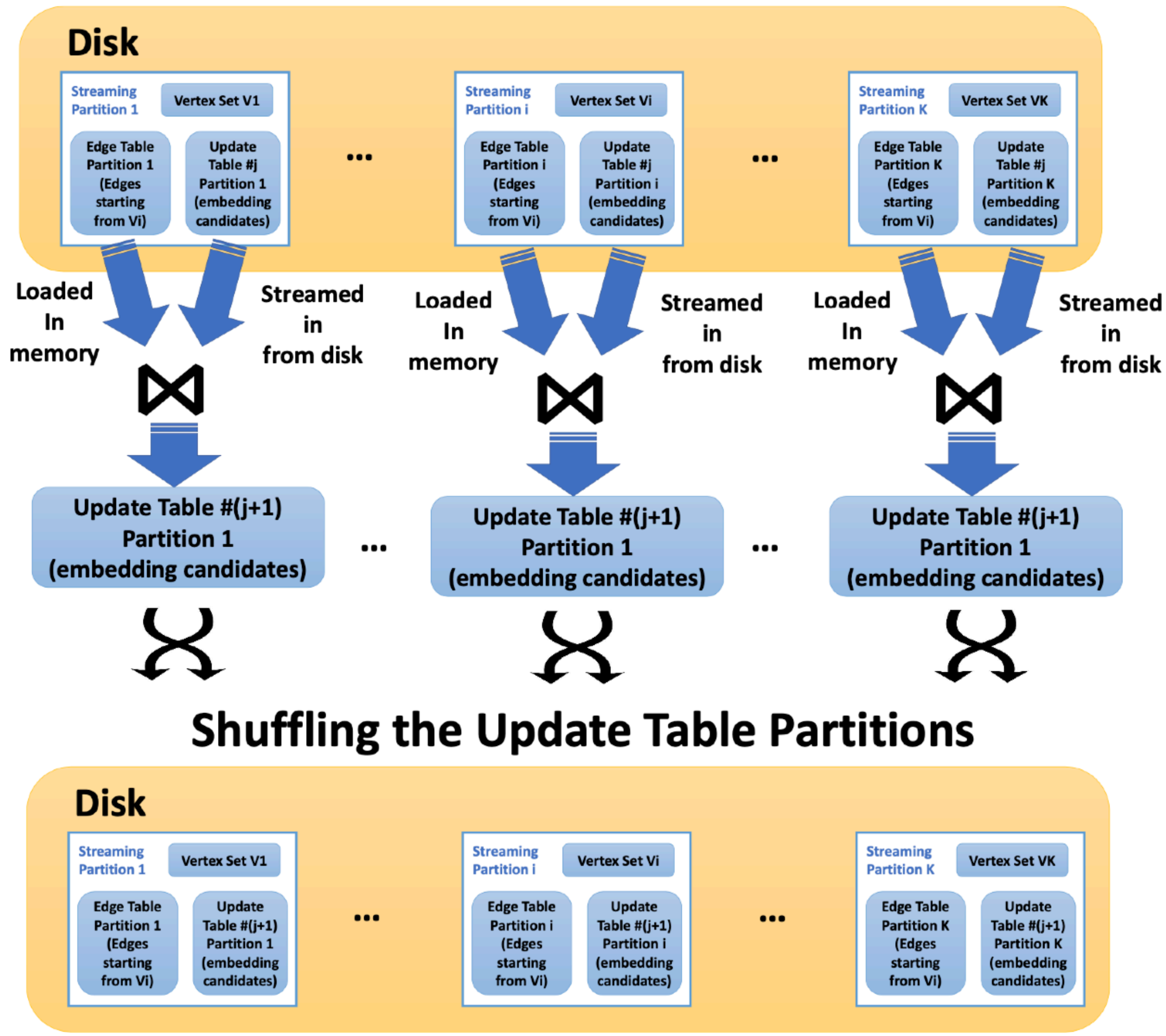
- Single-machine systems
  - RStream (OSDI'18)
  - AutoMine (SOSP'19)
  - Peregrine (EuroSys'20)
  - Pangolin (VLDB'20)
  - Kaleido (ICDE'20)
- Distributed systems
  - Arabesque (SOSP'15)
  - G-Miner (Eurosys'18), G-Thinker (ICDE'20)
  - Fractal (SIGMOD'19)
  - GraphPi (SC'20)

Arabesque: Exhaustive Check



Teixeira et al. Arabesque: A System for Distributed Graph Mining. SOSP'15

RStream: Leveraging Relational Algebra



Wang et al. RStream: Marrying Relational Algebra with Streaming for Efficient Graph Mining on A Single Machine. OSDI'18

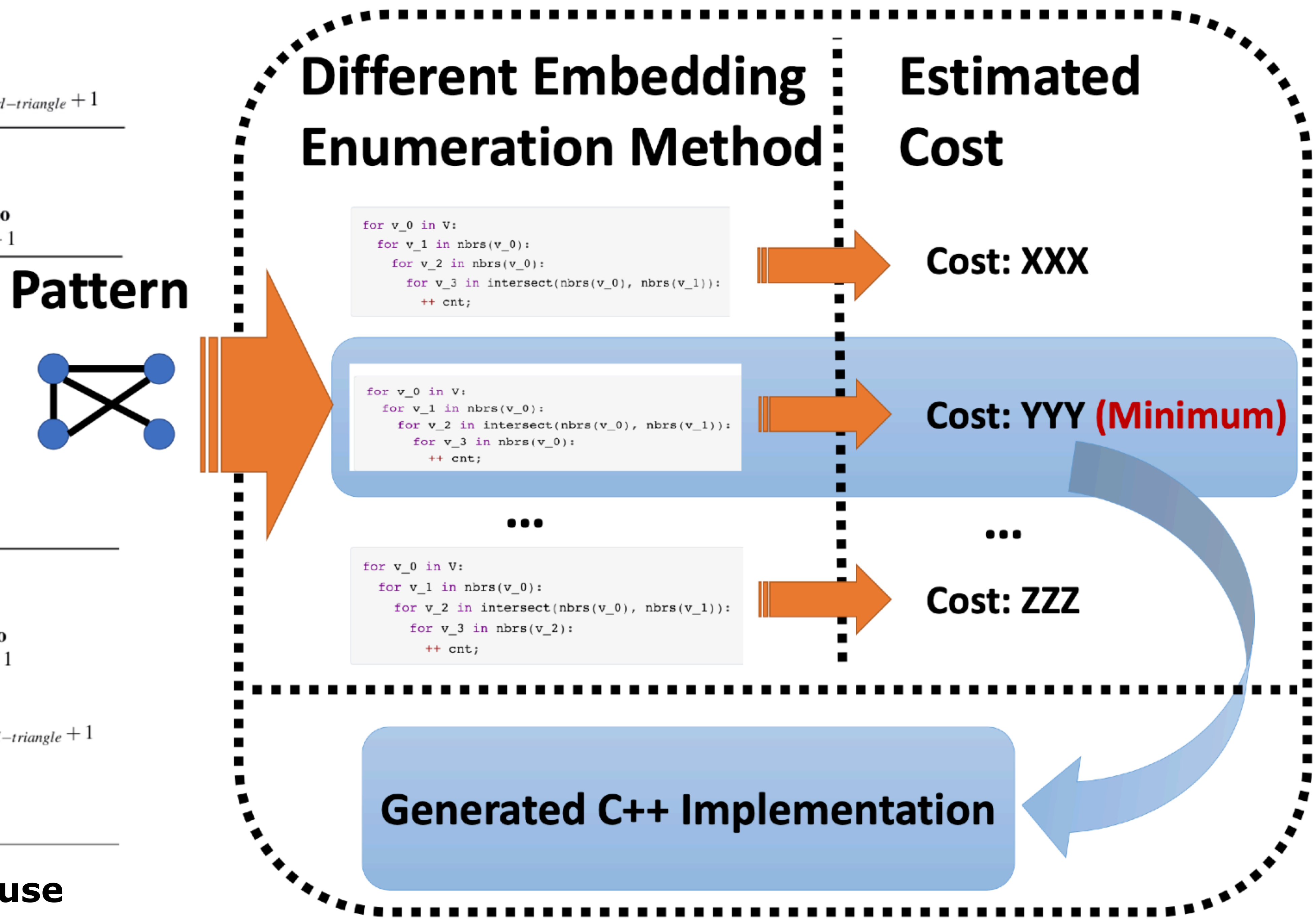
AutoMine: Compiler-generated Pattern Enumeration using Cost Model

for $v_0 \in V$ **do**
  for $v_1 \in N(v_0)$ **do**
    for $v_2 \in N(v_0) \cap N(v_1)$ **do**
      for $v_3 \in N(v_0)$ **do**
        $count_{tailed-triangle} \leftarrow count_{tailed-triangle} + 1$

for $v_0 \in V$ **do**
  for $v_1 \in N(v_0)$ **do**
    for $v_2 \in N(v_0) \cap N(v_1)$ **do**
      for $v_3 \in N(v_0) \cap N(v_1) \cap N(v_2)$ **do**
        $count_{4-clique} \leftarrow count_{4-clique} + 1$

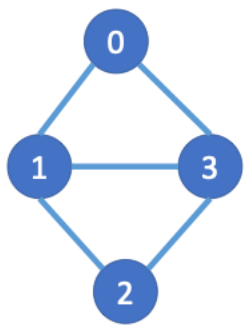
for $v_0 \in V$ **do**
  for $v_1 \in N(v_0)$ **do**
    for $v_2 \in N(v_0) \cap N(v_1)$ **do**
      for $v_3 \in N(v_0) \cap N(v_1) \cap N(v_2)$ **do**
        $count_{4-clique} \leftarrow count_{4-clique} + 1$
      end for
      for $v_3 \in N(v_0)$ **do**
        $count_{tailed-triangle} \leftarrow count_{tailed-triangle} + 1$
      end for
    end for
  end for
end for

Computation Reuse



Mawhirter et al. AutoMine: Harmonizing High-level Abstraction and High Performance for Graph Mining. SOSP'19

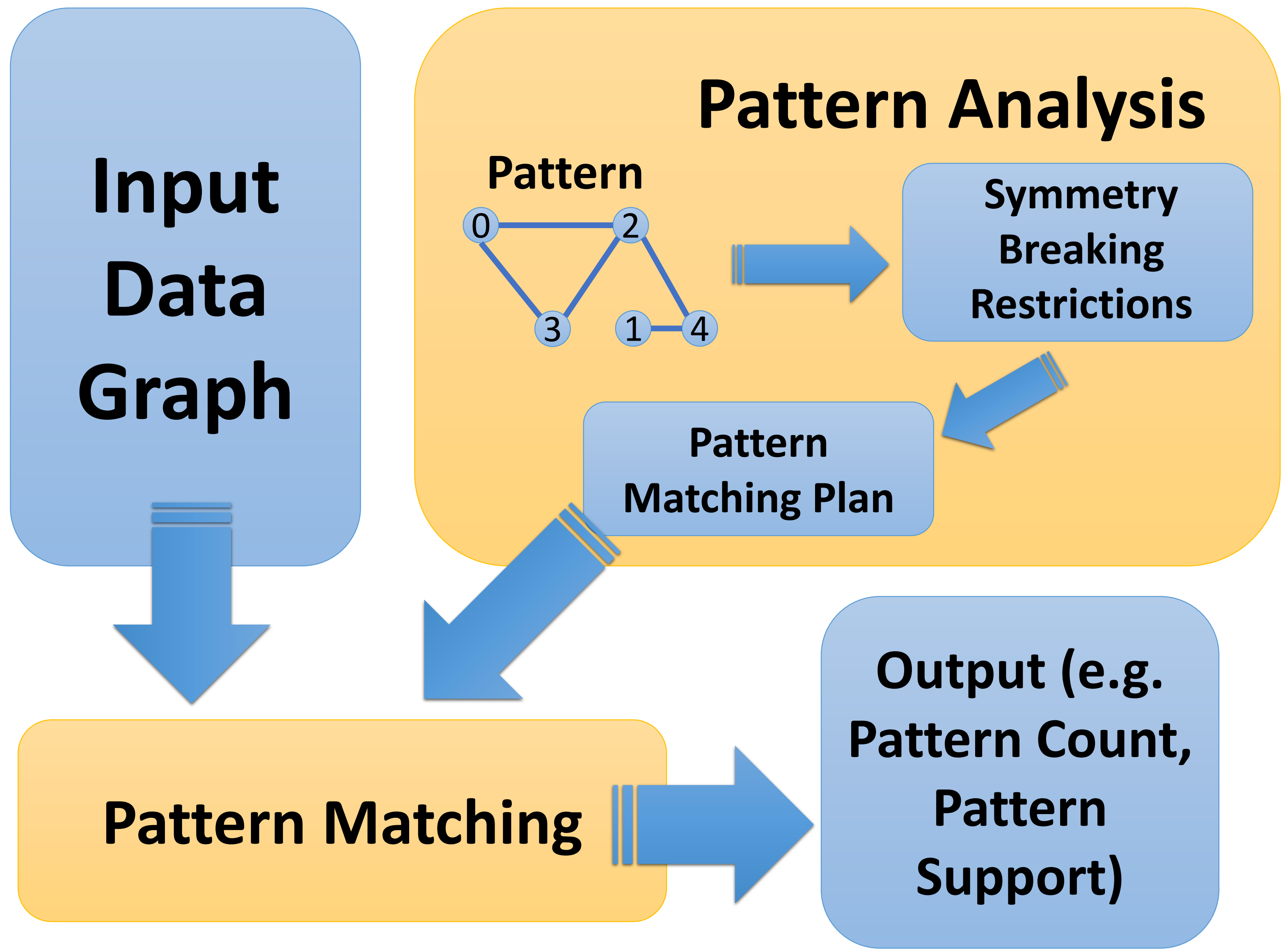
Peregrine: Complete Symmetry Breaking



A Symmetric Pattern

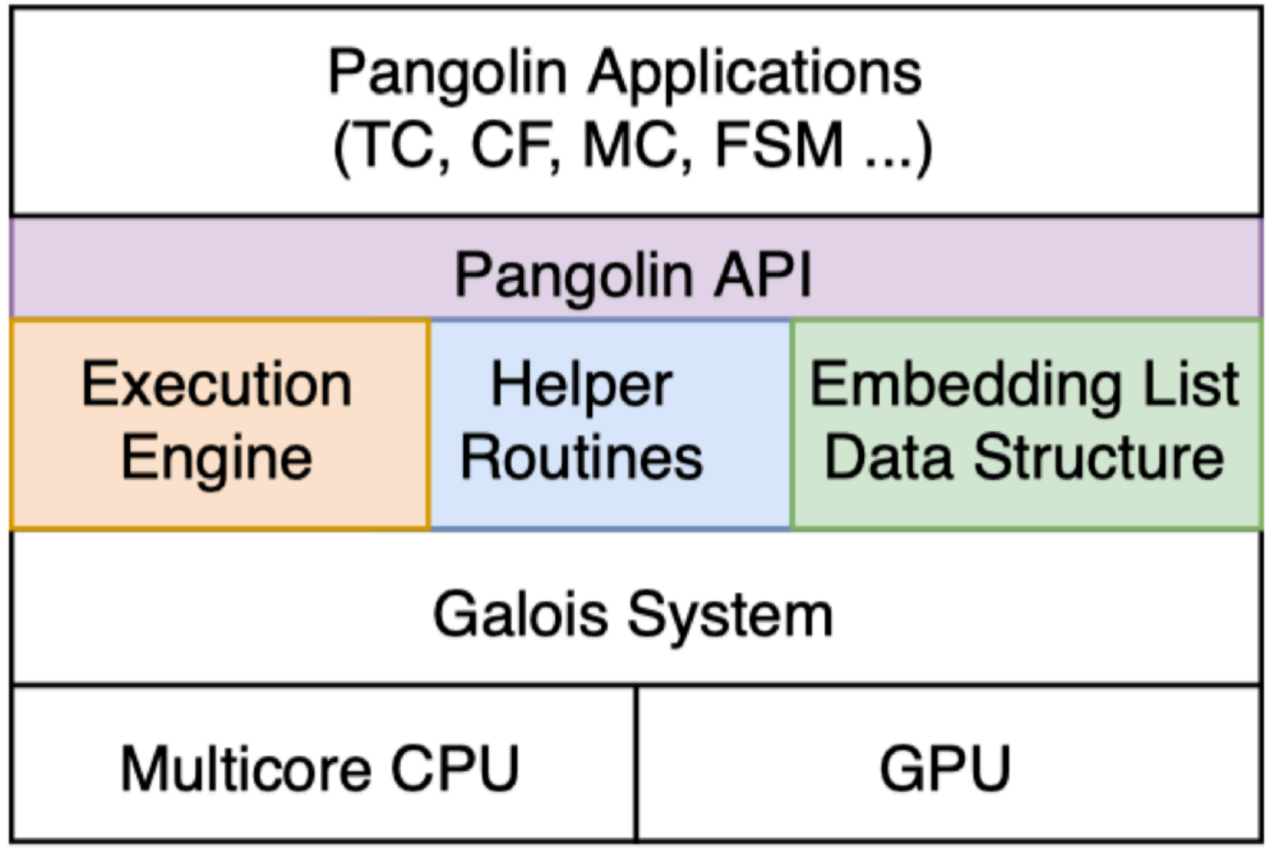
The four embeddings are redundant:
(v0,v1,v2,v3)
(v2,v1,v0,v3)
(v2,v3,v0,v1)
(v0,v3,v2,v1)
➜ should be only counted once

Symmetry breaking: add constraints—
(v0<v2) and (v1<v3)
➜ only one embedding left



Jamshidi et al. Peregrine: A Pattern-Aware Graph Mining System. EuroSys'20

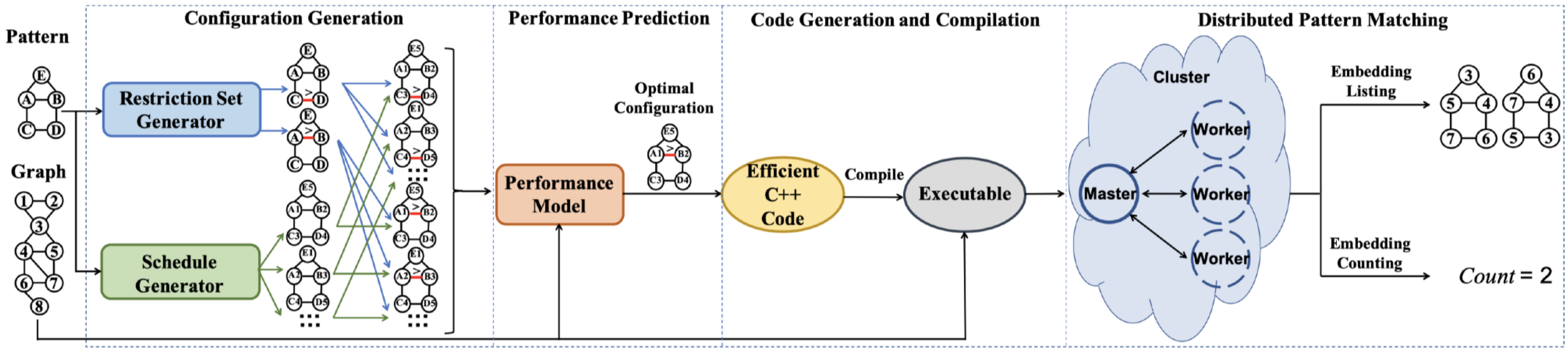
Pangolin: Application-specific Optimizations with flexible APIs




```
bool toExtend(Embedding emb, Vertex v);
bool toAdd(Embedding emb, Vertex u)
bool toAdd(Embedding emb, Edge e)
Pattern getPattern(Embedding emb)
Pattern getCanonicalPattern(Pattern pt)
Support getSupport(Embedding emb)
Support Aggregate(Support s1, Support s2)
bool toPrune(Embedding emb);
```

Chen et al. Pangolin: An Efficient and Flexible Graph Mining System on CPU and GPU. VLDB'20

GraphPi: Choose the Best Symmetry Breaking



- Introduced the restriction set generator
  - Systematically explore various symmetry breaking restrictions
  - Eliminate all redundant computation due to symmetry
- Replicated distributed execution
- If the innermost K for-loops are independent, the count is calculated mathematically

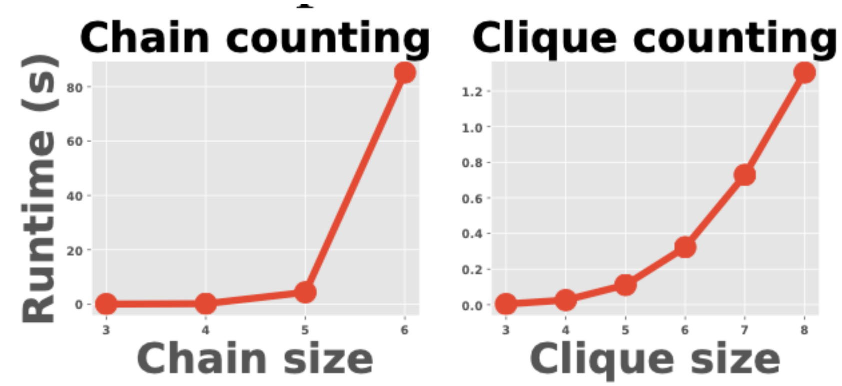
Shi et al. GraphPi: High Performance Graph Pattern Matching through Effective Redundancy Elimination. SC'20

Existing Graph Mining Systems

- **Single-machine systems**
  - RStream (OSDI'18):
    - Relational algebra based API and implementation
  - AutoMine (SOSP'19):
    - Compiler generated algorithms for pattern enumeration
  - Peregrine (EuroSys'20):
    - Pattern-based programming model enabling pattern-aware optimization
  - Pangolin (VLDB'20):
    - A set of flexible APIs enables powerful pattern-specific optimizations
    - The first graph mining system supporting GPU
  - Kaleido (ICDE'20):
    - Succinct intermediate data representation && faster isomorphism test
- **Distributed systems**
  - Arabesque (SOSP'15):
    - Exhaustively check all subgraphs up to the pattern size
  - G-Miner (Eurosys'18), G-Thinker (ICDE'20)
    - Subgraph-centric programming model with partitioned graph
  - Fractal (SIGMOD'19):
    - DFS-based embedding exploration; build-from-scratch paradigm to reduce memory footprint
  - GraphPi (SC'20):
    - Search for better symmetry breaking; an mathematical method to speedup counting

Further Improving Performance?

