COLLEGE OF ARTS AND SCIENCES

Department of Computer Science

IEEE BigData 2020

Now Taking Place Virtually

Tutorial 3: Systems and Algorithms for Massively Parallel Graph Mining

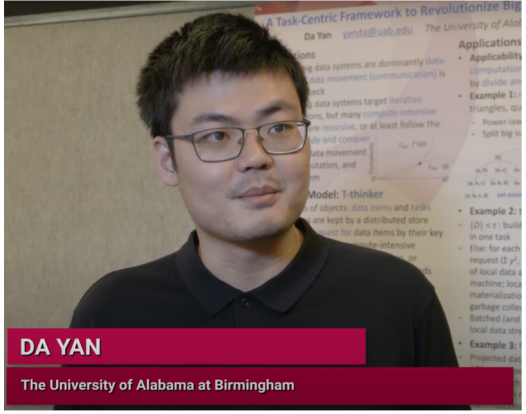
Da Yan yanda@uab.edu

Guimu Guo guimuguo@uab.edu

Department of Computer Science

The University of Alabama at Birmingham (UAB)

Part I: Delivered by Da Yan



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

Part II: Delivered by Guimu Guo

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

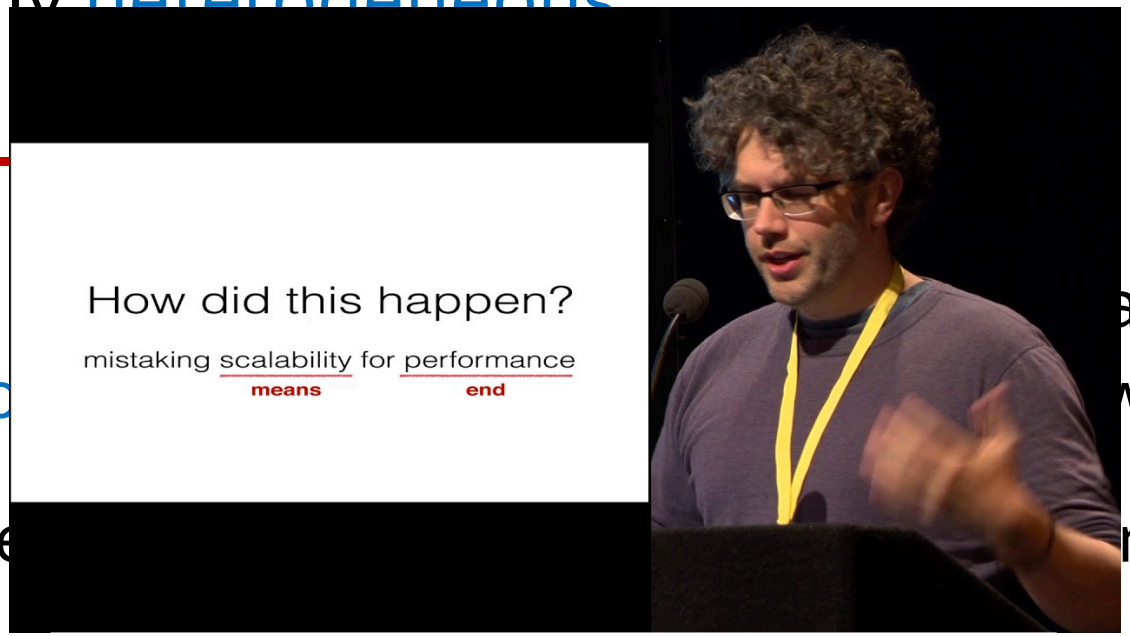
- **Graph Mining Problems: A Categorization**
- Think Like a Vertex: Data-Intensive Mining
- Why TLAV is not Sufficient? Compute-Intensive!
- PrefixFPM for Frequent Subgraph Mining
- G-thinker: Think Like a Subgraph
- TLAS Algorithms for Subgraph Mining

Problem Categorization

Unlike relational database, operations on graph are highly heterogeneous

No one-

- There aph heterc work
- Simple rts !



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 !

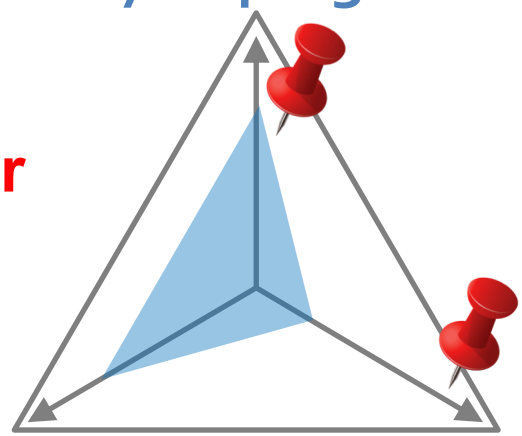
Problem Categorization

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

We assume $G = (V, E)$

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:
Pregel Algorithms for Graph Connectivity Problems with Performance Guarantees. Proc. VLDB Endow. 7(14): 1821-1832 (2014)

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

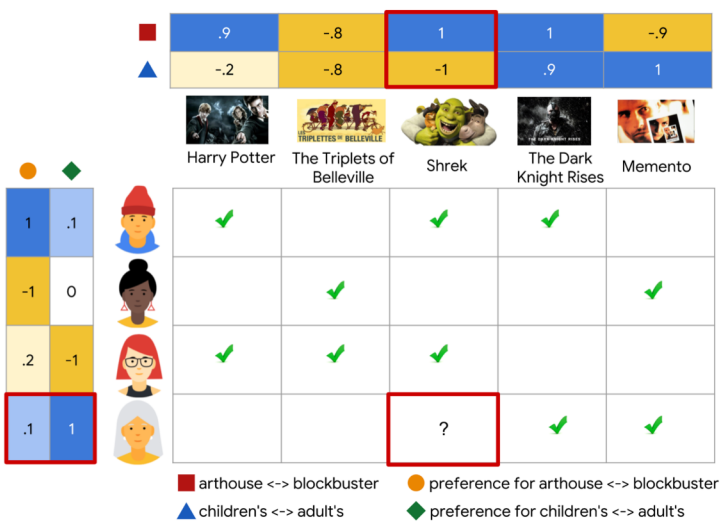
Wenfei Fan, Floris Geerts, Frank Neven:
Making Queries Tractable on Big Data with Preprocessing. Proc. VLDB Endow. 6(9): 685-696 (2013)

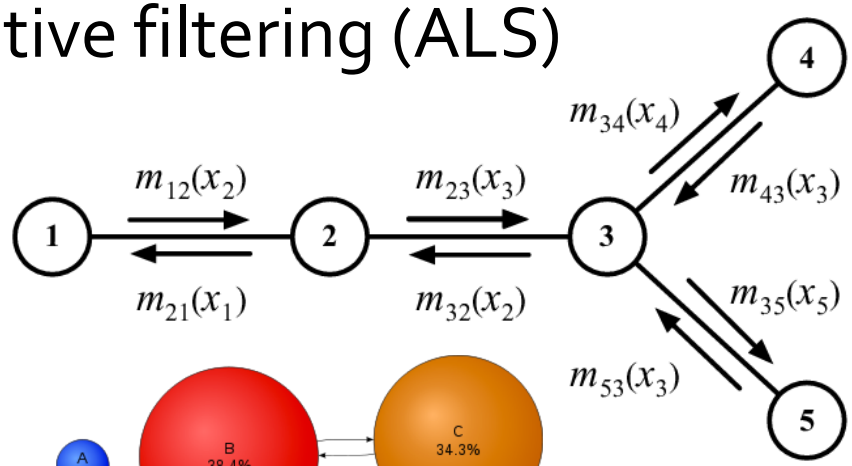
Yufei Tao, Wenqing Lin, Xiaokui Xiao:
Minimal MapReduce algorithms. SIGMOD Conference 2013: 529-540

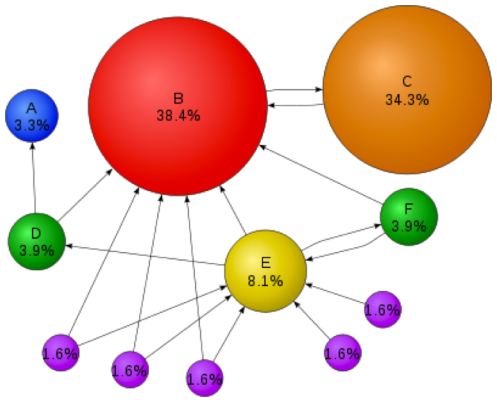
Problem Categorization

Category I: Iterative Algorithms

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



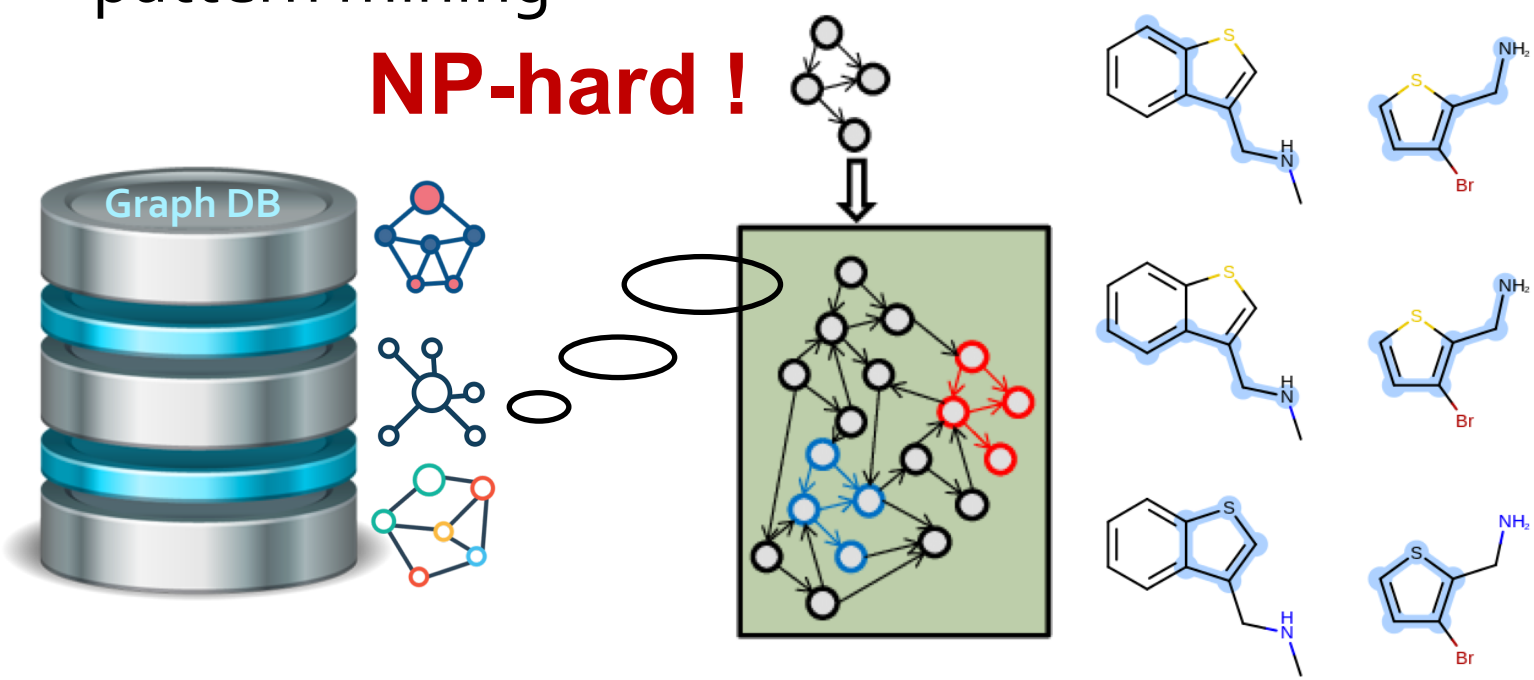




Problem Categorization

Category II: Patterns from Transaction DB

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

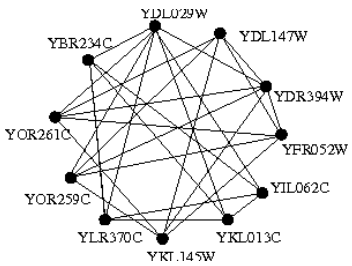


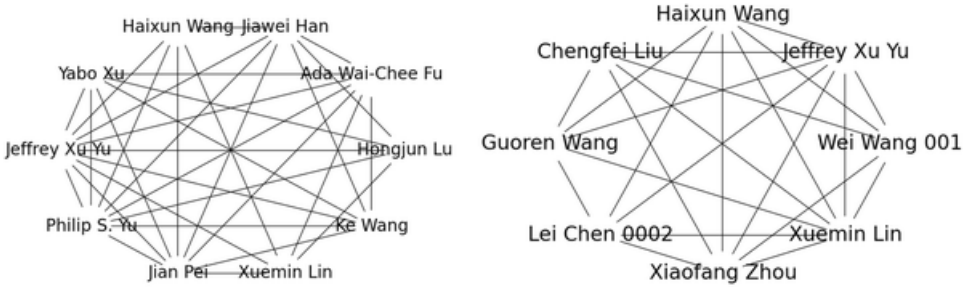
Problem Categorization

Category III: Patterns from a Big Graph

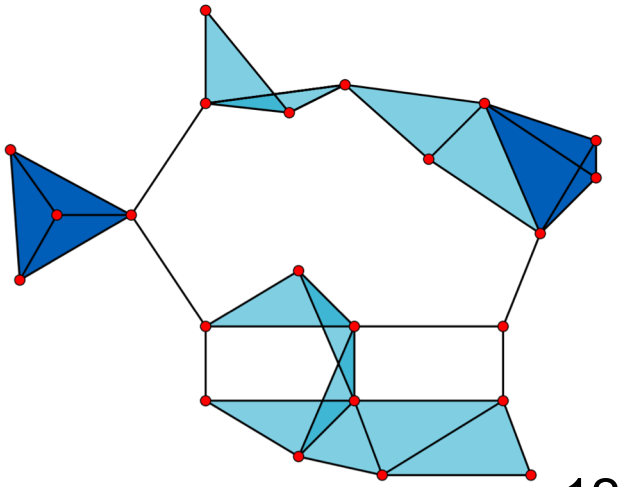
- Examples: dense subgraph mining (triangle, *k*-truss, clique, *k*-plex, γ-quasi-clique), graph matching (not mining?)

Often NP-hard !





(a) $Q = \{\text{Jiawei Han}\}, k = 3$ **(b)** $Q = \{\text{Xuemin Lin, Jeffrey Xu Yu}\}, k = 3$



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

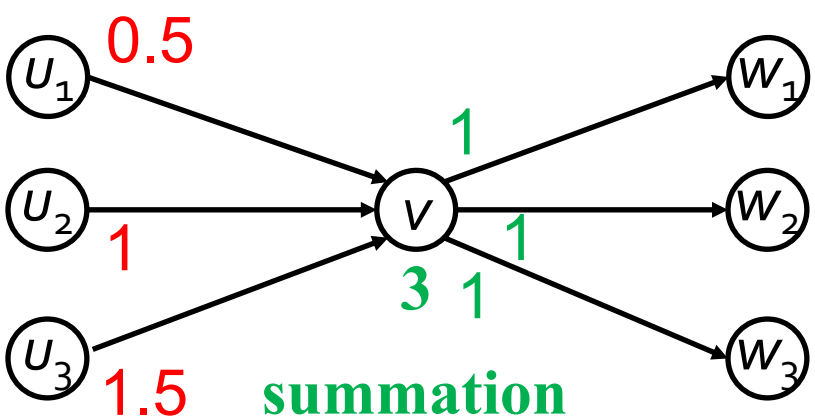
Category I: Iterative Algorithms

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.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();
}
```

TLAV is not just for mining

Category I: Iterative Algorithms

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.[1] 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.

Need pointer jump for "log" rounds

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

Work may increase in parallel setting

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

Pregel

GraphLab

APACHE GIRAPH

Spark

GraphX

… …

Category I: Iterative Algorithms

Design Variants in Pregel-Like Systems

API: GAS (Gather, Apply and Scatter)

GraphLab

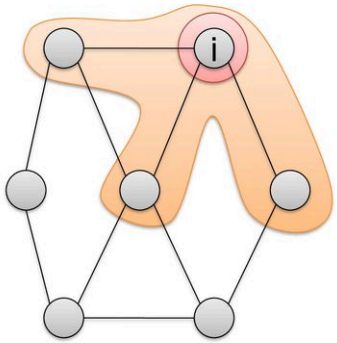
as an example

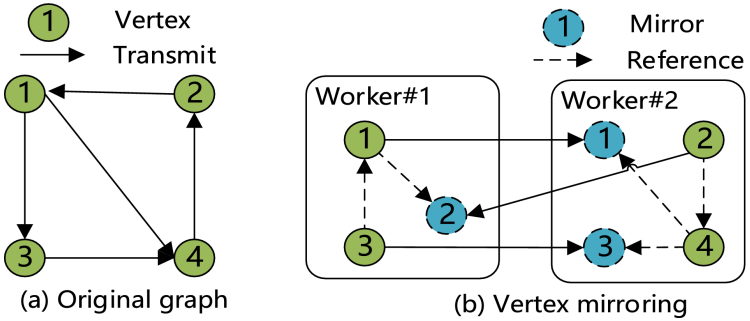
Vertex-Programs directly **read** the neighbors state

