Scalable Mining of Maximal Quasi-Cliques: An Algorithm-System Codesign Approach

Guimu Guo
University of Alabama at Birmingham
guimuguo@uab.edu

Da Yan
University of Alabama at Birmingham
yanda@uab.edu

M. Tamer Özsu
University of Waterloo
tamer.ozsu@uwaterloo.ca

Zhe Jiang
University of Alabama
zjiang@cs.ua.edu

Jalal Khalil
University of Alabama at Birmingham
jalalk@uab.edu

ABSTRACT

Given a user-specified minimum degree threshold γ, a γ-quasi-clique is a subgraph $g = (V_g, E_g)$ where each vertex $v \in V_g$ connects to at least γ fraction of the other vertices (i.e., $\lceil \gamma \cdot (|V_g| - 1) \rceil$ vertices) in g. Quasi-clique is one of the most natural definitions for dense structures useful in finding communities in social networks and discovering significant biomolecule structures and pathways. However, mining maximal quasi-cliques is notoriously expensive.

In this paper, we design parallel algorithms for mining maximal quasi-cliques on G-thinker, a distributed graph mining framework that decomposes mining into compute-intensive tasks to fully utilize CPU cores. We found that directly using G-thinker results in the straggler problem due to (i) the drastic load imbalance among different tasks and (ii) the difficulty of predicting the task running time. We address these challenges by redesigning G-thinker's execution engine to prioritize long-running tasks for execution, and by utilizing a novel timeout strategy to effectively decompose long-running tasks to improve load balancing. While this system redesign applies to many other expensive dense subgraph mining problems, this paper verifies the idea by adapting the state-of-the-art quasi-clique algorithm, Quick, to our redesigned G-thinker. Extensive experiments verify that our new solution scales well with the number of CPU cores, achieving 201× runtime speedup when mining a graph with 3.77M vertices and 16.5M edges in a 16-node cluster.

PVLDB Reference Format:
Guimu Guo, Da Yan, M. Tamer Özsu, Zhe Jiang, and Jalal Khalil. Scalable Mining of Maximal Quasi-Cliques: An Algorithm-System Codesign Approach. PVLDB, 14(4): 573 - 585, 2021.
doi:10.14778/3436905.3436916

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/yanlab19870714/gthinkerQC.

1 INTRODUCTION

Given a degree threshold γ and an undirected graph G, a γ-quasi-clique is a subgraph of G, denoted by $g = (V_g, E_g)$, where each vertex connects to at least $\lceil \gamma \cdot (|V_g| - 1) \rceil$ other vertices in g. Quasi-clique is a natural generalization of clique that is useful in mining various networks, such as finding protein complexes or biologically relevant functional groups [5, 8, 10, 21, 29, 35], and social communities [20, 25] that can correspond to cybercriminals [37], botnets [34, 37] and spam/phishing email sources [33, 36]. Mining maximal quasi-cliques is notoriously expensive [32] and the state-of-the-art algorithms [26, 31, 39] were only tested on small graphs. For example, Quick [26], the best among existing algorithms, was only tested on graphs with thousands of vertices [26]. This has hampered its use in real applications involving big graphs.

In this paper, we design parallel algorithms for mining maximal quasi-cliques that scale to big graphs. Our algorithms follow the idea of divide and conquer which partitions the problem of mining a big graph into tasks that mine smaller subgraphs for concurrent execution. This has been made possible recently by the G-thinker [38] framework for distributed graph mining which avoids the IO bottleneck for data movement that exists in other existing data-intensive systems which could result in a throughput comparable or even less than a single-threaded program [2, 13].

However, we found that porting such a divide-and-conquer algorithm (hereafter called divisible algorithm for simplicity) directly to the current G-thinker implementation still leads to the straggler problem. This is because the state-of-the-art divisible algorithms for mining dense subgraphs such as quasi-cliques and k-plexes [15] are much more difficult than the applications that G-thinker already implemented, such as maximum clique finding and triangle counting [38]. For example, [32] showed that even the problem of detecting whether a given quasi-clique in a graph is maximal is NP-hard. Unlike those problems considered in [38] where the running times of individual tasks are relatively short, quasi-clique mining generates tasks of drastically different running time, which was not sufficiently handled by the G-thinker engine.

