The VLDB Journal
https://doi.org/10.1007/s00778-021-00688-z

SPECIAL ISSUE PAPER

G-thinker: a general distributed framework for finding qualified subgraphs in a big graph with load balancing

Da Yan[1] · Guimu Guo[1] · Jalal Khalil[1] · M. Tamer Özsu[2] · Wei-Shinn Ku[3] · John C. S. Lui[4]

Received: 30 August 2020 / Revised: 3 April 2021 / Accepted: 10 July 2021

Abstract
Finding 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 that attempt to scale them are all IO-bound in execution. We propose the first truly CPU-bound distributed framework called *G-thinker* for subgraph finding algorithms, which adopts a task-based computation model, and which also provides a user-friendly subgraph-centric vertex-pulling API for writing distributed subgraph finding algorithms that can be easily adapted from existing serial 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 idle time of CPU cores. To further improve load balancing on graphs where the workloads of individual tasks can be drastically different due to biased graph density distribution, we propose to prioritize the scheduling of those tasks that tend to be long running for processing and decomposition, plus a timeout mechanism for task decomposition to prevent long-running straggler tasks. The idea has been integrated into a novelty algorithm for maximum clique finding (MCF) that adopts a hybrid task decomposition strategy, which significantly improves the running time of MCF on dense and large graphs: The algorithm finds a maximum clique of size 1,109 on a large and dense *WikiLinks* graph dataset in 70 minutes. 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.

Keywords Graph mining · Subgraph-centric · CPU-bound · Compute-intensive · Clique · Triangle · Subgraph matching

1 Introduction

Problems that G-thinker Targets. 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 [37], quasi-clique enumeration [21], triangle listing and counting [16], subgraph matching [19], etc. These problems have a wide range of applications including social network analysis and biological network investigation. They also often have a high time complexity (e.g., finding maximum cliques is NP-hard),

✉ Da Yan
yanda@uab.edu

Guimu Guo
guimuguo@uab.edu

Jalal Khalil
jalalk@uab.edu

M. Tamer Özsu
tamer.ozsu@uwaterloo.ca

Wei-Shinn Ku
weishinn@auburn.edu

John C. S. Lui
cslui@cse.cuhk.edu.hk

[1] Department of Computer Science, The University of Alabama at Birmingham, Birmingham, AL, USA

[2] David R. Cheriton School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada

[3] Department of Computer Science and Software Engineering, Auburn University, Auburn, AL, USA

[4] Department of Computer Science and Engineering, The Chinese University of Hong Kong, Shatin, N.T, Hong Kong

Published online: 04 August 2021

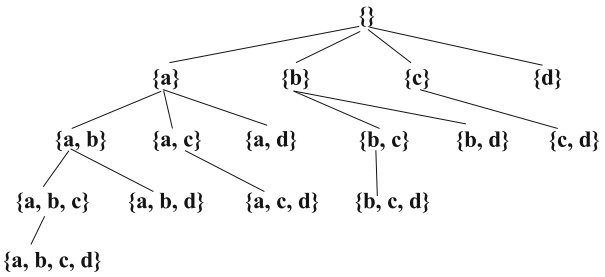


Fig. 1 Set-Enumeration Tree

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. Thus, existing serial algorithms cannot scale to modern web-scale big graphs.

Subgraph finding is usually solved by divide and conquer. Taking dense subgraph mining as an example, a common solution is to organize the giant search space of the power set of V into a set-enumeration tree [21].

Figure 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, [21] 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.

Problems Not Targeted by G-thinker. 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 [33,40].

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 of 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 [35], RStream [38] and Nuri [17]. 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 [34] or for many graph transactions [20,45], and they can be directly used.

Motivations for G-thinker. A natural solution is to utilize many CPU cores to divide the computation workloads of subgraph finding, but there is a challenge intrinsic to the nature of subgraph finding algorithms: *The number of subgraphs induced by the power sets of V is exponential to the graph size itself*, and it is impractical to keep/materialize all of them in memory; however, *out-of-core subgraph processing generates an IO bottleneck* that reduces CPU core utilization rate.

In fact, as shall be clear in Sect. 2, all existing graph-parallel systems have an IO-bound execution engine, making them inefficient for subgraph finding.

We propose G-thinker for CPU-bound parallel subgraph finding while keeping memory consumption low. As an illustration, it takes merely 354 seconds in total and 3.8 GB memory per machine in our 16-node cluster to find the maximum clique (with 129 vertices) on the big Friendster social network of [13] 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 in by the newly added vertices. If the mining of the tree branch under S is expensive, we can further divide it into child branches (rooted at child nodes of S) for parallel mining; otherwise, the entire tree 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 $|g|$, we have (1) the IO cost of materializing g by collect-

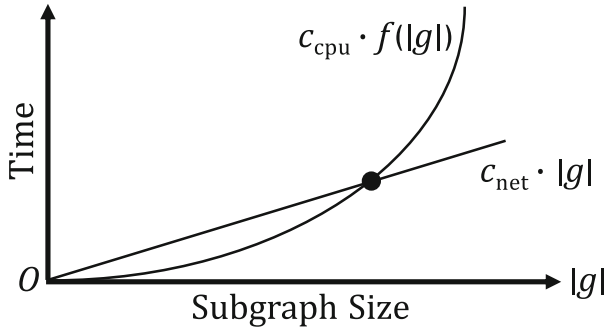


Fig. 2 Computation and communication costs of a task in terms of time. Here, a task that spawns from a vertex v needs to pull those vertices in v's neighborhood that are necessary for computation first by communication to construct a task subgraph g, whose data volume (and hence transmission time) is given by $O(|g|) = c_{net} \cdot |g|$. The computation over g often has a high time complexity $O(f(|g|) = c_{cpu} \cdot f(|g|)$, where $f(|g|)$ is problem dependent, e.g., $|E|^{1.5}$ for triangle counting while being exponential for maximum clique finding

ing 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 computation and communication are well overlapped.

To effectively 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 (needed to construct subgraph g), other tasks (e.g., with g already constructed) can continue their computation to keep CPU cores busy.* This approach also bounds memory cost since only a bounded pool of tasks is in memory, refilled with new tasks only when the number of active tasks is insufficient to keep CPU cores busy.

Contributions. The main contributions of this work are summarized as follows:

- The framework design of G-thinker satisfies all 7 desirabilities established in Sect. 3 necessary for scalability and efficiency of subgraph finding problems.
- 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 parallel algorithms easily adaptable from serial versions.
- G-thinker (http://bit.ly/gthinker) is open-sourced with detailed documentation, and extensive experiments are conducted to compare the scalability and efficiency of G-thinker with existing systems.

As a journal extension of our ICDE conference paper [44], this paper makes new improvements as follows:

- We now compare the systems over a total of 9 graph datasets, including 4 new large datasets. Newly introduced datasets expose some inefficiency of the previous G-thinker implementation in load balancing, esp. when some tasks can be drastically more expensive than others, which occurred on new datasets. To further explore how denser and larger graphs (especially those with a large maximum clique size) impact G-thinker's load balancing mechanism, 9 additional graphs were further explored, 4 of which have maximum vertex degree close to or beyond $|V|/2$ and 5 of which meet our density and size requirements for data selection. Experiments show that G-thinker scales well to even the *WikiLinks* graph dataset with a maximum clique of size 1,109!
- We re-examine our old system design that assumes tasks can be decomposed to the extent that each task is not a long-running straggler. We find that long-running straggler tasks are not prioritized for task decomposition, and we thus implemented a new G-thinker system with a better task prioritization and load balancing strategy that significantly improves the performance when straggler tasks exist.
- For the application of maximum clique finding, the performance was found to be poor on the new *LiveJournal* dataset (c.f. Table 2 in Sect. 11). We identified a problem with the previous task decomposition strategy that does not effectively utilize a **vertex-coloring based pruning** technique and designed a new task decomposition strategy that (1) utilizes that pruning and (2) effectively decides the decomposition timing using a new **task timeout strategy**. We then identify that the vertex coloring process itself can become a performance bottleneck when the task subgraph g is large, and thus, we design another algorithm where the old task decomposition strategy is adopted instead when g is large. This hybrid task decomposition strategy reduces the running time on *LiveJournal* from 1300 seconds to 168 seconds and also performs the best on the other dense and large graphs that we tested.

Paper Organization. The rest of this paper is organized as follows. Section 2 reviews existing graph-parallel systems and explains why they are IO-bound. Section 3 then establishes 7 desirable features for a scalable and efficient subgraph finding system and overviews the system architecture of G-thinker that meets all the 7 features. Section 4 introduces the adopted subgraph-centric programming API, and Sect. 5, 6 and 7 then introduce how to write application

code for maximum clique finding (MCF), triangle counting and subgraph matching, respectively. Section 8 describes the system design focusing on the two pillars to achieve CPU-bound performance, i.e., vertex cache and task management. Section 9 then describes the load balancing issue we found about MCF in our experiments with the basic G-thinker system and proposes an improved system design to allow better load balancing. Based on this design, we present an improved G-thinker algorithm for MCF that addresses the load balancing issue. Finally, Sect. 11 reports the experimental results and Sect. 12 concludes this paper.

2 Related work

This section reviews existing graph-parallel systems and explains why they are IO-bound and not suitable for compute-intensive subgraph finding problems.

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 [11] 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 [22], a vertex needs to first receive a value from each in-neighbor and then simply adds it to the current PageRank value. IO-bound execution can be catastrophic for computation problems beyond those with a low time complexity. For example, even for triangle counting with time complexity $O(|E|^{1.5})$, [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 less than half a minute.

In fact, McSherry et. al [24] have noticed that existing graph-parallel systems are comparable and sometimes slower than a single-threaded program. In another recent post by McSherry [2], he further indicated that the current distributed implementations "scale" (i.e., using aggregate IO bandwidth), but their performance does not get to "a simple single-threaded implementation."

Categorization of Graph-Parallel Systems. Our book [39] classifies 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 [41] and Giraph++ [36] are merely extensions to the vertex-centric systems.

Vertex-Centric Systems. Pioneered by Pregel [22], a number of distributed systems have been proposed for simple iterative graph processing [23]. 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 [7,18,30] by streaming vertices and edges from disk to memory for batched state updates; however, their workloads are still disk IO-bound. The vertex-centric programming API is also not convenient for writing subgraph finding algorithms that operate on subgraphs.

Subgraph-Centric Systems. Recently, a few systems began to explore a think-like-a-subgraph programming model, including distributed systems NScale [27], Arabesque [35] and G-Miner [6] and single-machine systems RStream [38] and Nuri [17]. 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 breadth-first search (BFS) around each vertex, implemented as k rounds of MapReduce computations to avoid keeping the numerous subgraphs in memory; (ii) NScale then mines these subgraphs in parallel by reducers. Since this design requires that all subgraphs be constructed before any of them can begin its computation, it leads to poor CPU utilization and the straggler's problem.

Arabesque [35] 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 we shall show in Sect. 11) as the number of subgraphs grows exponentially.

The task-based vertex-pulling API of **G-thinker** is first proposed by our G-thinker preprint [40], but our execution engine design is now significantly improved to eliminate the bad designs mentioned there. In our task-based vertex-pulling API, tasks are spawned from individual vertices, and

a task can grow its associated subgraph by requesting adjacent vertices and edges for subsequent computation. This API is then followed by **G-Miner** [6], as indicated by the statement below Fig. 1 of [6]: "*The task model is inspired by the task concept in G-thinker*." The original G-thinker prototype in our preprint [40] is to verify that our API can significantly improve the performance of subgraph finding compared with existing systems, but the execution engine there is still 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 G-thinker does) 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 [38] is a single-machine out-of-core system which proposes a so-called GRAS model to emulate 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 [17] 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 Sect. 11 shall show, RStream and Nuri are not anywhere close to when G-thinker runs just on a single machine.

3 G-thinker overview

Figure 3 shows the basic architecture of G-thinker on a cluster of 3 machines. Let's temporarily ignore the top part of Fig. 3 for now, which are load balancing improvements to be presented in Sects. 9 and 10. 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 v's 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 together constitute 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 then computes upon. For example, consider the problem of mining maximal γ-quasi-cliques ($\gamma \geq 0.5$) for which [21] 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 second-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 the *Quick* algorithm [15,21]. To avoid double-counting, a vertex v only requests those vertices whose ID is larger than v (recall Fig. 1), so that a quasi-clique whose smallest vertex is u must be found by the task spawned from u.

Such a subgraph finding algorithm is implemented in G-thinker by specifying two 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 lists of those vertices requested by the current task 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 computation 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 the UDF *compute*(*frontier*). This allows multiple tasks to share requested vertices to minimize redundant vertex requests, and once a vertex in the cache is no longer requested by any task in the

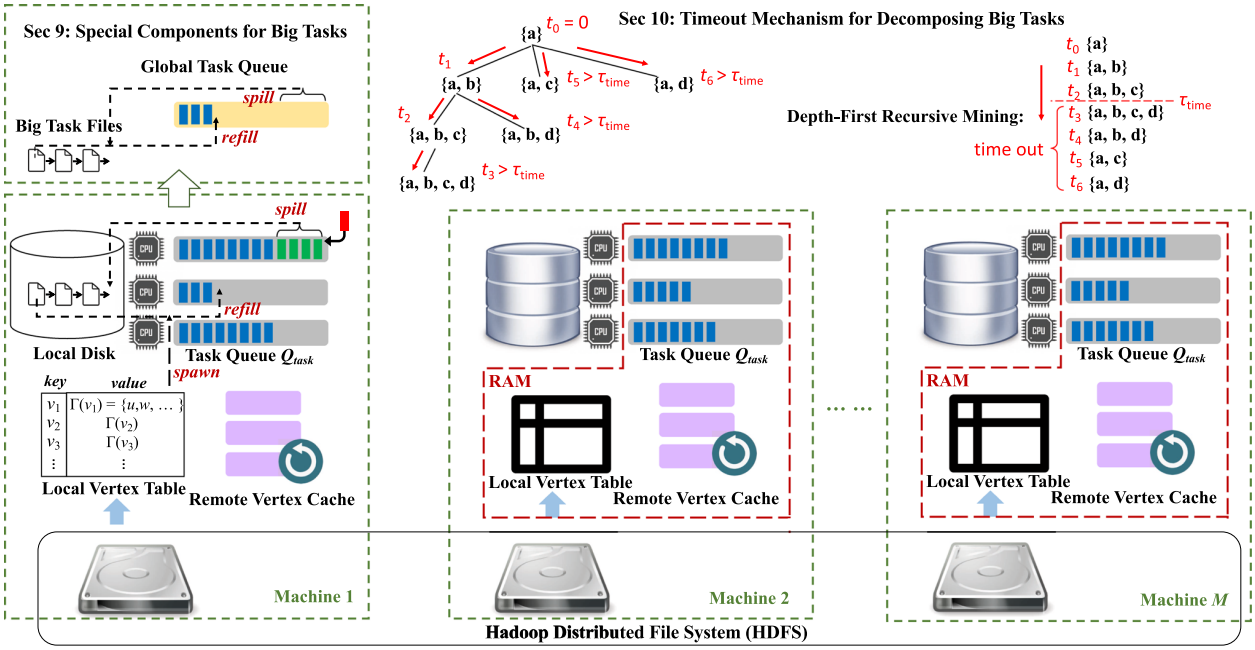


Fig. 3 G-thinker Architecture Overview (New Techniques to be Presented in Sects. 9 and 10 are Sketched at the Top)

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, since the holding of those vertices in *frontier* will be released by the current task right after *compute*(.) returns.

To maximize CPU core utilization, each computing 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 computing 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 new tasks 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 needs 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 shown in Machine 1 of Fig. 3, each machine maintains a list of task files spilled from the task queues of the computing 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 the local vertex table. Note that tasks are spilled to disks and loaded back in batches to minimize the number of random IOs, as well as lock-contention by the computing threads on a global task-file list that tracks the current task files.

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 appending them to the task-file list on the local disk. These tasks will later be loaded by a computing thread for processing when its task queue needs a refill.

Desirabilities. This architecture design always guarantees that a computing 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 computation workloads, the linear IO cost of fetching/moving data is seldom a bottleneck. This architecture exhibits an excellent performance in most of the applications and datasets we tested, where the computation cost of an individual task is much less than the total computation workloads. The above assumption may not hold in some (but rare) cases which we do identify after testing on more datasets in this journal extension. This issue requires a smarter system-algorithm codesign to allow more effective load balancing, and we will describe these improvements later in Sects. 9 and 10.

Other desirabilities of our architecture design 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

Table 1 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

disks (and loaded back) in batches to achieve serial disk IO, and spilled tasks are prioritized for refilling task queues of the computing 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 and maintenance; (4) in contrast, tasks are totally independent (due to the 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 high network throughput; and (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 computing threads for parallel processing; moreover, work stealing among machines will send tasks from busy machines to idle machines.

We remark that G-thinker is the only system that achieves these desirabilities and hence CPU-bound computation workloads. Table 1 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 multiple computing threads, while inserting newly requested vertices and tracking whether an existing vertex can be evicted in the meanwhile; (2) how can we guarantee that a task will not request the adjacency list of a vertex v which has already been requested by another task in the same machine (even if response $\Gamma(v)$ has not been received yet) 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 the 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 Sect. 8, after we present G-thinker API and applications.

4 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 the set of vertices 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 the CPU core that it occupies. Another iteration of t will be scheduled once all its data responses are received.

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

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

Figure 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.[1]) **(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 this task in the next iteration.

Note that the template argument <VertexT> can be specified as any user-defined vertex class. For example, a vertex object v can have a label field, and each adjacency list item $u \in \Gamma(v)$ can be associated with the label of u, and the label and/or weight of edge (v, u). Moreover, the graph does not need to be undirected though in our applications discussed in this paper, we have $u \in \Gamma(v) \Leftrightarrow v \in \Gamma(u)$. For applications that need bidirectional search, a vertex can even maintain two adjacency lists, one for in-neighbors and one for out-neighbors. In a nutshell, our API is totally flexible to implement algorithms for weighted, labeled and/or directed graphs. When we pull a vertex by its ID of type <KeyT>, the entire vertex object with all its content (including $\Gamma(v)$)

[1] For a labeled graph, v's adjacency list item to its neighbor u may also keep the labels of u and edge (v, u).

