Tutorial 3: Systems and Algorithms for Massively Parallel Graph Mining

Da Yan  yanda@uab.edu
Guimu Guo  guimuguo@uab.edu

Department of Computer Science
The University of Alabama at Birmingham (UAB)
Overview

- Graph Mining Problems: A Categorization
- Think Like a Vertex: Data-Intensive Mining
- Why TLAV is not Sufficient? Compute-Intensive!
- PrefixFPM for Frequent Subgraph Mining
- G-thinker: Think Like a Subgraph
- TLAS Algorithms for Subgraph Mining
Overview

- Graph Mining Problems: A Categorization
- Think Like a Vertex: Data-Intensive Mining
- Why TLAV is not Sufficient? Compute-Intensive!
- PrefixFPM for Frequent Subgraph Mining
- G-thinker: Think Like a Subgraph
- TLAS Algorithms for Subgraph Mining
Overview

• **Graph Mining Problems: A Categorization**

• Think Like a Vertex: Data-Intensive Mining

• Why TLAV is not Sufficient? Compute-Intensive!

• PrefixFPM for Frequent Subgraph Mining

• G-thinker: Think Like a Subgraph

• TLAS Algorithms for Subgraph Mining
Problem Categorization

Unlike relational database, operations on graph are highly heterogeneous.

No one-size-fits-all solution!

- There have been attempts to unify many graph heterogeneous algorithms into one framework.
- Simple programming, but performance hurts!

Frank McSherry, Michael Isard, Derek Gordon Murray: *Scalability! But at what COST?* HotOS 2015
Problem Categorization

Time-Complexity Perspective

• Some are data-intensive (low complexity)
  - Peeling algorithm of $k$-core: $O(n)$
  - Random walks (e.g., PageRank)
• Some are compute-intensive!
  - Dense subgraph structure mining: quasi-cliques, $k$-plexes
  - Frequent (and closed) subgraph pattern mining

NP-hard!
Problem Categorization

A Big Data Framework Should

• Be easy to program, simple model and simple API
• Cover a sufficient range of problems for system reuse

But… do not attempt to cover everything!

Out Solution: Divide and Conquer
Problem Categorization

Our Approach

• Divide the graph mining problem space
• Develop a dedicated framework for each problem category
• Ensure both program simplicity and execution efficiency
Problem Categorization

Category I: Iterative Algorithms

- Low time complexity: $O(|E|)$ per iteration (message passing between vertices), $O(polylog(|G|))$ iterations

Da Yan, James Cheng, Kai Xing, Yi Lu, Wilfred Ng, Yingyi Bu:

Lu Qin, Jeffrey Xu Yu, Lijun Chang, Hong Cheng, Chengqi Zhang, Xuemin Lin:
Scalable big graph processing in MapReduce. SIGMOD Conference 2014: 827-838

Wenfei Fan, Floris Geerts, Frank Neven:

Yufei Tao, Wenqing Lin, Xiaokui Xiao:
Minimal MapReduce algorithms. SIGMOD Conference 2013: 529-540
Problem Categorization

Category I: Iterative Algorithms

- Examples: PageRanks (random walks), loopy belief propagation, collaborative filtering (ALS)

Mostly node and/or edge scoring
Problem Categorization

Category II: Patterns from Transaction DB

- Examples: (closed, maximal) frequent subgraph pattern mining

NP-hard!
Problem Categorization

Category III: Patterns from a Big Graph

- Examples: dense subgraph mining (triangle, $k$-truss, clique, $k$-plex, $\gamma$-quasi-clique), graph matching (not mining?)

Often NP-hard!
Overview

- Graph Mining Problems: A Categorization
- **Think Like a Vertex: Data-Intensive Mining**
- Why TLAV is not Sufficient? Compute-Intensive!
- PrefixFPM for Frequent Subgraph Mining
- G-thinker: Think Like a Subgraph
- TLAS Algorithms for Subgraph Mining
Category I: Iterative Algorithms

Programming Model
- Think like a vertex (TLAV), or its variants
- Information passed along edges
- Vertex states get updated till converge
- Output is usually a value for each vertex
Category I: Iterative Algorithms

TLAV Model

\[ v.v compute(messages) \]

```
public void compute(Iterator msgs) {
    double sum = 0, value;

    while (msgs.hasNext())
        sum += msgs.next();
    value = 0.15 / numVertices + 0.85 * sum;
    setValue(value);

    double error = getGlobalError();
    if (error > epsilon)
        sendMessageToNeighbors(value / numEdges);
    else
        voteToHalt();
}
```
Motivation for Parallel/Distributed Mining

• Low time complexity: CPU is not the bottleneck
• Example: Connected Components is $O(|V| + |E|)$

Abstract

We study a class of simple algorithms for concurrently computing the connected components of an $n$-vertex, $m$-edge graph. Our algorithms are easy to implement in either the COMBINING CRCW PRAM or the MPC computing model. For two related algorithms in this class, we obtain $\Theta(lg n)$ step and $\Theta(m lg n)$ work bounds. For two others, we obtain $O(lg^2 n)$ step and $O(m lg^2 n)$ work bounds, which are tight for one of them. All our algorithms are simpler than related algorithms in the literature. We also point out some gaps and errors in the analysis of previous algorithms. Our results show that even a basic problem like connected components still has secrets to reveal.

Sixue Liu, Robert E. Tarjan:
Simple Concurrent Labeling Algorithms for Connected Components. SOSA@SODA 2019: 3:1-3:20
Category I: Iterative Algorithms

Motivation for TLAV Systems

• Why bother parallel/distributed computing if the serial complexity is $O(|V|)$?
• Answer: input graph is huge
  - Parallel graph loading from HDFS
  - Part of a dataflow in Hadoop Ecosystem
  - One machine does not have enough RAM space
  - Fast to code in TLAV API
Category I: Iterative Algorithms

Existing TLAV Systems

• Numerous papers, but many only works for special applications where new vertex state is an aggregate of incoming messages (e.g., no pointer jumping)

Our focus in this tutorial
Category I: Iterative Algorithms

Design Variants in Pregel-Like Systems

API: GAS (Gather, Apply and Scatter)

Vertex-Programs directly read the neighbors state

```
GraphLab_PageRank(i)
    // Compute sum over neighbors
    total = 0
    foreach (j in in_neighbors(i)):
        total = total + R[j] * w_{ji}

    // Update the PageRank
    R[i] = 0.15 + total

    // Trigger neighbors to run again
    if R[i] not converged then
        foreach (j in out_neighbors(i)):
            signal vertex-program on j
```

GraphLab

as an example

Vertex-cut
Google

TLAV Pioneer

» Message passing
» Iterative

• Each iteration is called a superstep

2020 Test of Time Award

Pregel: A System for Large-Scale Graph Processing

Grzegorz Malewicz, Matthew H. Austern, Aart J.C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski

Not sure how much impact inside Google…

But inspires many TLAV works in academia.
Google’s Pregel

Vertex Partitioning

Graph representation and vertex partitions:

- **Graph**: Nodes 0 to 8 connected with edges.
- **Vertex Partitioning**:
  - **$M_0$**: Nodes 0, 3, 6 with values 1, 3, 5, 8.
  - **$M_1$**: Nodes 1, 4, 7 with values 0, 2, 3, 4, 8.
  - **$M_2$**: Nodes 2, 5, 8 with values 1, 3, 4, 7, 6.
Google’s Pregel

Programming Interfaces

» `u.compute(msgs)`
» `u.send_msg(v, msg)`
» `get_superstep_number()`
» `u.vote_to_halt()`