```
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
```





Vertex-cut

Google

TLAV Pionee

» Message pas

» Iterative

- Each iterati

Not sur

But insp

Home About Membership PODS Publications History

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

dl.acm.org › doi

Pregel: a system for large-scale graph processing - ACM ...

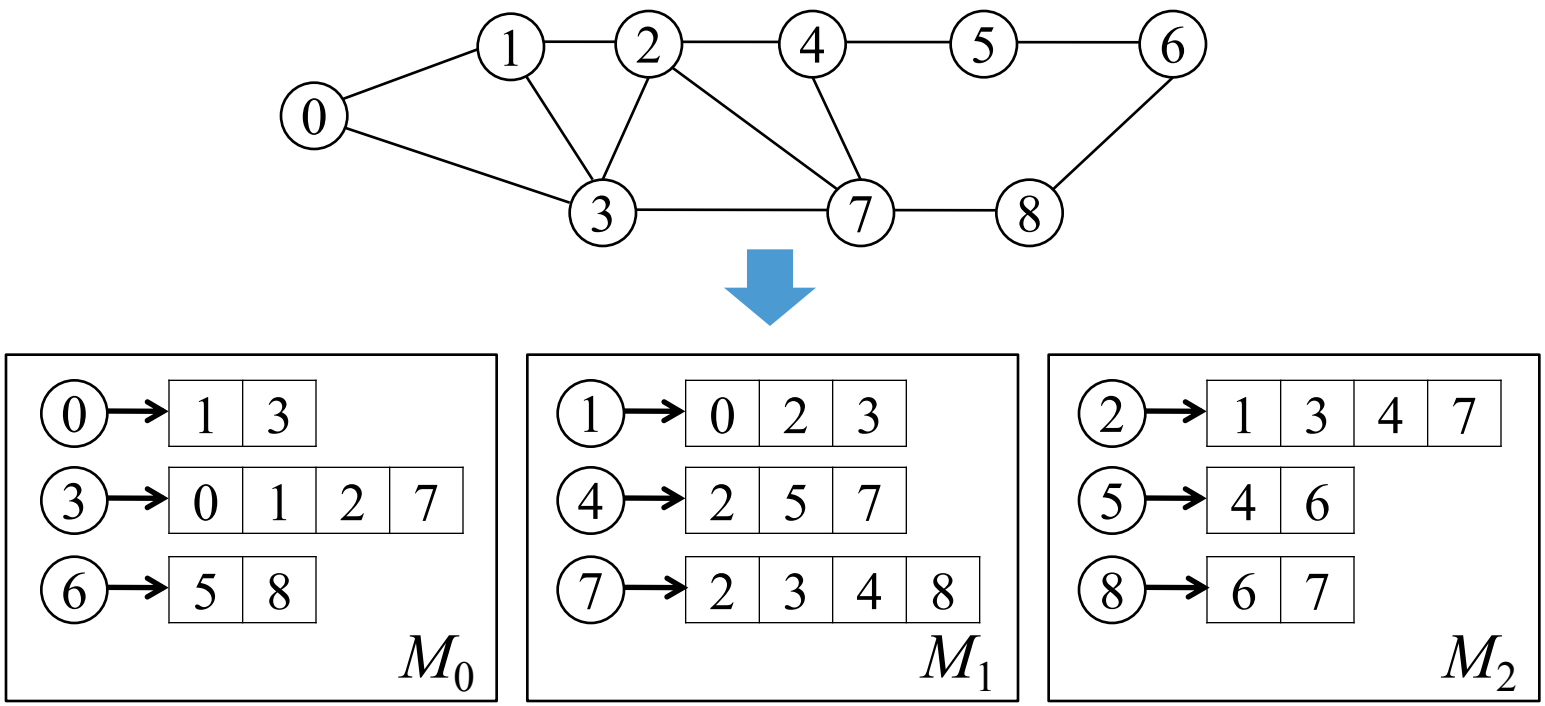
The **scale** of these **graphs** - in some cases billions of vertices, trillions of edges - poses challenges to their efficient **processing**. In this paper we present a ...

by G Malewicz · 2010 · Cited by 4000 · Related articles

ABSTRACT · References · Index Terms

Google's Pregel

Vertex Partitioning



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

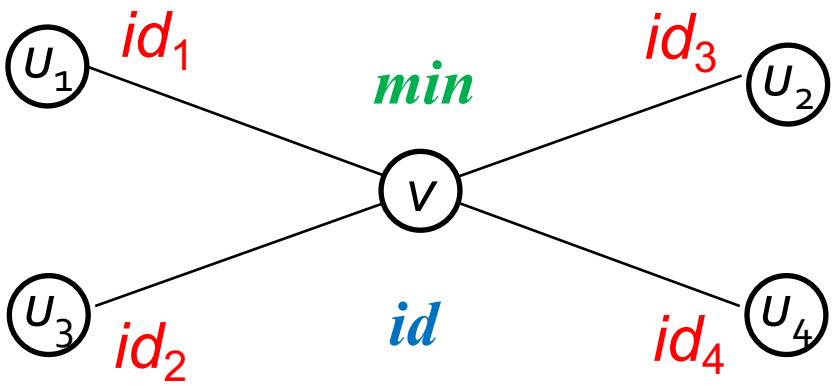
Stop condition

» All vertices are halted, and
» No pending messages for the next superstep

Google's Pregel

Hash-Min: Connected Components

***v*.compute(*messages*)**

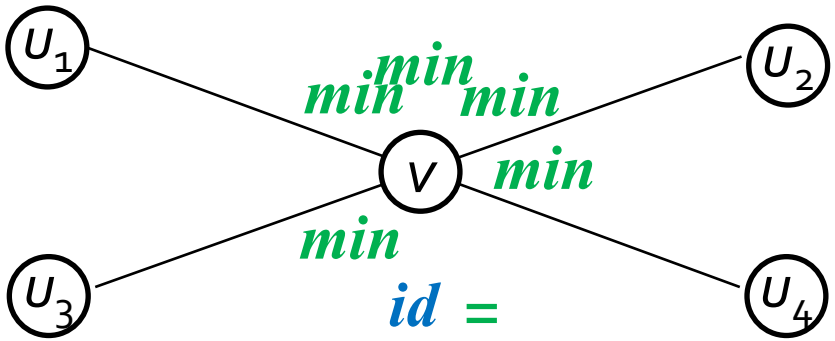


Google's Pregel

Hash-Min: Connected Components

***v*.compute(*messages*)**

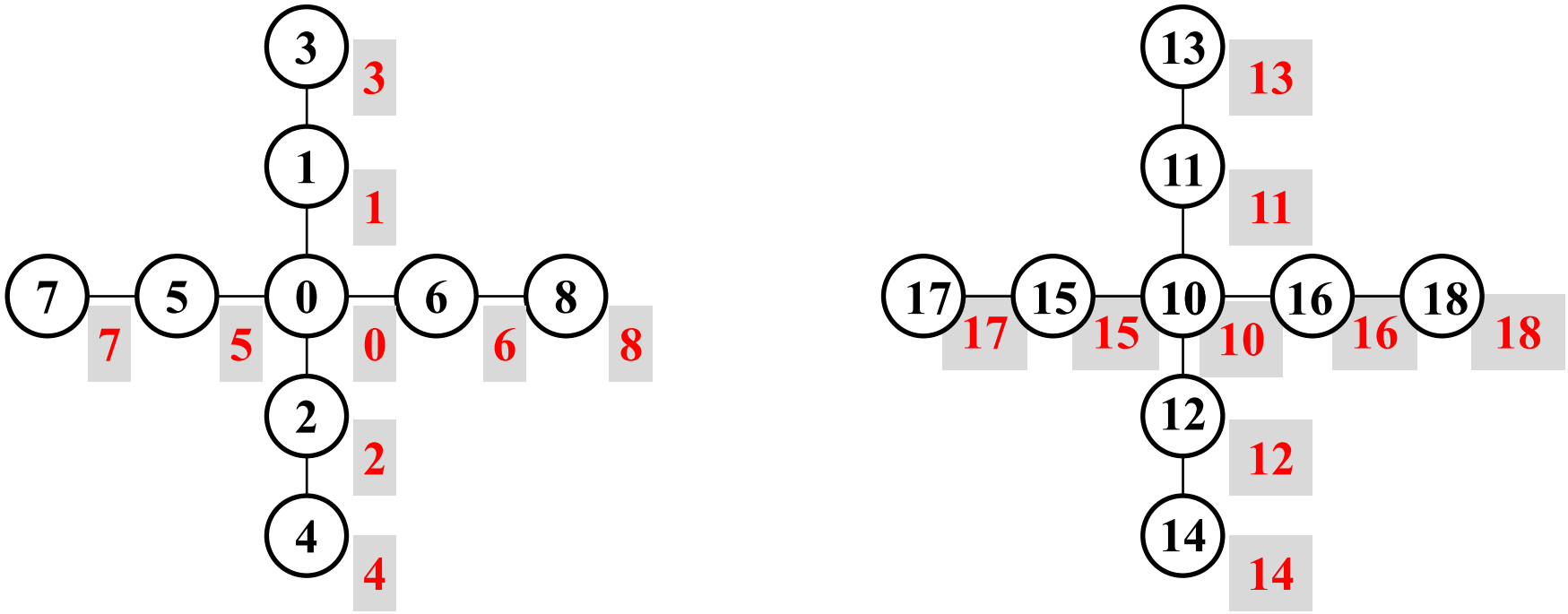
If *min* < *id*:



Finally, vote to halt

Google's Pregel

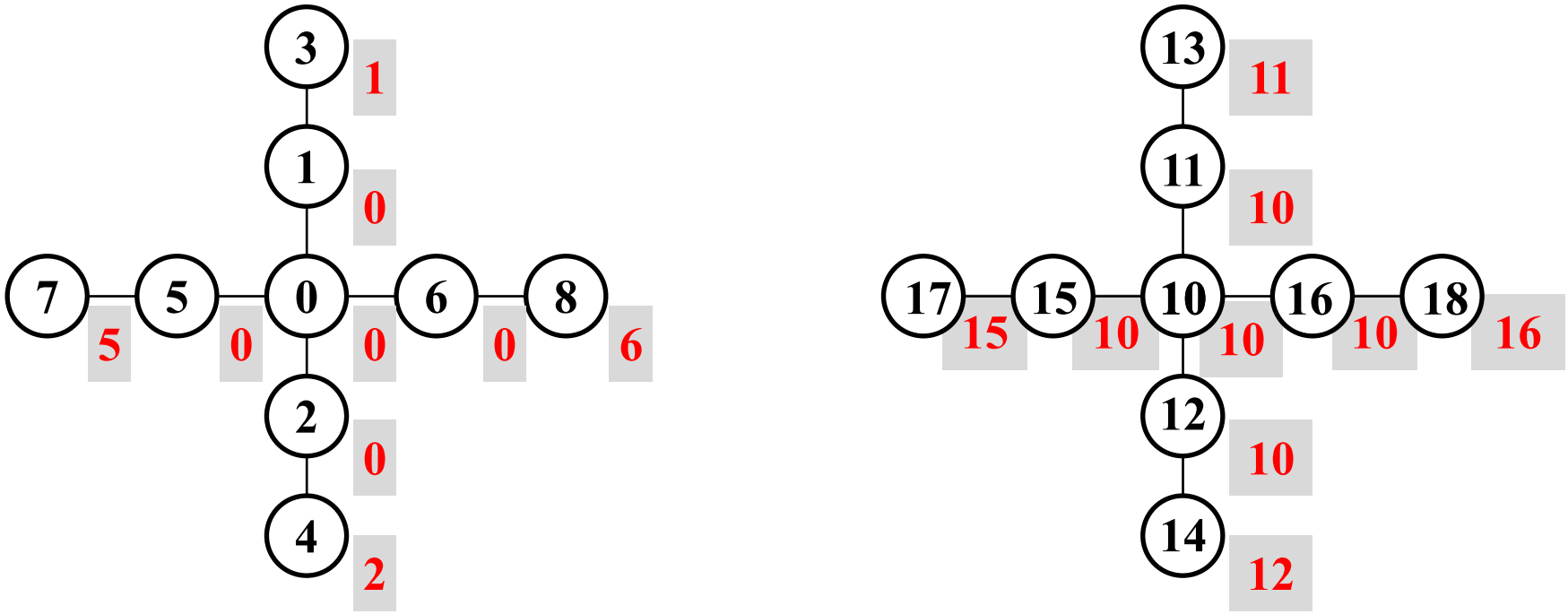
Hash-Min: Connected Components



Superstep 1

Google's Pregel

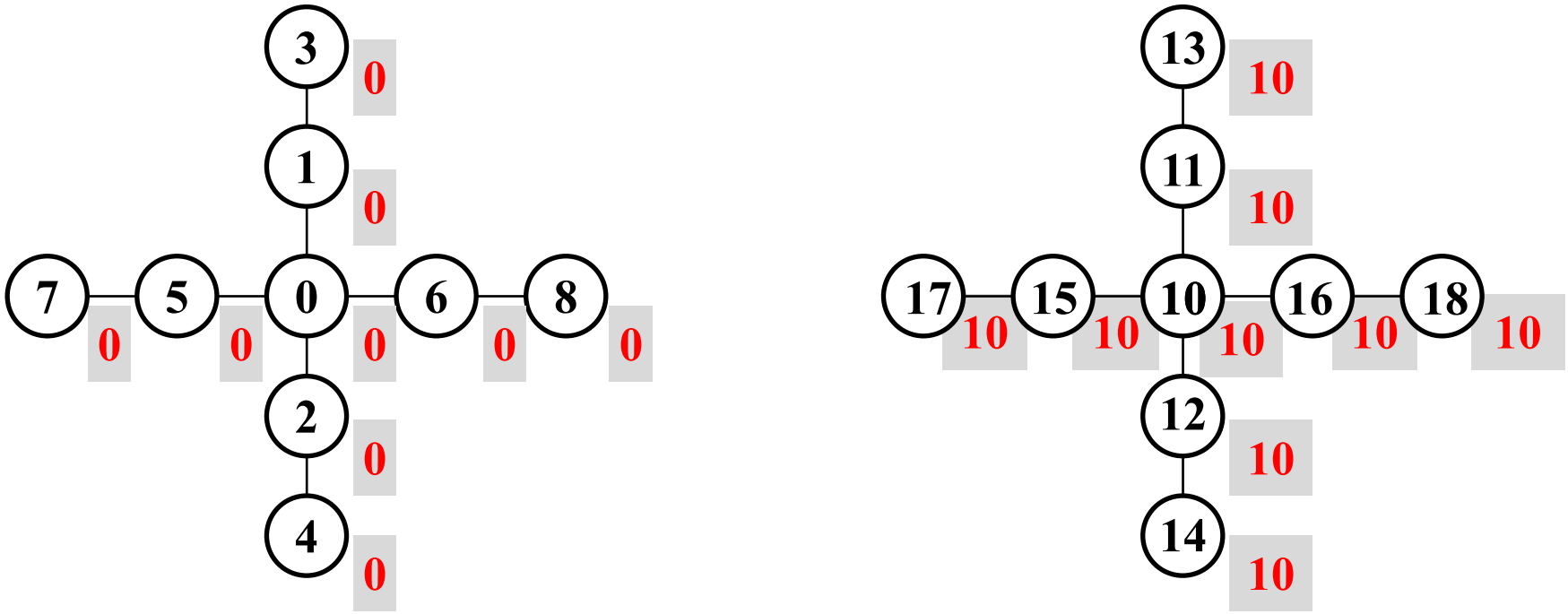
Hash-Min: Connected Components



Superstep 2

Google's Pregel

Hash-Min: Connected Components



Superstep 3

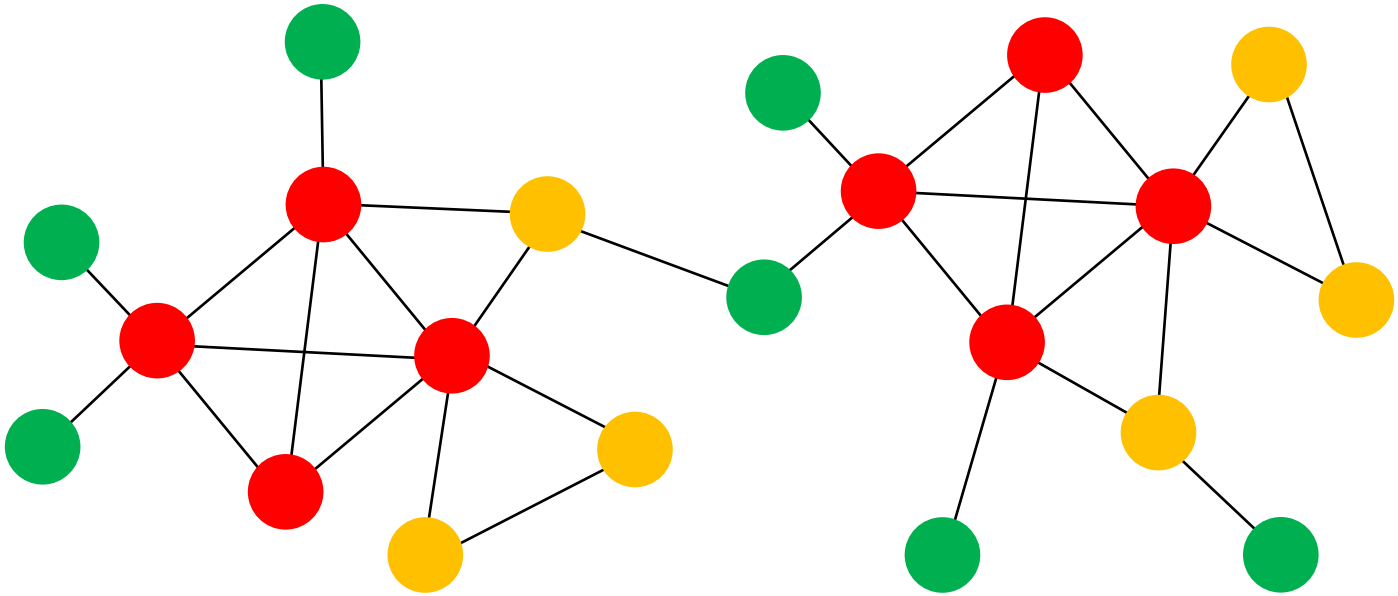
Google's Pregel

Computing *k*-Core

$k = 3$

v.compute($messages$)

Delete received neighbors
If *v.degree* < *k*
    delete *v*
    send "*v*" to neighbors
Vote to halt



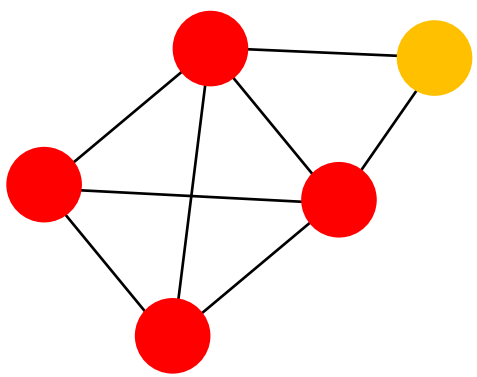
Google's Pregel

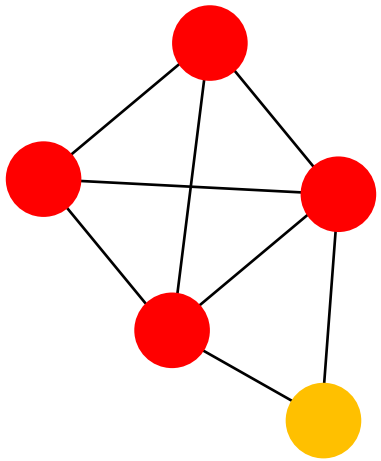
Computing *k*-Core

$k = 3$

v.compute($messages$)

Delete received neighbors
If *v.degree* < *k*
    delete *v*
    send "*v*" to neighbors
Vote to halt





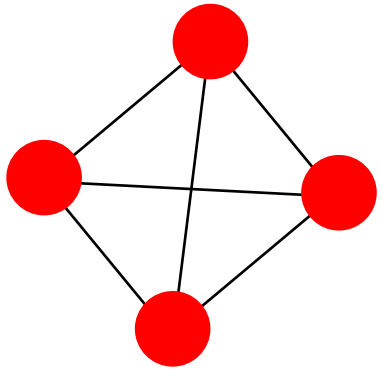
Google's Pregel

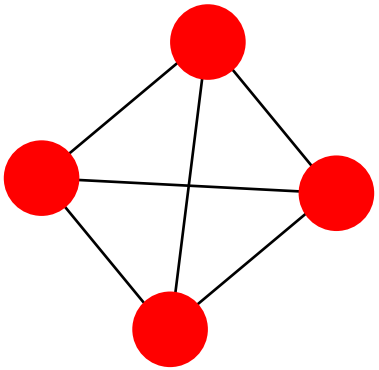
Computing *k*-Core

$k = 3$

v.compute($messages$)

Delete received neighbors
If *v.degree* < *k*
    delete *v*
    send "*v*" to neighbors
Vote to halt



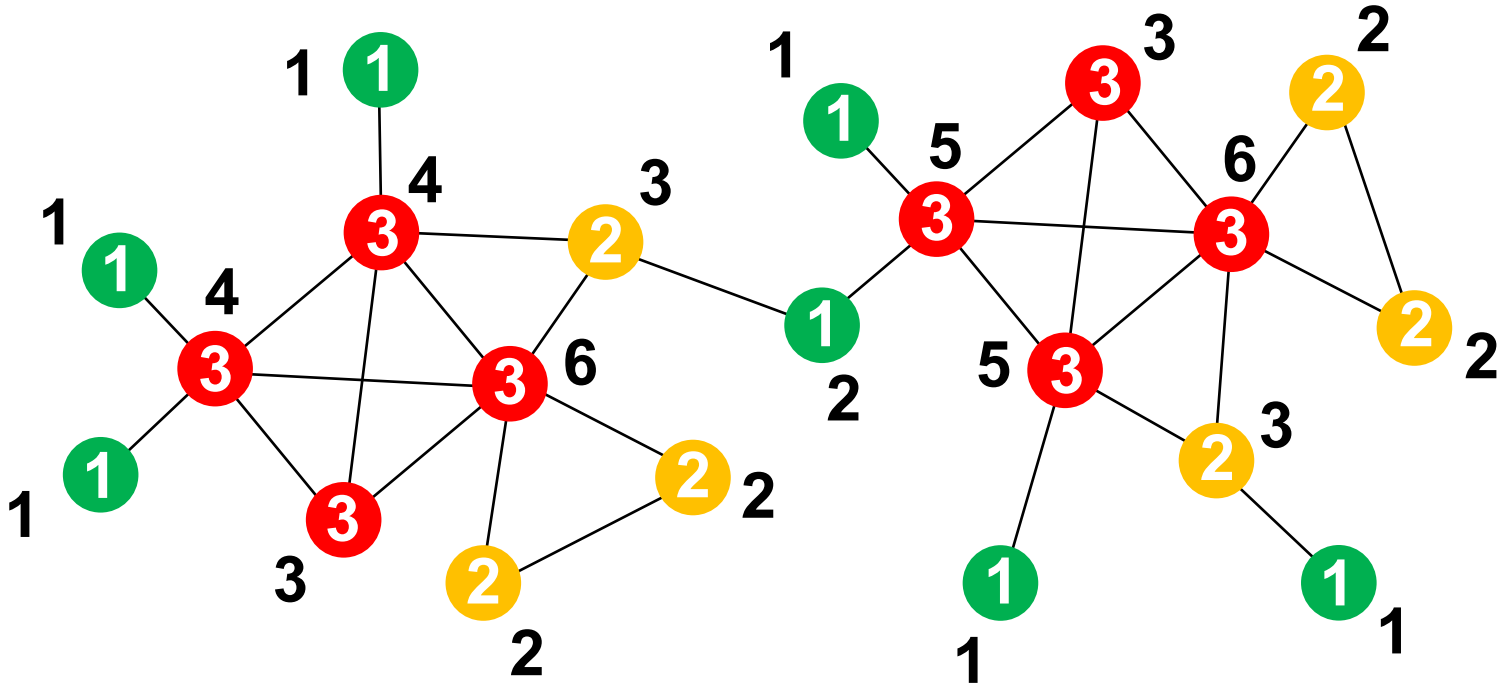


Google's Pregel

Core Decomposition

Computing core number of every vertex

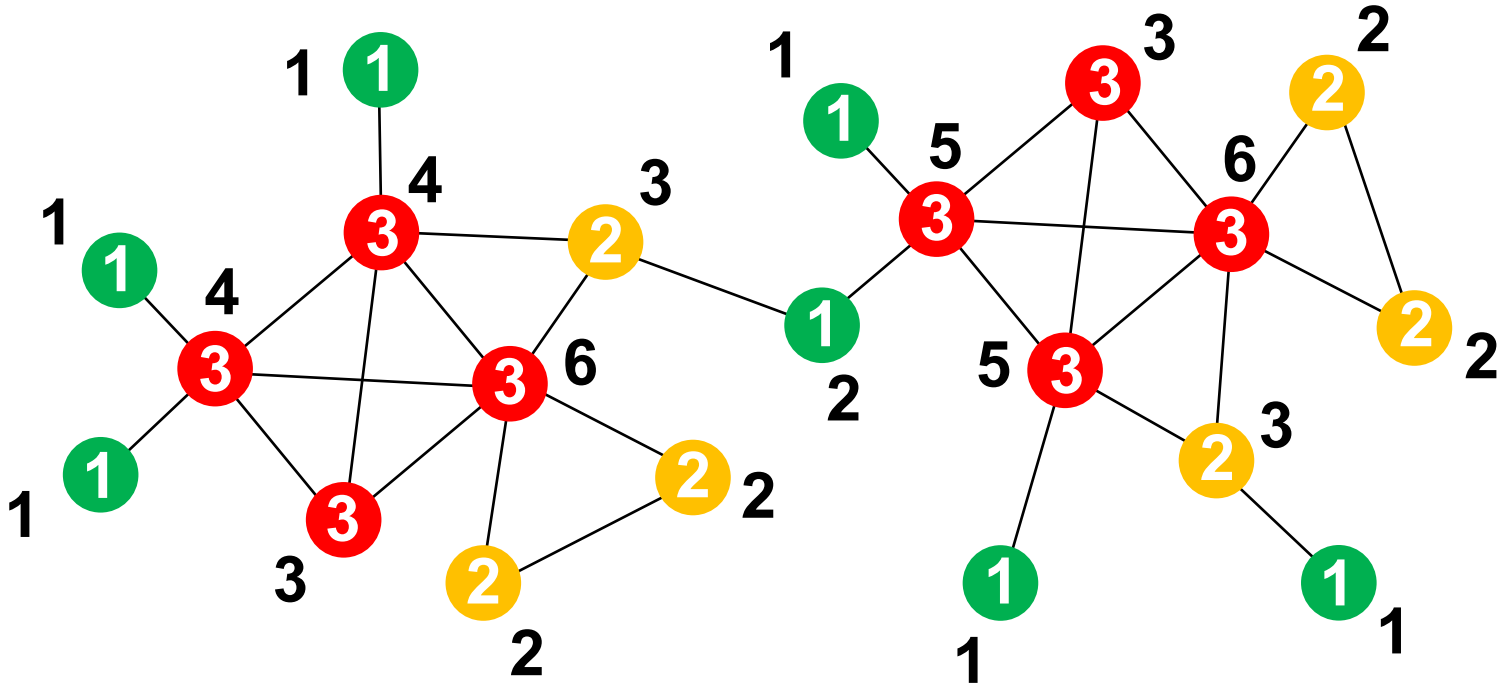
Initializing core numbers with vertex degrees



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

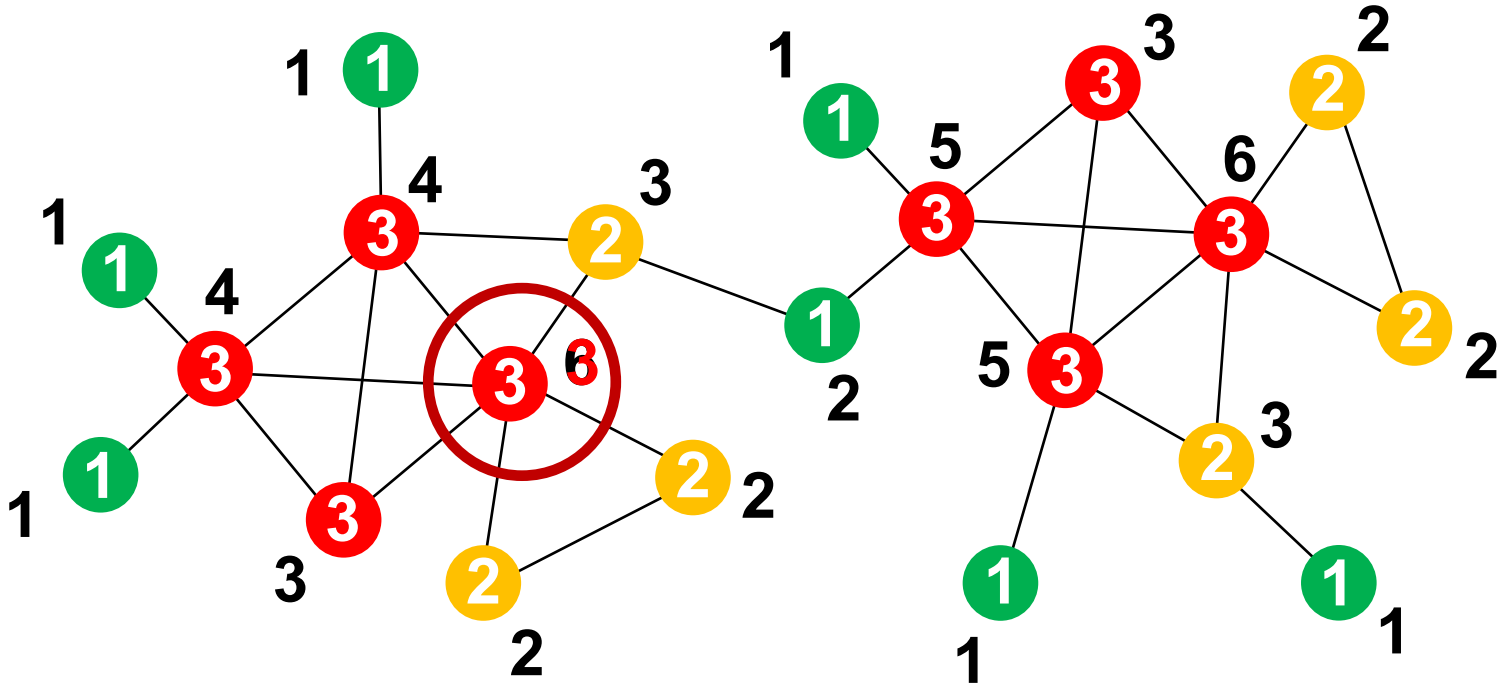
core value = 3

Google's Pregel

Core Decomposition

support = 3

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

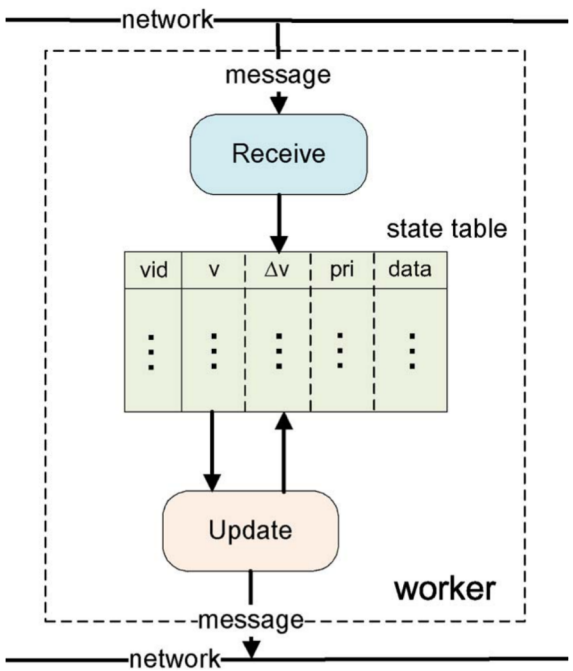


Yanfeng Zhang, Qixin Gao, Lixin Gao, Cuirong Wang:
Maiter: An Asynchronous Graph Processing Framework for Delta-Based Accumulative Iterative Computation. IEEE Trans. Parallel Distributed Syst. 25(8): 2091-2100 (2014)

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:

$$\begin{aligned}
&\texttt{receive:} \begin{cases} \texttt{Whenever receiving } m_j\texttt{,} \\ \Delta\check{v}_j \leftarrow \Delta\check{v}_j \oplus m_j. \end{cases} \\
&\texttt{update:} \begin{cases} \check{v}_j \leftarrow \check{v}_j \oplus \Delta\check{v}_j; \\ \texttt{For any } h\texttt{, if } g_{\{j,h\}}(\Delta\check{v}_j) \neq \mathbf{0}, \\ \quad \texttt{send value } g_{\{j,h\}}(\Delta\check{v}_j) \texttt{ to } h; \\ \Delta\check{v}_j \leftarrow \mathbf{0}, \end{cases}
\end{aligned} \tag{9}$$

Yanfeng Zhang, Qixin Gao, Lixin Gao, Cuirong Wang:
Maiter: An Asynchronous Graph Processing Framework for Delta-Based Accumulative Iterative Computation. IEEE Trans. Parallel Distributed Syst. 25(8): 2091-2100 (2014)

Maiter

DAIC

- Unlike GraphLab, DAIC has exactness guarantee

Table 1: A list of DAIC algorithms

algorithm	$g_{\{i,j\}}(x)$	$\oplus$	v_j^0	Δv_j^1
SSSP	$x + A(i,j)$	min	∞	$0\ (j = s)$ or $\infty\ (j \neq s)$
Connected Components	$A(i,j) \cdot x$	max	-1	j
PageRank	$d \cdot A(i,j) \cdot \frac{x}{\lvert N(j)\rvert}$	$+$	0	$1 - d$
Adsorption	$p_i^{cont} \cdot A(i,j) \cdot x$	$+$	0	$p_j^{inj} \cdot I_j$
HITS (*authority*)	$d \cdot A(i,j) \cdot x$	$+$	0	1
Katz metric	$\beta \cdot A(i,j) \cdot x$	$+$	0	$1\ (j = s)$ or $0\ (j \neq s)$
Jacobi method	$-\frac{A_{ji}}{A_{jj}} \cdot x$	$+$	0	$\frac{b_j}{A_{jj}}$
SimRank	$\frac{C \cdot A(i,j)}{\lvert I(a)\rvert\lvert I(b)\rvert} \cdot x$	$+$	$\lvert I(a) \cap I(b)\rvert\ (a \neq b)$ or $1\ (a = b)$	$\frac{\lvert I(a)\rvert\lvert I(b)\rvert}{C}\ (a \neq b)$ or $0\ (a = b)$
Rooted PageRank	$A(j,i) \cdot x$	$+$	0	$1\ (j = s)$ or $0\ (j \neq s)$

Chronicle of BD Systems

Each round of shuffling means data go through HDFS once

MapReduce papers explode !!! (MapReduce Hype)

Where am I ?

Giraph born

Incubation period …

2010

2012

2009

2013

Pregel born

GraphLab GraphChi (OSDI'12)

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

Spark (NSDI'12)

Chronicle of BD Systems

Do Pregel !

Ph.D.
HKUST

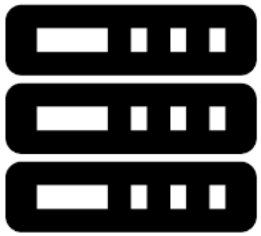
Postdoc @CUHK

2009 — 2013 — 2014 — 2016

Querying and Indexing Geo-Spatial Data

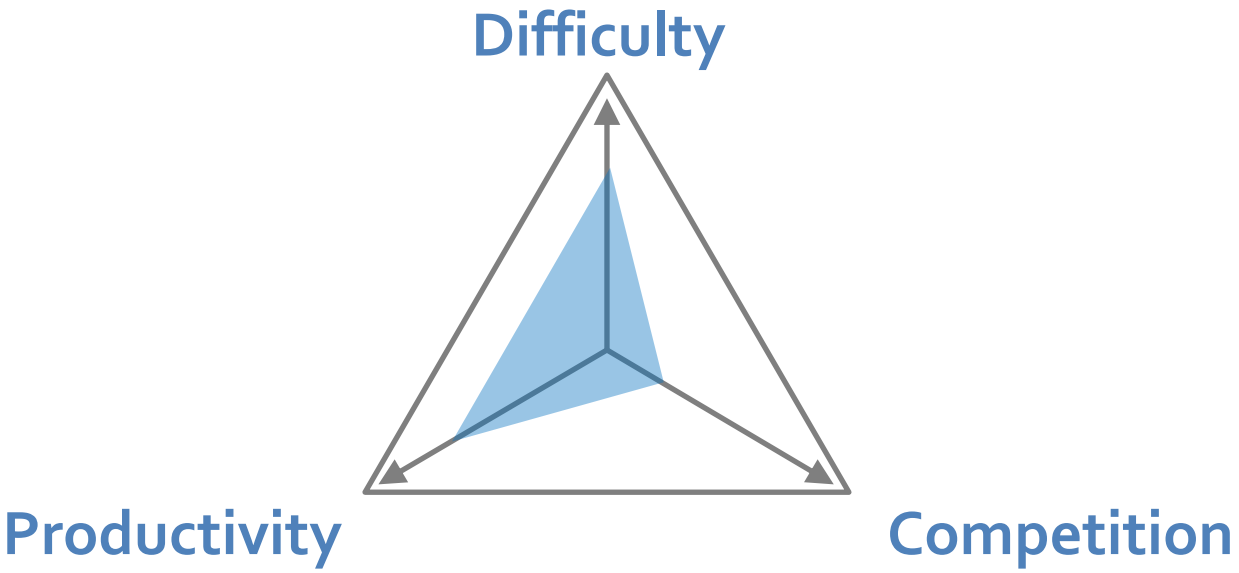
Data Uncertainty

HK 2015 Young Scientist Award



Pregel Research Back in 2013

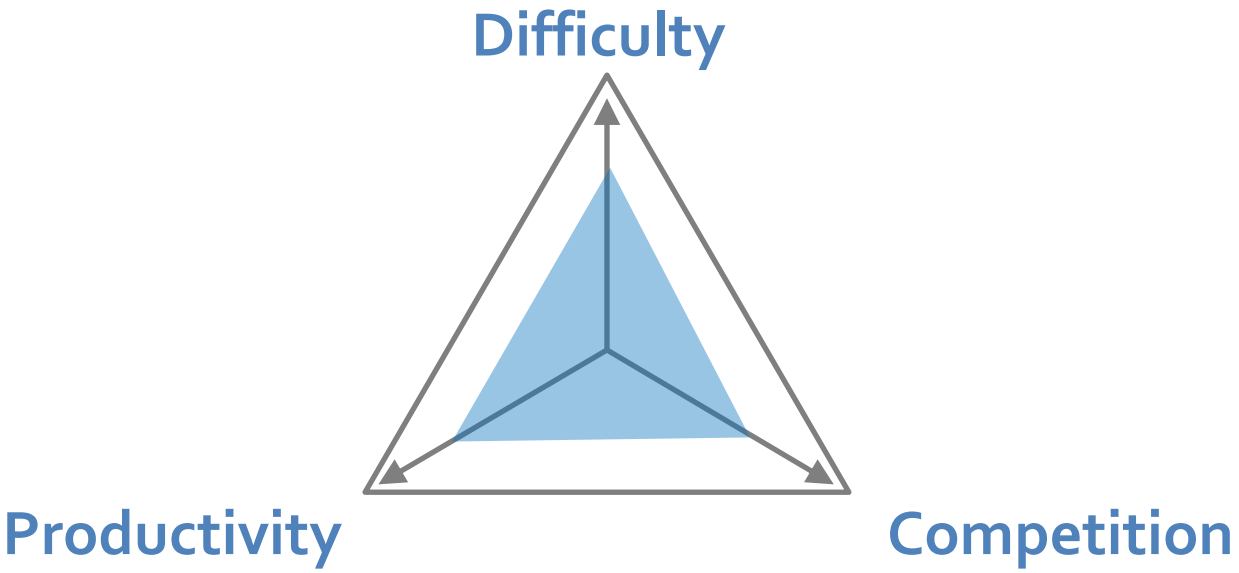
Difficulty – Productivity – Competition



Pregel Research Now

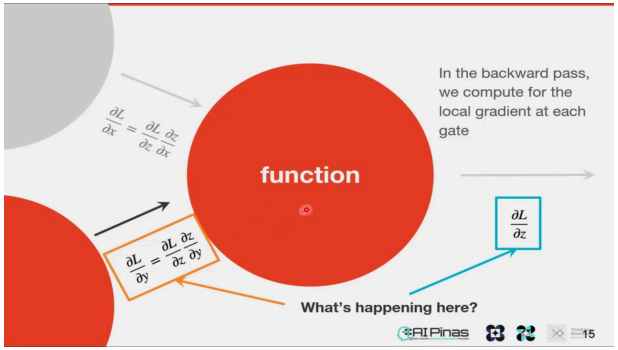
Many opportunities already explored, tools are useful

Difficulty – Productivity – Competition



Deep Learning Now

Just for Fun…



TensorFlow > API > TensorFlow Core v2.3.0 > Python

tf.keras.layers.ConvLSTM2D

TensorFlow 1 version

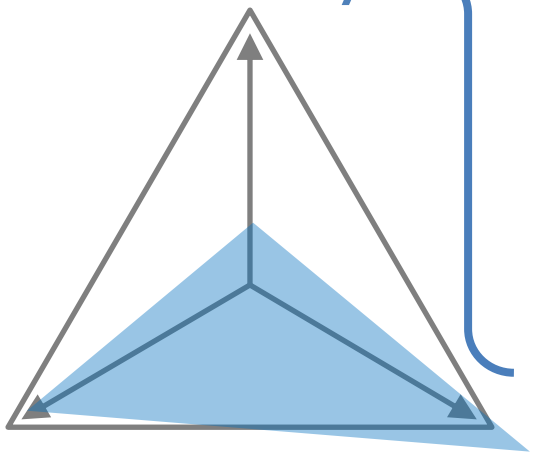
View source on GitHub

TensorFlow > Resources > Addons > API

tfa.losses.SigmoidFocalCrossEntropy

View source on GitHub

Implements the focal loss function.



Kaiming He

FOLLOW

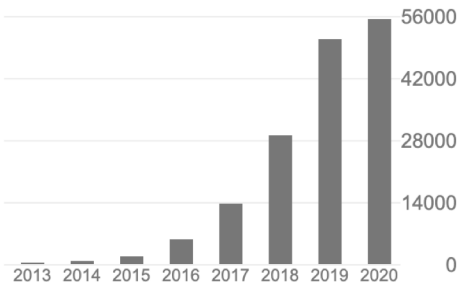
Research Scientist, Facebook AI Research (FAIR)
Verified email at fb.com - Homepage

Computer Vision Machine Learning

TITLE	CITED BY	YEAR
Deep Residual Learning for Image Recognition K He, X Zhang, S Ren, J Sun Computer Vision and Pattern Recognition (CVPR), 2016	61575	2016
Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks S Ren, K He, R Girshick, J Sun Neural Information Processing Systems (NIPS), 2015	23544	2015

Cited by

	All	Since 2015
Citations	160156	157729
h-index	55	55
i10-index	61	61



paperswithcode.com/sota

Computer Vision

- Semantic Segmentation — 68 benchmarks, 1348 papers with code
- Image Classification — 170 benchmarks, 1185 papers with code
- Object Detection — 139 benchmarks, 971 papers with code
- Image Generation — 122 benchmarks, 473 papers with code
- Pose Estimation — 234 benchmarks, 443 papers with code

See all 916 tasks

Natural Language Processing

- Machine Translation — 51 benchmarks, 941 papers with code
- Language Modelling — 16 benchmarks, 900 papers with code
- Question Answering — 59 benchmarks, 808 papers with code
- Sentiment Analysis — 46 benchmarks, 555 papers with code
- Text Generation — 44 benchmarks, 387 papers with code

See all 338 tasks

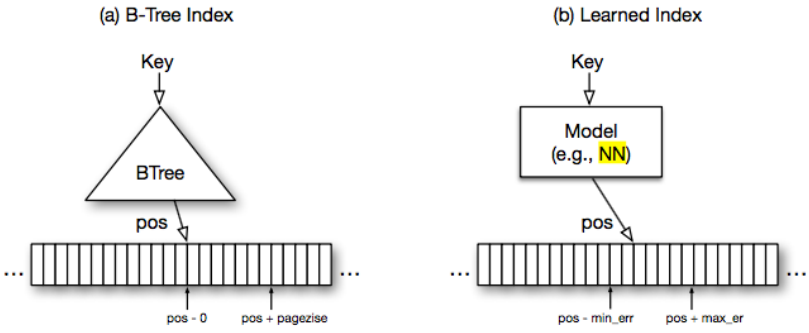
Medical

- Medical Image
- Drug Discovery
- Lesion Segmentation
- Brain Tumor Segmentation
- COVID-19 Diagnosis

ode.com/task/semantic-segmentation n

Abstract:

Database and Artificial Intelligence (AI) can benefit from each oth
more intelligent (AI4DB). For example, traditional empirical datab
estimation, join order selection, knob tuning, index and view sele
requirement for large-scale database instances, various applicati
the cloud. Fortunately, learning-based techniques can alleviate th
techniques can optimize AI models (DB4AI). For example, AI is h
it requires developers to write complex codes and train complica
used to reduce the complexity of using AI models, accelerate AI



Our Works

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

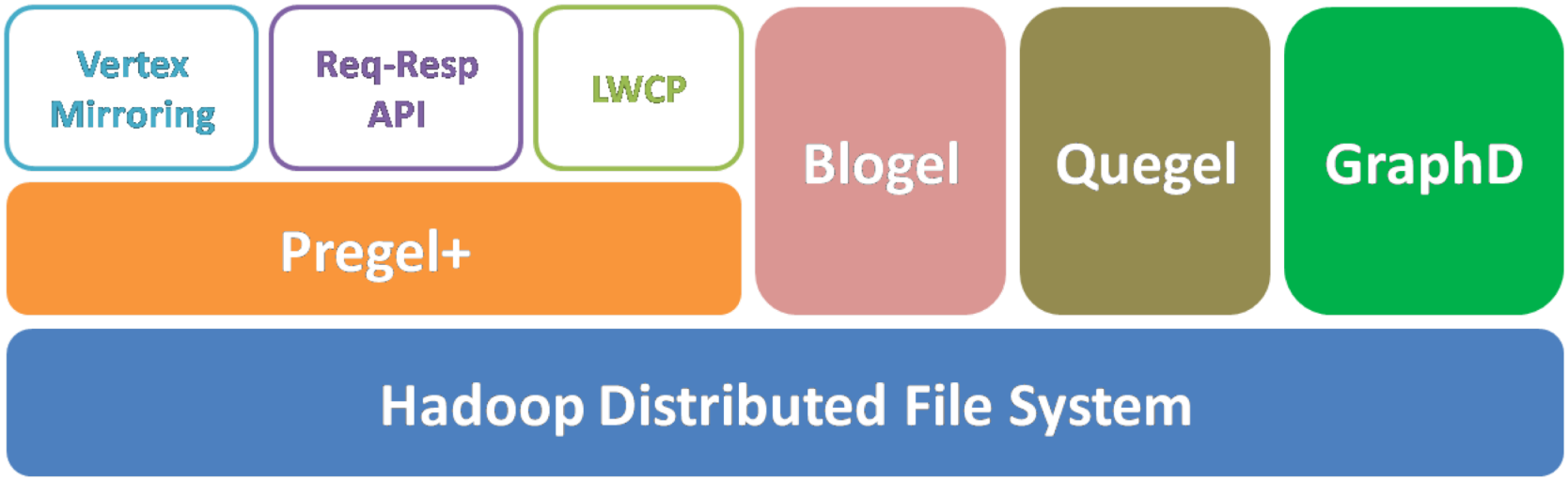
Blogel: graph partitioning + think like a partition (PVLDB'14)

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

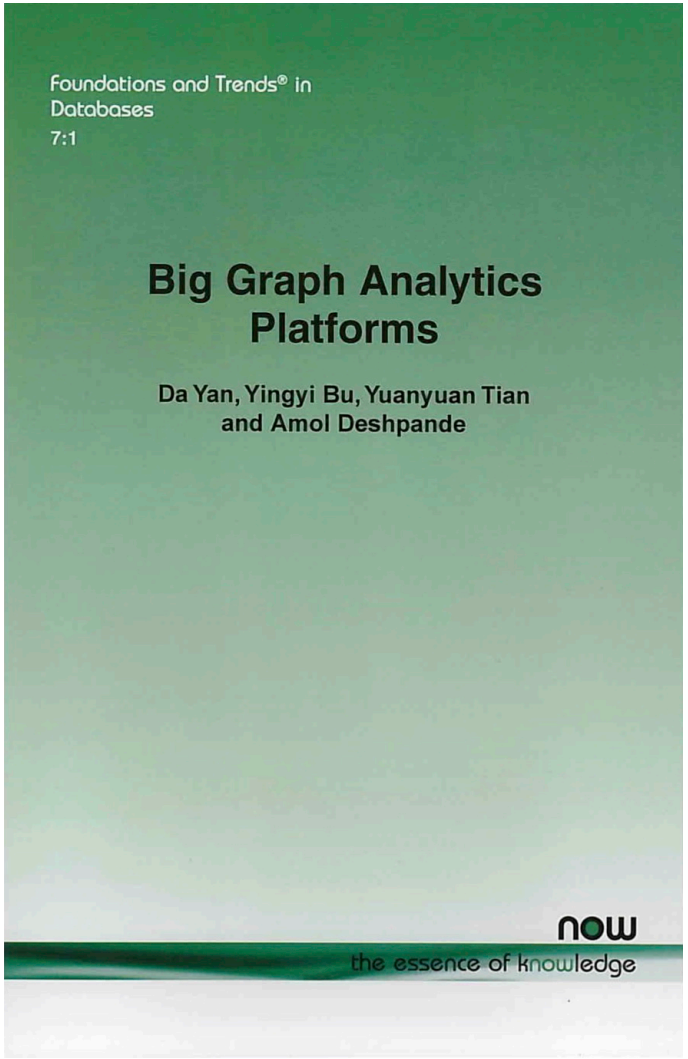


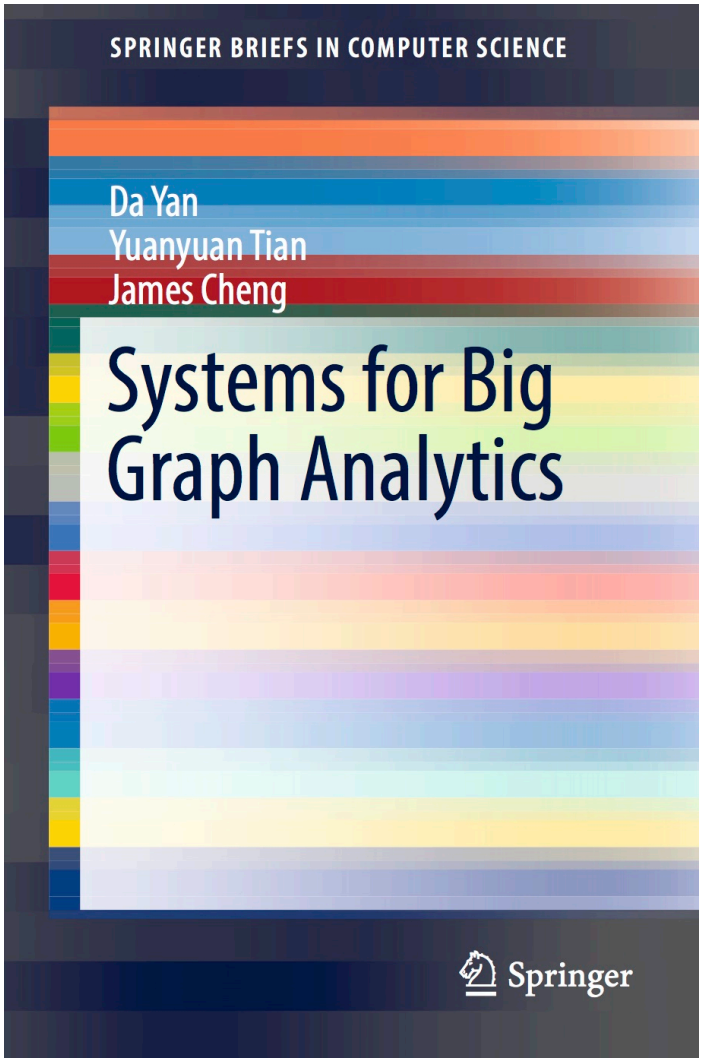
http://www.cse.cuhk.edu.hk/systems/graph/

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	192	2014
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	158	2014
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	109	2015
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	82	2014
Big graph analytics platforms D Yan, Y Bu, Y Tian, A Deshpande Foundations and Trends in Databases 7 (1-2), 1-195	52	2017

Books


Foundations and Trends® in
Databases
7:1
Big Graph Analytics
Platforms
Da Yan, Yingyi Bu, Yuanyuan Tian
and Amol Deshpande
now
the essence of knowledge


SPRINGER BRIEFS IN COMPUTER SCIENCE
Da Yan
Yuanyuan Tian
James Cheng
Systems for Big
Graph Analytics
Springer

Practical Pregel Alogorithms

Pra ... '14]

» T

» I

... tour,

... nts,

Theoretical Complexity Results of Graph Algorithms in PREGEL

ETH zürich

Microsoft Research

- **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

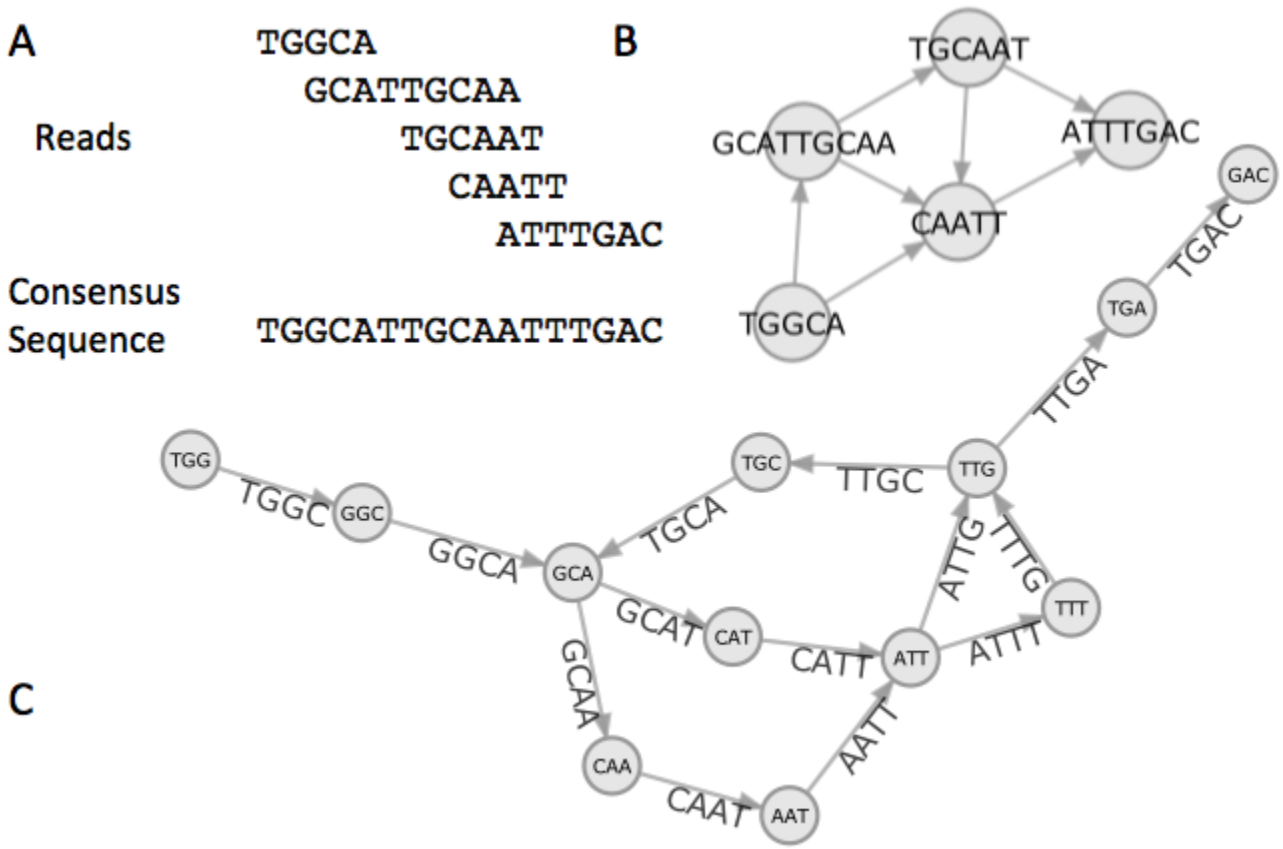
Open Area of Research

Practical PREGEL Algorithms for Massive Graphs [http://www.cse.cuhk.edu.hk] 46/ 185

Practical Pregel Alogorithms

ACM/IEEE TCBB journal version also

Application: Genome Assembly [ICDE'18]