On the other hand, G-thinker's graph-divisible computing paradigm is a perfect fit for dense subgraph mining problems, and all we need to do is to redesign G-thinker's execution engine to address the straggler problem. Before the advent of G-thinker, such parallelization was not easy. For example, [32] makes it a future work "Can the algorithms for quasi-cliques be parallelized effectively?". Addressing the load balancing issue of G-thinker would not only benefit parallel quasi-clique mining, but also the parallelization of many other graph-divisible algorithms [7, 11, 15–17, 27, 28].

We adopt an algorithm-system codesign approach to parallelize quasi-clique mining, and the main contributions are as follows:

Proceedings of the VLDB Endowment, Vol. 14, No. 4 ISSN 2150-8097.
doi:10.14778/3436905.3436916

- We redesigned G-thinker's execution engine to prioritize the execution of big tasks that tend to be stragglers. Specifically, we add a global task queue to keep big tasks which is shared by all mining threads in a machine for prioritized fetching; task stealing is used to balance big tasks among machines.
- We improved Quick by integrating new pruning rules that are highly effective, and fixing some missed boundary cases in Quick that could lead to missed results. The new algorithm, called Quick+, is then parallelized using G-thinker API.
- We achieved effective and early decomposition of big tasks by a novel timeout strategy, without the need to predict task running time which is very difficult.

Our parallel solution has been extensively verified over real graphs. For example, we are able to obtain 201× speedup when mining 0.89-quasi-cliques on the *Patent* graph with 3.77M vertices and 16.5M edges in our 16-node cluster: the total serial mining time of 25,369 sec are computed by our parallel solution in 126 sec.

The rest of this paper is organized as follows. Section 2 reviews related work on quasi-clique mining and graph computing time prediction. Section 3 formally defines our notations, the general divisible algorithmic framework for dense subgraph mining which is adopted by Quick and which is amenable to parallelization using G-thinker. Section 4 then demonstrates that the tasks of Quick+ can have drastically different running time, and describes the straggler problem that we faced. Section 5 then reviews the original execution engine of G-thinker and describes our redesign to prioritize big tasks for execution. Section 6 then outlines our Quick+ algorithm and Section 7 presents its adaptation on G-thinker as well as a version using timeout-based task decomposition. Finally, Section 8 reports our experiments and Section 9 concludes this paper.

2 RELATED WORK

A few seminal works devised branch-and-bound subgraph searching algorithms for mining quasi-cliques, such as Crochet [22, 31] and Cocain [39] which finally led to the Quick algorithm [26] that integrated all previous search space pruning techniques and added new effective ones. However, we find that some pruning techniques are not utilized or fully utilized by Quick. Even worse, Quick may miss results. We will elaborate on these weaknesses in Section 6.

Sanei-Mehri et al. [32] noticed that if γ'-quasi-cliques ($\gamma' > \gamma$) are mined first using Quick which are faster to find, then it is more efficient to expand these "kernels" to generate γ-quasi-cliques than to mine them from the original graph. Their kernel expansion is conducted only on those largest γ'-quasi-cliques extracted by postprocessing, in order to find big γ-quasi-cliques as opposed to all of them to keep time tractable. However, this work does not fundamentally address the scalability issue: (1) it only studies the problem of enumerating k big maximal quasi-cliques containing kernels rather than all valid ones, and these subgraphs can be clustered in one region (e.g., they overlap on a smaller clique) while missing results on other parts of the data graph, compromising result diversity; (2) the method still needs to first find some γ'-quasi-cliques to grow from and this first step is still using Quick; and (3) the method is not guaranteed to return exactly the set of top-k maximal quasi-cliques. We remark that the kernel-based acceleration technique is orthogonal to our parallel algorithm and can be easily incorporated.