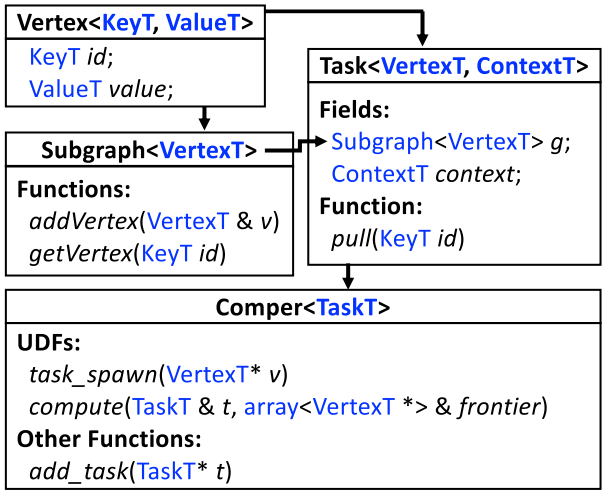


Fig. 4 Programming interface

gets fetched even though we simply say $\Gamma(v)$ gets fetched for simplicity.

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 $\underline{add\_task(t)}$ 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 *compute*(.) 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 vertices in *frontier*.

If g is too big (e.g., the number of vertices and/or edges is too large), in UDF *compute*(.), users may further decompose the task and add the generated tasks to the current comper's Q_{task} by calling $\underline{\text{add\_task(.)}}$. These tasks may be spilled to disk due to Q_{task} being full and then fetched by other compers or even 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 the 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 second by default). Before a job terminates, another synchronization is performed to make sure data from all tasks are aggregated.

We use aggregator in various applications: In maximum clique finding, aggregator tracks the maximum clique currently found, which is used by 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 triangle count for reporting.

Users can also trim the adjacency list of each vertex using a **(7) Trimmer** class. For example, in 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 as a preprocessing step right after the input graph is loaded, so that later during vertex pulling, only trimmed adjacency lists are responded back in order to reduce communication.

In the next three sections, we describe how to write 3 subgraph finding applications using our API: (1) maximum clique finding (MCF), (2) triangle counting (TC) and (3) subgraph matching (GM), respectively.

We have also conducted a separate study in [15] on computing maximal γ-quasi-cliques [21] with G-thinker, which has achieved over 370× speedup when mining the *YouTube* graph with over 1M vertices in our small 16-node cluster (32 threads each, 512 totally). Since the quasi-clique mining algorithm itself is very complicated, we leave it to our separate work of [15] and focus here on introducing the applications MCF, TC and GM.

5 Application I: Maximum clique finding

We next illustrate how to write a G-thinker program for the problem of finding a maximum clique following the set-enumeration 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 of each other, a vertex in $ext(S)$ should be the common neighbor of all vertices in S. For example, in Fig. 5, assume that $S = \{1, 2\}$, then $ext(S) = \{3, 4, 5, 8\}$ since they connect to both Vertices 1 and 2; Vertex 7, on the other hand, is not in $ext(S)$ since it is not connected with 2 so cannot be in the same clique with 2.

Initially, top-level tasks are $\langle S, ext(S)\rangle = \langle v, \Gamma_>(v)\rangle$, one for each vertex v. For example, in Fig. 5, we have tasks $\langle 1, \{2, 3, 4, 5, 6, 7, 8\}\rangle$, $\langle 2, \{3, 4, 5, 8\}\rangle$, etc.

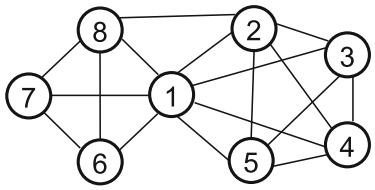


Fig. 5 A Graph for Illustrating the MCF Algorithm

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)

Fig. 6 UDF task_spawn(.) for Maximum Clique Finding

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 $\Gamma_>(S)$. For example, in Fig. 5, assume that $S = \{2, 3\}$, then $\Gamma(S) = \{1, 4, 5\}$ and $\Gamma_>(S) = \{4, 5\}$.

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 $|ext(S)|$ sub-tasks: $\langle S \cup u,\ \Gamma_>(S) \cap \Gamma_>(u)\rangle$, one for each $u \in ext(S)$. For example, in Fig. 5, consider a task with $S = \{1, 2\}$ and $ext(S) = \{3, 4, 5, 8\}$, then extending S with $u = 3$ gives a new task with $S' = \{1, 2, 3\}$ and $ext(S') = \Gamma_>(S) \cap \Gamma_>(u) = \{4, 5\}$, while extending S with $u = 8$ gives a new task with $S' = \{1, 2, 8\}$ and $ext(S') = \Gamma_>(S) \cap \Gamma_>(u) = \emptyset$.

To process a 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, we can obtain $C \cup S$ as a clique of G. For example, in Fig. 5, consider a task with $S = \{1, 2\}$ and $ext(S) = \{3, 4, 5, 8\}$, then we only need to mine cliques from the subgraph induced by {3,4,5,8} which gives a maximum clique $C = \{3, 4, 5\}$. Since both 1 and 2 connect to all vertices in C, we can obtain the current maximum clique $C \cup S = \{1, 2\} \cup \{3, 4, 5\} = \{1, 2, 3, 4, 5\}$.

Based on the above idea, the two UDFs of *Comper* are sketched in Figs. 6 and 7, respectively. In the pseudocode, 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 $t.context$ in the pseudocode. 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, let us consider *task_spawn*(v) in Fig. 6, which directly exits if v cannot generate a clique larger than S_{max} even if all v's neighbors are included (Line 1), where S_{max} is obtained from the aggregator. Otherwise, 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 then sets $t.S$ set as $\{v\}$. To construct $t.g$ as the subgraph induced by $\Gamma_>(v) \cup v$, task 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). For example, in Fig. 5, consider a task t spawned from $v = 2$, then it will pull vertices in $\Gamma_>(2) = \{3, 4, 5, 8\}$. As another example, assume that $S_{max} = \{1, 6, 7, 8\}$ has been identified, and t is spawned from $v = 7$ with $\Gamma_>(7) = \{8\}$, then Line 1 will prune t since, even if we include 8 into $S = \{7\}$, we only have a clique of 2 vertices which is smaller than S_{max}.

Next, let us consider *compute*(t, $frontier$), the algorithm of which is shown in Fig. 7. If $|t.S| = 1$ (Line 1), then t is a newly spawned top-level task with $S = \{v\}$ and with input argument *frontier* containing all pulled vertices $\in \Gamma_>(v)$. Line 2 thus constructs $t.g$ using the 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 thus we skip Line 2.

If $t.g$ is too big, e.g., has more than τ_{split} vertices (Line 3), we continue to create next-level tasks that are with smaller subgraphs (Lines 4–9). Here, τ_{split} is a user-specified threshold (set as 200,000 by default). 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.

On the other hand, 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). Note that even though we do not need that check for a split task due to Line 8, the check is useful for initial tasks spawned from individual vertices. If t is not pruned, we then run the serial MCF algorithm of [37] 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 with $t.S$ plus S'_{max}, where S'_{max} denotes the largest clique of $t.g$.

Here, *compute*(t, $frontier$) always returns *false* (c.f. Lines 11 and 14) to indicate that t is finished and can be freed from memory. In other applications like mining quasi-

```
Comper::task_spawn(v)
1: if |S_max| ≥ 1 + |Γ>(v)| then return
2: create a task t    // t = <v, Γ>(v)>
3: t.S ← {v}
4: for each u ∈ Γ>(v) do   t.pull(u)
5: add_task(t)

Comper::compute(t, frontier)
1 : if |t.S| = 1 then    // t.S = {v}
      // frontier contains Γ>(u) for all u ∈ Γ>(v)
2 :   construct t.g as the subgraph of G induced by
Γ>(v) ∪ v  // now consider a general task t = <S, Γ>(S)>
3 : if |V(t.g)| > τ_split then
4 :   for each u ∈ V(t.g) do
5 :     create a task t'
6 :     t'.S ← t'.S ∪ {u}
7 :     construct t'.g as the subgraph of t.g induced
by Γ>(t.S ∪ u)   // here, Γ>(t.S ∪ u) is u's "filtered"
adjacency list Γ>(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 τ vertices
11:   if |t.S| + |V(t.g)| ≤ |S_max| then return false
12:   S'_max ← 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 ← t.S ∪ S'_max
14: return false
```

Fig. 7 UDF compute(.) for Maximum Clique Finding

cliques, t may need to pull the neighbors of those vertices in *frontier* to further grow $t.g$ (into a 2-hop ego-network of the spawning vertex v), in which iteration $compute(t, frontier)$ should return *true*.

Also, note that using a set-enumeration search tree is not the only way to avoid redundancy. For example, in Sect. 7, we will see that our subgraph matching algorithm partitions the search space using different instances of the same vertex label.

6 Application II: Triangle counting

The next application we describe is triangle counting, which counts the total number of triangles in a big input graph. We want each triangle $\triangle v_1v_2v_3$ (w.l.o.g., $v_1 < v_2 < v_3$) to be counted exactly once, i.e., in the subgraph spawned by v_1 which is the smallest vertex (in terms of ID) in $\triangle v_1v_2v_3$. Let us reuse the graph in Fig. 5 for illustration: $\triangle 135$ should be counted by a task spawned from vertex 1 rather than 3 or 5, while $\triangle 345$ should be counted by a task spawned from 3.

We let v_1 count $\triangle v_1v_2v_3$ by checking whether $v_3 \in \Gamma(v_2)$, and since $v_3 > v_2$, we only need to check whether

```
Comper::task_spawn(v)
// Γ>(v) = {u1, u2, …, uk} sorted by vertex ID
1: if k < 2 then return
2: create a task t
3: t.S ← {v}
4: for each u ∈ {u1, u2, …, uk–1} do t.pull(u)
5: t.context ← uk
6: add_task(t)

Comper::compute(t, frontier)
1: count ← triangle_count(frontier, t.context)
2: aggregate(count)
3: return false

triangle_count(frontier, last)
// frontier = {o1, o2, …, ok–1}
// where oi = <ui, Γ>(ui)>; last = uk
1 : count ← 0
2 : v1_nbs ← {u1, u2, …, uk}
3 : for each j ∈ {1, 2, …, k–1} do
// define p as the position in v2's neighbor list v2_nbs
// define q as the position in v1's neighbor list v1_nbs
4 :   v2_nbs ← Γ>(uj)   // Γ>(uj) = {w1, w2, …, wm}
5 :   p ← 1,   q ← j + 1
6 :   while p < m and q < k do
7 :     if v2_nbs[p] = v1_nbs[q] do
8 :       count ← count + 1
9 :       p ← p + 1,   q ← q + 1
10:     else if v2_nbs[p] > v1_nbs[q] do p ← p + 1
11:     else do q ← q + 1  // v2_nbs[p] < v1_nbs[q]
12: return count
```

Fig. 8 Application Code for Triangle Counting

$v_3 \in \Gamma_>(v_2)$. For example, in Fig. 5, $\triangle 135$ is determined by checking that $5 \in \Gamma_>(3) = \{4, 5\}$; note that this builds first upon the fact that $3, 5 \in \Gamma_>(1)$.

As a result, we implement a trimmer to trim the adjacency list of any vertex v into $\Gamma_>(v)$. The trimmer also sorts the vertex IDs in the trimmed adjacency list in increasing order, which is useful in an efficient method to check condition $v_3 \in \Gamma_>(v_2)$ to be presented later.

For the sorted adjacency list $\Gamma_>(v_1) = \{u_1, u_2, \ldots, u_k\}$, each u_i can be v_2 in a $\triangle v_1v_2v_3$ except for the largest (and last) neighbor u_k, since there does not exist any vertex $v_3 \in \Gamma_>(v_1)$ such that $v_3 > v_2 (= u_k)$. For example, in Fig. 5 when $v_1 = 2$, $\Gamma_>(2) = \{3, 4, 5, 8\}$ and $u_k = 8$, then apparently 3, 4, 5 cannot be in $\Gamma_>(8)$. Therefore, a task t spawned by v_1 does not need to pull $\Gamma_>(u_k)$ to check condition $v_3 \in \Gamma_>(v_2)$.

Based on the above idea, the two UDFs of *Comper* are sketched in Fig. 8. First, let us consider $task\_spawn(v_1)$ in Fig. 8, which directly exits if $k < 2$ (Line 1) since in this case, $\Gamma_>(v_1) = \{u_1\}$ where $v_2 = u_1$ is the only neighbor of

v_1, and a triangle cannot be generated from only two vertices $\{v_1, v_2\}$. Otherwise, a task t is created (Line 2) with $t.S$ set as $\{v\}$ (Line 3). Line 4 then pulls $\Gamma(v_2)$ for all neighbors $v_2 \in \Gamma_>(v_1)$ except for the largest neighbor u_k. We also let task t track u_k in its context (Line 5), since it is needed to recover $\Gamma_>(v_1)$ later in *compute*(t,*frontier*) over which v_3 is iterated. For example, in Fig. 5 when $v_1 = 2$, $\Gamma_>(2) = \{3, 4, 5, 8\}$ and $u_k = 8$, so t pulls 3, 4, 5 and $t.context = 8$. Finally, the task is added to task queue Q_{task} of the comper that calls *task_spawn*(v) (Line 6).

Next, let us consider *compute*(t, $frontier$) as shown in Fig. 8. Specifically, Line 1 counts the number of triangles using the pulled *frontier* containing $\Gamma_>(v_2)$ for all $v_2 \in \Gamma_>(v_1) - u_k$, the details of which are described in the next paragraph. Line 2 then aggregates the counted triangle count to the local aggregator, which maintains the total number of triangles counted by the current worker, and which is periodically synchronized with aggregators of other machines to report the total number of triangles currently counted. Finally, *compute*(t,*frontier*) returns *false* to finish t in Line 3.

Finally, let us see how t counts the triangles which is given by function *triangle_count*(*frontier*, *last*) in Fig. 8. Specifically, a counter is initialized as 0 in Line 1 (to be incremented later in Line 8). Line 2 then recovers $\Gamma_>(v_1)$ as an array $v1\_nbs = \{u_1, u_2, \ldots, u_k\}$, where $u_1, u_2, \ldots$, and u_{k-1} are obtained from *frontier* and u_k is obtained from $t.context$. For example, in Fig. 5 when $v_1 = 2$, we recover $\Gamma_>(2)$ as $v1\_nbs \cup u_k = \{3, 4, 5\} \cup 8$.

We remark that array $v1\_nbs$ is also sorted by vertex ID since Line 4 of *Comper*::*task_spawn*(v) poses requests in order and this order is preserved when G-thinker receives responses into *frontier*.

Next, for each vertex $v_2 \in$ *frontier* which is given by $u_j = v1\_nbs[j]$ (Line 3), we first obtain the neighbor list $\Gamma_>(u_j)$ which is given by another array $v2\_nbs$ (Line 4). Then, we iterate v_3 over $v1\_nbs$ (i.e., $v_3 \in \Gamma_>(v_1)$) to determine if v_3 is also contained in $v2\_nbs$ (i.e., $\Gamma_>(v_2)$) and increment the counter if so (Line 8). For example, in Fig. 5 when $v_1 = 2$, $u_j \in \Gamma_>(2) = \{3, 4, 5, 8\}$. For a particular u_j (i.e., v_2), say 3, we have $v2\_nbs = \Gamma_>(3) = \{4, 5\}$, and we can determine if $\triangle 234$ exists (note that $4 \in \Gamma_>(2)$ and $4 > v_2 = 3$) by checking if $4 \in v2\_nbs$.

Given a particular v_2, the logic of checking all $v_3 \in \Gamma_>(v_1)$ with $v_3 > v_2$ for the condition of whether $v_3 \in \Gamma_>(v_2)$ is given by Lines 5–11. The key insight is that both $v1\_nbs$ and $v2\_nbs$ have their elements already ordered by vertex ID, so a merge-like operation can be applied that only require one pass over $v1\_nbs$ and $v2\_nbs$.

Specifically, let p (resp. q) be the current iterating position in $v2\_nbs$ (resp. $v1\_nbs$). Line 5 sets q as $(j + 1)$ to ensure that $v_3 > v_2$, i.e., v_3 iterates $v1\_nbs$ from the first vertex that is larger than v_2. Then, each time v_3 as given by $v1\_nbs[q]$ is compared with the next smallest vertex in $v2\_nbs$ (i.e., $\Gamma_>(v_2)$) to see if they are the same; if so, v_3 is in $\Gamma_>(v_2)$, and thus, the counter is incremented (Lines 7–8). Note that during the entire process of checking all $v_3 \in \Gamma_>(v_1)$, only one pass over $v2\_nbs$ (i.e., $\Gamma_>(v_2)$) is required as p only gets incremented.

7 Application III: Subgraph matching

The next application is subgraph matching. In the sequel, we first define the subgraph matching problem, then review the related work to motivate the use of G-thinker as an alternative and finally describe how to write G-thinker programs for subgraph matching.

Subgraph Matching. Given a small query graph G_Q, subgraph matching finds all its occurrences in a big data graph G. To illustrate, consider the query graph G_Q shown in Fig. 9(a), and the goal is to find from the data graph shown in Fig. 9(b) those subgraphs that match G_Q in terms of labels and edges between labels.

Note that each vertex in G_Q (resp. G) has a unique integer ID and a label. Let us define $k_1k_2k_3k_4k_5$ as a match where vertex with ID k_i in G is matched to vertex ⓘ in G_Q. For example, in Fig. 9(b), 25478 matches G_Q, while 25178 does not since G does not have edge $(1, 5)$ that corresponds to (③, ②) in G_Q.

Existing Methods: Vertex Traversal Plus Join. Almost all existing works on distributed graph matching follow the evaluation paradigm of (i) *acyclic path traversal* (IO-bound) plus (ii) *subgraph join* (with intermediate subgraph instances materialized). For example, both [14] and [31] first decompose a query graph into small acyclic subgraphs called twigs (see Fig. 9(c) for an illustration) and use vertex-centric graph exploration to find subgraph instances that match those twigs, and then join twigs on joint vertices (e.g., ③ and ② in Fig. 9(c)) to obtain the subgraphs that match G_Q.