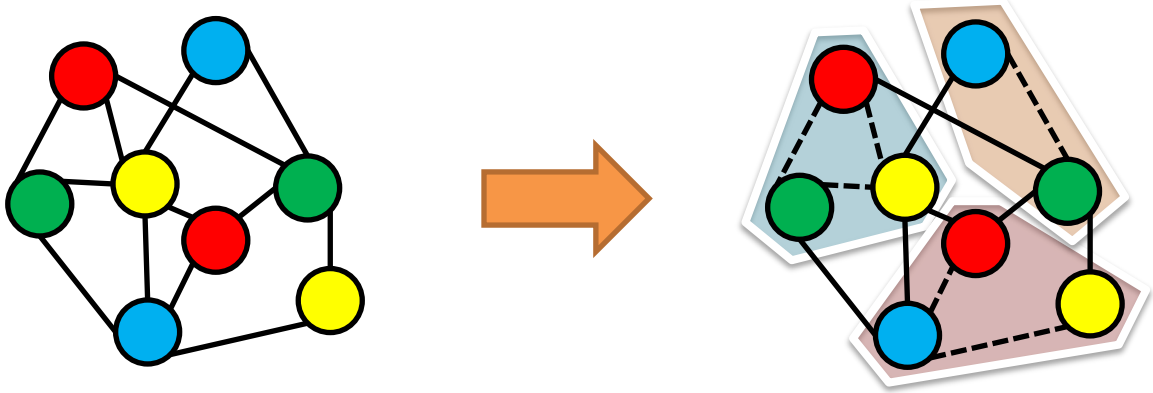


Block-Centric Computation

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:

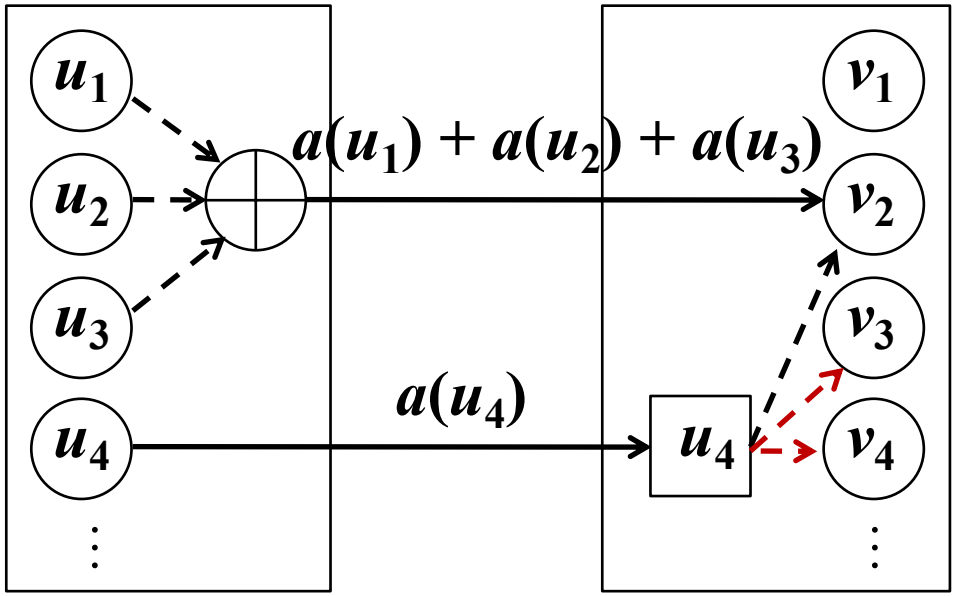
- Blogel is the overall winner. The execution time of Blogel-B is shortest, but Blogel-V is faster when we consider the end-to-end processing including data loading and partitioning (§ 5.1).
- Existing graph processing systems are inefficient over graphs with large diameters, such as the road network (§ 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 # of messages transmitted

- Vertex Mirroring
- Request-Respond Paradigm



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
- » $O(|V| / \#\{machines\})$ 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

Da Yan, Guimu Guo, Md Mashiur Rahman Chowdhury, M. Tamer Özsu, Wei-Shinn Ku, John C. S. Lui:
G-thinker: A Distributed Framework for Mining Subgraphs in a Big Graph. ICDE 2020: 1369-1380

Motivation

Will be presented by

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

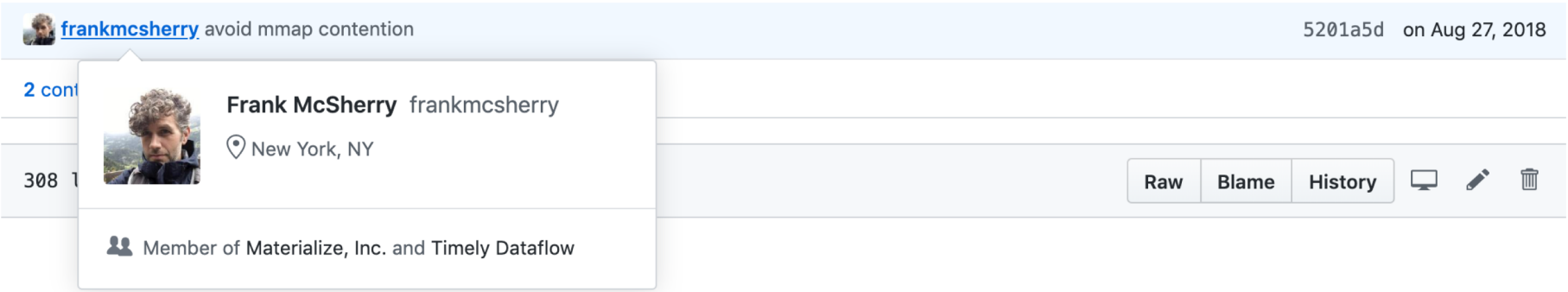
To present in next section

Da Yan, Wenwen Qu, Guimu Guo, Xiaoling Wang:
PrefixFPM: A Parallel Framework for General-Purpose Frequent Pattern Mining. ICDE 2020: 1938-1941

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

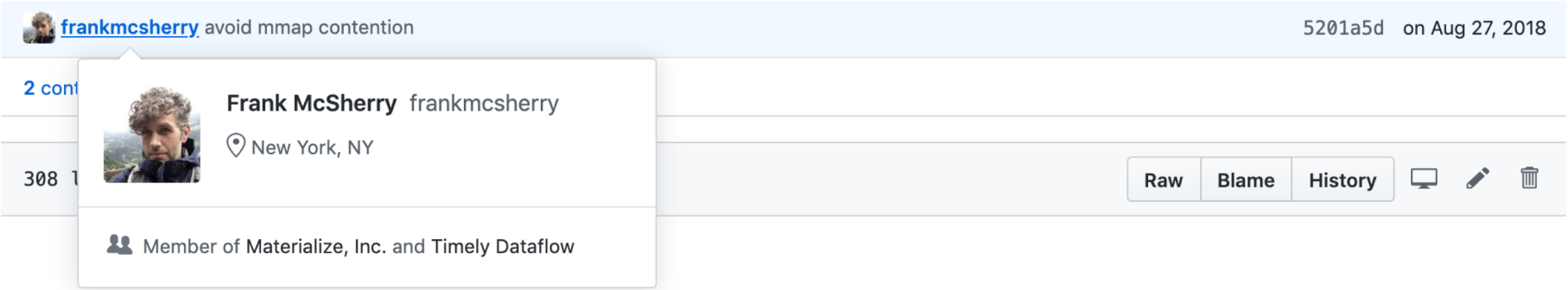
Several years ago Michael Isard, Derek Murray, and I wrote a paper: "Scalability, but at what COST". The premise of the paper was that the current excitement about distributed computation (e.g. Hadoop, Spark) produced implementations that improve when you give them more resources (they "scale") but whose 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')



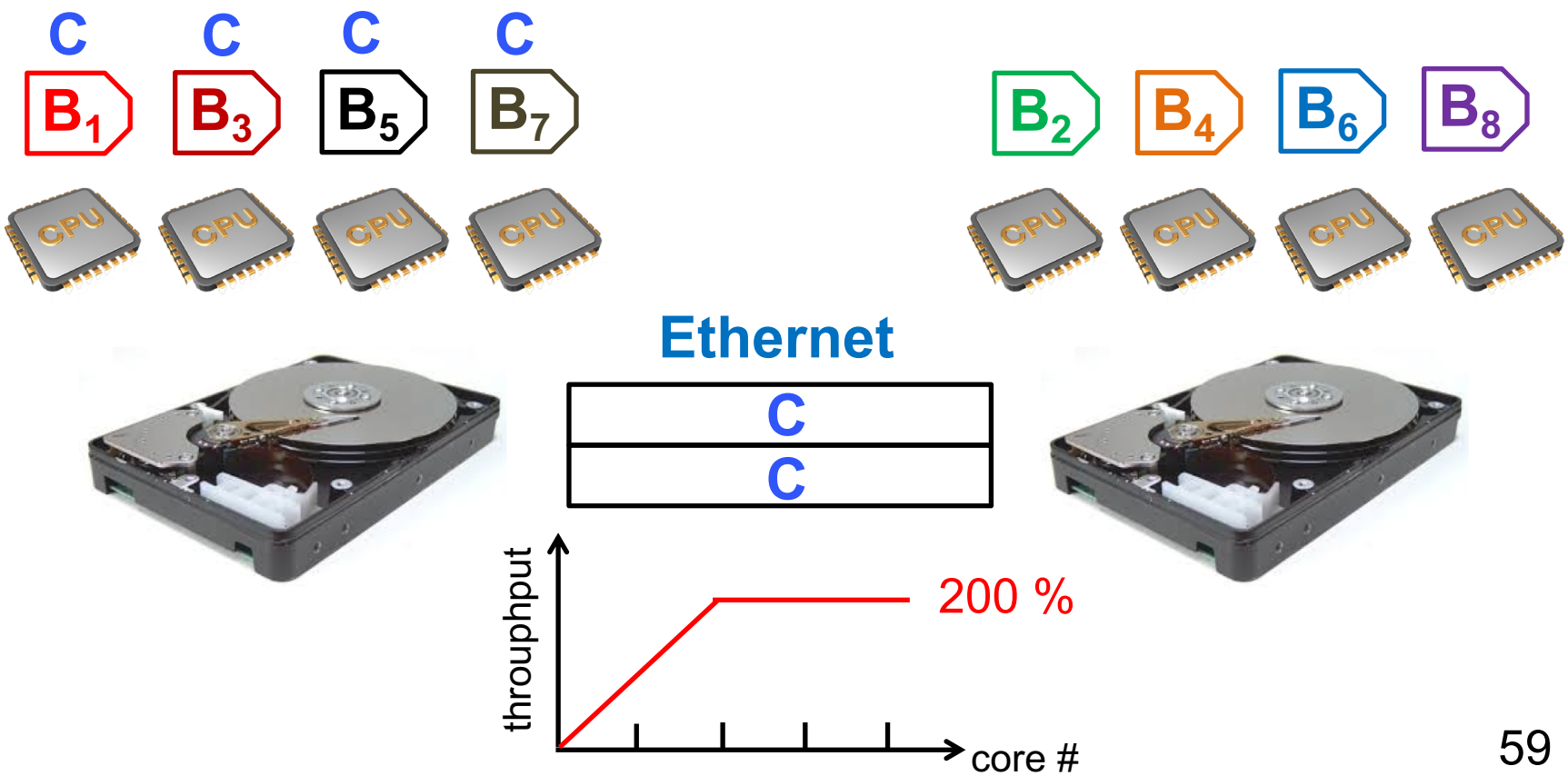
COST in the land of databases **Cont.**

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

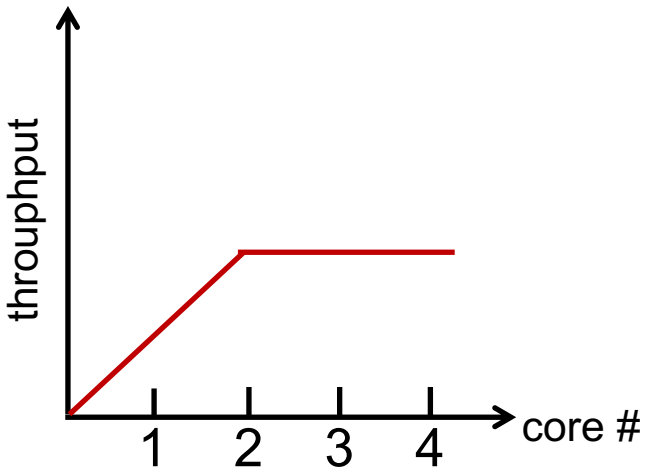
Data-Intensive v.s. Compute-Intensive



Motivation

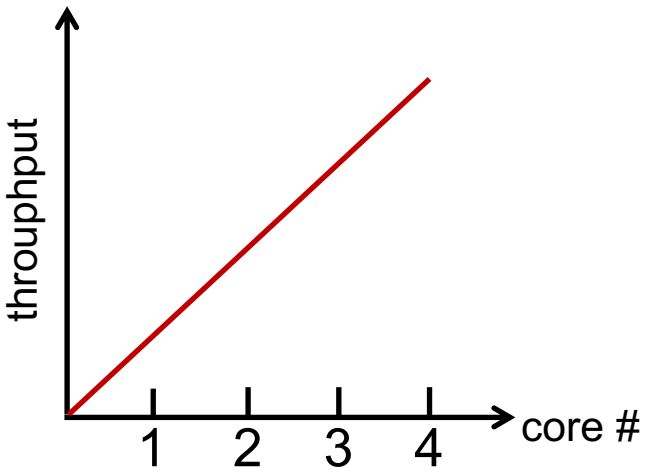
Data-Intensive v.s. Compute-Intensive

Data-intensive
IO-bound



more disks
faster networks
faster disks

Compute-intensive
CPU-bound



more CPUs
more memory

Motivation

Shumo Chu, James Cheng:
Triangle listing in massive networks. ACM Trans. Knowl. Discov. Data 6(4): 17:1-17:32 (2012)

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

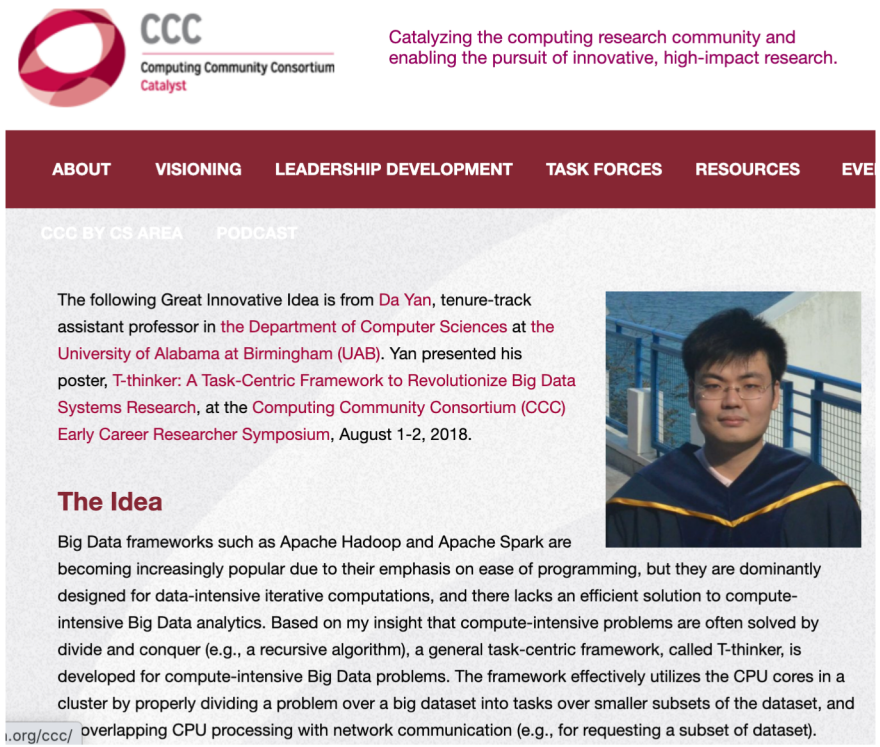
For triangle counting in a large graph that cannot fit in main memory, parallel algorithms that apply the MapReduce framework were proposed recently [Suri and Vassilvitskii 2011]. Their algorithms are exact and do not require to keep the entire input graph in main memory at each individual machine. Our 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.

Da Yan, Guimu Guo, Md Mashiur Rahman Chowdhury, M. Tamer Özsu, John C. S. Lui, Weida Tan: **T-thinker: a task-centric distributed framework for compute-intensive divide-and-conquer algorithms.** PPoPP 2019: 411-412

Motivation

T-thinker: Divide and Conquer

- Task-based: data pulling + computation on top



CCC
Computing Community Consortium
Catalyst

Catalyzing the computing research community and enabling the pursuit of innovative, high-impact research.

ABOUT VISIONING LEADERSHIP DEVELOPMENT TASK FORCES RESOURCES EVE

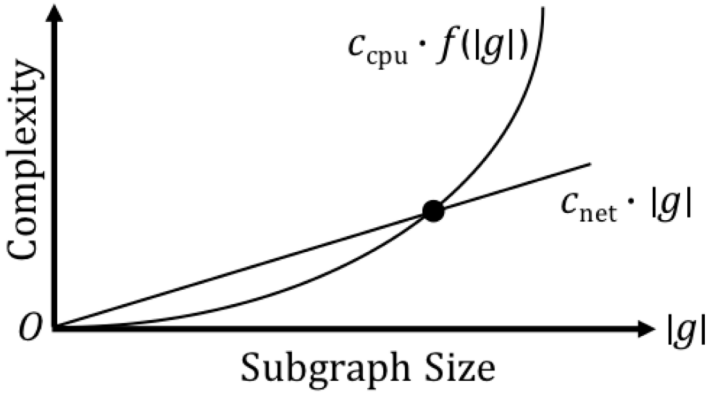
CCC BY CS AREA PODCAST

The following Great Innovative Idea is from Da Yan, tenure-track assistant professor in the Department of Computer Sciences at the University of Alabama at Birmingham (UAB). Yan presented his poster, T-thinker: A Task-Centric Framework to Revolutionize Big Data Systems Research, at the Computing Community Consortium (CCC) Early Career Researcher Symposium, August 1-2, 2018.

The Idea

Big Data frameworks such as Apache Hadoop and Apache Spark are becoming increasingly popular due to their emphasis on ease of programming, but they are dominantly designed for data-intensive iterative computations, and there lacks an efficient solution to compute-intensive Big Data analytics. Based on my insight that compute-intensive problems are often solved by divide and conquer (e.g., a recursive algorithm), a general task-centric framework, called T-thinker, is developed for compute-intensive Big Data problems. The framework effectively utilizes the CPU cores in a cluster by properly dividing a problem over a big dataset into tasks over smaller subsets of the dataset, and overlapping CPU processing with network communication (e.g., for requesting a subset of dataset).

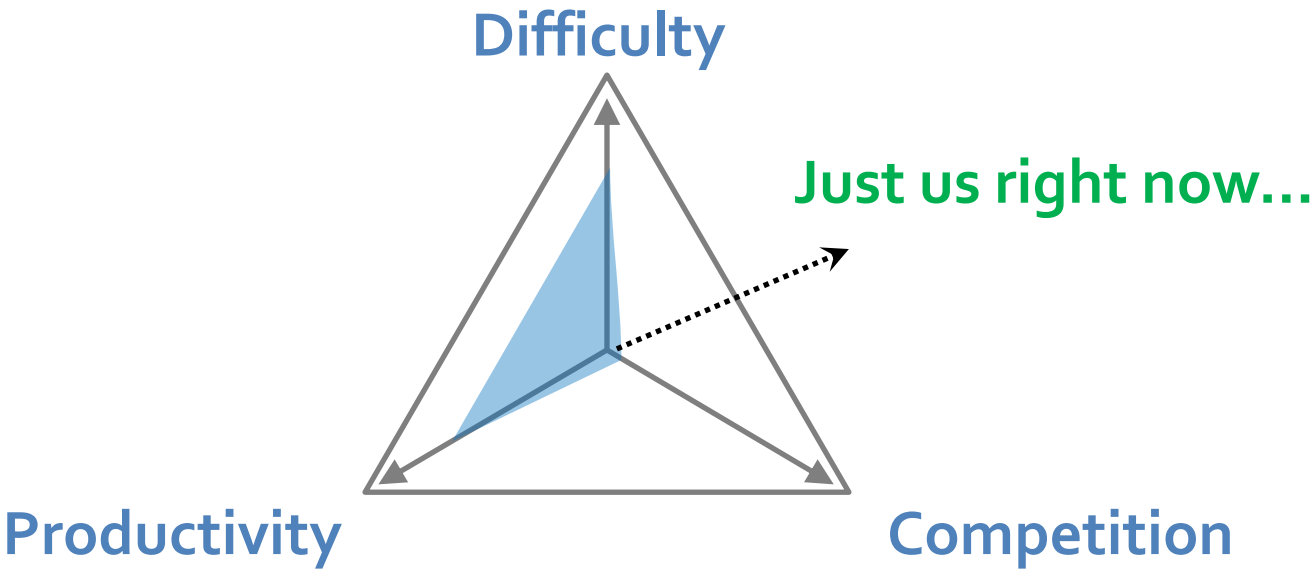
.org/ccc/



T-thinker Research Now

Difficulty – Productivity – Competition

We have queued many works to be done !

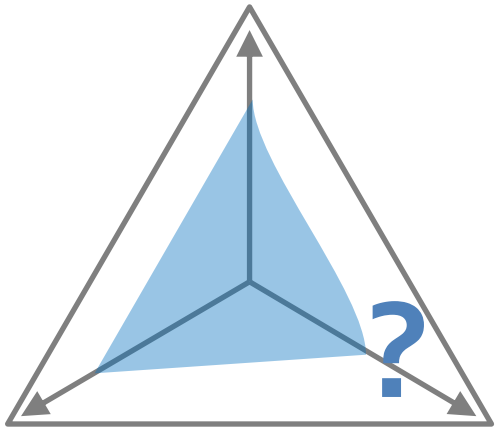


Five Years from Now?

Google Scholar **Time to make impact!**

Cited by

	All	Since 2020
Citations	+????	+????
h-index	+??	+??
i10-index	+??	+??

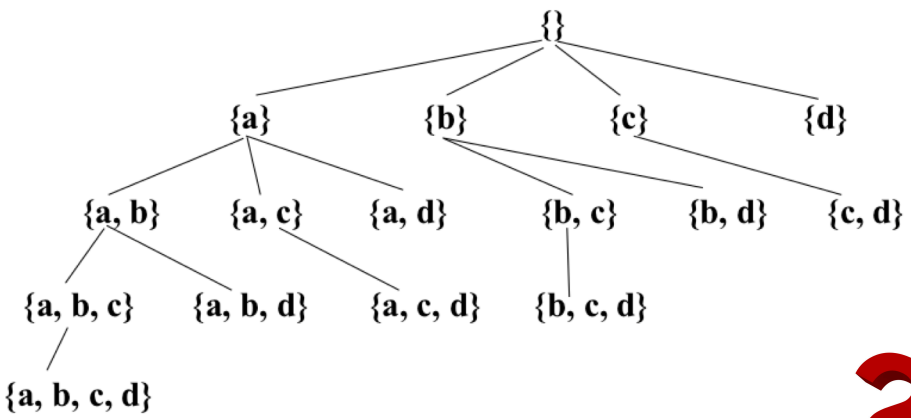


Competition

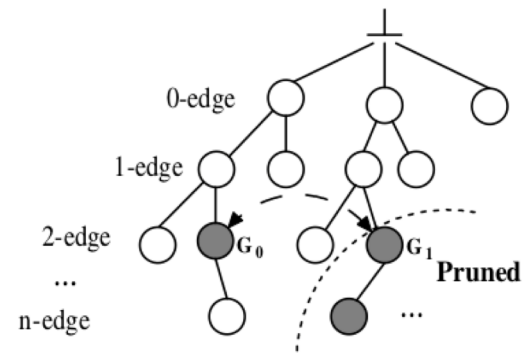
Motivation

Programming Simplicity Over-Emphasized

pattern-to-instance



instance-to-pattern



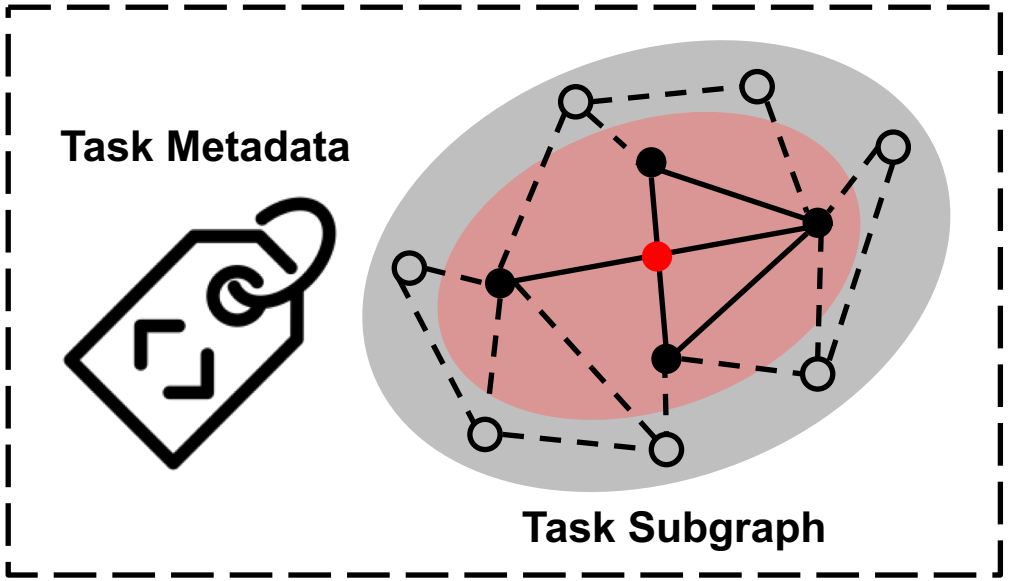
Embedding-Centric: Arabesque (SOSP'15), RStream (OSDI'18)

G-thinker

Will be presented by

Task-Centric Programming Interface

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



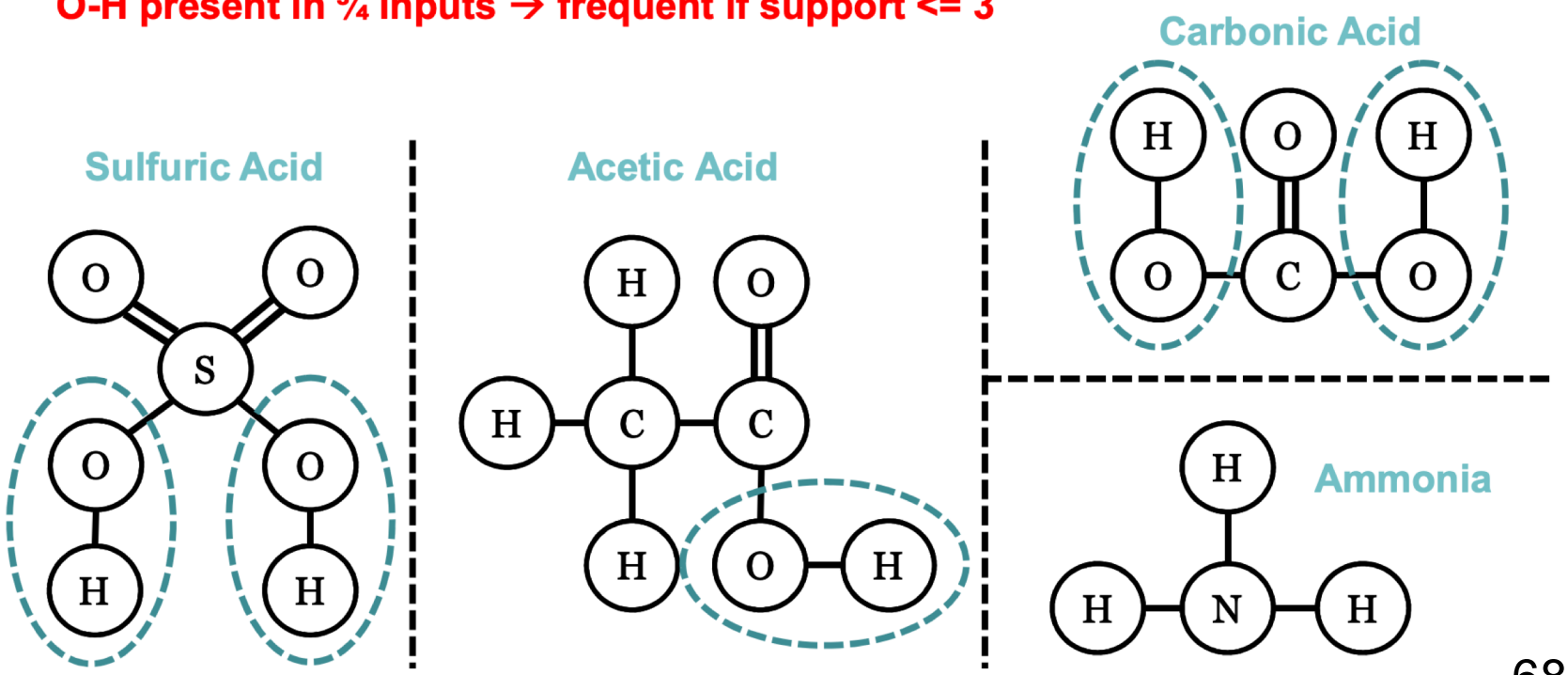
Overview

- 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 ¾ inputs → frequent if support <= 3



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 >= a support threshold

Market Bucket Transactions

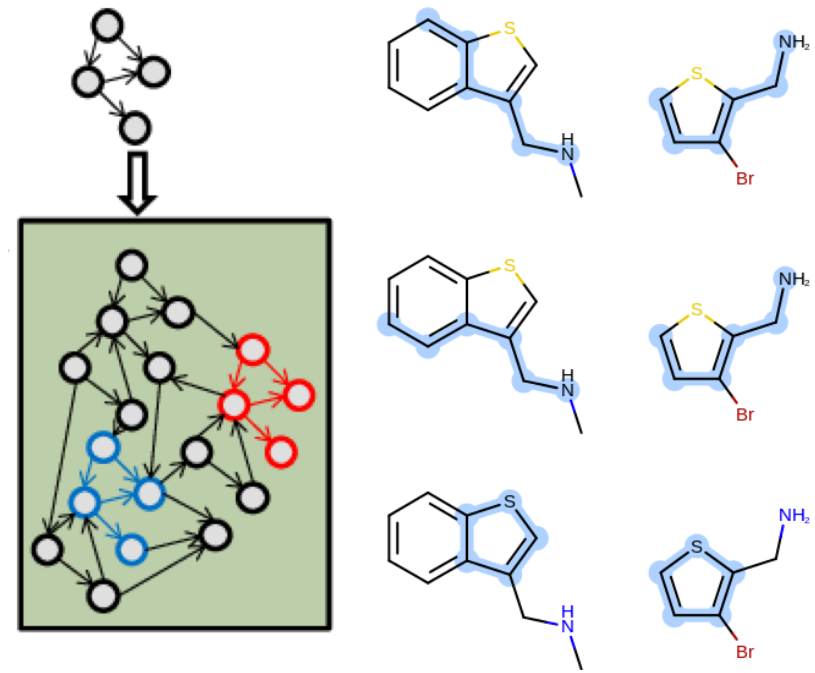
Transaction ID	Items Bought
1	Bread, Milk
2	Bread, Diaper, Beer, Eggs
3	Milk, Diaper, Beer, Coke
4	Bread, Milk, Diaper, Beer ✓
5	Beer, Milk, Diaper, Coke
6	Beer, Milk, Diaper, Bread ✓

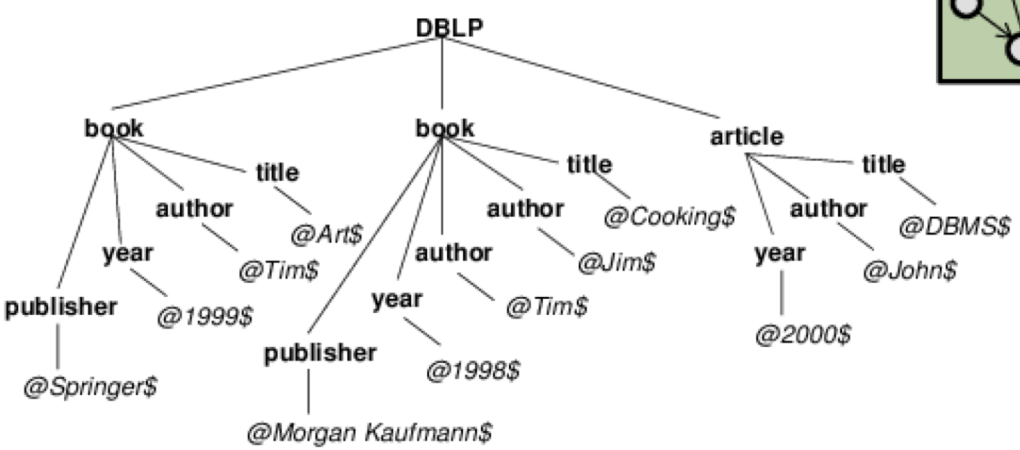
Support (Count): σ{Milk, Bread, Diaper} = 2

Category II: Frequent Patterns

Types of Patterns

- Itemset
- Subsequence
- Subgraph
- Subtree

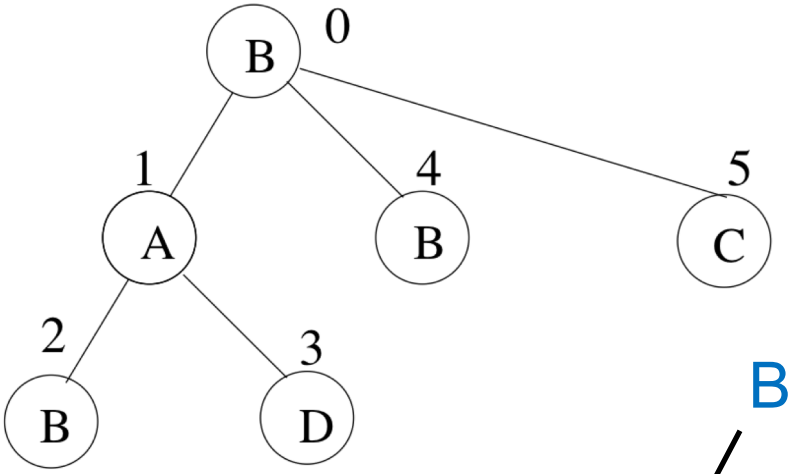


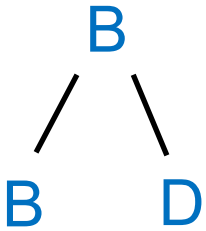


Category II: Frequent Patterns

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

109 1,639 eReader PDF

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.

2008

Haoyuan Li, Yi Wang, Dong Zhang, Ming Zhang, Edward Y. Chang: **Pfp: parallel fp-growth for query recommendation.** RecSys 2008: 107-114

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
  - Subsequence
  - Subgraph
  - Subtree
  - …

Itemset $\xrightarrow{\text{order-free}}$ FP-tree $\xrightarrow{\text{parallel}}$ PFP

Subsequence, Subgraph, Subtree, … } Prefix Projection $\xrightarrow{\text{parallel}}$ PrefixPFM

order matters

Depth-First Pattern Search/Recursion !

Category II: Frequent Patterns

PrefixSpan (Sequential Pattern Mining)

SID	Sequence
s_1	$ABCBC$
s_2	$BABC$
s_3	AB
s_4	BC

(a) D

Transaction DB

$\xrightarrow{A}$

SID	Sequence
s_1	$\_BCBC$
s_2	$\_BC$
s_3	$\_B$

(b) $D|_A$

Projected DB

$\xrightarrow{B}$

SID	Sequence
s_1	$\_CBC$
s_2	$\_C$
s_3	$\_$

(c) $D|_{AB}$

$\xrightarrow{C}$

SID	Sequence
s_1	$\_BC$
s_2	$\_$

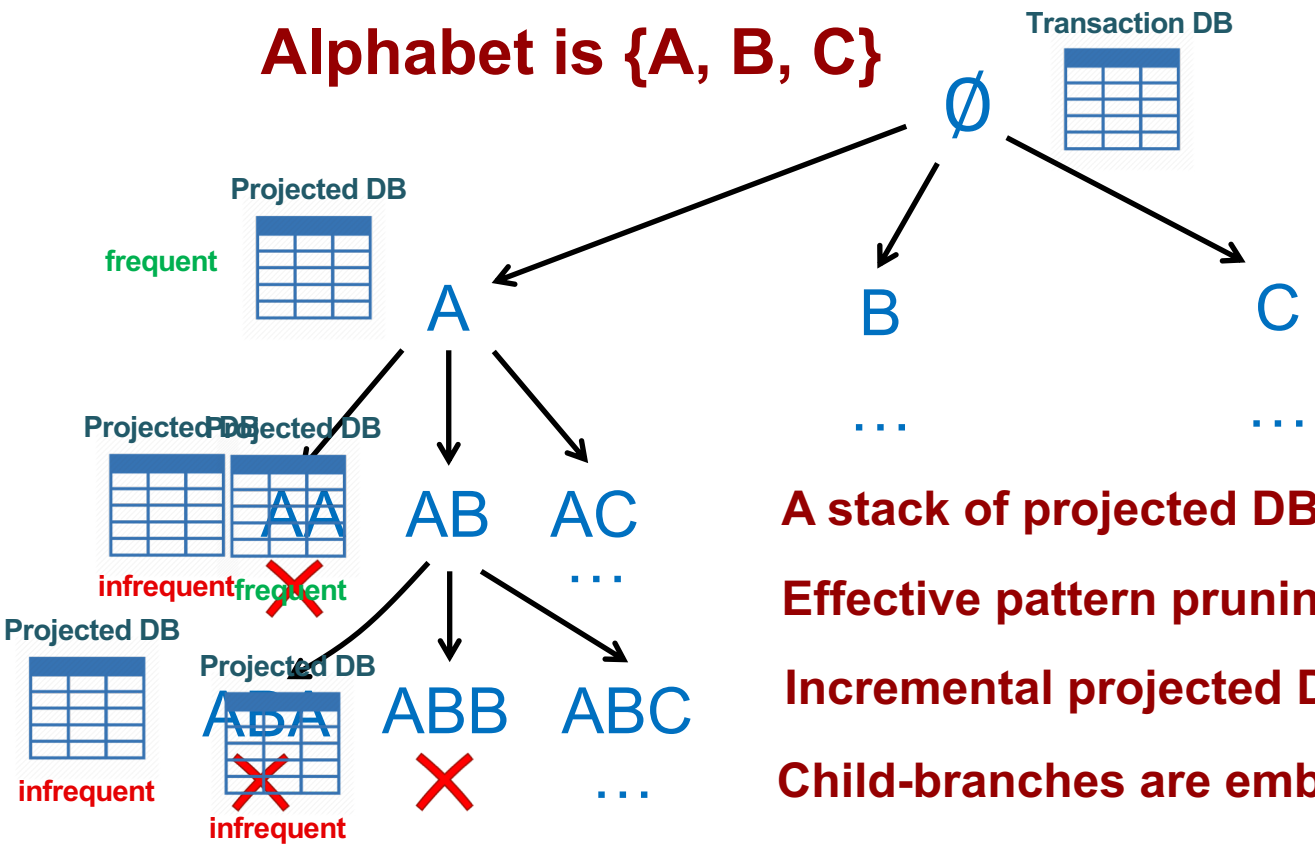
(d) $D|_{ABC}$

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

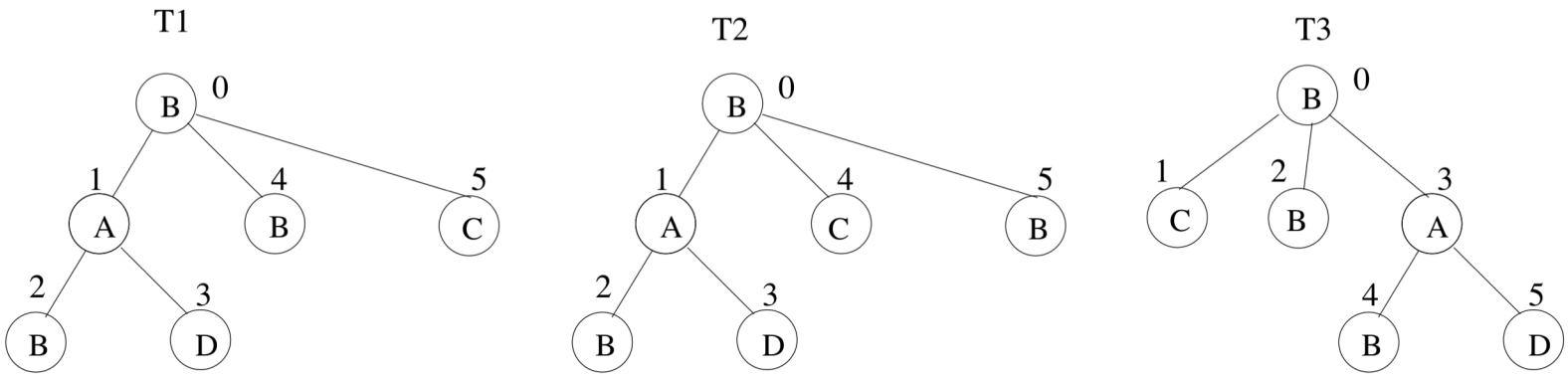
Child-branches are embarrassingly parallel !!!

Prefix Projection

From Sequences to Graphs

Duplicated patterns for different sequences

Encode a tree into a sequence



BAB$D$$BC BAB$D$$CB BCBAB$D$$

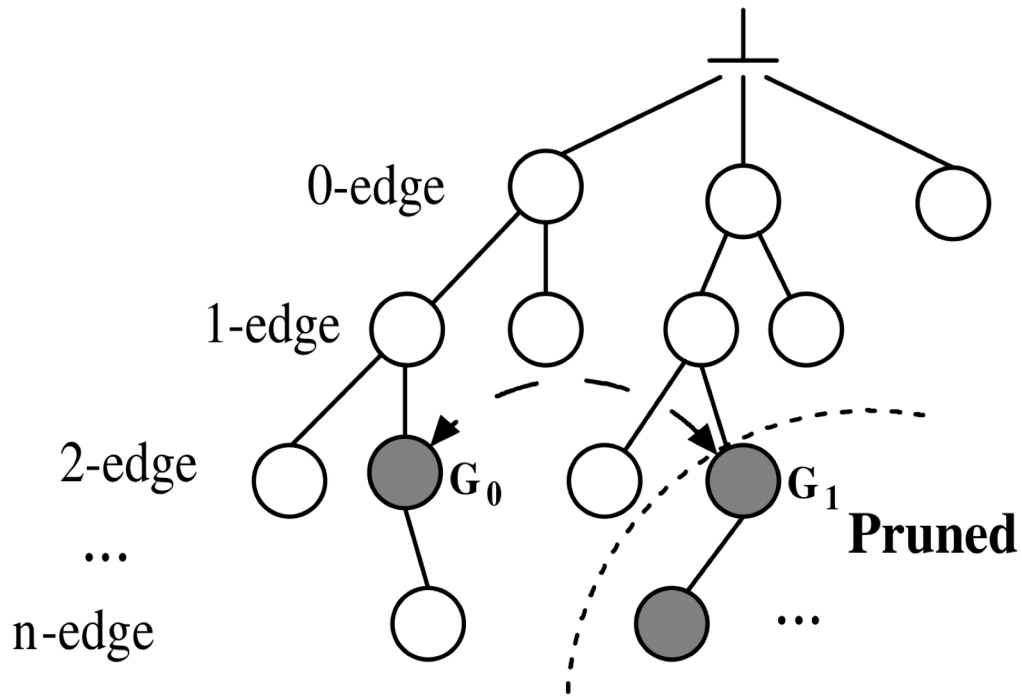
canonical encoding

Mohammed Javeed Zaki:
Efficiently Mining Frequent Embedded Unordered Trees. Fundam. Inform. 66(1-2): 33-52 (2005)

Prefix Projection

From Sequences to Graphs

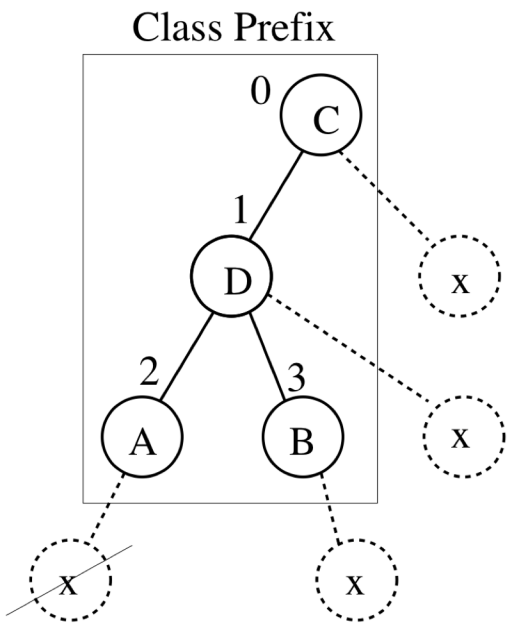
Deduplicate by canonical encoding



Prefix Projection

From Sequences to Graphs

Truncated pattern growth



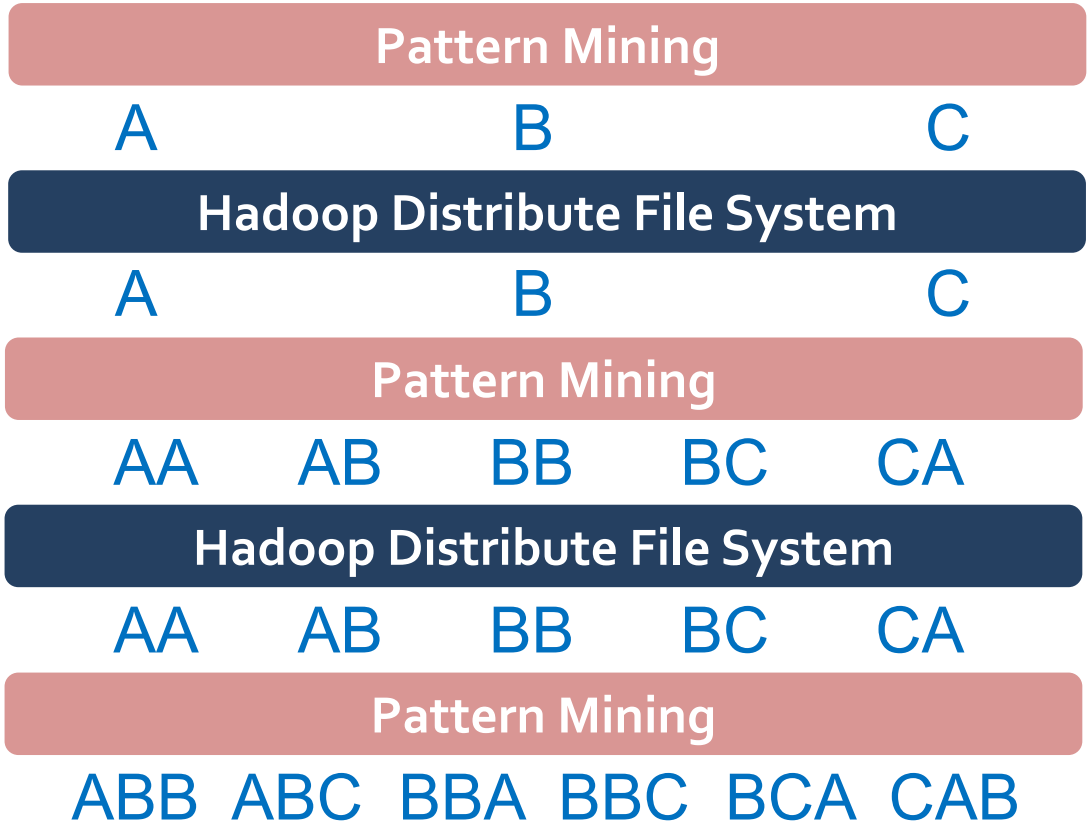
Related Work

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



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,
100071, China

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

Itemset Patterns Solved

ABSTRACT

Frequent itemset mining (FIM) is a useful tool for discovering frequently co-occurrent 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

...ped parallel algorithm on Web data to support *query recommendation* (or *related search*).

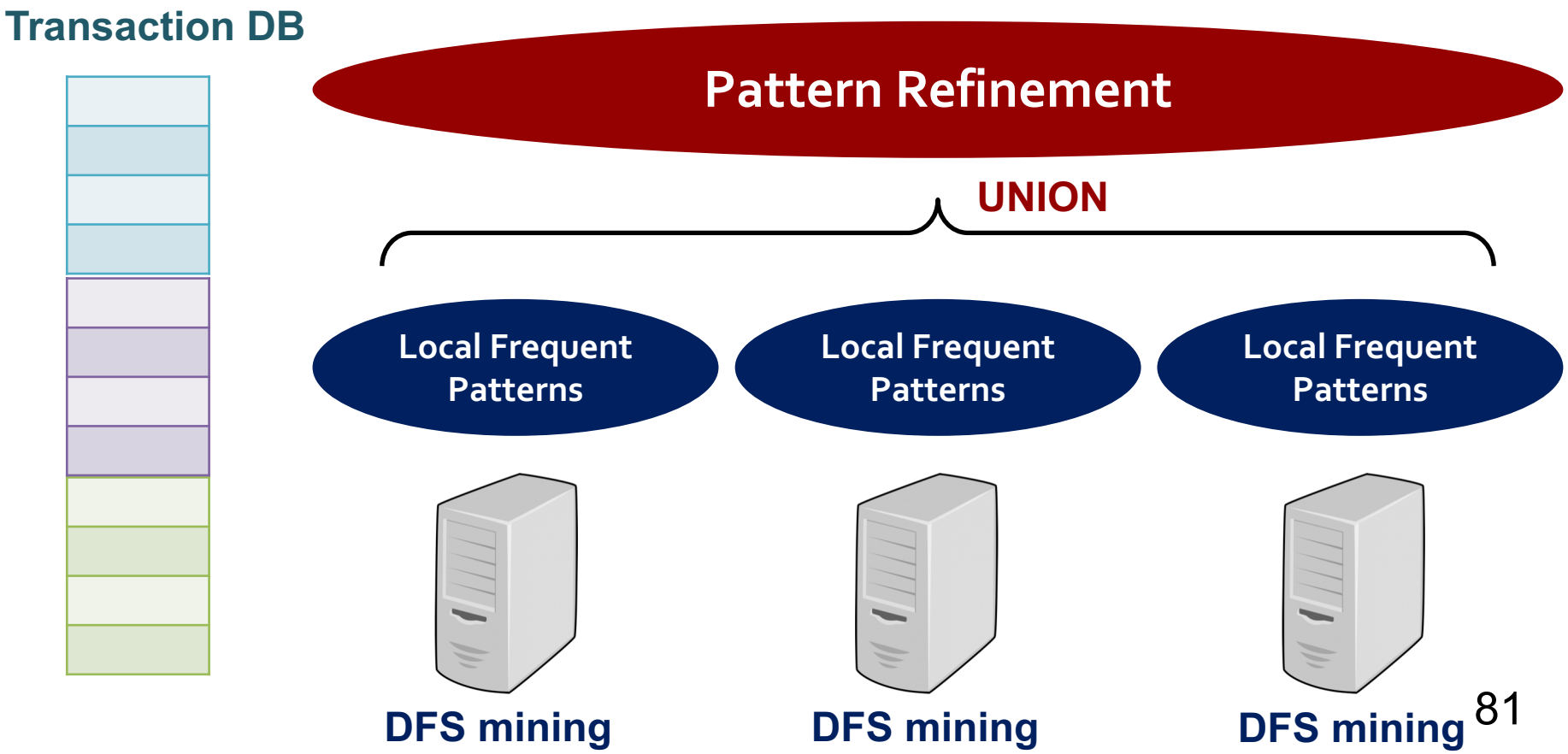
FIM is a useful tool for discovering frequently co-occurrent items. Existing FIM algorithms such as Apriori [9] and FP-Growth [6] can be resource intensive when a mined dataset is huge. Parallel algorithms were developed for reducing memory use and computational cost on each machine. Early efforts (related 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

2014

Wenqing Lin, Xiaokui Xiao, Gabriel Ghinita:
Large-scale frequent subgraph mining in MapReduce. ICDE 2014: 844-855

Related Work

Graph Patterns: Depth-First Pattern Growth



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

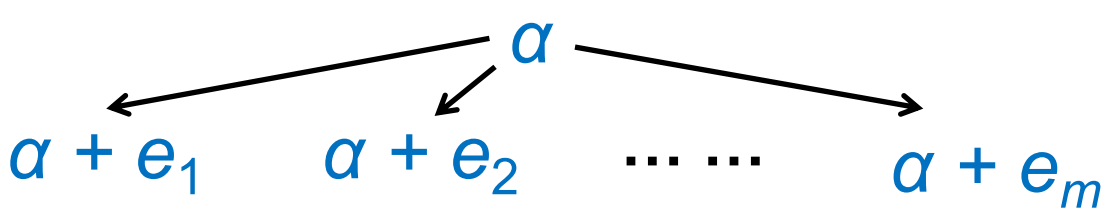
Forced isomorphism check, subgraph materialization

Check out G-thinker !!!

Think Like A Task

Each pattern α is associated with a task t_α

- Task t_α checks the frequency of α over $D|_\alpha$
- Task t_α grows α 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

