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
Coexpression networks

[Magwene et al. (2004)]
[Grandjean, M. (2016)]
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 $f : V_0 \rightarrow V_1$ 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 $\{(u, v) | u, v \in V_g, (u, v) \in E\} = E_g$
  - Consider vertices: if two vertices $u$ and $v$ are in $g$, and there is an edge between them in $G$, the edge must be also in $g$
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: \( C(\text{vertex-induced 3-chain}) = C(\text{edge-induced 3-chain}) - 3C(\text{edge-induced triangle}). \)
    \( C = \text{count} \)
  - The # of vertex-induced count of 3-chain of \( G \) (on the right) = \( 8 - 3 \times 2 = 2 \)
Edge/Vertex-induced Embeddings

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

```plaintext
1: for v₀ ∈ V do
2:   for v₁ ∈ N(v₀) do
3:     for v₂ ∈ N(v₀) - N(v₁) do
4:       count_{vertex-induced} ← count_{vertex-induced} + 1
5:     end for
6:   end for
7: end for
```

Vertex-induced 3-chain

```plaintext
1: for v₀ ∈ V do
2:   for v₁ ∈ N(v₀) do
3:     for v₂ ∈ N(v₀) do
4:       count_{edge-induced} ← count_{edge-induced} + 1
5:     end for
6:   end for
7: end for
```

Edge-induced 3-chain

• In the talk, we consider edge-induced embeddings

*N* 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

Step i

Node 0

Size-\(i\) Embedding Candidates part 0

Extend by one edge/vertex

Size-(\(i+1\)) Embedding Candidates part 0

Filter and process

Size-(\(i+1\)) Embedding Candidates part 0

Node 1

Size-\(i\) Embedding Candidates part 1

Size-(\(i+1\)) Embedding Candidates part 1

Node 2

Size-\(i\) Embedding Candidates part 2

Size-(\(i+1\)) Embedding Candidates part 2

Shuffle across all nodes for load balancing

Step \(i+1\)

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

1: for $v_0 \in V$ do
2:     for $v_1 \in N(v_0)$ do
3:         for $v_2 \in N(v_0) \cap N(v_1)$ do
4:             for $v_3 \in N(v_0) \cap N(v_1) \cap N(v_2)$ do
5:                 $count_{\text{triangle}} \leftarrow count_{\text{triangle}} + 1$
6:         end for
7:     end for
8: end for

Different Embedding Enumeration Method

Computation Reuse

Estimated Cost

Cost: XXX

Cost: YYY (Minimum)

Cost: ZZZ

Generated C++ Implementation

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:
1. \((v_0,v_1,v_2,v_3)\)
2. \((v_2,v_1,v_0,v_3)\)
3. \((v_2,v_3,v_0,v_1)\)
4. \((v_0,v_3,v_2,v_1)\)

→ should be only counted once

Symmetry breaking: add constraints—
1. \((v_0<v_2)\) and \((v_1<v_3)\)
2. only one embedding left

Jamshidi et al. Peregrine: A Pattern-Aware Graph Mining System. EuroSys’20
Pangolin: Application-specific Optimizations with flexible APIs

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

```c
1 bool toExtend(Embedding emb, Vertex v);
2 bool toAdd(Embedding emb, Vertex u)
3 bool toAdd(Embedding emb, Edge e)
4 Pattern getPattern(Embedding emb)
5 Pattern getCanonicalPattern(Pattern pt)
6 Support getSupport(Embedding emb)
7 Support Aggregate(Support s1, Support s2)
8 bool toPrune(Embedding emb);
```
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

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

\[ p = (V_p, E_p) \]

\( V_C \) \( V_p \)

\( (v_0, v_1, v_2, \ldots, |V_p| - 1) \)

\( v_i \)

\( M \)

*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

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?
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

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

Diagram:

Compilation Program → DwarvesGraph API → Compiler → Algorithm Generation Engine → Compiler Backend → Runtime (Libraries for graph computing) → Processed application program → Results
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?

![Diagram](image)

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.