Note that when G_Q contains cycles, vertex-centric graph exploration alone is not sufficient. For example, in Fig. 9(b), suppose that we perform vertex-centric exploration on G along query graph path ③ → ① → ②, we will explore from vertex 1 (or 4) to 2 and then to 5 simply according to neighbors' labels. Then, we need to check all b-labeled neighbors of vertex 5 to make sure vertex 1 (or 4) is among them, which is essentially an equi-join on k_3 (the vertex ID in G matched to query vertex ③) rather than a simple label-based exploration.

More recently, PruneJuice [29] adopts a similar two-stage approach: (1) vertex-centric path exploration to efficiently prune partial matches (e.g., $k_1k_2k_3k_4k_5$) that cannot be matched, and (2) a refinement step called *Template-Driven Search* (TDS) to eliminate false positives. The differences from [14] and [31] are that (i) both stages are conducted

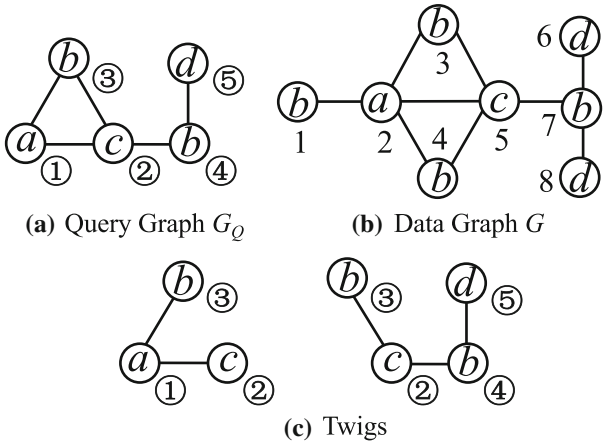


Fig. 9 An Example of Subgraph Matching

using vertex-centric message passing, and there is no subgraph join; (ii) Stage (1) is for pruning purpose rather than to generate partially matched twigs for later join, and two topological requirements are checked, i.e., cycle constraints and path constraints, which are efficient to execute in the vertex-centric paradigm and considers topological requirements beyond the immediate neighborhood of a vertex.

The pruning of PruneJuice in Stage (1) is very effective as demonstrated in [29], but the problem is with Stage (2) that refines each match with TDS, which verifies that each vertex visited in G meets its neighborhood constraints set by G_Q, where previously visited vertices need to be revisited making the checking IO-bound. In fact, the refinement could be even worse than subgraph join since matched vertices need to be revisited which requires additional vertex-centric message passing.

Another recent work, [25], proposes to combine binary joins with a novel worst-case optimal joins over partially matched subgraphs to construct matched subgraph instances in G, which has been demonstrated to be more efficient than using just one kind of subgraph-join operator. However, just like all the previously mentioned approaches, exponentially many partially matched subgraph instances need to be materialized making the execution IO-bound. An important contribution is, however, made by [25] where a cost-based optimizer is designed to find an efficient subgraph-join plan to execute.

Motivations to Use G-thinker. A CPU-effective solution requires us to think outside the box of existing vertex-traversal-plus-join design that is intrinsically IO-bound. G-thinker comes to the rescue exactly: assume that our match starting from a vertex v_Q in G_Q, and all other vertices of G_Q are within k hops from v_Q; then for each data vertex v in G that is matched to v_Q, if we pull all vertices with in k hops from v in G to construct a subgraph g to match upon, we will not miss any result and g can be checked simply by backtracking search without the need of materializing partially matched subgraph instances! This also avoids revisiting previous vertices as in PruneJuice's TDS that causes repeated data transmission.

With the above strong motivations, it is very promising to build a subgraph matching engine on top of G-thinker that is able to translate any user-provided query graph G_Q into our G-thinker UDFs (i.e., execution plans) for efficient execution. This would require (i) an initial statistics collection over the input graph G to be used by a query optimizer to find an efficient plan, (ii) a translator that translates the plan found into G-thinker UDFs *task_spawn*(v) and *compute*(t,*frontier*), (iii) change G-thinker execution engine from one-time job computation into one that supports on-demand online querying continuously, using a method similar to how we adapt our offline Pregel+ [42] system for vertex-centric computation into Quegel [43,46], a vertex-centric online query engine for graph traversal queries such as shortest paths and reachability.

Even though such a general-purpose subgraph matching engine on top of G-thinker is strongly motivated, its development workloads go beyond a journal extension and we are working on it as our future work.

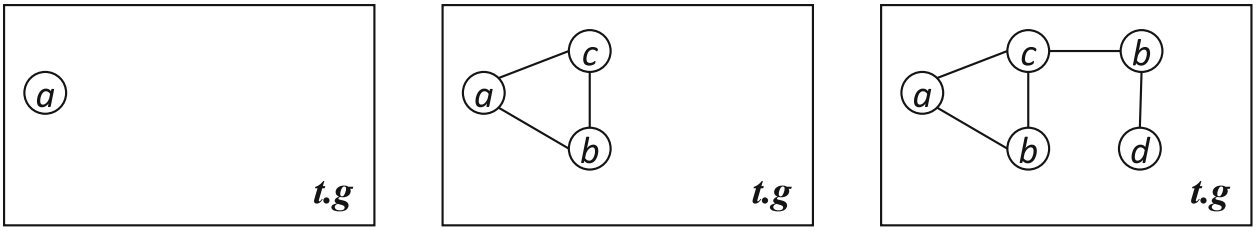
We next conduct a proof-of-concept study on the efficiency of the above solution by hard-coding the G-thinker UDFs for the specific query graph G_Q shown in Fig. 9(a), considering only the matching constraints within each vertex's immediate neighborhood (called local constraints in PruneJuice for pruning purpose) during the vertex-pulling stage to construct task subgraph g for subsequent backtracking search for matches.

G-thinker Algorithm for Subgraph Matching. We assume that each adjacency list item already contains the corresponding vertex's label besides the vertex ID. If this is not the case, one may use our Pregel algorithm for *attribute broadcast* as described in [42] to preprocess the data graph G in a cost linear to the size of G.

We implement a trimmer that removes all adjacency list items of vertices in G whose labels are not among "*a*," "*b*," "*c*" and "*d*" upon G is loaded from HDFS.

We start the subgraph matching from query vertex ① with label "a" (c.f. Fig. 9(a)), where a task is spawned from each data vertex in G with label "a" (denoted by v_a) to grow its subgraph g for later backtracking search. Note that every matched subgraph instance will be found since the subgraph must contain an a-labeled vertex. Also note that this subgraph will only be found once, i.e., by the task spawned from v_a.

The above logic is implemented in *Comper*'s UDF *task_spawn*(v_a), which creates a task t for a vertex v_a only if (1) v_a's label is "*a*," and (2) $\Gamma(v_a)$ contains both labels "*b*" and "*c*" (since query vertex ① in Fig. 9(a) has two neighbors with labels "*b*" and "*c*"). It then lets t pull those neighbors of v_a with label "*b*" or "*c*," and adds v_a to $t.g$. By the end



(a) *task_spawn*(v_a) **(b)** *compute*(t, *frontier*), iteration #1 **(c)** *compute*(t, *frontier*), iteration #2

Fig. 10 Illustration of the Subgraph Content in G-thinker for Subgraph Matching at the End of Different Steps

of *task_spawn*(v_a), the content of g is shown in Fig. 10(a). This UDF also sets $t.context$ (used for keeping the iteration number) to 1 so that when t calls UDF *compute*(.) later, it will enter the processing logic for iteration 1 (c.f., Fig. 10(b)).

Note that unlike the previous applications MCF and TC we saw in the previous two sections where UDF *compute*(.) has only one iteration, UDF *compute*(.) here in our subgraph matching application has two iterations. This is because, for example, query vertex ⑤ in Fig. 9(a) is two hops away from ① so we need to get vertices up to two hops from v_a in G for backtracking.

In UDF *compute*(t, $frontier$), we maintain the iteration number in $t.context$, based on which *compute*(.) branches to the processing of iteration 1 or iteration 2. Newly-spawned tasks will enter iteration 1 since $t.context$ is set as 1 by *task_spawn*(.). At the end of iteration 1, *compute*(.) will increment $t.context$ so that next time when it is called, it will enter iteration 2 for processing.

In iteration 1, we split *frontier* into two vertex sets: V_b (resp. V_c) which consists of vertices with label "b" (resp. "c"). While a vertex in V_c definitely matches vertex ② in Fig. 9(a), a vertex in V_b may match either ③ or ④. Therefore, for each vertex $v_c \in V_c$, we split those b-labeled vertices in $\Gamma(v_c)$ into two sets:

- U_1 consisting of those vertices that are also in V_b (i.e., they can match vertex ③ or vertex ④);
- U_2 consisting of the rest (which can only match vertex ④ since they are not neighbors of v_a).

Based on these two sets U_1 and U_2, we have 4 cases:

Case 1: $U_1 = \emptyset$. In this case, we prune v_c since v_c does not have a neighbor matching vertex ③.

Case 2: $|U_1| = 1$ and $U_2 = \emptyset$. In this case, we also prune v_c since v_c does not have two b-labeled neighbors.

Case 3: $|U_1| = 1$ and $U_2 \neq \emptyset$. In this case, the vertex in U_1 has to match vertex ③, and the vertex matching vertex ④ has to be from U_2. We thus pull all vertices of U_2 in order to find a neighbor matching vertex ⑤.

Case 4: $|U_1| > 1$. In this case, the vertex matching vertex ④ can be from either U_1 or U_2, and thus we pull all vertices from $U_1 \cup U_2$.

At the beginning of iteration 1, $t.g$ only contains v_a. When processing v_c, if Case 3 or Case 4 holds, we add v_c and edge (v_a, v_c) to $t.g$, and for each vertex $v_b \in U_1$ (i.e., v_b can match vertex ③), we add v_b and edges (v_a, v_b), (v_c, v_b) to $t.g$. Figure 10(b) provides an illustration of $t.g$.

Then in iteration 2 of *compute*(t, $frontier$), $frontier$ contains all vertices pulled by iteration 1, all of which have label "b" that can match vertex ④.

Let the set of all vertices with label "c" in $t.g$ (i.e., matching vertex ②) be V_c. Then, for each vertex $v_b \in$ *frontier*, we define V_d as the set of all vertices in $\Gamma(v_b)$ with label "d" (i.e., matching vertex ⑤).

If $V_d \neq \emptyset$, (1) we add v_b to $t.g$, and (2) for every vertex $v_c \in V_c \cap \Gamma(v_b)$, we add edge (v_c, v_b) (that matches (②, ④)) to $t.g$, and (3) for every vertex $v_d \in V_d$, we add v_d and edge (v_b, v_d) to $t.g$. Figure 10(c) provides an illustrative example of $t.g$ after the above processing.

Finally at the end of iteration 2, we run a backtracking algorithm on $t.g$ to enumerate all subgraphs that match the query graph. If we count the number of matched subgraphs rather than output them directly, we can sum the count of matched subgraphs to the aggregator similarly as in triangle counting.

Our graph matching algorithm does not decompose a big subgraph g into multiple tasks as in the application of MCF described in Sect. 5. But this can be easily done if splitting by a-labeled vertex alone leads to very unbalanced task workload distribution (e.g., some a-labeled vertex has a very high degree): if the subgraph g is big, we can further decompose it among the set of c-labeled neighbors of v_a that match query vertex ④, denoted by V_c. Specifically, for each $v_c \in V_c$, we can generate a task that further matches b-labeled vertices (e.g., split into U_1 and U_2); the task assumes that v_a and v_c are already matched to vertex ① and vertex ② in G_Q, respectively. In general, if g is big, a task t may continue to decompose g by looking at one more vertex in the query

graph (given that previous query vertices are already matched to the respective vertices in G).

8 System design and implementation

As Sect. 3 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 Sects. 8.1 and 8.2 , respectively. Now, let us 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 that holds $\langle v, \Gamma(v)\rangle$ in its 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 equivalently, 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 the input graph, spawns all other threads, periodically synchronizes job status to monitor job progress and to decide task stealing plans among workers.

G-thinker's communication module sends requests and responses in batches to guarantee high communication throughput while keeping latency low. Compers append pull-requests to the sending module, and the receiving module receives responses, inserts the received vertices (along with their adjacency lists) into T_{cache}, and notifies those pending tasks that are waiting for these requested vertices to update their task readiness.

In the next two subsections, we describe our design of vertex cache and task management, respectively.

8.1 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.

Figure 11 illustrates the design of T_{cache} with $k = 10$ buckets and $hash(v) = v \bmod 10$, where each row corresponds to a bucket. In reality, we set $k = 10,000$ which exhibits low bucket contention in our tests. As Fig. 11 shows, each bucket (i.e., row) consists of 3 hash tables (i.e., 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 containing $\Gamma(v)$ 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 T_{cache}. We now explain how each bucket of T_{cache} is updated atomically by various threads (i.e., compers, response receiving thread, and GC) with illustrations using Fig. 11.

There are 4 kinds of atomic operations OP1 – OP4 as described below. Since a bucket is protected by a mutex, only one operation may proceed at each time.

(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. 11). Moreover, if *lock-count*(v) was 0, it should be erased from Z-table (e.g., vertex 10 in Fig. 11). Otherwise, **Case 2.1:** if v is not found in R-table (see vertex 21 in Fig. 11), then it is requested for the first time, and thus v is inserted into R-table with *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 *lock-count*(v) is incremented to indicate that one more task is waiting for $\Gamma(v)$ (see vertex 20 in Fig. 11).

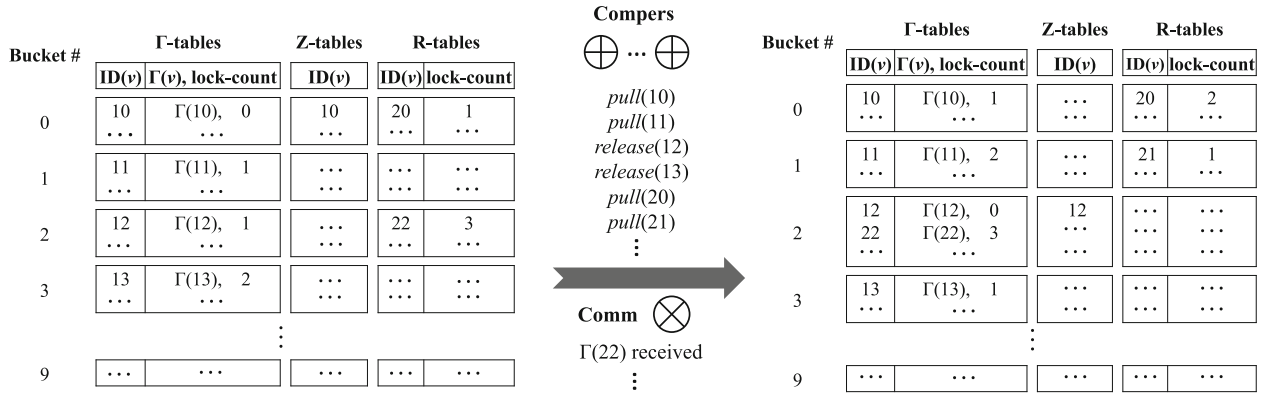


Fig. 11 Structure of Vertex Cache T_{cache} with $B_i = id(v) \bmod 10$; with Contents Before and After Atomic Update Operations

(OP2) When a response receiving thread receives a 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. 11). Note that *lock-count*(v) is directly transferred from v's old entry in R-table to its new entry in Γ-table, and the latter 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 *lock-count*(v) in the Γ-table (see Vertices 12 and 13 in Fig. 11), and if *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 the Z-table, GC removes their entries in both the Z-table and the Γ-table. Imagine, for example, the eviction of vertex 12 from T_{cache} shown on the right of Fig. 11.

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 only a small fraction of memory in each machine.

We include entries in R-tables when counting 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 mean 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 will (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 tasks that need v, 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}. Note that the error is bounded by merely $n_{comper} \times \delta$ 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 the 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 ultimately 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 in our tests (see Table 8(b) in Sect. 11 for the details).

8.2 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 pool of currently active tasks to release resources, so that more tasks can then be fetched for processing.

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 their requested vertices. Specifically, each comper maintains in-memory tasks in 3 task containers: a task queue Q_{task} that we already saw, a task buffer B_{task} and a task table T_{task}.

We now introduce why they are needed. Figure 12 summarizes the components in a comper (Fig. 12(b)), and their interactions with other worker components shared by all compers in a machine (Fig. 12(a)).

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 [26] $\mathcal{L}_{file}$ of file metadata for later loading, as shown in Fig. 12(a) with the list head and tail marked.

Recall that tasks are refilled when $|Q_{task}| \leq C$. To refill tasks, (i) $\mathcal{L}_{file}$ is examined first and a file is digested (if it exists) to load its C tasks; (ii) if $\mathcal{L}_{file}$ is empty, a comper then tries to generate new tasks from those vertices in T_{local} that have not spawned tasks (using *Comper*'s UDF *task_spawn*(v)) yet, 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 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 sources for task refilling like $\mathcal{L}_{file}$ and T_{local} are shared by all compers in a machine, the lock contention cost is amortized since a batch of tasks are fetched each time when 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. 12(b)) 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. 12(b)). Since each machine runs multiple compers, when a response containing $\Gamma(v)$ is received, the receiving thread needs to update the status of those tasks from all compers that are waiting for v. In other words, for each pending task t, it needs to track which comper holds t in its own task table T_{task}.

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. 11, when we introduced the vertex cache, 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. 12(a)).