```
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

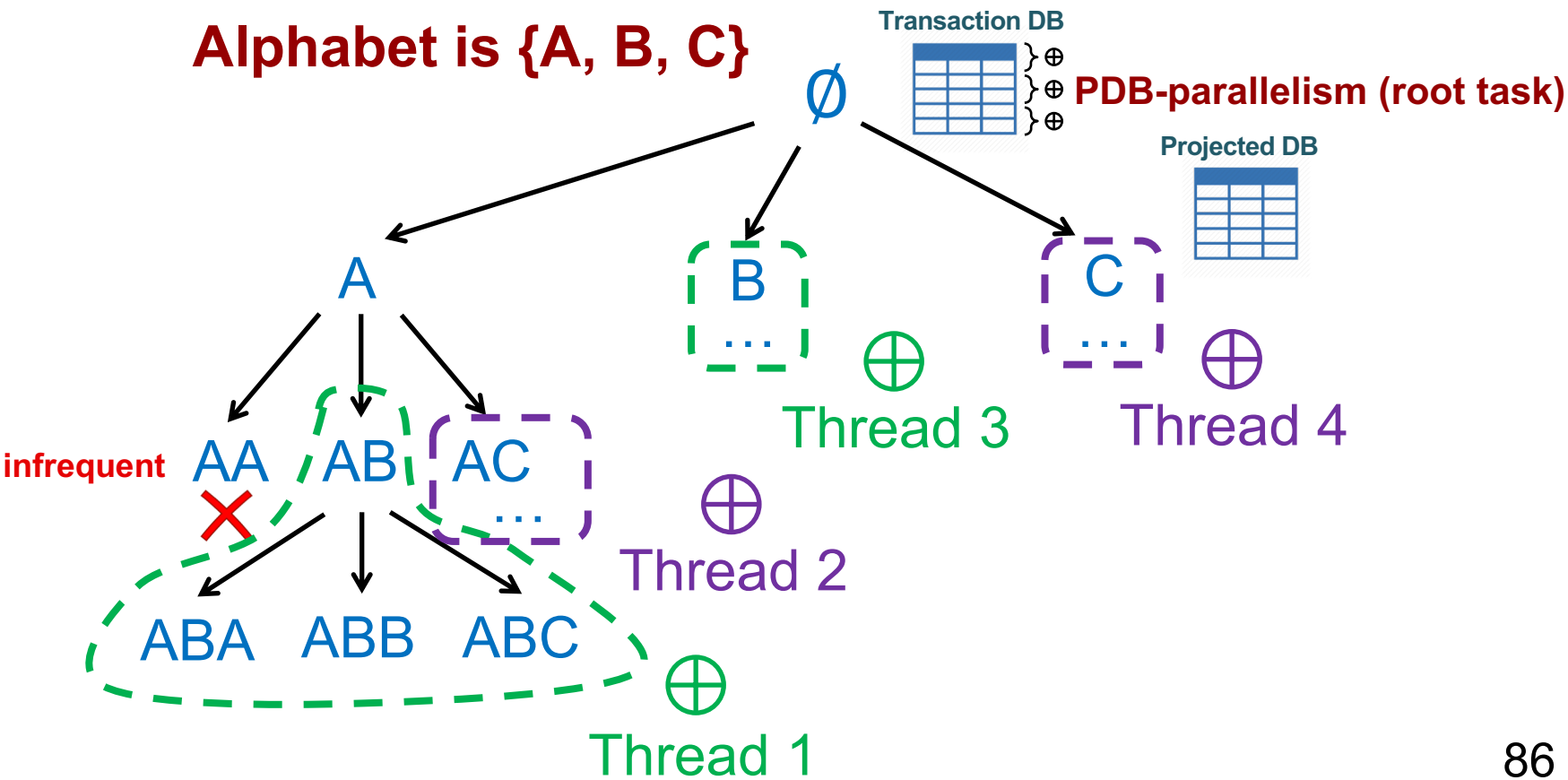
Trans	ProjTrans	Pattern
int *transaction_id* // transaction data	int *transaction_id* // transaction match	// α and $D\|_\alpha$ UDF: print(ostream& fout)