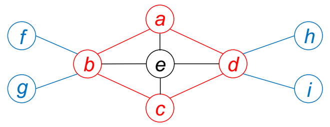

Figure 1: An Illustrative Graph

Other than [32], quasi-cliques have seldom been considered in a big graph setting. Quick [26] was only tested on two small graphs: one with 4,932 vertices and 17,201 edges, and the other with 1,846 vertices and 5,929 edges. In fact, earlier works [22, 31, 39] formulate quasi-clique mining as frequent pattern mining problems where the goal is to find quasi-clique patterns that appear in a significant portion of small graph transactions in a graph database. Some works consider big graphs but not the problem of finding all valid quasi-cliques, but rather those that contain a particular vertex or a set of query vertices [12, 14, 24] to aggressively narrow down the search space by sacrificing result diversity, with some additional pruning rules beyond Quick, some in the query-vertex context.

There is another definition of quasi-clique based on edge density [4, 14, 30] rather than vertex degree, but it is essentially a different kind of dense subgraph definition. As [14] indicates, the edge-density based quasi-cliques are less dense than our degree-based quasi-cliques, and thus we focus on degree-based quasi-cliques in this paper as in [14]. Brunato et al. [9] further consider both vertex degree and edge density. There are also many other definitions of dense subgraphs [7, 11, 15–17, 27, 28], and they all follow a similar divisible algorithmic framework as Quick (c.f. Section 3).

A recent work proposed to use machine learning to predict the running time of graph computation for workload partitioning [18], but the graph algorithms considered there are iterative algorithms that do not have unpredictable pruning rules and thus the running time can be easily estimated. This is not the case in quasi-clique mining (c.f. Section 4), and dense subgraph mining in general which adopts divide-and-conquer (and often recursive) algorithms, calling for a new solution for effective task workload partitioning.

3 PRELIMINARIES

Graph Notations. We consider an undirected graph $G = (V, E)$ where V (resp. E) is the set of vertices (resp. edges). The vertex set of a graph G can also be explicitly denoted as $V(G)$. We use $G(S)$ to denote the subgraph of G induced by a vertex set $S \subseteq V$, and use $|S|$ to denote the number of vertices in S. We also abuse the notation and use v to mean the singleton set $\{v\}$. We denote the set of neighbors of a vertex v in G by $N(v)$, and denote the degree of v in G by $d(v) = |N(v)|$. Given a vertex subset $V' \subseteq V$, we define $N_{V'}(v) = \{u \mid (u, v) \in E, u \in V'\}$, i.e., $N_{V'}(v)$ is the set of v's neighbors inside V', and we also define $d_{V'}(v) = |N_{V'}(v)|$.

To illustrate the notations, consider the graph G shown in Figure 1. Let us use v_a to denote Vertex ⓐ (the same for other vertices), thus we have $N(v_d) = \{v_a, v_c, v_e, v_h, v_i\}$ and $d(v_d) = 5$. Also, let $S = \{v_a, v_b, v_c, v_d, v_e\}$, then $G(S)$ is the subgraph of G consisting of the vertices and edges in red and black.

Given two vertices $u, v \in V$, we define $\delta(u, v)$ as the number of edges on the shortest path between u and v. We call G as connected if $\delta(u, v) < \infty$ for any $u, v \in V$. We further define $N_k(v) = \{u \mid \delta(u, v) = k\}$ and define $N_k^+(v) = \{u \mid \delta(u, v) \leq k\}$. In a nutshell, $N_k^+(v)$ are the set of vertices reachable from v within k

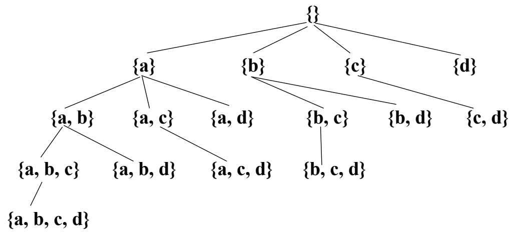

Figure 2: Set-Enumeration Tree

hops, and $N_k(v)$ are the set of vertices reachable from v in k hops but not in $(k-1)$ hops. Then, we have $N_0(v) = v$ and $N_1(v) = N(v)$, and $N_k^+(v) = N_0(v) + N_1(v) + \ldots + N_k(v)$. For 2-hop neighbors, we define $B(v) = N_2(v)$ and $\mathbb{B}(v) = N_2^+(v)$.