Called inside `u.compute(msgs)`
Google’s Pregel

Vertex state
  » Active / inactive
  » Reactivated by messages

Stop condition
  » All vertices are halted, and
  » No pending messages for the next superstep
Google’s Pregel

Hash-Min: Connected Components

\( v.\text{compute}(messages) \)
Google’s Pregel

Hash-Min: Connected Components

\( v.\text{compute}(messages) \)

If \( \text{min} < \text{id} \):

Finally, vote to halt
Google’s Pregel

Hash-Min: Connected Components

Superstep 1
Google’s Pregel

Hash-Min: Connected Components

Superstep 2
Google’s Pregel

Hash-Min: Connected Components

Superstep 3
Google’s Pregel

Computing $k$-Core

$k = 3$

$v.vcompute(messages)$

Delete received neighbors
If $v.d\text{egree} < k$
delete $v$
send “v” to neighbors
Vote to halt
Google’s Pregel

Computing $k$-Core

$v.\text{compute}(messages)$

$k = 3$

Delete received neighbors
If $v.\text{degree} < k$
  delete $v$
  send “$v$” to neighbors
Vote to halt

$k = 3$
Google’s Pregel

Computing $k$-Core

$v$.

\begin{itemize}
\item $k = 3$
\item Delete received neighbors
\item If $v$.degree < $k$
\item delete $v$
\item send “$v$” to neighbors
\item Vote to halt
\end{itemize}
Google’s Pregel

Core Decomposition

Computing core number of every vertex
Initializing core numbers with vertex degrees
Google’s Pregel

Core Decomposition

h-index operator: list adjacent nodes from large to small based on the core value until support = core number
Each vertex iteratively call the h-index operator till convergence

Google’s Pregel

Core Decomposition

h-index operator: list adjacent nodes from large to small based on the core value until support = core number
Maiter

DAIC

- Unlike GraphLab, DAIC has exactness guarantee

3.2 Asynchronous DAIC

DAIC can be performed asynchronously. That is, a vertex can start update at any time based on whatever it has already received. We can describe asynchronous DAIC as follows, each vertex $j$ performs:

\begin{align}
\text{receive:} \quad \{ & \text{Whenever receiving } m_j, \\
& \Delta \hat{v}_j \leftarrow \Delta \hat{v}_j \oplus m_j; \\
& \hat{v}_j \leftarrow \hat{v}_j \oplus \Delta \hat{v}_j; \}
\end{align}

\begin{align}
\text{update:} \quad \{ & \text{For any } h, \text{ if } g_{\{j,h\}}(\Delta \hat{v}_j) \neq 0, \\
& \text{send value } g_{\{j,h\}}(\Delta \hat{v}_j) \text{ to } h; \\
& \Delta \hat{v}_j \leftarrow 0, \}
\end{align}
Maiter

DAIC

• Unlike GraphLab, DAIC has exactness guarantee

<table>
<thead>
<tr>
<th>algorithm</th>
<th>$g_{i,j}(x)$</th>
<th>$\oplus$</th>
<th>$v^0_j$</th>
<th>$\Delta v^1_j$</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSSP</td>
<td>$x + A(i,j)$</td>
<td>min</td>
<td>$\infty$</td>
<td>$0 (j = s)$ or $\infty (j \neq s)$</td>
</tr>
<tr>
<td>Connected Components</td>
<td>$A(i,j) \cdot x$</td>
<td>max</td>
<td>$-1$</td>
<td>$\frac{1}{j}$</td>
</tr>
<tr>
<td>PageRank</td>
<td>$d \cdot A(i,j) \cdot \frac{x}{</td>
<td>N(j)</td>
<td>}$</td>
<td>+</td>
</tr>
<tr>
<td>Adsorption</td>
<td>$p^\text{cont}_i \cdot A(i,j) \cdot x$</td>
<td>+</td>
<td>$0$</td>
<td>$p^\text{inj}_j \cdot I_j$</td>
</tr>
<tr>
<td>HITS (authority)</td>
<td>$d \cdot A(i,j) \cdot x$</td>
<td>+</td>
<td>$0$</td>
<td>$1$ (if $j = s$) or $0$ (if $j \neq s$)</td>
</tr>
<tr>
<td>Katz metric</td>
<td>$\beta \cdot A(i,j) \cdot x$</td>
<td>+</td>
<td>$0$</td>
<td>$1$ (if $j = s$) or $0$ (if $j \neq s$)</td>
</tr>
<tr>
<td>Jacobi method</td>
<td>$-\frac{A_{jj}}{A_{ij}} \cdot x$</td>
<td>+</td>
<td>$0$</td>
<td>$\frac{b_j}{A_{jj}}$</td>
</tr>
<tr>
<td>SimRank</td>
<td>$\frac{C \cdot A(i,j)}{</td>
<td>I(a)</td>
<td></td>
<td>I(b)</td>
</tr>
<tr>
<td>Rooted PageRank</td>
<td>$A(j,i) \cdot x$</td>
<td>+</td>
<td>$0$</td>
<td>$0 (j = s)$ or $0 (j \neq s)$</td>
</tr>
</tbody>
</table>
Chronicle of BD Systems

Each round of shuffling means data go through HDFS once

MapReduce papers explode !!!
(MapReduce Hype)

2009
Pregel born

2010
Incubation period ...

2012
GraphChi (OSDI’12)

2013
Giraph born

Putting everything in (distributed) memory (RDD), add more operations than MapReduce

Where am I ?

Spark (NSDI’12)
Chronicle of BD Systems

2009: 搬砖使我快乐

2013: Querying and Indexing Geo-Spatial Data

2014: Data Uncertainty

2016: Do Pregel!

Ph.D. HKUST

Postdoc @CUHK

HK 2015 Young Scientist Award
Pregel Research Back in 2013

Difficulty – Productivity – Competition
Pregel Research Now
Many opportunities already explored, tools are useful
Difficulty – Productivity – Competition
Deep Learning Now

Just for Fun…

Difficulty

Productivity

Competition
Abstract:

Database and Artificial Intelligence (AI) can benefit from each other. More intelligent databases can benefit more intelligent AI models (AI4DB). For example, traditional empirical database estimation, join order selection, knob tuning, index and view selection, and storage are all optimized for large-scale database instances, various applications, and the cloud. Fortunately, learning-based techniques can alleviate these bottlenecks, and techniques can optimize AI models (DB4AI). For example, AI is smart, but it requires developers to write complex code and train complicated models. These tasks can be used to reduce the complexity of using AI models, accelerate AI application, and provide better services.
Our Works

**Pregel+**: techniques to reduce message numbers (PVLDB’14, WWW’15, ICDE’18, TCBB’19)

**Blogel**: graph partitioning + think like a partition (PVLDB’14)

**Quegel**: online query processing with interactive speed (PVLDB’16)

**GraphD**: distributed out-of-core execution (TPDS’18)

**LWFT**: faster checkpointing and recovery (ICPP’19)
Our Works

Vertex Mirroring
Req-Resp API
LWCP
Blogel
Quegel
GraphD
Pregel+

Hadoop Distributed File System

http://www.cse.cuhk.edu.hk/systems/graph/
Our Works

- **Blogel: A block-centric framework for distributed computation on real-world graphs**
  D Yan, J Cheng, Y Lu, W Ng
  Proceedings of the VLDB Endowment 7 (14), 1981-1992

- **Large-scale distributed graph computing systems: An experimental evaluation**
  Y Lu, J Cheng, D Yan, H Wu
  Proceedings of the VLDB Endowment 8 (3), 281-292

- **Effective techniques for message reduction and load balancing in distributed graph computation**
  D Yan, J Cheng, Y Lu, W Ng
  Proceedings of the 24th International Conference on World Wide Web, 1307-1317

