G-thinker: A Distributed Framework for Mining Subgraphs in a Big Graph

Da Yan*1, Guimu Guo*2, Md Mashiur Rahman Chowdhury*3,
M. Tamer Özsu†4, Wei-Shinn Ku‡5, John C.S. Lui+6

*University of Alabama at Birmingham, {1yanda, 2guimuguo, 3mashiur}@uab.edu
†University of Waterloo, 4tamer.ozsu@uwaterloo.ca
‡Auburn University, 5weishinn@auburn.edu
+The Chinese University of Hong Kong, 6cslui@cse.cuhk.edu.hk

***Abstract*—Mining from a big graph those subgraphs that satisfy certain conditions is useful in many applications such as community detection and subgraph matching. These problems have a high time complexity, but existing systems to scale them are all IO-bound in execution. We propose the first truly CPU-bound distributed framework called G-thinker that adopts a user-friendly subgraph-centric vertex-pulling API for writing distributed subgraph mining algorithms. To utilize all CPU cores of a cluster, G-thinker features (1) a highly-concurrent vertex cache for parallel task access and (2) a lightweight task scheduling approach that ensures high task throughput. These designs well overlap communication with computation to minimize the CPU idle time. Extensive experiments demonstrate that G-thinker achieves orders of magnitude speedup compared even with the fastest existing subgraph-centric system, and it scales well to much larger and denser real network data. G-thinker is open-sourced at http://bit.ly/gthinker with detailed documentation.**

***Index Terms*—graph mining, subgraph-centric, CPU-bound, compute-intensive, clique, triangle, subgraph matching**

I. Introduction

Target Problem. Given a graph $G=(V,E)$ where V (resp. E) is the vertex (resp. edge) set, we consider the problem of finding those subgraphs of G that satisfy certain conditions. It may enumerate or count all these subgraphs, or simply output the largest subgraph. Examples include maximum clique finding [31], quasi-clique enumeration [17], triangle listing and counting [12], subgraph matching [15], etc. These problems have a wide range of applications including social network analysis and biological network investigation. These problems have a high time complexity (e.g., finding maximum clique is NP-hard), since the search space is the power set of V: for each subset $S \subseteq V$, we check whether the subgraph of G induced by S satisfies the conditions. Few existing algorithms can scale to big graphs such as online social networks.

Subgraph mining is usually solved by divide and conquer. A common solution is to organize the giant search space of V's power set into a set-enumeration tree [17]. Fig. 1 shows the set-enumeration tree for a graph G with four vertices $\{a,b,c,d\}$ where $a<b<c<d$ (ordered by ID). Each node in the tree represents a vertex set S, and only vertices larger than the last (and also largest) vertex in S are used to extend S. For example, in Fig. 1, node $\{a,c\}$ can be extended with d but not b as $b<c$; in fact, $\{a,b,c\}$ is obtained by extending $\{a,b\}$ with c. Edges are often used for the early pruning of a tree branch. For example, to find cliques, one only needs to extend a vertex set S with those vertices in $(V-S)$ that are common neighbors of every vertex of S, since all vertices in a clique are mutual neighbors. Also, [17] shows that to find γ-quasi-cliques ($\gamma \geq 0.5$), one only needs to extend S with those vertices that are within 2 hops from every vertex of S.

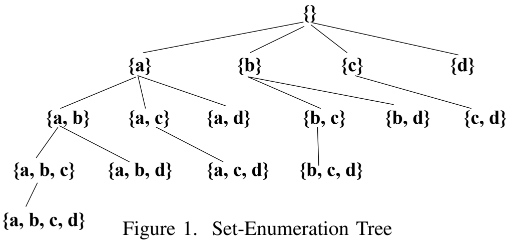


Figure 1. Set-Enumeration Tree

Problem Not Targeted. The problems we consider above share two common features: (1) pattern-to-instance: the structural or label constraints of a target subgraph (i.e., pattern) are pre-defined, and the goal is to find subgraph instances in a big graph that satisfy these constraints; (2) there exists a natural way to avoid redundant subgraph checking, such as by comparing vertex IDs in a set-enumeration tree, or partitioning by different vertex instances of the same label as in [27], [34].

Some graph-parallel systems attempt to unify the above problems with frequent subgraph pattern mining (FSM), in order to claim that their models are "more general". However, FSM is an intrinsically different problem: the patterns are not pre-defined but rather checked against the frequency of matched subgraph instances, which means that (i) the problem is in an instance-to-pattern style (not our pattern-to-instance). Moreover, frequent subgraph patterns are usually examined using pattern-growth, and to avoid generating the same pattern from different sub-patterns, (ii) expensive graph isomorphism checking is conducted on each newly generated subgraph, as in Arabesque [29], RStream [32] and Nuri [13]. This is a bad design choice since graph isomorphism checking should be totally avoided in pattern-to-instance subgraph mining. After all, FSM is a specific problem whose parallel solutions have been well-studied, be it for a big graph [28] or for many graph transactions [37], [16], and they can be directly used.

Motivations. A natural solution is to utilize many CPU cores to divide the computation workloads of subgraph mining, but

2375-026X/20/$31.00 ©2020 IEEE
DOI 10.1109/ICDE48307.2020.00122

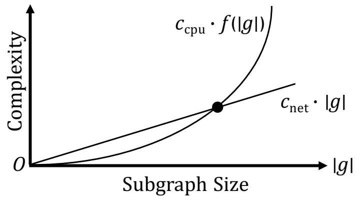


Figure 2. Costs of a Task

there is a challenge intrinsic to the nature of subgraph mining algorithms: *the number of subgraphs induced from vertex sets can be exponential to graph size* and it is impractical to keep/materialize all of them in memory; however, *out-of-core subgraph processing can be an IO bottleneck* that reduces CPU core utilization. In fact, as shall be clear in Sec. II, all existing graph-parallel systems have an IO-bound execution engine, making them inefficient for the subgraph mining.

We propose G-thinker for CPU-bound parallel subgraph mining while keeping memory consumption low. As an illustration, it takes merely 252 seconds in total and 3.1 GB memory per machine in our 15-node cluster to find the maximum clique (with 129 vertices) on the big Friendster social network of [11] containing 65.6 M vertices and 1,806 M edges. Note that the clique decision problem is NP-complete.

The success of G-thinker lies in a design that keeps CPU cores busy. Specifically, it divides the mining problem into independent tasks, e.g., represented by different tree branches in Fig. 1. Note that each tree node represents a vertex set S that are already assumed to be in an output subgraph, and incorporating more vertices into S (i.e., going down the search tree) reduces the number of other candidate vertices to consider as more structural constraints are brought by the newly added vertices. If the mining of tree branch under S is expensive, we can further divide it into child branches for parallel mining; otherwise, the whole branch can be mined by a conventional serial algorithm to keep a CPU core busy.

Each tree node in Fig. 1 thus corresponds to a task that finds qualified subgraphs assuming vertices in S are already incorporated. For such a task, let us denote g as the subgraph induced by S plus other candidate vertices (not pruned by vertices in S) to be considered for forming an output subgraph; we can thus consider the task as a mining problem on the smaller subgraph g rather than the input graph. Using divide and conquer, g shrinks as we move down the set-enumerate search tree. Now consider Fig. 2, where we denote the size of g by $|g|$: (1) the IO cost of materializing g by collecting vertices and edges is linear to $|g|$; and (2) the CPU cost of mining g increases quickly with $|g|$ since the mining algorithm has a high time complexity. Thus, even though network is slower than CPUs, the CPU cost of mining g surpasses the IO cost to construct g when $|g|$ is not too small. This enables hiding IO cost inside the concurrent CPU processing when overlapped.

To overlap computation and communication, *G-thinker keeps a pool of active tasks for processing at any time, so that while some tasks are waiting for their data (to construct subgraph g), other tasks (e.g., with g already constructed) can continue their computation (i.e., mining) to keep CPU cores busy.* This approach also bounds memory cost as only a bounded pool of tasks is in memory, refilled with new tasks only when there are insufficient tasks to keep CPU cores busy.

Contributions. The contributions of G-thinker are as follows:

- The framework design satisfies all 7 desirabilities established in Sec. III necessary for scalability and efficiency.
- A novel vertex cache design is proposed to support highly-concurrent vertex accesses by tasks.
- A lightweight task scheduling workflow is designed with low scheduling overhead, which is able to balance the workloads and minimize CPU idle time.
- An intuitive subgraph-centric API allows programmers to use task-based vertex pulling to write mining algorithms.
- We open-sourced G-thinker at http://www.cs.uab.edu/yanda/gthinker with detailed documentation, and have conducted extensive experiments on Microsoft Azure to evaluate its superior scalability and efficiency.

The rest of this paper is organized as follows. Sec. II reviews existing graph-parallel systems and explains why they are IO-bound. Sec. III then establishes 7 desirable features for a scalable and efficient subgraph mining system, and overviews G-thinker's system architecture that meets all the 7 features. Sec. IV introduces the adopted subgraph-centric programming API, and Sec. V describes the system design focusing on the two pillars to achieve CPU-bound performance, i.e., vertex cache and task management. Finally, Sec. VI reports the experimental results and Sec. VII concludes this paper.

II. Related Work

This section explains the concepts of IO-bound and CPU-bound workloads, and reviews existing graph-parallel systems.