To illustrate using Figure 1, we have $N(v_e) = \{v_a, v_b, v_c, v_d\}$, $B(v_e) = \{v_f, v_g, v_h, v_i\}$, and $\mathbb{B}(v_e)$ consisting of all vertices.

Problem Definition. We next formally define our problem.

Definition 1 (γ-quasi-clique). A graph $G = (V, E)$ is a γ-quasi-clique ($0 \leq \gamma \leq 1$) if G is connected, and for every vertex $v \in V$, its degree $d(v) \geq \lceil \gamma \cdot (|V| - 1) \rceil$.

If a graph is a γ-quasi-clique, then its subgraphs usually become uninteresting, so we only mine maximal γ-quasi-clique as follows:

Definition 2 (Maximal γ-quasi-clique). Given graph $G = (V, E)$ and a vertex set $S \subseteq V$, $G(S)$ is a maximal γ-quasi-clique of G if $G(S)$ is a γ-quasi-clique, and there does not exist a superset $S' \supset S$ such that $G(S')$ is a γ-quasi-clique.

To illustrate using Figure 1, consider $S_1 = \{v_a, v_b, v_c, v_d\}$ (i.e., vertices in red) and $S_2 = S_1 \cup v_e$. If we set $\gamma = 0.6$, then both S_1 and S_2 are γ-quasi-cliques: every vertex in S_1 has at least 2 neighbors in $G(S_1)$ among the other 3 vertices (and $2/3 > 0.6$), while every vertex in S_2 has at least 3 neighbors in $G(S_2)$ among the other 4 vertices (and $3/4 > 0.6$). Also, since $S_1 \subset S_2$, $G(S_1)$ is not a maximal γ-quasi-clique.

In the literature of dense subgraph mining, researchers usually only strive to find big dense subgraphs, such as the largest dense subgraph [15, 24, 27, 28], the top-k largest ones [32], and those larger than a predefined size threshold [15, 16, 26]. There are two reasons. (i) Small dense subgraphs are common and thus statistically insignificant and not interesting. For example, a single vertex itself is a quasi-clique for any γ, and so is an edge with its two end-vertices. (ii) The number of dense subgraphs grows exponentially with the graph size and is thus intractable unless we focus on finding large ones. In fact, it has been shown that even the problem of detecting if a given quasi-clique is maximal is NP-hard [32], and clique relaxations (aka. pseudo-cliques) are known to be much more expensive than clique mining [7, 15, 16, 32]. Following [26], we use a minimum size threshold τ_{size} to return only large quasi-cliques.

Definition 3 (Problem Statement). Given a graph $G = (V, E)$, a minimum degree threshold $\gamma \in [0, 1]$ and a minimum size threshold τ_{size}, we aim to find all the vertex sets S such that $G(S)$ is a maximal γ-quasi-cliques of G, and that $|S| \geq \tau_{size}$.

For ease of presentation, when $G(S)$ is a valid quasi-clique, we simply say that S is a valid quasi-clique.

Framework for Recursive Mining. In general pseudo-clique mining problems (including ours), the giant search space of a graph $G = (V, E)$, i.e., V's power set, can be organized as a set-enumeration tree [26]. Figure 2 shows the set-enumeration tree T for a graph G with four vertices $\{a, b, c, d\}$ where $a < b < c < d$ (ordered by ID). Each tree node represents a vertex set S, and only vertices larger than the largest vertex in S are used to extend S. For example, in Figure 2, 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.

Let us denote T_S as the subtree of the set-enumeration tree T rooted at a node with set S. Then, T_S represents a search space for all possible pseudo-cliques that contain all vertices in S. In other words, let Q be a pseudo-clique found by T_S, then $Q \supseteq S$.