---

[Diagrams illustrating the concepts]
A Simple Example

Partial-embedding centric APIs

User programs

```c++
1 // User-invoking functions
2 int get_pattern_count();
3 std::vector<Embedding> materialize(PartialEmbedding pe, int num);
4 // User-defined functions (UDFs)
5 void process_partial_embedding(PartialEmbedding pe, int count);
```

```cpp
1 // -------- three_chain.cc --------
2 D_GENERATE_CLASS(ThreeChainCountingClass);
3 // ... load the input graph & other initializations
4 ThreeChainCountingClass * app = new ThreeChainCountingClass(graph);
5 cout<<"three-chain count: "<<app->get_pattern_count()<<endl;

```

Compilation program

```cpp
1 int pattern_count = 0;
2 int num_embeddings_to_list = 100;
3 std::vector<Embedding> listed_embeddings;
4 void process_partial_embedding(PartialEmbedding pe, int count) {
5     if (pe.subpattern_id == 0) {
6         int remained = num_embeddings_to_list - pattern_count;
7         if (remained > 0) listed_embeddings.add_all(materialize(pe, remained));
8         pattern_count += count;
9     }
10 }
```

Pattern counting with bounded embedding list

- 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
  - $\text{Dom}(A) = \{0,1\}; \text{Dom}(B) = \{0,1,2,3\}; \text{Dom}(C) = \{0,1,2,3\}$

- **Support**—a metric to quantize the *frequency* of a pattern
  - We use the minimum image-based (MINI) support definition*
  - MINI support is 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 $\leq$ 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
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

### Algorithm

**Input**: Pattern $p$, Cutting Set $V_C$

1. Decompose $p$ using $V_C$ to generate $K$ subpatterns, and the shrinkage patterns
2. $\text{num\_shrinkages}_i = \text{HashTable}()$ ($1 \leq i \leq K$)
3. for all $e_c = (v_0, v_1, \ldots, v_{|V_C|-1})$ matching the cutting set $V_C$ do
   4. $M \leftarrow 1$
   5. for all $i \leftarrow 1 \ldots K$ do
      6. $M_i \leftarrow$ the number of $pe$ extending $e_c$ and matching the $i$-th subpattern
      7. $\text{num\_shrinkages}_{i} = \text{clear}()$
      8. $M \leftarrow M \times M_i$
   9. end for
10. for all $e$ extending $e_c$ and matching one of the shrinkage patterns do
    11. for all $i \leftarrow 1 \ldots K$ do
        12. $\text{pe}_i \leftarrow \text{extract\_subpattern\_embedding}(e, i)$
        13. $\text{num\_shrinkages}_{[\text{pe}_i]} \leftarrow \text{num\_shrinkages}_{[\text{pe}]} + 1$
    14. end for
    15. end for
16. for all $i \leftarrow 1 \ldots K$ do
    17. for all $pe$ extending $e_c$ and matching the $i$-th subpattern do
        18. $\text{count} \leftarrow M/M_i - \text{num\_shrinkages}_{\text{pe}}$
        19. if $\text{count} > 0$ then
            20. $\text{process\_partial\_embedding}(\text{pe}, \text{count})$
        21. end if
    22. end for
13. end for

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

**V_C** is determined by the algorithm generation engine.

Compute the count of each $T_k$ 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.
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 should be executed $n^5p^{10}$ times
    • Patents graph: $n=3.8M, \text{avg}_\text{deg}=8.76, p=2.3 \times 10^{-6}$, line 5 is estimated to execute $3.28 \times 10^{-24}$ times
    • In reality, Patents graph has 3M 5-cliques, line 5 executed for $3M \times 5!$ times

Counting $k$-clique

1: for $v_0 \in V(G)$ do
2:   for $v_1 \in N(v_0)$ do
3:     for $v_2 \in N(v_0) \cap N(v_1)$ do
4:       $\ldots$
5:         for $v_{k-1} \in N(v_0) \cap N(v_1) \cap \ldots \cap N(v_{k-2})$ do
6:           $cnt \leftarrow cnt + 1$
7:         end for
8:       $\ldots$
9:     end for
10:   end for
11: end for