IO-bound v.s. CPU-bound. The throughput of CPU computation is usually much higher than the IO throughput of disks and the network. However, existing Big Data systems dominantly target IO-bound workloads. For example, the word-count application of MapReduce [10] emits every word onto the network, and for each word that a reducer increments its counter, the word needs to be received by the reducer first. Similarly, in the PageRank application of Pregel [18], a vertex needs to first receive a value from its in-neighbor, and then simply adds it to the current PageRank value. IO-bound execution can be catastrophic for computing problems beyond low-complexity ones. For example, even for triangle counting whose time complexity $O(|E|^{1.5})$ is not very high, [9] reported that the state-of-the-art MapReduce algorithm uses 1,636 machines and takes 5.33 minutes on a small graph, on which their single-threaded algorithm uses < 0.5 minute.

In fact, McSherry et. al [20] have noticed that existing systems are comparable and sometimes slower than a single-threaded program. In another recent post by McSherry [1], he further indicated that the current distributed implementations "scale" (i.e., using aggregate IO bandwidth) but performance does not get to "a simple single-threaded implementation".

Categorization of Graph-Parallel Systems. Our book [33] has classified graph-parallel systems into vertex-centric systems, subgraph-centric systems and others (e.g., matrix-based).

Vertex-centric systems compute one value for each vertex (or edge), and the output data volume is linear to that of the input graph. In contrast, subgraph-centric systems output subgraphs that may overlap, and the output data volume can be exponential to that of the input graph. Note that based on this categorization, block-centric systems such as Blogel [35] and Giraph++ [30] are merely extensions to vertex-centric systems.

Vertex-Centric Systems. Pioneered by Google's Pregel [18], a number of distributed systems have been proposed for simple iterative graph processing [19]. They advocate a think-like-a-vertex programming model, where vertices communicate with each other by message passing along edges to update their states. Computation repeats in iterations until the vertex states converge. In these systems, the number of messages transmitted in an iteration is usually comparable to the number of edges in the input graph, making the workloads communication-bound. To avoid communication, single-machine vertex-centric systems emerge [14], [25], [7] by streaming vertices and edges from disk to memory for batched state updates; their workloads are still disk IO-bound. The vertex-centric programming API is also not convenient for writing subgraph mining algorithms that operate on subgraphs.

Subgraph-Centric Systems. Recently, a few systems began to explore a think-like-a-subgraph programming model for mining subgraphs including distributed systems NScale [23], Arabesque [29] and G-Miner [6], and single-machine systems RStream [32] and Nuri [13]. Despite more convenient programming interfaces, their execution is still IO-bound.

Assume that subgraphs of diameter k around individual vertices need to be examined, then **NScale** (i) first constructs those subgraphs through BFS around each vertex, implemented as k rounds of MapReduce computation to avoid keeping the numerous subgraphs in memory; (ii) it then mines these subgraphs in parallel. This design requires that all subgraphs be constructed before any of them can begin its mining, leading to poor CPU utilization and the straggler's problem.

Arabesque [29] is a distributed system where every machine loads the entire input graph into memory, and subgraphs are constructed and processed iteratively. In the i-th iteration, Arabesque expands the set of subgraphs with i edges/vertices by one more adjacent edge/vertex, to construct subgraphs with $(i+1)$ edges/vertices for processing. New subgraphs that pass a filtering condition are further processed and then passed to the next iteration. For example, to find cliques, the filtering condition checks whether a subgraph g is a clique; if so, g is passed to the next iteration to grow larger cliques. Obviously, Arabesque materializes subgraphs represented by all nodes in the set-enumeration tree (recall Fig. 1) in a BFS manner which is IO-bound. As an in-memory system, Arabesque attempts to compress the numerous materialized subgraphs using a data structure called ODAG, but it does not address the scalability limitation as the number of subgraphs grows exponentially.

The current paper is the conference submission of our **G-thinker** preprint [34] with the execution engine design significantly improved. Our preprint has briefly described our task-based vertex pulling API, where tasks are spawned from individual vertices, and a task can grow its associated subgraph by requesting adjacent vertices and edges for subsequent mining. The API is then followed by **G-Miner** [6], as indicated by the statement below Figure 1 of [6]: "The task model is inspired by the task concept in G-thinker". The old G-thinker prototype is to verify that our new API can significantly improve the mining performance compared with existing systems, but the execution engine is a simplified IO-bound design that does not even consider multithreading; it runs multiple processes in each machine for parallelism which cannot share data.

G-Miner adds multithreading support to our old prototype to allow tasks in a machine to share vertices, but the design is still IO-bound. Specifically, the threads in a machine share a common list called *RCV cache* for caching vertex objects which becomes a bottleneck of task concurrency. G-Miner also requires graph partitioning as a preprocessing job, but real big graphs often do not have a small cut and are expensive to partition; we thus adopt the approach of Pregel to hash vertices to machines by vertex ID to avoid this startup overhead.

All tasks in G-Miner are generated at the beginning (rather than when task pool has space as we do) and kept in a disk-resident priority queue. Each task t in the queue is indexed by a key computed via locality-sensitive hashing (LSH) on its set of requested vertices, to let nearby tasks in the queue share requested vertex objects to maximize data reuse. Unfortunately, this design does more harm than good: because tasks are not processed in the order of their generation (but rather LSH order), an enormous number of tasks are buffered in the disk-resident task queue since some partially computed tasks are sitting at the end of the queue while new tasks are dequeued to expand their subgraphs. Thus, reinserting a partially processed task into the disk-resident task queue for later processing becomes the dominant cost for a large graph.

RStream [32] is a single-machine out-of-core system which proposes a so-called GRAS model to simulate Arabesque's filter-process model, utilizing relational joins. Their experiments show that RStream is several times faster than Arabesque even though it uses just one machine, but the improvement is mainly because of eliminating network overheads (recall that Arabesque materializes subgraphs represented by all nodes in a set-enumeration tree). Also, the execution of RStream is still IO-bound as it is an out-of-core system.

Nuri [13] aims to find the k most relevant subgraphs using only a single computer, by prioritized subgraph expansion. However, since the subgraph expansion is in a best-first manner (Nuri is single-threaded), the number of buffered subgraphs can be huge, and their on-disk subgraph management can be IO-bound. As Sec. VI shall show, RStream and Nuri are not anywhere close to G-thinker running on a single machine.

III. G-THINKER OVERVIEW

Fig. 3 shows the architecture of G-thinker on a cluster of 3 machines. We assume that a graph is stored as a set of vertices, where each vertex v is stored with its adjacency list $\Gamma(v)$ that keeps its neighbors. G-thinker loads an input graph from the

Hadoop Distributed File System (HDFS). As Fig. 3 shows, each machine only loads a fraction of vertices along with their adjacency lists into its memory, kept in a local vertex table. Vertices are assigned to machines by hashing their vertex IDs, and the aggregate memory of all machines is used to keep a big graph. The local vertex tables of all machines form a distributed key-value store where any task can request for $\Gamma(v)$ using v's ID.

G-thinker computes in the unit of tasks, and each task is associated with a subgraph g that it constructs and mines upon. For example, consider the problem of mining maximal γ-quasi-cliques ($\gamma \geq 0.5$) for which [17] shows that any two vertices in a γ-quasi-clique must be within 2 hops. One may spawn a task from each individual vertex v, request for its neighbors (in fact, their adjacency lists) in Iteration 1, and when receiving them, request for the 2nd-hop neighbors (in fact, their adjacency lists) in Iteration 2 to construct the 2-hop ego-network of v for mining maximal quasi-cliques using a serial algorithm like that of [17]. To avoid double-counting, a vertex v only requests those vertices whose ID is larger than v (recall from Fig. 1), so that a quasi-clique whose smallest vertex is u must be found by the task spawned from u.

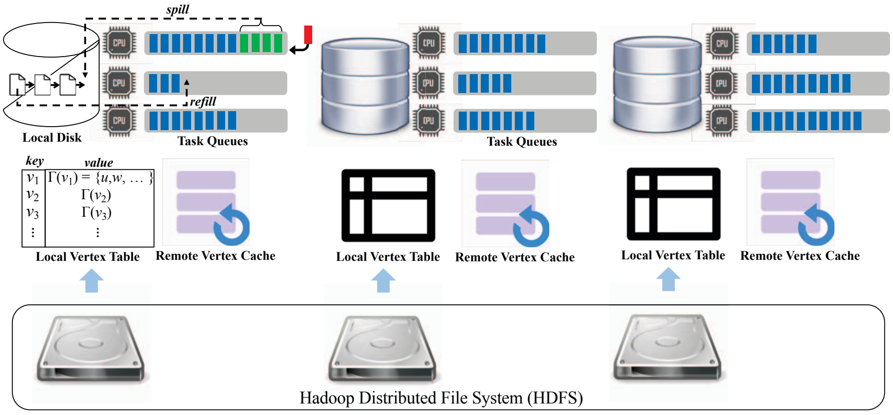


Figure 3. G-thinker Architecture Overview

Such a subgraph mining algorithm is implemented by specifying 2 user-defined functions (UDFs): (1) *spawn*(v) indicating how to spawn a task from each individual vertex in the local vertex table; (2) *compute*(*frontier*) indicating how a task processes an iteration where *frontier* keeps the adjacency list of the requested vertices in the previous iteration. In a UDF, users may request the adjacency list of a vertex u to expand the subgraph of a task, or even decompose the subgraph by creating multiple new tasks to divide the mining workloads.