We represent the task of mining T_S as a pair $\langle S, ext(S) \rangle$, where S is the set of vertices assumed to be already included, and $ext(S) \subseteq (V - S)$ keeps those vertices that can extend S further into a γ-quasi-clique. As we shall see, many vertices cannot form a γ-quasi-clique together with S and can thus be safely pruned from $ext(S)$; therefore, $ext(S)$ is usually much smaller than $(V - S)$.

Note that the mining of T_S can be recursively decomposed into the mining of the subtrees rooted at the children of node S in T_S, denoted by $S' \supset S$. Note that since $ext(S') \subset ext(S)$, the subgraph induced by nodes of a child task $\langle S', ext(S') \rangle$ is smaller.

This set-enumeration approach typically requires postprocessing to remove non-maximal pseudo-cliques from the set of valid pseudo-cliques found [26]. For example, when processing task that mines $T_{\{b\}}$, vertex a is not considered and thus the task has no way to determine that $\{b, c, d\}$ is not maximal, even if $\{b, c, d\}$ is invalidated by $\{a, b, c, d\}$ which happens to be a valid pseudo-clique, since $\{a, b, c, d\}$ is processed by the task mining $T_{\{a\}}$. But this postprocessing is efficient especially when the number of valid pseudo-cliques is not big (as we only find large pseduo-cliques).

4 CHALLENGES IN LOAD BALANCING

We explain the straggler problem using two large graphs *YouTube* and *Patent* shown in Table 3 of Section 8. We show that (1) the running time of tasks spans a wide range, (2) even tasks with subgraphs of similar size- and degree-related features can have drastically different running time, and hence (3) expensive tasks cannot be effectively predicted using regression models in machine learning.

Specifically, we ran quasi-clique mining using G-thinker where each task is spawned from a vertex v and mines the entire set-enumeration subtree $T_{\{v\}}$ in serial without generating any subtasks. As shall be clear from rules (P1) and (P2) in Section 6, vertices with low degrees can be pruned using a k-core algorithm, and vertices in $ext(S)$ have to be within $f(\gamma)$ hops from v. Our reported experiments has applied these pruning rules so that (i) low-degree vertices are directly pruned without generating tasks, (ii) the subgraphs have been pruned not to include vertices pruned by (P1) and (P2).

Also, we only report the actual time of mining $T_{\{v\}}$ for each task, not including any system-level overheads for task scheduling and vertex data requesting, though the latter cost is not a bottleneck since almost all mining threads in G-thinker are busy on the actual mining workloads when there are enough tasks to process [38].

Table 1 (resp. Table 2) shows the task-subgraph features of the top-10 longest-running tasks on *YouTube* with $\gamma = 0.9$ (resp. *Patent* with $\gamma = 0.89$) including the number of vertices and edges, the maximum and average vertex degree, the k-core number (aka. degeneracy) of the subgraph, the actual serial mining time on the subgraph, along with the predicted time using support vector regression. The tasks are listed in ascending order of running time (c.f. Column "Task Time"), and the time unit is millisecond (ms).

Table 1: Features of 10 Most Expensive Tasks on *YouTube*

|V|	|E|	Max Degree	|E|/|V|	Core #	Task Time	Predicted Time
2,570	72,678	1,583	28.28	43	13,033	899.67
3,588	82,727	1,417	23.06	37	13,407	1,128.13
3,228	100,177	2,127	31.04	49	13,623	1,505.75
2,646	75,747	1,646	28.63	44	13,893	969.04
2,755	78,375	1,597	28.45	45	15,011	1,028.77
5,074	162,249	2,721	31.98	50	15,015	1,924.41
3,177	101,008	1,850	31.80	49	15,267	1,521.73
2,321	55,094	1,320	23.74	38	15,584	529.61
3,723	113,828	1,849	30.58	46	16,881	1,745.78
26,235	694,686	7,105	26.48	51	3,645,905	1,015.08