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()

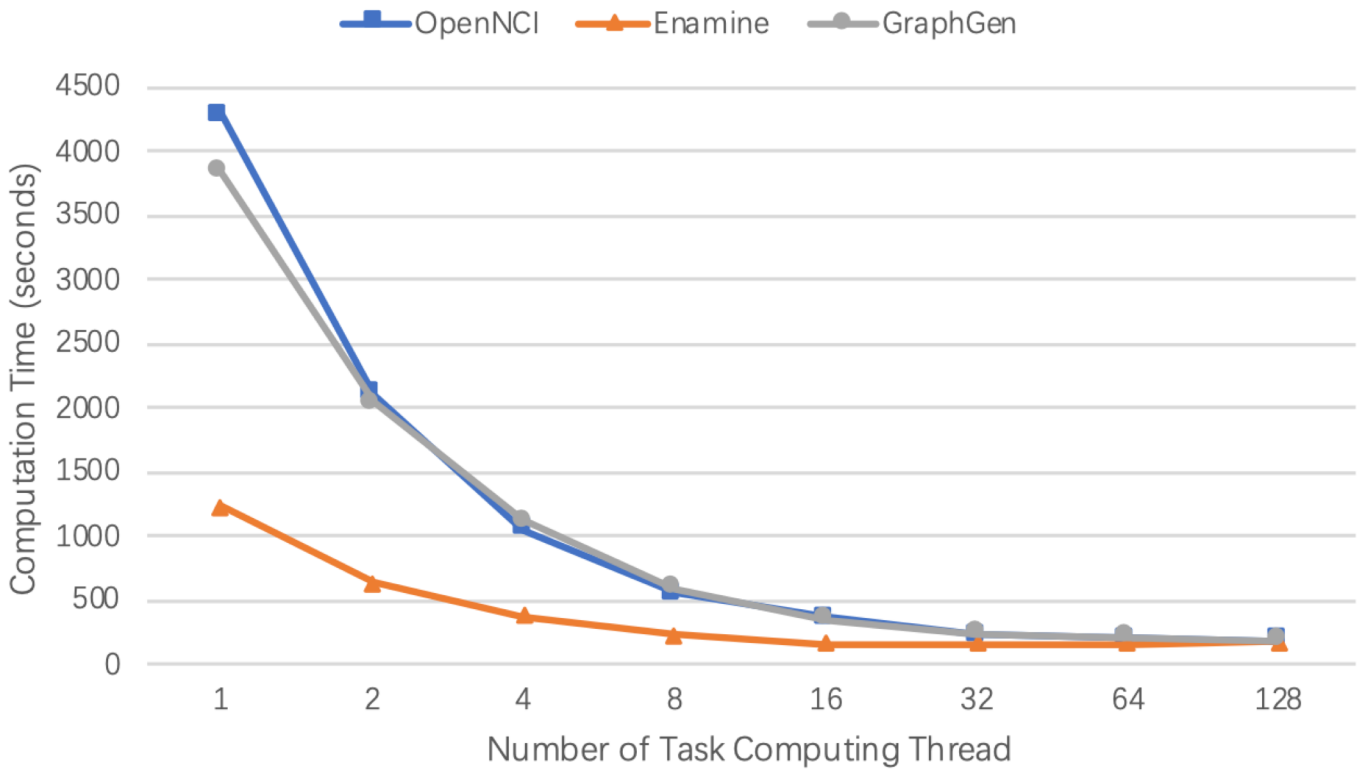
Concurrent Processing

Task Parallelism



Performance

gSpan Parallelization



Open Source Software

Check out Our GitHub Page

- https://github.com/yanlab19870714/PrefixFPM
- Demo: https://youtu.be/PfioCoGDpsw

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

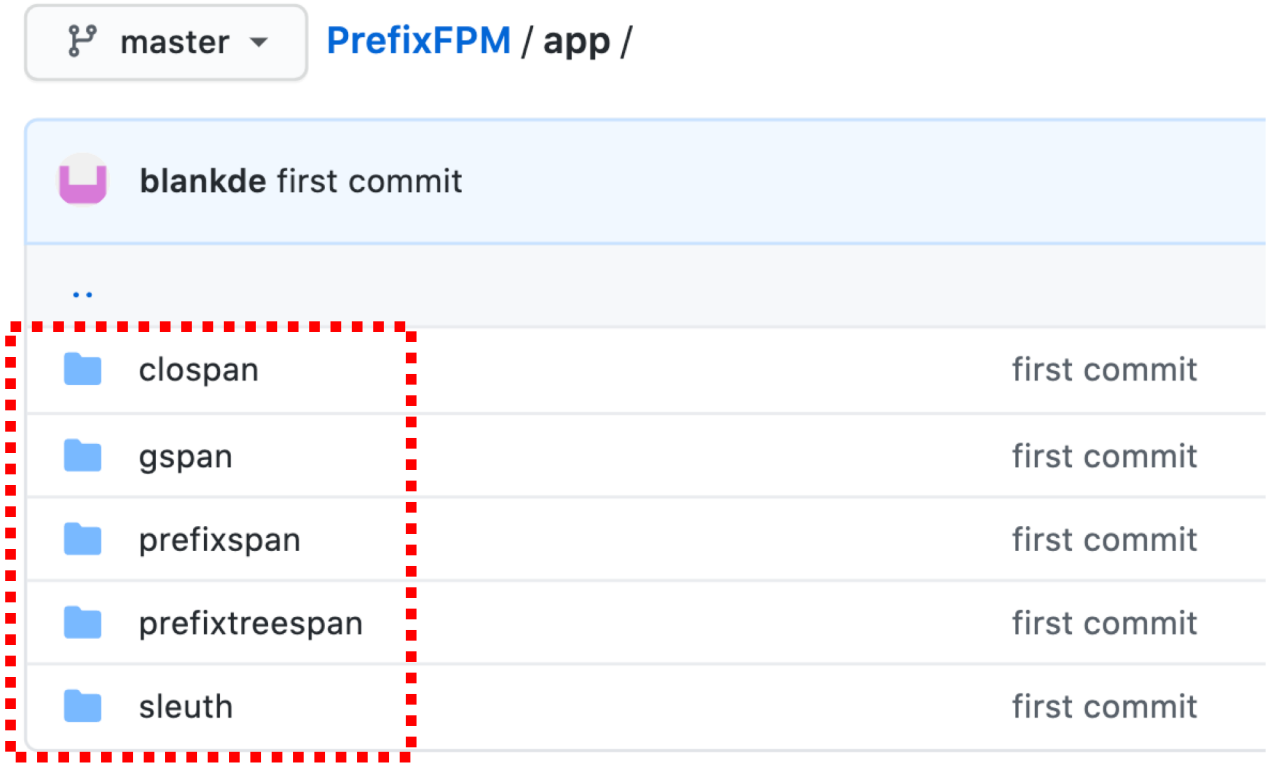
The applicatons are adapted from:

- PrefixSpan: http://chasen.org/~taku/software/prefixspan/prefixspan-0.4.tar.gz
- gSpan: https://github.com/rkwitt/gboost/tree/master/src-gspan
- Sleuth: http://www.cs.rpi.edu/~zaki/www-new/pmwiki.php/Software/Software#toc19

Recent Improvement

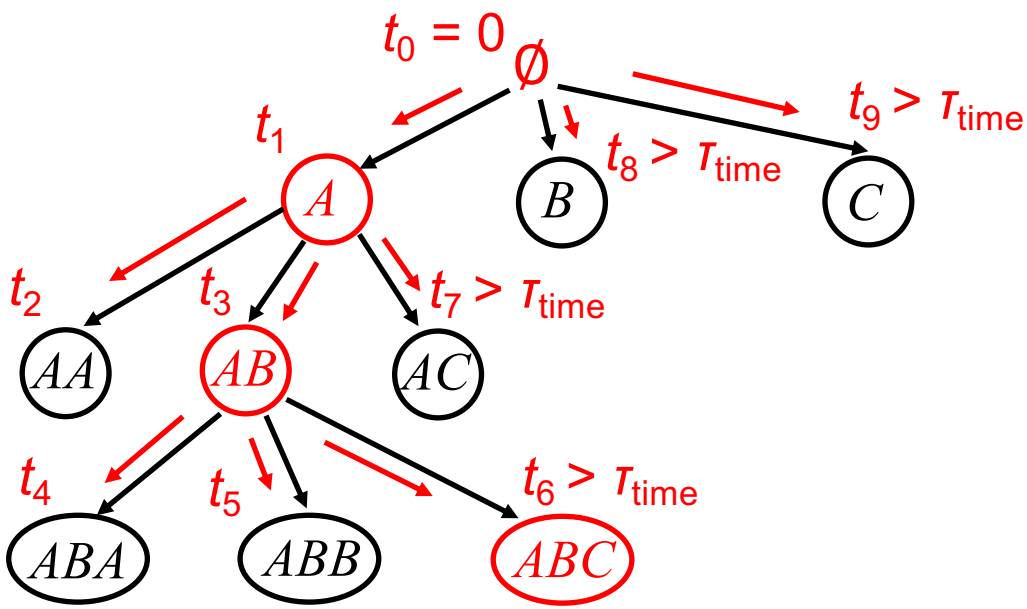
Check out Our GitHub Page

- https://github.com/wenwenQu/PrefixFPM
- Now with more applications



Recent Improvement

Timeout Mechanism




```
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);
            while(Task* t= get_next_child()){
                queue().push(t);
            q_mtx.unlock();
            return;
        }
        if(needSplit()){
            q_mtx.lock();
            queue().push(t);
            q_mtx.unlock();
        }
        else{
            t->run(fout);
            delete t;
        }
    }
}
```

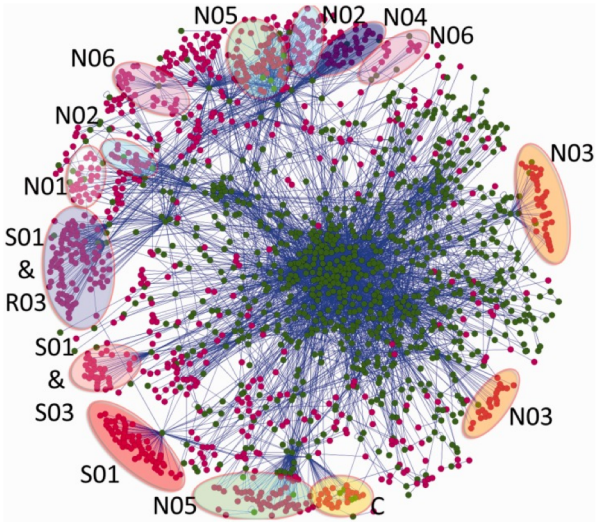
Overview

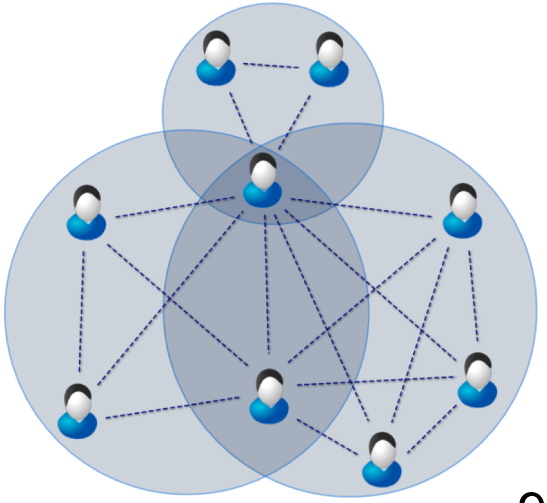
- Graph Mining Problems: A Categorization
- Think Like a Vertex: Data-Intensive Mining
- Why TLAV is not Sufficient? Compute-Intensive!
- PrefixFPM for Frequent Subgraph Mining
- **G-thinker: Think Like a Subgraph**
- TLAS Algorithms for Subgraph Mining

Target Problems

Finding dense subgraphs in a big graph

- Maximum/maximal Cliques
- Quasi-cliques, *k*-plexes
- Other definitions
  - *n*-cliques
  - *n*-clans
  - *n*-clubs
  - …

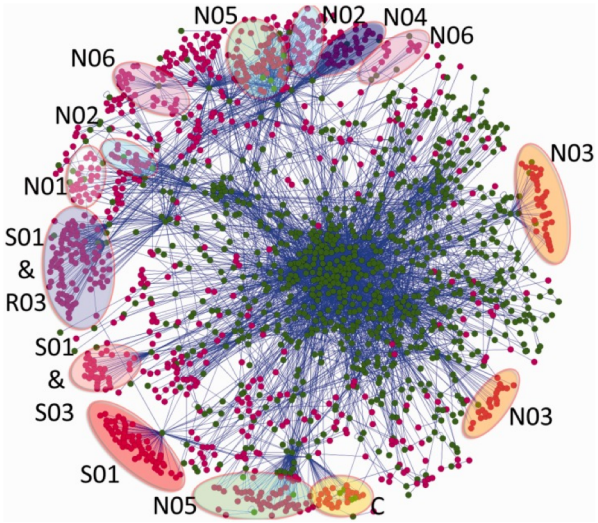


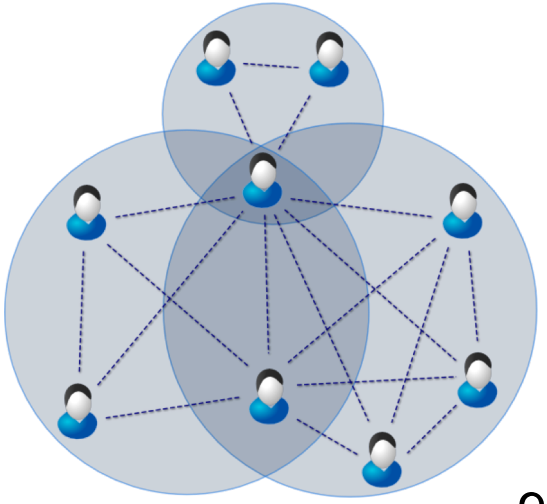


Target Problems

Finding dense subgraphs in a big graph

- Applications:
  - Social communities
  - Gene regulatory network, PPI network

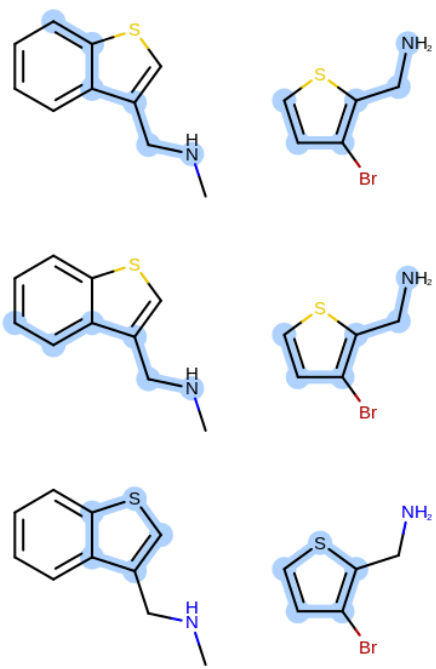


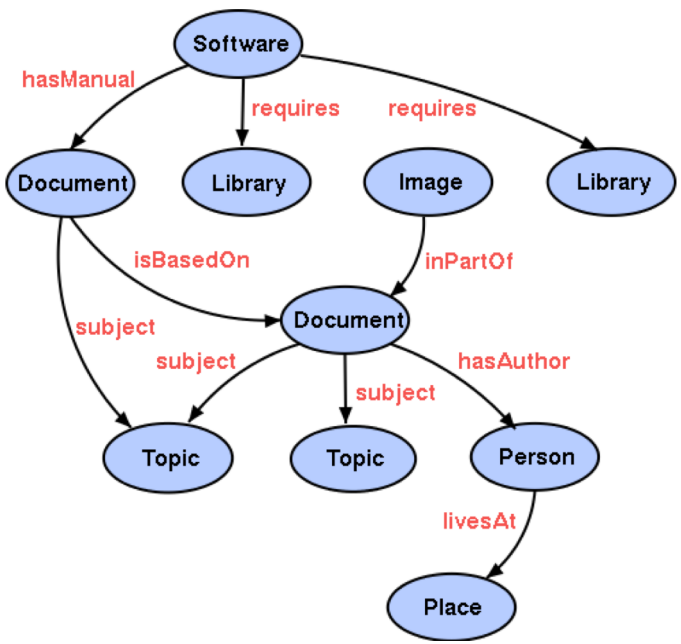


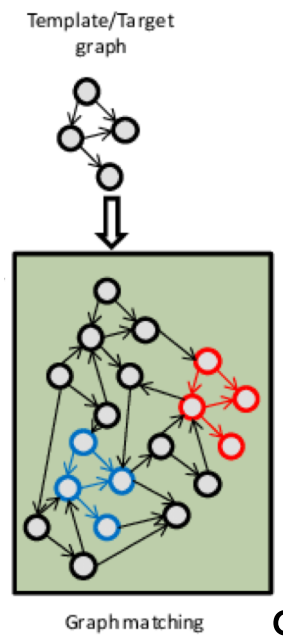
Target Problems

Subgraph matching (not mining)

- Graph pattern recognition (e.g., on biological networks)
- Searching text-rich networks like the Semantic Web





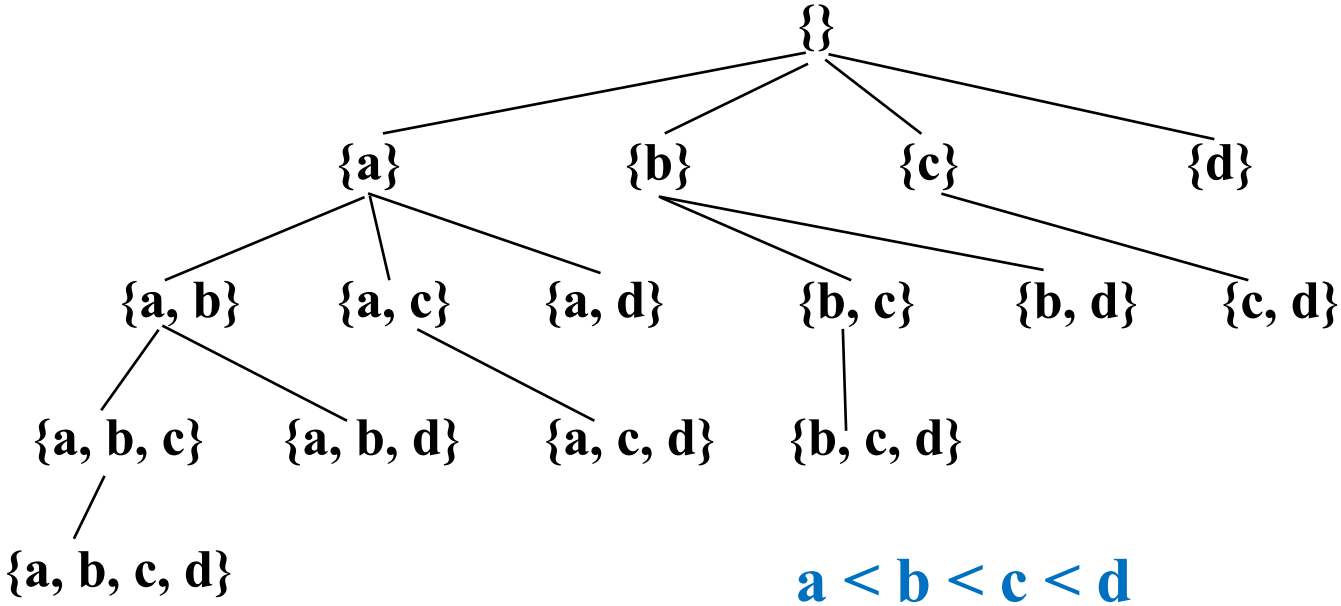


S: set of nodes already in subgraph
ext(*S*): set of valid candidates to add

Problem Features

High computational complexity

Search space: power set of vertex set



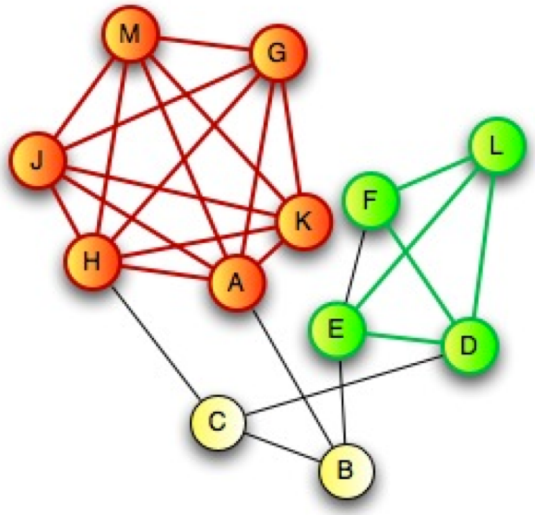
a < b < c < d

Problem Features

Divide and conquer

Serial algorithms: subgraph backtracking

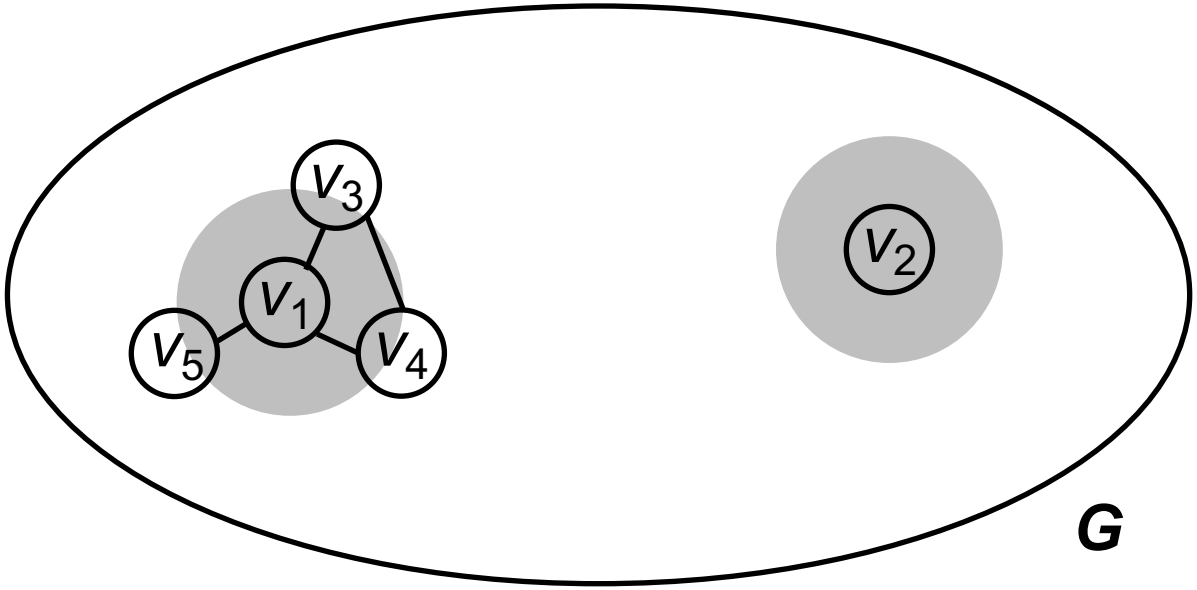
Example: Finding Maximal Cliques



Problem Features

Divide and conquer

Serial algorithms: subgraph backtracking

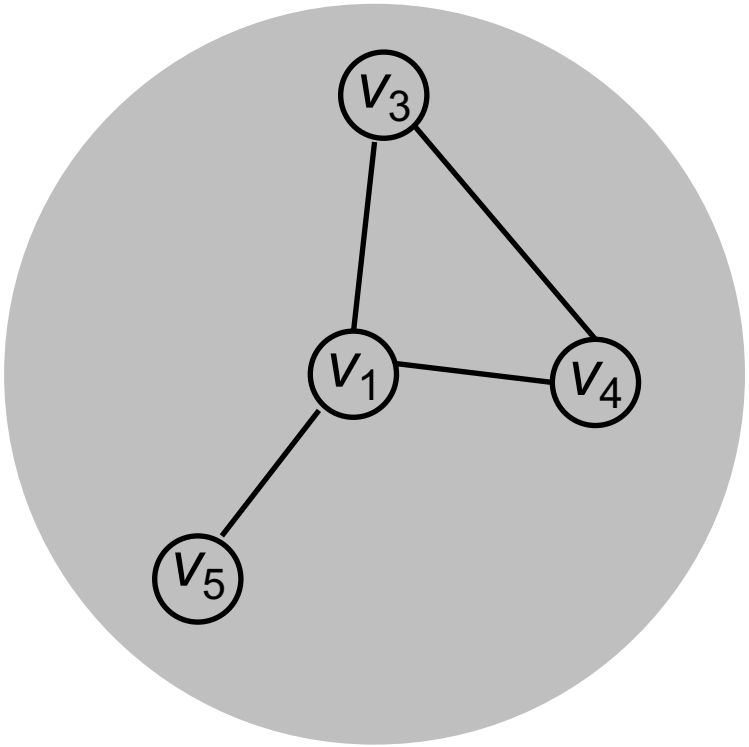


One-Hop Neighborhood

Problem Features

Divide and conquer **Recursive: v_1's one-hop**

Serial algorithms: subgraph backtracking

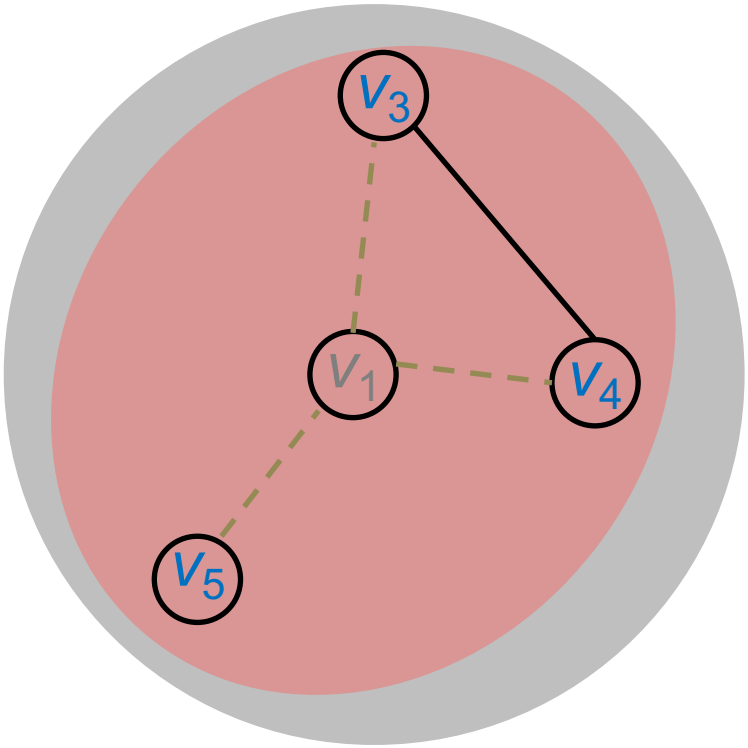


Problem Features

Divide and conquer

Recursive: $\{v_1 + v_3\}$'s one-hop

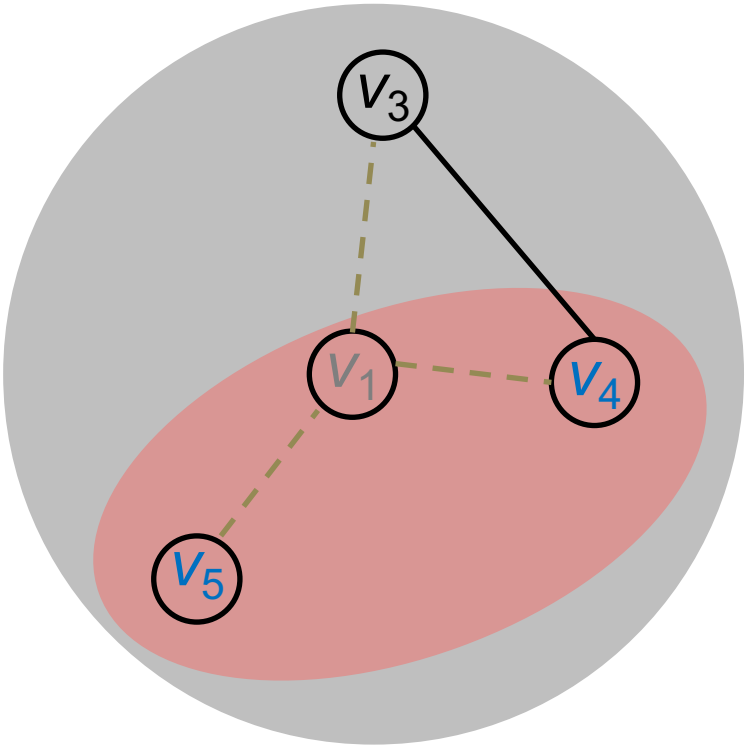
Serial algorithms: subgraph backtracking