- **Pregel algorithms for graph connectivity problems with performance guarantees**
  D Yan, J Cheng, K Xing, Y Lu, W Ng, Y Bu
  Proceedings of the VLDB Endowment 7 (14), 1821-1832

- **Big graph analytics platforms**
  D Yan, Y Bu, Y Tian, A Deshpande
  Foundations and Trends in Databases 7 (1-2), 1-195
Books

Big Graph Analytics Platforms
Da Yan, Yingyi Bu, Yuanyuan Tian
and Amol Deshpande

Systems for Big Graph Analytics
Da Yan
Yuanyuan Tian
James Cheng

SPRINGER BRIEFS IN COMPUTER SCIENCE

Springer
Theoretical Complexity Results of Graph Algorithms in PREGEL

**Balanced Practical PREGEL Algorithms (BPPA)**
- Linear Space Usage: $O(d(v))$
- Linear Computation Cost: $O(d(v))$
- Linear Communication Cost: $O(d(v))$
- (At Most) Logarithmic Number of Rounds: $O(\log n)$ super-steps

**Examples:** Connected components, spanning tree, Euler tour, BFS, Pre-order and Post-order Traversal

Open Area of Research
Practical Pregel Algorithms

ACM/IEEE TCBB journal version also

Application: Genome Assembly [ICDE’18]
Block-Centric Computation

Blogel: Block-Centric Model [PVLDB’14]

» Orders of magnitude performance improvement

• e.g., one hour → 10 seconds
Block-Centric Computation

Blogel: Block-Centric Model [PVLDB’14]

Experimental Analysis of Distributed Graph Systems

Khaled Ammar, M. Tamer Özsu
David R. Cheriton School of Computer Science
University of Waterloo, Waterloo, Ontario, Canada
{khaled.ammar, tamer.ozsu}@uwaterloo.ca

The major findings of our study are the following:

- **Blogel is the overall winner.** The execution time of Blogel-B is shortest, but Blogel-V is faster when we consider the end-to-end processing including data loading and partitioning (§ 5.1).
- Existing graph processing systems are inefficient over graphs with large diameters, such as the road network (§ 5.3, 5.6, 5.8).
- GraphLab performance is sensitive to cluster size (Section 5.4).
- Giraph has a similar performance to GraphLab when both systems use random partitioning (§ 5.5).
Message Reduction

Message Reduction in Pregel+ [WWW’15]

» Two techniques to reduce number of messages transmitted
  • Vertex Mirroring
  • Request-Respond Paradigm

\[
a(u_1) + a(u_2) + a(u_3) + a(u_4)
\]
Online Graph Querying

Quegel: Query-Centric Framework [PVLDB’16]

» A graph query usually has a light workload
  • Only a portion of the whole graph gets accessed
» Existing solutions are unsatisfactory

» Orders of magnitude performance improvement
  • e.g., point-to-point shortest path query on Twitter
  • Giraph: > 100 seconds per query
  • Quegel: 3 queries per second
Out-of-Core Support

GraphD: Out-of-Core Pregel [TPDS’18]
» Desktop PCs + Gigabit Ethernet switch
» Disk streaming bandwidth >> network bandwidth
» Each machine stores edges and messages on local disk
» \( O(|V| / \# \text{machines}) \) memory on each machine
» Same performance as in-memory Pregel
Fault Tolerance

Lightweight Checkpointing & Vertex Logging [ICPP’19]

» Only checkpointing vertex states, not messages
» Edges are checkpointed incrementally
» Generating messages from vertex states during recovery
» Logging vertex states to local disks to avoid recomputation during recovery
Overview

• Graph Mining Problems: A Categorization
• Think Like a Vertex: Data-Intensive Mining
• **Why TLAV is not Sufficient? Compute-Intensive!**
• PrefixFPM for Frequent Subgraph Mining
• G-thinker: Think Like a Subgraph
• TLAS Algorithms for Subgraph Mining
Motivation

Graph Mining Tasks?

• Some are data-intensive / IO-bound
  - Peeling algorithm of *k*-core: $O(n)$
  - Random walks (e.g., PageRank)
• Some are compute-intensive!
  - Dense subgraph structure mining: quasi-cliques, *k*-plexes
  - Frequent (and closed) subgraph pattern mining

To present in next section
Motivation

Data-Intensive v.s. Compute-Intensive
  • Fundamental and familiar to the DB community
  • But people simply get lost (the ‘system hype’)

COST in the land of databases

Several years ago Michael Isard, Derek Murray, and I wrote a paper: "Scalability, but at what COST". The premise of the paper was that the current excitement about distributed computation (e.g. Hadoop, Spark) produced implementations that improve when you give them more resources (they "scale") but whose performance never quite gets to where you would be with a simple single-threaded implementation on a laptop.

https://github.com/frankmcsherry/blog/blob/master/posts/2017-09-23.md
Motivation

Data-Intensive v.s. Compute-Intensive

• Fundamental and familiar to the DB community
• But people simply get lost (the ‘system hype’)

Cont.

COST in the land of databases

One conclusion was that the Systems community (which I think of as SOSP/OSDI+/) was overexcited about big data processing, but not especially good at it yet. Specifically, graph processing, which is one place where interesting algorithms start to make a difference (no more word count). One could imagine the folks who study large-scale data processing professionally, over in the databases community, snickering just a bit.

This post is for you, databases community. <3
Motivation

Data-Intensive v.s. Compute-Intensive

Ethernet

200 %
Motivation

Data-Intensive v.s. Compute-Intensive

Data-intensive
IO-bound

Compute-intensive
CPU-bound

more disks
faster networks
faster disks

more CPUs
more memory
Motivation

Shumo Chu, James Cheng:

cse.cuhk.edu.hk/~jcheng/papers/triangle_tkdd12.pdf

For triangle counting in a large graph that cannot fit in main memory, parallel algorithms that apply the MapReduce framework were proposed recently [Suri and Vassilvitskii 2011]. Their algorithms are exact and do not require to keep the entire input graph in main memory at each individual machine. Our approach is orthogonal to their approach of parallelization for triangle counting. However, we note that the MapReduce framework may not be suitable for the task of triangle counting. As a cross reference, the experiments in [Suri and Vassilvitskii 2011] show that, for the same dataset LJ, the fastest of their parallel algorithms running on 1,636 machines takes 5.33 minutes, while our algorithms running on a single machine use less than 0.5 minute. Note that their algorithm is proven to be work efficient. The much longer running time may be due to the hidden cost needed in the shuffling phase between Mappers and Reducers, because to make triangle counting work in a MapReduce framework, the algorithm has to produce a huge amount of intermediate data. Moreover, many researchers or the average users may not have access to or may not be willing to pay for the computing resource of hundreds to thousands of machines. On the contrary, our algorithms are efficient and require only one ordinary machine.
Motivation

T-thinker: Divide and Conquer

- Task-based: data pulling + computation on top
T-thinker Research Now

Difficulty – Productivity – Competition

We have queued many works to be done!

Just us right now…
Five Years from Now?

Time to make impact!