Table 2: Features of 10 Most Expensive Tasks on *Patent*

|V|	|E|	Max Degree	|E|/|V|	Core #	Task Time	Predicted Time
109	4,232	93	38.83	64	729,769	5.53
93	3,197	80	34.38	60	1,006,208	3.84
104	3,914	88	37.63	64	1,053,326	4.99
95	3,332	82	35.07	60	1,083,755	4.07
69	1,786	65	25.88	43	1,198,085	1.48
78	2,282	69	29.26	48	1,220,241	2.32
72	1,950	66	27.08	45	1,411,622	1.75
79	2,346	69	29.70	49	1,757,738	2.43
88	2,873	75	32.65	55	2,658,704	3.32
76	2,167	68	28.51	47	2,878,700	2.11

In Table 1, the last task takes more than 1 hour (3645.9 seconds) to complete, while the entire G-thinker job only takes 61 minutes and 33.2 seconds, clearly indicating that this task is a straggler. In fact, even if we sum the mining time of all tasks, the total is just 5.5 times that of this straggler task, meaning that the speedup ratio is locked at 5.5× if we do not further decompose an expensive task.

In Table 2, the last 9 tasks all take more than 1000 seconds, so unlike *YouTube* with one particularly expensive tasks, *Patent* has a few of them, so the computing thread that gets assigned most of those tasks will become a straggler. In fact, the job takes only 55 minutes and 25.4 seconds, but the last task alone takes 2878.7 seconds, clearly a straggler. In fact, on both graphs, there are tasks taking less than 1 ms, so the task time spans 8 orders of magnitude!

Note that in the tables, we already have size- and degree-based features of a task-subgraph, as well as the more advanced feature of subgraph degeneracy that reflects graph density. We have extensively tested the various machine learning models for task-time regression using the above input features along with the top-10 highest vertex degrees and top-10 vertex core indices, but none of the models can effectively predict the time-consuming tasks. In both Tables 1 and 2, the last column shows the predicted time using support vector regression trained using all the task statistics, and we can see that the predicted times are way off the ground truth.

We remark that this difficulty is because the set-enumeration search is exponential in nature, and the timing when pruning rules are applicable changes dynamically during the mining depending on the vertex connections, and cannot be effectively predicted other than conducting the actual divisible mining.

To visualize how each subgraph feature impacts the task running time, we plot the impacts of average degree, and core # in the two subplots in Figure 3 for the *YouTube* graph, where we excluded the sole straggler task that takes 3645.9 seconds which would otherwise

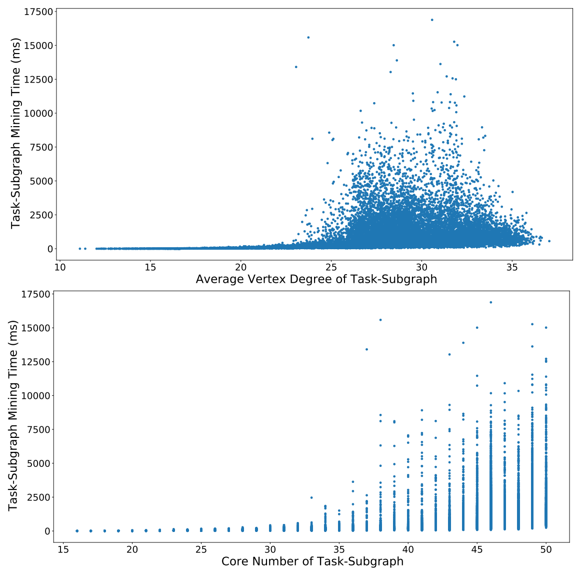

Figure 3: Subgraph Features v.s. Task Time on *YouTube*

flatten other points to near 0 on the y-axis. We can see that for about the same feature values, the time can vary a lot along the vertical direction, and this happens unless the subgraph is very small (e.g., less than 1000 vertices or average degree less than 20). No wonder that the expensive tasks cannot be predicted from these features. The complete plots of all features on both graph data are put in our technical report [19] due to space limitation.

Solution Overview. We address the above challenges from both the algorithmic and the system perspectives. **Algorithmically**, straggler tasks need to be divided into subtasks with controllable running time even though the actual running time needed by a task is difficult to predict; this will be addressed in Section 7. However, even with effective task decomposition algorithms, the **system** still needs to have a mechanism to schedule straggler tasks early so that its workloads can be partitioned and concurrently processed as early as possible; we address this in Section 5 below.