Assume that a task t requests a set of vertices denoted by $P(t)$ in an iteration. In the task table T_{task} as illustrated in Fig. 12(b), an entry for a task t maintains key $id(t)$ and value $\langle met(t), req(t)\rangle$, where $\mathbf{req}(\mathbf{t})$ denotes the number of requested vertices $|P(t)|$, and $\mathbf{met}(\mathbf{t})$ denotes how many of them are already available. Both $met(t)$ and $req(t)$ are properly set when a comper inserts t into its own task table 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 Sect. 8.1 is described. The receiving thread also retrieves v's pending task list from its R-table entry, and for each $id(t)$ in

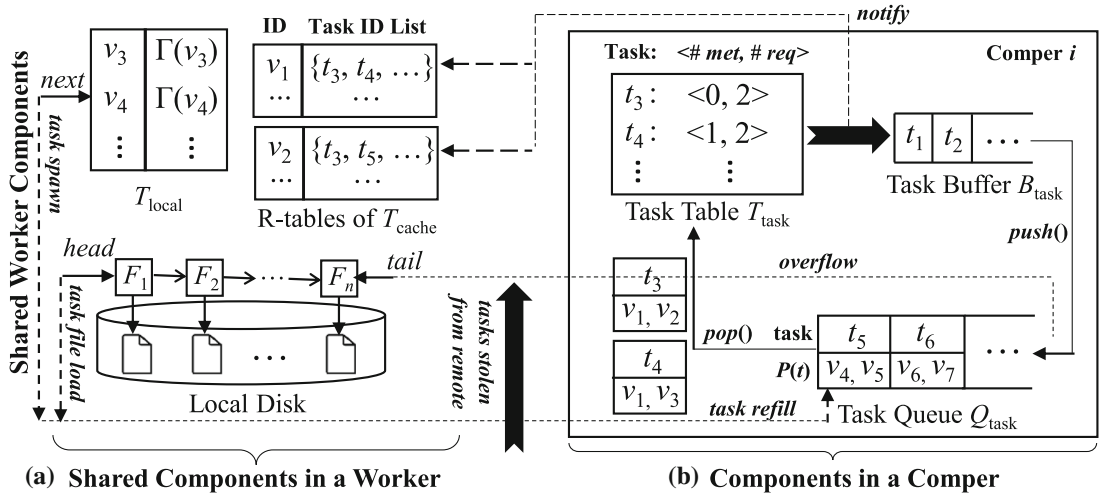


Fig. 12 Components in a Comper & Shared Components in a Worker

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}.

The Algorithm of a Comper. Now that we have seen the task containers maintained by a comper, we are ready to describe how a comper runs task computations. Every comper repeats the two operations *pop*() and *push*() (see Fig. 12(b)) in each round, and executes until the job end-signal is set by the main thread.

Push: If B_{task} is not empty, **push()** gets a task t from B_{task} and computes t 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. 12(b)). In UDF *compute*(.), when a task t pulls v (i.e., when users call *pull*(v)), v is simply added to $P(t)$. The actual examination of v on T_{cache} is done by *pop*() below.

Pop: 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 Sect. 8.1), 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 $\mathcal{L}_{file}$, (2) B_{task}, (3) spawning new tasks from vertices in T_{local}. This strategy prioritizes tasks that were processed the earliest, which minimizes the number of disk-resident tasks spilled from Q_{task}.

A task t always releases its holding of all its previously requested non-local vertices from T_{cache} after each iteration of computation (i.e., call of UDF *compute*(t, *frontier*)), 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), to keep memory consumption bounded. 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 (with other worker machines) if there are more tasks (e.g., stolen from other workers).

A G-thinker job terminates if the main thread of all machines find that all their compers are idle.

One problem remains: when a comper pulls vertices of $P(t) = \{v_1, v_2, \ldots, v_\ell\}$ 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 will get 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, the comper 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. Specifically, the main threads of all workers periodically (every 1 second by default) 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 the stealing plans before the next progress synchronization.

The number of remaining tasks at a worker is estimated from $|\mathcal{L}_{file}|$ and the number of unprocessed vertices in T_{local}. Tasks stolen by a worker are added to its $\mathcal{L}_{file}$ to be fetched by its compers.

We, however, find that enabling work stealing leads to almost the same performance as without work stealing, since most tasks exchanged among workers are not time-consuming to compute, and the time used to transmit such a task t (including its subgraph g) is not shorter than the time to compute t directly.

It is, therefore, important to locate and only exchange tasks that are time-consuming to compute, as in our G-thinker improvement to be described in Sect. 9.

Fault Tolerance. Our design also naturally supports checkpointing for fault tolerance: worker states (e.g., $\mathcal{L}_{file}$, 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 (since T_{cache} starts "cold").

9 System improvement for load balancing

Motivation for Improvement. While the previous design works well on the applications and graphs which we tested in our ICDE conference paper [44], we tested more graph datasets in this journal extension and found some cases where the previous design is inefficient. Specifically, on the *LiveJournal* dataset (see Table 2 in Sect. 11) with 7,489,073 vertices and 112,305,407 edges, the system takes over 1,300 seconds to find the maximum clique, which is quite slow considering that it only takes 354 seconds on the larger *Friendster* dataset with 65,608, 366 vertices and 1,806,067,135 edges.

The inefficiency is mainly caused by the drastically different costs of tasks in *LiveJournal*, where the largest vertex degree is 1,053,720. In contrast, the largest vertex degree in *Friendster* is merely 5,124. Since tasks spawned from high-degree vertices of *LiveJournal* can be very expensive, the previous design that operates on task-batches and endeavors to fill local task queues is no longer suitable: some tasks in a local task queue can be so expensive that the comper becomes the straggler.

Handling such graphs require a redesign of the system engine to identify and prioritize expensive tasks for computation and/or task decomposition, along with algorithmic improvement to most effectively utilize the new design's performance potential. In fact, more effective work stealing is now possible since the new design can identify expensive tasks. With both the system and algorithm improvement, the running time of our MCF application on *LiveJournal* reduces from over 1,300 seconds to within 168 seconds, and other graphs with quite a few high-degree vertices see similar improvements.

The rest of this section describes how we redesign G-thinker's system engine for better load balancing, while our new application code for MCF (maximum clique finding) on top will be described in Sect. 10.

New Worker Components. Our new design aims to prevent expensive tasks from being buffered in the local queue of a comper, but rather to move them around to idle threads for timely processing.

To achieve this goal, we categorize tasks into two categories: (1) big tasks whose subgraphs are large and expensive to compute upon, and (2) small tasks that are relatively more efficient to compute.

Our new engine is designed to allow big tasks to be scheduled as soon as possible, always before small tasks. For this purpose, we maintain separate task containers for big tasks and small ones, and always prioritize the containers of big tasks for examination and processing.

Figure 13 shows our improved design where the old comper-specific task containers Q_{task}, B_{task} and T_{task} (shown in Fig. 12(b)) are now denoted by Q_{local}, B_{local} and T_{local}, respectively, which are used to keep small tasks only. We similarly maintain three global task containers Q_{global}, B_{global} and T_{global} accessible to all the compers in a worker, to keep big tasks. Also, small tasks (resp. big tasks) that are spilled from Q_{local} (resp. Q_{global}) are now tracked by file-list $\mathcal{L}_{local}$ (resp. $\mathcal{L}_{global}$).

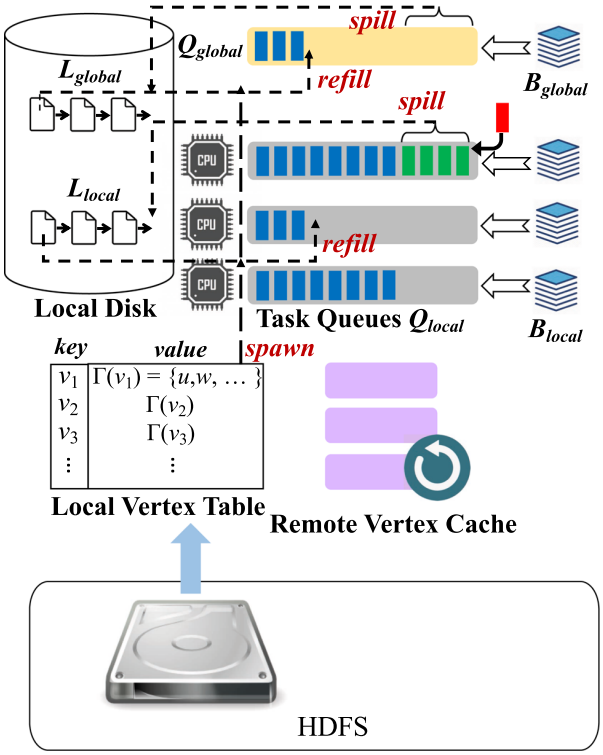


Fig. 13 Components in a Worker after Improvement

We define a user-specified threshold τ_{big} so that if a task t has a subgraph with potentially more than τ_{big} vertices to check, it is appended to Q_{global}; otherwise, it is appended to Q_{local} of the current comper. It could be difficult to decide the subgraph size of t as it changes with vertex pulling. So when t is still requesting vertices to construct its subgraph, we consider t as a big task iff the number of vertices to pull in the current iteration of *compute*(.) is at least τ_{big}, which prioritizes its execution to construct the potentially big subgraph early (recall Fig. 6); while when t is mining its constructed subgraph, for example in MCF where $t = \langle S, \Gamma_{>}(S)\rangle$ (recall Fig. 7), we consider t as a big task iff $|\Gamma_{>}(S)| > \tau_{big}$, since there are $|\Gamma_{>}(S)|$ vertices to check to expand S.

In our improved G-thinker system, *Comper* now has a new UDF *is_bigtask*(t) which is the only addition to the original G-thinker API shown in Fig. 4, and for MCF, we implement it to return whether $\max\{|P(t)|, |\Gamma_{>}(S)|\} > \tau_{big}$. Note that in MCF, $|\Gamma_{>}(S)| = |V(t.g)|$, i.e., the number of vertices in task subgraph $t.g$.

In general, to allow our other two applications to run with the improved system, we implement *Comper*::*is_bigtask*(t) simply to return whether $\max\{|P(t)|, |V(t.g)|\} > \tau_{big}$. Even though TC and GM do not decompose big tasks further, big tasks are still prioritized in scheduling and thus load balancing is improved. The default value of τ_{big} is 1,000 which we found to consistently work well in our various tests. If users call *add_task*(t), the function will put t into either Q_{local} or Q_{global} based on the above condition judgment.

Changes to the Algorithm of a Comper. There are three major changes made to prioritize big tasks.

The **first change** is with "push": a comper keeps flowing those tasks that have their requested data ready for computation, by (i) first fetching a big task from B_{global} for computing. The task may need to be appended back to Q_{global}, or may be decomposed into smaller tasks to be appended to Q_{global} or the comper's Q_{local} depending on child-task size. (ii) If B_{global} is, however, found to be empty, the comper will instead fetch a small task from its B_{local} for computation.

The **second change** is with "pop": a comper always fetches a task from Q_{global} first. If (I) Q_{global} is locked by another comper (i.e., a try-lock failure), or if (II) Q_{global} is found to be empty, the thread will then pop a task from its local queue Q_{local}.

In Case (I) when checking Q_{global} to pop, if the number of tasks is below a batch size C, the comper will try to refill a batch of tasks from $\mathcal{L}_{global}$. We do not check B_{global} for refill since it is shared by all compers which will incur frequent locking overheads. Note that "push" already keeps flowing big tasks in B_{global} that are ready.

In Case (II) when there is no big task to pop, a comper will check its Q_{local} to pop, before which if the number of tasks therein is below a batch, task refill happens where lies our **third change** as presented below.

Specifically, the comper will refill tasks from $\mathcal{L}_{local}$, and then from its B_{local} in this prioritized order to minimize the number of partially processed tasks buffered on local disk that are tracked by $\mathcal{L}_{local}$.

If both $\mathcal{L}_{local}$ and B_{local} are still empty, the comper will then spawn a batch of new tasks from vertices in the local vertex table for refill. However, we stop as soon as a spawned task is big, which is then added to Q_{global} (previous tasks are added to Q_{local}). This avoids generating many big tasks out of one refill.

New Work Stealing Strategy. Finally, since the main performance bottleneck is caused by big tasks, task stealing is conducted only on big tasks to balance them among machines. The number of pending big tasks (in Q_{global} plus $\mathcal{L}_{global}$) in each machine is periodically collected by a master (every 1 second by default), which computes their average and generates stealing plans to make the number of big tasks on every machine close to that average. If a machine needs to take (resp. give) less than a batch of C tasks, these tasks are taken from (resp. appended to) the global task queue Q_{global}; otherwise, we allow at most one task file (containing C tasks) to be transmitted to avoid frequent task thrashing that overloads the network bandwidth. Note that in one load balancing

cycle (i.e., 1 second by default) at most C tasks are moved at each machine.

10 Improved algorithm for maximum clique

Motivation for the Timeout Mechanism. Recall that in our MCF application, each task is denoted by $\langle S, ext(S)\rangle$, where S is the set of vertices already included in a subgraph g, and $ext(S) = \Gamma_{>}(S)$ is the set of vertices that can extend g into a clique. As shown in Lines 3–9 of Fig. 7 in Sect. 5, our MCF program already recursively decomposes a big task $\langle S, \Gamma_{>}(S)\rangle$ into $|ext(S)|$ sub-tasks: $\langle S \cup u,\ \Gamma_{>}(S) \cap \Gamma_{>}(u)\rangle$, one for each $u \in ext(S)$. Since our new system prioritizes such big tasks for decomposition, speedup is expected. For example, on *LiveJournal*, we find that simply running with our new system reduces the job execution time from 1300 seconds to 425 seconds. However, it is still slower than on *Friendster* (taking 354 seconds only) the graph size of which is one order of magnitude larger.

The problem with this solution is that, even though load balancing improves, a lot of computation overheads are spent not on the actual mining, but rather on creating decomposed tasks whose subgraphs need to be materialized, and these subgraphs can be big themselves. See, for example, Line 7 of Fig. 7 in Sect. 5. We are using the default parameter $\tau_{split} = 200,000$ here, and if we reduce τ_{split} further, this overhead will increase fast as more tasks need to be decomposed. For example, with $\tau_{split} = 150,000$, the program could not complete in 90 minutes and was thus terminated by us.

Note that the for-loop in Line 4 of Fig. 7 runs for $|V(t.g)|$ iterations which is larger than $\tau_{split} = 200,000$, meaning that the task will generate over 200,000 new tasks by constructing their subgraphs which is very time-consuming. Fortunately, on *LiveJournal* we found that there are only 22 such tasks that need decomposition, and the benefit from better load balancing achieved outweigh the task creation overhead.

In fact, depending on how vertices are interconnected (e.g., vertex degree distribution), different tasks with comparable subgraph sizes can have drastically different computation cost, and some tasks with much fewer than $\tau_{split} = 200,000$ vertices in its subgraph can still be very expensive and require decomposition.

However, naïvely reducing τ_{split} backfires since a lot of tasks that are not expensive also get decomposed, causing a lot of overheads in materializing new task subgraphs. To explore finer-grained task decomposition to further improve load balancing without paying an overly high cost on unnecessary subgraph materialization, we propose a so-called *timeout mechanism* to locate and only decompose expensive tasks.

To provide an intuition of our timeout mechanism using the set-enumeration tree shown in Fig. 1, we would like to let a task explore the search space in depth-first order initially by backtracking as in a serial algorithm to avoid any subgraph materialization. If the task runs for a duration of τ_{time}, a timeout occurs and the task then wraps the remaining search space as new tasks for divide and conquer. The timeout threshold τ_{time} is set to 1 second by default which works well in our tests. We will elaborate on the algorithm later in this section.

The Serial MCF Algorithm: A Review. Recall that we run the serial MCF algorithm of [37] in Line 12 of Fig. 7 to mine the maximum clique of $t.g$ if a task t does not need decomposition. Our timeout mechanism actually relies on this algorithm which we review next.

This serial algorithm itself is a recursive one that conducts a depth-first traversal of our set-enumeration search tree, and it uses an upper bound computed by vertex coloring to prune the search space. Specifically, this method assigns colors to vertices, so that no two adjacent vertices are colored with the same color. Vertex colors are used for pruning in [37]'s algorithm.

Note that the number of colors is an upper bound of the size of a maximum clique in a graph, since no two vertices in a clique can share the same color (as they are mutual neighbors). Since vertex-coloring is NP-hard, an approximate coloring algorithm called *ColorSort* is applied in [37] which assigns a color to each vertex in $ext(S)$ (i.e., the set of candidate vertices to extend S into a clique).

In *ColorSort*$(ext(S))$, all vertices in $ext(S)$ are colored one by one where a vertex v is inserted to the first possible color class C_k so that v is not adjacent to all the vertices already in C_k. If v has at least one adjacent vertex in each color class $C_1, \ldots, C_k$, then a new color class C_{k+1} is created to insert v. Finally, *ColorSort*$(ext(S))$ concatenates all the vertex sets $C_1, C_2, \ldots$ to return an updated candidate list of $ext(S)$ where vertices are ordered in non-decreasing order of color-value k. Note that the last color-value in the returned $ext(S)$ is also the largest color, which is again an upper bound of the number of colors in the graph g induced by $ext(S)$, and hence an upper bound of the maximum clique size in g.

Figure 14 shows the pseudocode of [37]'s algorithm, where the maximum clique found so far is kept in S_{max}, and $max\_clique(S, ext(S), C)$ is recursive (see its Line 8) and extends S using the candidate vertices in $ext(S)$. Here, $ext(S)$ is organized as a vertex array, and C is a vertex-color array where the i-th element is the color-value of the i-th vertex in $ext(S)$.

The MCF mining over g starts by calling function $MCF(g)$. Initially, $S = \emptyset$, and Line 1 creates the initial $ext(S) = V(g)$ by sorting the vertices of g in non-increasing order of degree (i.e., high-degree vertices go the first), which is found to provide a tighter color-based bound [37].