<table>
<thead>
<tr>
<th>Cited by</th>
<th>All</th>
<th>Since 2020</th>
</tr>
</thead>
<tbody>
<tr>
<td>Citations</td>
<td>+????</td>
<td>+????</td>
</tr>
<tr>
<td>h-index</td>
<td>+??</td>
<td>+??</td>
</tr>
<tr>
<td>i10-index</td>
<td>+??</td>
<td>+??</td>
</tr>
</tbody>
</table>

Competition
Motivation

Programming Simplicity Over-Emphasized

pattern-to-instance

Embedding-Centric: Arabesque (SOSP’15), RStream (OSDI’18)
G-thinker

Task-Centric Programming Interface

» UDF: `spawn_task(vertex)`
» UDF: `compute(task, frontier)`
» `pull(vertex_id)`
» `add_task(task)`

Will be presented by
Overview

- Graph Mining Problems: A Categorization
- Think Like a Vertex: Data-Intensive Mining
- Why TLAV is not Sufficient? Compute-Intensive!
- **PrefixFPM for Frequent Subgraph Mining**
- G-thinker: Think Like a Subgraph
- TLAS Algorithms for Subgraph Mining
Frequent Subgraph Pattern Mining

Discovery of graph structures that occur a significant number of times across a set of graphs

O-H present in $\frac{3}{4}$ inputs $\rightarrow$ frequent if support $\leq 3$

Sulfuric Acid

Acetic Acid

Carbonic Acid

Ammonia
Category II: Frequent Patterns

Frequent Pattern Mining in General

- A database of transactions
- Concepts:
  - Transaction
  - Pattern
  - Support: count or fraction
- Problem:
  - Finding all patterns with support $\geq$ a support threshold

<table>
<thead>
<tr>
<th>Transaction ID</th>
<th>Items Bought</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Bread, Milk</td>
</tr>
<tr>
<td>2</td>
<td>Bread, Diaper, Beer, Eggs</td>
</tr>
<tr>
<td>3</td>
<td>Milk, Diaper, Beer, Coke</td>
</tr>
<tr>
<td>4</td>
<td>Bread, Milk, Diaper, Beer</td>
</tr>
<tr>
<td>5</td>
<td>Beer, Milk, Diaper, Coke</td>
</tr>
<tr>
<td>6</td>
<td>Beer, Milk, Diaper, Bread</td>
</tr>
</tbody>
</table>

Support (Count): $\sigma\{\text{Milk, Bread, Diaper}\} = 2$
Category II: Frequent Patterns

Types of Patterns

- Itemset
- Subsequence
- Subgraph
- Subtree
Category II: Frequent Patterns

Pattern Constraints

• Gap or path length constraints
• Induced v.s. embedded

is an embedded pattern
Category II: Frequent Patterns

Compute-intensive: need parallel computing to scale

The complexity of mining maximal frequent itemsets and maximal frequent patterns

Author: Guizhen Yang  Authors Info & Affiliations

Publication: KDD '04: Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining  •  August 2004  •  Pages 344–353  •  https://doi.org/10.1145/1014052.1014091

ABSTRACT

Mining maximal frequent itemsets is one of the most fundamental problems in data mining. In this paper we study the complexity-theoretic aspects of maximal frequent itemset mining, from the perspective of counting the number of solutions. We present the first formal proof that the problem of counting the number of distinct maximal frequent itemsets in a database of transactions, given an arbitrary support threshold, is #P-complete, thereby providing strong theoretical evidence that the problem of mining maximal frequent itemsets is NP-hard.
Category II: Frequent Patterns

Our Goals

• **Efficient**: all CPU cores are busy mining
• **General Programming Framework**: can be customized to handle all kinds of patterns

- Itemset, order-free
- FP-tree, parallel
- PFP

- Subsequence
- Subgraph
- Subtree
- ...

Prefix Projection, parallel
PrefixPFM

Depth-First Pattern Search/Recursion!
Category II: Frequent Patterns

PrefixSpan (Sequential Pattern Mining)

<table>
<thead>
<tr>
<th>SID</th>
<th>Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_1$</td>
<td>ABCBC</td>
</tr>
<tr>
<td>$s_2$</td>
<td>BABC</td>
</tr>
<tr>
<td>$s_3$</td>
<td>AB</td>
</tr>
<tr>
<td>$s_4$</td>
<td>BC</td>
</tr>
</tbody>
</table>

(a) $D$

Projected DB

Transaction DB

<table>
<thead>
<tr>
<th>SID</th>
<th>Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_1$</td>
<td>_BCBC</td>
</tr>
<tr>
<td>$s_2$</td>
<td>_BC</td>
</tr>
<tr>
<td>$s_3$</td>
<td>_B</td>
</tr>
<tr>
<td>$s_4$</td>
<td></td>
</tr>
</tbody>
</table>

(b) $D|_A$

<table>
<thead>
<tr>
<th>SID</th>
<th>Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_1$</td>
<td>_CBC</td>
</tr>
<tr>
<td>$s_2$</td>
<td>_C</td>
</tr>
<tr>
<td>$s_3$</td>
<td>_</td>
</tr>
</tbody>
</table>

(c) $D|_{AB}$

<table>
<thead>
<tr>
<th>SID</th>
<th>Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_1$</td>
<td>_BC</td>
</tr>
<tr>
<td>$s_2$</td>
<td>_</td>
</tr>
</tbody>
</table>

(d) $D|_{ABC}$

Pattern: ABC
Category II: Frequent Patterns

Depth-First Pattern Growth

Alphabet is \{A, B, C\}

A stack of projected DBs with shrinking size
Effective pattern pruning
Incremental projected DB construction
Child-branches are embarrassingly parallel!!!
Prefix Projection

From Sequences to Graphs

Encode a tree into a sequence

Duplicate patterns for different sequences

BAB$D$$B$C$

BAD$D$$B$C$

BC$B$AB$D$$

canonical encoding

Mohammed Javeed Zaki:
Prefix Projection

From Sequences to Graphs

Deduplicate by canonical encoding

Diagram:

- 0-edge
- 1-edge
- 2-edge
- \ldots
- n-edge

\[ G_0 \]

\[ G_1 \]

Pruned
Prefix Projection

From Sequences to Graphs

Truncated pattern growth
### Related Work

**Transactions and patterns are repeatedly dumped and loaded**

**Not enjoying a shrinking projected DB for frequency examination**

**Early MapReduce-based Papers on Itemsets**

**Breadth-First Pattern Growth**

<table>
<thead>
<tr>
<th>Pattern Mining</th>
<th>Pattern Mining</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>Hadoop Distribute File System</td>
<td>Hadoop Distribute File System</td>
</tr>
<tr>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>Pattern Mining</td>
<td>Pattern Mining</td>
</tr>
<tr>
<td>ABB</td>
<td>ABC</td>
</tr>
</tbody>
</table>
Related Work

Google’s PFP (Parallel FP-growth)

Depth-First Pattern Growth

**PFP: Parallel FP-Growth for Query Recommendation**

Haoyuan Li  
Google Beijing Research,  
Beijing, 100084, China

Yi Wang  
Google Beijing Research,  
Beijing, 100084, China

Dong Zhang  
Google Beijing Research,  
Beijing, 100084, China

Ming Zhang  
Dept. Computer Science,  
Peking University, Beijing,  
100087, China

Edward Chang  
Google Research, Mountain View, CA 94043, USA

**ABSTRACT**

Frequent itemset mining (FIM) is a useful tool for discovering frequently co-occurring items. Since its inception, a number of significant FIM algorithms have been developed to speed up mining performance. Unfortunately, when the dataset size is huge, both the memory use and computational cost can still be prohibitively expensive. In this work, we propose to parallelize the FP-Growth algorithm (we call our parallel algorithm PFP) on distributed machines. PFP developed parallel algorithm on Web data to support query recommendation (or related search).