- Observation: all existing systems consider each given pattern **as a while**
  - Empirically, the embedding enumeration cost can increase rapidly as the pattern size grows
- We build a new graph mining system based on **pattern decomposition***
  - Decompose a target pattern into several smaller subpatterns
  - Compute the count of each
  - The results of the target (original pattern) can be calculated using the subpattern counts with very low additional cost
- Our project started in Fall 2019, many new papers came out in 2020, but fortunately our performance is still better than all
  - The importance of algorithm

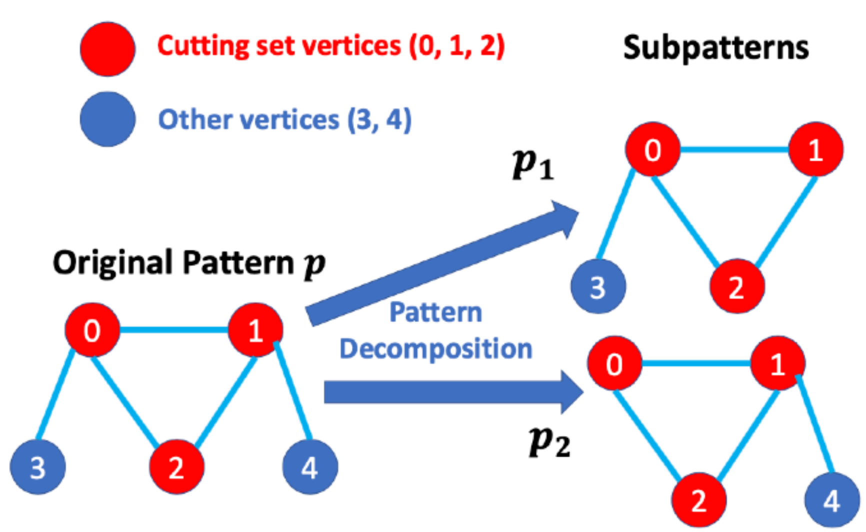


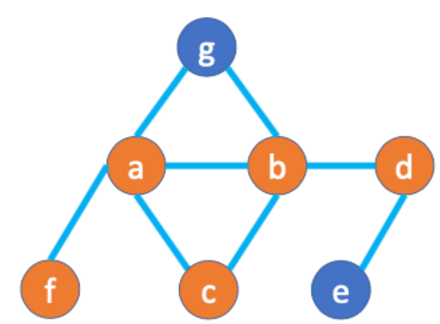
Execution time of AutoMine (our own implementation) on EmailEuCore graph. 6-chain embeddings is 19,620× compared to 3-chain enumeration

* Ali Pinar, et al. Escape: Efficiently counting all 5-vertex subgraphs. WWW'17

Counting with Pattern Decomposition

- We explain the algorithm using relational algebra as a mathematical tool
  - The implementation still uses vertex-set-based method
- A pattern decomposition of pattern graph $p = (V_p, E_p)$ is determined by the vertex cutting set V_C
  - A subset of V_p, of which the removal breaks p into K connected components
  - An edge-induced embedding of p can be represented by a $|V_p|$-tuple $(v_0, v_1, v_2, \ldots, |V_p| - 1)$
    - v_i is the vertex in the embedding (subgraph) that matches the vertex i in the pattern graph
    - Each unique embedding corresponds to M tuples due to symmetric
- We can organize such tuples in a conceptual embedding table





Input graph

(a,b,c,f,d): a tuple for an embedding of p

(b,a,c,d,f): duplicated tuple

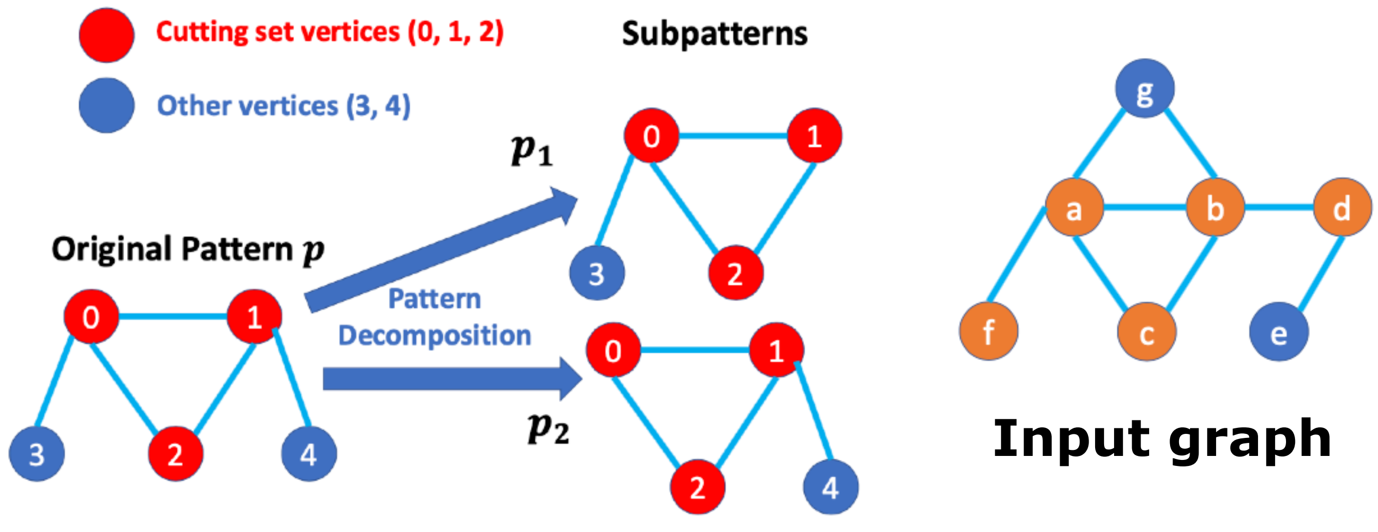
* Ali Pinar, et al. Escape: Efficiently counting all 5-vertex subgraphs. WWW'17

Counting with Pattern Decomposition

- The K subpatterns correspond to K embedding tables: $T_1, T_2, \ldots, T_K$
- T_{K+1}: relational join of all $T_1, T_2, \ldots, T_K$ using the columns associated with the cutting set V_C as keys
  - Contains all edge-induced embeddings of the original pattern p
- However, T_{K+1} contains more tuples for two reasons:
  - Symmetric: different tuples represent the same embedding—valid embeddings counted multiple times
  - *Duplicated elements*: embeddings matching the subpatterns contain one or more same vertices other than for cutting sets—invalid embeddings

* Ali Pinar, et al. Escape: Efficiently counting all 5-vertex subgraphs. WWW'17

Embedding Tables



- However, T_{K+1} contains more tuples for two reasons:
  - Symmetric: different tuples represent the same embedding—valid embeddings counted multiple times
  - *Duplicated element*: embeddings matching the subpatterns contain the same vertices other than for cutting sets—invalid embeddings
- **How to eliminate them?**

T_1

v_0	v_1	v_2	v_3
a	b	g	f
a	g	b	f
a	b	g	c
a	g	b	c
a	c	b	f
a	b	c	f
a	c	b	g
a	b	c	g
b	g	a	d
b	a	g	d
b	g	a	c
b	a	g	c
b	a	c	g
b	c	a	g
b	a	c	d
b	c	a	d

$\bowtie$

T_2

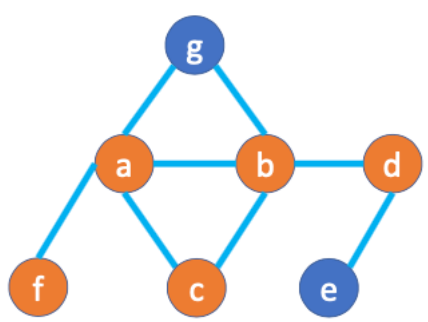
v_0	v_1	v_2	v_4
b	a	g	f
g	a	b	f
b	a	g	c
g	a	b	c
c	a	b	f
b	a	c	f
c	a	b	g
b	a	c	g
g	b	a	d
a	b	g	d
g	b	a	c
a	b	g	c
a	b	c	g
c	b	a	g
a	b	c	d
c	b	a	d

T_3

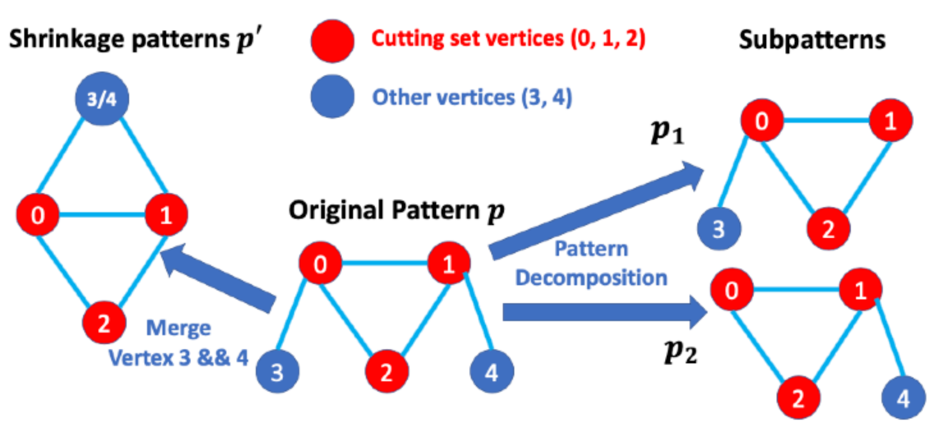
v_0	v_1	v_2	v_3	v_4
a	b	g	f	d
a	b	g	f	c
a	b	g	c	d
a	b	g	c	c
a	b	c	f	g
a	b	c	f	d
a	b	c	g	g
a	b	c	g	d
b	a	g	d	f
b	a	g	d	c
b	a	g	c	f
b	a	g	c	c
b	a	c	g	f
b	a	c	g	g
b	a	c	d	f
b	a	c	d	g

Shrinkage Pattern

- Generated by shrinking at least two vertices in pattern graph p belonging to different subpatterns
  - Specifically construct the patterns that contain duplicated elements
  - It is proved that this method can eliminate all invalid tuples*
- After eliminating embeddings matching shrinkage patterns (invalid), handling duplicate tuples (valid counted many times) is easier
  - Divide by multiplicity



Input graph



T_1

v_0	v_1	v_2	v_3
a	b	g	f
a	g	b	f
a	b	g	c
a	g	b	c
a	c	b	f
a	b	c	f
a	c	b	g
a	b	c	g
b	g	a	d
b	a	g	d
b	g	a	c
b	a	g	c
b	a	c	g
b	c	a	g
b	a	c	d
b	c	a	d

⋈

T_2

v_0	v_1	v_2	v_4
b	a	g	f
g	a	b	f
b	a	g	c
g	a	b	c
c	a	b	f
b	a	c	f
c	a	b	g
b	a	c	g
g	b	a	d
a	b	g	d
g	b	a	c
a	b	g	c
a	b	c	g
c	b	a	g
a	b	c	d
c	b	a	d

T_3

v_0	v_1	v_2	v_3	v_4
a	b	g	f	d
a	b	g	f	c
a	b	g	c	d
a	b	g	c	c
a	b	c	f	g
a	b	c	f	d
a	b	c	g	g
a	b	c	g	d
b	a	g	d	f
b	a	g	d	c
b	a	g	c	f
b	a	g	c	c
b	a	c	g	f
b	a	c	g	g
b	a	c	d	f
b	a	c	d	g

Filtering all entries that v3=v4

T_4

v_0	v_1	v_2	v_3	v_4
a	b	g	f	d
a	b	g	f	c
a	b	g	c	d
a	b	c	f	g
a	b	c	f	d
a	b	c	g	d
b	a	g	d	f
b	a	g	d	c
b	a	g	c	f
b	a	c	g	f
b	a	c	d	f
b	a	c	d	g

* Ali Pinar, et al. Escape: Efficiently counting all 5-vertex subgraphs. WWW'17

Is Decomposition Always Better?

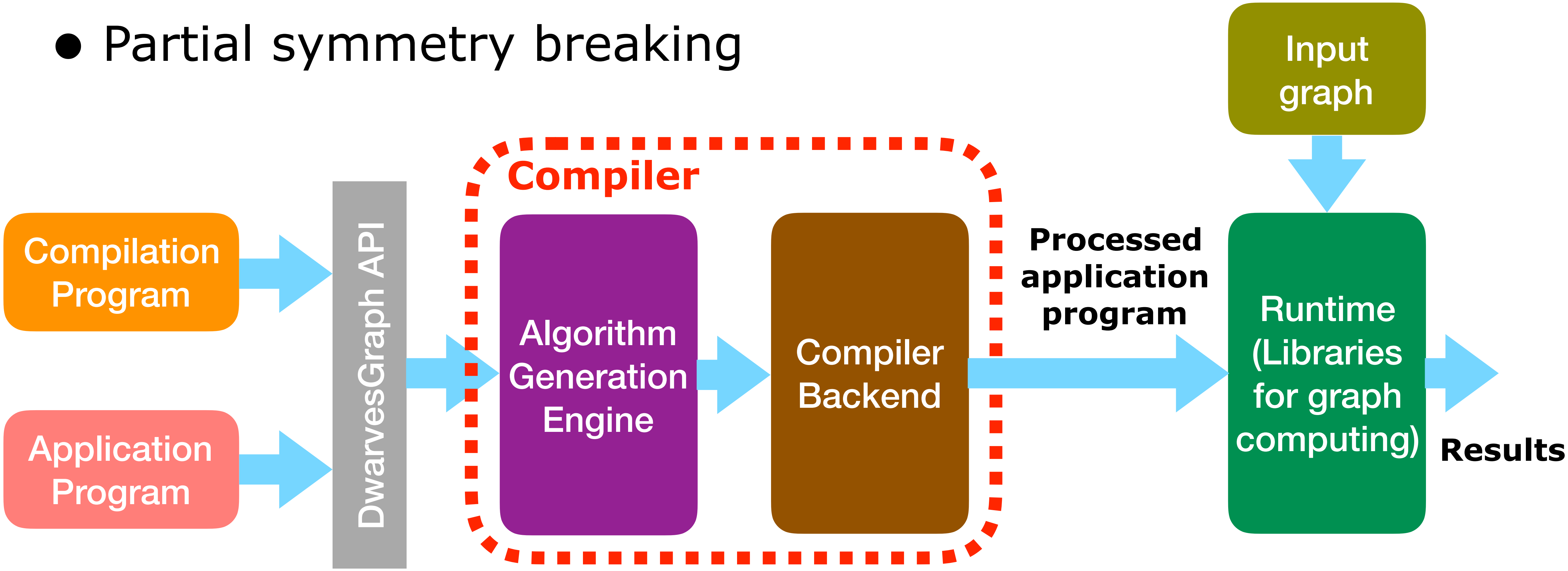
- The answer is **NO**. It does not guarantee the total runtime reduction
  - The combined number of enumerated subpatterns may be increased—we did not observe it
  - Some subpatterns after the decomposition may be very frequent
    - One of the subpatterns of a size-5 pattern is the very frequent 4-loop
- A performance model is necessary to estimate the cost of computation to avoid these cases

System Challenges

- While the decomposition-based algorithm is known, there are several challenges in building a general system
- **Challenge 1**: huge algorithm search space
  - A pattern specification typically has multiple patterns
  - 112 patterns for 6-motif; 823 patterns for 7-motif
  - With computation reuse, the mining of these patterns are fused together
  - Cutting set for each should be determined jointly
- **Challenge 2**: fast and accurate cost estimation
- **Challenge 3**: decomposition not compatible with symmetry breaking
- **Challenge 4**: beyond counting—advanced mining tasks such as frequent subgraphs mining (FSM)

DwarvesGraph: A Decomposition-based Graph Mining System

- We build a new graph mining system based on pattern decomposition
  - APIs to support various mining tasks
  - Approximate-mining based cost model
  - Efficient decomposition space search
  - Partial symmetry breaking

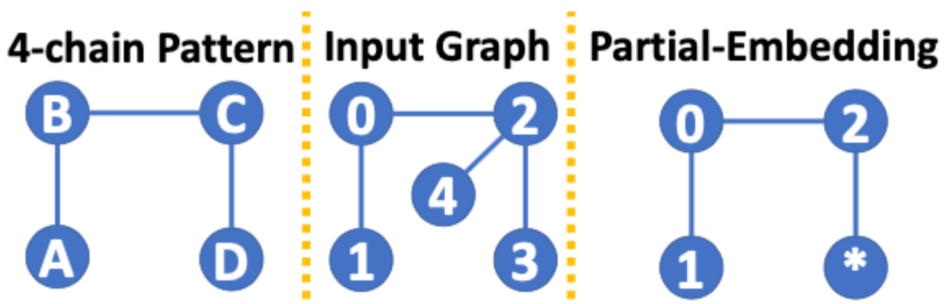


DwarvesGraph APIs

- Users provide two programs
  - *Application program*: specify the major user-defined logics
  - *Compilation program*: specify the patterns to mine and invoke the compiler to generate the code
- APIs—both for convenient use and advanced applications:
  - High-level: `int get_pattern_count();`
  - Low-level: partial-embedding-centric model

Partial-Embedding Centric Model

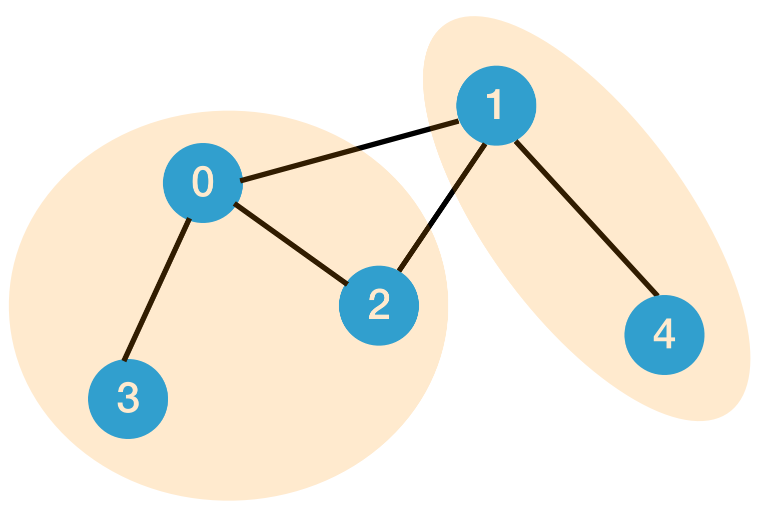
- A new model designed for decomposition-based graph mining
  - A partial-embedding matches a subpattern
- void process_partial_embedding(PartialEmbedding pe, int count);
  - Invoked by the system when the partial-embedding pe can be “extended” to reach at least one complete embedding of the whole pattern
  - The count indicates how many complete embeddings can *be extended from the partial-embedding*
- std::vector<Embedding> materialize(PartialEmbedding pe, int num);
  - Concretize the first num embeddings of the whole pattern from the partial embedding
- These seem to be arbitrary, what are the system guarantees?

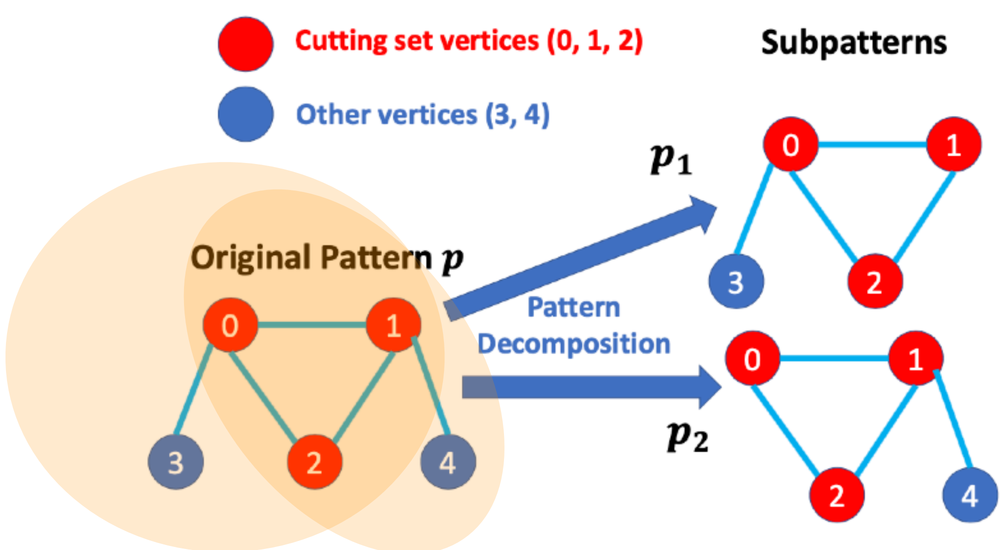


process_partial_embedding is called when (1,0,2,*) is identified the * can be either 3 or 4. Two ways to reach the whole embedding from the partial embedding: count=2 passed as the parameter.