MCF(g)
// the global maximum clique S_{max} is initialized as $\emptyset$
1: sort vertices of g in non-increasing order of degree
// assume the maximum degree in g is d_{max}
2: vertex degree array $C[.] \leftarrow \{1, 2, \ldots, d_{max}, (d_{max}+1), (d_{max}+1), \ldots, (d_{max}+1)\}$
3: max_clique($\emptyset$, $V(g)$, C)

max_clique(S, $ext(S)$, C)
// $ext(S)$ is structured as a vertex array
// C is a color array where $C[v]$ = the color of $v \in ext(S)$
1 : **while** $ext(S) \neq \emptyset$ **do**
2 : $v \leftarrow$ the last element in $ext(S)$
// v has the largest color-value
3 : **if** $|S| + C[v] > |S_{max}|$ **then**
4 : $S' \leftarrow S \cup \{v\}$
5 : $ext(S') \leftarrow ext(S) \cap \Gamma(v)$
6 : **if** $ext(S') \neq \emptyset$ **then**
7 : $C', ext(S') \leftarrow ColorSort(ext(S'))$
8 : max_clique($S', ext(S'), C'$)
9 : **else if** $|S'| > |S_{max}|$ **then** $S_{max} \leftarrow S'$
10: $ext(S) \leftarrow ext(S) - v$
// pop the last element of $ext(S)$
11: **else return**

Fig. 14 Serial MCF Algorithm of [37]

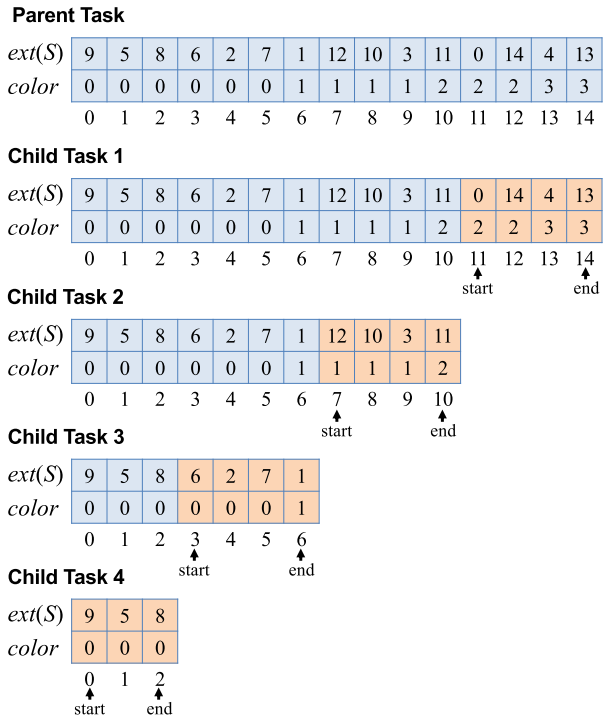


Fig. 15 Task Decomposition in the Timeout Mechanism

Then, $max\_clique(S, ext(S), C)$ is called in Line 3 with the prepared S, $ext(S)$, and an initial color array C prepared by Line 2 which we will explain later after describing $max\_clique(S, ext(S), C)$ below.

In $max\_clique(S, ext(S), C)$, we extend S with one more vertex $v \in ext(S)$ starting from the last vertex in $ext(S)$ backwards (Lines 1–2). To avoid redundant checking, the vertex v checked by an iteration is removed from $ext(S)$ in Line 10 before we consider the next iteration. Since we guarantee that vertices in $ext(S)$ are sorted by color (c.f., Lines 7–8), the candidate vertex v in Line 2 is guaranteed to have the largest color-value among the remaining vertices in $ext(S)$.

For the current vertex v, its color-value $C[v]$ serves as an upper bound of the maximum clique that can be found from the subgraph induced by the current $ext(S)$. So if $|S| + C[v] \leq |S_{max}|$, we cannot generate a larger clique than S_{max} by extending S with $ext(S)$. Thus, if the condition in Line 3 does not hold, then we can skip the current and all subsequent iterations (i.e., return directly in Line 11), since vertices in $ext(S)$ are popped backwards and $C[v]$ cannot increase with iterations.

Otherwise, a larger clique than S_{max} could be found, and thus, we create $S' = S \cup v$ (Line 4) and $ext(S') = \Gamma_{>}(S')$ (Line 5) and recursively call $max\_clique(.)$ over $\langle S', ext(S')\rangle$ if $ext(S') \neq \emptyset$ (Lines 6–8). While if $ext(S') = \emptyset$, clique S' cannot be further extended, and we check if it is larger than S_{max} and updates S_{max} if so (Line 9).

In the actual implementation, we directly reuse S for S' to avoid set copy: Line 4 becomes $S \leftarrow S \cup v$ but we will pop v out of S right after Line 9, so that S is recovered for use in the next iteration.

Finally, let us go back to Line 2 of $MCF(g)$: We need to assign the initial $C[i]$ to be a safe upper bound for the correctness of Line 3 in $max\_clique(.)$. For this purpose, we set $C[i] = d_{max} + 1$ where d_{max} is the largest vertex degree. This is correct since $(d_{max} + 1)$ is a natural upper bound of clique size (a vertex in a clique can be adjacent to $\leq d_{max}$ neighbors); we can further tighten $C[i]$ to be $V(g) = i$ if $i \leq d_{max}$, since vertices in $ext(S)$ are popped backwards and $V(g)$ itself serves as a natural upper bound of clique size.

Motivation to Open up the Serial MCF Algorithm. While our timeout mechanism can work with either the color-ignorant decomposition strategy in Lines 4–9 of Fig. 7 in Sect. 5, or the color-based decomposition strategy of [37] shown in Fig. 14, it is more desirable to use the latter for two reasons: (1) Color-based pruning can be utilized, and more importantly, (2) we can decompose the search space of $ext(S)$ into segments to generate less tasks, which we explain using Fig. 15 next.

Recall from Sect. 5 that each task in MCF is given by a pair $t = \langle S, ext(S)\rangle$ where $ext(S) = \Gamma_{>}(S)$. As Fig. 14 shows,

the algorithm of [37] organizes $ext(S)$ as an array, and in each iteration the last element v (see Line 2) is considered which divides the search space into 2 cases: (1) v is included into the clique (see Line 4) and the mining continues with the remaining candidates $ext(S')$ (see Line 8), and (2) v is excluded from the clique (see Line 10) and the mining continues with the remaining candidates by continuing the next iteration (see Lines 1–2).

A nice feature of this solution is that we can partition the candidate checking over $ext(S)$ into groups as illustrated in Fig. 15, where a parent task is divided into 4 child-tasks: (1) Child-task 1 considers extending S with one vertex being vertex 0, or 14, or 4, or 13; (2) Child-task 2 considers extending S with vertex 12, or 10, or 3, or 11; (3) Child-task 3 considers extending S with vertex 6, or 7, or 2, or 1; and (4) Child-task 4 considers extending S with vertex 9, or 5, or 8. For each task, however, we need to maintain a starting pointer (e.g., to vertex 12 for Child-task 2) so that when running the algorithm of Fig. 14 to check $ext(S)$ backwards from the end, we stop the iterating (Line 1 while-loop of *max_clique*(.)) as soon as the starting pointer is passed.

Compared with using the color-ignorant decomposition strategy in Lines 4–9 of Fig. 7 in Sect. 5, the new decomposition strategy wins from 3 aspects. **(i) Firstly,** the color-ignorant strategy generates one child-task for each $v \in ext(S)$, and when using the default parameter $\tau_{split} = 200,000$, it means that at least 200, 000 child-tasks are generated. In contrast, the strategy in Fig. 15 allows each child-task to process a group of candidates in $ext(S)$ so that the number of child-tasks is much smaller and each child-task has enough workload to justify its subgraph materialization and can benefit from our timeout mechanism for effective load balancing. **(ii) Secondly,** the strategy in Fig. 15 allows each child-task to exclude candidates of previous child-tasks during task creation to reduce the number of task data to materialize. While Line 7 of Fig. 7 may materialize a more shrunk $t'.ext(S)$, this color-ignorant strategy has to materialize as many as $|ext(S)|$ child-tasks, and vertices are highly redundantly replicated in child-tasks which can otherwise be avoided via grouping and backtrackings (before timeout). **(iii) Thirdly,** the color-ignorant strategy cannot utilize the color-based pruning in Line 3 of *max_clique*(.) in Fig. 14.

Hybrid Task Decomposition Strategy. Interestingly, we find that using the color-based decomposition strategy alone is not optimal, since the coloring process itself has a time complexity quadratic to the number of vertices in a graph, which is extremely slow when there are many vertices (e.g., 200, 000 which is the default value of τ_{split}). In this case, the color-ignorant decomposition strategy is much faster since vertex coloring is avoided, and the candidate vertex pruning by Line 7 of Fig. 7 is very effective in shrinking task subgraphs.

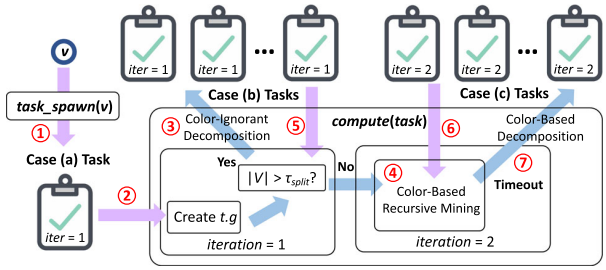


Fig. 16 Hybrid Task Decomposition

Comper::task_spawn(v)
1: **if** $|S_{\max}| \geq 1 + |\Gamma_{>}(v)|$ **then return**
2: create a task t // t = <v, $\Gamma_{>}(v)$>
3: $t.S \leftarrow \{v\}$, $t.iteration \leftarrow 1$, $t.start\_pos \leftarrow 0$
4: **for each** $u \in \Gamma_{>}(v)$ **do** $t.pull(u)$
5: *add_task*(t)

Fig. 17 UDF *task_spawn*(v) in Hybrid Task Decomposition

As a result, our final strategy is to use the color-ignorant decomposition strategy to decompose a task immediately if $|ext(S)| > \tau_{split}$. Otherwise, we use our color-based decomposition strategy which recursively mines a task subgraph until a timeout triggers task decomposition over the remaining workloads. Recall from Fig. 15 that we partition $ext(S)$ into segments of 4 vertices as groups, and we denote the number of vertices in each segment by τ_{seg}. The default setting is $\tau_{seg} = 10,000$ which works well on various graphs.

In our experiments in Sect. 11, we will compare this hybrid strategy with the algorithms that use either strategy but not both, to demonstrate the necessity and effectiveness of our hybrid task decomposition strategy.

The Final Algorithm. We now present our final MCF G-thinker algorithm that adopts the hybrid task decomposition strategy. In this algorithm, the *context* field of each task t is now given by a quintuple $\langle iteration, S, ext(S), C, start\_pos \rangle$, where $ext(S)$ is the candidate vertex array as illustrated in Fig. 15, *start_pos* is the start position in $ext(S)$ to check (also illustrated in Fig. 15), and C is the color array for the vertex array $ext(S)$.

Figure 16 overviews the algorithm workflow. Specifically, a task running UDF *compute*(t, *frontier*) can be either in iteration 1 or in iteration 2. Iteration 1 is dedicated to **Case (a):** a newly spawned vertex that constructs its subgraph from pulled vertices in *frontier* (c.f. ① in Fig. 16), or **Case (b):** a task created due to color-ignorant decomposition (c.f. ⑤). Iteration 2 is dedicated to **Case (c):** a task created due to color-based decomposition triggered by timeout (c.f. ⑥).

The algorithm of *Comper*'s UDF *task_spawn*(v) is shown in Fig. 17, which is almost the same as that in Sect. 5's old algorithm shown in Fig. 6. The difference is that when spawn-

```
Comper::compute(t, frontier)
1  : obtain S_max from the local aggregator
2  : if t.iteration = 1 then
3  :    construct t.g as the subgraph of G induced by Γ>(v) ∪ v
4  :    if |V(t.g)| > τ_split then
5  :       for each u ∈ V(t.g) do
6  :          create a task t'
7  :          t'.S ← t'.S ∪ {u}
8  :          construct t'.g as the subgraph of t.g induced by
Γ>(t.S ∪ u)
9  :          if |t'.S| + |V(t'.g)| > |S_max| then
10 :             t'.iteration ← 1,  t'.start_pos ← 0,  add_task(t')
11 :          else delete t'
12 :       return false
13 :    else if V(t.g) ≠ ∅ then
14 :       t.ext(S) ← sort vertices of t.g in non-increasing order
of degree
15 :       t.C ← {1, 2, …, d_max, (d_max + 1), (d_max + 1), …, (d_max
+ 1)}
16 : // code below will run by both iterations 1 and 2
17 : if |t.S| + |V(t.g)| ≤ |S_max| then return false
18 : if V(t.g) ≠ ∅ then
19 :    init_time ← the current time
20 :    S'_max ← a dummy size-(|S_max| − |t.S|) vertex set
21 :    S'_max ← max_clique_timeout(t.S, ∅, t.ext(S), t.C, S'_max,
t.start_pos, init_time)
22 :    if |S'_max| > |S_max| − |t.S| then  S_max ← t.S ∪ S'_max
23 : else if |t.S| > S_max then  S_max ← t.S
24 : return false
```

Fig. 18 UDF $compute(t, frontier)$ in Hybrid Decomposition

ing a task t out of v in Line 3, we set $t.iteration$ as 1 so that t will enter the first iteration in UDF $compute(t, frontier)$ later (c.f. ② in Fig. 16). We also set $t.start\_pos$ as 0 since the spawned task t should check all candidate vertices in array $ext(S)$ starting from the first vertex in $ext(S)$.

Next consider UDF $compute(t, frontier)$, where a task t can be either in iteration 1 or in iteration 2. The algorithm is shown in Fig. 18, where Lines 2–15 will only be run by a task in iteration 1, while Lines 17–24 will be run by tasks in both iteration 1 and iteration 2.

Specifically, $compute(t, frontier)$ first obtains the current maximum clique S_{max} from the aggregator in Line 1 to be used for pruning. Then, if t is newly spawned (i.e., Case (a)), it will run Line 3 to create its subgraph, where vertex objects in $\Gamma_>(v)$ are obtained from *frontier*; while if t falls in Case (b) (c.f. ⑤ in Fig. 16), *frontier* is empty since no vertex is pulled by the color-ignorant task decomposition, so Line 3 will do nothing.

Then, Lines 4–11 are almost the same as Lines 3–9 in Sect. 5's old algorithm shown in Fig. 7, where color-ignorant task decomposition is performed due to $ext(S) > \tau_{split}$. The difference is that in Line 10, we need to set the iteration of a newly-created task t' as 1 since t' falls in Case (b) (c.f. ③ in Fig. 16); we also need to set $t'.start\_pos$ as 0 since only a task of Case (c) may consider a subset of candidate in $ext(S)$ as shown in Fig. 15, while all other tasks need to check all candidates starting from the first vertex in array $ext(S)$.

After color-ignorant decomposition has generated all child-tasks (i.e., ③ in Fig. 16), *compute*(.) returns *false* in Line 12 to end the current parent-task.

In contrast, if $ext(s) \leq \tau_{split}$ in Line 4 and hence the color-ignorant task decomposition does not apply, *compute*(.) goes to Line 13 to prepare the initial candidate array $ext(S)$ and color array C in Lines 14–15 for recursive processing, which is the same as Lines 1–2 of $MCF(g)$ in Fig. 14. Note that the above lines are run by a task of Case (a) before recursion in Lines 17–24, while a task of Case (c) will directly run Lines 17–24 upon entering *compute*(.) since its iteration number is 2 and thus the condition in Line 2 fails (c.f. ⑥ in Fig. 16).

Next, consider Lines 17–24 of *compute*(.), which is similar to Lines 11–14 in Sect. 5's old algorithm shown in Fig. 7, except that rather than calling *max_clique*(.) of Fig. 14 to find the maximum clique from $t.g$, Line 21 now calls *max_clique_timeout*(.) (to be introduced soon) that may generate child-tasks due to timeout.

Specifically, if $ext(S)$ is empty (Line 18) and thus S cannot be further extended, we directly check if S can become a larger clique than S_{max} (Line 23). Otherwise, we call *max_clique_timeout*(.) to find the maximum clique from $t.g$, assuming that initially a clique of size S'_{max} is found (Lines 20–21). If a larger clique is found on $t.g$, it merges with S to generate a larger clique to update S_{max} (Line 22).

Note that both Lines 22 and 23 update S_{max} through the aggregator. There, a new clique is replaced into the aggregator only if the old S_{max} in the aggregator is smaller. Such a replacement may not actually happen since between Line 1 and Line 22 or 23, S_{max} could have been updated by another task run by another comper.

Finally, we present *max_clique_timeout*(.) which is called by Line 21 of Fig. 18 for the recursive mining of task $t = \langle S, ext(S)\rangle$ with a timeout mechanism to decompose long-running tasks. The algorithm of *max_clique_timeout*(.) is shown in Fig. 19, which assumes that all vertices of S_{prior} are already in a result clique, and which mines a maximum clique on the subgraph g induced by $ext(S_{prior})$, with the vertices of the current clique on g tracked by S for recursive expansion.

Note that (i) in our timeout mechanism, the recursive exploration of the set-enumeration tree happens before timeout, and (ii) *max_clique*(.) uses S_{max} for pruning during recursion as shown in Lines 3 and 9 of Fig. 14. Therefore, it will incur too many locking overheads if every task always obtains the up-to-date S_{max} from the local aggregator during recursion. We, therefore, only obtain S_{max} from the local aggregator when we create a new task for the first time (i.e., Line 1 of Fig. 18); while during recursion before a timeout, a task keeps using the latest S_{max} known by itself

```
max_clique_timeout(S_prior, S, ext(S), C, S_max, start_pos, init_time)
// S_prior is the set of vertices already in the clique, and we are mining
cliques on the subgraph g excluding S_prior
// S is the set of vertices in the current clique when mining g
// S_max is a local variable of compute(.) passed in as a reference
1 : while |ext(S)| – start_pos > τ_seg do
2 :    create a task t'
3 :    t'.iteration ← 2,  t'.start_pos ← |ext(S)| – τ_seg
4 :    t'.S ← S_prior ∪ S
5 :    t'.ext(S) ← ext(S),  t'.C ← t.C
6 :    add_task(t')
7 :    pop the last τ_seg elements from ext(S)
8 : while |ext(S)| – start_pos > 0 do
9 :    v ← the last element in ext(S)
10:    if |S| + C[v] > |S_max| then
11:       S' ← S ∪ {v}
12:       ext(S') ← ext(S) ∩ Γ(v)
13:       if ext(S') ≠ ∅ then
14:          C', ext(S')← ColorSort(ext(S'))
15:          if |S'| + last_element(C') > |S_max| then
16:             if now – init_time > τ_time then
17:                create a task t'
18:                t'.iteration ← 2,  t'.start_pos ← 0
19:                t'.S ← S_prior ∪ S',  t'.ext(S) ← ext(S')
20:                t'.C ← C',  add_task(t')
21:             else max_clique_timeout(S_prior, S', ext(S'), C', S_max, 0,
init_time)
22:       else if |S'| > |S_max| then S_max ← S'
23:       pop the last element of ext(S)
24:    else return
```

Fig. 19 Algorithm of *max_clique_timeout*(.)

in *max_clique_timeout*(.) without accessing the aggregator again. In other words, S_{max} in Fig. 19 is a reference object passed in at Line 21 of Fig. 18, rather than the value of the local aggregator.