As Fig. 3 shows, each machine also maintains a remote vertex cache to keep the requested vertices (and their adjacency lists) that are not in the local vertex table, for access by tasks via the input argument *frontier* to UDF *compute*(*frontier*). This allows multiple tasks to share requested vertices to minimize redundancy, and once a vertex in the cache is no longer requested by any task in the machine, it can be evicted to make room for other requested vertices. In UDF *compute*(*frontier*), a task is supposed to save the needed vertices and edges in *frontier* into its subgraph, as the vertices in *frontier* are released by G-thinker right after *compute*(.) returns.

To maximize CPU core utilization, each mining thread keeps a task queue of its own to stay busy and to avoid contention. Since tasks are associated with subgraphs that may overlap, it is infeasible to keep all tasks in memory. G-thinker only keeps a pool of active tasks in memory at any time by controlling the pace of task spawning. If a task is waiting for its requested vertices, it is suspended so that the mining thread can continue to process the next task in its queue; the suspended task will be added back to the queue once all its requested vertices become locally available, in which case we say that the task is **ready**.

Note that a task queue can become full if a task generates many subtasks into its queue, or if many waiting tasks become ready all at once (due to other machines' responses). To keep the number of in-memory tasks bounded, if a task queue is full but a new task is to be inserted, we spill a batch of tasks at the end of the queue as a file to local disk to make room.

As the upper-left corner of Fig. 3 shows, each machine maintains a list of task files spilled from the task queues of mining threads. To minimize the task volume on disks, when a thread finds that its task queue is about to become empty, it will first refill tasks into the queue from a task file (if it exists), before choosing to spawn more tasks from vertices in local vertex table. Note that tasks are spilled to disks and loaded back in batches to minimize the number of random IOs and lock-contention by mining threads on the task file list.

For load balancing, machines about to become idle will steal tasks from busy ones (could be spawned from their local vertex table) by prefetching a batch of tasks and adding them to the task file list on local disk. The tasks will be loaded by a mining thread for processing when its task queue needs a refill.

Desirabilities. Our design always guarantees that a mining thread has enough tasks in its queue to keep itself busy (unless the job has no more tasks to refill), and since each task has sufficient CPU-heavy mining workloads, the linear IO cost of fetching/moving data is seldom a bottleneck.

Other desirabilities include: (1) bounded memory consumption: only a pool of tasks is kept in memory at any time, local vertex table only keeps a partition of vertices, and remote vertex cache has a bounded capacity; (2) tasks spilled from task queues are written to disks (and loaded back) in batches to achieve serial disk IO, and spilled tasks are prioritized when refilling task queues of mining threads so that the number of tasks kept on disks is minimized (in fact negligible according to our experiments); (3) threads in a machine can share vertex data in the remote vertex cache, to avoid redundant vertex requesting; (4) in contrast, tasks

Table I
Feature Comparison of Subgraph-Centric Systems

	0. An in-memory task pool for timely computation	1. Bounded memory consumption	2. Task spilling and keeping number of tasks on disk small	3. Vertex sharing to avoid redundant communication	4. Tasks execute independently	5. Batched communication	6. Load balancing
G-thinker	✓	✓	✓	✓	✓	✓	✓
NScale	x	✓	x	x	x	✓	x
Arabesque	✓	x	x	✓	x	✓	✓
G-Miner	✓	✓	x	✓	x	✓	✓
RStream	x	✓	x	N/A	x	N/A	N/A
Nuri	✓	✓	x	N/A	x	N/A	N/A

are totally independent (due to divide-and-conquer logic) and will never block each other; (5) we also batch vertex requests and responses for transmission to combat round-trip time and to ensure throughput; (6) if a big task is divided into many tasks, these tasks will be spilled to disks to be refilled to the task queues of multiple mining threads for parallel processing; moreover, work stealing among machines will send tasks from busy machines to idle machines for processing.

We remark that G-thinker is the only system that achieves these desirabilities and hence CPU-bound mining workloads. Table I summarizes how existing subgraph-centric systems compare with G-thinker in terms of these desirabilities.

Challenges. To achieve the above desirabilities, we address the following challenges. For **vertex caching**, we need to ask the following questions: (1) how can we ensure high concurrency of accessing vertex cache by mining threads, while inserting newly requested vertices and tracking whether an existing vertex can be evicted; (2) how can we guarantee that a task will not request for the adjacency list of a vertex v which has been requested by another task in the same machine (even if response $\Gamma(v)$ has not been received) to avoid redundancy.

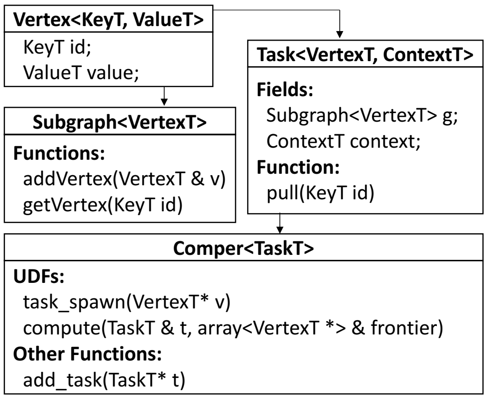
For **task management**, we need to consider (1) how to accommodate tasks that are waiting for data, (2) how can those tasks be timely put back to task queues when their data become ready, and (3) how to minimize CPU occupancy due to task scheduling. We will look at our solution to the above issues in Sec. V, after we present G-thinker API in Sec. IV below.

IV. Programming Model

Without loss of generality, assume that an input graph $G = (V, E)$ is undirected. Throughout this paper, we denote the set of neighbors of a vertex v by $\Gamma(v)$, and denote those in $\Gamma(v)$ whose IDs are larger than v by $\Gamma_{>}(v)$. We also abuse the notation v to mean the singleton set $\{v\}$. Below, we first introduce concepts including ***pull, task, comper*** and ***worker***.

Recall from Fig. 3 that G-thinker loads an input graph from HDFS into a distributed memory store where a task can request $\Gamma(v)$ by providing v's ID. Here, we say that the task **pulls** v.

Different tasks are independent, while each task t performs computation in iterations. If t needs to wait for data after an iteration, it is suspended to release CPU core. Another iteration of t will be scheduled once all its responses are received.



```
Vertex<KeyT, ValueT>
  KeyT id;
  ValueT value;

Task<VertexT, ContextT>
Fields:
  Subgraph<VertexT> g;
  ContextT context;
Function:
  pull(KeyT id)

Subgraph<VertexT>
Functions:
  addVertex(VertexT & v)
  getVertex(KeyT id)

Comper<TaskT>
UDFs:
  task_spawn(VertexT* v)
  compute(TaskT & t, array<VertexT *> & frontier)
Other Functions:
  add_task(TaskT* t)
```

Figure 4. Programming Interface

A process called **worker** is run on each machine, which in turn runs multiple mining threads called **compers**. Fig. 3 shows that each comper maintains its own task queue, denoted by Q_{task} hereafter. Given these concepts, let us see the API.

Programming Interface. G-thinker is written in C++. Its programming interface hides away execution details, and users only need to properly specify the data types and implement serial user-defined functions (UDFs) with application logic.

Fig. 4 sketches the core API including four classes to customize. **(1) Vertex**: each *Vertex* object v maintains an ID and a value field (which usually keeps v's adjacency list). **(2) Subgraph** provides the abstraction of a subgraph. **(3) Task**: a *Task* object maintains a subgraph g and another field *context* for keeping other contents. A task also provides a function pull(v) to request $\Gamma(v)$ for use by the task in the next iteration.

The classes introduced so far have no UDF and users only need to specify the C++ template arguments and to rename the new type using "typedef" for ease of use. In contrast, **(4) Comper** is a class that implements a comper thread, and provides two UDFs: (i) **task_spawn(v)**, where users may create tasks from a vertex v, and call add_task(t) to add each created task t to Q_{task}. (ii) **compute(t, frontier)**, which specifies how an existing task t is processed for one iteration; compute(.) returns *true* if another iteration of it should be called on task t, and *false* if the task is finished; input argument *frontier* is an array of previously requested vertices. When compute(.) is called, the adjacency lists of vertices in *frontier* should have been pulled to the local machine, and a task may expand its subgraph g by incorporating the pulled data and then continue to pull the neighbors of a vertex in *frontier*.

If g is too big, in compute(.) users may further decompose the task and add the generated tasks to the comper's Q_{task} by calling add_task(.). These tasks may be spilled to disk due to Q_{task} being full, and then fetched by other compers or workers.

There are also additional customizable classes omitted in Fig. 4. For example, **(5) Worker<ComperT>** implements a worker process, and provides UDFs for data import/export (e.g., how to parse a line on HDFS into a vertex object). **(6) Aggregator** allows users to aggregate results computed by tasks. In *Comper*::compute(.), users can let a task aggregate data to the aggregator or get the current aggregated value. If aggregator is enabled, each worker runs an aggregator thread, and these threads at all workers synchronize the aggregated values periodically at a user-specified frequency (1 s by default). Before job terminates, another synchronization is performed to make sure data from all tasks are aggregated.

We use aggregator in many applications: to find maximum cliques, aggregator tracks the maximum clique currently found for compers to prune search space; while in triangle counting, each task can sum the number of triangles currently found to a local aggregator in its machine; these local counts are periodically summed to get the current total count for reporting.