Partial-Embedding Centric Model

- **Complete Guarantee**:
  - If a partial-embedding pe matching a subpattern P_{sub} is passed to process_partial_embedding…
  - Then all other partial embeddings matching P_{sub} will be also passed
- **Coverage Guarantee**:
  - The set of subpatterns matched by the passed partial-embeddings must fully cover all vertices of the pattern graph
  - It is more relaxed than decomposition





A Simple Example

```
// User-invoking functions
int get_pattern_count();
std::vector<Embedding> materialize(PartialEmbedding pe, int num);
// User-defined functions (UDFs)
void process_partial_embedding(PartialEmbedding pe, int count);
```

Partial-embedding centric APIs

```
int pattern_count = 0;
int num_embeddings_to_list = 100;
std::vector<Embedding> listed_embeddings;
void process_partial_embedding(PartialEmbedding pe, int count) {
  if (pe.subpattern_id == 0) {
    int remained = num_embeddings_to_list - pattern_count;
    if (remained > 0) listed_embeddings.add_all(materialize(pe, remained));
    pattern_count += count;
  }
}
```

Pattern counting with bounded embedding list

Application program

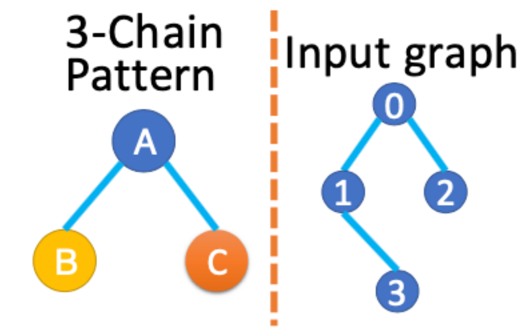
```
// ****** three_chain.cc ******
D_GENERATE_CLASS(ThreeChainCountingClass);
// ... load the input graph && other initializations
ThreeChainCountingClass * app = new ThreeChainCountingClass(graph);
cout<<"three-chain count: "<<app->get_pattern_count()<<endl;
// ****** three_chain_compiler.cc ******
Compiler * c = new Compiler();
Pattern p(3); // construct the 3-chain pattern
p.add_edge(0,1); p.add_edge(1,2);
c->compile(input="three_chain.cc", pattern=p,
   class_name="ThreeChainCountingClass");
```

Compilation program

User programs

- After processing the whole graph, `pattern_count` contains the total number of embeddings of the target pattern (*before removing multiplicity*) —ensured by the complete guarantee.
- The materialize function is flexible to let users to return a number of concrete embedding if needed.

Frequent Subgraph Mining (FSM)

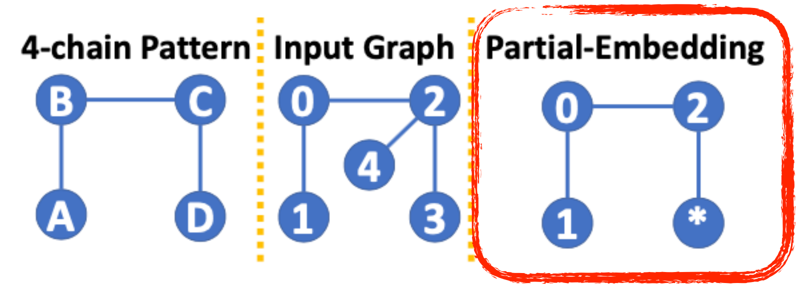


- **Domain** of a *pattern vertex*:
  - The set of input graph vertices that can map to it
  - Dom(A)={0,1};Dom(B)={0,1,2,3};Dom(C)={0,1,2,3}
- **Support**—a metric to quantize the *frequency* of a pattern
  - We use the minimum image-based (MINI) support definition*
  - MINI support=the size of the smallest domain across all pattern vertices
  - MINI support of the 3-chain on the input is |{0,1}|=2.
- FSM aims to discover frequent patterns
  - The application considers all patterns ≤ a certain size
  - Return a pattern if its support is no less than a user-specified threshold

* Bringmann et al. What is frequent in a single graph? PA-KDD'08

FSM in DwarvesGraph

```
D_GENERATE_CLASS(SupportCalculation);
bool domain[PATTERN_SIZE][INPUT_GRAPH_SIZE];
void process_partial_embedding(PartialEmbedding pe, int count) {
    for (i = 0; i < PATTERN_SIZE; ++ i) {
        int mapped_v = pe.get_vertex(i);
        if (mapped_v != UNDETERMINED) domain[i][mapped_v] = 1; }
}
SupportCalculation * app = new SupportCalculation(graph);
domain.clear(); app->execute(&process_partial_embedding);
support = graph.num_vertices;
for (i = 0; i < PATTERN_SIZE; ++ i)
    support = min(support, domain[i].num_non_zero_elements());
```



- returned tuple: `(1,0,2,*)`
- vertex D in 4-chain is UNDETERMINED
  — domain[A][1]=1
  — domain[B][0]=1
  — domain[C][2]=1
- `count` is not used

- The correctness is ensured by
  - Complete guarantee: all partial-embeddings of a subpattern are returned
  - Coverage guarantee: all vertices in the pattern graph are covered

Efficient Implementation

- The users are not aware of the decomposition-based algorithm implementation
- The DwarvesGraph compiler generates the implementations based on decomposition
- The partial-embedding centric model is general
  - The complete embedding can be considered as a special case for partial-embedding
  - If the implementation does not use decomposition, our API can still work—each embedding matches the complete pattern
- The compiler efficiently generates the procedure explained with embedding tables without expensive relational algebra
  - Unlike RStream, which implements relational algebra

Efficient Implementation

T_1

v_0	v_1	v_2	v_3
a	b	g	f
a	g	b	f
a	b	g	c
a	g	b	c
a	c	b	f
a	b	c	f
a	c	b	g
a	b	c	g
b	g	a	d
b	a	g	d
b	g	a	c
b	a	g	c
b	a	c	g
b	c	a	g
b	a	c	d
b	c	a	d

⋈

T_2

v_0	v_1	v_2	v_4
b	a	g	f
g	a	b	f
b	a	g	c
g	a	b	c
c	a	b	f
b	a	c	f
c	a	b	g
b	a	c	g
g	b	a	d
a	b	g	d
g	b	a	c
a	b	g	c
a	b	c	g
c	b	a	g
a	b	c	d
c	b	a	d

T_3

v_0	v_1	v_2	v_3	v_4
a	b	g	f	d
a	b	g	f	c
a	b	g	c	d
a	b	g	c	c
a	b	c	f	g
a	b	c	f	d
a	b	c	g	g
a	b	c	g	d
b	a	g	d	f
b	a	g	d	c
b	a	g	c	f
b	a	g	c	c
b	a	c	g	f
b	a	c	g	g
b	a	c	d	f
b	a	c	d	g

Filtering all entries that v3=v4

T_4

v_0	v_1	v_2	v_3	v_4
a	b	g	f	d
a	b	g	f	c
a	b	g	c	d
a	b	c	f	g
a	b	c	f	d
a	b	c	g	d
b	a	g	d	f
b	a	g	d	c
b	a	g	c	f
b	a	c	g	f
b	a	c	d	f
b	a	c	d	g

Input: Pattern p, Cutting Set V_C

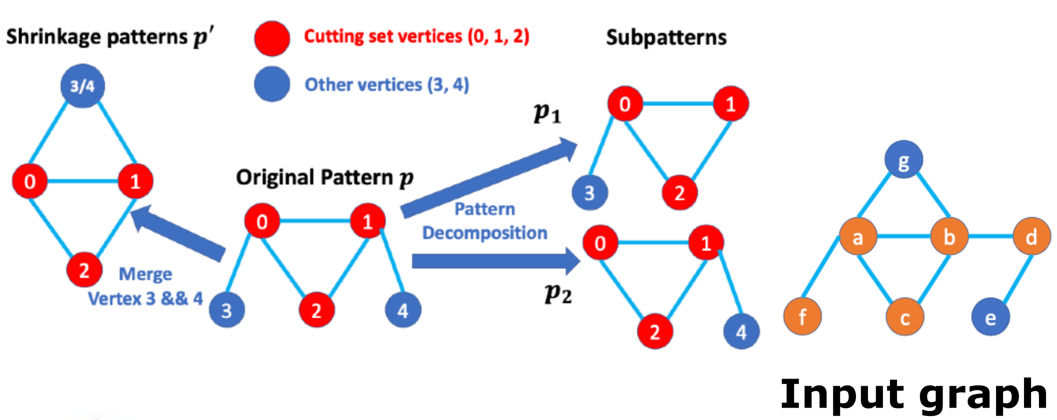
```
Decompose p using V_C to generate K subpatterns, and the shrinkage patterns
num_shrinkages_i = HashTable() (1 ≤ i ≤ K)
for all e_c = (v_0, v_1, ..., v_{|V_C|-1}) matching the cutting set V_C do
    M ← 1
    for all i ← 1...K do
        M_i ← the number of pe extending e_c and matching the i-th subpattern
        num_shrinkages_i.clear()
        M ← M × M_i
    end for
    for all e extending e_c and matching one of the shrinkage patterns do
        for all i ← 1...K do
            pe_i ← extract_subpattern_embedding(e, i)
            num_shrinkages_i[pe_i] ← num_shrinkages_i[pe_i] + 1
        end for
    end for
    for all i ← 1...K do
        for all pe extending e_c and matching the i-th subpattern do
            count ← M/M_i − num_shrinkages_i[pe]
            if count > 0 then
                process_partial_embedding(pe, count)
            end if
        end for
    end for
end for
```

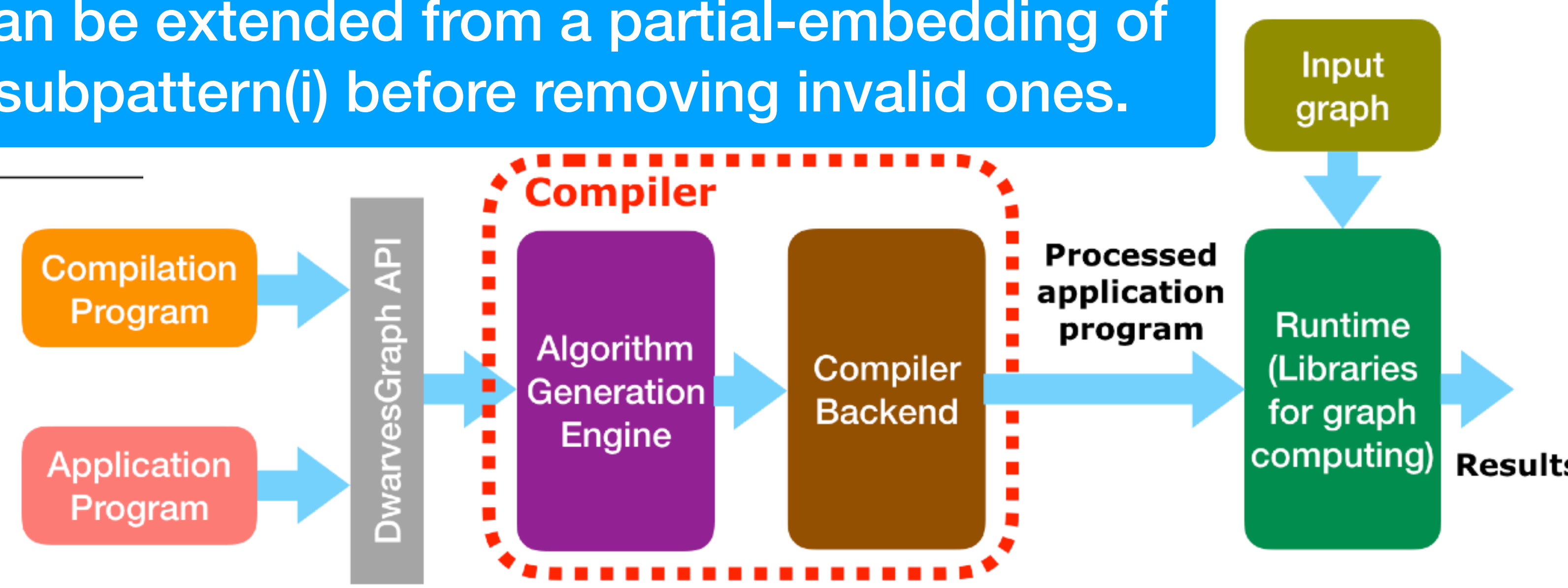
Vc is determined by the algorithm generation engine

Compute the count of each Tk with vertex-set-based method similar to AutoMine

For each embedding matching the shrinkage pattern, we check which partial-embedding does it contain. It should be removed from the count of *each* partial-embedding.

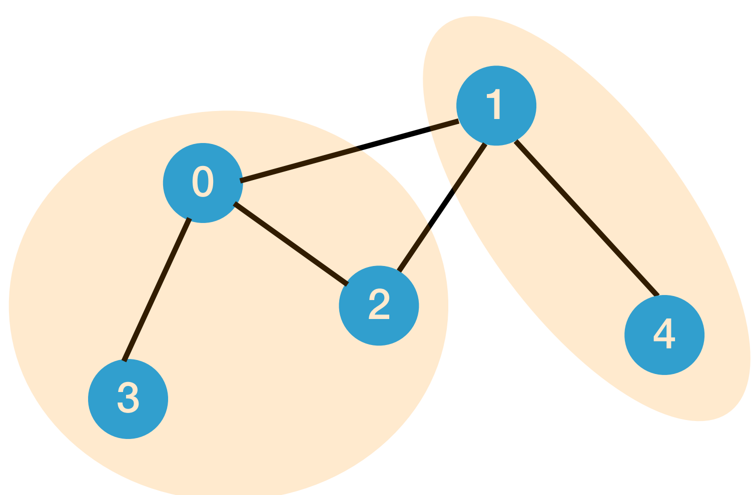
M/Mi: the number of complete embeddings that can be extended from a partial-embedding of subpattern(i) before removing invalid ones.

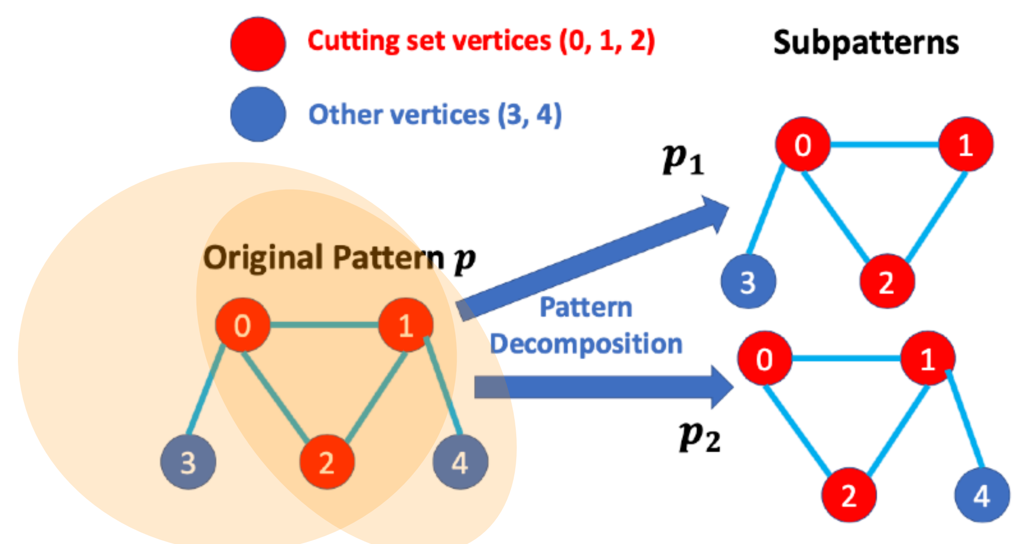




Summary

- The partial-embedding centric model is not tied to the decomposition
- The complete and coverage guarantee can ensure the correctness
- The system implementation based on decomposition ensures the stronger property:
  - All subpatterns share the cutting set V_C
  - The coverage guarantee just requires all vertices are covered—disjoint subpatterns that can cover all vertices also satisfy it—but not decomposition method
- The algorithm we described is a *template* for the compiler to generate codes for the given pattern graph
  - The cutting set determined by algorithm generate engine





We Still Need to Solve…

- The cost model for algorithm generation engine to evaluate different choices of cutting set
- Efficiently search the cutting sets across multiple patterns
- How to make symmetric breaking work for decomposition as much as possible?

Cost Model

- We need to quickly evaluate the performance of generated subpattern enumeration algorithms
  - Executing of algorithms on real datasets/machines is too expensive
- Pattern enumeration is a set of nested for-loops
  - The key problem: estimate the cost of each loop
- AutoMine* is the first system that uses a cost model to select pattern matching schedules for better performance
  - Problem: its cost model is over-simplified
    - Assumes that the algorithm runs on a random graph with *n* vertices, each vertex pair is connected by a fixed probability *p*
    - For counting k-clique, #iteration of 1st,2nd,3rd,…,k-th loop are $n, np, np^2, \ldots, np^{k-1}$. With k=5, line 6 is should be executed n^5p^{10} times
    - Patents graph: n=3.8M, avg_deg=8.76, p=2.3×10^{-6}, line 5 is estimated to execute 3.28×10^{-24} times
    - In reality, Patents graph has 3M 5-cliques, line 5 executed for $3M \times 5!$ times

```
1:  for v_0 ∈ V(G) do
2:    for v_1 ∈ N(v_0) do
3:      for v_2 ∈ N(v_0) ∩ N(v_1) do
4:        ...
5:          for v_{k-1} ∈ N(v_0) ∩ N(v_1) ∩ ... N(v_{k-2}) do
6:            cnt ← cnt + 1
7:          end for
8:        ...
9:      end for
10:   end for
11: end for
```

Counting k-clique

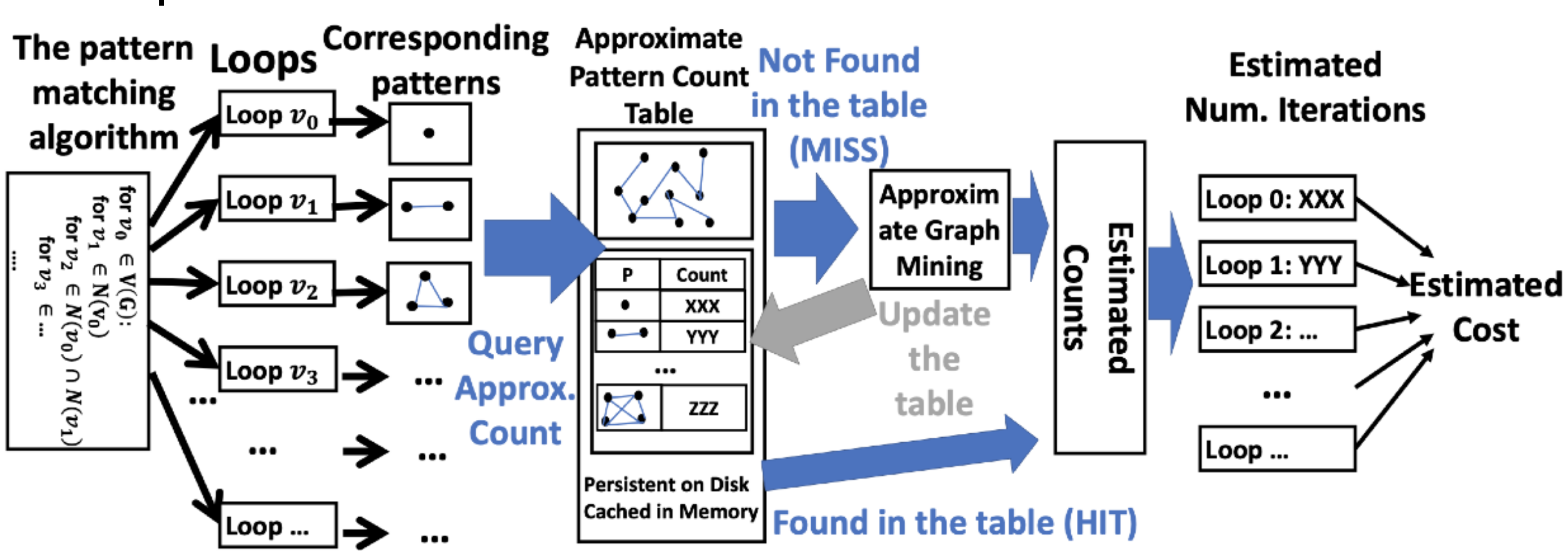
Mawhirter et al. AutoMine: Harmonizing High-level Abstraction and High Performance for Graph Mining. SOSP'19