Problem Features

Divide and conquer **Recursive: $\{v_1 + v_4\}$'s one-hop**

Serial algorithms: subgraph backtracking

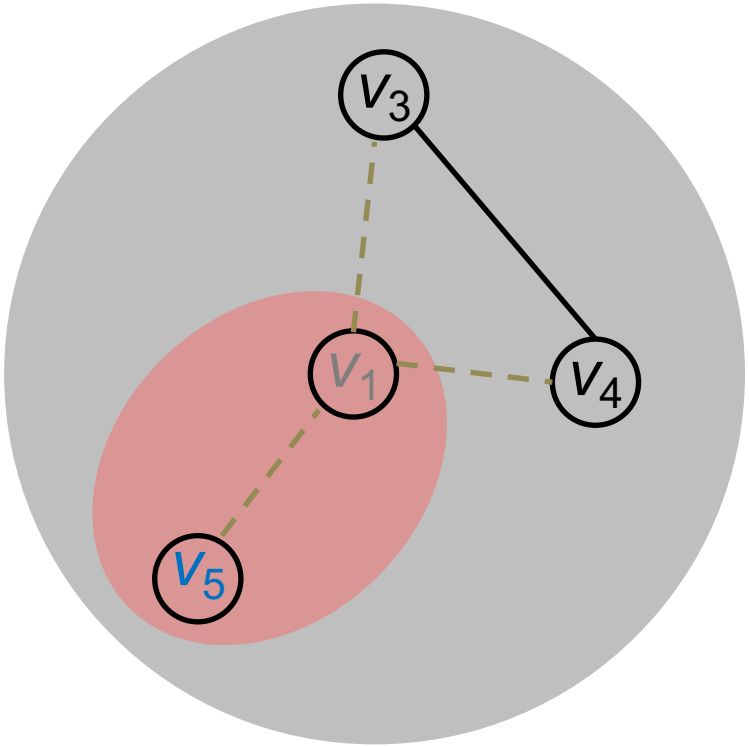


Problem Features

Divide and conquer

Recursive: $\{v_1 + v_5\}$'s one-hop

Serial algorithms: subgraph backtracking



Existing Solutions

MapReduce, Vertex-Centric Systems

Subgraph-Centric Systems

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

Arabesque's API

```
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

Abdul Quamar, Amol Deshpande, Jimmy J. Lin:
NScale: neighborhood-centric large-scale graph analytics in the cloud. VLDB J. 25(2): 125-150 (2016)

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

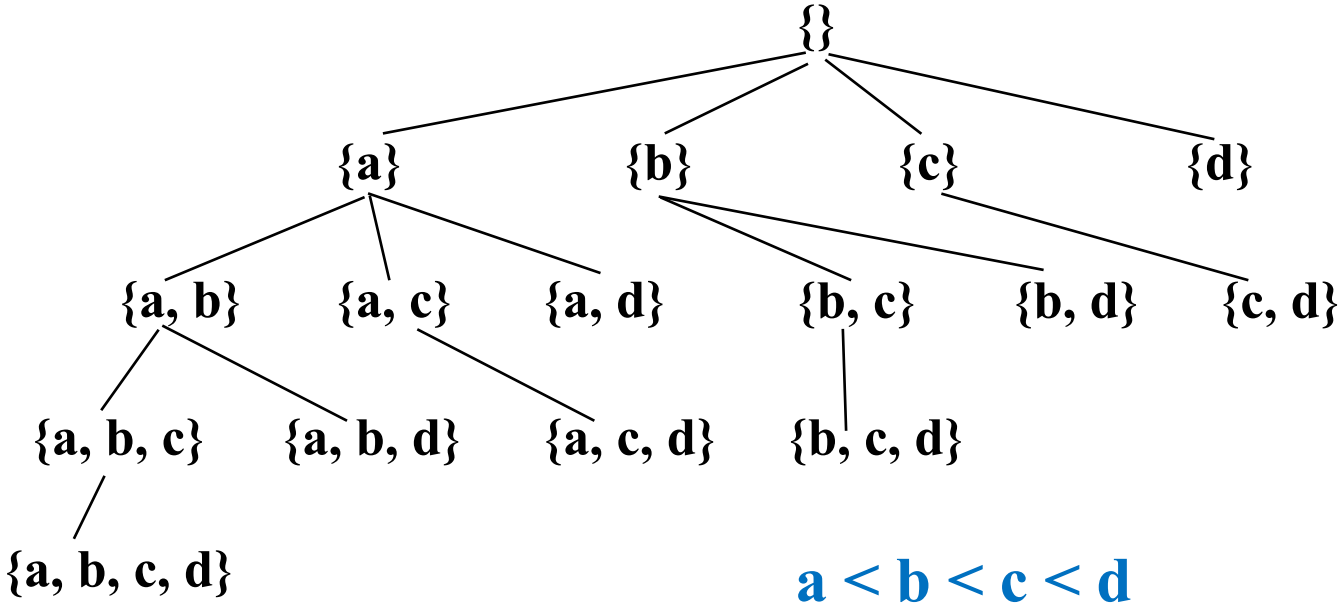
» Embedding-centric API
» To incorporate frequent subgraph pattern mining, need isomorphism check for each materialized subgraph

Not needed in our subgraph mining problem!

Problem Features

High computational complexity

Search space: power set of vertex set



$a < b < c < d$

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

Recall PrefixFPM

Arabesque

» Embedding-centric API

» To incorporate frequent subgraph pattern mining, need isomorphism check for each materialized subgraph

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

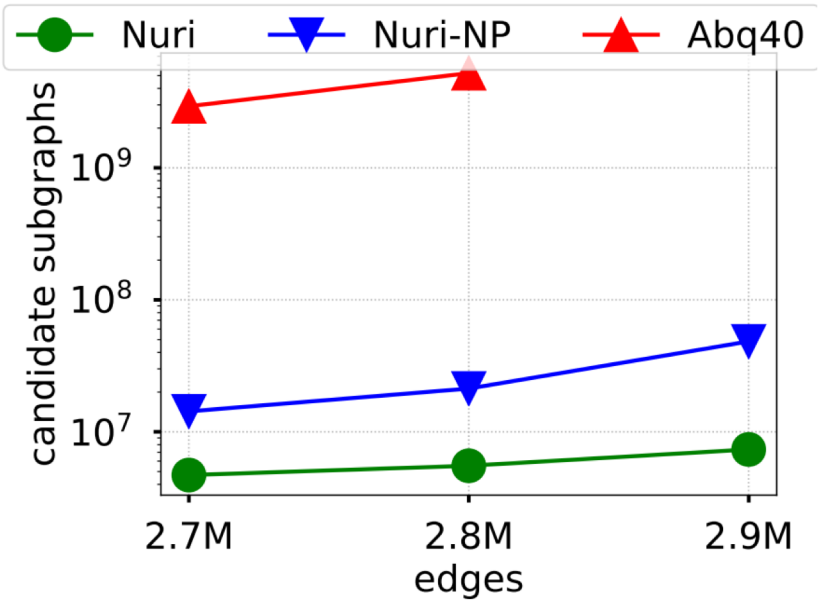
Nuri

» To find the *k* most relevant subgraphs using only a single computer

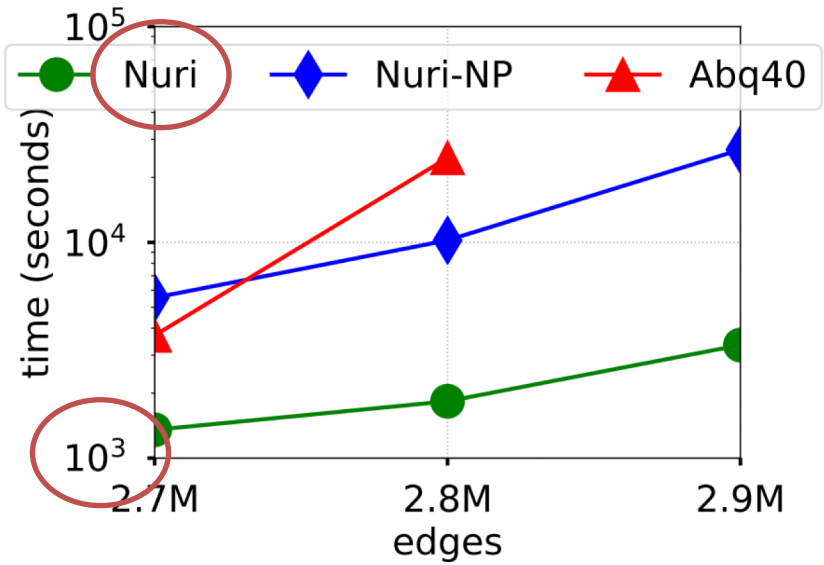
» Prioritized subgraph expansion

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*



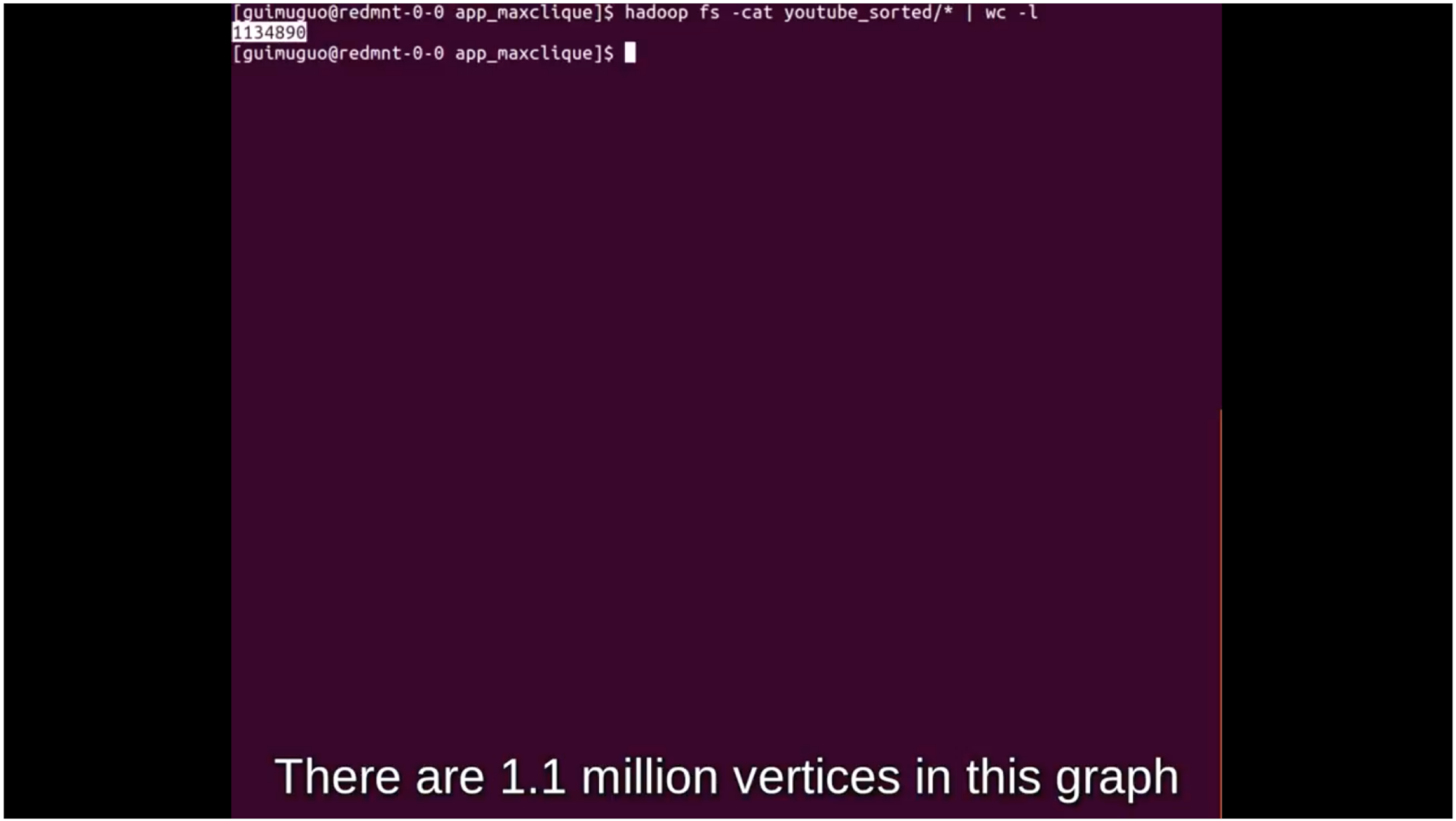
(a) subgraphs



(b) time

Figure 11: Clique Discovery (YouTube)

A G-thinker Demo



A G-thinker Demo

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

```
Load Time : 2.427893 seconds

============ PROGRESS REPORT ============
Total number of tasks processed: 722
Worker 0: 722 tasks processed


============ PROGRESS REPORT ============
Total number of tasks processed: 5555
Worker 0: 5555 tasks processed (+ 4833)


============ PROGRESS REPORT ============
Total number of tasks processed: 15120
Worker 0: 15120 tasks processed (+ 9565)


============ PROGRESS REPORT ============
Total number of tasks processed: 26087
Worker 0: 26087 tasks processed (+ 10967)


============ PROGRESS REPORT ============
Total number of tasks processed: 26087
Worker 0: 26087 tasks processed (+ 0)


============ PROGRESS REPORT ============
Total number of tasks processed: 26282
Worker 0: 26282 tasks processed (+ 195)

Qmax: 11010 18173 10844 20688 20117 6426 20727 20151 15144 20934 1072 7828 311 20200 7864 20
235 5204
|Qmax| = 17