Users can also trim the adjacency list of each vertex using a **(7) Trimmer** class. For example, for subgraph matching, vertices and edges (i.e., items in $\Gamma(v)$) in the data graph whose labels do not appear in the query graph can be safely pruned. Also, when following a search tree as in Fig. 1, we can trim each vertex v's adjacency list $\Gamma(v)$ into $\Gamma_>(v)$ since a vertex set S is always expanded by adjacent vertices with larger IDs.

If enabled, trimming is performed right after graph loading, so that later during vertex pulling, only trimmed adjacency lists are responded back in order to reduce communication.

An Application: Maximum Clique Finding. We next illustrate how to write a G-thinker program for the problem of finding a maximum clique following the search tree in Fig. 1.

We denote a task by $\langle S, ext(S)\rangle$, where S is the set of vertices already included in a subgraph g, and $ext(S)$ is the set of vertices that can extend g into a clique. Since vertices in a clique are mutual neighbors, a vertex in $ext(S)$ should be the common neighbor of all vertices in S. Initially, top-level tasks are $\langle S, ext(S)\rangle = \langle v, \Gamma_>(v)\rangle$, one for each vertex v.

Let us generalize our notations to denote the common neighbors of vertices in set S by $\Gamma(S)$, and denote those in $\Gamma(S)$ with IDs are larger than all vertices in S by $\Gamma_>(S)$. Since vertices in a clique are mutual neighbors, we can recursively decompose a task $\langle S, \Gamma_>(S)\rangle$ (note that $ext(S) = \Gamma_>(S)$) into $|ext(S)|$ subtasks: $\langle S \cup u,\ \Gamma_>(S) \cap \Gamma_>(u)\rangle$, one for each $u \in ext(S)$.

To process task $\langle S, ext(S)\rangle$, one only needs to mine the subgraph induced by $ext(S)$ (denoted by g) for its cliques, because for any clique C found in g, $C \cup S$ is a clique of G.

Based on the above idea, *Comper*'s 2 UDFs are sketched in Fig. 5, where we denote the vertex set of a subgraph g by $V(g)$. Also, for a task $t = \langle S, ext(S)\rangle$ in our problem, $t.context$ keeps S, i.e., the set of vertices already assumed to be in a clique to find. We thus directly use $t.S$ instead of $t.context$. We assume that an aggregator maintains the maximum clique currently found, and we denote its vertex set by S_{max}. We also assume that for any vertex v, $\Gamma(v)$ has been trimmed as $\Gamma_>(v)$.

First consider task_spawn(v) in Fig. 5, which directly exits if v cannot form a clique larger than S_{max} even if all v's neighbors are included (Line 1), where S_{max} is obtained from the aggregator. If not pruned, a task t is created (Line 2) which corresponds to a top-level task $\langle S, ext(S)\rangle = \langle v, \Gamma_>(v)\rangle$. Line 3 sets $t.S$ set as $\{v\}$. To construct $t.g$ as the subgraph induced by $\Gamma_>(v)$, t requires the edges of vertices in $\Gamma_>(v)$ and thus it pulls these vertices (Line 4). Finally, the task is added to task queue Q_{task} of the comper that calls task_spawn(v) (Line 5).

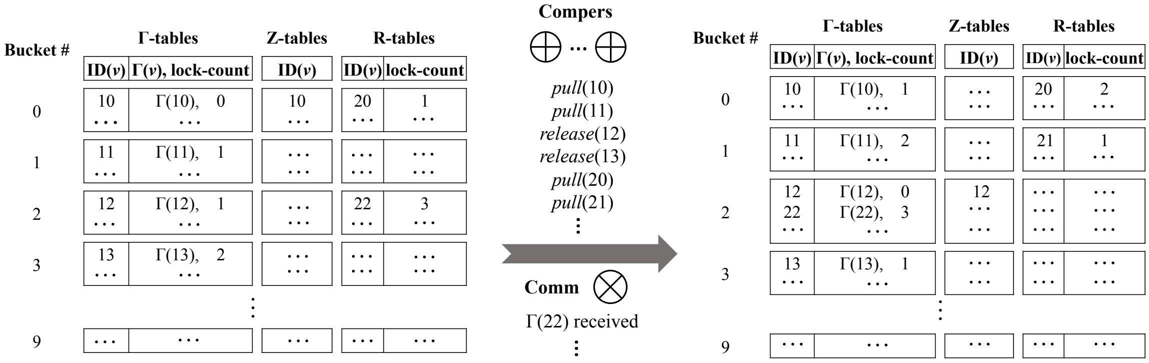
```
Comper::task_spawn(v)
1: if $|S_{max}| \geq 1 + |\Gamma_>(v)|$ then return
2: create a task $t$   // $t = <v, \Gamma_>(v)>$
3: $t.S \leftarrow \{v\}$
4: for each $u \in \Gamma_>(v)$ do $t$.pull($u$)
5: add_task($t$)

Comper::compute(t, frontier)
1  : if $|t.S| = 1$ then   // $t.S = \{v\}$
         // frontier contains $\Gamma_>(u)$ for all $u \in \Gamma_>(v)$
2  :     construct $t.g$ as the subgraph of $G$ induced by $\Gamma_>(v)$
       // now consider a general task $t = <S, \Gamma_>(S)>$
3  : if $|V(t.g)| > \tau$ then
4  :     for each $u \in V(t.g)$ do
5  :         create a task $t'$
6  :         $t'.S \leftarrow t'.S \cup \{u\}$
7  :         construct $t'.g$ as the subgraph of $t.g$ induced by $\Gamma_>(t.S \cup u)$
             // here, $\Gamma_>(t.S \cup u)$ is $u$'s "filtered" adjacency list $\Gamma_>(u)$ in $t.g$
8  :         if $|t'.S| + |V(t'.g)| > |S_{max}|$ then add_task($t'$)
9  :         else delete $t'$
10: else // $t.g$ has no more than $\tau$ vertices
11:     if $|t.S| + |V(t.g)| \leq |S_{max}|$ then return false
12:     $S'_{max} \leftarrow$ run serial algorithm on $t.g$,
        with current maximum clique size = $|S_{max}| - |t.S|$
13:     if $|S'_{max}| > |S_{max}| - |t.S|)$ then $S_{max} \leftarrow t.S \cup S'_{max}$
14: return false
```

Figure 5. Application Code for Finding Maximum Clique

Next consider compute$(t, frontier)$. If $|t.S| = 1$ (Line 1), then t is a newly spawned top-level task with $S = \{v\}$ and *frontier* containing all pulled vertices $\in \Gamma_>(v)$. Line 2 thus constructs $t.g$ using vertices (and their adjacency lists) in *frontier*. When constructing $t.g$, we filter any adjacency list item w if $w \notin \Gamma_>(v)$ since w is 2 hops away from v. In contrast, if the condition in Line 1 does not hold, then the current task t was generated by decomposing a bigger upper-level task, which should have already constructed $t.g$ and we skip Line 2.

If $t.g$ is too big, e.g., has more than τ vertices (Line 3), we continue to create next-level tasks that are with smaller subgraphs (Lines 4–9). Here, τ is a user-specified threshold (set as 400,000 by default which works well). Let the current task be $t = \langle S,\ \Gamma_>(S)\rangle$, then Lines 5–7 create a new task $t' = \langle S \cup u, \Gamma_>(S) \cap \Gamma_>(u)\rangle$ for each $u \in \Gamma_>(S)$. If $t'.S$ extended with all vertices in $t'.g$, namely $ext(t'.S)$, still cannot form a clique larger than S_{max}, then t' is pruned and thus freed from memory (Line 9); otherwise, t' is added to Q_{task} (Line 8) so that the system will schedule it for processing.

If $t.g$ is small enough (Line 10), it is mined and S_{max} is updated if necessary (Lines 11-13). Specifically, we prune t if $t.S$ extended with all vertices in $t.g$, namely $ext(t.S)$, still cannot form a clique larger than S_{max} (Line 11). Otherwise, we run the serial mining algorithm of [31] on $t.g$ assuming that a clique of size $\Delta = |S_{max}| - |t.S|$ is already found (for pruning). This is because vertices of $t.S$ are already assumed to be in a clique to find, and to generate a clique larger than S_{max}, $t.S$ should be extended with a clique of $t.g$ with more than Δ vertices. Line 13 updates S_{max} if a larger clique is formed



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	···	···	···	···	···

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	···	···	···	···	···

Figure 6. Structure of Vertex Cache T_{cache} with $B_i = id(v) \mod 10$ & Contents Before and After Atomic Update Operations

with $t.S$ plus the largest clique of $t.g$, denoted by S'_{max}.

Here, compute$(t, frontier)$ always returns *false* (Lines 11 and 14) to indicate that t is finished and can be freed from memory. In other applications like mining quasi-cliques, t may need to pull neighbors of vertices in *frontier* to further grow $t.g$, in which case compute$(t, frontier)$ should return *true*.

Also note that set-enumeration tree is not the only way to avoid redundancy. More algorithms can be found in our preprint [34] where a graph matching algorithm partitions the search space using different instances of the same vertex label.

V. System Design & Implementation

As Sec. III indicates, G-thinker has 2 key modules that enable CPU-bound execution: (1) a vertex cache for accessing by tasks with a high concurrency and (2) a lightweight task generation and scheduling module which delivers a high task throughput while keeping memory consumption bounded. We shall discuss them in Sec. V-A and V-B, respectively. Now, we first overview the components and threads in G-thinker.