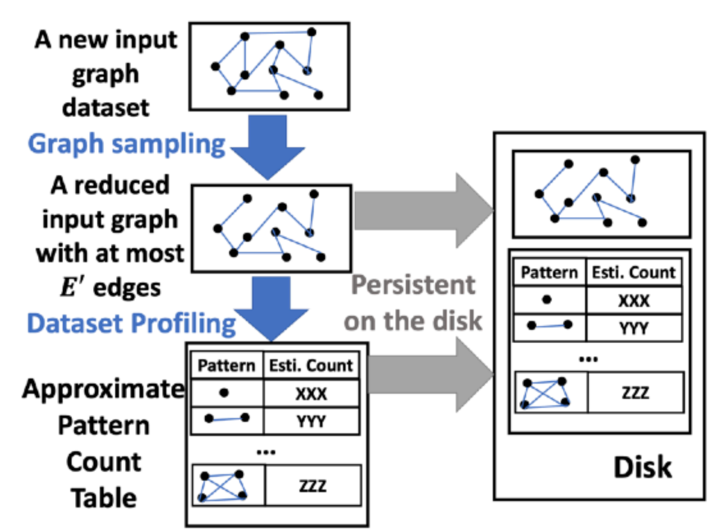
A New Cost Model

- Key insight: every iteration corresponds to a match of a pattern
- The problem is converted to the pattern count estimation of the input graph
  - Can be *approximate*
  - Only need to be *relative*
- A new cost model based on approximate graph mining
  - Generate a reduced graph by sampling input graph
    - At most 32M edges
  - Run neighborhood sampling in ASAP* to get the approximation of the patterns up to certain size, store the results in table persisted in disk
  - During algorithm search, query the table to get the cost of loop based on the count of the corresponding pattern
- Obtain the count of frequent patterns accurately, while underestimating that of the infrequent ones

```
Cnt ← 0
for i ← 1...NumSamples do
    v0 ← UniformSample(V(G))
    v1 ← UniformSample(N(v0))
    v2 ← UniformSample(N(v1))
    if v2 ∈ N(v0) then
        Cnt ← Cnt + |V(G)|·|N(v0)|·|N(v1)|
    end if
end for
Cnt ← Cnt/NumSamples/6
```

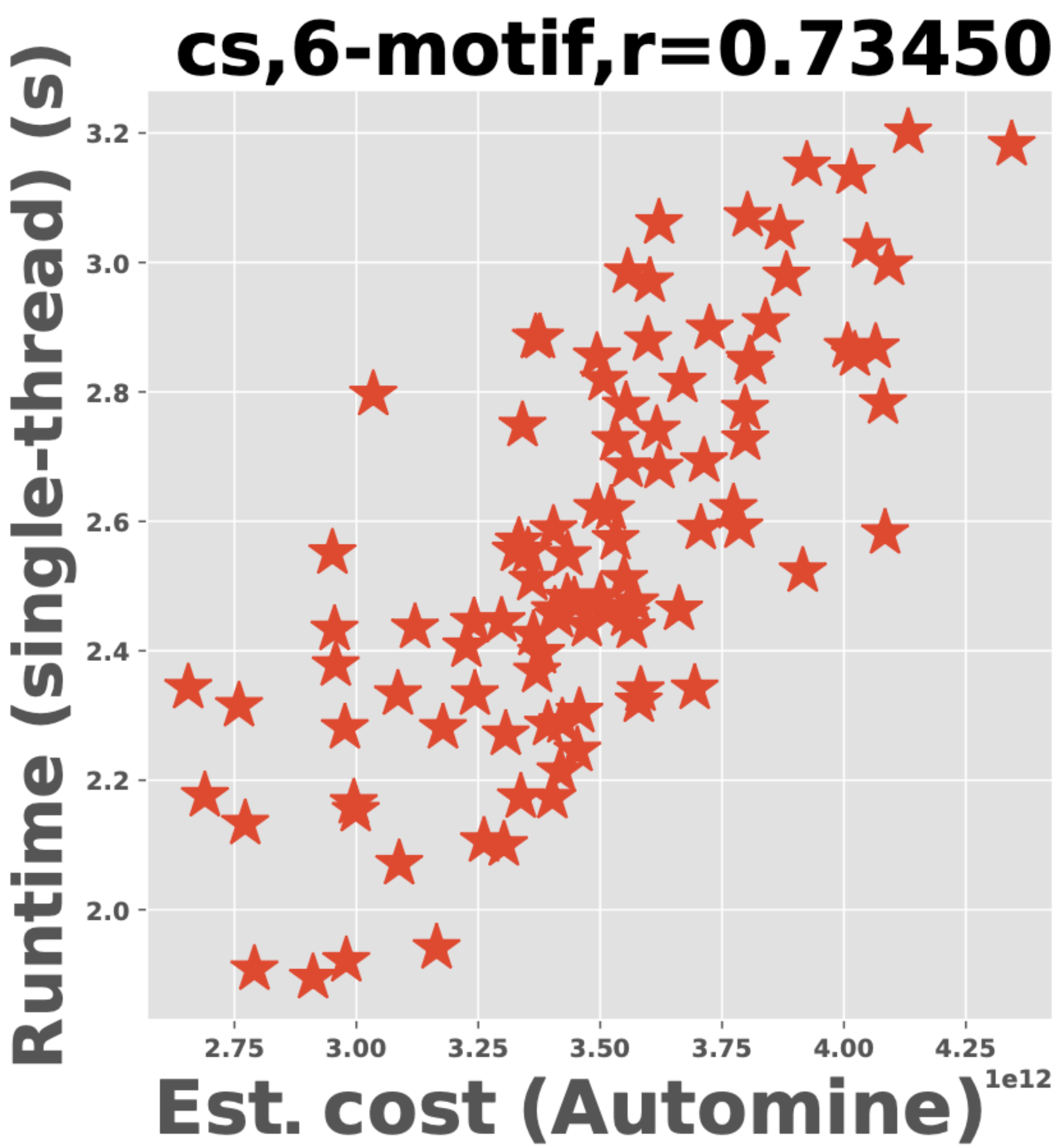
Graph	Profiling Time (s)
CiteSeer	1.96
MiCo	3.50
Patents	6.64
LiveJournal	7.14
Friendster	7.10

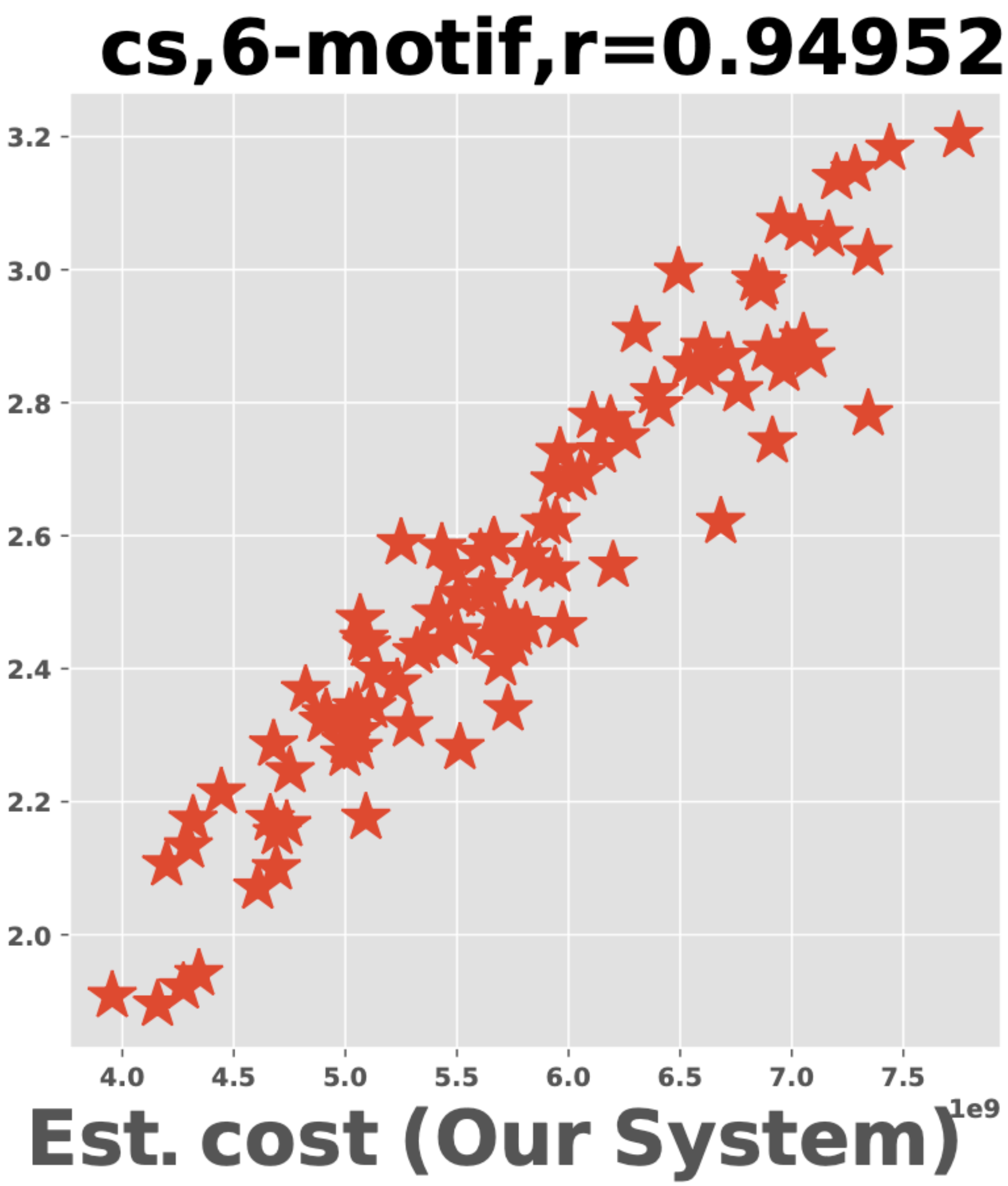




* Iyer et al. ASAP: Fast, approximate graph pattern mining at scale. OSDI'18

The New Cost Model Effectiveness

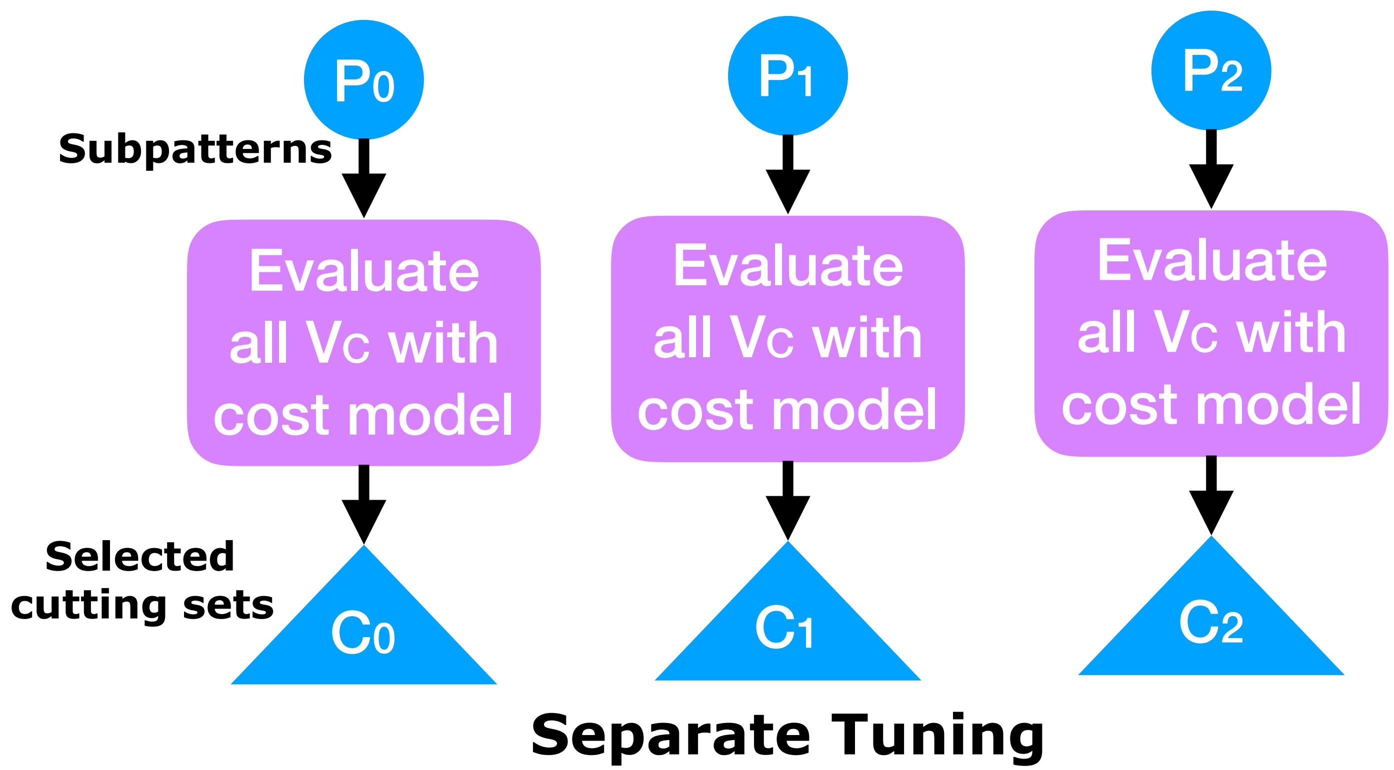




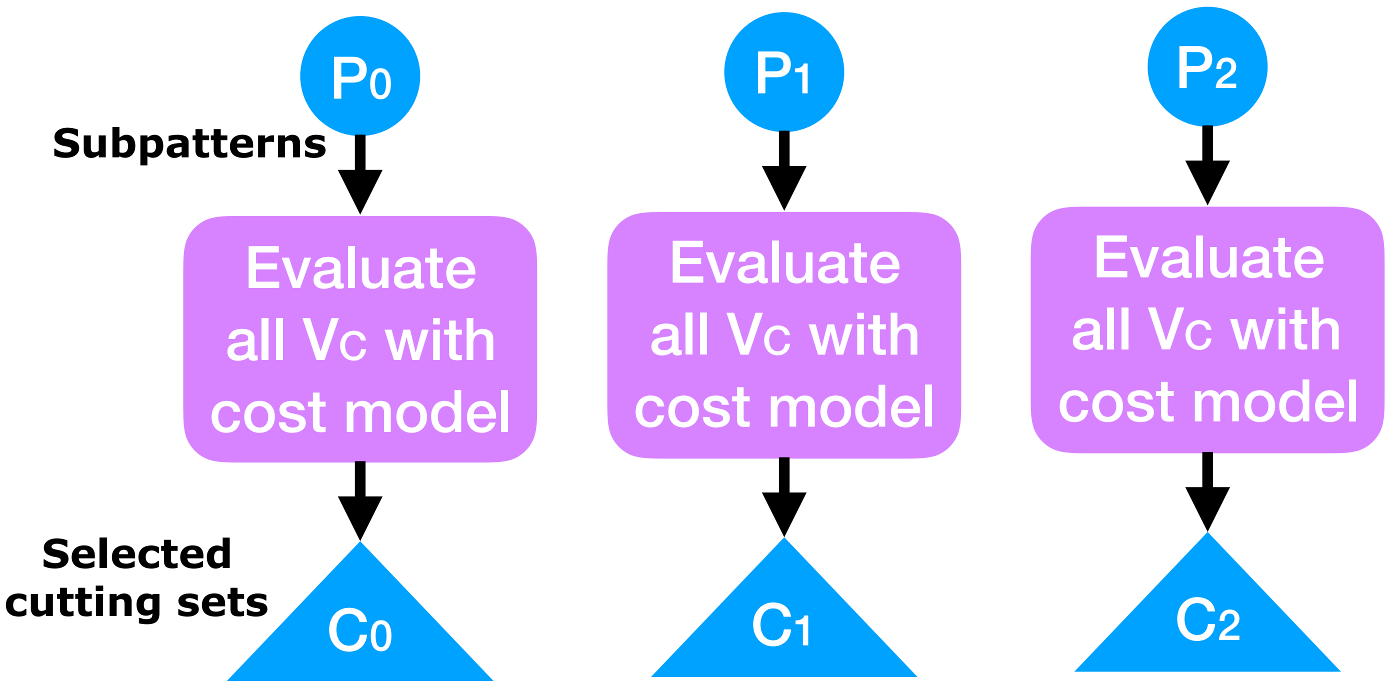
AutoMine **Our Method**

Decomposition Space Search

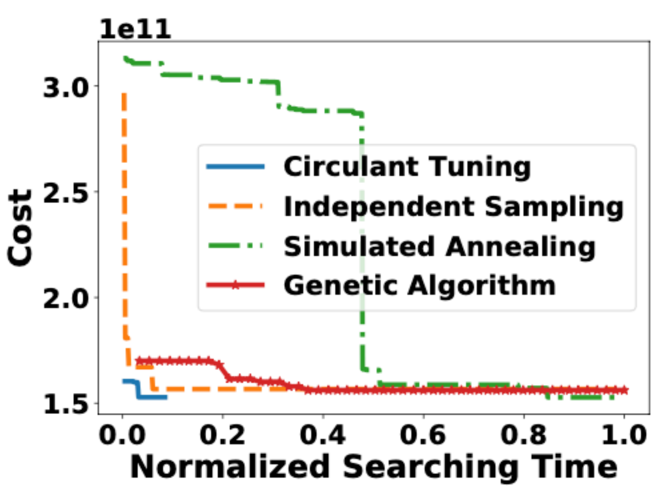
- The graph mining applications need to handle multiple patterns
  - Motif Counting (MC) aims at counting all connected patterns with a particular size
- Need to select a cutting set for each pattern
- With computation reuse, enumeration of multiple patterns can be fused, the search becomes **joint**
- We propose the circulant tuning method with fast convergence

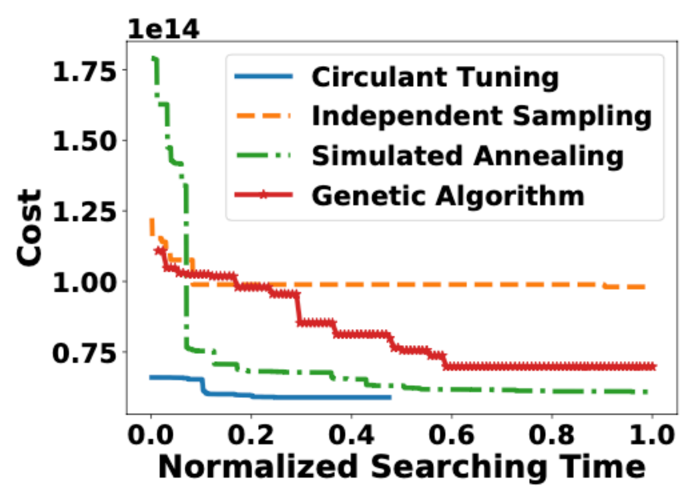


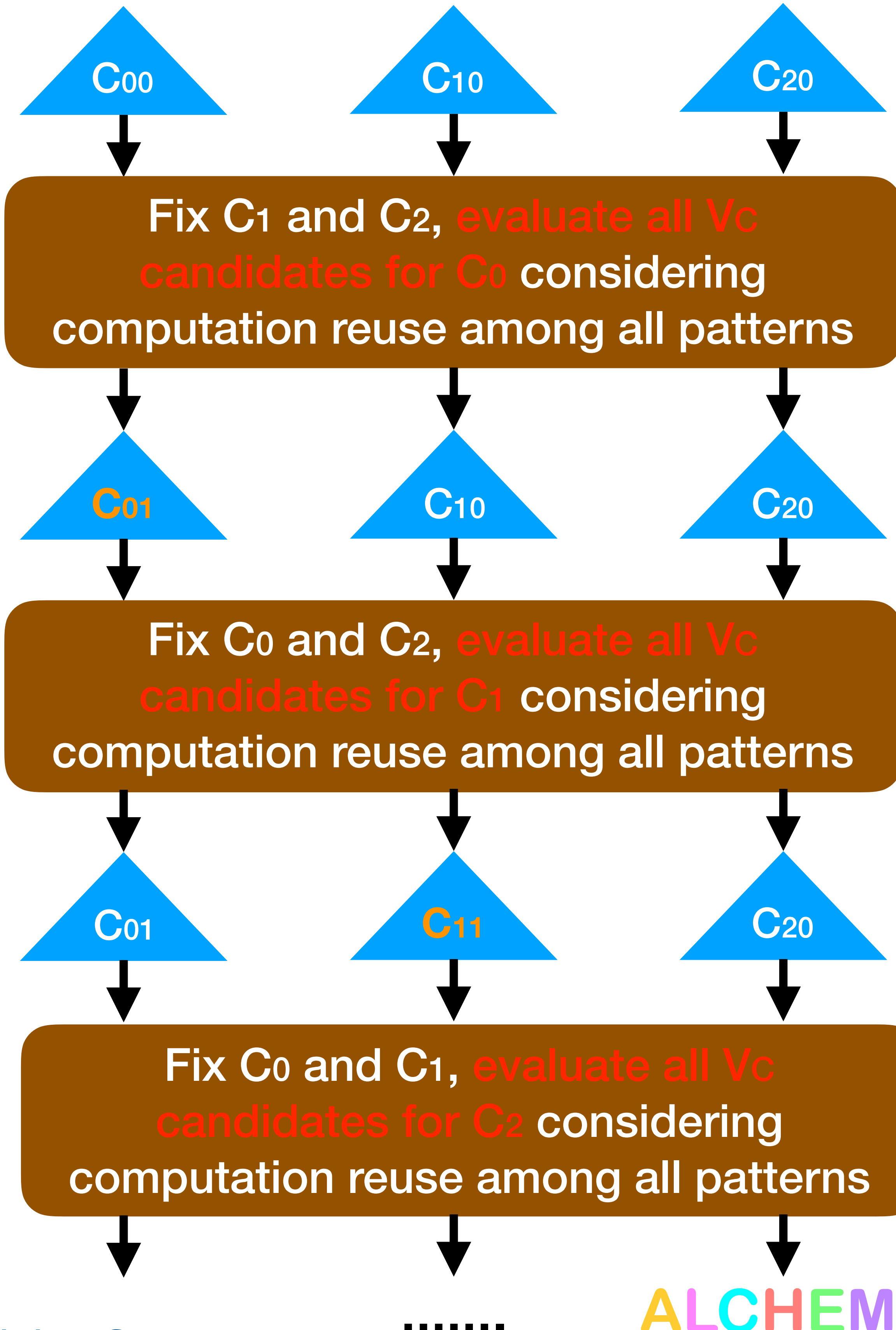
Circulant Tuning



Separate Tuning







Circulant Tuning

Partial Symmetry Breaking

- Problem: with symmetry breaking for subpatterns, the complete embeddings cannot be correctly joined