FIM is a useful tool for discovering frequently co-occurring items. Existing FIM algorithms such as Apriori [9] and FP-Growth [6] can be resource intensive when a mined dataset is huge. Parallel algorithms were developed for reducing memory use and computational cost on each machine. Early efforts (related work is presented in greater detail in Section 1.1) focused on speeding up the Apriori algorithm. Since the FP-Growth algorithm has been shown to run much faster.
Related Work

Graph Patterns: Depth-First Pattern Growth

Transaction DB

Pattern Refinement

UNION

Local Frequent Patterns

DFS mining

Local Frequent Patterns

DFS mining

Local Frequent Patterns

DFS mining
Related Work

Graph Patterns: Breath-First Pattern Growth

• Arabesque (SOSP 2015)
  - Distributed framework, embedding-centric API
  - Combined with dense subgraph mining, graph matching

• RStream (OSDI 2018)
  - Same API, single-machine out-of-core
  - Faster than Arabesque

Pattern-to-instance, not instance-to-pattern
Forced isomorphism check, subgraph materialization

Check out G-thinker !!!
Think Like A Task

Each pattern $\alpha$ is associated with a task $t_\alpha$

• Task $t_\alpha$ checks the frequency of $\alpha$ over $D|\alpha$
• Task $t_\alpha$ grows $\alpha$ by one more element for recursion

If $D|\alpha$ is large, child-patterns are wrapped as tasks (added to a shared queue) for concurrent processing
• Otherwise, the whole pattern subtree is processed by the current tasks
Think Like A Task

```cpp
void run(ostream& fout){
    if(!pre_check(fout)) return;
    //generate new patterns
    setChildren(children);
    //run new child tasks
    while(Task* t= get_next_child()){  
        if(needSplit()){  
            q_mtx.lock();  
            queue().push(t);  
            q_mtx.unlock();  
        }
        else{  
            t->run(fout);  
            delete t;  
        }
    }
}
```

Task queue is a stack
Old tasks are prioritized
# Programming Interface

<table>
<thead>
<tr>
<th>Trans</th>
<th>ProjTrans</th>
<th>Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>int transaction_id // transaction data</td>
<td>int transaction_id // transaction match</td>
<td>// α and D</td>
</tr>
<tr>
<td></td>
<td></td>
<td>UDF: print(ostream&amp; fout)</td>
</tr>
</tbody>
</table>

## Task <PatternT, ChildrenT, TransT>

- PatternT pattern
- ChildrenT children
- UDF: setChildren()
- UDF: Task* get_next_child() //"new" a task from a child pattern
- UDF: bool pre_check(ostream& fout)
- UDF: bool needSplit()
- Entry Function: run(ostream& fout)

## Worker <TaskT>

- ifstream input_file
- UDF: readNextTrans(vector<TransT>& D)
- UDF: setRoot(stack<TaskT*>& Q_task)
- Entry Function: run()
Concurrent Processing

Task Parallelism

Alphabet is \{A, B, C\}

Transaction DB

Projected DB

PDB-parallelism (root task)

Thread 1

Thread 2

Thread 3

Thread 4
Performance

gSpan Parallelization

![Graph showing performance comparison between OpenNCI, Enamine, and GraphGen. The x-axis represents the number of task computing threads, ranging from 1 to 128, and the y-axis represents computation time in seconds, ranging from 0 to 4500. The graph includes three lines, each representing a different tool: OpenNCI in blue, Enamine in orange, and GraphGen in gray. As the number of threads increases, the computation time decreases for all tools, indicating improved performance with parallelization.]
Open Source Software

Check out Our GitHub Page

- [https://github.com/yanlab19870714/PrefixFPM](https://github.com/yanlab19870714/PrefixFPM)
- Demo: [https://youtu.be/PfoCoGDp5w](https://youtu.be/PfoCoGDp5w)

---

**README.md**

**PrefixFPM: A Parallel Framework for Mining Frequent Patterns**

Following divide and conquer, this system treats each pattern to examine and extend as a task, and parallelizes all tasks as much as possible following the idea of divide and conquer. This allows it to use all CPU cores in a machine to mine frequent patterns over big data.

There are 3 applications on top of our framework. Here are the folder structures:

- system: the general-purpose programming interface
- prefixspan: the PrefixSpan parallel version developed on top for sequence mining
- gspan: the gSpan parallel version developed on top for subgraph mining
- sleuth: the Sleuth parallel version developed on top for subtree mining

The applicatons are adapted from:

- **PrefixSpan**: [http://chasen.org/~taku/software/prefixspan/prefixspan-0.4.tar.gz](http://chasen.org/~taku/software/prefixspan/prefixspan-0.4.tar.gz)
- **gSpan**: [https://github.com/rkwitt/gboost/tree/master/src-gspan](https://github.com/rkwitt/gboost/tree/master/src-gspan)
Recent Improvement

Check out Our GitHub Page

- [https://github.com/wenwenQu/PREFIXFPM](https://github.com/wenwenQu/PREFIXFPM)
- Now with more applications
Recent Improvement

Timeout Mechanism

```cpp
void run(ostream& fout) {
    {pseudocode: track task starting time here}
    if (!pre_check(fout)) return;
    // generate new patterns
    setChildren(children);
    // run new child tasks
    while (Task* t = get_next_child()) {
        if (timeout) {
            q_mtx.lock();
            queue().push(t);
            q_mtx.unlock();
        } else if (needSplit()) {
            q_mtx.lock();
            queue().push(t);
            q_mtx.unlock();
        } else {
            t->run(fout);
            delete t;
        }
    }
}
```
Overview

- Graph Mining Problems: A Categorization
- Think Like a Vertex: Data-Intensive Mining
- Why TLAV is not Sufficient? Compute-Intensive!
- PrefixFPM for Frequent Subgraph Mining
- **G-thinker: Think Like a Subgraph**
- TLAS Algorithms for Subgraph Mining
Target Problems

Finding dense subgraphs in a big graph

- Maximum/maximal Cliques
- Quasi-cliques, $k$-plexes
- Other definitions
  - $n$-cliques
  - $n$-clans
  - $n$-clubs
  - ...
Target Problems

Finding dense subgraphs in a big graph

• Applications:
  - Social communities
  - Gene regulatory network, PPI network
Target Problems

Subgraph matching (not mining)
- Graph pattern recognition (e.g., on biological networks)
- Searching text-rich networks like the Semantic Web
Problem Features

High computational complexity

Search space: power set of vertex set

S: set of nodes already in subgraph
ext(S): set of valid candidates to add
Problem Features

Divide and conquer

Serial algorithms: subgraph backtracking

Example: Finding Maximal Cliques
Problem Features

Divide and conquer

Serial algorithms: subgraph backtracking

One-Hop Neighborhood

G
Problem Features

Divide and conquer

Serial algorithms: subgraph backtracking

Recursive: $v_1$’s one-hop
Problem Features

Divide and conquer  
Recursive: \(\{v_1 + v_3\}\)’s one-hop

Serial algorithms: subgraph backtracking
Problem Features

Divide and conquer

Serial algorithms: subgraph backtracking

Recursive: \{v_1 + v_4\}'s one-hop
Problem Features

Divide and conquer

Serial algorithms: subgraph backtracking

Recursive: \( \{v_1 + v_5\} \)’s one-hop
Existing Solutions

MapReduce, Vertex-Centric Systems

Subgraph-Centric Systems

NScale, Arabesque, Rstream, Nuri, G-Miner, …

Arabesque’s API

```java
boolean filter (Embedding e) {
    return isClique(e); }
void process (Embedding e) { output(e); }
```

(c) Finding cliques (vertex-based exploration)

API is natural
Still IO-bound
Existing Solutions

NScale

» BFS around each node to construct $k$-hop subgraph
» Conducted by $k$ MapReduce jobs (IO-bound)
» Intermediate subgraphs materialized (HDFS overhead)
» Subgraphs are then distributed for processing by reducers (straggler’s problem)
Existing Solutions