5 G-THINKER AND ITS REDESIGN

G-thinker API. The distributed system G-thinker [38] computes in the unit of tasks. A task t maintains a subgraph g that it constructs and then mines. Each initial task is spawned from an individual vertex v and requests for the adjacency lists of its surrounding vertices (whose IDs are in v's adjacency list). When the one-hop neighbors of v are received, t can continue to grow its subgraph g by requesting the second-hop neighbors. When g is fully constructed, t can then mine it or decompose it to generate smaller tasks.

To avoid double-counting, a vertex v only requests those vertices with ID $> v$. In Figure 2, each level-1 singleton node $\{v\}$ corresponds to a G-thinker task spawned from v, and it only examines those vertices with ID $> v$, so that a quasi-clique whose smallest vertex is v is found exactly in the set-enumeration subtree $T_{\{v\}}$ (recall Figure 2) by the task spawned from v.

To write a G-thinker algorithm, a user only implements 2 user-defined functions (UDFs): (1) *spawn*(v) indicating how to spawn a task from each individual vertex of the input graph; (2) *compute*(t, *frontier*) indicating how a task t processes an iteration where *frontier*

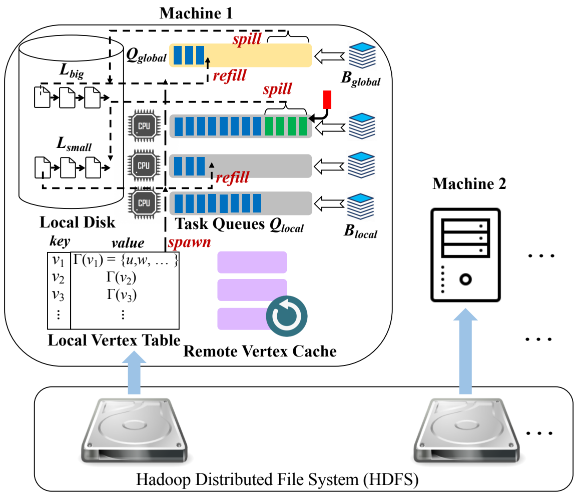

Figure 4: G-thinker Architecture Overview

keeps the adjacency lists of the requested vertices in the previous iteration. In a UDF, users may request for the adjacency list of a vertex u to expand the subgraph g of a task t, or even to decompose g by creating multiple new tasks with smaller subgraphs, which corresponds to branching a node into its children in Figure 2.

UDF *compute*(t, *frontier*) is called in iterations for growing task t's subgraph in a breadth-first manner. If some requested vertices are not locally available, t will be suspended so that its mining thread can continue to process other tasks; t will be scheduled to call *compute*(.) again once all its requested data become locally available.

UDF *compute*(t, *frontier*) returns *true* if the task t needs to call *compute*(.) for more iterations for further processing; it returns *false* if t is finished so that G-thinker will delete t to release space.

In this paper, we maintain G-thinker's programming interface as described above while redesigning its parallel execution engine so that big tasks can be scheduled early to partition its computations.

The Original System Architecture. Figure 4 shows the architecture (components) of G-thinker on a cluster of machines (the yellow global task queue is the new addition by our redesign).

We assume that a graph is stored as a set of vertices, where each vertex v is stored with its adjacency list $N(v)$ that keeps its neighbors. G-thinker loads an input graph from HDFS. As Figure 4 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 constitute a distributed key-value store where any task can request for $N(v)$ using v's ID.

G-thinker spawns initial tasks from each individual vertex v in the local vertex table. As Figure 4 shows, each machine also maintains a remote vertex cache to keep the requested vertices (and their adjacency lists) that are not in the local vertex table, for access by tasks via the input argument *frontier* to UDF *compute*(t, *frontier*). This allows multiple tasks to share requested vertices to minimize redundancy. In *compute*(t, *frontier*), task t is supposed to save the needed vertices and edges in *frontier* into its subgraph, as G-thinker releases t's hold of those vertices in *frontier* right after *compute*(t, *frontier*) returns, and they may be evicted from the vertex cache.