Refer to Line 19 of *compute*(.) in Fig. 18 again: We first obtain the current time as the initial time to pass in *max_clique_timeout*(.). This is the initial time for recursion that stays the same during the recursive processing by *max_clique_timeout*(.), so that a task can know the elapsed time since the recursion begins, to check whether a timeout should be triggered to decompose the task.

Line 20 of *compute*(.) also computes S'_{max} to pass into *max_clique_timeout*(.) in Line 21 for recursion. Note that while the recursion may update the current maximum clique (by Line 22 of *max_clique_timeout*(.)), aggregator is not accessed during the entire process of recursion before timeout. While this task will not be notified about a larger clique found by other concurrent tasks, this is acceptable since it takes at most τ_{time} time before decomposition, and the reduced aggregator contention outweighs the early pruning enabled otherwise.

Now, let us look at *max_clique_timeout*(.). Lines 1–7 corresponds to the creation of child-tasks 1 to 3 in Fig. 15. Note that Line 3 sets $t'.iteration$ as 2 so that the child-task will run the task logic for Case (c) when calling *compute*(.) later (c.f. ⑥ in Fig. 16). The last group that contains at most τ_{seg} tasks (i.e., child-task 4 in Fig. 15) is actually processed by the current task in Lines 8–24, the logic of which is similar to *max_clique*(.) in Fig. 14 where candidate vertices are processed one-by-one backwards in $ext(S)$, except that (i) the processing stops once *start_pos* is passed (c.f. the condition in Line 8), and that (ii) if timeout occurs at a vertex $v \in ext(S)$ (Line 16), task $t' = \langle S', ext(S')\rangle$ where $S' = S \cup v$ is processed by a new task (Lines 17–20, c.f. ⑦ in Fig. 16) rather than recursively processed by the current task as in Line 21, and this also happens for the remaining search space (Lines 8-9) of the task due to timeout.

Note that each new child-task (with iteration number being 2) will later call *compute*(.) where Line 21 of Fig. 18 will call *max_clique_timeout*(.) to correctly enter its candidate processing logic in Lines 8–24 again.

Also, note that our timeout mechanism only decomposes a task to its proper granularity. Refer back to the upper right corner of Fig. 3 for an intuitive illustration. For example, let the initial time of recursion be t_0, and assume that in a Level-k recursion of *max_clique_timeout*(.), we find the current time $t_1 > t_0 + \tau_{time}$ for the first time in Line 16, then the remaining candidate vertices are wrapped as Level-$(k+1)$ tasks. Then, when we backtrack to Level-$(k-1)$, all remaining candidate vertices at Level-k are wrapped as Level-k tasks since their current time $t_2 > t_1$ and thus $t_2 > t_0 + \tau_{time}$. This process repeats during the backtracking, generating tasks at different levels.

Finally, note that each task t maintains $t.S = S_{prior}$ in its context and mines $t.g$ for a maximum clique S so that $S_{prior} \cup S$ is a valid clique that can be compared with S_{max} in the aggregator for replacement in Line 22 of Fig. 18. To allow task decomposition, a new task t' will have $t'.S = S_{prior} \cup S$ (c.f. Lines 4 and 19) and when it is scheduled for computation later, Line 21 of Fig. 18 will continue to find a maximum clique on $t'.g$ induced by $t'.ext(S)$ by calling *max_clique_timeout*(.) again.

11 Experiments

This section reports our extensive experiments. Specifically, Sect. 11.1 describes our experimental setting. Then, Sect. 11.2 compares G-thinker with existing systems, Sect. 11.3 compares the variants of our improved MCF algorithms, Sect. 11.4 studies the scalability of G-thinker, Sect. 11.5 studies the effect of various system and algorithm parameters on the performance, and finally, Sect. 11.6 reports our efforts to explore more challenging graph characteristics (size and density) and more diversified graph types, to demonstrate the generality and excellent performance of our solution even for graphs with extreme conditions.

Table 2 Graph Datasets for System Comparison

Dataset	$\|V\|$	$\|E\|$	$\|E\|/\|V\|$	Max Degree	$\|Q_{max}\|$	Source
YouTube	1,134,890	2,987,624	2.63	28,754	17	https://snap.stanford.edu/data/com-Youtube.html
Pokec	1,632,803	22,301,964	13.66	14,854	29	http://konect.cc/networks/soc-pokec-relationships
Skitter	1,696,415	11,095,298	6.54	35,455	67	https://snap.stanford.edu/data/as-Skitter.html
WikiTalk	2,394,385	4,659,565	1.95	100,029	26	https://snap.stanford.edu/data/wiki-Talk.html
Orkut	3,072,441	117,184,899	38.14	33,313	51	https://snap.stanford.edu/data/com-Orkut.html
Patent	3,774,768	16,518,947	4.38	793	11	https://snap.stanford.edu/data/cit-Patents.html
LiveJournal	7,489,073	112,305,407	15.00	1,053,720	9	http://konect.cc/networks/livejournal-groupmemberships
BTC	164,732,473	361,411,047	2.19	1,637,619	5	http://km.aifb.kit.edu/projects/btc-2009
Friendster	65,608,366	1,806,067,135	27.53	5,124	129	https://snap.stanford.edu/data/com-Friendster.html

11.1 Experimental setting

Systems Compared. We compare G-thinker with the state-of-the-art graph-parallel systems, including (1) the most popular vertex-centric system, **Giraph** [8], which is used to demonstrate that the vertex-centric model does not scale for subgraph mining, (2) **G-Miner** [6] open-sourced at [3], and (3) **Arabesque** [35] open-sourced at [1]. **NScale** [27] is not open-sourced and hence not compared. We also compare with the single-machine out-of-core subgraph-centric system such as **RStream** [38].

Applications. We use the 3 applications that we described in earlier section for performance study: (1) maximum clique finding (**MCF**), (2) triangle counting (**TC**), and (3) sub-graph matching (**GM**). We compare with Giraph on MCF and TC as their vertex-centric algorithms exist [5,28]. Arabesque also only provides MCF and TC implementations which we use for comparison.

For the MCF application, we consider five G-thinker implementations: (1) **MCF:** running the old MCF algorithm in Figs. 6 and 7 on the old G-thinker system described in Sect. 5; (2) **MCF-I:** running the old MCF algorithm in Figs. 6 and 7 on the improved G-thinker system described in Sect. 9; (3) **MCF-H:** running our new MCF algorithm with hybrid task decomposition as shown in Fig. 16 on our improved G-thinker; (4) **MCF-C:** the variant of MCF-H that only uses color-based task decomposition; in contrast, recall that MCF-I is the other variant that only splits tasks using τ_{split} (i.e., color-ignorant decomposition); finally (5) **MCF-S:** the variant of MCF-H that does not group candidate vertices into segments of size τ_{seg} as illustrated in Fig. 15 while still using the timeout mechanism; this corresponds to the case of MCF-H by fixing $\tau_{seg} = 1$, which generates a lot of decomposed tasks upon timeout, one for each candidate vertex.

Code. All relevant code and documentation related to our old G-thinker system can be found at the G-thinker website.[2] The improved G-thinker system and the code on top (including all MCF algorithm variants) are also open-sourced.[3]

Datasets. Table 2 shows the 9 real graph datasets that we use in our experiments for system comparison: (1) **YouTube** is the friendship social network of YouTube users; (2) **Pokec** is the friendship network from the Slovak social network Pokec; (3) **Skitter** is the Internet topology graph obtained from traceroutes run daily in 2005; (4) **WikiTalk** is the Wikipedia user network where edge (i, j) represents that user i edited a talk page of user j; (5) **Orkut** is the friendship social network of Orkut; (6) **Patent** is the citation graph of US patents where citations are made by patents granted between 1975 and 1999; (7) **LiveJournal** is the bipartite network of LiveJournal users and their group memberships; (8) **BTC** is the Billion Triple Challenge (BTC) 2019 dataset, a large-scale RDF crawl conducted from 12/12/2018 until 01/11/2019; and (9) **Friendster** is an online gaming network where users can form friendship edges.

Some of the above graph datasets are originally directed, and we have converted every directed edge as undirected since our applications work on undirected graphs. These datasets cover all types including social networks, RDF graphs, citation networks, bipartite networks, etc., and they exhibit different characteristics such as size and degree distribution.

For GM, we use the query graph of Fig. 9(a) and randomly generate a label for each vertex in the data graph among $\{a, b, c, d, e, f, g\}$ following a uniform distribution. Since only 7 labels are scattered in a big graph, a subgraph matching job is expected to be highly compute-intensive with many matched subgraphs.

Cluster Setting. All our experiments were conducted on a cluster of 16 machines each with 64 GB RAM, AMD EPYC 7281 CPU (16 cores and 32 threads) and 22TB disk. All reported results were averaged over 3 repeated runs. G-thinker requires only a tiny portion of the available disk and RAM space in our experiments.

[2] https://yanlab19870714.github.io/yanda/gthinker

[3] https://github.com/guimuguo/G-thinker

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}). We, therefore, omit disk space report.

11.2 System comparison

Default Parameters. Unless otherwise stated, our experiments use the well-tested default parameters listed as follows, which are verified to perform consistently well in various applications on various graphs.

Our improved G-thinker considers a task as a big task (i.e., adds it to Q_{global} instead of Q_{local}) if the task's subgraph has more than 1,000 vertices. Recall that tasks are flushed and refilled in batches. For big tasks, the batch size is $C = 10$, and for small tasks, the batch size is $C = 150$. We allow the number of tasks in task table plus task buffer to be at most $D = 8C$.

The capacity of remote vertex cache T_{cache} (i.e., c_{cache}) is 2M. GC evicts vertices only when the size of T_{cache} overflows, i.e., $s_{cache} > (1 + \alpha) \cdot c_{cache}$ with overflow tolerance parameter $\alpha = 0.2$. For MCF, the color-ignorant task decomposition threshold is $\tau_{split} = 200,000$, and the timeout threshold used in color-based task decomposition is $\tau_{time} = 1$ second.

In this subsection, for all our 3 applications, we run on both the old G-thinker system described in Sect. 5 (denoted by G-thinker) and on the improved G-thinker described in Sect. 9 (denoted by G-thinker+).

For MCF, we only consider two variants: one that runs the old MCF algorithm in Figs. 6 and 7 on G-thinker, the other that runs our new MCF-H on G-thinker+. Other MCF variants will be compared more comprehensively in Sect. 11.3.

Comparison among Distributed Systems. Table 3 reports (1) the job running time and (2) the peak memory consumption (taking the maximum over all machines) of our 3 applications over the 9 datasets shown in Table 2. We can see that Arabesque and Giraph incur huge memory consumption and could not scale to large datasets like *LiveJournal*, *BTC* and *Friendster*, since they keep the materialized subgraphs in memory.

G-Miner is relatively memory-efficient since it keeps tasks (containing subgraphs) in a disk-resident task priority queue; G-Miner is also more efficient than Arabesque and Giraph. However, while G-Miner can handle some large datasets such as *Friendster*, its performance can be tens of times slower than *G-thinker*. This is caused by its IO-bound disk-resident task queue, where task insertions are costly when the graph size and hence task number become large. G-Miner also fails to finish any application on *LiveJournal* and *BTC* within 24 hours, which is likely because of the uneven vertex degree distribution of these graphs where the dense part incurs enormous computation workloads (recall from Table 2 that their max degrees are beyond 1M), and G-Miner is not able to handle such scenarios efficiently.

As Table 3 shows, G-thinker consistently uses less memory than G-Miner and is consistently much faster than G-Miner: G-thinker is up to tens of times faster (c.f. the results on datasets *Okurt*, *Patent*, *Friendster*), without counting those experiments that G-Miner cannot finish in 24 hours.

Now, let us compare G-thinker with its improved version, G-thinker+, with better load balancing by prioritizing big tasks for scheduling. Note that in the application MCF, G-thinker+ runs MCF-H with our hybrid task decomposition strategy. We can see that (i) G-thinker+ is faster for most of the cases, and very close to G-thinker in the other cases. In contrast, (ii) G-thinker+ often wins over G-thinker by a large margin, such as in the experiments of MCF on *Skitter*, *WikiTalk*, *Orkut*, *LiveJournal* and *BTC*, and in the experiments of GM on *Orkut* and *Friendster*. This makes G-thinker+ a safe choice to begin with if users do not know which one to choose.

The original G-thinker engine, however, has its own merit when load balancing is not an issue, since there is no locking overhead on a global task queue. For example, G-thinker wins over G-thinker+ the most in the MCF experiment on *Friendster*, i.e., 354.23 seconds v.s. 441.24 seconds. This is because as a social network, *Friendster* does not have a very high degree vertex (recall from Fig. 2 that the maximum degree is 5,124) to spawn a straggler task.

On the other hand, when stragglers exist, using G-thinker+ combined with our new MCF-H algorithm with advanced task decomposition strategy can significantly boost the performance, e.g., from 1,300.34 seconds to only 167.67 seconds on *LiveJournal*.

Comparison with Single-Machine Systems. We also tested RStream whose code for TC and clique listing are provided [4]. However, their clique code does not output correct results. For triangle counting, RStream takes 53 seconds on *YouTube*, 283 seconds on *Skitter*, and 3,713 seconds on *Orkut*; in contrast, our G-thinker running with a single machine takes only 4 seconds on *YouTube*, 30 seconds on *Skitter*, and 210 seconds on *Orkut*. This large performance difference is as expected since RStream runs out-of-core and is IO-bound. For big graphs such as *BTC* and *Friendster*, RStream used up all our disk space.

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 Fig. 11 of [17] shows, Nuri takes over 1,000 seconds to find the maximum clique of *YouTube*, while our G-thinker demo video on http://bit.ly/gthinker shows that running 8 threads on one machine takes only 9.449 seconds to find the maximum clique.

Comparison with Graph DBMS Neo4j. Neo4j is a graph database management system that supports a query language called Cypher. With Cypher, we can write a query for sub-

Table 3 System Comparison

Dataset	Arabesque	Giraph	G-Miner	G-thinker	G-thinker+	
(a) Triangle Counting (TC):						**#{triangles}**
Youtube	88.88 s / 2.9 GB	67.61 s / 1.7 GB	7.54 s / 0.5 GB	7.05 s / 0.3 GB	**6.60*** s / 0.3 GB	3,056,386
Pokec	112.01 s / 5.6 GB	85.21 s / 3.3 GB	39.35 s / 0.6 GB	8.89 s / 0.6 GB	**8.86*** s / 0.5 GB	32,557,458
Skitter	133.48 s / 4.8 GB	67.25 s / 4.4 GB	34.80 s / 0.5 GB	7.86 s / 0.5 GB	**7.83*** s / 0.4 GB	28,769,868
WikiTalk	70.93 s / 2.8 GB	53.96 s / 1.7 GB	24.75 s / 0.5 GB	5.63 s / 0.3 GB	**5.61*** s / 0.3 GB	9,203,519
Orkut	783.51 s / 17.7 GB	179.27 s / 16.9 GB	667.00 s / 2.3 GB	23.46 s / 1.3 GB	**21.75*** s / 1.3 GB	627,584,181
Patent	128.33 s / 4.8 GB	97.62 s / 2.9 GB	380.89 s / 0.6 GB	**10.18*** s / 0.5 GB	10.34 s / 0.5 GB	7,515,023
LiveJournal	x	x	> 24 hr / 4.5 GB	**568.50*** s / 1.6 GB	578.87 s / 1.3 GB	141,388,608
BTC	x	x	> 24 hr / 6.7 GB	**103.97*** s / 3.5 GB	106.01 s / 3.4 GB	28,498,939
Friendster	x	x	10,915 s / 9.2 GB	531.45 s / 3.7 GB	**520.76*** s / 3.9 GB	4,173,724,142
(b) Maximum Clique Finding (MCF):						**Maximum Clique Size**
Youtube	95.04 s / 4.2 GB	142.39 s / 6.9 GB	12.07 s / 0.5 GB	8.74 s / 0.3 GB	**6.49*** s / 0.3 GB	17
Pokec	108.77 s / 8.2 GB	x	52.74 s / 0.7 GB	10.29 s / 0.6 GB	**8.64*** s / 0.5 GB	29
Skitter	233.45 s / 5.6 GB	x	141.42 s / 0.7 GB	56.02 s / 0.5 GB	**23.70*** s / 0.5 GB	67
WikiTalk	150.74 s / 5.3 GB	x	27.96 s / 0.6 GB	13.74 s / 0.3 GB	**8.74*** s / 0.3 GB	26
Orkut	3,231.27 s / 40.0 GB	x	691.00 s / 2.5 GB	70.84 s / 1.4 GB	**40.59*** s / 1.5 GB	51
Patent	98.07 s / 6.8 GB	x	439.83 s / 0.9 GB	9.44 s / 0.5 GB	**9.00*** s / 0.5 GB	11
LiveJournal	x	x	> 24 hr / 4.9 GB	1,300.34 s / 2.6 GB	**167.67*** s / 3.8 GB	9
BTC	x	x	> 24 hr / 7.3 GB	326.80 s / 3.8 GB	**120.82*** s / 4.9 GB	5
Friendster	x	x	10,295 s / 7.8 GB	**354.23*** s / 3.8 GB	441.24 s / 4.6 GB	129
(c) Subgraph Matching (GM):						**#{matched subgraphs}**
Youtube	–	–	19.74 s / 0.5 GB	**5.51*** s / 0.3 GB	6.16 s / 0.3 GB	75,591,525
Pokec	–	–	27.61 s / 0.7 GB	7.71 s / 0.4 GB	**7.50*** s / 0.4 GB	137,807,964
Skitter	–	–	20.78 s / 0.7 GB	7.91 s / 0.5 GB	**7.84*** s / 0.4 GB	3,848,300,318
WikiTalk	–	–	23.27 s / 0.8 GB	6.50 s / 0.5 GB	**6.47*** s / 0.5 GB	1,874,319,577
Orkut	–	–	98.98 s / 1.4 GB	71.45 s / 1.8 GB	**38.02*** s / 1.8 GB	103,900,798,537
Patent	–	–	23.85 s / 0.5 GB	**6.66*** s / 0.3 GB	7.66 s / 0.3 GB	3,113,131
LiveJournal	–	–	> 24 hr / 8.2 GB	420.61 s / 6.8 GB	**374.72*** s / 5.4 GB	351,168,596,617
BTC	–	–	> 24 hr / 6.0 GB	**116.17*** s / 5.0 GB	118.31 s / 3.2 GB	4,966,832,095
Friendster	–	–	3,669.00 s / 9.2 GB	2,649.92 s / 8.7 GB	**846.14*** s / 9.5 GB	422,289,263,153

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

graph matching (GM) for execution by Neo4j. For example, the query graph in Fig. 9(a) can be represented as:

```
MATCH (n1:v {label:'a'})
 -[:r]-(n2:v {label: 'b'})-[:r]-(n3:v
{label:'c'})-[:r]-(n4:v {label:'b'})
-[:r]-(n5:v {label:'d'}) WHERE
(n1)-[:r]-(n3) RETURN count(*) as count
```

We thus compare the performance of GM using Neo4j with that using G-thinker. One difficulty we encountered is that since our graphs are large, importing such a graph into Neo4j is very slow. This data import issue has also been reported in other works [12,32]. For example, [32] indicates that "Loading data into SQL Server takes significantly less time than loading into Neo4j. The time grows linearly with the sample size for SQL and grows exponentially for Neo4j." After exploration, we found that using the "neo4j-admin import"[4] command to import large graphs into Neo4j is efficient, which is adopted in our experiments reported below.