Refer to Fig. 3 again. Each worker machine maintains a *local vertex table* (denoted by T_{local} hereafter), and a *cache for remote vertices* (denoted by T_{cache} hereafter). When a task t requests $v \notin T_{local}$, the request is sent to the worker holding $\langle v, \Gamma(v)\rangle$ in T_{local}; the received response $\langle v, \Gamma(v)\rangle$ is then inserted into T_{cache}. Refer to *Task*::pull(v) from Fig. 4 again. If $v \in T_{local}$, t.pull(v) obtains v directly; otherwise, t has to wait for v's response to arrive and we say that t is ***pending***. When all vertices that t waits for are received into T_{cache}, we say that t is ***ready***. In *Comper*::compute$(t, frontier)$, each element in $frontier$ is actually a pointer to either a local vertex in T_{local}, or a remote vertex cached in T_{cache}.

Each machine (or, worker) runs 4 kinds of threads: (1) **compers** which compute tasks by calling *Comper*'s 2 UDFs; (2) communication threads which handle vertex pulling; (3) garbage collecting thread (abbr. **GC**) which keeps T_{cache}'s capacity bounded by periodically evicting unused vertices; (4) the main thread which loads input data, spawns all other threads, and periodically synchronizes job status to monitor progress, and to decide task stealing plans among workers.

G-thinker's communication module sends requests and responses in batches to guarantee throughput while keeping latency low. Compers append pull-requests to the sending module, and the receiving module receives responses, inserts the received vertices to T_{cache}, and notifies pending tasks.

A. Data Cache for Remote Vertices

In a machine, multiple compers may concurrently access T_{cache} for vertices, while the received responses also need to be concurrently inserted into T_{cache}; also, GC needs to concurrently evict unused vertices to keep T_{cache} bounded.

To support high concurrency, we organize T_{cache} as an array of k buckets, each protected by a mutex. A vertex object v is maintained in a bucket B_i where i is computed by hashing v's ID. Operations on two vertices v_1 and v_2 can thus happen together as long as v_1 and v_2 are hashed to different buckets.

Fig. 6 illustrates the design of T_{cache} with $k = 10$ buckets and $hash(v) = v$ mod 10, where each row corresponds to a bucket. In reality, we set $k = 10{,}000$ which exhibits low bucket contention. As Fig. 6 shows, each bucket (row) consists of 3 hash tables (column cells): a Γ-table, a Z-table and an R-table:

- **Γ-table** keeps each cached vertex $\langle v, \Gamma(v)\rangle$ for use by compers. Each vertex entry also maintains a counter *lock-count*(v) tracking how many tasks are currently using v, which is necessary to decide whether v can be evicted.
- **Z-table** (or, zero-table) keeps track of those vertices in Γ-table with *lock-count*$(v) = 0$, which can be safely evicted. Maintaining Z-table is critical to the efficiency of GC, since GC can quickly scan each Z-table (rather than the much larger Γ-table) to evict vertices, minimizing the time of locking a bucket to allow access by other threads.
- **R-table** (or, request-table) tracks those vertices already requested but whose responses with $\Gamma(v)$ have not been received yet, and it is used to avoid sending any duplicate request. Each vertex v in the R-table also maintains *lock-count*(v) to track how many tasks are waiting for v, which will be transferred into Γ-table when $\Gamma(v)$ is received.

Atomic Operations on $\mathbf{T_{cache}}$. We now explain how each bucket of T_{cache} is updated atomically by various threads (e.g., compers, response receiving thread, and GC) with illustrations using Fig. 6. There are 4 kinds of operations OP1–OP4:

(OP1) First, a comper may request for $\Gamma(v)$ to process a task. In this case, v's hashed bucket is locked for update. **Case 1:** if v is found in Γ-table, *lock-count*(v) is incremented and $\Gamma(v)$ is directly returned (see Vertices 10 and 11 in Fig. 6). Moreover, if *lock-count*(v) was 0, it should be erased from Z-table (e.g., Vertex 10). Otherwise, **Case 2.1:** if v is not found in R-table (see Vertex 21 in Fig. 6), then it is requested for the first time, and thus v is inserted into R-table with $\textit{lock-count}(v) = 1$, and v's request is appended to the sending module for batched transmission. Otherwise, **Case 2.2:** v is already requested, and thus $\textit{lock-count}(v)$ is incremented to indicate that one more task is waiting for $\Gamma(v)$ (see Vertex 20).

(OP2) When a response receiving thread receives response $\langle v, \Gamma(v)\rangle$, it will lock v's hashed bucket to move v from R-table to Γ-table (see Vertex 22 in Fig. 6). Note that $\textit{lock-count}(v)$ is directly transferred from v's old entry in R-table to its new entry in Γ-table, which also obtains $\Gamma(v)$ from the response.

(OP3) When a task t finishes an iteration, for every requested remote vertex v, t will release its holding of $\Gamma(v)$ in the Γ-table of v's hashed bucket. This essentially locks the bucket to decrement $\textit{lock-count}(v)$ in the Γ-table (see Vertices 12 and 13 in Fig. 6), and if $\textit{lock-count}(v)$ becomes 0, v is inserted into the bucket's Z-table to allow its eviction by GC (see Vertex 12).

(OP4) When GC locks a bucket to evict vertices in Z-table, GC removes their entries in both Γ-table and Z-table. Imagine, e.g., evicting Vertex 12 from T_{cache} shown on the right of Fig. 6.

As a bucket is protected by a mutex, only one of the above 4 operations OP1 – OP4 may proceed at each time.

Lifecycle of a Remote Vertex. Let us denote the total number of vertices in both Γ-tables and R-tables by $\mathbf{s_{cache}}$, GC aims to keep s_{cache} bounded by a capacity $\mathbf{c_{cache}}$. By default, we set $c_{cache} = 2M$ which takes a small fraction of machine memory.

We include entries in R-tables into s_{cache} because if v is in an R-table, $\Gamma(v)$ will finally be received and added to Γ-table. In contrast, we ignore entries in Z-tables when counting s_{cache} since they are just subsets of entries in Γ-tables. The number of buffered messages is also bounded by s_{cache} since a request for v (and its response) implies an entry of v in an R-table.

We now illustrate how OP1–OP4 update s_{cache} by considering the lifecycle of a remote vertex v initially not cached in T_{cache}. When a task t requests v, the comper that runs t will (1) insert v's entry into R-table (we refer to the R-table of the bucket where v is hashed, but omit the mentioning of bucket hereafter for simplicity), hence $s_{cache} = |\Gamma\text{-tables}| + |\text{R-tables}|$ is incremented by 1, and (2) trigger the sending of v's request.

When response $\langle v, \Gamma(v)\rangle$ is received, the receiving thread moves v's entry from R-table to Γ-table (hence s_{cache} remains unchanged). Finally, when v is released by all related tasks, GC may remove v from T_{cache} (hence s_{cache} is decremented by 1), including v's entries in both Γ-table and Z-table. If a subsequent task requests v again, the above process repeats.

Keeping s_{cache} Bounded. Since s_{cache} is updated by compers and GC, we alleviate contention by maintaining s_{cache} approximately: each comper (resp. GC) thread maintains a local counter that gets committed to s_{cache} when it reaches a user-specified count threshold δ (resp. $-\delta$), by adding it to s_{cache} and resetting the counter as 0. We set $\delta = 10$ by default, which exhibits low contention on s_{cache}, and a small estimation error compared with the magnitude of s_{cache}: bounded by $n_{comper} \times \delta$ where n_{comper} is the number of compers in a machine.

We try to keep the size of T_{cache}, i.e., s_{cache}, bounded by capacity c_{cache}, and if T_{cache} overflows, compers stop fetching new tasks for processing, while old tasks still get processed after their requested vertices are received in T_{cache}, so that these vertices can then be released to allow GC to evict them to reduce s_{cache}. To minimize GC overhead, we adopt a "lazy" strategy which evicts vertices only when T_{cache} overflows. To remove $\delta_{cache} = (s_{cache} - c_{cache})$ vertices, the buckets of T_{cache} are checked by GC in a round-robin order: GC locks one bucket B_i at a time to evict vertices tracked by B_i's Z-table one by one. This process goes on till δ_{cache} vertices are evicted.

In our lazy strategy, compers stop fetching new tasks only if $s_{cache} > (1+\alpha) \cdot c_{cache}$, where $\alpha > 0$ is a user-defined overflow tolerance parameter. GC periodically wakes up to check this condition. If $s_{cache} \leq (1+\alpha) \cdot c_{cache}$, GC sleeps immediately to release its CPU core. Otherwise, GC attempts to evict up to $\delta_{cache} = (s_{cache} - c_{cache}) > \alpha \cdot c_{cache}$ vertices, which is good since batched vertex removal amortizes bucket locking overheads. GC may fail to remove δ_{cache} vertices since some tasks are still holding their requested vertices for processing, but enough vertices will be released finally for GC to remove. This is because these tasks will complete their current iteration and release their requested vertices. We set $\alpha = 0.2$ by default which works well (see Sec. VI Table V (b) for the details).