T_1

v_0	v_1	v_2	v_3
a	b	g	f
a	g	b	f
a	b	g	c
a	g	b	c
a	c	b	f
a	b	c	f
a	c	b	g
a	b	c	g
b	g	a	d
b	a	g	d
b	g	a	c
b	a	g	c
b	a	c	g
b	c	a	g
b	a	c	d
b	c	a	d

⋈

T_2

v_0	v_1	v_2	v_4
b	a	g	f
g	a	b	f
b	a	g	c
g	a	b	c
c	a	b	f
b	a	c	f
c	a	b	g
b	a	c	g
g	b	a	d
a	b	g	d
g	b	a	c
a	b	g	c
a	b	c	g
c	b	a	g
a	b	c	d
c	b	a	d

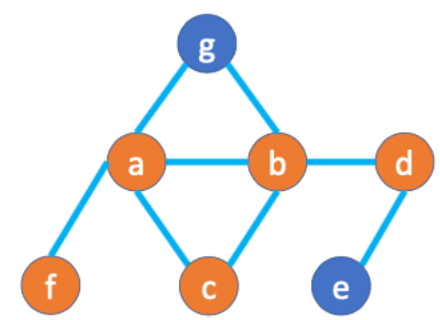
T_3

v_0	v_1	v_2	v_3	v_4
a	b	g	f	d
a	b	g	f	c
a	b	g	c	d
a	b	g	c	c
a	b	c	f	g
a	b	c	f	d
a	b	c	g	g
a	b	c	g	d
b	a	g	d	f
b	a	g	d	c
b	a	g	c	f
b	a	g	c	c
b	a	c	g	f
b	a	c	g	g
b	a	c	d	f
b	a	c	d	g

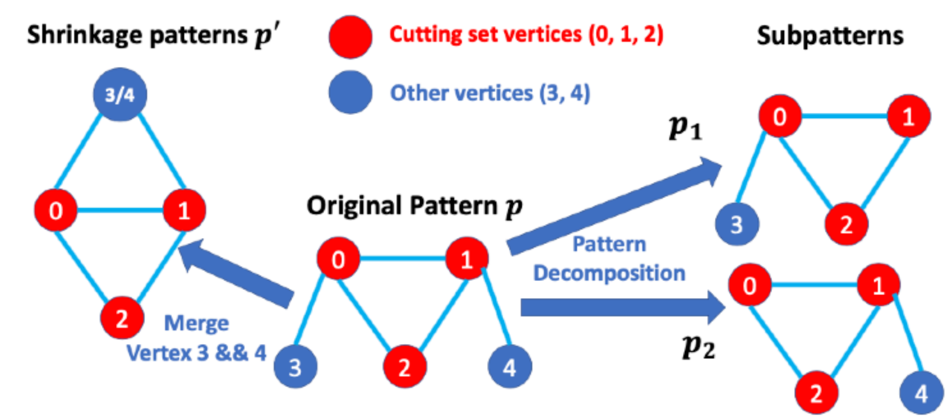
Filtering all entries that v3=v4

T_4

v_0	v_1	v_2	v_3	v_4
a	b	g	f	d
a	b	g	f	c
a	b	g	c	d
a	b	c	f	g
a	b	c	f	d
a	b	c	g	d
b	a	g	d	f
b	a	g	d	c
b	a	g	c	f
b	a	c	g	f
b	a	c	d	f
b	a	c	d	g



Input graph




```
for v0 ∈ V(G) do
    for v1 ∈ N(v0) do
        for v2 ∈ N(v0) ∩ N(v1) do
            extend_triangle(v0, v1, v2)
        end for
    end for
end for
```

Partial Symmetry Breaking

```
for v0 ∈ V(G) do
    for v1 ∈ N(v0) ∩ {v1 | v1 > v0} do
        for v2 ∈ N(v0) ∩ N(v1) ∩ {v2 | v2 > v1} do
            extend_triangle(v0, v1, v2)
            extend_triangle(v0, v2, v1)
            extend_triangle(v1, v0, v2)
            extend_triangle(v1, v2, v0)
            extend_triangle(v2, v0, v1)
            extend_triangle(v2, v1, v0)
        end for
    end for
end for
```

Only applied when the gain is larger than a threshold, based on cost model.

Symmetry breaking:
"$v_2<v_1$" for p_1
"$v_0<v_2$" for p_2

T_1

v_0	v_1	v_2	v_3
a	g	b	f
a	g	b	c
a	c	b	f
a	c	b	g
b	g	a	d
b	g	a	c
b	c	a	g
b	c	a	d

⋈

T_2

v_0	v_1	v_2	v_4
b	a	g	f
b	a	g	c
b	a	c	f
b	a	c	g
a	b	g	d
a	b	g	c
a	b	c	g
a	b	c	d

T_3

v_0	v_1	v_2	v_3	v_4

Solution: *perform symmetry breaking for "subpattern of subpattern" when possible and compensate for asymmetric parts*

DwarvesGraph Evaluation

- **System:**
  - Each node has two 8-core Intel Xeon E5-2630 CPUs (hyperthreading disabled) and 64GB DRAM
  - GPU (Pangoline): NVIDIA V100 GPU with 32GB memory
  - Arabeque and Fractal (distributed) use 8 nodes. Arabeque uses Hadoop 2.7.7, Fractal uses Apache Spark 2.2.0
- **Applications:**
  - Motif Counting (MC): count all connected vertex-induced patterns with a particular size
  - Pseudo Clique Mining (PC): A vertex-induced pattern is a pseudo clique if the number of its edges is no less than $n(n-1)/2-k$, n is the #vertex and k is a parameter
  - Frequent Subgraph Mining (FSM)
- **Other systems:**
  - In-house AutoMine implementation
  - RStream (OSDI'18)
  - Arabesque (SOSP'15)
  - Peregrine (EuroSys'20)
  - Pangoline (CPU/GPU) (VLDB'20)
  - Fractal (SIGMOD'19)
  - GraphPi (SC'20)

Graph	Abbr.	|V|	|E|	|L|
CiteSeer [5, 18, 43]	cs	3.3K	4.5K	6
EmailEuCore [28, 54]	ee	1.0K	16.1K	42
WikiVote [26]	wk	7.1K	100.8K	N/A
MiCo [16]	mc	96.6K	1.1M	29
Patents [27]	pt	3.8M	16.5M	N/A
Labeled-Patents [27]	lpt	2.7M	14.0M	37
LiveJournal [4, 30]	lj	4.8M	42.9M	N/A
Friendster [53]	fr	65.6M	1.8B	N/A
RMAT-100M [10]	rmat	100M	1.6B	N/A

Graph datasets

App	Graph	Our Impl.	Original Impl.
3-MC	wk	27.3ms	34.5ms
	mc	161ms	230ms
	pt	0.9s	1.9s
	lj	9.0s	13.4s
4-MC	wk	7.0s	11.5s
	mc	31.7s	45.2s
	pt	24.3s	82.1s
	lj	457m	367m
5-MC	wk	4345s	5300s
	mc	2.91h	5.56h
	pt	54m	117m

Comparing with AutoMine, RStream, and Arabesque

App.	G	DwarvesGraph	AutomineInHouse	RStream	Arabesque
3-MC	cs	0.16ms	0.16ms (1.0x)	142ms (888x)	10.1s (**63,125x**)
	ee	0.8ms	7.3ms (8.9x)	21.0s (25,471x)	10.2s (12,352x)
	wk	7.6ms	27.3ms (3.6x)	17.9m (141,437x)	12.1s (1,586x)
	pt	335.7ms	931ms (2.8x)	104.1m (18,611x)	96.4s (287x)
	mc	48.0ms	161ms (3.4x)	144.8m (181,051x)	21.1s (440x)
	lj	2.8s	9.0s (3.3x)	T	24.3m (529x)
4-MC	cs	0.2ms	4.8ms (23x)	3.7s (17,647x)	9.9s (46,794x)
	ee	9.4ms	920ms (98x)	132.4m (**842,186x**)	19.1s (2,023x)
	wk	60.0ms	7.0s (117x)	T	402.2s (6,704x)
	pt	1.5s	24.3s (16x)	T	68.3m (2,711x)
	mc	1.3s	31.7s (24x)	T	42.8m (1,942x)
	lj	32.8s	456.5m (**836x**)	T	C
5-MC	cs	1.4ms	332ms (229x)	146.4s (101,124x)	11.4s (7,843x)
	ee	360.3ms	104.8s (291x)	T	19.4m (3,233x)
	wk	5.3s	72.4m (823x)	T	C
	pt	32.6s	53.9m (99x)	T	C
	mc	114.7s	174.6m (91x)	T	C
	lj	167.7m	T	T	C
6-MC	cs	247.0ms	35.9s (145x)	108.7m (26,403x)	48.7s (197x)
	ee	91.3s	259.0m (170x)	T	C
	wk	38.7m	T	T	C
	pt	57.9m	T	T	C
7-PC	cs	0.3ms	0.5ms (1.7x)		
	ee	719ms	67.1s (93x)		
	wk	735ms	90.8s (24x)		
	pt	499ms	15.7s (31x)		
8-PC	cs	0.3ms	0.5ms (1.7x)		
	ee	1.3s	433.1s (322x)		
	wk	1.2s	463.0s (387x)		
	pt	582ms	86.2s (148x)		
FSM-300	cs	0.2ms	0.3ms (1.5x)	522ms (2,609x)	10.3s (51,315x)
	ee	0.2ms	0.2ms (1.0x)	3.6s (18,090x)	9.6s (48,235x)
	lpt	20.8s	20.3s (0.98x)	4,713.5s (226x)	C
	mc	308ms	441ms (1.4x)	149.1m (29,013x)	C
FSM-3K	cs	0.6ms	0.6ms (1.0x)	77.9ms (130x)	9.6s (15,931x)
	ee	0.2ms	0.2ms (1.0x)	210ms (1,049x)	9.8s (48,985x)
	lpt	18.6s	18.1s (0.98x)	89.0m (287x)	C
	mc	124ms	300ms (2.4x)	141.9m (68,813x)	157.9s (1,276x)

Graph	Abbr.	|V|	|E|	|L|
CiteSeer [5, 18, 43]	cs	3.3K	4.5K	6
EmailEuCore [28, 54]	ee	1.0K	16.1K	42
WikiVote [26]	wk	7.1K	100.8K	N/A
MiCo [16]	mc	96.6K	1.1M	29
Patents [27]	pt	3.8M	16.5M	N/A
Labeled-Patents [27]	lpt	2.7M	14.0M	37
LiveJournal [4, 30]	lj	4.8M	42.9M	N/A
Friendster [53]	fr	65.6M	1.8B	N/A
RMAT-100M [10]	rmat	100M	1.6B	N/A

Graph datasets

Comparing with Peregrine, Pangolin, and Fractal

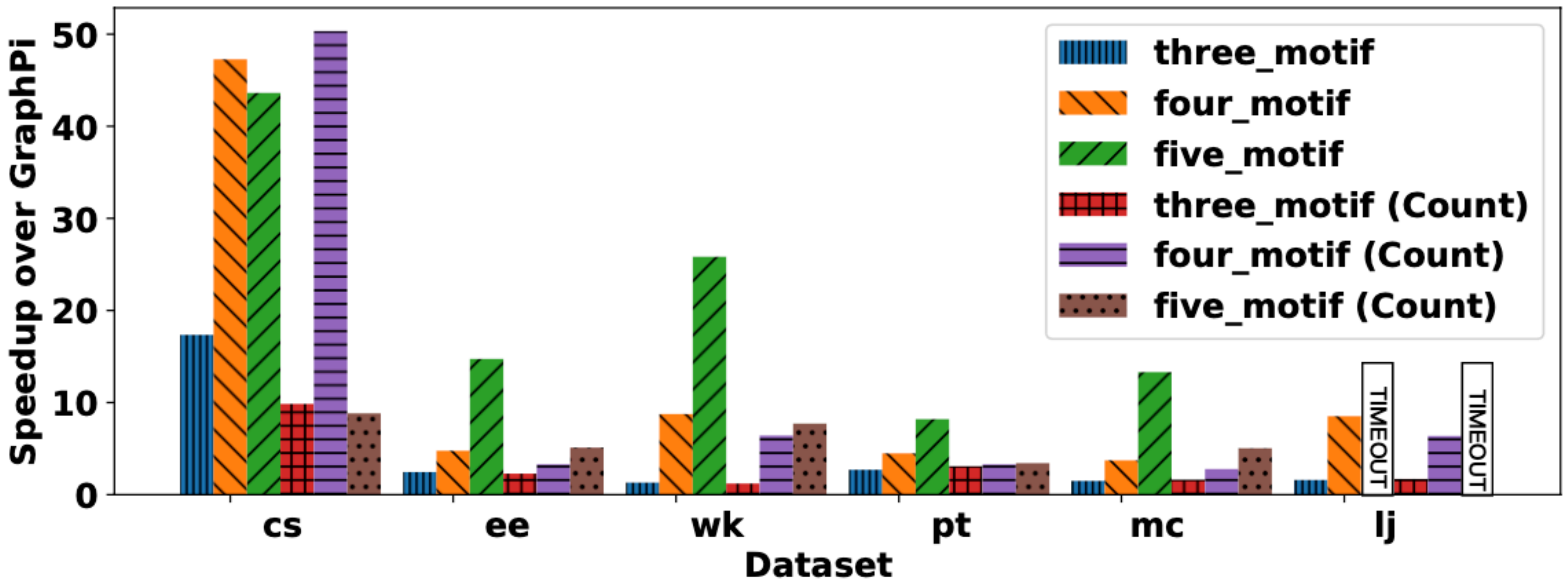
App.	G	DwarvesGraph	Peregrine	Pangolin(CPU/GPU)	Fractal
3-MC	cs	0.16ms	5.8ms	5.0ms / 0.1ms	5.9s
	pt	0.3s	1.4s	1.4s / 0.2s	79.7s
	mc	48ms	60ms	280ms / 14.1ms	12.9s
4-MC	cs	0.2ms	21.2ms	15.3ms / 0.7ms	6.0s
	pt	1.5s	11.2s	329.5s / 8.0s	141.6s
	mc	1.3s	5.3s	242.7s / 3.7s	58.4s
5-MC	cs	1.4ms	41.7ms	688.3ms / 1.3ms	6.1s
	pt	32.6s	513.6s	C / C	4517.0s
	mc	114.7s	5,635.1s	C / C	1240.0s
6-MC	cs	0.2s	0.8s	14.9s / C	4.6s
	pt	3,472.6s	T	C / C	T
FSM-100	mc	14.0s	C	C / C	346.6s
FSM-300		9.6s	C	C / C	280.2s
FSM-1K		2.5s	1,782.2s	C / C	169.1s
FSM-3K		0.5s	189.3s	C / C	109.4s
FSM-1K	lpt	1,511.5s	T	C / C	T
FSM-10K		71.4s	34,403.6s	C / C	T
FSM-20K		9.0s	4,781.0s	333.3s / C	270.1s
FSM-25K		2.7s	1,353.3s	126.5s / C	250.7s

Graph	Abbr.	|V|	|E|	|L|
CiteSeer [5, 18, 43]	cs	3.3K	4.5K	6
EmailEuCore [28, 54]	ee	1.0K	16.1K	42
WikiVote [26]	wk	7.1K	100.8K	N/A
MiCo [16]	mc	96.6K	1.1M	29
Patents [27]	pt	3.8M	16.5M	N/A
Labeled-Patents [27]	lpt	2.7M	14.0M	37
LiveJournal [4, 30]	lj	4.8M	42.9M	N/A
Friendster [53]	fr	65.6M	1.8B	N/A
RMAT-100M [10]	rmat	100M	1.6B	N/A

Graph datasets

Pangolin's GPU performance is competitive, but achieved with a significantly more expensive device (NVIDIA V100-32GB).

Comparing with GraphPi



GraphPi only handles the individual pattern and does not support FSM

Graph	Abbr.	|V|	|E|	|L|
CiteSeer [5, 18, 43]	cs	3.3K	4.5K	6
EmailEuCore [28, 54]	ee	1.0K	16.1K	42
WikiVote [26]	wk	7.1K	100.8K	N/A
MiCo [16]	mc	96.6K	1.1M	29
Patents [27]	pt	3.8M	16.5M	N/A
Labeled-Patents [27]	lpt	2.7M	14.0M	37
LiveJournal [4, 30]	lj	4.8M	42.9M	N/A
Friendster [53]	fr	65.6M	1.8B	N/A
RMAT-100M [10]	rmat	100M	1.6B	N/A

Graph datasets

Decomposition Space Search Methods

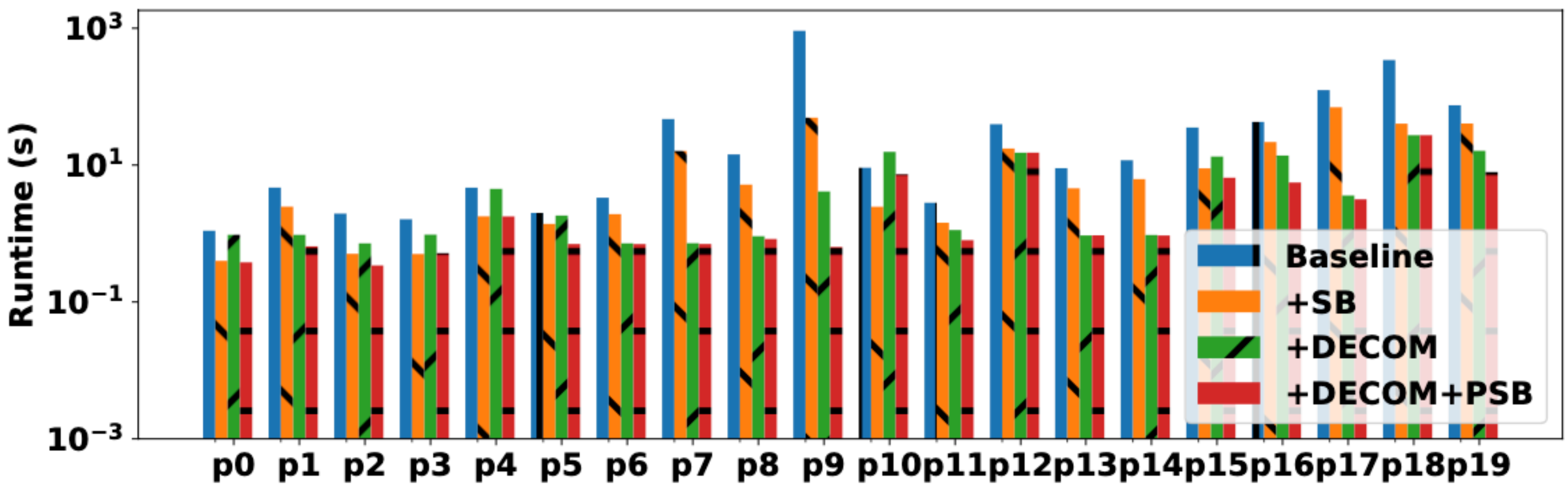
Circulant tuning is slower than separate tuning

R: random; S: separate tuning; C: circulant tuning
RT: runtime; ST: search time

App.	Graph	R-RT	S-RT	S-ST	C-RT	C-ST
5-MC	cs	5.3ms	2.2ms	5.0ms	1.4ms	0.6s
	ee	917ms	392ms	5.0ms	360ms	0.4s
	wk	14.4s	8.2s	5.0ms	5.3s	0.8s
	pt	69.3s	36.9s	1.7ms	32.6s	0.7s
6-MC	cs	576ms	280ms	37.2ms	247ms	325s
	ee	437.7s	99.8s	35.8ms	91.3s	252s
	wk	–	2,515.9s	40.1ms	2,320.0	409s
	pt	–	3,688.5s	43.7ms	3,472.6	222s

Circulant tuning achieves up to 1.57x speedup. For large graph, the benefit is more and search time can be amortized.