Neo4j has two releases that we tested, "Community Server" and "Neo4j Desktop." We first deployed Neo4j Desktop 1.4.1 with DBMS version 4.2.3 on a MacBook Pro laptop with 2.6 GHz 6-core Intel Core i7 CPU, 16 GB DDR4 RAM, and 512 GB SSD. Table 4(a) reports Neo4j's data import time and query execution time for GM when we apply the Cypher query above that corresponds to the query graph in Fig. 9(a), over those datasets that Neo4j is able to process on our laptop. We also include the running time of G-thinker for comparison. We can see that G-thinker is orders of magnitude faster than Neo4j for GM; in fact, it is even faster than the data import phase of Neo4j alone, thanks to G-thinker's efficient CPU-intensive backtracking workloads.

[4] https://neo4j.com/docs/operations-manual/current/tutorial/neo4j-admin-import/.

Table 4 Neo4j v.s. G-thinker for GM

(a) GM on Neo4j Desktop

Dataset	Data Import	Neo4j Execution Time	G-thinker Time
Youtube	11.85 s	321.17 s	6.16 s
Pokec	20.19 s	261.94 s	7.50 s
Skitter	12.11 s	> 8 hours	7.84 s
WikiTalk	19.83 s	> 8 hours	6.47 s
Orkut	Input Error	-	38.02 s
Patent	18.55 s	75.40 s	7.66 s

(b) GM on Neo4j Community Server

Dataset	Data Import	Neo4j Execution Time	G-thinker Time
Youtube	11.38 s	785.50 s	6.16 s
Pokec	24.18 s	1,142.64 s	7.50 s
Skitter	15.63 s	> 8 hours	7.84 s
WikiTalk	12.87 s	> 8 hours	6.47 s
Orkut	89.73 s	> 8 hours	38.02 s
Patent	24.61 s	87.49 s	7.66 s

Table 5 Neo4j v.s. G-thinker for TC

(a) TC on Neo4j Desktop

Dataset	Data Import	Neo4j Execution Time	G-thinker Time
Youtube	11.85 s	4.49 s	6.60 s
Pokec	20.19 s	17.15 s	8.86 s
Skitter	12.11 s	14.00 s	7.83 s
WikiTalk	19.83 s	8.67 s	5.61 s
Orkut	Input Error	-	21.75 s
Patent	18.55 s	10.77 s	10.34 s
LiveJournal	118.23 s	2,207.37 s	578.87 s

(b) TC on Neo4j Community Server

Dataset	Data Import	Neo4j Execution Time	G-thinker Time
Youtube	11.38 s	6.22 s	6.60 s
Pokec	24.18 s	20.87 s	8.86 s
Skitter	15.63 s	18.51 s	7.83 s
WikiTalk	12.87 s	10.25 s	5.61 s
Orkut	89.73 s	255.19 s	21.75 s
Patent	24.61 s	12.14 s	10.34 s
LiveJournal	95.44 s	2,457.71 s	578.87 s
BTC	626.84 s	15,089.57 s	106.01 s
Friendster	2,873.47 s	4,993.42 s	520.76 s

Neo4j Desktop seems to have a bug that leads to an input error "missing END_ID field" when we import *Orkut*, as we double-checked the input file but the format of the reported line is correct. This problem does not pop up in Community Server as Table 4(b) shows.

As Table 4(a) shows, Neo4j has reasonable performance on *YouTube*, *Pokec* and *Patent* given that the jobs were run on a laptop. However, on *Skitter* and *WikiTalk*, Neo4j cannot finish in 8 hours even though their time on G-thinker is similar to that of the other 3 datasets on G-thinker, and hence, we terminated the jobs.

We also deployed Neo4j's Community Edition 4.2.3 on a server in our cluster, but the performance was found to be worse than the Desktop release as shown in Table 4(b), possibly due to CPU frequency being lower. While *Orkut* can be imported in this setting, the GM job cannot finish on *Orkut* in 8 hours, while G-thinker finishes it in only 38.02 seconds.

Regarding the other two applications, TC and MCF, Neo4j only supports finding maximal cliques but not maximum cliques, so we only report the results of triangle counting, which are shown in Table 5. Note that *BTC* and *Friendster* cannot fit in the RAM of our laptop so are not reported. We can see that Neo4j performs well on small graphs though still slower than G-thinker, but on big graphs like *Orkut* and *BTC*, Neo4j can be one to two orders of magnitude slower, demonstrating the necessity of having a system like G-thinker for mining big graphs.

11.3 Comparison of MCF variants

We next compare the 5 variants of G-thinker's MCF algorithms. As a recap, MCF denotes the old MCF algorithm on the basic G-thinker system, while the other 4 variants are on the improved G-thinker: MCF-I runs the old algorithm (with color-ignorant task decomposition strategy only), MCF-H uses both the color-ignorant and the color-based task decomposition strategies, and MCF-C only uses the color-based strategy, while MCF-S is similar to MCF-H but set $\tau_{seg} = 1$.

Table 6 shows the performance of our 5 MCF variants on our 9 datasets. *Friendster* does not have a load balancing issue since its vertex degree distribution is not very biased and the major workloads of mining come from the large number of vertices, and therefore, MCF performs the best and much better than all the other variants that attempt to balance tasks using a shared big-task queue Q_{global} on each machine.

For all the other datasets, MCF-H performs consistently well. Note that although MCF-H is not the best on some datasets, the performance difference from the best algorithm is negligible and likely due to the randomness in different program runs caused by OS thread scheduling. For example, *Patent* essentially is insensitive to which algorithm is chosen and always completes in around 9 seconds.

The importance of our new task decomposition strategies is well demonstrated on *LiveJournal*. (1) Using the old algorithm, the old G-thinker needs 1,300 seconds due to the stragglers' problem. (2) By simply switching to the new system that prioritizes big tasks for processing, the time is already reduced to 425 seconds, which shows the effectiveness of our G-thinker improvement that adds Q_{global} to each machine. (3) By further using our new MCF-H algorithm with the advanced hybrid task decomposition strategies, the time is further reduced to 167.67 seconds, which shows the effectiveness of our new MCF algorithm.

Table 6 Comparison of MCF Variants

Dataset	MCF	MCF-I	MCF-H	MCF-C	MCF-S
Youtube	8.74 s / 0.3 GB	8.66 s / 0.3 GB	**6.49*** s / 0.3 GB	6.88 s / 0.3 GB	8.54 s / 0.4 GB
Pokec	10.29 s / 0.6 GB	9.68 s / 0.6 GB	**8.64*** s / 0.5 GB	9.89 s / 0.5 GB	9.63 s / 0.5 GB
Skitter	56.02 s / 0.5 GB	54.95 s / 0.4 GB	**23.70*** s / 0.5 GB	23.80 s / 0.5 GB	55.96 s / 0.5 GB
WikiTalk	13.74 s / 0.3 GB	11.58 s / 0.4 GB	8.74 s / 0.3 GB	**8.63*** s / 0.4 GB	13.67 s / 0.3 GB
Orkut	70.84 s / 1.4 GB	53.33 s / 1.4 GB	40.59 s / 1.5 GB	**39.14*** s / 1.5 GB	54.68 s / 1.4 GB
Patent	9.44 s / 0.5 GB	9.08 s / 0.5 GB	9.00 s / 0.5 GB	9.36 s / 0.5 GB	**8.93*** s / 0.5 GB
LiveJournal	1,300.34 s / 2.6 GB	425.06 s / 2.6 GB	**167.67*** s / 3.8 GB	716.09 s / 11.1 GB	1,124.32 s / 2.5 GB
BTC	326.80 s / 3.8 GB	311.68 s / 3.8 GB	**120.82*** s / 4.9 GB	631.58 s / 12.8 GB	322.50 s / 3.9 GB
Friendster	**354.23*** s / 3.8 GB	429.83 s / 4.6 GB	441.24 s / 4.6 GB	437.29 s / 4.5 GB	434.50 s / 4.4 GB

As an ablation study, (4) MCF-C removes the color-ignorant task decomposition strategy so that vertex coloring has to be run even for tasks with big subgraphs, which can be time-consuming. We can see that the running time is increased from 167.67 seconds to 716 seconds, even longer than MCF-I.

As another ablation study, (5) MCF-S does not allow a new task to handle a group of candidate vertices when time-out triggers task decomposition. This makes the new tasks having a lot of duplicate vertices materialized which can be avoided otherwise through backtracking over the candidate-grouped tasks (c.f. Fig. 15). We can see that the running time is increased from 167.67 seconds to 1,124.32 seconds, indicating the necessity of candidate-grouped task decomposition.

Overall, the integration of our color-based task decomposition strategy is non-trivial and has to be combined with color-ignorant task decomposition (to avoid expensive coloring in large task subgraphs) and with candidate-grouped task decomposition while using the timeout mechanism. Missing any technique here could backfire even compared with the simply *MCF-I*.

11.4 Scalability

Since both G-thinker and G-Miner can scale to *Friendster*, we compare their scalability using MCF. G-Miner additionally requires vertices to be pre-partitioned before computation. Unfortunately, we are not able to partition *Friendster* when there are only 2 machines or less, as G-Miner reports an error caused by sending more data than MPI_Send allows (the data size exceeds the range of "int"), and we denote these results as "Partitioning Error" in Table 7 on system scalability results.

Table 7(a) reports the vertical scalability when we use 16 machines but vary the number threads (or, compers). We can see that additional threads improves the performance of both systems, but the improvement is not significant beyond 16 threads. This limitation is because tasks spawned from many low-degree vertices do not generate large enough subgraphs to hide IO cost in the computation. Our machines are connected by GigE, and the problem may disappear if 10 GigE is used. G-thinker is also significantly faster than G-Miner.

Since 16 computing threads is the limit of vertical scalability, Table 7(b) reports the horizontal scalability when we vary the number of machines but fix the number of threads on each machine to 16. We can see that additional machines improve the performance and the speedup is significant. G-thinker is also significantly faster than G-Miner. The only exception is that running with 2 machines is slower than using 1 machine, and the slowdown is caused by the introduction of network communication which is, however, amortized well as more machines are used so that each machine incurs less communication volume.

Execution with a single machine is also interesting to explore when studying vertical scalability, since there are no remote data to request and this IO overhead (that causes task waiting) is avoided. Table 7(c) shows the results when we vary the number of threads, and we can see significant speedup all the way to 32 compers.

11.5 Effect of parameters

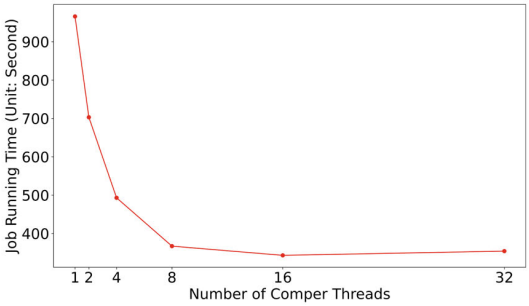
System Parameters. Table 8(a) shows the performance of G-thinker when we change vertex cache capacity c_{cache}. We can see that while small values of c_{cache} such as 0.2M and 0.02M make the performance much slower, the improvement from 2M to 20M is not significant; in contrast, to get this small improvement, the memory cost is significantly increased (from 3.9 GB to 9.7 GB). This justifies our default choice $c_{cache} = 2$M.

Table 8(b) shows the performance of G-thinker when we change the overflow-tolerance parameter α. Recall that GC keeps evicting unused vertices when the 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}

Table 7 MCF over Friendster: Scalability Results (along with the Speedup Charts)

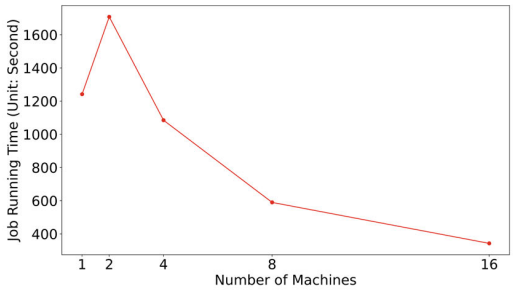
(a) Vertical Scalability (16 Machines)

# threads	G-Miner	G-thinker
1	> 24 hr / 7.8 GB	966 s / 4.0 GB
2	68,248 s / 7.8 GB	703 s / 4.4 GB
4	22,417 s / 7.9 GB	493 s / 4.5 GB
8	11,004 s / 7.9 GB	367 s / 3.7 GB
16	10,644 s / 7.8 GB	343 s / 3.7 GB
32	10,295 s / 7.8 GB	354 s / 3.8 GB



(b) Horizontal Scalability (16 Threads)

# machines	G-Miner	G-thinker
1	Partitioning Error	1,242 s / 17.3 GB
2	Partitioning Error	1,708 s / 11.5 GB
4	52,755 s / 14.0 GB	1,085 s / 7.1 GB
8	27,458 s / 9.0 GB	590 s / 4.8 GB
16	10,644 s / 7.8 GB	343 s / 3.7 GB



(c) Vertical Scalability (1 Machine)

# threads	G-Miner	G-thinker
1	Partitioning Error	8,556 s / 17.2 GB
2	Partitioning Error	4,688 s / 17.1 GB
4	Partitioning Error	2,670 s / 17.2 GB
8	Partitioning Error	1,702 s / 17.1 GB
16	Partitioning Error	1,242 s / 17.3 GB
32	Partitioning Error	1,082 s / 17.4 GB

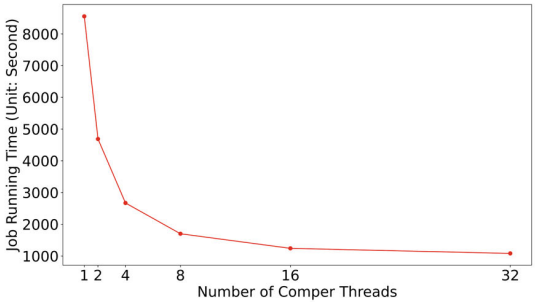


Table 8 MCF over Friendster: System Parameters

C_{cache}	Time	Memory
20M	291 s	9.7 GB
2M	359 s	3.9 GB
0.2M	667 s	2.3 GB
0.02M	958 s	1.7 GB

Note: M = 1,000,000

(a) Effect of c_{cache}

α	Time	Memory
0.002	383 s	4.0 GB
0.02	382 s	4.0 GB
0.2	359 s	3.9 GB
2	277 s	5.6 GB

(b) Effect of α

may contain $1.2 \cdot c_{cache}$ vertices (almost $1/3$ the memory used by the former), but the speedup is not very significant. This justifies that $\alpha = 0.2$ is a good trade-off between memory usage and task throughput.

Other system parameters are similarly chosen with extensive tests, and the complete results are omitted.

MCF Algorithm Parameters. Table 9 shows the performance of MCF-H when we fix one parameter of τ_{spilt} and τ_{seg} as the default value and vary the other.

As Table 9 (a) shows, $\tau_{spilt} = 200,000$ gives the best performance as it improves load balancing. We have also tried $\tau_{spilt} = 150,000$ but the time skyrockets, and we have to cut it after running for 90 minutes. This is because there are simply too many subgraphs being decomposed using the color-ignorant strategy, and since this strategy generates a task for each candidate in $ext(S)$, many subgraphs are materialized causing most time devoted to subgraph materialization.

Table 9 MCF Parameters, on *LiveJournal*

τ_{split}	Time	Memory
200,000	167.67 s	3.8 GB
250,000	233.16 s	4.0 GB
300,000	233.26 s	4.0 GB
350,000	286.25 s	4.2 GB
400,000	291.23 s	4.2 GB

(a) Effect of τ_{split} (τ_{seg} = 10,000)

τ_{split}	Time	Memory
100	1,558.82 s	6.1 GB
500	303.13 s	8.0 GB
1,000	241.79 s	6.8 GB
2,000	210.96 s	8.0 GB
5,000	196.48 s	4.8 GB
8,000	193.35 s	3.9 GB
10,000	167.67 s	3.8 GB
15,000	253.83 s	3.2 GB
20,000	245.48 s	3 GB
40,000	365.51 s	2.8 GB
80,000	723.42 s	2.5 GB
100,000	755.95 s	2.6 GB

(b) Effect of τ_{seg} (τ_{split} = 200,000)

As Table 9 (b) shows, $\tau_{seg} = 10,000$ gives the best performance. Smaller values of τ_{seg} cause more subgraph materialization overheads, while larger values of τ_{seg} limit the degree of parallelism as each decomposition generates fewer tasks.

11.6 MCF Experiments on larger and denser graphs

Recall from Table 2 that the largest maximum clique size we found on the previously tested datasets is 129, which is on *Friendster* that has the most edges (1.8 billion) and a high average degree of 27.53.

Given that our improved MCF-H algorithm is powerful in finding the maximum cliques, we would like to study its performance limit by pushing toward larger and denser graphs that are likely to have a very large maximum clique size. For this reason, we searched extensively the public graph datasets with three criteria that should be met together: (1) $|V| > 1,000,000$, (2) $|E|/|V| > 5$ and (3) maximum vertex degree $\geq 10,000$. We also pay special attention to graphs where the maximum vertex degree is close to or even larger than $|V|/2$, which are likely to have a very dense core.