B. Task Management

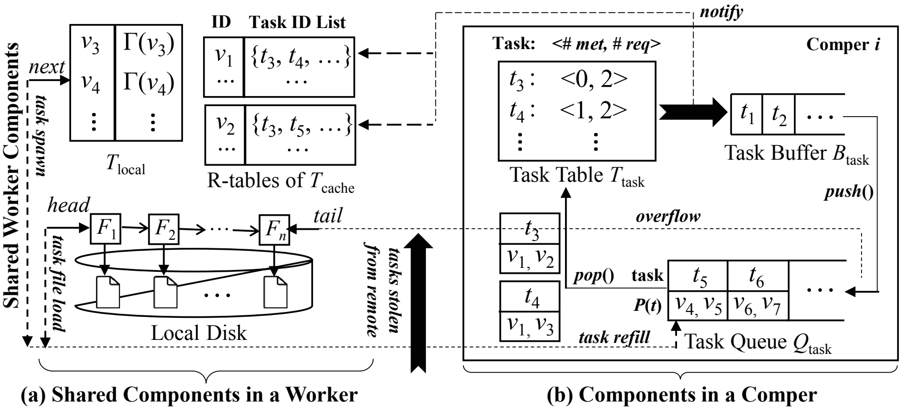
G-thinker aims to minimize task scheduling overheads. Tasks are generated and/or fetched for processing only if memory permits; otherwise, G-thinker focuses on finishing the current active tasks to release resources for taking more tasks.

Task Containers. At any time, a pool of tasks are kept in memory, which allows CPU cores to process ready tasks when pending tasks are waiting for requested vertices. Specifically, each comper maintains in-memory tasks in 3 task containers: a task queue Q_{task} that we already see, a task buffer B_{task} and a task table T_{task}. We now introduce why they are needed.

Fig. 7 (right side) summarizes the components in a comper, and its interaction with other worker components shared by all compers in a machine (left side). The upper-left corner shows the local vertex table T_{local} that holds locally loaded vertices, which are used to spawn new tasks. The next vertex in T_{local} to spawn new tasks is tracked by the "next" pointer.

(1) Task Queue $\mathbf{Q_{task}}$. We have introduced it before: for example, a task t is added to Q_{task} when add_task(t) (c.f., Fig. 4) is called. Since compers do not share a global task queue, contention due to task fetching is minimized, but it is important to keep Q_{task} not too empty so that its comper is kept busy computing tasks. To achieve this goal, we define a task-batch to contain C tasks, and try to keep Q_{task} to contain at least C tasks (i.e., one batch). By default, $C = 150$ which is observed to deliver high task throughput. Whenever a comper finds that $|Q_{task}| \leq C$, it tries to refill Q_{task} with a batch of tasks so that $|Q_{task}|$ gets back to $2C$.

Also, the capacity of Q_{task} should be bounded to keep memory consumption bounded, and we set it to contain at most $3C$ tasks. When Q_{task} is full and another task t needs to be appended to Q_{task}, the last C tasks in Q_{task} are spilled as a batch into a file on disk for later processing, so that t can then be appended rendering $|Q_{task}| = 2C + 1$. This design allows tasks to be spilled to disk(s) in batches to ensure sequential IO. Each machine tracks the task files with a concurrent linked list [21] $\mathscr{L}_{file}$ of file metadata for later loading, as shown in Fig. 7 with the list head and tail marked.



(a) Shared Components in a Worker

(b) Components in a Comper

Figure 7. Components in a Comper & Shared Components in a Worker

Recall that tasks are refilled when $|Q_{task}| \leq C$. To refill tasks, (i) $\mathscr{L}_{file}$ is examined first and a file is digested (if it exists) to load its C tasks; (ii) if $\mathscr{L}_{file}$ is empty, a comper then tries to generate new tasks from those vertices in T_{local} that have not spawned tasks (using UDF *Comper*::task_spawn(v)), by locking and forwarding the "next" pointer of T_{local}.

This strategy prioritizes partially-processed tasks over new tasks, which keeps the number of disk-buffered tasks minimal, and encourages data reuse in T_{cache}. While task spilling only seldom occurs due to our prioritizing rule, it still needs to be properly handled since (1) many pending tasks may become ready together to be added to Q_{task}, and (2) a task with a big subgraph may generate many new tasks and add them to Q_{task}.

While refilling sources like $\mathscr{L}_{file}$ and T_{local} are shared by all compers in a machine, lock contention cost is amortized since a batch of tasks are fetched each time a resource is locked.

(2) Task Buffer $\mathbf{B_{task}}$. Since Q_{task} needs to be refilled from the head of the queue and to spill tasks from the tail, both by its comper, Q_{task} is designed as a deque only updated by one thread, i.e., its comper. Thus, when a response receiving thread finds that a pending task t becomes ready, it has to append t to another concurrent queue B_{task} (see Fig. 7) to be later fetched by the comper into Q_{task} for processing.

(3) Task Table $\mathbf{T_{task}}$. Recall that pending tasks are suspended and properly notified when responses arrive. A comper keeps its pending tasks in a hash table T_{task} (see Fig. 7). Since each machine runs multiple compers, when response $\Gamma(v)$ is received, the receiving thread needs to update the status of those tasks from all compers that are waiting for v, i.e., for each pending task t, it needs to track which comper holds t.

For this purpose, each comper maintains a sequence number n_{seq}. Whenever it inserts a task t into T_{task}, it associates t with a 64-bit task ID $id(t)$ which concatenates a 16-bit comper ID with the 48-bit n_{seq}, and n_{seq} is then incremented for use by the next task to insert. Given $id(t)$, the receiving thread can easily obtain the comper that holds t, to get its T_{task} for update.

In Fig. 6, we simplified vertex v's entry in an R-table as maintaining only a counter *lock-count*(v), but it actually maintains the ID list of those tasks that requested v (see Fig. 7).

Assume that a task t requests a set of vertices $P(t)$ in an iteration. In T_{task} in Fig. 7, an entry for a task t maintains key $id(t)$ and value $\langle met(t), req(t)\rangle$, where $\mathbf{req(t)}$ denotes the number of requested vertices $|P(t)|$, and $\mathbf{met(t)}$ denotes how many of them are already available. Both $met(t)$ and $req(t)$ are properly set when a comper inserts t into its T_{task}.

When the receiving thread receives $\Gamma(v)$, v's entry is moved from R-table in T_{cache} to Γ-table as operation OP2 in Sec. V-A is described. The receiving thread also retrieves v's pending task list from its R-table entry, and for each $id(t)$ in the list, it updates t's entry in the task table T_{task} of the comper that holds t, by incrementing $met(t)$; if $met(t) = req(t)$, t becomes ready and the receiving thread moves t from T_{task} to B_{task}.

Algorithm of a Comper. Every comper repeats the two operations *pop*() and *push*() (see Fig. 7) each round, and executes until the job end-signal is set by the main thread.

- If B_{task} is not empty, **push()** gets a task t from B_{task} and computes for one iteration. Note that since t is in B_{task}, its requested remote vertices have all been cached in T_{cache}. If t is not finished (i.e., UDF compute(t, *frontier*) returns *true*), t is appended to Q_{task} along with the IDs of newly requested vertices $P(t)$ (see Fig. 7). In UDF compute(.), when a task t pulls v, v is simply added to $P(t)$. The actual examination of v on T_{cache} is done by *pop*() below.
- If Q_{task} is not empty, **pop()** fetches a task t along with $P(t)$ from the head of Q_{task} for processing. Every non-local vertex $v \in P(t)$ is requested from T_{cache}: (i) if at least one remote $v \in P(t)$ cannot be found from T_{cache} (i.e., in Γ-table of v's hashed bucket, recall OP1 in Sec. V-A), t is added to T_{task} as pending; (ii) otherwise, t computes for more iterations until when $P(t)$ has remote vertices to wait for (hence t is added to T_{task}), or when t is finished.

Task refilling is handled by *pop*(). Before *pop*() pops a task from Q_{task} as described above, it first checks if $|Q_{task}| \leq C$. If so, it tries to first fill Q_{task} with tasks from one of the following sources in prioritized order: (1) a task file from $\mathscr{L}_{file}$, (2) B_{task}, (3) spawning new tasks from vertices in T_{local}. This prioritizes tasks that were processed the earliest, which minimizes the number of spilled tasks on disks.

A task t always releases all its previously requested non-local vertices from T_{cache} after each iteration of computation, so that they can be evicted by GC in time.

Note that *pop*() generates new requests (hence adds vertices to T_{cache}), while *push*() consumes responses and releases vertices on hold in T_{cache} (so that GC may evict them). A comper keeps running *push*() in every round so that if there are tasks in B_{task}, they can be timely processed to release space in T_{cache}. In contrast, comper runs *pop*() in a round only if (1) the capacity of T_{cache} permits (i.e., $s_{cache} \leq (1+\alpha)c_{cache}$), and (2) the number of tasks in T_{task} and B_{task} together does not exceed a user-defined threshold D ($= 8C$ by default which is found to provide good task throughput). Otherwise, new task processing will be blocked till *push*() unlocks sufficient vertices for GC to evict. Note that it is important to run *push*() in every round so that tasks can keep flowing even after *pop*() blocks. If both *pop*() and *push*() fail to process a task after a round, the comper is considered as **idle**: there is no task in Q_{task} and B_{task}, and there is no more new task to spawn (but T_{task} may contain pending tasks). In this case, the comper sleeps to release CPU core but may be awakened by the main thread which synchronizes status periodically if there are more tasks (e.g., stolen from other workers). A job terminates if the main thread of all machines find all their compers idle.