Partial Symmetry Breaking and Decomposition

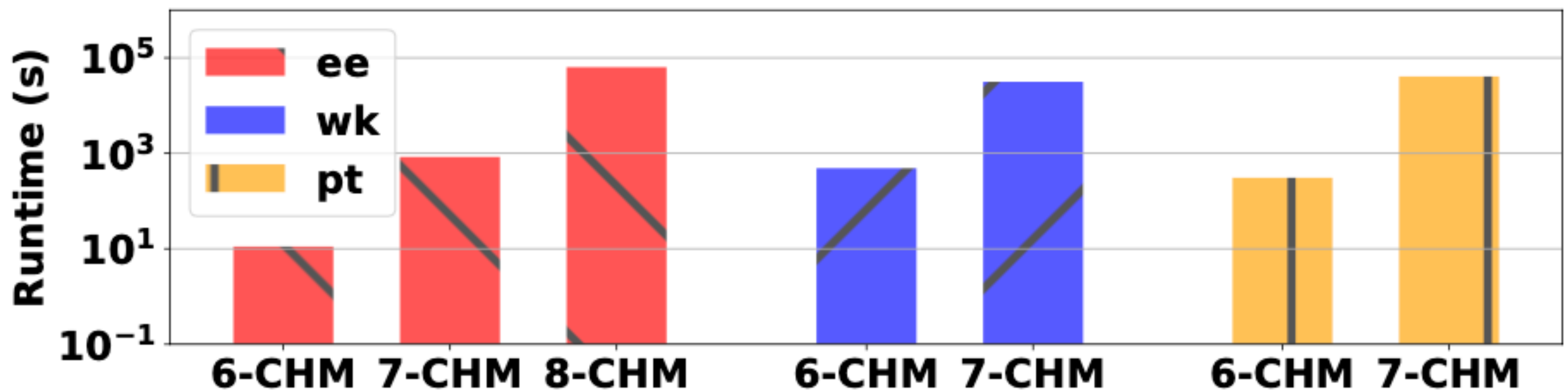


p0 — p19 are all size-5 patterns except for 5-clique

Large Graphs and Large Patterns

Graph	#Vertices	#Edges	App.	Runtime (s)
fr	65.6M	1.8B	4-Motif	4,301
			4-Chain	862
rmat	100M	1.6B	4-Motif	5,900
			4-Chain	800

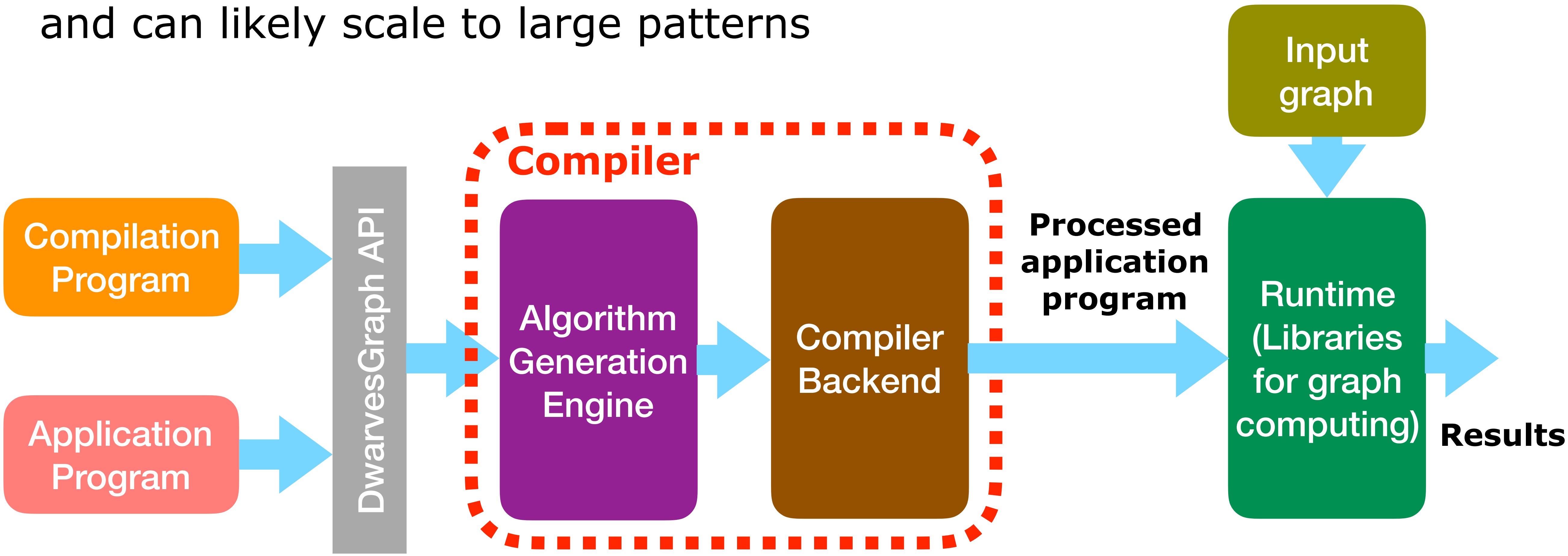
None of the previous exact graph mining systems have reported 4-motif results on graphs at this scale



k-CHM: k-chain mining
Keep increasing the size of the pattern until the task cannot finish within 24 hours

DwarvesGraph: A Decomposition-based Graph Mining System

- We build a new graph mining system based on pattern decomposition
  - APIs to support various mining tasks
  - Approximate-mining based cost model
  - Efficient decomposition space search
  - Partial symmetry breaking
- The results show that our system is faster than all existing systems and can likely scale to large patterns



Why Distributed Graph Mining System?

- Single-machine shared memory systems (Peregrine, Pangolin, AutoMine,DwarvesGraph)
  - Both #cores and amount of memory is limited to one machine
- Single-machine out-of-core systems (RStream)
  - Scale to large graph with external storage
  - #cores is still limited to one machine
  - Sacrifice efficiency: to fully utilize disk bandwidth, a less efficient algorithm with graph streaming and relational join—much slower than recent systems
- Distributed systems with graph replication (Arabesque, Fractal, GraphPi)
  - The complete graph data replicated in each machine
  - Scaling #cores, but not memory
- Distributed systems with **graph partition** (G-Miner, G-Thinker)
  - Both #cores and amount of memory can scale
  - Current systems sacrifice efficiency and programmability

Graph Mining Systems with Graph Partition

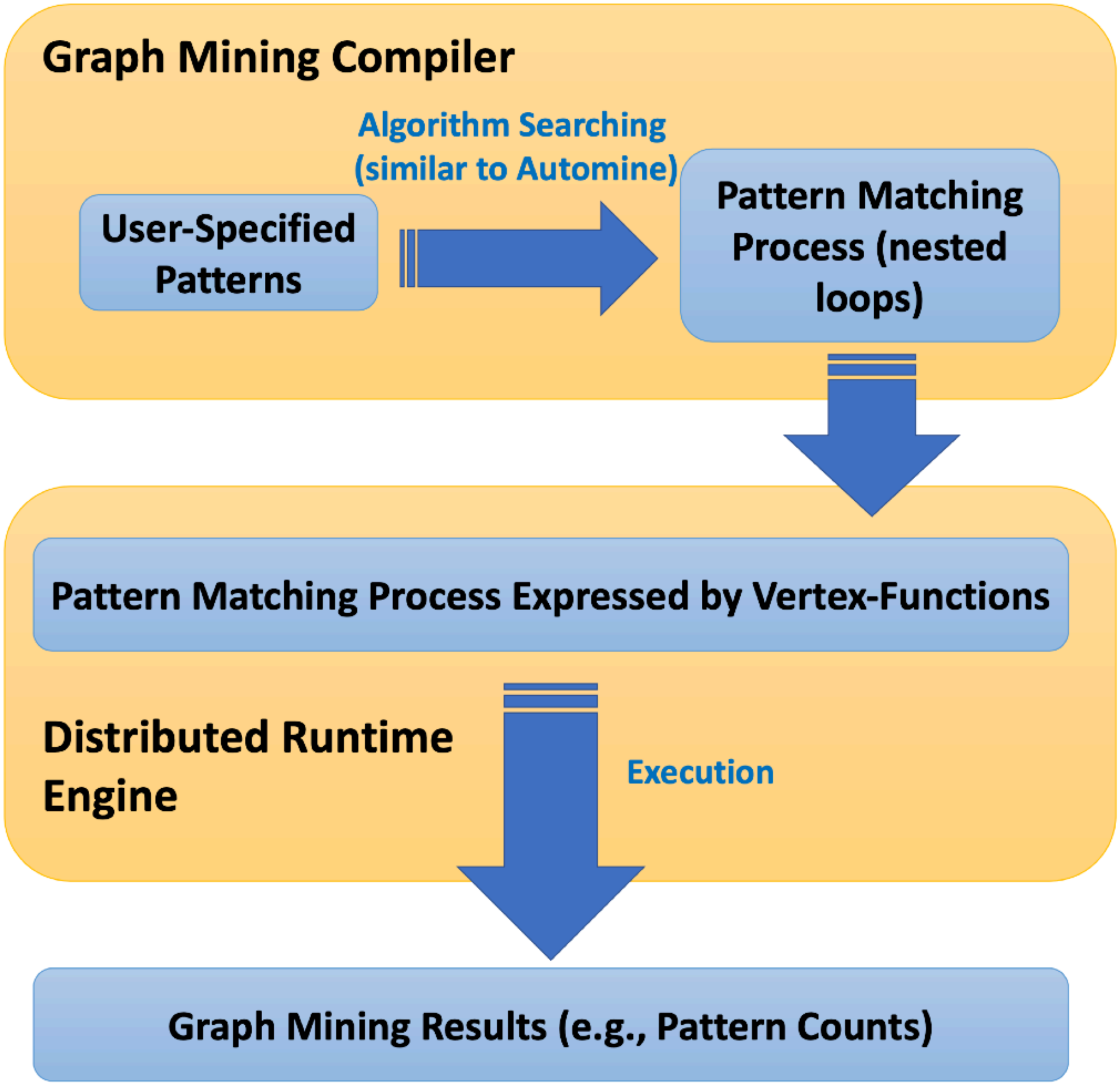
- **Poor programmability**: complicated task-based subgraph-centric model
  - Users responsible for dividing the enumeration process into a number of subgraph-centric tasks
  - Each task: users specify the subgraph containing all data needed
    - Example: clique, task—"counting the number of k-cliques containing a given vertex", subgraph—an induced subgraph including all 1-hope neighbors of the vertex
  - Users need to handle system problems such as stragglers
- **Inefficient system** for communication/computation scheduling
  - Reference-counting based SW cache with GC for fetched remote data
  - Triangle counting on Patents dataset (3.8M vertices)
    - G-Thinker (8 nodes: 16 cores each, 128 cores in total): 285.3s
    - A simple single-thread implementation (reported in AutoMine): 6.2s
    - Peregrine with 16 cores single machine: 1.1s

Goals & Problems

- **Khuzdul**: A distributed graph mining system with graph partition with simple programming interface and high performance
- Problem 1: Can users just specify the patterns without considering all other system issues?
- Problem 2: How to efficiently control scheduling of computation and communication?
- Problem 3: How to achieve efficient implementation?

Khuzdul: Key Ideas

- Breaking down the vertex-set-based enumeration process into smaller operations that can be expressed with vertex functions
  - Vertex functions transparent to users, unlike graph processing—"think like the vertex"
- The inter-loop dependency among vertex functions
- Efficient multi-level scheduler designed for inter-loop dependency
- Optimizations to reduce data movements

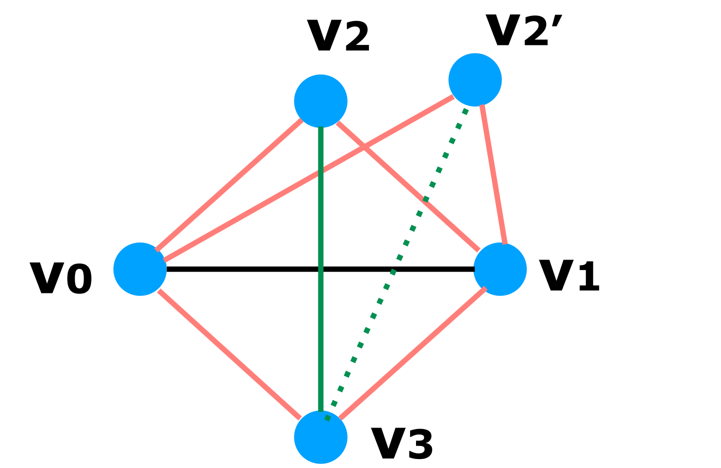


Inter-loop Dependent Vertex Function

```
for v_0 in V(G):
  for v_1 in N(v_0):
    S0 = intersect(N(v_0), N(v_1)):
      for v_2 in S0:
        S1 = intersect(S0, N(v_2))
        for v_3 in S1:
          ++ 4-clique-count
```

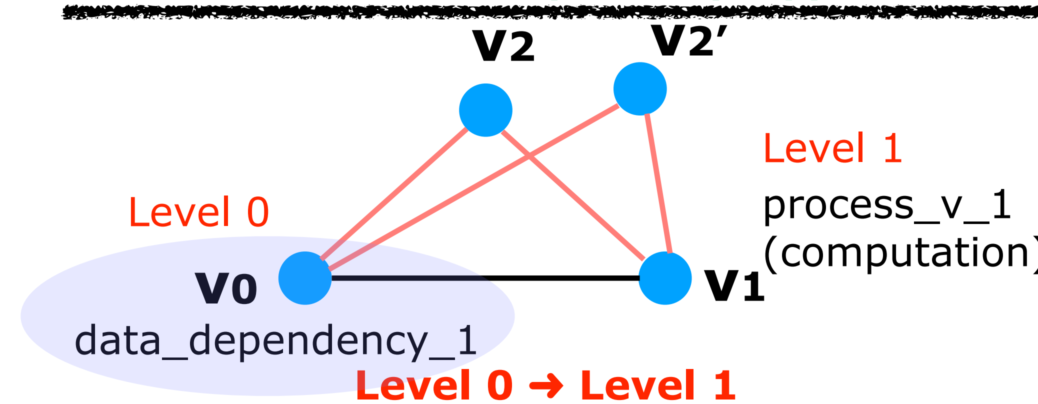
Inter-loop data dependency

A reduce operation: No loop-carried (intra-loop) data dependency

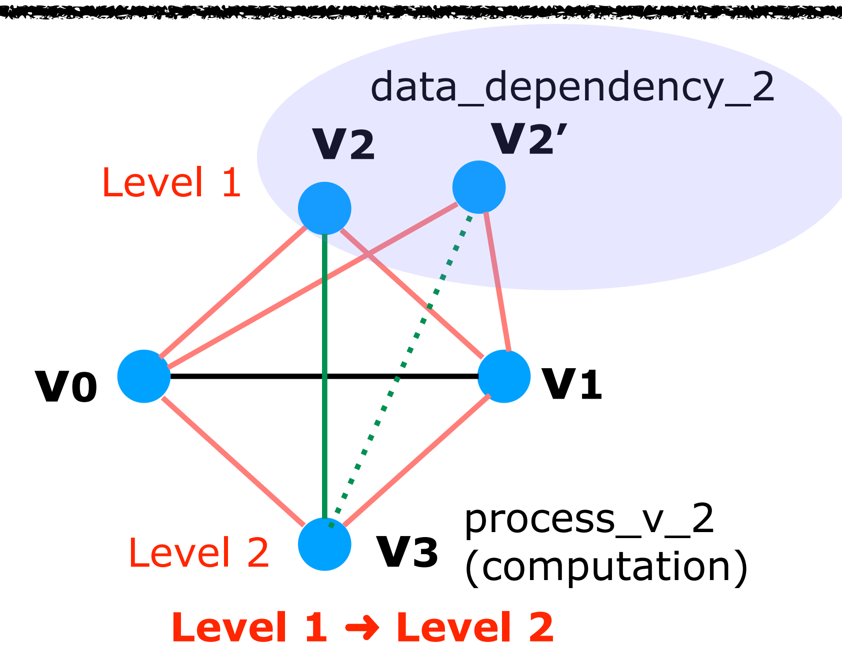


$S0=intersect(N(v_0),N(v_1))$
$S1=intersect(S0,N(v_2))$

Extract the inter-loop dependency explicitly by several vertex-functions




```
data_dependency_0 = SOME_INITIALIZER
for v_0 in V(G):
  v_1_candidates, data_dependency_1 = process_v_0(data_dependency_0, v_0, N(v_0))
  for v_1 in v_1_candidates:
    v_2_candidates, data_dependency_2 = process_v_1(data_dependency_1, v_1, N(v_1))
    for v_2 in v_2_candidates:
      v_3_candidates = process_v_2(data_dependency_2, v_2, N(v_2))
        ...
          for v_n-1 in v_n-1_candidates:
            process_v_n-1(data_dependnecy_n-1, v_n-1, N(v_n-1))
```



Inter-loop Dependent Vertex Function

- We can illustrate the idea conceptually with Python-like pseudocode

```
for v_0 in V(G):
  for v_1 in N(v_0):
    S0 = intersect(N(v_0), N(v_1)):
      for v_2 in S0:
        S1 = intersect(S0, N(v_2))
        for v_3 in S1:
          ++ 4-clique-count
```

```
def process_v_0(data_dependency_0, v_0, N(v_0)):
  dependency = {}
  depdendency["v_0_nbrs"] = N(v_0)
  return N(v_0), dependency

def process_v_1(data_dependency_1, v_1, N(v_1)):
  N(v_0) = data_dependency_1["v_0_nbrs"]
  S0 = intersect(N(v_0), N(v_1))
  dependency = {}
  dependency["s_0"] = S0
  return S0, dependency

def process_v_2(data_dependency_2, v_2, N(v_2)):
  S0 = data_dependency_2["s_0"]
  S1 = intersect(S0, N(v_2))

data_dependency_0 = SOME_INITIALIZER
for v_0 in V(G):
    v_1_candidates, data_dependency_1 = process_v_0(data_dependency_0, v_0, N(v_0))
  for v_1 in v_1_candidates:
    v_2_candidates, data_dependency_2 = process_v_1(data_dependency_1, v_1, N(v_1))
    for v_2 in v_2_candidates:
      v_3_candidates = process_v_2(data_dependency_2, v_2, N(v_2))
        ...
            for v_n-1 in v_n-1_candidates:
            process_v_n-1(data_dependnecy_n-1, v_n-1, N(v_n-1))
```

Inter-loop Dependent Vertex Function

- Our system generates C++ codes based on the user-specified patterns

```
Aggregator<uint64_t> triangle_cnt_agg;

class ProcessSecondTriangleVertex: public VertexFunction {
  Set<VertexFunction*> process_vertex(VertexId v, VertexSet neighbors, Objects shared_objs, Buffer workspace) {
    VertexSet * v_0_nbrs = shared_ojbs->get("v_0_nbrs"); // obtain the shared N(v_0) from the previous vertex-function
    VertexSet intersection_result(workspace); // use the workspace buffer to create the vertex set containing N(v_0)
    \intersect N(v_1)
    // cnt += |N(v_0)\intersect N(v_1)|
    VertexSet::intersect(v_0_nbrs, &neighbours, &intersection_result);
    triangle_cnt_agg.add(intersection_result.size());
  }
  return NULL;
}
```

Inter-loop data dependence

```
class ProcessFirstTriangleVertex: public VertexFunction {
  Set<VertexFunction*> process_vertex(VertexId v, VertexSet neighbors, Objects shared_objs, Buffer workspace) {
    Set<VertexFunction*> S;
    // specify the objects to be shared with new vertex-functions
    Objects objs_to_share;
    objs_to_share.put("v_0_nbrs", &neighbours);
    // for v_1 \in N(v_0):
    for (VertexId u in neighbours) {
      // allocate a vertex-function to calculate 'N(v_0) \intersect N(v_1)'
      VertexFunction * f = allocate_vertex_function<ProcessSecondTriangleVertex>(u);
      f->set_shared_objs(objs_to_share);
      // the buffer needed by the new vertex-function stores N(v_0) \intersect N(v_1)
      // whose size cannot exceed min(|N(v_0)|, |N(v_1)|)
      f->set_workspace_size(min(get_degree(u), get_degree(v)) * sizeof(VertexId));
      S.add(f);   Level 0 → Level 1
    }
    return S;
  }
};
```

```
Implemented Triangle-Counting Algorithm:
N(v): the neighbour vertex set of v;
cnt = 0;
for v_0 in V(G):
    for v_1 in N(v_0):
        cnt += |N(v_0)\intersect N(v_1)|
```

Abstract Execution Model

How to pick up the next vertex-function determines the concrete execution model.
FIFO: breadth-first schedule
FILO: depth-first schedule