Carlos H. C. Teixeira, Alexandre J. Fonseca, Marco Serafini, Georgos Siganos, Mohammed J. Zaki, Ashraf Aboulnaga:

Arabesque: a system for distributed graph mining. SOSP 2015: 425-440

Arabesque

» Embedding-centric API
» Subgraphs are materialized in memory and checked in increasing size 1, 2, ...
» Distributed, each machine keeps an entire graph

In contrast, G-thinker uses backtracking as much as possible
Existing Solutions

Carlos H. C. Teixeira, Alexandre J. Fonseca, Marco Serafini, Georgos Siganos, Mohammed J. Zaki, Ashraf Aboulnaga:

Arabesque: a system for distributed graph mining. SOSP 2015: 425-440

Arabesque

» Embedding-centric API
» To incorporate frequent subgraph pattern mining, need isomorphism check for each materialized subgraph

Not needed in our subgraph mining problem!
Problem Features

High computational complexity

Search space: power set of vertex set

\{a\}, \{b\}, \{c\}, \{d\}, \{a, b\}, \{a, c\}, \{a, d\}, \{b, c\}, \{b, d\}, \{c, d\}, \{a, b, c\}, \{a, b, d\}, \{a, c, d\}, \{b, c, d\}, \{a, b, c, d\}

\text{a} < \text{b} < \text{c} < \text{d}
Existing Solutions

Carlos H. C. Teixeira, Alexandre J. Fonseca, Marco Serafini, Georgos Siganos, Mohammed J. Zaki, Ashraf Aboulnaga:
Arabesque: a system for distributed graph mining. SOSP 2015: 425-440

Arabesque

» Embedding-centric API
» To incorporate frequent subgraph pattern mining, need isomorphism check for each materialized subgraph

Recall PrefixFPM

Not needed in our subgraph mining problem!

API is simple but… execution is inefficient
Existing Solutions

Kai Wang, Zhiqiang Zuo, John Thorpe, Tien Quang Nguyen, Guoqing Harry Xu: 
**RStream: Marrying Relational Algebra with Streaming for Efficient Graph Mining on A Single Machine.**
OSDI 2018: 763-782

**RStream**

» Inherits the embedding-centric API
» Single-machine out-of-core
» More efficient than Arabesque

Disk IO-bound
Solutions so far do not scale with CPU cores!

Existing Solutions

2018

Aparna Joshi, Yu Zhang, Petko Bogdanov, Jeong-Hyon Hwang:
An Efficient System for Subgraph Discovery. BigData 2018: 703-712

Nuri

» To find the $k$ most relevant subgraphs using only a single computer
» Prioritized subgraph expansion

openly available.

Both Nuri and Nuri-NP are implemented as single-threaded Java programs using the standard Java 8 distribution. In our experiments, each of these versions was run on a single core
Figure 11: Clique Discovery (YouTube)
A G-thinker Demo

```
[guimugo@redmnt-0-0 app_maxclique]$ hadoop fs -cat youtube_sorted/* | wc -l
1134890
[guimugo@redmnt-0-0 app_maxclique]$ 
```

There are 1.1 million vertices in this graph
A G-thinker Demo

Check out our demo at http://bit.ly/gthinker

The Curse of IO

COST in the land of databases

Several years ago Michael Isard, Derek Murray, and I wrote a paper: "Scalability, but at what COST". The premise of the paper was that the current excitement about distributed computation (e.g. Hadoop, Spark) produced implementations that improve when you give them more resources (they "scale") but whose performance never quite gets to where you would be with a simple single-threaded implementation on a laptop.

If that sounds surprising, ... well hello there! Maybe go check out the paper, the matching blog post, a follow-up post about even larger datasets, and another followup post about another paper from the same community a year later.

One conclusion was that the Systems community (which I think of as SOSP/OSDI/+ was overexcited about big data processing,

Current data-intensive frameworks are IO-bound!
Our Solution

Think Like a Task

• Target giant search space, high time complexity
• Target problems amenable to divide and conquer

Lots of problems !!!


\[
\begin{align*}
  c_{\text{cpu}} \cdot f(|g|) \\
  c_{\text{net}} \cdot |g|
\end{align*}
\]
Our Solution

Examples that fall in Think-Like-a-Task Paradigm

• Some divide-and-conquer (often recursive) algorithms that are straightforward to implement on G-thinker (beyond simple ones like triangle counting and maximal clique finding, i.e., Bron-Kerbosch algorithm)

Guimei Liu, Limsoon Wong: Effective Pruning Techniques for Mining Quasi-Cliques. ECML/PKDD (2) 2008: 33-49

Alessio Conte, Tiziano De Matteis, Daniele De Sensi, Roberto Grossi, Andrea Marino, Luca Versari: D2K: Scalable Community Detection in Massive Networks via Small-Diameter k-Plexes. KDD 2018: 1272-1281

G-thinker History

Prototype: API proposed
- No multithreading support, running multiple processes on each machine
- Inefficient engine:
  - Vertex cache cannot be concurrently visited by tasks
  - LSH-based cache design that keeps a lot of partial tasks on disk

Basically our old G-thinker system with multithreading support
But inherits all bad designs, and thus do not scale

Our new G-thinker implementation has great documentation, and we actively provide technical support 😊
G-thinker Overview

Local Disk

Task Queues

spill

refill

Local Vertex Table

Remote Vertex Cache

Hadoop Distributed File System (HDFS)
G-thinker

Vertex Partitioning

Distributed Key-Value Store
- Key: vertex ID
- Value: (labeled) adjacency list

\[
\begin{align*}
M_0 & \quad 0 \rightarrow 1 \ 3 \\
 & \quad 3 \rightarrow 0 \ 1 \ 2 \ 7 \\
 & \quad 6 \rightarrow 5 \ 8 \\
M_1 & \quad 1 \rightarrow 0 \ 2 \ 3 \\
 & \quad 4 \rightarrow 2 \ 5 \ 7 \\
 & \quad 7 \rightarrow 2 \ 3 \ 4 \ 8 \\
M_2 & \quad 2 \rightarrow 1 \ 3 \ 4 \ 7 \\
 & \quad 5 \rightarrow 4 \ 6 \\
 & \quad 8 \rightarrow 6 \ 7
\end{align*}
\]
G-thinker

Vertex Pulling & Remote Vertex Caching

Tasks can share vertices to reduce # of remote vertex requests
G-thinker

Task-Centric Programming Interface

» **UDF**: \textit{spawn\_task}(vertex)

» **UDF**: \textit{compute}(task, frontier)

» \textit{pull}(vertex\_id)

» \textit{add\_task}(task)
**G-thinker**

**Vertex Cache: High Concurrency**

» Vertices hashed to different buckets can be accessed concurrently

<table>
<thead>
<tr>
<th>Bucket #</th>
<th>( \Gamma )-tables</th>
<th>Z-tables</th>
<th>R-tables</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ID((v))</td>
<td>(\Gamma((v)), \text{lock-count})</td>
<td>ID((v))</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>...</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>2</td>
<td>12</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>3</td>
<td>13</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>9</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

\[ B_v = v \mod 10 \]

\(\Gamma\)-table is for getting adjacency list given a vertex ID
Counter tracks how many tasks lock a vertex

Vertices hashed to buckets

Buckets protected by mutex

Each bucket has
- A \(\Gamma\)-table
- A Z-table
- An R-table
Z-table tracks the subset of vertices that can be evicted. Otherwise, GC has to scan an entire Γ-table.

**G-thinker**

**Vertex Cache: High Concurrency**

» Vertices hashed to different buckets can be accessed concurrently.