One problem remains: consider when a comper pulls $P(t) = \{v_1, v_2, \ldots, v_\ell\}$ in *pop*() for a popped task t, if $v_1 \notin T_{local} \cup T_{cache}$, t will be added to the comper's T_{task} and its request for v_1 is sent. Now assume that $v_2, \ldots, v_\ell \in T_{local}$. If the receiving thread receives $\Gamma(v_1)$ and updates t's entry in T_{task} before the comper requests v_ℓ and increments $met(t)$, then the receiving thread fails to move t from T_{task} to B_{task} and t will stay in T_{task} forever. To avoid this problem, in *pop*(), a comper will check if $met(t) = req(t)$ after requesting all vertices in $P(t)$ against T_{cache}, and if so, it moves t from T_{task} to B_{task} by itself.

Task Stealing. To balance workloads among all machines, we let all the workers synchronize their task processing progresses, and let those machines that are about to become idle to prefetch tasks from heavily-loaded ones for processing. Currently, the main threads of workers synchronize their progresses which are gathered at a master worker, who generates the stealing plans and distributes them back to the main threads of other workers, so that they can collectively execute these plans before synchronizing progress again. The number of remaining tasks at a worker is estimated from $|\mathscr{L}_{file}|$ and the number of unprocessed vertices in T_{local}. Tasks stolen by a worker are added to its $\mathscr{L}_{file}$ to be fetched by its compers.

Fault Tolerance. G-thinker also supports checkpointing for fault tolerance: Worker states (e.g., $\mathscr{L}_{file}$, Q_{task}, T_{task} and B_{task}, and task spawning progress) and outputs can be periodically committed to HDFS as a checkpoint. When a machine fails, the job can rerun from the latest checkpoint, but tasks in T_{task} and B_{task} need to be added back to Q_{task} in order to request vertices into T_{cache} again (as T_{cache} starts "cold").

VI. Experiments

We compared G-thinker with the state-of-the-art graph-parallel systems, including (1) the most popular vertex-centric system, Giraph [8] (to verify that vertex-centric model does not scale for subgraph mining), (2) G-Miner [6] open-sourced at [2], and (3) Arabesque [29]. NScale [23] is not open-sourced. We also tested the single-machine out-of-core subgraph-centric system such as RStream [32].

Table II
Graph 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

Table III
System Comparison

Dataset	Arabesque	Giraph	G-Miner	G-thinker
(a) Triangle Counting (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) Maximum Clique Finding (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) Subgraph Matching (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.

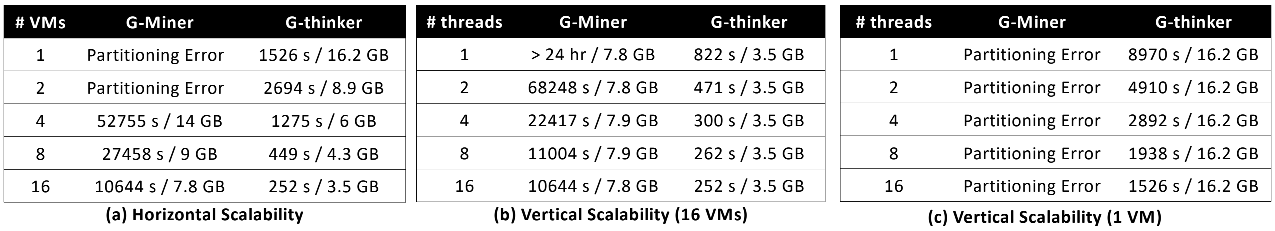
We used the 3 applications also used in the experiments of [34], [6] for performance study: (1) maximum clique finding (**MCF**), (2) triangle counting (**TC**), and (3) subgraph matching (**GM**). We compared with Giraph on MCF and TC as their vertex-centric algorithms exist [24], [5]. Arabesque also only provided MCF and TC implementations. All relevant code can be found at http://www.cs.uab.edu/yanda/gthinker.

Table II shows real graph datasets used: *Youtube* [39], *Skitter* [26], *Orkut* [22], *BTC* [4] and *Friendster* [11]. They have different characteristics, such as size and degree distribution.

All our experiments were conducted on a cluster of 16 virtual machines (VMs) on Microsoft Azure. Each VM (model D16S_V3 deployed with CentOS 7.4) has 16 CPU cores, 64 GB RAM, and is mounted with a 512 GB managed disk. Each experiment was repeated 10 times, and all reported results were averaged over the 10 runs. We observed in all our experiments that the disk space consumed by G-thinker is negligible (since compers prioritize spilled tasks when refilling their Q_{task}), and thus we omit disk space report.

Comparison among Distributed Systems. Table III reports the (1) running time and (2) peak VM memory consumption (taking the maximum over all machines) of our 3 applications over the 5 datasets shown in Table II. We can see that Arabesque and Giraph incurred huge memory consumption

Table IV
MCF OVER FRIENDSTER: SCALABILITY RESULTS



# VMs	G-Miner	G-thinker
1	Partitioning Error	1526 s / 16.2 GB
2	Partitioning Error	2694 s / 8.9 GB
4	52755 s / 14 GB	1275 s / 6 GB
8	27458 s / 9 GB	449 s / 4.3 GB
16	10644 s / 7.8 GB	252 s / 3.5 GB

(a) Horizontal Scalability

# threads	G-Miner	G-thinker
1	> 24 hr / 7.8 GB	822 s / 3.5 GB
2	68248 s / 7.8 GB	471 s / 3.5 GB
4	22417 s / 7.9 GB	300 s / 3.5 GB
8	11004 s / 7.9 GB	262 s / 3.5 GB
16	10644 s / 7.8 GB	252 s / 3.5 GB

(b) Vertical Scalability (16 VMs)

# threads	G-Miner	G-thinker
1	Partitioning Error	8970 s / 16.2 GB
2	Partitioning Error	4910 s / 16.2 GB
4	Partitioning Error	2892 s / 16.2 GB
8	Partitioning Error	1938 s / 16.2 GB
16	Partitioning Error	1526 s / 16.2 GB

(c) Vertical Scalability (1 VM)

and could not scale to large datasets like *BTC* and *Friendster*, since they keep materialized subgraphs in memory.

G-Miner is memory-efficient as it keeps tasks (containing subgraphs) in a disk-resident task priority queue; it is also more efficient than Arabesque and Giraph. However, while G-Miner scales to large datasets, its performance is very slow on them. This is caused by its IO-bound disk-resident task queue, where task insertions are costly when the data size and hence task number become large. G-Miner also failed to finish any application on *BTC* within 24 hours, which is likely because of the uneven vertex degree distribution of *BTC* where the dense part of *BTC* incurs enormous computation workloads, and G-Miner is not able to handle such scenarios efficiently.

As Table III shows, G-thinker consistently uses less memory than G-Miner and is faster from a few times to 2 orders of magnitude (e.g., see TC and MCF on *Friendster*). One exception is MCF over *Skitter*, where we find that this is because of the different task processing order of G-Miner and G-thinker. Specifically, G-thinker processes tasks approximately in the order of how the vertices in T_{local} are ordered (as tasks are spawned from vertices in T_{local} on demand when memory permits), while G-Miner prioritizes tasks using locality sensitive hashing over $P(t)$, which is the set of vertices to pull by a task t. MCF uses the latest maximum clique found to prune search space, and G-Miner happens to process the maximum clique much earlier. However, this is irrelevant to system design and really depends on how vertices are ordered in the input file (and hence in T_{local} after graph loading).

Comparison with Single-Machine Systems. We also ran RStream whose code for TC and clique listing are provided [3]. However, their clique code does not output correct results. For triangle counting, RStream takes 53s on *Youtube*, 283s on *Skitter*, and 3713s on *Orkut*; in contrast, our G-thinker running with a single machine takes only 4s on *Youtube*, 30s on *Skitter*, and 210s on *Orkut*. This is no wonder as RStream runs out-of-core and is IO-bound. For the 2 big graphs *BTC* and *Friendster*, RStream used up all our disk space, while G-thinker takes 1223s and 4282s, respectively, on one machine.

Nuri is also not competitive with G-thinker, since Nuri is implemented as a single-threaded Java program while G-thinker can use all CPU cores for mining. As Figure 11 of [13] shows, Nuri takes over 1000s to find the maximum clique of *Youtube*, while our G-thinker demo video on http://www.cs.uab.edu/yanda/gthinker shows that running 8 threads on one machine takes only 9.449s to find the maximum clique.

Scalability. Since both G-thinker and G-Miner can scale to *Friendster*, we compare their scalability using the application MCF. G-Miner additionally requires vertices to be pre-partitioned before running the subgraph-centric computation. Unfortunately, we are not able to partition *Friendster* when there are only 2 machines or less, as G-Miner's partitioning program reports an error caused by sending more data than MPI_Send allows (the data size becomes negative as it exceeds the range of "int"), and we denote these results as "Partitioning Error" in Table IV on system scalability results.

Table IV(a) reports the horizontal scalability when we vary the number of VMs as 1, 2, 4, 8, 16. We can see that additional machines generally improves the performance, and G-thinker is round 50 times faster than G-Miner. The only exception is when the VM number goes from 1 to 2: running G-thinker on 1 machine is faster which is because in a single machine, tasks do not need to request remote vertices and hence wait.

Table IV(b) reports the vertical scalability when we use 16 VMs but vary the number threads (compers for G-thinker) on each VM as 1, 2, 4, 8, 16. We can see that additional threads improves the performance of both systems but G-thinker is significantly faster. However, the improvement from 8 VMs to 16 is not significant because tasks spawned from many low-degree vertices do not generate large enough subgraphs to hide IO cost in the computation, but this can be solved by bundling tasks of low-degree vertices into big tasks as done in [38] (a potential future work); also, our machines were connected by GigE and the problem may disappear if 10 GigE is used.