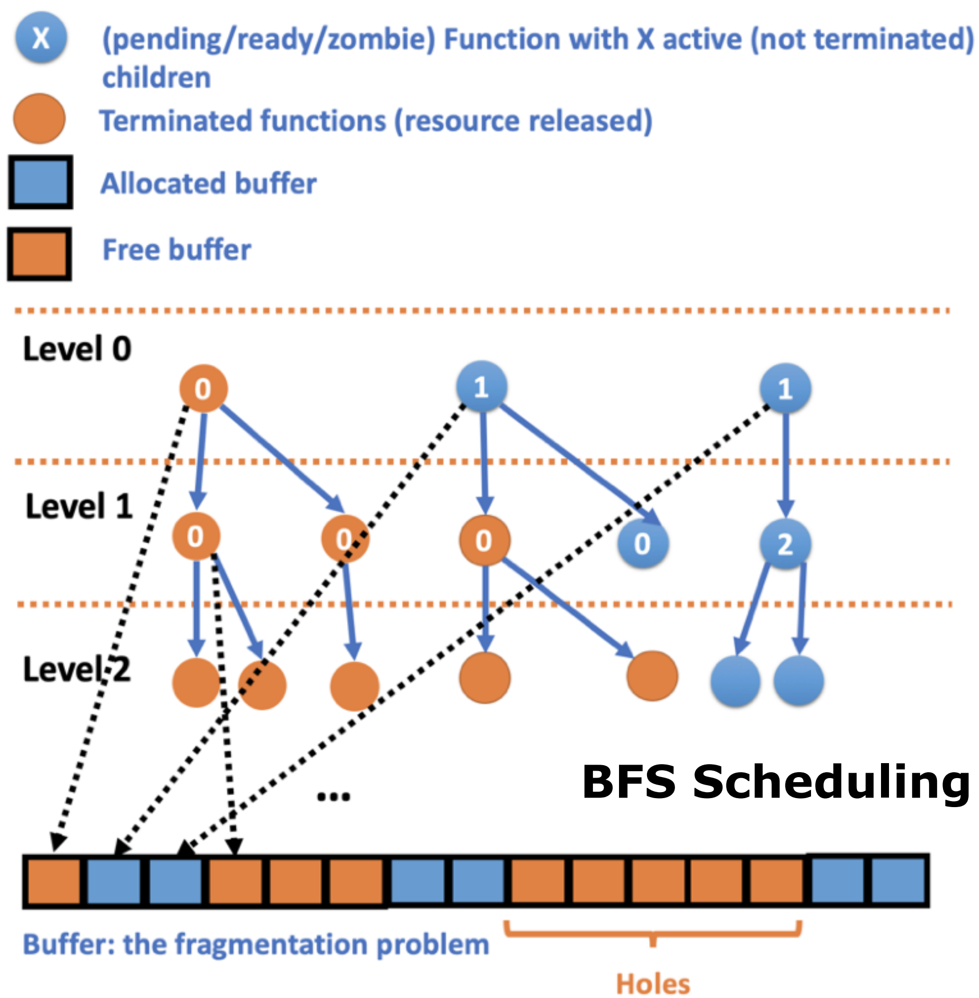
$\boldsymbol{F_v} \leftarrow$ a set of user-provided initial vertex-functions
while $\boldsymbol{F_v}$ is not empty **do**
  pick up a vertex-function $\boldsymbol{f_v}$ from $\boldsymbol{F_v}$
  remove $\boldsymbol{f_v}$ from $\boldsymbol{F_v}$
  $\boldsymbol{New\_F_v} \leftarrow$ execute($\boldsymbol{f_v}$)
  for all $\boldsymbol{f_v} \in \boldsymbol{New\_F_v}$ **do**
    add $\boldsymbol{f_v}$ to $\boldsymbol{F_v}$
  end for
end while

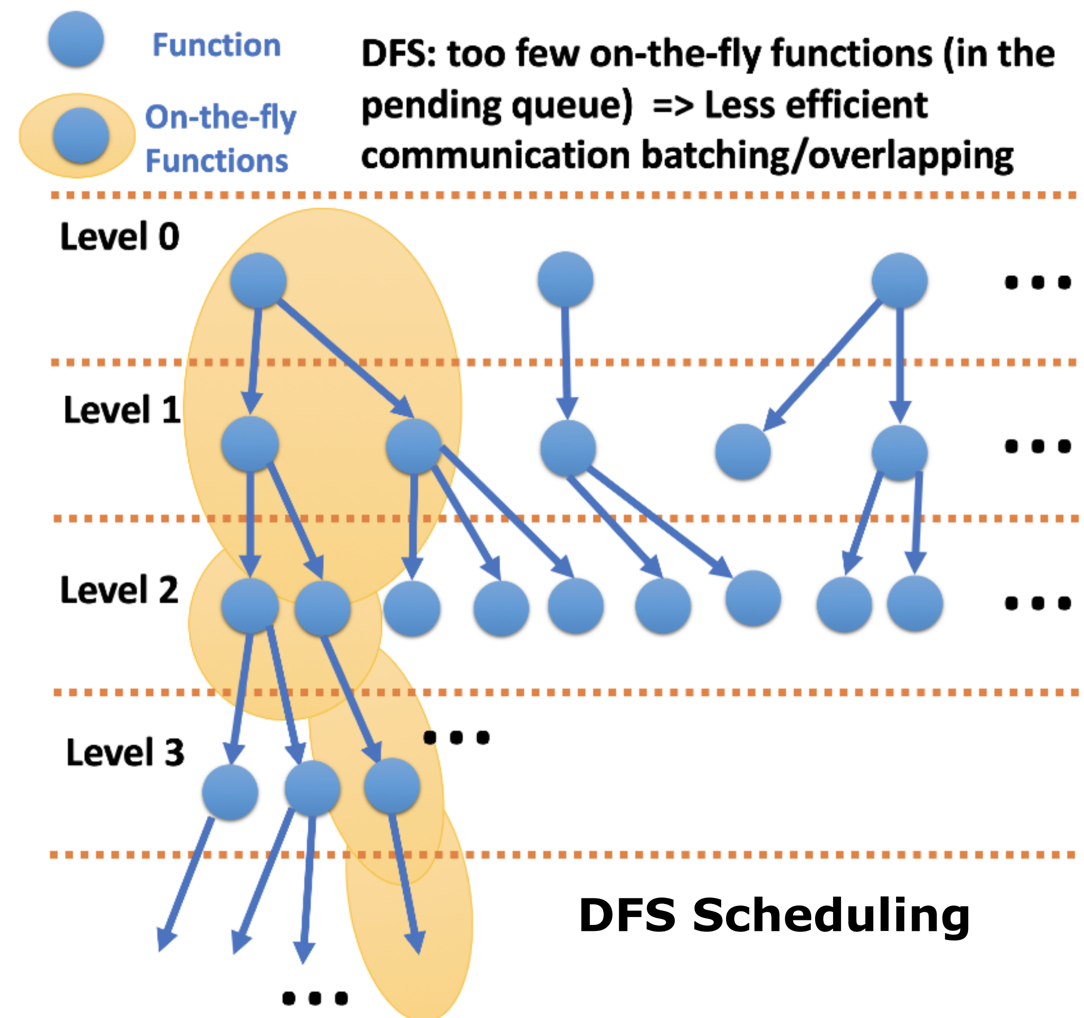
consume item

produce new item

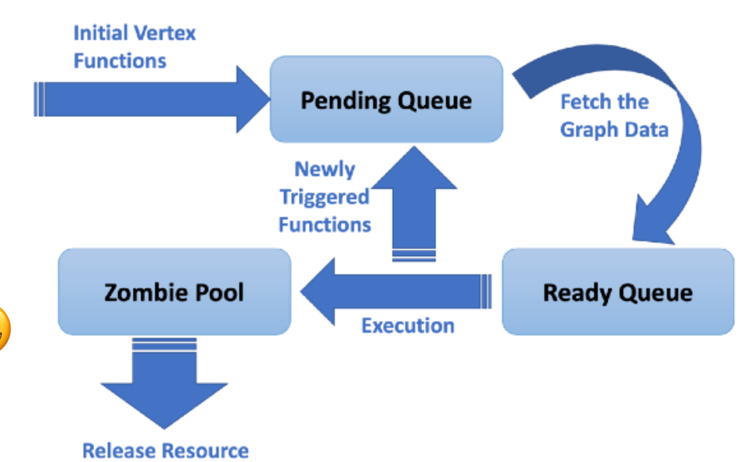
The execution of a vertex-function may trigger new vertex-functions in the next loop level (in the algorithm).

Problems of Conventional Scheduler

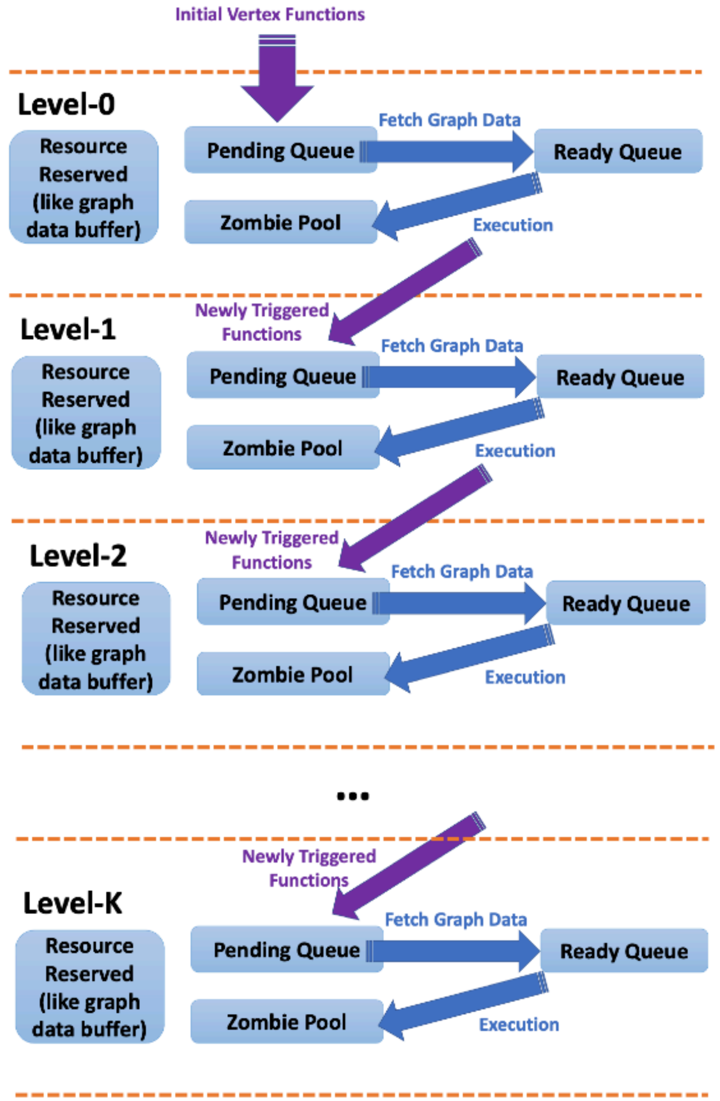


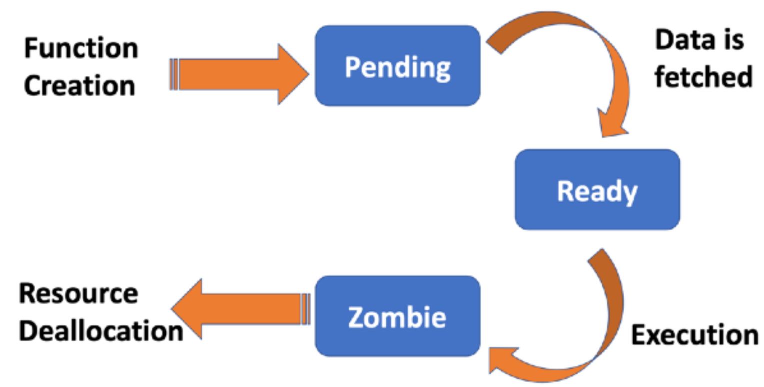


- FIFO (BFS)
  - Memory fragmentation 😫
  - High overhead 😫
  - Communication batching and overlapping with computation 😁
- FILO (DFS)
  - No effective communication batching and communication/computation overlapping 😫