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 a 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

```
1: Cnt ← 0
2: for i ← 1…NumSamples do
3:   v0 ← UniformSample(V(G))
4:   v1 ← UniformSample(N(v0))
5:   if v2 ∈ N(v1) then
6:     Cnt ← Cnt + |V(G)| - |N(v0)| - |N(v1)|
7:   end if
8: end for
9: Cnt ← Cnt / NumSamples
```

* 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

![Diagram showing the process of evaluating subpatterns and selecting cutting sets for separate tuning.]
Circulant Tuning

Subpatterns

P0
Evaluate all Vc with cost model
C0
Selected cutting sets

P1
Evaluate all Vc with cost model
C1

P2
Evaluate all Vc with cost model
C2

Separate Tuning

Fix C1 and C2, evaluate all Vc candidates for C0 considering computation reuse among all patterns
C00

Fix C0 and C2, evaluate all Vc candidates for C1 considering computation reuse among all patterns
C01

Fix C0 and C1, evaluate all Vc candidates for C2 considering computation reuse among all patterns
C02

High Performance Graph Mining Systems
Problem: with symmetry breaking for subpatterns, the complete embeddings cannot be correctly joined.

Symmetry breaking:
- "v₂<v₁" for p₁
- "v₀<v₂" for p₂

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

Only applied when the gain is larger than a threshold, based on cost model.
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)
- Arabeque (SOSP’15)
- Peregrine (EuroSys’20)
- Pangoline (CPU/GPU) (VLDB’20)
- Fractal (SIGMOD’19)
- GraphPi (SC’20)

<table>
<thead>
<tr>
<th>Graph</th>
<th>App</th>
<th>Graph</th>
<th>Our Impl.</th>
<th>Original Impl.</th>
</tr>
</thead>
<tbody>
<tr>
<td>CiteSeer [5, 18, 43]</td>
<td>wk</td>
<td>27.3ms</td>
<td>34.5ms</td>
<td></td>
</tr>
<tr>
<td></td>
<td>mc</td>
<td>161ms</td>
<td>230ms</td>
<td></td>
</tr>
<tr>
<td></td>
<td>pt</td>
<td>0.9s</td>
<td>1.9s</td>
<td></td>
</tr>
<tr>
<td></td>
<td>lj</td>
<td>9.0s</td>
<td>13.4s</td>
<td></td>
</tr>
<tr>
<td>EmailEuCore [28, 54]</td>
<td>wk</td>
<td>7.1K</td>
<td>100.8K</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ee</td>
<td>1.0K</td>
<td>16.1K</td>
<td></td>
</tr>
<tr>
<td>WikiVote [26]</td>
<td></td>
<td>7.1K</td>
<td>100.8K</td>
<td></td>
</tr>
<tr>
<td></td>
<td>wk</td>
<td>1.0K</td>
<td>16.1K</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>64GB</td>
<td>100.8K</td>
<td></td>
</tr>
<tr>
<td>MiCo [16]</td>
<td>mc</td>
<td>96.6K</td>
<td>1.1M</td>
<td></td>
</tr>
<tr>
<td>Patents [27]</td>
<td>pt</td>
<td>3.8M</td>
<td>16.5M</td>
<td></td>
</tr>
<tr>
<td>Labeled-Patents [27]</td>
<td>lpt</td>
<td>2.7M</td>
<td>14.0M</td>
<td></td>
</tr>
<tr>
<td>LiveJournal [4, 30]</td>
<td>lj</td>
<td>4.8M</td>
<td>42.9M</td>
<td></td>
</tr>
<tr>
<td>Friendster [53]</td>
<td></td>
<td>65.6M</td>
<td>1.8B</td>
<td></td>
</tr>
<tr>
<td>RMAT-100M [10]</td>
<td></td>
<td>100M</td>
<td>1.6B</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>N/A</td>
<td></td>
</tr>
</tbody>
</table>