real    0m9.449s
user    0m0.024s
sys     0m0.010s
[guimuguo@redmnt-0-0 app_
```

http://bit.ly/gthinker

It took 9.449 seconds

The Curse of IO

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

frankmcsherry avoid mmap contention 5201a5d on Aug 27, 2018

2 contributors

307 lines (176 sloc) | 38.6 KB

Raw | Blame | History

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 !

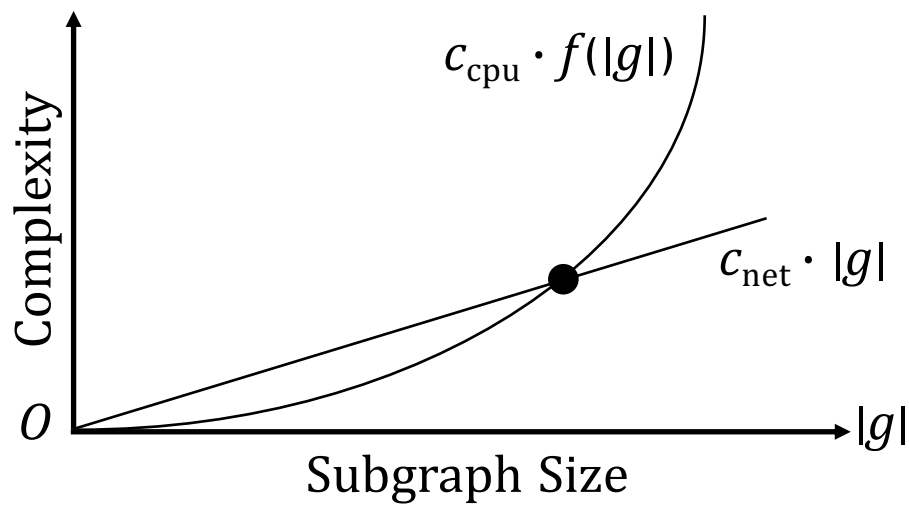
Lots of problems !!!

Our Solution

Think Like a Task

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

Da Yan, Guimu Guo, Md Mashiur Rahman Chowdhury, M. Tamer Özsu, John C. S. Lui, Weida Tan: **T-thinker: a task-centric distributed framework for compute-intensive divide-and-conquer algorithms.** PPoPP 2019: 411-412



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

Yi Zhou, Jingwei Xu, Zhenyu Guo, Mingyu Xiao, Yan Jin:
Enumerating Maximal *k*-Plexes with Worst-Case Time Guarantee. AAAI 2020: 2442-2449

⋮

G-thinker History

Da Yan, Hongzhi C ... ui:
G-thinker: Big Gr ... 2017)

EuroSys '18, April 23–26, 2018, Porto, Portugal

Prototype: AI
No multithrea ... each machine
Inefficient en
- **Vertex cac**
- **LSH-basec ... on disk**

Hongzhi Chen, Mia
G-Miner: an effici ... 2:12

Basically our ... port added
But inherits a

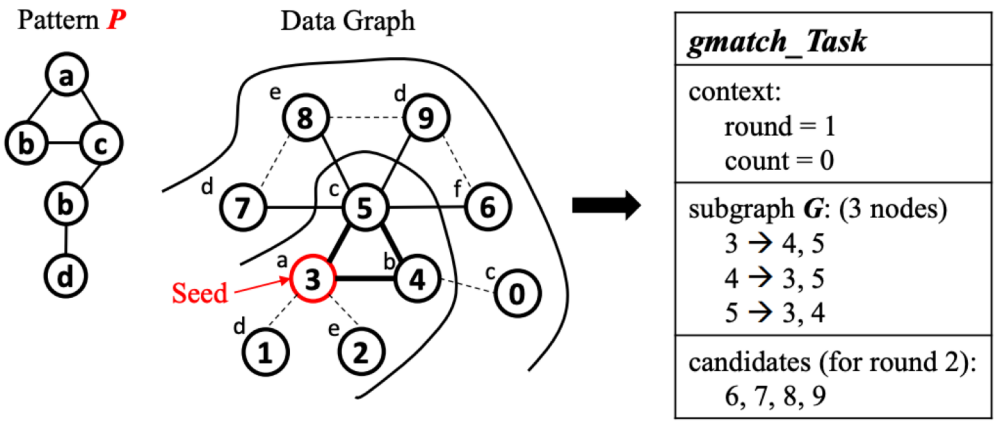
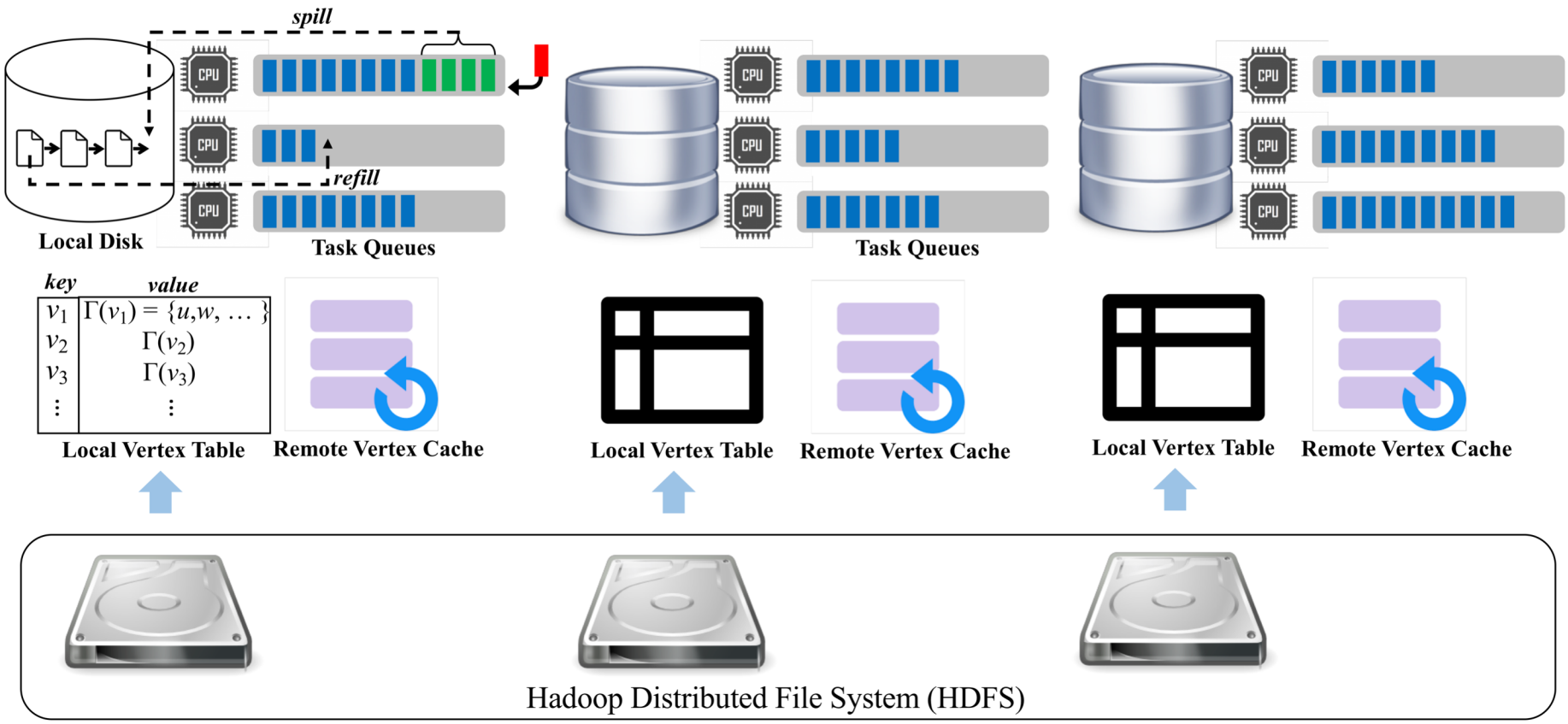


Figure 1: Task model for graph matching

The task model is inspired by the task concept in G-thinker [36]

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

G-thinker Overview

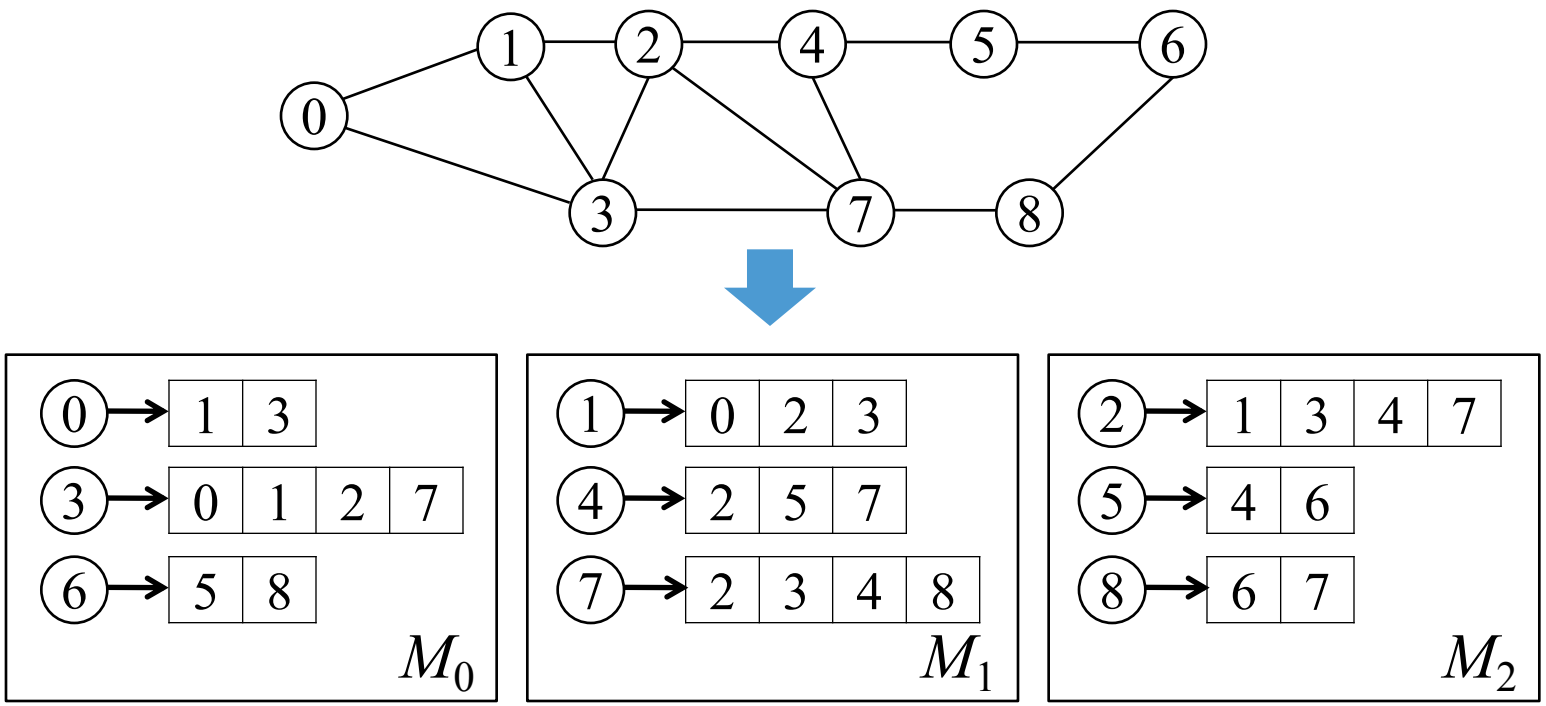


G-thinker

Vertex Partitioning

Distributed Key-Value Store

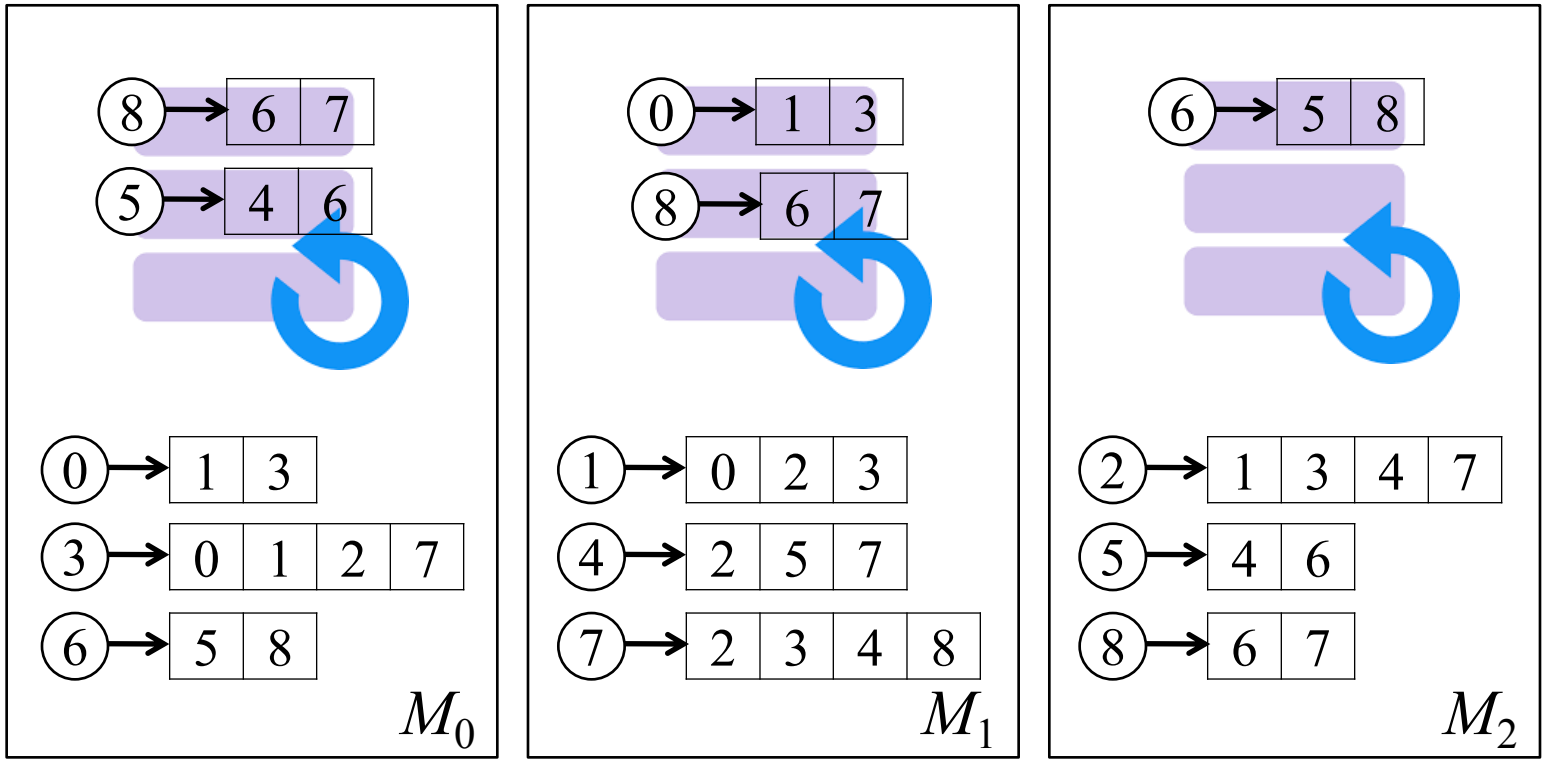
- Key: vertex ID
- Value: (labeled) adjacency list



G-thinker

Vertex Pulling & Remote Vertex Caching

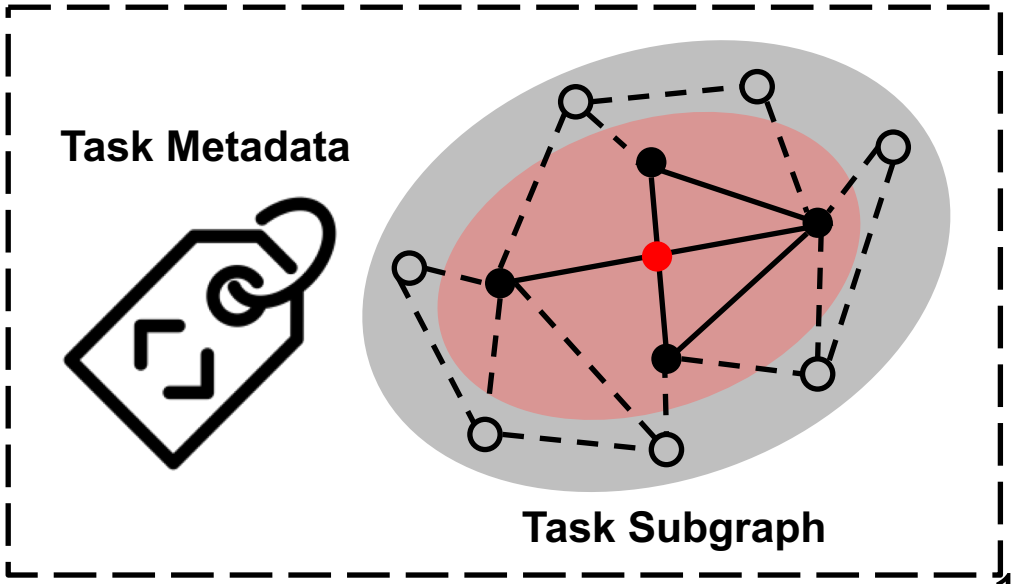
Tasks can share vertices to reduce # of remote vertex requests



G-thinker

Task-Centric Programming Interface

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



Γ-table is for getting adjacency list given a vertex ID

Counter tracks how many tasks lock a vertex

G-thinker

Vertex Cache: High Concurrency

» Vertices hashed to different buckets can be accessed concurrently

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

A bucket

Vertices hashed to buckets

Buckets protected by mutex

Each bucket has

- A Γ-table
- A Z-table
- An R-table

$B_v = v \bmod 10$

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

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

A bucket

Vertices hashed to buckets

Buckets protected by mutex

Each bucket has

- A Γ-table
- A Z-table
- An R-table

$B_v = v \bmod 10$

R-table tracks those vertices requested but not received yet

Used to avoid sending the same vertex-request again

G-thinker

Vertex Cache: High Concurrency

» Vertices hashed to different buckets can be accessed concurrently

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

A bucket

Vertices hashed to buckets

Buckets protected by mutex

Each bucket has

- A Γ-table
- A Z-table
- An R-table

$B_v = v \bmod 10$

G-thinker

Vertex Cache: High Concurrency

» Computing threads

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

pull(10)
pull(11)

Bucket #	Γ-tables ID(v)	Γ-tables Γ(v), lock-count	Z-tables ID(v)	R-tables ID(v)	R-tables lock-count
0	10 ...	Γ(10), 1 ...	... ...	20 ...	2 ...
1	11 ...	Γ(11), 2 ...	... ...	21 ...	1 ...
2	12 22 ...	Γ(12), 0 Γ(22), 3 ...	12	... ...	... ...
3	13 ...	Γ(13), 2 ...	... ...	... ...	... ...
⋮					
9	...	...	...	...	...

G-thinker

Vertex Cache: High Concurrency

» Computing threads

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

release(12)
release(13)

Bucket #	Γ-tables		Z-tables	R-tables	
	ID(v)	Γ(v), lock-count	ID(v)	ID(v)	lock-count
0	10 ...	Γ(10), 1 ...	... ...	20 ...	2 ...
1	11 ...	Γ(11), 2 ...	... ...	21 ...	1 ...
2	12 22 ...	Γ(12), 0 Γ(22), 3 ...	12 ...	... ...	... ...
3	13 ...	Γ(13), 1 ...	... ...	... ...	... ...
		⋮			
9	...	...	...	...	...

G-thinker

Vertex Cache: High Concurrency

» Computing threads

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

pull(20)
pull(21)

Bucket #	Γ-tables		Z-tables	R-tables	
	ID(v)	Γ(v), lock-count	ID(v)	ID(v)	lock-count
0	10 ...	Γ(10), 1 ...	... ...	20 ...	2 ...
1	11 ...	Γ(11), 2 ...	... ...	21 ...	1 ...
2	12 22 ...	Γ(12), 0 Γ(22), 3 ...	12	... ...	... ...
3	13 ...	Γ(13), 2 ...	... ...	... ...	... ...
		⋮			
9	...	...	...	...	...

G-thinker

Vertex Cache: High Concurrency

» Response receiving thread

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

Γ(22) received

Bucket #	Γ-tables		Z-tables	R-tables	
	ID(v)	Γ(v), lock-count	ID(v)	ID(v)	lock-count
0	10	Γ(10), 1	···	20	2
1	11	Γ(11), 2	···	21	1
2	12 **22**	Γ(12), 0 **Γ(22), 3**	12	··· ···	··· ···
3	13	Γ(13), 2	···	···	···
⋮					
9	···	···	···	···	···

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

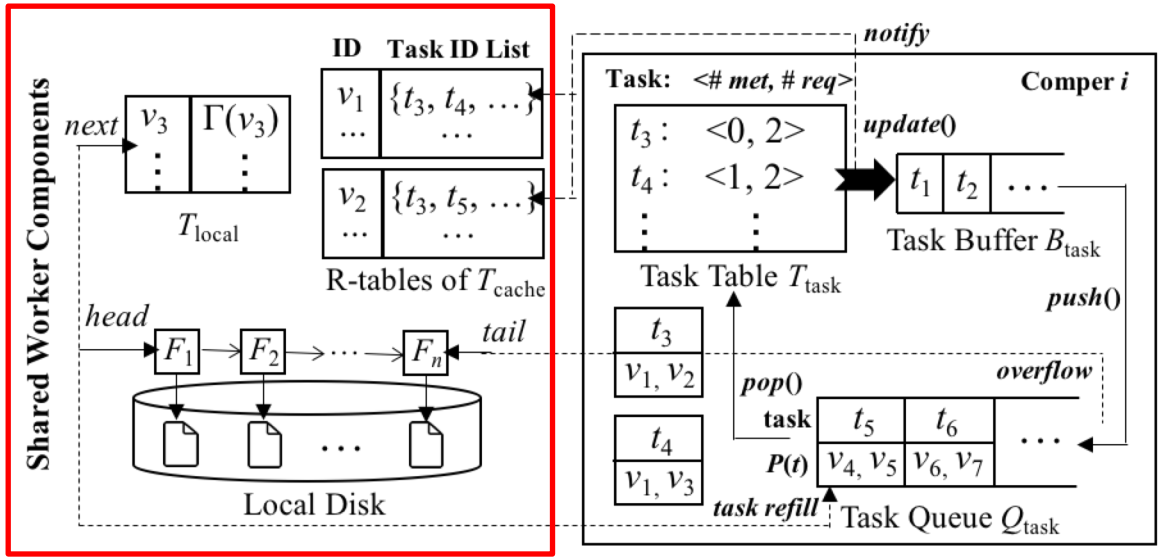
Bucket #	Γ-tables		Z-tables	R-tables	
	ID(v)	Γ(v), lock-count	ID(v)	ID(v)	lock-count
0	10 ...	Γ(10), 1 ...	... ...	20 ...	2 ...
1	11 ...	Γ(11), 2 ...	... ...	21 ...	1 ...
2	~~12~~ 22 ...	~~Γ(12), 0~~ Γ(22), 3 ...	~~12~~	... ...	... ...
3	13 ...	Γ(13), 2 ...	... ...	... ...	... ...
			⋮		
9	...	...	...	...	...