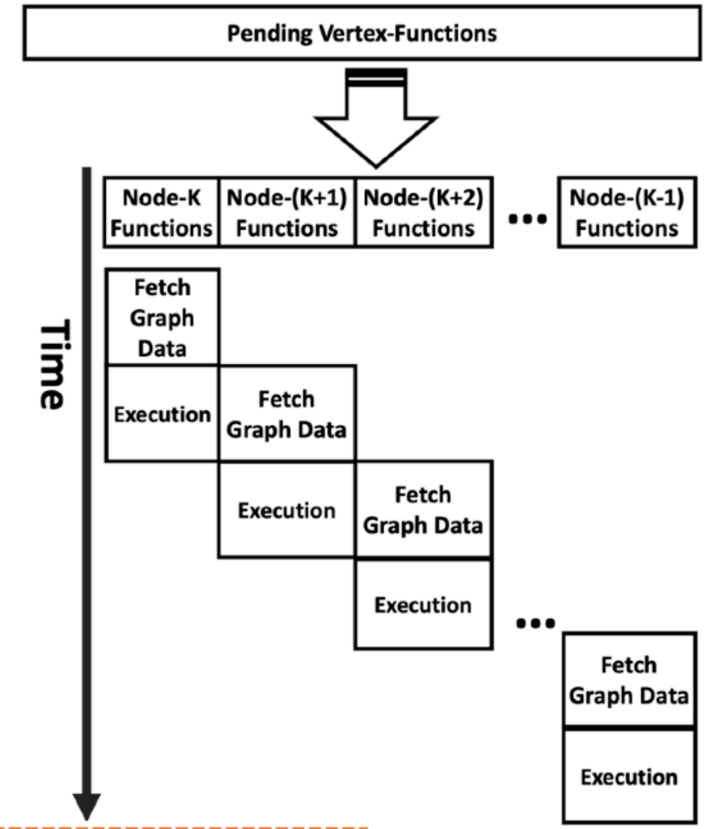


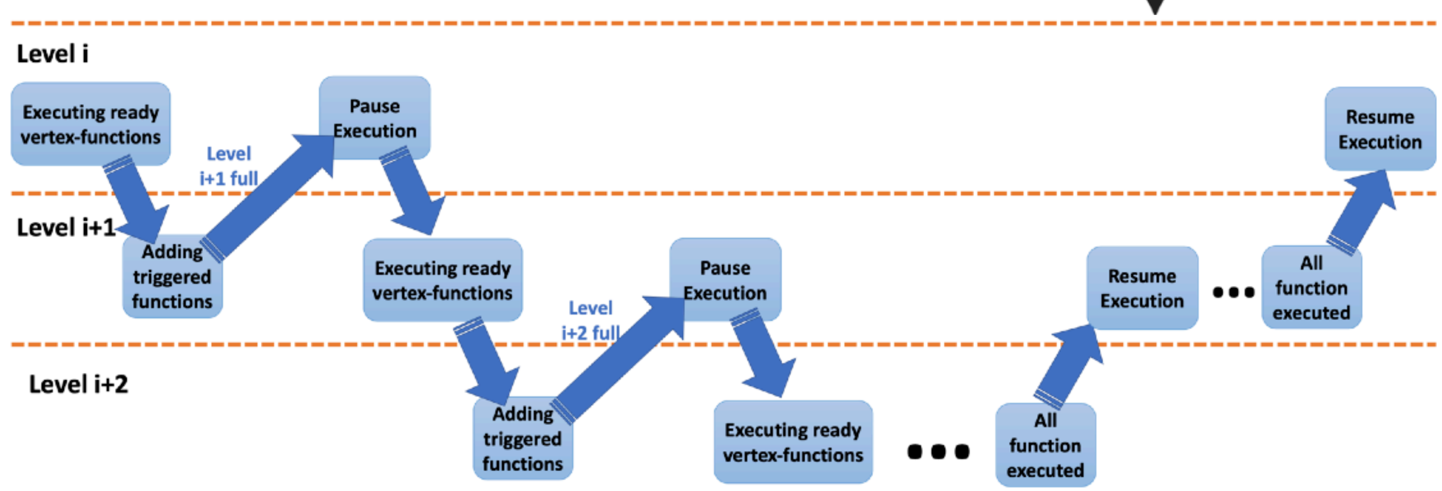
Multi-level Scheduler





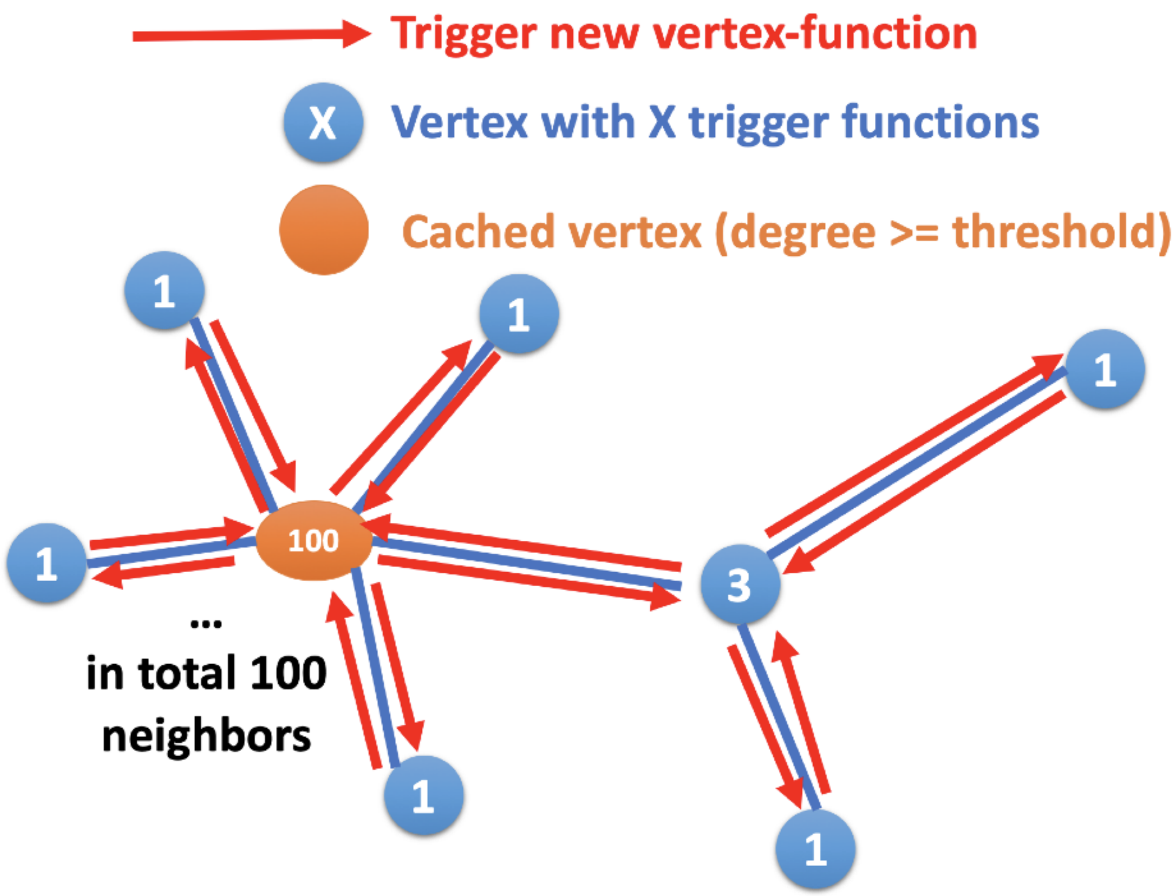
Vertex function life time

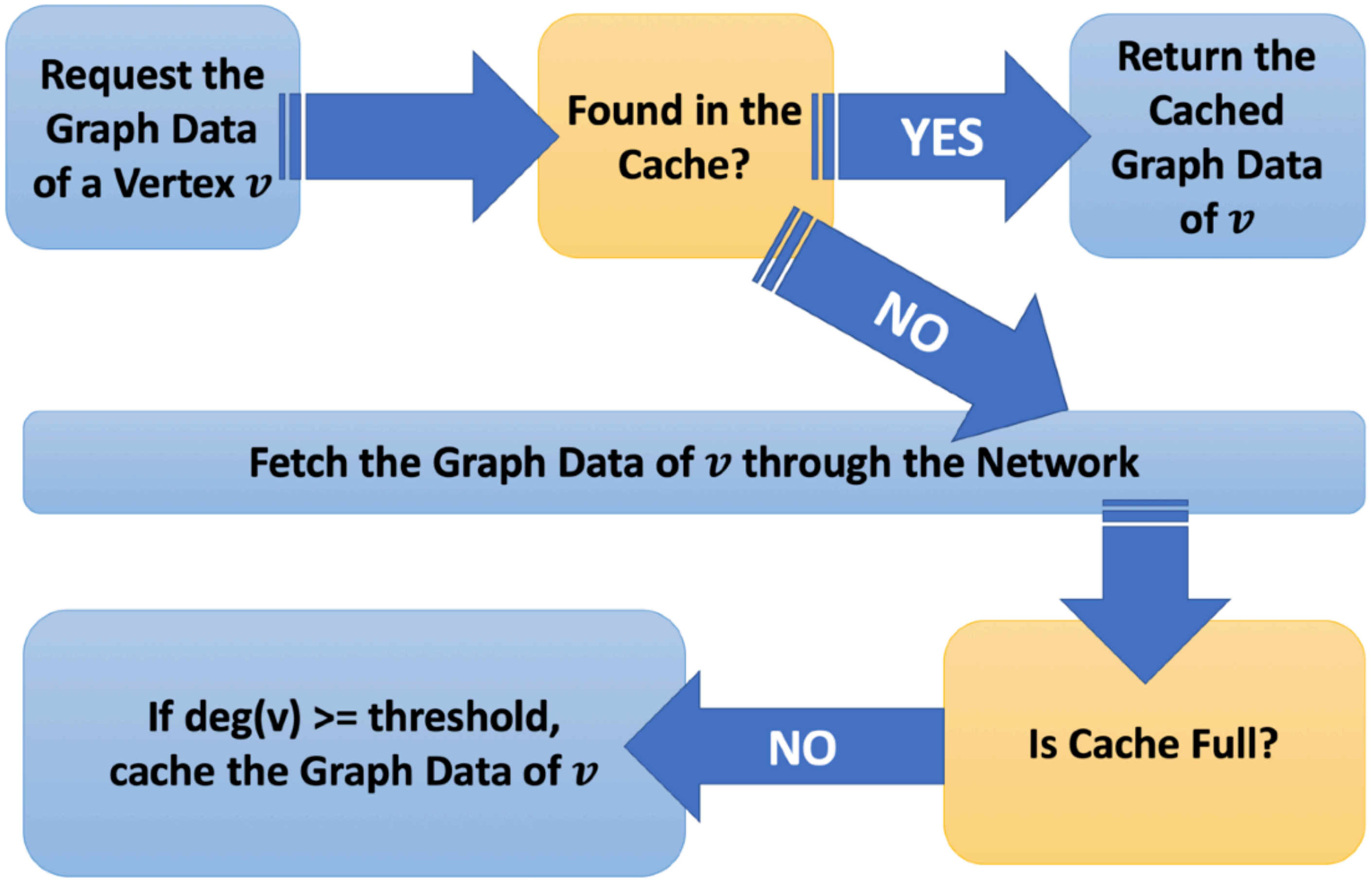




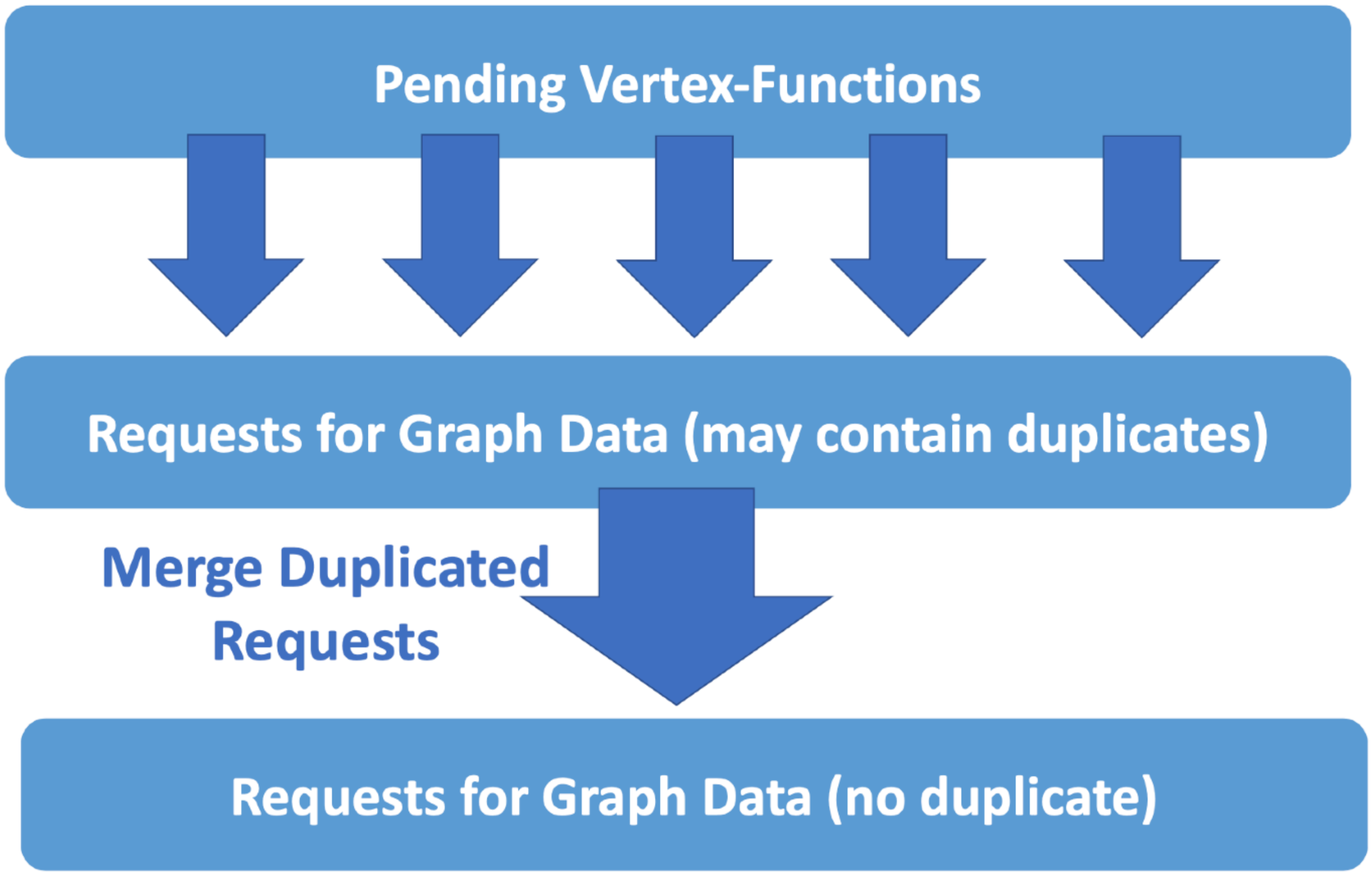
- **Same level**
  - Communication batching and overlapping with computation
  - Shuffle the vertex functions into different groups, each group only fetches the data from one node
  - Each vertex function only accesses the neighbors of a given vertex—all in one node
- **Different level**
  - Avoid memory fragmentation and reference counting

Reducing Communication with SW Cache



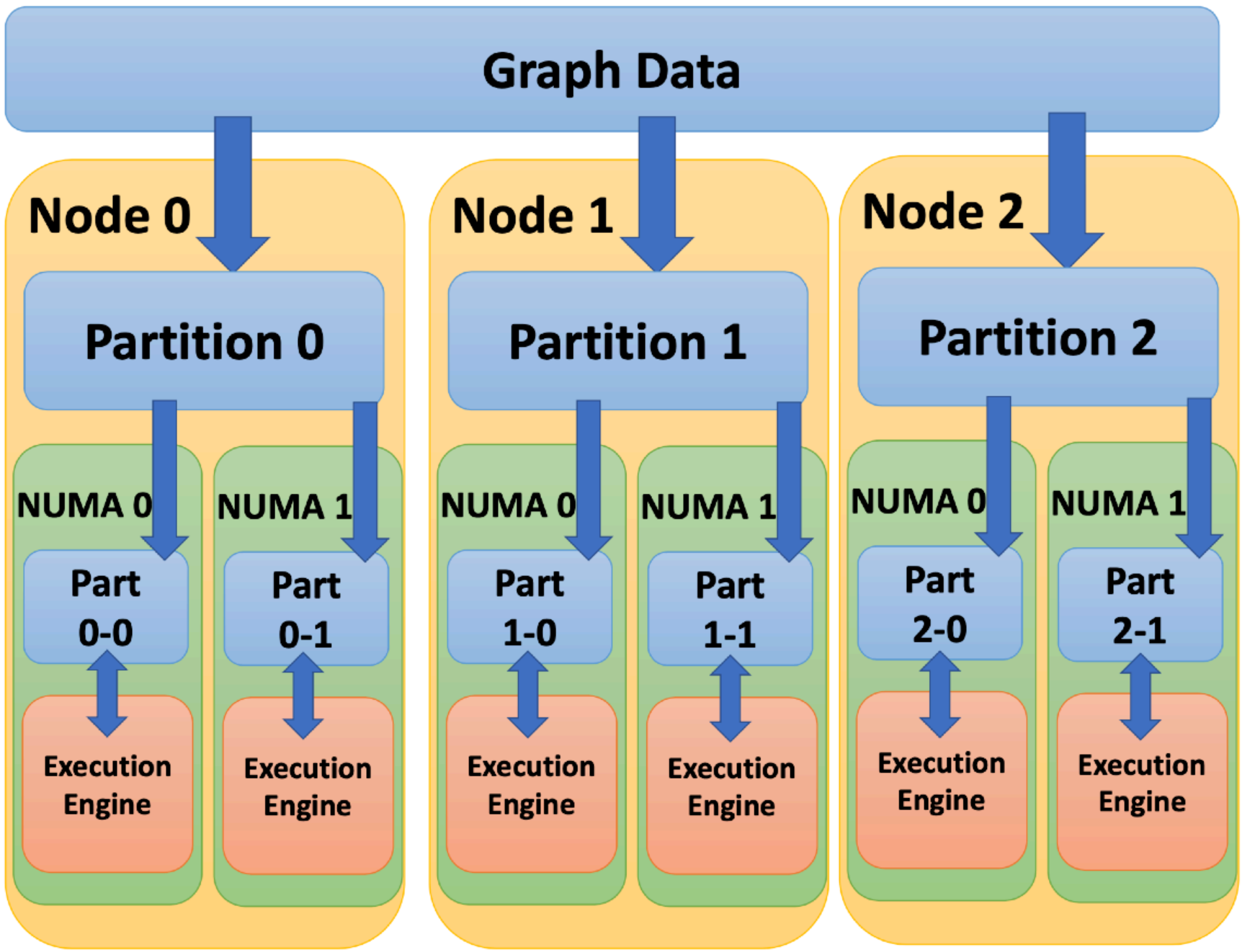


Communication Merging



- Efficient hash-table based implementation that allows false negative in merging

NUMA Subpartition



- Each NUMA socket maintains an execution engine
  - All the buffers of this engine (e.g., fetched graph data, workspace buffer) are NUMA-local
  - Avoid expensive cross-socket memory accesses

Khuzdul Evaluation

- **System:**
  - Each node has two 8-core Intel Xeon E5-2630 CPUs (hyperthreading disabled) and 64GB DRAM
  - Network: 56GBps InfiniBand
- **Applications: triangle, 3-motif, 4-clique, 5-clique**
- **Single-machine systems**
  - In-house AutoMine implementation
  - Peregrine (EuroSys'20)
  - Pangoline (CPU/GPU) (VLDB'20)
- **Distributed Systems**
  - *Partitioned* graph: G-thinker (ICDE'20)
  - *Replicated* graph: GraphPi (SC'20)

Graph datasets

Comparing with G-Thinker: Partitioned Graph Triangle Counting

Graph	Khuzdul	G-thinker
wk	24.4ms	1.0s (41.0x)
mc	40.0ms	2.1s (52.5x)
pt	254.9ms	285.3s (1,119.3x)
lj	826.8ms	31.6s (38.2x)
uk	690.1s	CRASHED
tw	2171.2s	CRASHED
fr	81.2s	CRASHED

Comparing with GraphPi: Replicated Graph

Application	Graph	Khuzdul (8-node)	GraphPi (8-node)
Triangle	wk	24.4ms	534.3ms
	Mc	40.0ms	704.4ms
	pt	257.5ms	6.7s
	lj	826.8ms	9.8s
	uk	690.1s	1268.4s
	tw	2171.2s	2886.5s
	Fr	81.2s	169.2s
3-motif	wk	24.8ms	1.1s
	mc	52.8ms	1.5s
	pt	429.3ms	13.8s
	lj	1.9s	20.1s
	uk	3,005.1s	1,380.7s
	tw	9401.0s	3,032.1s
	Fr	190.5s	388.5s

Comparing with GraphPi: Replicated Graph

Application	Graph	Khuzdul (8-node)	GraphPi (8-node)
4-clique	wk	43.0ms	500.4ms
	mc	321.5ms	844.0ms
	pt	412.7ms	6.7s
	lj	5.3s	12.8s
	uk	17,241.0s	31,008.6s
	tw	18,817.0s	TIMEOUT
	fr	190.5s	177.8s
5-clique	wk	78.1ms	522.1ms
	mc	11.1s	8.2s
	pt	822.5ms	6.8s
	lj	188.6s	174.7s
	Fr	220.3s	260.0s

Comparing with Single-Machine Systems

Application	Graph	Khuzdul (8-node / single-node)	Automine	Peregrine	Pangolin
Triangle	wk	24.4ms / 23.4ms	9.9ms	8.3ms	7ms
	mc	40.0ms / 100.2ms	52.3ms	68.7ms	56ms
	pt	257.5ms / 1.6s	330.7ms	1.1s	289ms
	lj	826.8ms / 5.5s	2.8s	3.8s	2.2s
	uk	690.1s / 6132.8s	7305.0s	4667.0s	26.6s
	tw	2171.2s / 16806.9s	30866.1s	20605.5s	747.7s
	Fr	81.2s / 515.3s	378.3s	305.2s	384.6s

Comparing with Single-Machine Systems

Application	Graph	Khuzdul (8-node / single-node)	Automine	Peregrine	Pangolin
3-motif	Wk	24.8ms / 39.8ms	27.4ms	26.9ms	45ms
	mc	52.8ms / 255.2ms	160.3ms	84.7ms	288ms
	pt	429.3ms / 2.8s	930.9ms	1.7s	1.5s
	lj	1.9s / 12.8s	8.9s	4.6s	29.2s
	uk	3,005.1s / 24700.190s	TIMEOUT (>10 hours)	4660.9s	TIMEOUT
	tw	9401.0s / TIMEOUT	TIMEOUT	20477.6s	TIMEOUT
	fr	190.5s / 1420.8s	1206.8s	316.1s	6305.9s

Comparing with Single-Machine Systems

Application	Graph	Khuzdul (8-node / single-node)	Automine	Peregrine	Pangolin
4-clique	wk	43.0ms / 77.0ms	47.2ms	104.6ms	47ms
	mc	321.5ms / 1.9s	1.2s	1.8s	2.8s
	pt	412.7ms / 2.6s	381.0ms	1.3s	773ms
	lj	5.3s / 37.6s	31.3s	49.6s	54.7s
	uk	17,241.0s / TIMEOUT	TIMEOUT	TIMEOUT	MEM
	tw	18,817.0s / TIMEOUT	TIMEOUT	TIMEOUT	MEM
	Fr	157.9s / 887.6s	570.3s	1237.5s	MEM

Comparing with Single-Machine Systems

Application	Graph	Khuzdul (8-node / single-node)	Automine	Peregrine	Pangolin
5-clique	Wk	78.1ms / 226.3ms	124.7ms	477.2ms	146ms
	mc	11.1s / 71.2s	46.8s	78.0s	132.0s
	pt	822.5ms / 5.3s	408.4ms	1.5s	967ms
	lj	188.6s / 1385.8s	982.9s	2076.6s	MEM
	Fr	220.3s / 1593.9s	900.2s	3032.8s	MEM

Communication/Computation Overlapping

Application	Graph	Runtime / Communication Time on the Critical Path (with overlap)	Runtime / Communication Time on the Critical Path (without overlap)
3-motif	mc	52.8ms / 2.4ms	80.7ms / 24.9ms
	pt	429.3ms / 144.6ms	556.7ms / 276.0ms
	lj	1.86s / 0.13s	2.81s / 1.03s
	Fr	190.5s / 11.0s	296.6s / 116.4s
5-clique	mc	11.1s / 0.017s	11.8s / 0.6s
	pt	822.5ms / 171.7ms	938.1ms / 305.0ms
	lj	188.6s / 0.56s	197.9s / 13.4s
	Fr	220.3s / 15.0s	343.8s / 138.3s

Duplicated Request Merging

Application	Graph	Runtime/Communication Volume (with request merging)	Runtime/Communication Volume (without request merging)
3-motif	mc	52.8ms / 338.7MB	61.1ms / 623.3MB
	pt	429.3ms / 1.8GB	440.6ms / 2.1GB
	lj	1.86s / 16.9GB	2.31s / 37.8GB
	Fr	190.5s / 3.2TB	192.0s / 3.4TB
5-clique	mc	11.1s / 12.9GB	18.5s / 281.0GB
	pt	822.5ms / 1.3GB	861.3ms / 1.8GB
	lj	188.6s / 79.5GB	223.4s / 3.5TB
	Fr	220.3s / 744.7GB	722.0s / 13.9TB

SW Graph Cache

Application	G	Runtime / Communication Time on the Critical Path / Communication Volume (with cache)	Runtime / Communication Time on the Critical Path / Communication Volume (without cache)
3-motif	mc	52.8ms / 2.4ms / 0.34GB	52.3ms / 1.7ms / 0.34GB
	pt	429.3ms / 144.6ms / 1.8GB	417.1ms / 126.8ms / 2.0GB
	lj	1.86s / 0.13s / 16.9GB	1.92s / 0.18s / 18.3GB
	Fr	190.5s / 11.0s / 3.2TB	200.7s / 22.5s / 4.2TB
	uk	3,005.126s / 1.9s / 829.2GB	4595.6s / 638.5s / 83.2TB
5-clique	mc	11.1s / 0.017s / 12.9GB	11.1s / 0.019s / 16.6GB
	pt	822.5ms / 171.7ms / 1.3GB	855.9ms / 209.5ms / 1.8GB
	lj	188.6s / 0.56s / 79.5GB	187.1s / 0.56s / 121.9GB
	Fr	220.3s / 15.0s / 744.7GB	255.4s / 48.6s / 3.7TB

NUMA-aware Graph Subpartition

Application	Graph	Runtime with sub-partitioning	Runtime without sub-partitioning
3-motif	mc	52.8ms	74.5ms
	pt	429.3ms	970.5ms
	lj	1.86s	3.1s
	Fr	190.5s	327.2s
5-clique	mc	11.1s	16.1s
	pt	822.5ms	2.0s
	lj	188.6s	319.8s
	Fr	220.3s	447.5s

Scalability: LiveJournal Graph

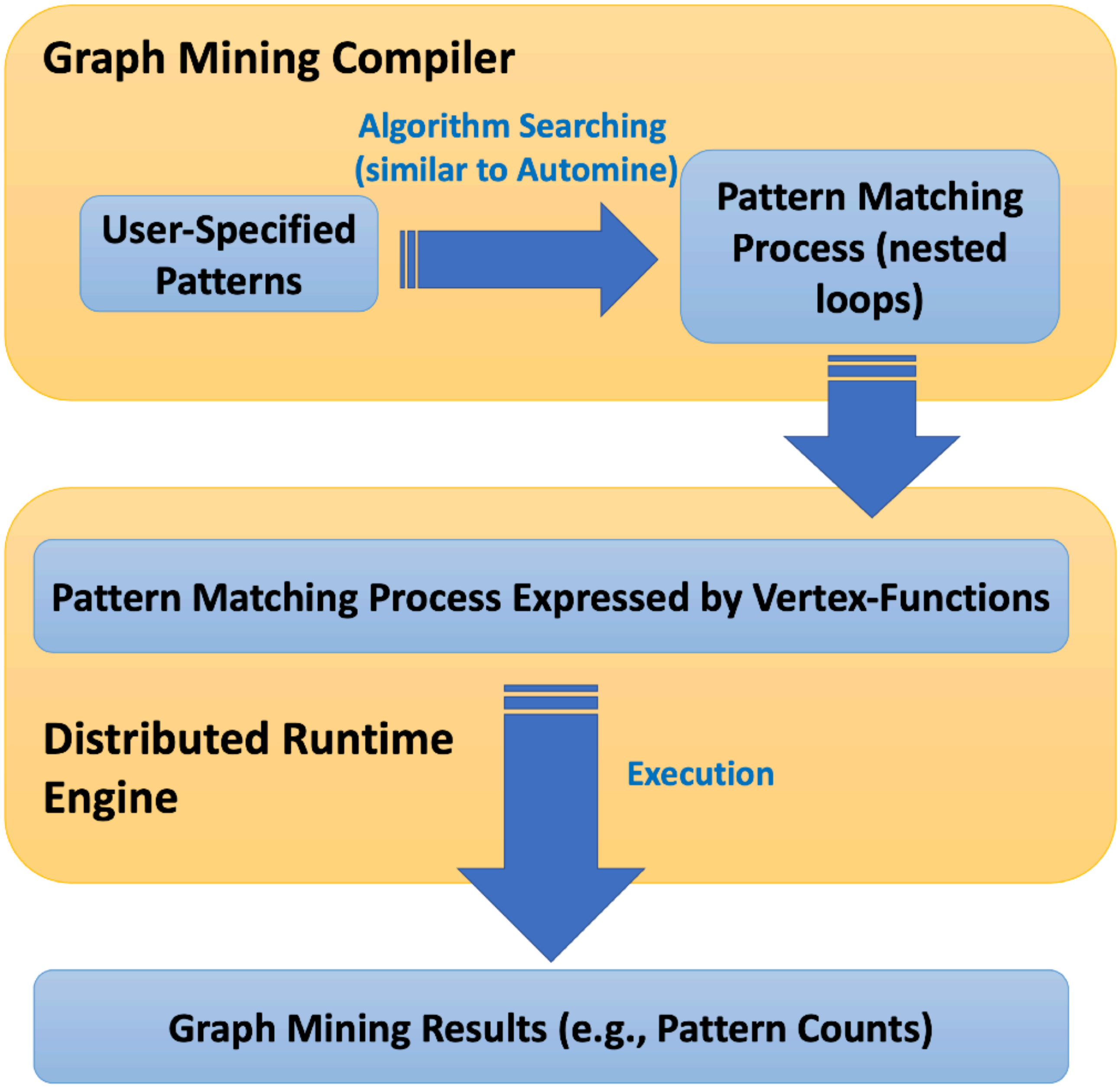
#Nodes	Triangle	3-Motif	4-Clique	5-Clique
1	5.5s	12.8s	37.6s	1385.8s
2	2.8s	6.5s	19.1s	704.7s
4	1.5s	3.4s	9.9s	361.4s
8	826.8ms	1.9s	5.3s	188.6s
8-node speedup over 1-node	6.65x	6.74x	7.09x	7.35x

Scalability: Friendster Graph

#Nodes	Triangle	3-Motif	4-Clique	5-Clique
1	515.3s	1420.8s	887.6s	1593.9s
2	257.3s	709.3s	494.7s	816.7s
4	150.4s	364.8s	289.7s	421.3s
8	81.2s	190.5s	157.9s	220.3s
8-node speedup over 1-node	6.35x	7.46x	5.62x	7.24x

Khuzdul: Key Ideas

- Breaking down the vertex-set-based enumeration process into smaller operations that can be expressed with vertex functions
  - Vertex functions transparent to users, unlike graph processing—“think like the vertex”
- The inter-loop dependency among vertex functions
- Efficient multi-level scheduler designed for inter-loop dependency
- Optimizations to reduce data movements



High Performance Graph Mining Systems

Xuehai Qian
University of Southern California