Execution with a single machine is also interesting to explore when studying vertical scalability, since there are no remote data to request. Table IV(c) shows the results when we vary the number of threads (compers for G-thinker) as 1, 2, 4, 8, 16. We observe almost linear speedup for G-thinker (i.e., computation is perfectly divided among CPU cores), which is expected since a task never needs to wait for remote vertices.

System Parameters. G-thinker uses our well-tested stable system parameters. For example, the capacity of T_{cache} (i.e., c_{cache}) is 2M; while GC evicts vertices only when the size of T_{cache} overflows, i.e., $s_{cache} > (1+\alpha)\cdot c_{cache}$ with overflow tolerance parameter $\alpha = 0.2$. We have conducted extensive experiments to find the default parameters and to verify that the performance is insensitive to changes nearby them in various applications and datasets. Due to space limit, here we illustrate parameter stability with 2 representative experiments on parameters c_{cache} and α, respectively. For each parameter that we change, all other parameters are fixed as default values.

Table V
MCF over Friendster: System Parameters (M = 1,000,000)

C_{cache}	Time	Memory
20M	210 s	8.4 GB
2M	252 s	3.5 GB
0.2M	451 s	2 GB
0.02M	829 s	1.5 GB

(a) Effect of c_{cache}

α	Time	Memory
0.002	262 s	3 GB
0.02	257 s	3.2 GB
0.2	252 s	3.5 GB
2	221 s	3.7 GB

(b) Effect of α

Table V(a) shows the performance of G-thinker when we change vertex cache capacity c_{cache}. We can see that while as small value of c_{cache} such as 0.2M and 0.02M makes the performance much slower, the improvement from 2M to 20M is not significant (still between 200s and 300s); in contrast, to get this small improvement, the memory cost is more than doubled (from 3.5 GB to 8.4 GB), which is not worthwhile.

Table V(b) shows the performance of G-thinker when we change overflow-tolerance parameter α. Recall that GC keeps evicting unused vertices when vertex cache overflows, and a larger α means that GC is "lazier" and acts only when a large capacity overflow occurs (hence more memory usage). We can see that larger α only slightly improves the performance. In fact, when $\alpha = 2$, T_{cache} may contain $3 \cdot c_{cache}$ vertices as compared with the default α where T_{cache} may contain $1.2 \cdot c_{cache}$ vertices, but the speed up is not obvious (despite almost $3\times$ more memory used). This justifies that $\alpha = 0.2$ is a good tradeoff between memory usage and task throughput.

Other system parameters are similarly chosen with extensive tests, and the results are omitted due to space limit.

VII. Conclusion

We presented a distributed system called G-thinker for large-scale subgraph mining, featuring its CPU-bound design in contrast to existing IO-bound Big Data systems.

To the best of our knowledge, G-thinker is the first truly CPU-bound graph-parallel system for subgraph mining, and it provides a user-friendly subgraph-centric programming interface based on task-based vertex pulling where users can easily write distributed subgraph mining programs. This is the first of a series of CPU-bound systems we plan to develop following our task-based T-thinker paradigm [36]. Another one is [37].

Acknowledgements. This work is partially supported by NSF OAC-1755464 (CRII), South Big Data Hub Azure Research Award, NSF IIS-1618669 (III) and ACI-1642133 (CICI), NSERC of Canada, and Hong Kong GRF 14201819.

References

[1] COST in the Land of Databases. https://github.com/frankmcsherry/blog/blob/master/posts/2017-09-23.md.

[2] G-Miner Code. https://github.com/yaobaiwei/GMiner.

[3] RStream Code. https://github.com/rstream-system.

[4] BTC. http://km.aifb.kit.edu/projects/btc-2009.

[5] Y. Bu, V. R. Borkar, J. Jia, M. J. Carey, and T. Condie. Pregelix: Big(ger) graph analytics on a dataflow engine. *PVLDB*, 8(2):161–172, 2014.

[6] H. Chen, M. Liu, Y. Zhao, X. Yan, D. Yan, and J. Cheng. G-miner: an efficient task-oriented graph mining system. In *EuroSys*, pages 32:1–32:12, 2018.

[7] J. Cheng, Q. Liu, Z. Li, W. Fan, J. C. S. Lui, and C. He. VENUS: vertex-centric streamlined graph computation on a single PC. In *ICDE*, pages 1131–1142, 2015.

[8] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan. One trillion edges: Graph processing at facebook-scale. *PVLDB*, 8(12):1804–1815, 2015.

[9] S. Chu and J. Cheng. Triangle listing in massive networks. *TKDD*, 6(4):17:1–17:32, 2012.

[10] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.

[11] Friendster. http://snap.stanford.edu/data/com-friendster.html.

[12] X. Hu, Y. Tao, and C. Chung. I/o-efficient algorithms on triangle listing and counting. *ACM Trans. Database Syst.*, 39(4):27:1–27:30, 2014.

[13] A. Joshi, Y. Zhang, P. Bogdanov, and J. Hwang. An efficient system for subgraph discovery. In *IEEE Big Data*, pages 703–712, 2018.

[14] A. Kyrola, G. E. Blelloch, and C. Guestrin. Graphchi: Large-scale graph computation on just a PC. In *OSDI*, pages 31–46, 2012.

[15] J. Lee, W. Han, R. Kasperovics, and J. Lee. An in-depth comparison of subgraph isomorphism algorithms in graph databases. *PVLDB*, 6(2):133–144, 2012.

[16] W. Lin, X. Xiao, and G. Ghinita. Large-scale frequent subgraph mining in mapreduce. In *ICDE*, pages 844–855, 2014.

[17] G. Liu and L. Wong. Effective pruning techniques for mining quasi-cliques. In *PKDD*, pages 33–49, 2008.

[18] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD Conference*, pages 135–146, 2010.

[19] R. R. McCune, T. Weninger, and G. Madey. Thinking like a vertex: A survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Comput. Surv.*, 48(2):25:1–25:39, 2015.

[20] F. McSherry, M. Isard, and D. G. Murray. Scalability! but at what cost? In *HotOS*, 2015.

[21] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *PODC*, pages 267–275, 1996.

[22] Orkut. http://konect.uni-koblenz.de/networks/orkut-links.

[23] A. Quamar, A. Deshpande, and J. Lin. Nscale: neighborhood-centric large-scale graph analytics in the cloud. *The VLDB Journal*, pages 1–26, 2014.

[24] L. Quick, P. Wilkinson, and D. Hardcastle. Using pregel-like large scale graph processing frameworks for social network analysis. In *ASONAM*, pages 457–463, 2012.

[25] A. Roy, I. Mihailovic, and W. Zwaenepoel. X-stream: edge-centric graph processing using streaming partitions. In *SOSP*, pages 472–488, 2013.

[26] Skitter. http://konect.uni-koblenz.de/networks/as-skitter.

[27] Z. Sun, H. Wang, H. Wang, B. Shao, and J. Li. Efficient subgraph matching on billion node graphs. *PVLDB*, 5(9):788–799, 2012.

[28] N. Talukder and M. J. Zaki. A distributed approach for graph mining in massive networks. *Data Min. Knowl. Discov.*, 30(5):1024–1052, 2016.

[29] C. H. C. Teixeira, A. J. Fonseca, M. Serafini, G. Siganos, M. J. Zaki, and A. Aboulnaga. Arabesque: a system for distributed graph mining. In *SOSP*, pages 425–440, 2015.

[30] Y. Tian, A. Balmin, S. A. Corsten, S. Tatikonda, and J. McPherson. From "think like a vertex" to "think like a graph". *PVLDB*, 7(3):193–204, 2013.

[31] E. Tomita and T. Seki. An efficient branch-and-bound algorithm for finding a maximum clique. In *DMTCS*, pages 278–289, 2003.

[32] K. Wang, Z. Zuo, J. Thorpe, T. Q. Nguyen, and G. H. Xu. Rstream: Marrying relational algebra with streaming for efficient graph mining on A single machine. In *OSDI*, pages 763–782, 2018.

[33] D. Yan, Y. Bu, Y. Tian, and A. Deshpande. Big graph analytics platforms. *Foundations and Trends in Databases*, 7(1-2):1–195, 2017.

[34] D. Yan, H. Chen, J. Cheng, M. T. Özsu, Q. Zhang, and J. C. S. Lui. G-thinker: Big graph mining made easier and faster. *CoRR*, abs/1709.03110, 2017.

[35] D. Yan, J. Cheng, Y. Lu, and W. Ng. Blogel: A block-centric framework for distributed computation on real-world graphs. *PVLDB*, 7(14):1981–1992, 2014.

[36] D. Yan, G. Guo, M. M. R. Chowdhury, M. T. Özsu, J. C. S. Lui, and W. Tan. T-thinker: a task-centric distributed framework for compute-intensive divide-and-conquer algorithms. In *PPoPP*, pages 411–412, 2019.

[37] D. Yan, W. Qu, G. Guo, and X. Wang. PrefixFPM: a parallel framework for general-purpose frequent pattern mining. In *ICDE*, 2020.

[38] Y. Yang, D. Yan, S. Zhou, and G. Guo. Parallel clique-like subgraph counting and listing. In *ER*, 2019.

[39] Youtube. https://snap.stanford.edu/data/com-youtube.html.