<table>
<thead>
<tr>
<th>Bucket #</th>
<th>Γ-tables</th>
<th>Z-tables</th>
<th>R-tables</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ID((v))</td>
<td>Γ((v)), lock-count</td>
<td>ID((v))</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>Γ(10), 0</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>Γ(11), 1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>12</td>
<td>Γ(12), 1</td>
<td>22</td>
</tr>
<tr>
<td>3</td>
<td>13</td>
<td>Γ(13), 2</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ B_v = v \mod 10 \]
G-thinker

Vertex Cache: High Concurrency

- Vertices hashed to different buckets can be accessed concurrently

R-table tracks those vertices requested but not received yet
Used to avoid sending the same vertex-request again

**Bucket #** | **Γ-tables** | **Z-tables** | **R-tables**
---|---|---|---
0 | ID(v) | Γ(10), lock-count | ID(v) | ID(v), lock-count
   | 10 | 0 | 10 | 20 | 1
1 | 11 | 1 | ... | ... | ...
2 | 12 | 1 | ... | 22 | 3
3 | 13 | 2 | ... | ... | ...
... | ... | ... | ... | ... | ...
9 | ... | ... | ... | ... | ...

\[ B_v = v \mod 10 \]
G-thinker

Vertex Cache: High Concurrency
» Computing threads

<table>
<thead>
<tr>
<th>Bucket #</th>
<th>Γ-tables</th>
<th>Z-tables</th>
<th>R-tables</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>10</td>
<td>Γ(10), 0</td>
<td>20</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>Γ(11), 1</td>
<td>...</td>
</tr>
<tr>
<td>2</td>
<td>12</td>
<td>Γ(12), 1</td>
<td>...</td>
</tr>
<tr>
<td>3</td>
<td>13</td>
<td>Γ(13), 2</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>9</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bucket #</th>
<th>Γ-tables</th>
<th>Z-tables</th>
<th>R-tables</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>10</td>
<td>Γ(10), 1</td>
<td>...</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>Γ(11), 2</td>
<td>...</td>
</tr>
<tr>
<td>2</td>
<td>12</td>
<td>Γ(12), 0</td>
<td>...</td>
</tr>
<tr>
<td>3</td>
<td>13</td>
<td>Γ(13), 2</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>9</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

pull(10) pull(11)
G-thinker

Vertex Cache: High Concurrency
» Computing threads

<table>
<thead>
<tr>
<th>Bucket #</th>
<th>Γ-tables</th>
<th>Z-tables</th>
<th>R-tables</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ID((v))</td>
<td>Γ((v)), lock-count</td>
<td>ID((v))</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>Γ(10), 0</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>Γ(11), 1</td>
<td>...</td>
</tr>
<tr>
<td>2</td>
<td>12</td>
<td>Γ(12), 1</td>
<td>...</td>
</tr>
<tr>
<td>3</td>
<td>13</td>
<td>Γ(13), 2</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>9</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bucket #</th>
<th>Γ-tables</th>
<th>Z-tables</th>
<th>R-tables</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>10</td>
<td>Γ(10), 1</td>
<td>...</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>Γ(11), 2</td>
<td>...</td>
</tr>
<tr>
<td>2</td>
<td>12</td>
<td>Γ(12), 0</td>
<td>12</td>
</tr>
<tr>
<td>3</td>
<td>13</td>
<td>Γ(13), 1</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

release(12)
release(13)
# G-thinker

## Vertex Cache: High Concurrency
» Computing threads

<table>
<thead>
<tr>
<th>Bucket #</th>
<th>Γ-tables</th>
<th>Z-tables</th>
<th>R-tables</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ID((v))</td>
<td>(\Gamma(v)), lock-count</td>
<td>ID((v))</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>Γ(10), 0</td>
<td>20</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>Γ(11), 1</td>
<td>...</td>
</tr>
<tr>
<td>2</td>
<td>12</td>
<td>Γ(12), 1</td>
<td>22</td>
</tr>
<tr>
<td>3</td>
<td>13</td>
<td>Γ(13), 2</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bucket #</th>
<th>Γ-tables</th>
<th>Z-tables</th>
<th>R-tables</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>10</td>
<td>Γ(10), 1</td>
<td>...</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>Γ(11), 2</td>
<td>21</td>
</tr>
<tr>
<td>2</td>
<td>12</td>
<td>Γ(12), 0</td>
<td>12</td>
</tr>
<tr>
<td>3</td>
<td>13</td>
<td>Γ(13), 2</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

*pull*(20)
*pull*(21)
G-thinker

Vertex Cache: High Concurrency

» Response receiving thread
G-thinker

Vertex Cache: High Concurrency
» Garbage Collection (GC) thread

Lazy exec: triggered by cache overflow
Scanning Z-tables repeatedly
Lock one bucket each time
• Remove an entry in Z-table
• Remove that entry in Γ-table

<table>
<thead>
<tr>
<th>Bucket #</th>
<th>Γ-tables</th>
<th>Z-tables</th>
<th>R-tables</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ID(v)γ(v), lock-count</td>
<td>ID(v)γ(v), lock-count</td>
<td>ID(v)γ(v), lock-count</td>
</tr>
<tr>
<td>0</td>
<td>10 Γ(10), 1</td>
<td>...</td>
<td>20 ...</td>
</tr>
<tr>
<td>1</td>
<td>11 Γ(11), 2</td>
<td>...</td>
<td>21 ...</td>
</tr>
<tr>
<td>2</td>
<td>12 Γ(12), 0</td>
<td>12 ...</td>
<td>...</td>
</tr>
<tr>
<td>3</td>
<td>13 Γ(13), 2</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>9</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
G-thinker

Task Management

» Shared data store on a machine
» Accessed by all computing threads
G-thinker

Task Management

» Shared data store on a machine
» Accessed by all computing threads

Data Src 1: Local Vertex Table

Data Src 2: Disk-Buffered Tasks

Data Src 3: Tasks Waiting for Remote Data
G-thinker

Task Management

» Task containers maintained by each computing thread
G-thinker

Task Management

» Task queue for each thread (no contention)
» Underflow: get a batch of tasks from shared sources
» Overflow: buffer a batch of tasks at tail to disk
G-thinker

Task Management

» For each popped task, pull requested vertices
» If need to wait for remote vertex: enter task-table
» Set <#met, #req>

Container 1: Task Queue

Container 2: Task Table
G-thinker

Task Management

» Vertex response triggers task update
» #met ++; If #met == #req, move task to $B_{task}$
G-thinker

**Task Management**

» A computing thread repeats
  
  • **pop()**: pop a task for pulling/computing, underflow handling
  
  • **push()**: fetch a task from $B_{task}$ for computing, add to $Q_{task}$
G-thinker

Task Management

» Task stealing:

• Estimate pending tasks: local table + disk-buffered tasks

• Stolen tasks are buffered to disks for computing thread to fetch
G-thinker

Performance

Datasets:

| Dataset | |V| | |E| | Max Degree | Avg Degree |
|---------|---------|---------|---------|---------|---------|---------|---------|
| Youtube | 1,134,890 | 2,987,624 | 28,754 | 5.27 |
| Skitter | 1,696,415 | 11,095,298 | 35,455 | 13.08 |
| Orkut | 3,072,441 | 117,184,899 | 33,313 | 76.28 |
| BTC | 164,732,473 | 361,411,047 | 1,637,619 | 4.39 |
| Friendster | 65,608,366 | 1,806,067,135 | 5,124 | 55.06 |