Graph datasets
## Comparing with AutoMine, RStream, and Arabesque

<table>
<thead>
<tr>
<th>App.</th>
<th>G</th>
<th>DwarvesGraph</th>
<th>AutoMineInHouse</th>
<th>RStream</th>
<th>Arabesque</th>
</tr>
</thead>
<tbody>
<tr>
<td>3-MC</td>
<td>cs</td>
<td>0.16ms</td>
<td>0.16ms (1.0x)</td>
<td>142ms (888x)</td>
<td>10.1s (63,125x)</td>
</tr>
<tr>
<td></td>
<td>ee</td>
<td>0.8ms</td>
<td>7.3ms (8.9x)</td>
<td>21.0s (25,471x)</td>
<td>10.2s (12,352x)</td>
</tr>
<tr>
<td></td>
<td>wk</td>
<td>7.6ms</td>
<td>27.3ms (3.6x)</td>
<td>17.9m (141,437x)</td>
<td>12.1s (1,586x)</td>
</tr>
<tr>
<td></td>
<td>pt</td>
<td>335.7ms</td>
<td>931ms (2.8x)</td>
<td>104.1m (18,611x)</td>
<td>96.4s (287x)</td>
</tr>
<tr>
<td></td>
<td>mc</td>
<td>48.0ms</td>
<td>161ms (3.4x)</td>
<td>144.8m (181,051x)</td>
<td>21.1s (440x)</td>
</tr>
<tr>
<td></td>
<td>lj</td>
<td>2.8s</td>
<td>9.0s (3.3x)</td>
<td>T</td>
<td>24.3m (529x)</td>
</tr>
<tr>
<td>4-MC</td>
<td>cs</td>
<td>0.2ms</td>
<td>4.8ms (23x)</td>
<td>3.7s (17,647x)</td>
<td>9.9s (46,794x)</td>
</tr>
<tr>
<td></td>
<td>ee</td>
<td>9.4ms</td>
<td>920ms (98x)</td>
<td>132.4m (842,186x)</td>
<td>19.1s (2,023x)</td>
</tr>
<tr>
<td></td>
<td>wk</td>
<td>60.0ms</td>
<td>7.0s (117x)</td>
<td>T</td>
<td>402.2s (6,704x)</td>
</tr>
<tr>
<td></td>
<td>pt</td>
<td>1.5s</td>
<td>24.3s (16x)</td>
<td>T</td>
<td>68.3m (2,711x)</td>
</tr>
<tr>
<td></td>
<td>mc</td>
<td>1.3s</td>
<td>31.7s (24x)</td>
<td>T</td>
<td>42.8m (1,942x)</td>
</tr>
<tr>
<td></td>
<td>lj</td>
<td>32.8s</td>
<td>456.5m (836x)</td>
<td>T</td>
<td>C</td>
</tr>
<tr>
<td>5-MC</td>
<td>cs</td>
<td>1.4ms</td>
<td>332ms (229x)</td>
<td>146.4s (101,124x)</td>
<td>11.4s (7,843x)</td>
</tr>
<tr>
<td></td>
<td>ee</td>
<td>360.3ms</td>
<td>104.8s (291x)</td>
<td>T</td>
<td>19.4m (3,233x)</td>
</tr>
<tr>
<td></td>
<td>wk</td>
<td>5.3s</td>
<td>72.4m (823x)</td>
<td>T</td>
<td>C</td>
</tr>
<tr>
<td></td>
<td>pt</td>
<td>32.6s</td>
<td>53.9m (99x)</td>
<td>T</td>
<td>C</td>
</tr>
<tr>
<td></td>
<td>mc</td>
<td>114.7s</td>
<td>174.6m (91x)</td>
<td>T</td>
<td>C</td>
</tr>
<tr>
<td></td>
<td>lj</td>
<td>167.7m</td>
<td>T</td>
<td>T</td>
<td>C</td>
</tr>
<tr>
<td>6-MC</td>
<td>cs</td>
<td>247.0ms</td>
<td>35.9s (145x)</td>
<td>108.7m (26,403x)</td>
<td>48.7s (197x)</td>
</tr>
<tr>
<td></td>
<td>ee</td>
<td>91.3s</td>
<td>259.0m (170x)</td>
<td>T</td>
<td>C</td>
</tr>
<tr>
<td></td>
<td>wk</td>
<td>38.7m</td>
<td>T</td>
<td>T</td>
<td>C</td>
</tr>
<tr>
<td></td>
<td>pt</td>
<td>57.9m</td>
<td>T</td>
<td>T</td>
<td>C</td>
</tr>
<tr>
<td>7-PC</td>
<td>cs</td>
<td>0.3ms</td>
<td>0.5ms (1.7x)</td>
<td>T</td>
<td>C</td>
</tr>
<tr>
<td></td>
<td>ee</td>
<td>719ms</td>
<td>67.1s (93x)</td>
<td>T</td>
<td>C</td>
</tr>
<tr>
<td></td>
<td>wk</td>
<td>735ms</td>
<td>90.8s (24x)</td>
<td>T</td>
<td>C</td>
</tr>
<tr>
<td></td>
<td>pt</td>
<td>499ms</td>
<td>15.7s (31x)</td>
<td>T</td>
<td>C</td>
</tr>
<tr>
<td>8-PC</td>
<td>cs</td>
<td>3.3K</td>
<td>3.8K</td>
<td>6</td>
<td>10.3s (51,315x)</td>
</tr>
<tr>
<td></td>
<td>ee</td>
<td>1.0K</td>
<td>16.1K</td>
<td>9.6s (48,235x)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>wk</td>
<td>7.1K</td>
<td>100.8K</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td></td>
<td>pt</td>
<td>96.6K</td>
<td>1.1M</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td></td>
<td>mc</td>
<td>3.8M</td>
<td>16.5M</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td></td>
<td>lj</td>
<td>2.7M</td>
<td>14.0M</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td></td>
<td>fr</td>
<td>4.8M</td>
<td>42.9M</td>
<td>N/A</td>
<td></td>
</tr>
<tr>
<td></td>
<td>rmat</td>
<td>65.6M</td>
<td>1.8B</td>
<td>N/A</td>
<td></td>
</tr>
</tbody>
</table>