Table 10 shows 9 datasets of various graph types that we found meeting the above criteria. Specifically, the first 4 datasets have the maximum vertex degree being close to or larger than $|V|/2$, while the other 5 meet our 3 degree and size criteria. Interestingly, only two graphs have large maximum cliques: *WikiLinks* and *WebUK* have maximum clique sizes of 1,109 and 944, respectively, while all the other datasets have a maximum clique size of no more than 55. This shows that (1) the maximum clique size is graph-dependent and is not easy to predict unless being actually mined, which shows the importance of having a fast MCF algorithm like our MCF-H; (2) graphs with a very high maximum vertex degree usually do not produce a large maximum clique size, which is a surprising result. For example, *MovieLens* is very dense with an average degree of 142.42 and a maximum degree $> |V|/2$, but $|Q_{max}|$ is merely 29.

Also, recall that the timeout threshold τ_{time} of MCF-H is set to 1 second by default. However, we find that this setting will lead to too many task decompositions on *WikiLinks* and *WebUK* that have a maximum clique size of around 1,000, since when the recursion depth is high, our timeout-based decomposition will generate too many tasks at deep levels (recall from the upper-left corner of Fig. 3), making the cost of materializing task subgraphs the dominant cost. Therefore, we tuned τ_{time} for these two datasets and found that $\tau_{time} = 2,500$ seconds gives the best performance for such deep-recursive scenarios in general.

Table 11 shows the performance of our 5 MCF algorithm variants on our 9 datasets in Table 10, where we used $\tau_{time} = 2,500$ seconds on *WikiLinks* and *WebUK*, and used the default $\tau_{time} = 1$ second for the other datasets. We can see that our recommended algorithm, MCF-H, performs consistently the best or close to the best on all the 9 datasets.

Table 10 Graph Datasets for Testing MCF Performance Limit on Larger and Denser Graphs

Dataset	$\|V\|$	$\|E\|$	$\|E\|/\|V\|$	Max Degree	$\|Q_{max}\|$	Graph Type	Source
MovieLens	70,155	9,991,339	142.42	35,100	29	Recommendation Networks	http://networkrepository.com/rec-movielens.php
Yeast	6,008	156,945	26.12	2,557	33	Biological Networks	http://networkrepository.com/bio-grid-yeast.php
Wiktionary	1,905,460	4,794,624	2.52	1,329,427	16	Online Contact Network	http://networkrepository.com/edit-frwiktionary.php
Tech-ip	2,250,498	21,643,497	9.62	1,833,161	4	Technological Networks	http://networkrepository.com/tech-ip.php
WikiLinks	12,150,976	288,257,813	23.72	962,969	1,109	Hyperlink Network	http://konect.cc/networks/wikipedia_link_en
WebUK	18,483,186	261,787,258	14.16	194,955	944	Web Graphs	http://networkrepository.com/web-uk-2002-all.php
StackOverflow	2,584,164	28,183,518	10.91	44,065	55	Online Contact Network	https://snap.stanford.edu/data/sx-stackoverflow.html
Wiki-topcats	1,791,489	25,444,207	14.20	238,342	39	Wikipedia Hyperlinks	https://snap.stanford.edu/data/wiki-topcats.html
DBpedia	18,268,991	126,890,209	6.95	612,308	41	Hyperlink Network	http://konect.cc/networks/dbpedia-link/

Table 11 Comparison of MCF Variants on Larger and Denser Graphs

Dataset	MCF	MCF-I	MCF-H	MCF-C	MCF-S
MovieLens	3,019.43 s / 0.5 GB	527.04 s / 1.1 GB	290.21 s / 1.6 GB	**277.18*** s / 1.6 GB	418.27 s / 1.3 GB
Yeast	3.03 s / 0.1 GB	**2.98*** s / 0.2 GB	3.08 s / 0.2 GB	3.08 s / 0.2 GB	3.09 s / 0.1 GB
Wiktionary	821.77 s / 1.0 GB	107.79 s / 1.1 GB	**106.52*** s / 1.2 GB	211.20 s / 6.6 GB	969.08 s / 1.0 GB
Tech-ip	32.51 s / 0.9 GB	32.78 s / 0.89 GB	**14.61*** s / 1.0 GB	16.57 s / 2.3 GB	70.39 s / 1.1 GB
WikiLinks	13,717.45 s / 25.8 GB	13,636.67 s / 29.1 GB	**5,170.68*** s / 27.8 GB	5,723.78 s / 28.0 GB	5,756.93 s / 26.5 GB
WebUK	691.98 s / 4.9 GB	714.85 s / 9.6 GB	**691.42*** s / 5.6 GB	693.20 s / 5.4 GB	693.20 s / 5.4 GB
StackOverflow	51.79 s / 0.7 GB	48.88 s / 0.6 GB	**26.63*** s / 0.8 GB	26.67 s / 0.8 GB	52.20 s / 0.7 GB
Wiki-topcats	199.39 s / 0.6 GB	202.65 s / 0.6 GB	28.37 s / 1.0 GB	**28.15*** s / 1.0 GB	257.57 s / 0.6 GB
DBpedia	300.32 s / 2.0 GB	297.24 s / 1.8 GB	**87.59*** s / 2.4 GB	153.34 s / 4.7 GB	325.46 s / 2.0 GB

Table 12 MCF-H Performance on *WikiLinks*: Effect of τ_{time}

τ_{time} (sec)	MCF-H Execution Time
5,000	100 min 56 sec
4,000	102 min 17 sec
3,000	106 min 36 sec
2,500	86 min 11 sec
2,000	198 min 02 sec

Table 13 Improved MCF-H on *WikiLinks*: Effect of τ_{dec}

τ_{dec}	Improved MCF-H Execution Time
0	86 min 11 sec
100	91 min 06 sec
500	94 min 06 sec
1,000	70 min 51 sec
5,000	194 min 30 sec

For example, MCF-H is one order of magnitude faster than MCF on *MovieLens* and many times faster than MCF on *Wiktionary*, *WikiLinks*, *Wiki-topcats* and *DBpedia*, demonstrating that our improved techniques in MCF-H effectively improved load balancing and eliminated straggler tasks.

As we mentioned earlier, it is an interesting new finding from running MCF-H over graphs in Table 10 that, for a graph with a large maximum clique size, we should not use the default $\tau_{time} = 1$ second but a much longer timeout threshold to prevent deep-recursive mining from generating too many small tasks at deep levels due to task timeout. To illustrate the impact of τ_{time}, we show the performance of MCF-H on *WikiLinks* in Table 12 when we vary τ_{time}. We can see that MCF-H achieves the best performance when $\tau_{time} = 2,500$ seconds. Larger τ_{time} reduces the effect in decomposing stragglers, but smaller τ_{time} leads to a lot of overhead caused by task decomposition in a deep set-enumeration tree. In fact, when we reduce $\tau_{time} = 1,000$, MCF-H could not finish in 8 hours and hence we terminated the job. Note that as shown in Table 11, MCF only takes 3.81 hours meaning that a very small τ_{time} backfires and the performance can be even worse than without conducting any task decomposition.

We further explored potential solutions to avoid decomposing tasks deep in the set-enumeration tree. Recall Fig. 19, where Line 16 decomposes the current task as soon as timeout is detected. We consider a variant where the if-condition in Line 16 additionally requires that $|ext(S)| \geq \tau_{dec}$, where τ_{dec} is a user-defined threshold that prevents a task from decomposition after timeout till backtracking to the level where $|ext(S)|$ is large enough. As a result, this strategy will avoid generating too many small tasks due to timeout.

We consider MCF-H expanded with the above improvement and ran it on *WikiLinks* with varying τ_{dec}. (Note that $\tau_{dec} = 0$ is essentially the original MCF-H.) The results are reported in Table 13, where we can see that $\tau_{dec} = 1,000$ significantly improves the performance compared with the original MCF-H (i.e., $\tau_{dec} = 0$). On the other datasets, this technique leads to performance similar to that of MCF-H, so we omit their results.

12 Conclusion

We presented the first truly CPU-bound graph-parallel system called G-thinker for large-scale subgraph finding, with a user-friendly subgraph-centric programming interface and a task-based execution engine. This journal extension further improved load balancing by proposing to add a global task queue to each machine for prioritized scheduling of big tasks. We also identified the performance weakness of a basic algorithm for maximum clique finding (MCF) and proposed hybrid task decomposition strategies (i.e., color-ignorant, plus color-based with timeout mechanism and candidate grouping) to scale to large graphs with a high vertex degree (e.g., over 1M on *LiveJournal*) and high density (e.g., $|E|/|V| = 142.42$ on *MovieLens*). Results show that our best MCF algorithm, MCF-H, is able to find large maximum cliques which require deep algorithmic recursion, such as 1,109 on *WikiLinks* in 70 minutes 51 seconds, and 944 on *WebUK* in

691 seconds, but for such graphs we need to use a large timeout threshold rather than the default value of 1 second to prevent task over-decomposition.

Many future works on top of G-thinker are currently under way in our research group, including mining pseudo-clique structures that are even more expensive to find than cliques, such as quasi-cliques and k-plexes. Another interesting future work is to develop a general-purpose engine for subgraph matching on top of G-thinker, which is a very appropriate topic as we have explained in Sect. 7, the subsection on "Motivations to Use G-thinker." Note that the search space of subgraph matching can be regarded as a state space tree [10] for efficient backtracking similar to how clique-like structures can be recursively searched using a set-enumeration tree, so all our proposed task decomposition and scheduling techniques in this paper can still be applied for parallel subgraph matching.

Acknowledgements This work was supported by NSF OAC-1755464 (CRII), IIS-1618669 (III) and ACI-1642133 (CICI), the NSERC of Canada and Hong Kong GRF 14201819. Guimu Guo acknowledges financial support from the Alabama Graduate Research Scholars Program (GRSP) funded through the Alabama Commission for Higher Education and administered by the Alabama EPSCoR.

References

1. Arabesque Code. https://github.com/qcri/Arabesque
2. COST in the Land of Databases. https://github.com/frankmcsherry/blog/blob/master/posts/2017-09-23.md
3. G-Miner Code. https://github.com/yaobaiwei/GMiner
4. RStream Code. https://github.com/rstream-system
5. Bu, Y., Borkar, V.R., Jia, J., Carey, M.J., Condie, T.: Pregelix: Big(ger) graph analytics on a dataflow engine. PVLDB **8**(2), 161–172 (2014)
6. Chen, H., Liu, M., Zhao, Y., Yan, X., Yan, D., Cheng, J.: G-miner: an efficient task-oriented graph mining system. In *EuroSys*, pages 32:1–32:12 (2018)
7. Cheng, J., Liu, Q., Li, Z., Fan, W., Lui, J. C. S., He, C.: VENUS: vertex-centric streamlined graph computation on a single PC. In *ICDE*, pages 1131–1142 (2015)
8. Ching, A., Edunov, S., Kabiljo, M., Logothetis, D., Muthukrishnan, S.: One trillion edges: Graph processing at facebook-scale. PVLDB **8**(12), 1804–1815 (2015)
9. Chu, S., Cheng, J.: Triangle listing in massive networks. *TKDD*, 6(4):17:1–17:32 (2012)
10. Csun, S., Luo, Q.: Parallelizing recursive backtracking based subgraph matching on a single machine. In *ICPADS*, pages 42–50. IEEE (2018)
11. Dean, J., Ghemawat, S.: Mapreduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150 (2004)
12. Dominguez-Sal, D., Urbón-Bayes, P., Giménez-Vañó, A., Gómez-Villamor, S., Martínez-Bazan, N., Larriba-Pey, J. L.: Survey of graph database performance on the HPC scalable graph analysis benchmark. In *WAIM Workshops*, volume 6185 of *Lecture Notes in Computer Science*, pages 37–48. Springer (2010)
13. Friendster. http://snap.stanford.edu/data/com-friendster.html
14. Gao, J., Zhou, C., Zhou, J., Yu, J. X.: Continuous pattern detection over billion-edge graph using distributed framework. In I. F. Cruz, E. Ferrari, Y. Tao, E. Bertino, and G. Trajcevski, editors, *ICDE*, pages 556–567. IEEE Computer Society (2014)
15. Guo, G., Yan, D., Özsu, M. T., Jiang, Z., Khalil, J.: Scalable mining of maximal quasi-cliques: An algorithm-system codesign approach. *PVLDB* (2021)
16. Hu, X., Tao, Y., Chung, C.: I/o-efficient algorithms on triangle listing and counting. *ACM Trans. Database Syst.*, 39(4):27:1–27:30 (2014)
17. Joshi, A., Zhang, Y., Bogdanov, P., Hwang, J.: An efficient system for subgraph discovery. In *IEEE Big Data*, pages 703–712 (2018)
18. Kyrola, A., Blelloch, G. E., Guestrin, C.: Graphchi: Large-scale graph computation on just a PC. In *OSDI*, pages 31–46 (2012)
19. Lee, J., Han, W., Kasperovics, R., Lee, J.: An in-depth comparison of subgraph isomorphism algorithms in graph databases. PVLDB **6**(2), 133–144 (2012)
20. Lin, W., Xiao, X., Ghinita, G.: Large-scale frequent subgraph mining in mapreduce. In *ICDE*, pages 844–855 (2014)
21. Liu, G., Wong, L.: Effective pruning techniques for mining quasi-cliques. In *PKDD*, pages 33–49 (2008)
22. Malewicz, G., Austern, M. H., Bik, A. J. C., Dehnert, J. C., Horn, I., Leiser, N., Czajkowski, G.: Pregel: a system for large-scale graph processing. In *SIGMOD Conference*, pages 135–146 (2010)
23. McCune, R. R., Weninger, T., Madey, G.: 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)
24. McSherry, F., Isard, M., Murray, D. G.: Scalability! but at what cost? In *HotOS* (2015)
25. Mhedhbi, A., Salihoglu, S.: Optimizing subgraph queries by combining binary and worst-case optimal joins. Proc. VLDB Endow. **12**(11), 1692–1704 (2019)
26. Michael, M. M., Scott, M. L.: Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *PODC*, pages 267–275 (1996)
27. Quamar, A., Deshpande, A., Lin, J.: Nscale: neighborhood-centric large-scale graph analytics in the cloud. *The VLDB Journal*, pages 1–26 (2014)
28. Quick, L., Wilkinson, P., Hardcastle, D.: Using pregel-like large scale graph processing frameworks for social network analysis. In *ASONAM*, pages 457–463 (2012)
29. Reza, T., Ripeanu, M., Tripoul, N., Sanders, G., Pearce, R.: Prunejuice: pruning trillion-edge graphs to a precise pattern-matching solution. In *SC*, pages 21:1–21:17. IEEE / ACM (2018)
30. Roy, A., Mihailovic, I., Zwaenepoel, W.: X-stream: edge-centric graph processing using streaming partitions. In *SOSP*, pages 472–488 (2013)
31. Shao, Y., Cui, B., Chen, L., Ma, L., Yao, J., Xu, N.: Parallel subgraph listing in a large-scale graph. In C. E. Dyreson, F. Li, and M. T. Özsu, editors, *SIGMOD*, pages 625–636. ACM (2014)
32. Sheroubi, M.: Benchmarking performance for neo4j in a social media application, https://people.ok.ubc.ca/rlawrenc/research/Students/MS_20_Thesis.pdf. *Computer Science Bachelor of Science Thesis, The University of British Columbia – Okanagan Campus* (2020)
33. Sun, Z., Wang, H., Wang, H., Shao, B., Li, J.: Efficient subgraph matching on billion node graphs. PVLDB **5**(9), 788–799 (2012)
34. Talukder, N., Zaki, M.J.: A distributed approach for graph mining in massive networks. Data Min. Knowl. Discov. **30**(5), 1024–1052 (2016)
35. Teixeira, C. H. C., Fonseca, A. J., Serafini, M., Siganos, G., Zaki, M. J., Aboulnaga, A.: Arabesque: a system for distributed graph mining. In *SOSP*, pages 425–440 (2015)
36. Tian, Y., Balmin, A., Corsten, S.A., Tatikonda, S., McPherson, J.: From "think like a vertex" to "think like a graph". PVLDB **7**(3), 193–204 (2013)
37. Tomita, E., Seki, T.: An efficient branch-and-bound algorithm for finding a maximum clique. In C. Calude, M. J. Dinneen, and

V. Vajnovszki, editors, *DMTCS*, volume 2731 of *Lecture Notes in Computer Science*, pages 278–289. Springer (2003)
38. Wang, K., Zuo, Z., Thorpe, J., Nguyen, T. Q., Xu, G. H.: Rstream: Marrying relational algebra with streaming for efficient graph mining on A single machine. In *OSDI*, pages 763–782 (2018)
39. Yan, D., Bu, Y., Tian, Y., Deshpande, A.: Big graph analytics platforms. Foundations and Trends in Databases **7**(1–2), 1–195 (2017)
40. Yan, D., Chen, H., Cheng, J., Özsu, M. T., Zhang, Q., Lui, J. C. S.: G-thinker: Big graph mining made easier and faster. *CoRR*, arXiv:1709.03110, (2017)
41. Yan, D., Cheng, J., Lu, Y., Ng, W.: Blogel: A block-centric framework for distributed computation on real-world graphs. PVLDB **7**(14), 1981–1992 (2014)
42. Yan, D., Cheng, J., Lu, Y., Ng, W.: Effective techniques for message reduction and load balancing in distributed graph computation. In *WWW*, pages 1307–1317 (2015)
43. Yan, D., Cheng, J., Özsu, M.T., Yang, F., Lu, Y., Lui, J.C.S., Zhang, Q., Ng, W.: A general-purpose query-centric framework for querying big graphs. Proc. VLDB Endow. **9**(7), 564–575 (2016)
44. Yan, D., Guo, G., Chowdhury, M. M. R., Özsu, M. T., Ku, W., Lui, J. C. S.: G-thinker: A distributed framework for mining subgraphs in a big graph. In *ICDE*, pages 1369–1380. IEEE (2020)
45. Yan, D., Qu, W., Guo, G., Wang, X.: Prefixfpm: A parallel framework for general-purpose frequent pattern mining. In *ICDE*, pages 1938–1941. IEEE (2020)
46. Zhang, Q., Yan, D., Cheng, J.: Quegel: A general-purpose system for querying big graphs. In F. Özcan, G. Koutrika, and S. Madden, editors, *SIGMOD*, pages 2189–2192. ACM (2016)