G-thinker

Task Management

» Shared data store on a machine

» Accessed by all computing threads

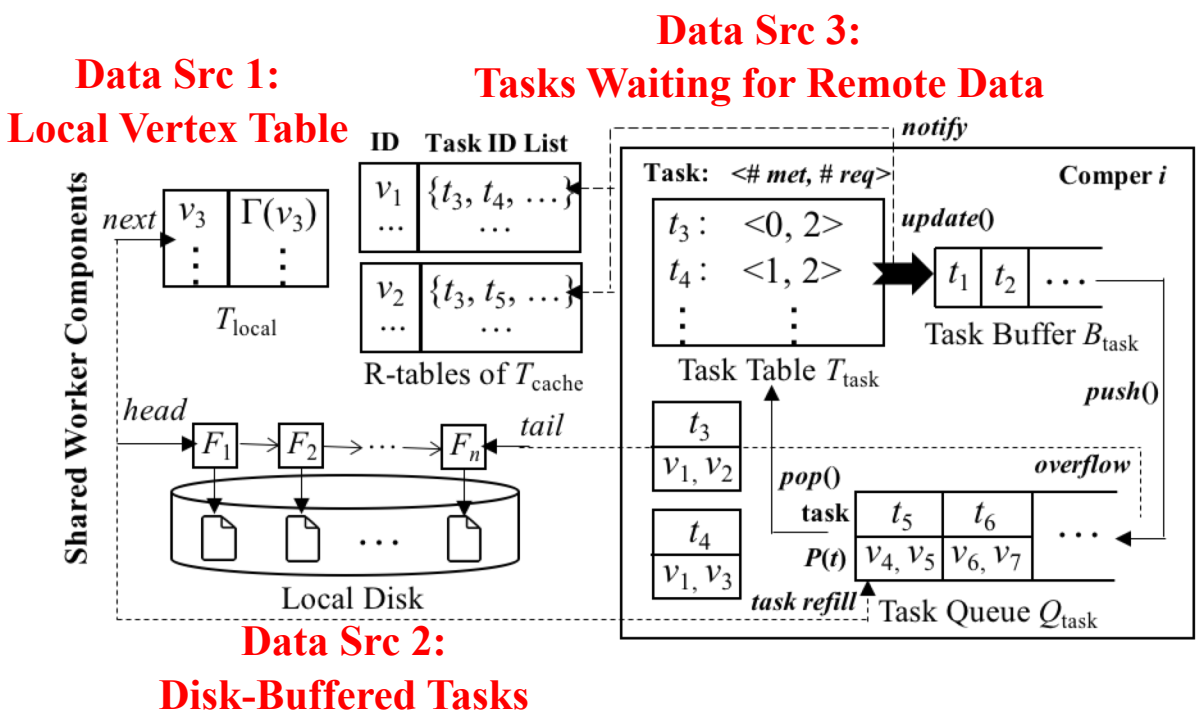


G-thinker

Task Management

» Shared data store on a machine

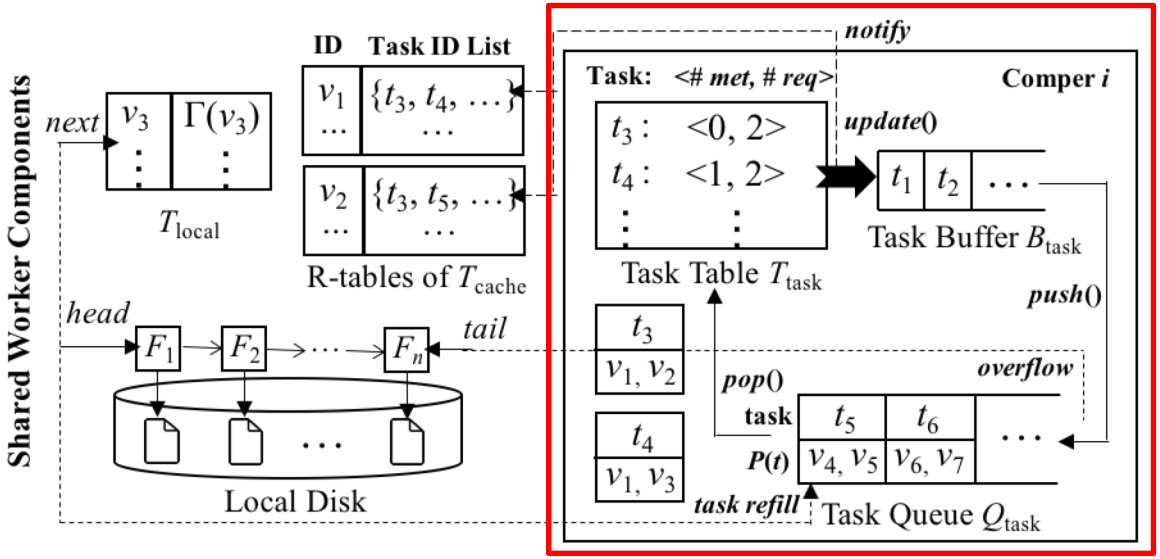
» Accessed by all computing threads



G-thinker

Task Management

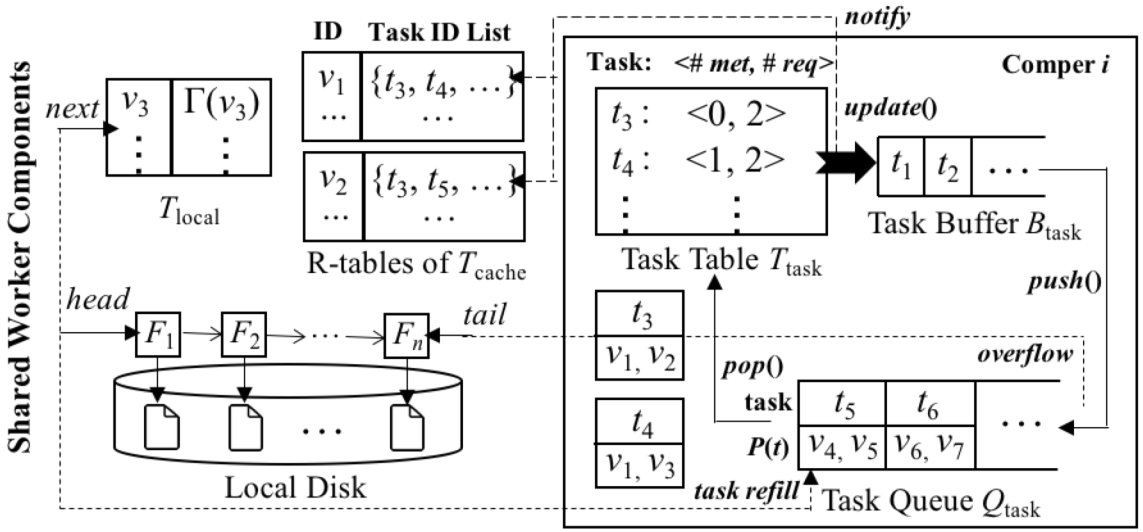
» Task containers maintained by each computing thread



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

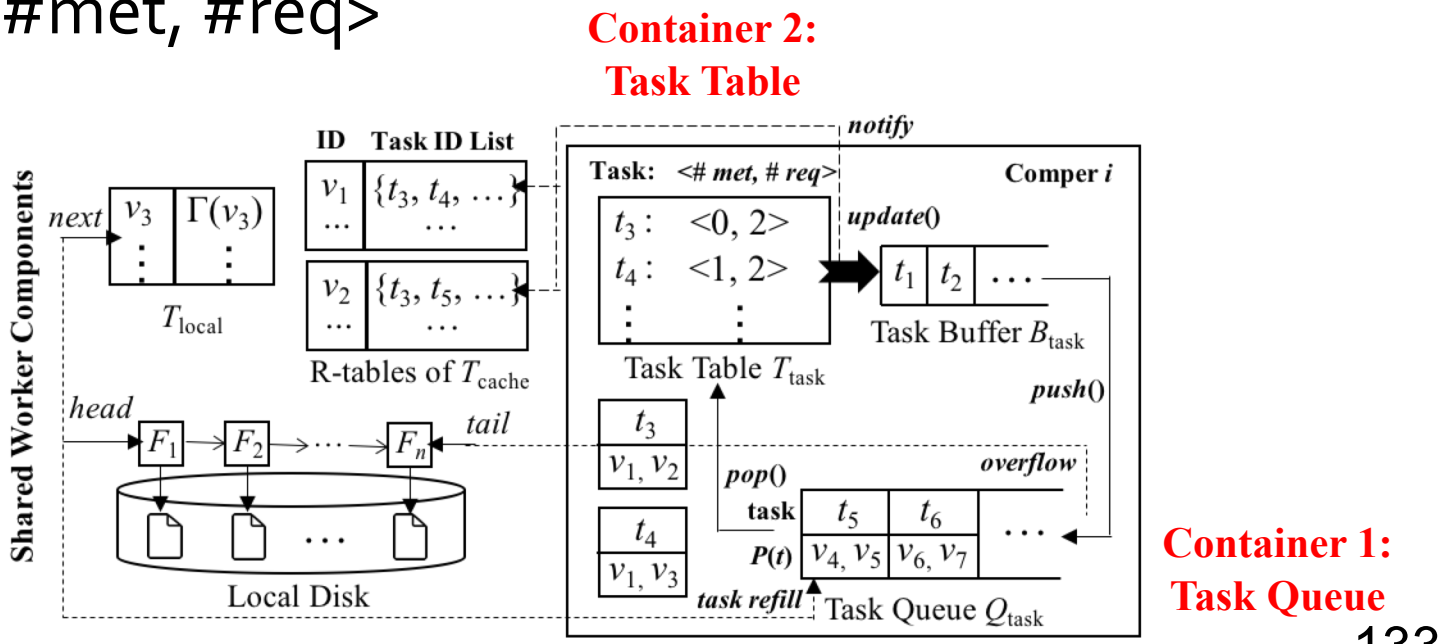


Container 1: Task Queue

G-thinker

Task Management

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

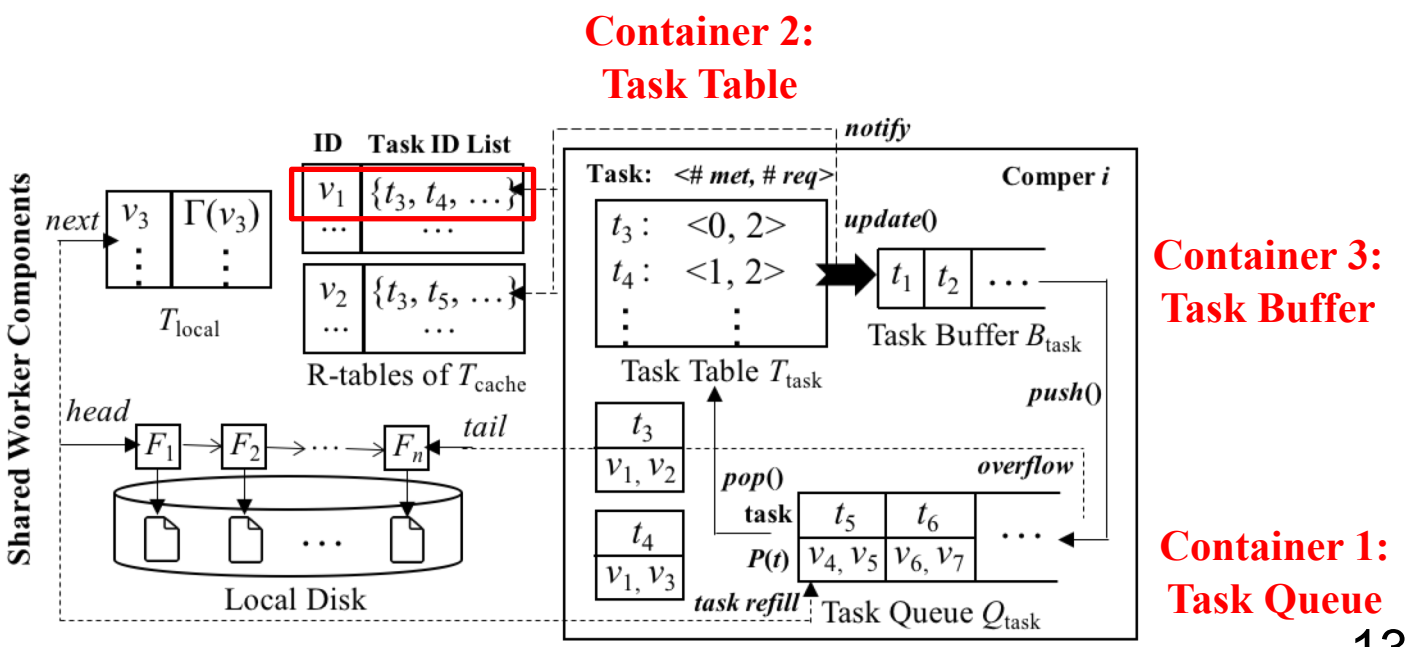


G-thinker

Task Management

» Vertex response triggers task update

» #met ++; If #met == #req, move task to B_{task}



G-thinker

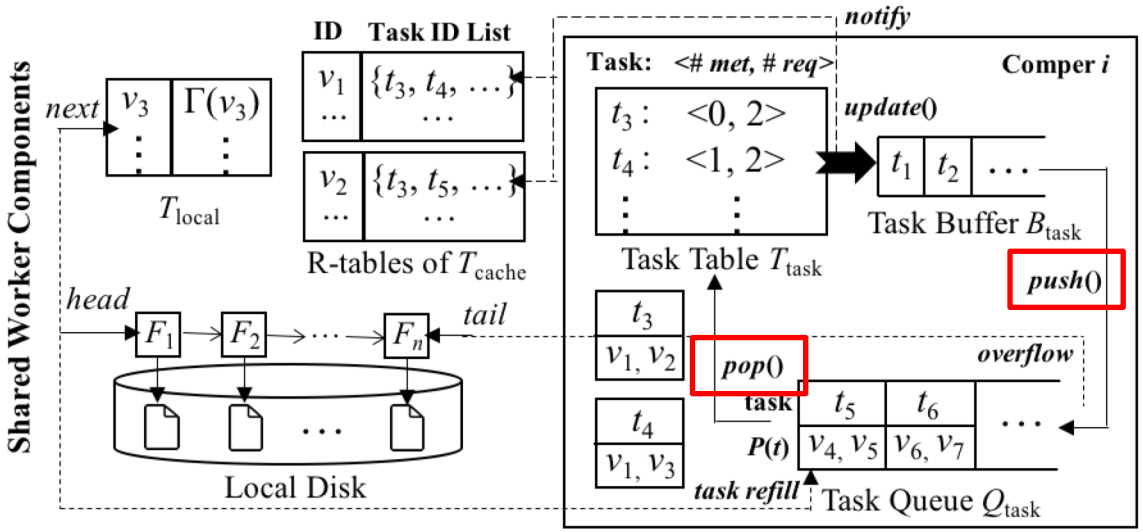
pop() is only called if vertex cache and task containers have enough space

push() is necessary to keep tasks flowing

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}

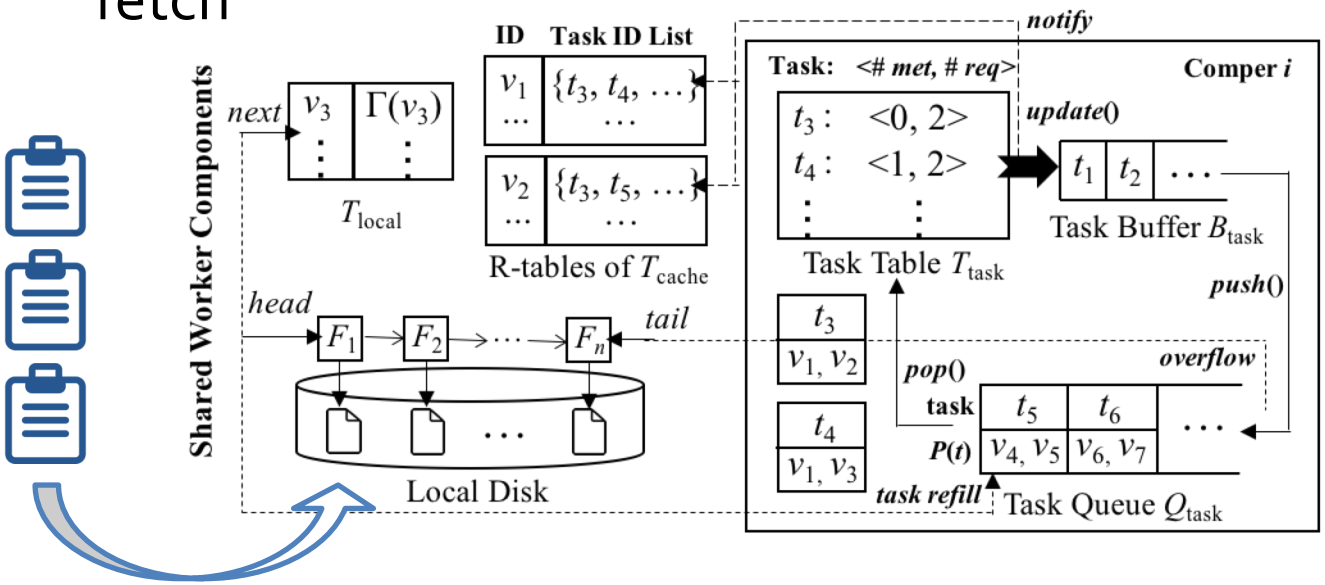


G-thinker

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

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

Performance

Dataset	Arabesque	Giraph	G-Miner	G-thinker
(a) TC :				
Youtube	60.3 s / 2.8 GB	74.3 s / 1.5 GB	14.7 s / 1 GB	7.1 s / 0.4 GB
Skitter	90.8 s / 4.8 GB	73.9 s / 3.9 GB	17.1 s / 1.1 GB	9.5 s / 0.5 GB
Orkut	533 s / 17.7 GB	197 s / 15 GB	667 s / 2.3 GB	26.6 s / 1.2 GB
BTC	x	x	> 24 hr / 6.7 GB	181 s / 3 GB
Friendster	x	x	10915 s / 9.2 GB	516 s / 3.1 GB
(b) MCF :				
Youtube	58.8 s / 4.7 GB	177 s / 6.2 GB	13.7 s / 1 GB	10.4 s / 0.5 GB
Skitter	145 s / 4.9 GB	x	26.2 s / 1.2 GB	85.6 s / 0.6 GB
Orkut	2007 s / 44.7 GB	x	691 s / 2.5 GB	95.9 s / 1.3 GB
BTC	x	x	> 24 hr / 7.3 GB	1831 s / 3 GB
Friendster	x	x	10644 s / 7.4 GB	252 s / 3.4 GB
(c) GM :				
Youtube	–	–	13.2 s / 0.8 GB	5.2 s / 0.5 GB
Skitter	–	–	13.9 s / 1.1 GB	7.8 s / 0.6 GB
Orkut	–	–	66.2 s / 2.2 GB	46.7 s / 1.5 GB
BTC	–	–	> 24 hr / 6 GB	153 s / 4.4 GB
Friendster	–	–	3669 s / 9.2 GB	1762 s / 7 GB

Note: (1) x = Out of memory; (2) "–" means inapplicable.

https://github.com/yanlab19870714/gthinkerQC

G-thinker+

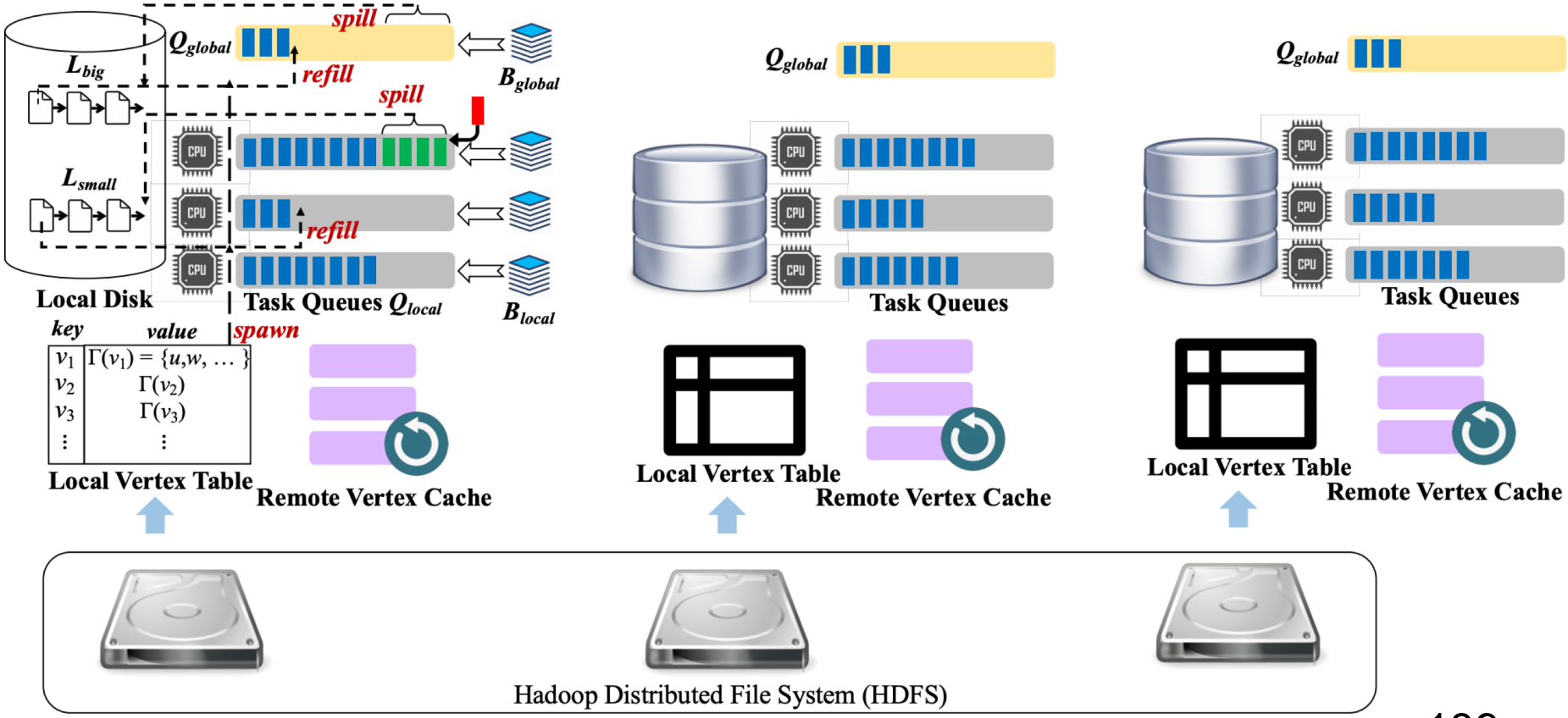
Recent improvement to G-thinker

- » 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:
Scalable Mining of Maximal Quasi-Cliques: An Algorithm-System Codesign Approach. CoRR abs/2005.00081 (2020)

https://github.com/yanlab19870714/gthinkerQC

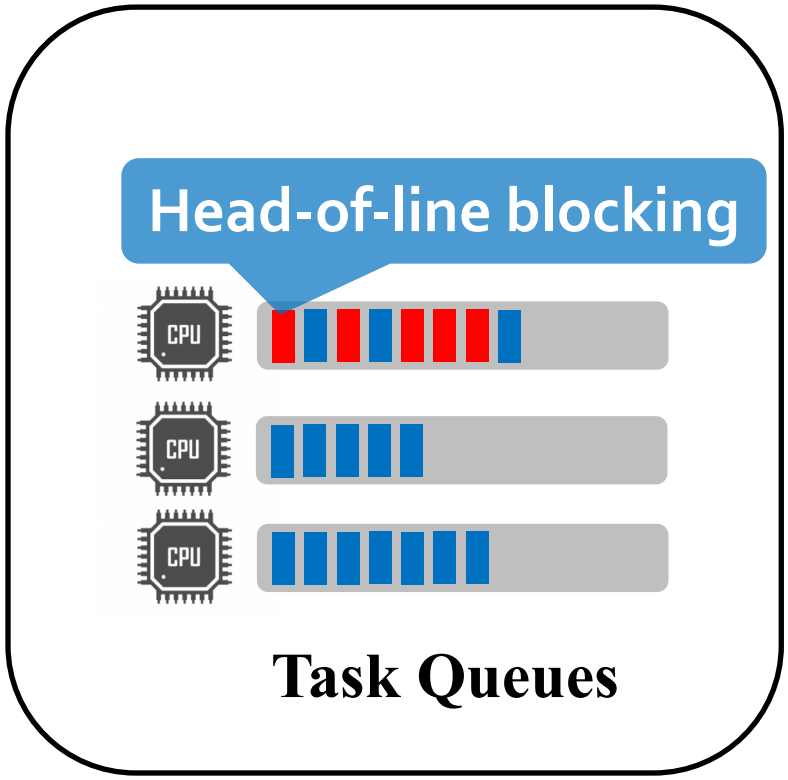
G-thinker+

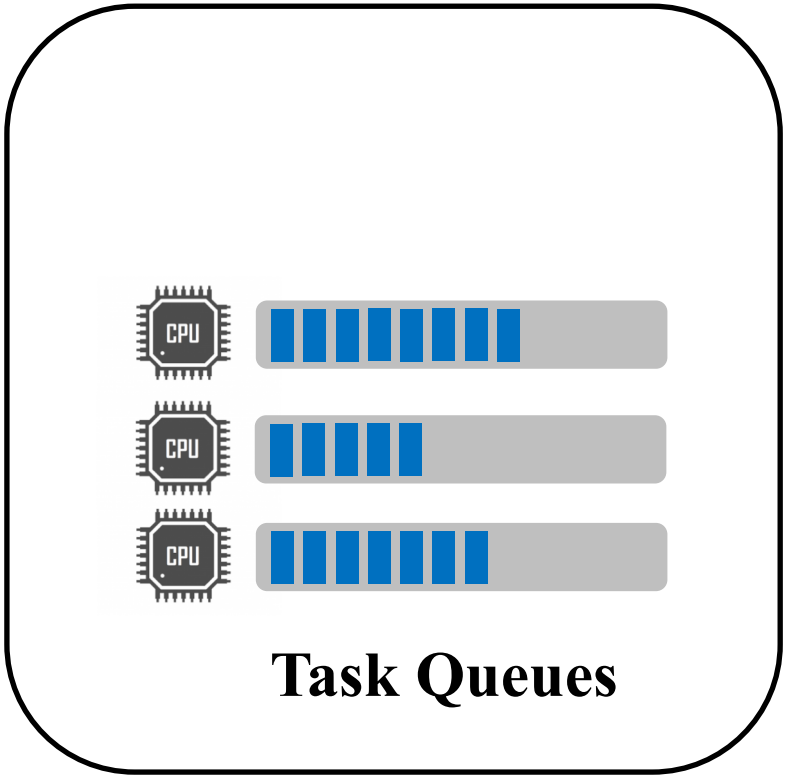


https://github.com/yanlab19870714/gthinkerQC

G-thinker+

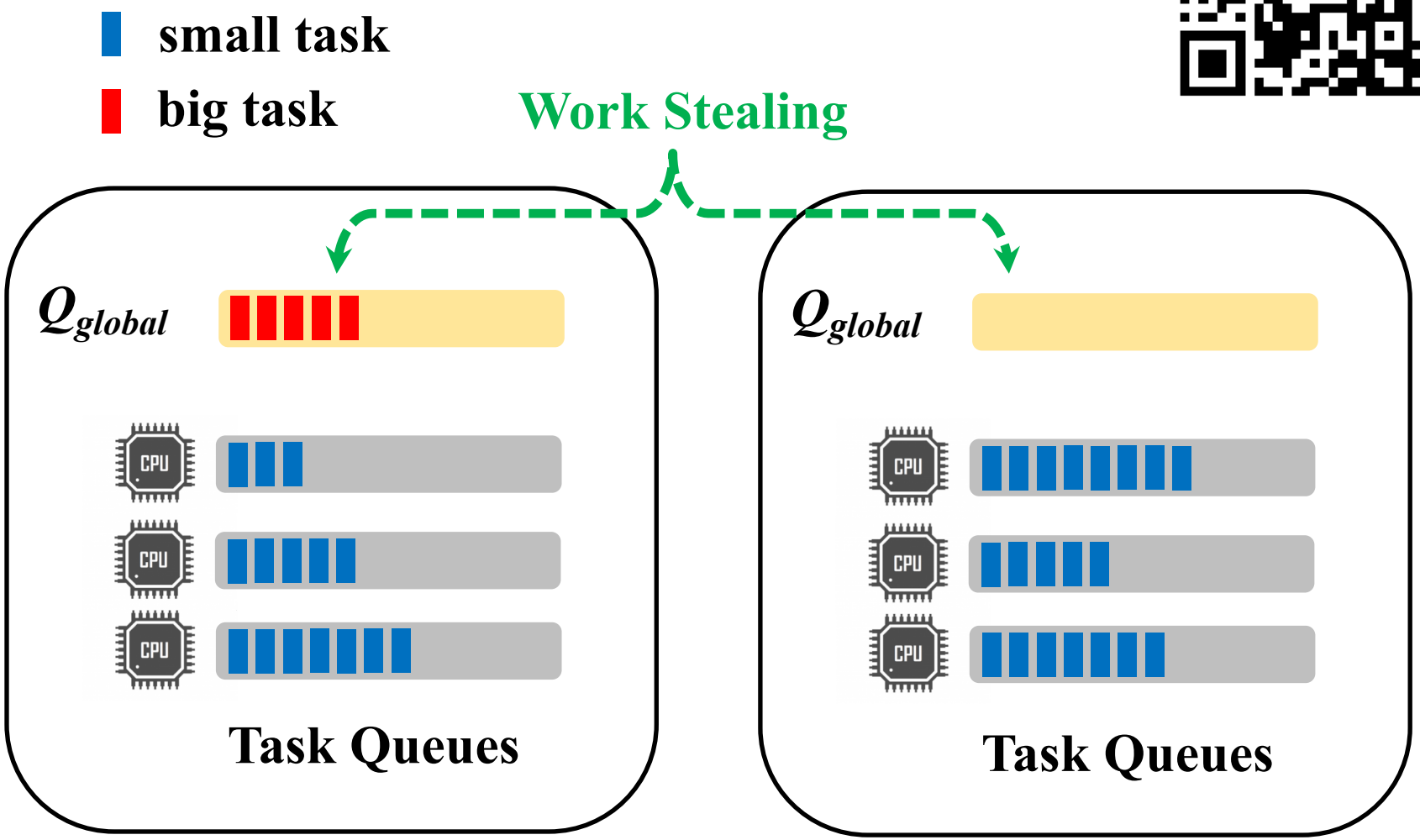
- small task
- big task





https://github.com/yanlab19870714/gthinkerQC

G-thinker+



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

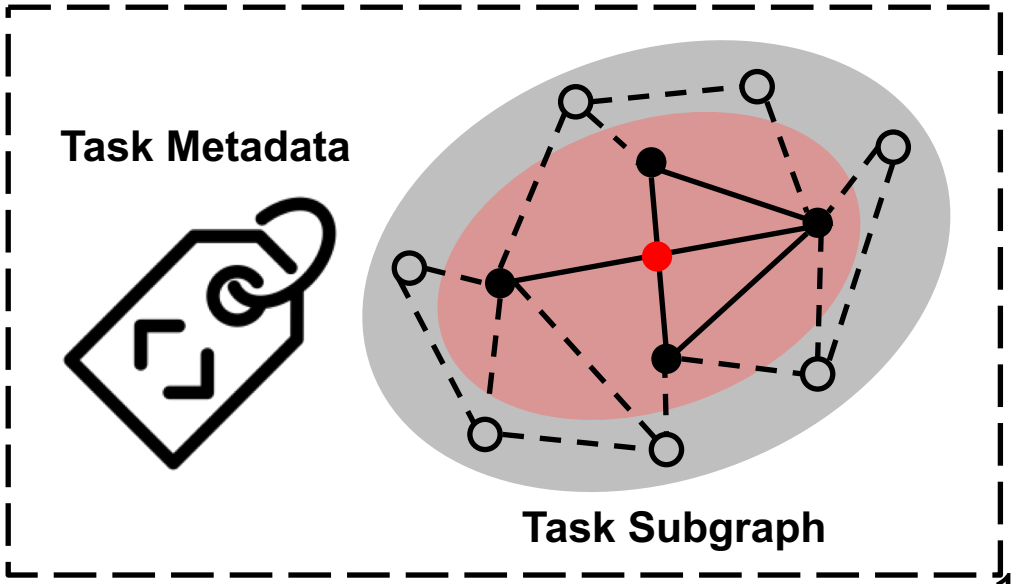
Algorithm Features

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

G-thinker

Task-Centric Programming Interface

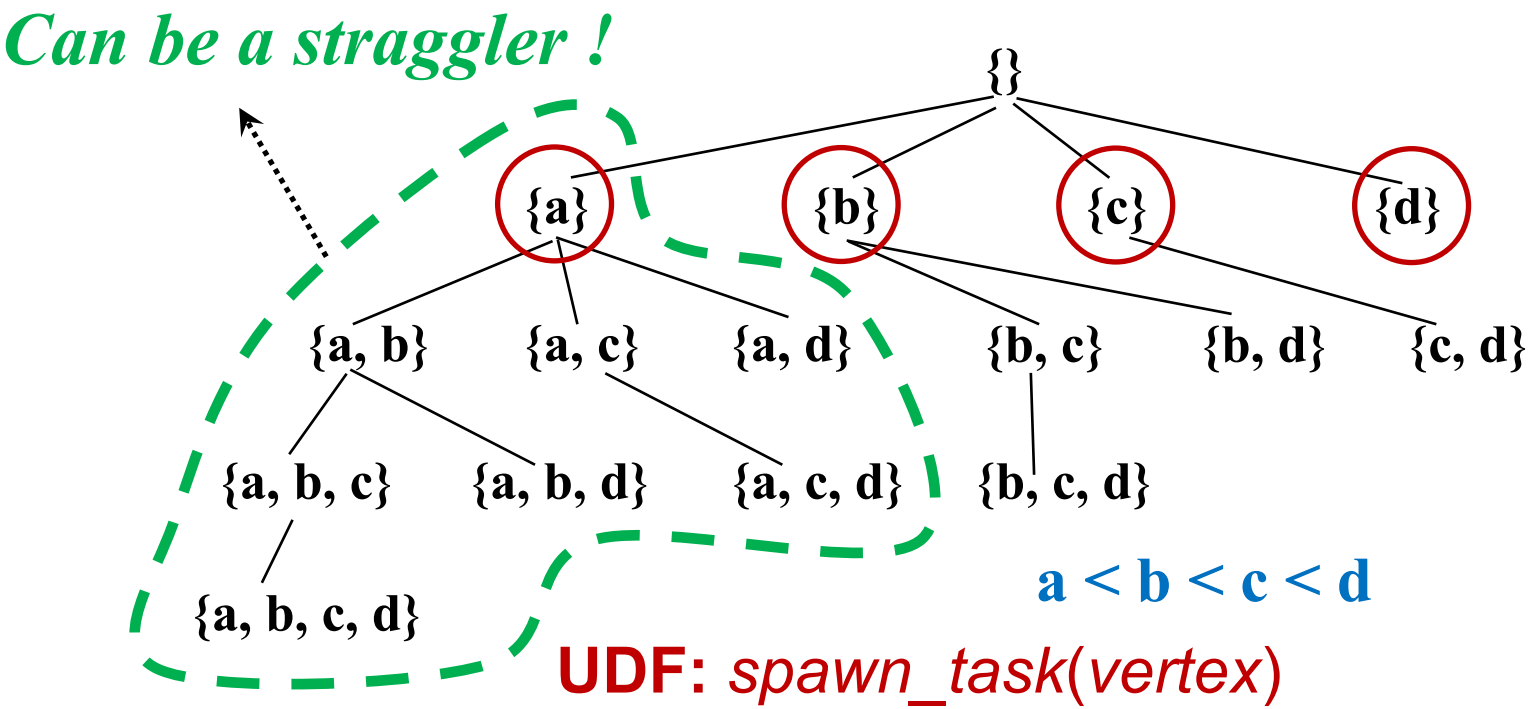
- » **UDF:** *spawn_task*(*vertex*)
- » **UDF:** *compute*(*task*, *frontier*)
- » *pull*(*vertex_id*)
- » *add_task*(*task*)



Algorithms on G-thinker

S: set of nodes already in subgraph

ext(S): set of valid candidates to add



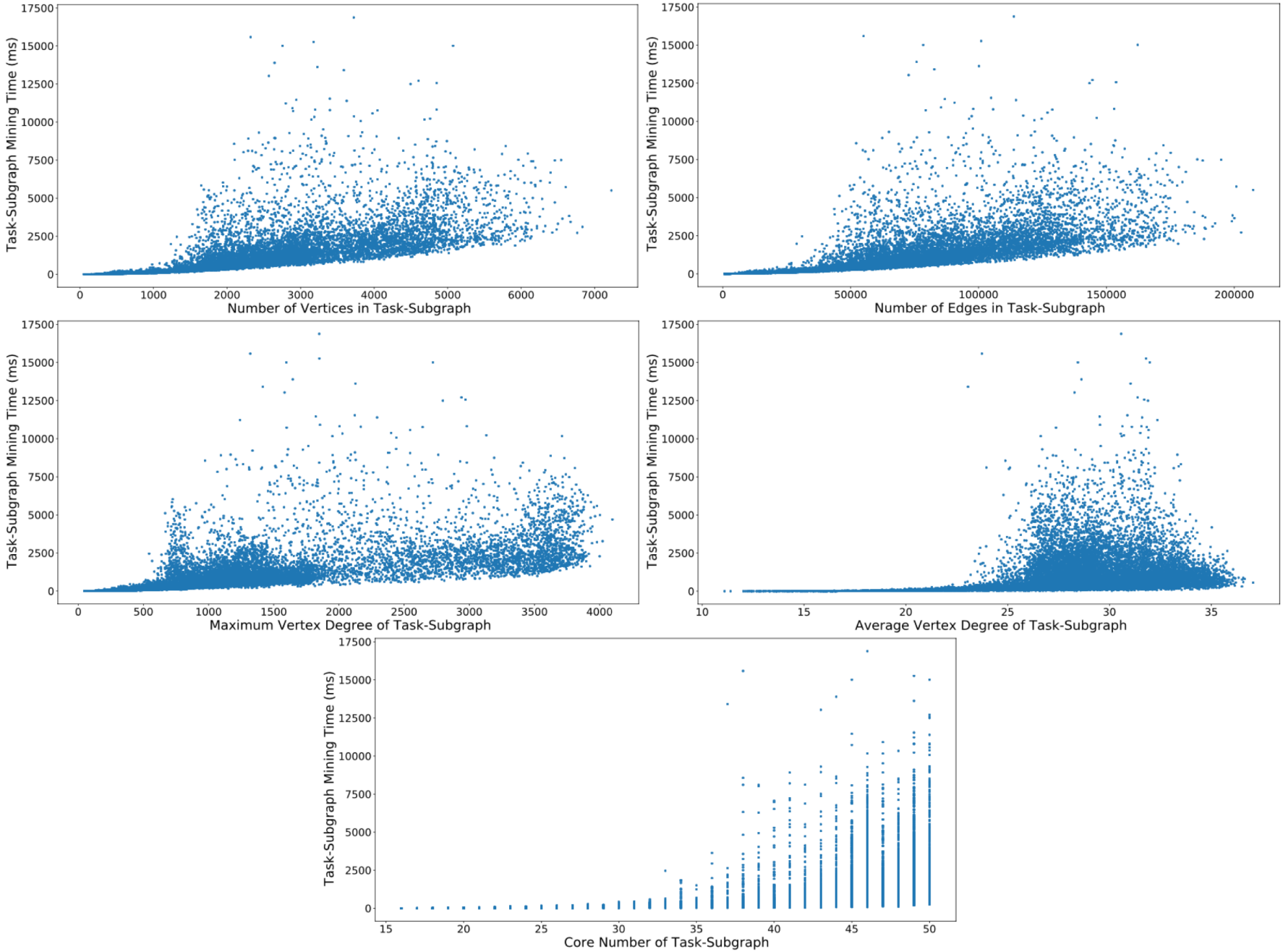


Figure 3: Subgraph Features v.s. Task Time on *YouTube*

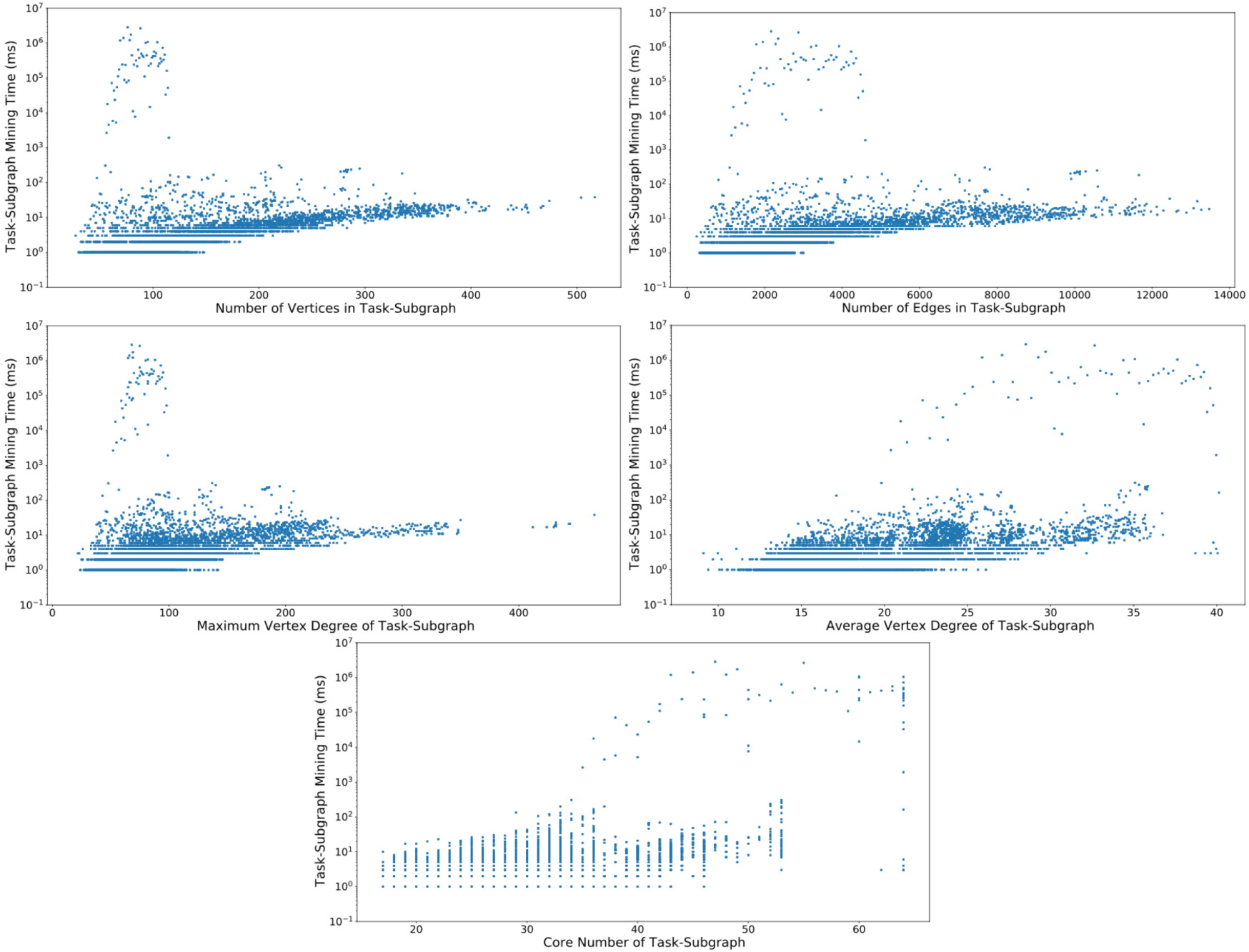
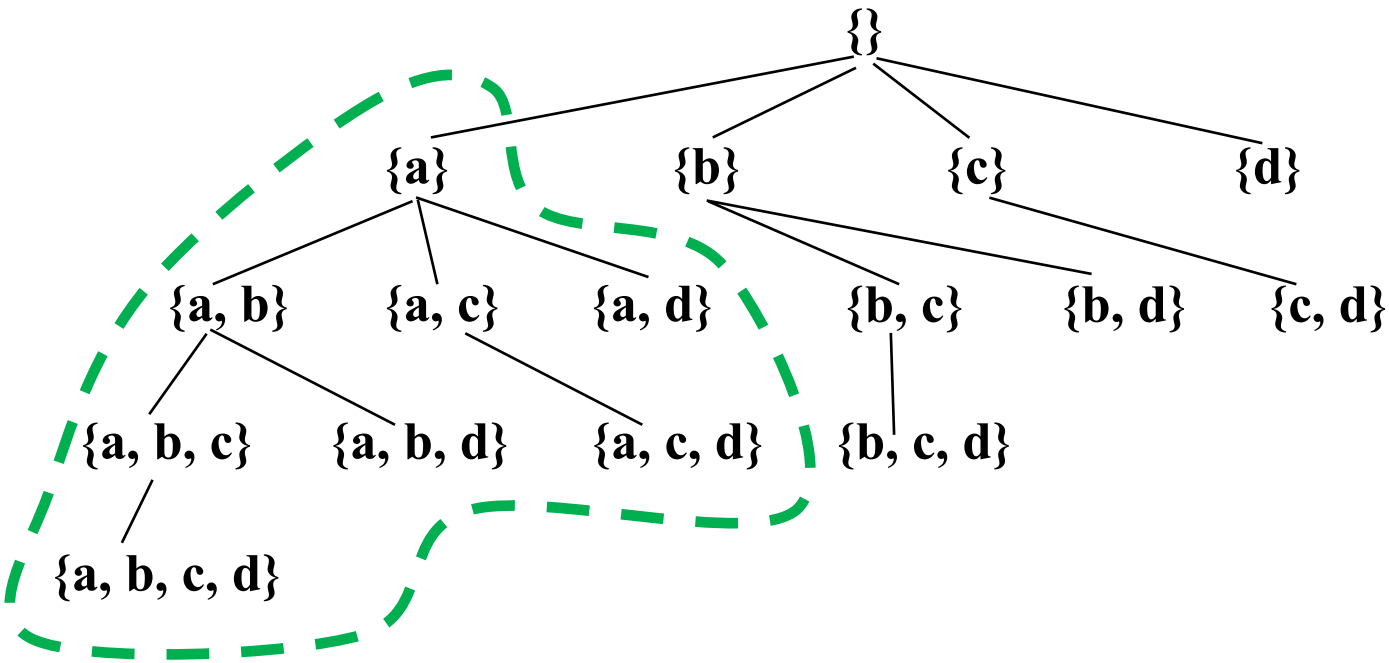


Figure 5: Subgraph Features v.s. Logarithmic Task Time on *Patent*

Algorithms on G-thinker

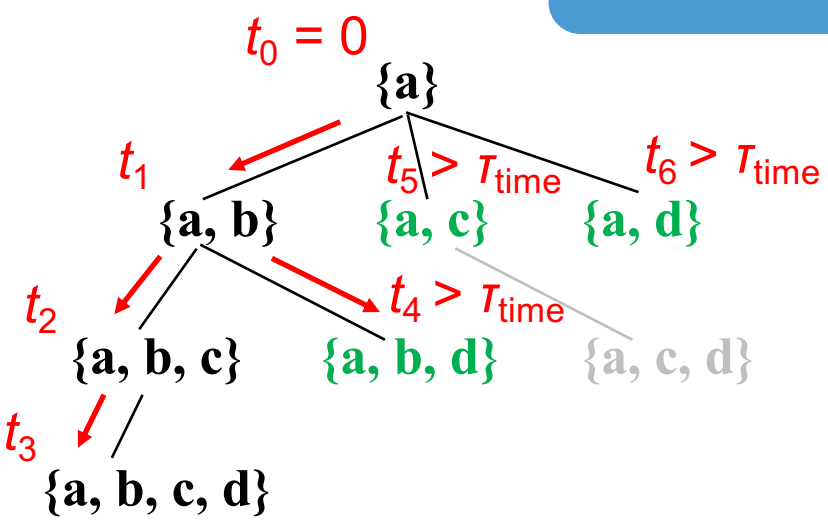
Timeout mechanism to avoid stragglers

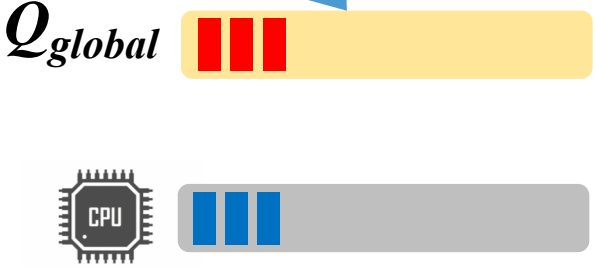


Algorithms on G-thinker

Timeout mechanism to avoid stragglers

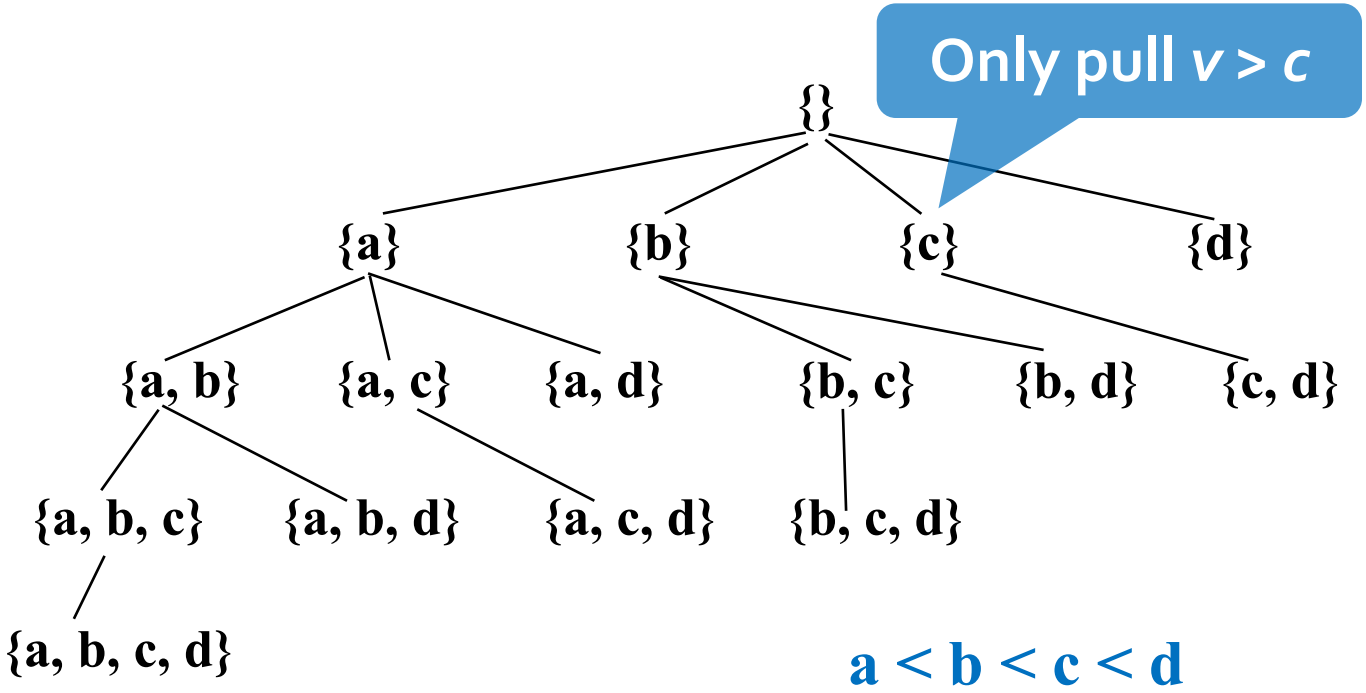
When {a, c} computes, it may timeout and decompose again





Algorithms on G-thinker

Some (*S*, *ext*(*S*))-tasks can be pruned



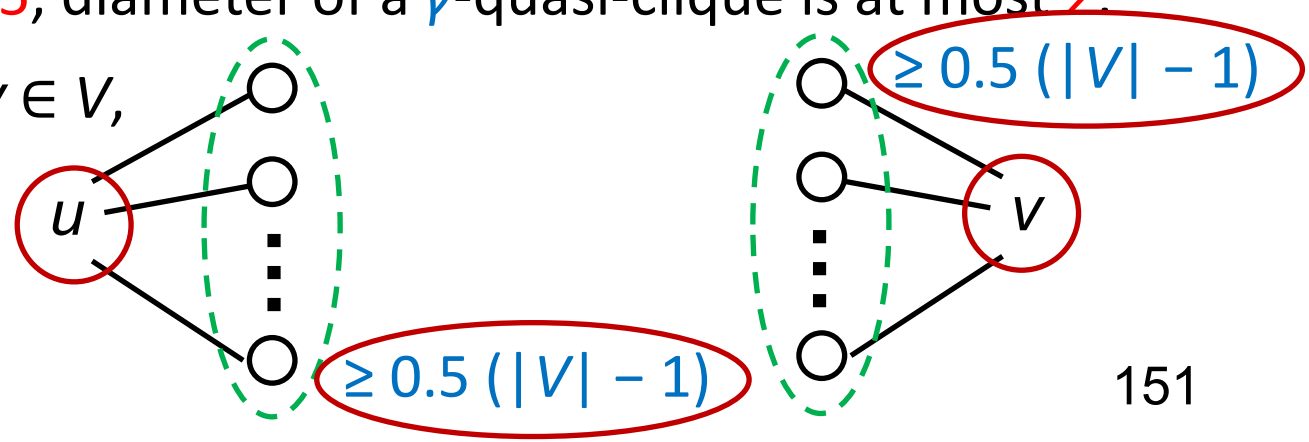
Algorithms on G-thinker

Quasi-clique:

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

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

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



Sum to $|V| + 1$

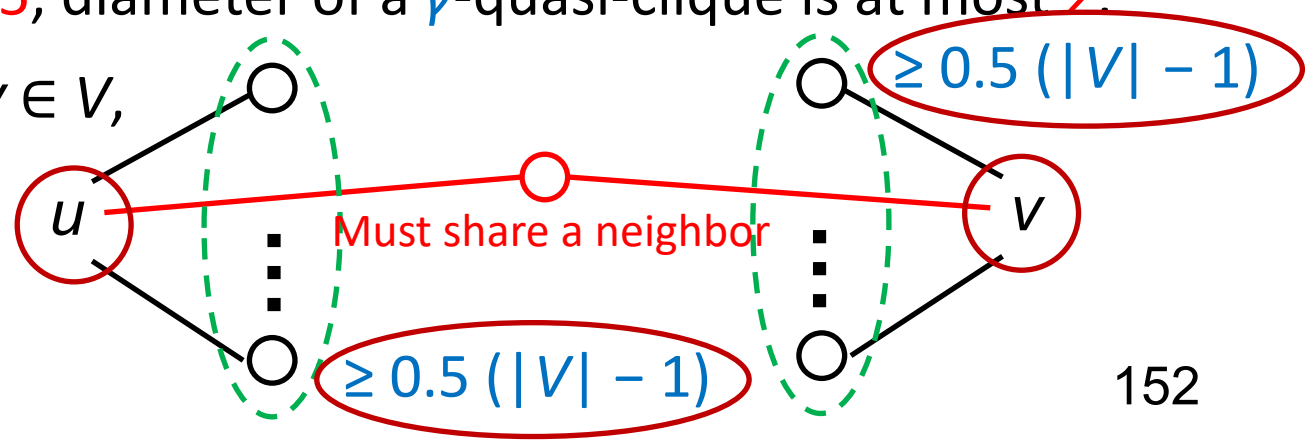
Algorithms on G-thinker

Quasi-clique:

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

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

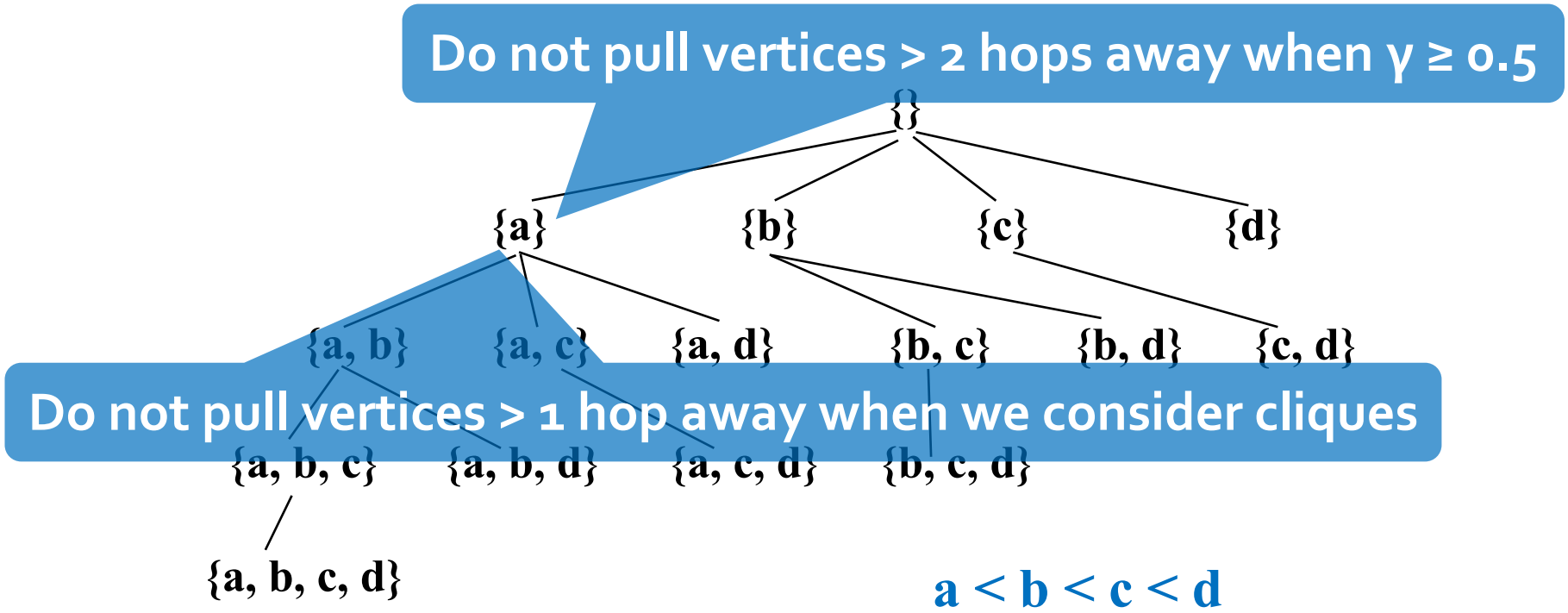
Proof: $\forall\ u, v \in V$,



Sum to $|V| + 1$

Algorithms on G-thinker

Some (*S*, *ext*(*S*))-tasks can be pruned

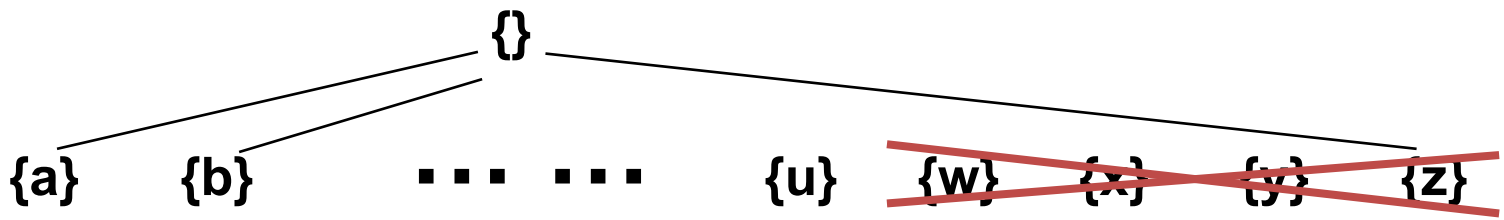


Assuming the mining of **maximal** quasi-cliques

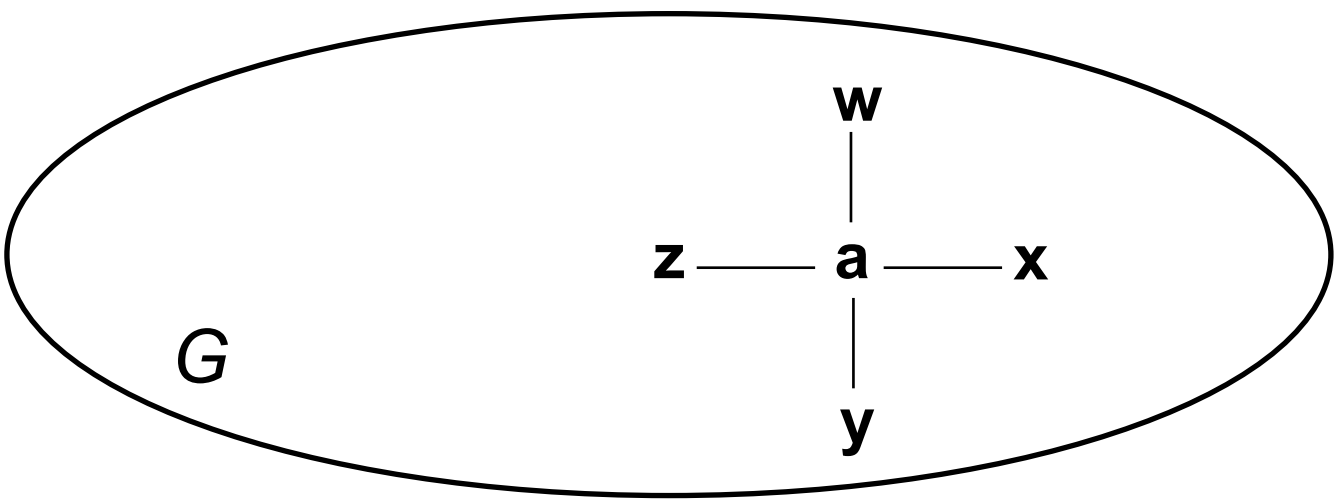
Algorithms on G-thinker

Some (*S*, *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

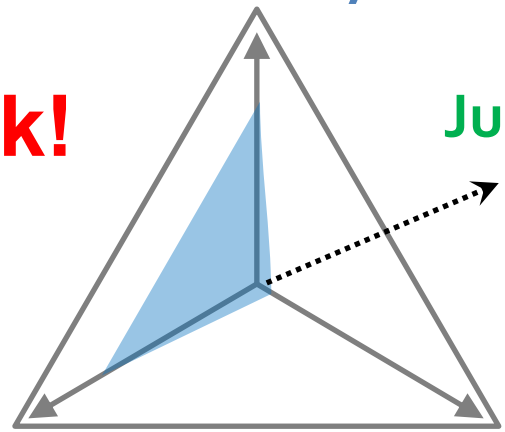
- Support of aggregator
  - Local threads update the local aggregator
  - Aggregators are synchronized every 1 second
  - If $|S| + |ext(S)|$ is less than the size of the current maximum clique found, prune task $(S, ext(S))$

Conclusion

… …

A number of papers are on their way

Follow our work!



GUO, Guimu

Contact Info:

Email: guimuguo@uab.edu

Webpage: https://guimuguo.github.io/

Thanks !

YAN, Da

Contact Info:

Email: yanda@uab.edu

Webpage: http://bit.ly/danielYan