### Graph datasets

- **CiteSeer [5, 18, 43]**
  - cs: 3.3K, 4.5K, 6
  - ee: 1.0K, 16.1K, 42
  - wk: 7.1K, 100.8K, N/A

- **EmailEuCore [28, 54]**
  - mc: 96.6K, 1.1M, 29
  - pt: 3.8M, 16.5M, N/A

- **WikiVote [26]**
  - cc: 2.7M, N/A

- **MiCo [16]**
  - pm: 2.7M, 14.0M, 37

- **Patents [27]**
  - lpt: 2.7M, 14.0M, 37

- **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

- **FEM-300**
  - cs: 2.0ms, 0.3ms (1.5x), 522ms (2,609x)
  - ee: 0.2ms, 0.2ms (1.0x), 3.6s (18,090x)
  - wk: 20.8s, 20.3s (0.98x), 4,713.5s (226x)
  - pt: 308ms, 441ms (1.4x), 149.1m (29,013x)

- **FIS-3K**
  - cs: 0.6s, 0.6ms (1.0x), 77.9ms (130x)
  - ee: 0.2ms, 0.2ms (1.0x), 210ms (1,049x)
  - wk: 18.6s, 18.1s (0.98x), 89.0m (287x)
  - pt: 124ms, 300ms (2.4x), 141.9m (68,813x)

**High Performance Graph Mining Systems**
Comparing with Peregrine, Pangolin, and Fractal

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

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

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

Graph datasets
## Decomposition Space Search Methods

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

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

- Circulant tuning is slower than separate tuning.
- 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

\[ p_0 - p_{19} \text{ are all size-5 patterns except for 5-clique} \]
Large Graphs and Large Patterns

<table>
<thead>
<tr>
<th>Graph</th>
<th>#Vertices</th>
<th>#Edges</th>
<th>App.</th>
<th>Runtime (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>fr</td>
<td>65.6M</td>
<td>1.8B</td>
<td>4-Motif</td>
<td>4,301</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>4-Chain</td>
<td>862</td>
</tr>
<tr>
<td>rmat</td>
<td>100M</td>
<td>1.6B</td>
<td>4-Motif</td>
<td>5,900</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>4-Chain</td>
<td>800</td>
</tr>
</tbody>
</table>

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-hop 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

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

```python
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**