Dataset | Arabesque | Giraph | G-Miner | G-thinker |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(a) TC</td>
<td>Youtube</td>
<td>60.3 s / 2.8 GB</td>
<td>74.3 s / 1.5 GB</td>
<td>14.7 s / 1 GB</td>
</tr>
<tr>
<td>Skitter</td>
<td>90.8 s / 4.8 GB</td>
<td>73.9 s / 3.9 GB</td>
<td>17.1 s / 1.1 GB</td>
<td>9.5 s / 0.5 GB</td>
</tr>
<tr>
<td>Orkut</td>
<td>533 s / 17.7 GB</td>
<td>197 s / 15 GB</td>
<td>667 s / 2.3 GB</td>
<td>26.6 s / 1.2 GB</td>
</tr>
<tr>
<td>BTC</td>
<td>x</td>
<td>x</td>
<td>&gt; 24 hr / 6.7 GB</td>
<td>181 s / 3 GB</td>
</tr>
<tr>
<td>Friendster</td>
<td>x</td>
<td>x</td>
<td>10915 s / 9.2 GB</td>
<td>516 s / 3 GB</td>
</tr>
<tr>
<td>(b) MCF</td>
<td>Youtube</td>
<td>58.8 s / 4.7 GB</td>
<td>177 s / 6.2 GB</td>
<td>13.7 s / 1 GB</td>
</tr>
<tr>
<td>Skitter</td>
<td>145 s / 4.9 GB</td>
<td>x</td>
<td>26.2 s / 1.2 GB</td>
<td>85.6 s / 0.6 GB</td>
</tr>
<tr>
<td>Orkut</td>
<td>2007 s / 44.7 GB</td>
<td>x</td>
<td>691 s / 2.5 GB</td>
<td>95.9 s / 1.3 GB</td>
</tr>
<tr>
<td>BTC</td>
<td>x</td>
<td>x</td>
<td>&gt; 24 hr / 7.3 GB</td>
<td>1831 s / 3 GB</td>
</tr>
<tr>
<td>Friendster</td>
<td>x</td>
<td>x</td>
<td>10644 s / 7.4 GB</td>
<td>252 s / 3.4 GB</td>
</tr>
<tr>
<td>(c) GM</td>
<td>Youtube</td>
<td>–</td>
<td>–</td>
<td>13.2 s / 0.8 GB</td>
</tr>
<tr>
<td>Skitter</td>
<td>–</td>
<td>–</td>
<td>13.9 s / 1.1 GB</td>
<td>7.8 s / 0.6 GB</td>
</tr>
<tr>
<td>Orkut</td>
<td>–</td>
<td>–</td>
<td>66.2 s / 2.2 GB</td>
<td>46.7 s / 1.5 GB</td>
</tr>
<tr>
<td>BTC</td>
<td>–</td>
<td>–</td>
<td>&gt; 24 hr / 6 GB</td>
<td>153 s / 4.4 GB</td>
</tr>
<tr>
<td>Friendster</td>
<td>–</td>
<td>–</td>
<td>3669 s / 9.2 GB</td>
<td>1762 s / 7 GB</td>
</tr>
</tbody>
</table>

Note: (1) x = Out of memory; (2) “−” means inapplicable.
Recent improvement to G-thinker

» Accepted and to appear in PVLDB 2021
» Besides local task queues for each computing thread, each machine adds a global task queue shared by all computing threads
» Time-consuming tasks are put in the global task queue for prioritized scheduling and for timely work stealing
» Motivated by the fact that some tasks can be stragglers in pseudo-clique mining

Guimu Guo, Da Yan, M. Tamer Özsu, Zhe Jiang:
https://github.com/yanlab19870714/gthinkerQC

G-thinker+
G-thinker+

- small task
- big task

Task Queues

Head-of-line blocking

Task Queues

https://github.com/yanlab19870714/gthinkerQC
G-thinker+

- small task
- big task

Work Stealing

Task Queues
Overview

• Graph Mining Problems: A Categorization
• Think Like a Vertex: Data-Intensive Mining
• Why TLAV is not Sufficient? Compute-Intensive!
• PrefixFPM for Frequent Subgraph Mining
• G-thinker: Think Like a Subgraph

• TLAS Algorithms for Subgraph Mining

Think like a subgraph, each task maintains a subgraph
Algorithms on G-thinker

Algorithm Features

• Divide and conquer, often recursive
• Some tasks can be stragglers
• Task time difficult to predict from graph characteristics
G-thinker

Task-Centric Programming Interface

» **UDF:** `spawn_task(vertex)`
» **UDF:** `compute(task, frontier)`
» `pull(vertex_id)`
» `add_task(task)`
Algorithms on G-thinker

$S$: set of nodes already in subgraph

$\text{ext}(S)$: set of valid candidates to add

Can be a straggler!

UDF: $\text{spawn\_task}(\text{vertex})$
Figure 3: Subgraph Features v.s. Task Time on YouTube
Figure 5: Subgraph Features v.s. Logarithmic Task Time on Patent
Algorithms on G-thinker

Timeout mechanism to avoid stragglers
Algorithms on G-thinker

Timeout mechanism to avoid stragglers

When \{a, c\} computes, it may timeout and decompose again
Algorithms on G-thinker

Some \((S, \text{ext}(S))\)-tasks can be pruned

\[
\{a\} \quad \{b\} \quad \{c\} \quad \{d\} \\
\{a, b\} \quad \{a, c\} \quad \{a, d\} \quad \{b, c\} \quad \{b, d\} \quad \{c, d\} \\
\{a, b, c\} \quad \{a, b, d\} \quad \{a, c, d\} \quad \{b, c, d\} \\
\{a, b, c, d\} \\
\]

Only pull \(v > c\)

\(a < b < c < d\)
Algorithms on G-thinker

Quasi-clique:

Given a user-specified minimum degree threshold $\gamma$, a $\gamma$-quasi-clique is a subgraph $G = (V, E)$ where each vertex $v \in V$ connects to at least $\gamma$ fraction of the other vertices (i.e., $\lceil \gamma \cdot (|V| - 1) \rceil$ vertices) in $G$.

Assume $\gamma \geq 0.5$, diameter of a $\gamma$-quasi-clique is at most 2.

Proof: $\forall u, v \in V,$

Sum to $|V| + 1$
Algorithms on G-thinker

Quasi-clique:

Given a user-specified minimum degree threshold $\gamma$, a $\gamma$-quasi-clique is a subgraph $G = (V, E)$ where each vertex $v \in V$ connects to at least $\gamma$ fraction of the other vertices (i.e., $\lceil \gamma \cdot (|V| - 1) \rceil$ vertices) in $G$.

Assume $\gamma \geq 0.5$, diameter of a $\gamma$-quasi-clique is at most 2.

Proof: $\forall u, v \in V,$

Sum to $|V| + 1$
Algorithms on G-thinker

Some \((S, \text{ext}(S))\)-tasks can be pruned

Do not pull vertices > 2 hops away when \(γ ≥ 0.5\)

Do not pull vertices > 1 hop away when we consider cliques

\[a < b < c < d\]
Assuming the mining of maximal quasi-cliques

Algorithms on G-thinker

Some \((S, \text{ext}(S))\)-tasks can be pruned

Max-degree vertex “a” covers \{w, x, y, z\}

Lots of other pruning rules, see our PVLDB’21 paper for details
Algorithms on G-thinker

Maximum Clique Finding

• Support of aggregator
  - Local threads update the local aggregator
  - Aggregators are synchronized every 1 second
  - If $|S| + |\text{ext}(S)|$ is less than the size of the current maximum clique found, prune task $(S, \text{ext}(S))$
Conclusion

A number of papers are on their way

Follow our work!

Just us right now…
Contact Info:
Email: guimuguo@uab.edu
Webpage: https://guimuguo.github.io/

GUO, Guimu

Thanks!

YAN, Da

Contact Info:
Email: yanda@uab.edu