```python
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))
```

High Performance Graph Mining Systems
Inter-loop Dependent Vertex Function

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

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

def process_v_1(data_dependency_1, v_0, 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_0, N(v_2)):
    S0 = data_dependency_2["s_0"]
    S1 = intersect(S0, N(v_2))
    return S1

data_dependency_0 = SOME_INITIALIZER
for v_0 in V(G):
    v_1_candidates = process_v_0(data_dependency_0, v_0, N(v_0))
    for v_1 in v_1_candidates:
        v_2_candidates = 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_dependency_n-1, v_n-1, N(v_n-1))
```

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 Dependent Vertex Function

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

```cpp
Aggregator<uint64_t> triangle_cnt_agg;

class ProcessSecondTriangleVertex: public VertexFunction {
    Set<VertexFunction*> process_vertex(VertxId v, VertexSet neighbors, Objects shared_objs, Buffer workspace) {
        VertexSet * v_0_nbrs = shared_objs->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;
}

class ProcessFirstTriangleVertex: public VertexFunction {
    Set<VertexFunction*> process_vertex(VertxId v, VertexSet neighbors, Objects shared_objs, Buffer workspace) {
        // specify the objects to be shared with new vertex-functions
        Objects ohs_to_share;
        ohs_to_share.put("v_0_nbrs", &neighbours);
        // for v_1 in N(v_0):
        for (VertxId 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(ohs_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(VertxId));
        } add(f);
    }
    return S;
}
```
Abstract Execution Model

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

1: \( F_v \leftarrow \) a set of user-provided initial vertex-functions
2: while \( F_v \) is not empty do
3:   pick up a vertex-function \( f_v \) from \( F_v \)
4:   remove \( f_v \) from \( F_v \)
5:   \( \text{New}_v \leftarrow \) execute(\( f_v \))
6:   for all \( f_v \in \text{New}_v \) do
7:     add \( f_v \) to \( F_v \)
8: end for
9: end while
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

- **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

1. Trigger new vertex-function
2. Vertex with X trigger functions
3. Cached vertex (degree >= threshold)

... in total 100 neighbors

- Request the Graph Data of a Vertex $v$
- Found in the Cache?
  - YES: Return the Cached Graph Data of $v$
  - NO: Fetch the Graph Data of $v$ through the Network

- If deg($v$) >= threshold, cache the Graph Data of $v$
- Is Cache Full?

High Performance Graph Mining Systems
Communication Merging

- Efficient hash-table based implementation that allows false negative in merging
● 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)
Comparing with G-Thinker: Partitioned Graph Triangle Counting

<table>
<thead>
<tr>
<th>Graph</th>
<th>Khuzdul</th>
<th>G-thinker</th>
</tr>
</thead>
<tbody>
<tr>
<td>wk</td>
<td>24.4ms</td>
<td>1.0s (41.0x)</td>
</tr>
<tr>
<td>mc</td>
<td>40.0ms</td>
<td>2.1s (52.5x)</td>
</tr>
<tr>
<td>pt</td>
<td>254.9ms</td>
<td>285.3s (1,119.3x)</td>
</tr>
<tr>
<td>lj</td>
<td>826.8ms</td>
<td>31.6s (38.2x)</td>
</tr>
<tr>
<td>uk</td>
<td>690.1s</td>
<td>CRASHED</td>
</tr>
<tr>
<td>tw</td>
<td>2171.2s</td>
<td>CRASHED</td>
</tr>
<tr>
<td>fr</td>
<td>81.2s</td>
<td>CRASHED</td>
</tr>
</tbody>
</table>
## Comparing with GraphPi: Replicated Graph

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

<table>
<thead>
<tr>
<th>Application</th>
<th>Graph</th>
<th>Khuzdul (8-node)</th>
<th>GraphPi (8-node)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4-clique</td>
<td>wk</td>
<td>43.0ms</td>
<td>500.4ms</td>
</tr>
<tr>
<td></td>
<td>mc</td>
<td>321.5ms</td>
<td>844.0ms</td>
</tr>
<tr>
<td></td>
<td>pt</td>
<td>412.7ms</td>
<td>6.7s</td>
</tr>
<tr>
<td></td>
<td>lj</td>
<td>5.3s</td>
<td>12.8s</td>
</tr>
<tr>
<td></td>
<td>uk</td>
<td>17,241.0s</td>
<td>31,008.6s</td>
</tr>
<tr>
<td></td>
<td>tw</td>
<td>18,817.0s</td>
<td>TIMEOUT</td>
</tr>
<tr>
<td></td>
<td>fr</td>
<td>190.5s</td>
<td>177.8s</td>
</tr>
<tr>
<td>5-clique</td>
<td>wk</td>
<td>78.1ms</td>
<td>522.1ms</td>
</tr>
<tr>
<td></td>
<td>mc</td>
<td>11.1s</td>
<td>8.2s</td>
</tr>
<tr>
<td></td>
<td>pt</td>
<td>822.5ms</td>
<td>6.8s</td>
</tr>
<tr>
<td></td>
<td>lj</td>
<td>188.6s</td>
<td>174.7s</td>
</tr>
<tr>
<td></td>
<td>Fr</td>
<td>220.3s</td>
<td>260.0s</td>
</tr>
</tbody>
</table>
# Comparing with Single-Machine Systems

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

<table>
<thead>
<tr>
<th>Application</th>
<th>Graph</th>
<th>Khuzdul (8-node / single-node)</th>
<th>Automine</th>
<th>Peregrine</th>
<th>Pangolin</th>
</tr>
</thead>
<tbody>
<tr>
<td>3-motif</td>
<td>Wk</td>
<td>24.8ms / 39.8ms</td>
<td>27.4ms</td>
<td>26.9ms</td>
<td>45ms</td>
</tr>
<tr>
<td>mc</td>
<td>52.8ms / 255.2ms</td>
<td>160.3ms</td>
<td>84.7ms</td>
<td>288ms</td>
<td></td>
</tr>
<tr>
<td>pt</td>
<td>429.3ms / 2.8s</td>
<td>930.9ms</td>
<td>1.7s</td>
<td>1.5s</td>
<td></td>
</tr>
<tr>
<td>lj</td>
<td>1.9s / 12.8s</td>
<td>8.9s</td>
<td>4.6s</td>
<td>29.2s</td>
<td></td>
</tr>
<tr>
<td>uk</td>
<td>3,005.1s / 24700.190s</td>
<td>TIMEOUT (&gt;10 hours)</td>
<td>4660.9s</td>
<td>TIMEOUT</td>
<td></td>
</tr>
<tr>
<td>tw</td>
<td>9401.0s / TIMEOUT</td>
<td>TIMEOUT</td>
<td>20477.6s</td>
<td>TIMEOUT</td>
<td></td>
</tr>
<tr>
<td>fr</td>
<td>190.5s / 1420.8s</td>
<td>1206.8s</td>
<td>316.1s</td>
<td>6305.9s</td>
<td></td>
</tr>
</tbody>
</table>
## Comparing with Single-Machine Systems

<table>
<thead>
<tr>
<th>Application</th>
<th>Graph</th>
<th>Khuzdul (8-node / single-node)</th>
<th>Automine</th>
<th>Peregrine</th>
<th>Pangolin</th>
</tr>
</thead>
<tbody>
<tr>
<td>4-clique</td>
<td>wk</td>
<td>43.0ms / 77.0ms</td>
<td>47.2ms</td>
<td>104.6ms</td>
<td>47ms</td>
</tr>
<tr>
<td></td>
<td>mc</td>
<td>321.5ms / 1.9s</td>
<td>1.2s</td>
<td>1.8s</td>
<td>2.8s</td>
</tr>
<tr>
<td></td>
<td>pt</td>
<td>412.7ms / 2.6s</td>
<td>381.0ms</td>
<td>1.3s</td>
<td>773ms</td>
</tr>
<tr>
<td></td>
<td>lj</td>
<td>5.3s / 37.6s</td>
<td>31.3s</td>
<td>49.6s</td>
<td>54.7s</td>
</tr>
<tr>
<td></td>
<td>uk</td>
<td>17,241.0s / TIMEOUT</td>
<td>TIMEOUT</td>
<td>TIMEOUT</td>
<td>MEM</td>
</tr>
<tr>
<td></td>
<td>tw</td>
<td>18,817.0s / TIMEOUT</td>
<td>TIMEOUT</td>
<td>TIMEOUT</td>
<td>MEM</td>
</tr>
<tr>
<td></td>
<td>Fr</td>
<td>157.9s / 887.6s</td>
<td>570.3s</td>
<td>1237.5s</td>
<td>MEM</td>
</tr>
</tbody>
</table>
## Comparing with Single-Machine Systems

<table>
<thead>
<tr>
<th>Application</th>
<th>Graph</th>
<th>Khuzdul (8-node / single-node)</th>
<th>Automine</th>
<th>Peregrine</th>
<th>Pangolin</th>
</tr>
</thead>
<tbody>
<tr>
<td>5-clique</td>
<td>Wk</td>
<td>78.1ms / 226.3ms</td>
<td>124.7ms</td>
<td>477.2ms</td>
<td>146ms</td>
</tr>
<tr>
<td></td>
<td>mc</td>
<td>11.1s / 71.2s</td>
<td>46.8s</td>
<td>78.0s</td>
<td>132.0s</td>
</tr>
<tr>
<td></td>
<td>pt</td>
<td>822.5ms / 5.3s</td>
<td>408.4ms</td>
<td>1.5s</td>
<td>967ms</td>
</tr>
<tr>
<td></td>
<td>lj</td>
<td>188.6s / 1385.8s</td>
<td>982.9s</td>
<td>2076.6s</td>
<td>MEM</td>
</tr>
<tr>
<td></td>
<td>Fr</td>
<td>220.3s / 1593.9s</td>
<td>900.2s</td>
<td>3032.8s</td>
<td>MEM</td>
</tr>
</tbody>
</table>
## Communication/Computation Overlapping

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

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

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

<table>
<thead>
<tr>
<th>Application</th>
<th>Graph</th>
<th>Runtime with sub-partitioning</th>
<th>Runtime without sub-partitioning</th>
</tr>
</thead>
<tbody>
<tr>
<td>3-motif</td>
<td>mc</td>
<td>52.8ms</td>
<td>74.5ms</td>
</tr>
<tr>
<td></td>
<td>pt</td>
<td>429.3ms</td>
<td>970.5ms</td>
</tr>
<tr>
<td></td>
<td>lj</td>
<td>1.86s</td>
<td>3.1s</td>
</tr>
<tr>
<td></td>
<td>Fr</td>
<td>190.5s</td>
<td>327.2s</td>
</tr>
<tr>
<td>5-clique</td>
<td>mc</td>
<td>11.1s</td>
<td>16.1s</td>
</tr>
<tr>
<td></td>
<td>pt</td>
<td>822.5ms</td>
<td>2.0s</td>
</tr>
<tr>
<td></td>
<td>lj</td>
<td>188.6s</td>
<td>319.8s</td>
</tr>
<tr>
<td></td>
<td>Fr</td>
<td>220.3s</td>
<td>447.5s</td>
</tr>
</tbody>
</table>
## Scalability: LiveJournal Graph

<table>
<thead>
<tr>
<th>#Nodes</th>
<th>Triangle</th>
<th>3-Motif</th>
<th>4-Clique</th>
<th>5-Clique</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5.5s</td>
<td>12.8s</td>
<td>37.6s</td>
<td>1385.8s</td>
</tr>
<tr>
<td>2</td>
<td>2.8s</td>
<td>6.5s</td>
<td>19.1s</td>
<td>704.7s</td>
</tr>
<tr>
<td>4</td>
<td>1.5s</td>
<td>3.4s</td>
<td>9.9s</td>
<td>361.4s</td>
</tr>
<tr>
<td>8</td>
<td>826.8ms</td>
<td>1.9s</td>
<td>5.3s</td>
<td>188.6s</td>
</tr>
<tr>
<td>8-node speedup over 1-node</td>
<td>6.65x</td>
<td>6.74x</td>
<td>7.09x</td>
<td>7.35x</td>
</tr>
</tbody>
</table>
## Scalability: Friendster Graph

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

Graph Mining Compiler

Algorithm Searching (similar to Automine)

Pattern Matching Process (nested loops)

User-Specified Patterns

Pattern Matching Process Expressed by Vertex-Functions

Distributed Runtime Engine

Execution

Graph Mining Results (e.g., Pattern Counts)
High Performance Graph Mining Systems

Xuehai Qian
University of Southern California