Note that if the machine memory is large enough, a pulled vertex will never be evicted, so every vertex will be pulled at most once.

If *compute*(t, *frontier*) returns *true*, t is added to a task queue to be scheduled to call *compute*(.) for more iterations; while if it returns *false*, t is finished and thus deleted to release space.

In the original G-thinker, each mining thread keeps a task queue Q_{local} of its own to stay busy and to avoid contention. Since tasks are associated with subgraphs that may overlap, it is infeasible to keep all tasks in memory. G-thinker only keeps a pool of active tasks in memory at any time by controlling the pace of task spawning. If a task is waiting for its requested vertices, it is suspended so that the mining thread can continue to process the next task in its queue; the suspended task will be added to a task buffer B_{local} by the data serving module once all its requested vertices become locally available, to be fetched by the mining thread for calling *compute*(.), and adding it to Q_{local} if *compute*(.) returns *true*.

Note that a task queue can become full if a task generates many subtasks into its queue, or if many tasks that are waiting for data become ready all at once. To keep the number of in-memory tasks bounded, if a task queue is full but a new task is to be inserted, we spill a batch of C tasks at the end of the queue as a file to local disk to make room. As the upper-left corner of Figure 4 shows, each machine maintains a list $\mathcal{L}_{small}$ of task files spilled from the task queues of mining threads. To minimize the task volume on disks, when a thread finds that its task queue is about to become empty, it will first refill tasks into the queue from a task file (if it exists), before choosing to spawn more tasks from vertices in local vertex table. Note that tasks are spilled to disks and loaded back in batches to be IO-efficient. For load balancing, machines about to become idle will steal tasks from busy ones by prefetching a batch of tasks and adding them as a file to $\mathcal{L}_{small}$. These tasks will be loaded by a mining thread for processing when its task queue needs a refill.

Note that while we materialize subgraphs for tasks, the above design ensures that only a pool of tasks are in memory and spilled tasks are temporarily kept on local disks. This is important to keep memory usage bounded as the number of tasks can grow exponentially with graph size. Also note that G-thinker is designed to be distributed mainly to use the CPU cores on all machines in a cluster, and therefore the IO and locking operations are well overlapped with and thus hidden by task computations [38].

System Redesign. Recall that a task in pseduo-clique mining can be very time consuming. If we only let each mining thread buffer pending tasks in its own local queue, big tasks in the queue cannot be moved around to idle threads in time until they reach the queue head, and they can be stuck by other time-consuming big tasks located earlier in the queue, causing the straggler problem. We now describe how we redesign the execution engine to allow big tasks to be scheduled as soon as possible, always before small tasks.

We maintain separate task containers for big tasks and small ones, and always prioritize the containers for big tasks for processing. Note that for the new engine to function, we also need our new task decomposition algorithms in Section 7 to ensure that a big task will not be computed for a long time before being decomposed, so that later big tasks can be timely scheduled for processing.

Specifically, we use the local task queues of the respective mining threads and the associated task containers (i.e., file list $\mathcal{L}_{small}$ and

Algorithm 1 Old Execution Procedure of a Computing Thread

```
1: while job end tag is not set by the main thread do
2:     if memory capacity permits then
3:         if $Q_{local}$ does not have enough tasks then refill $Q_{local}$
4:         pop a task $t$ from $Q_{local}$ and provide requested vertices
5:         if all vertices are ready, repeat compute($t$, frontier)
6:         if $t$ is not finished, suspend $t$ to wait for data
7:     obtain a task $t'$ from $B_{local}$
8:     repeat compute($t'$, frontier) till some vertex is not available
9:     if $t'$ is not finished, append $t'$ to $Q_{local}$
```

ready-task buffer B_{local}) to keep small tasks only. We similarly maintain a global task queue Q_{global} to keep big tasks shared by all computing threads, along with its associated task containers as shown in Figure 4, including file list $\mathcal{L}_{big}$ to buffer big tasks spilled from Q_{global}, and task buffer B_{global} to hold those big tasks that have their requested data ready for computation.

We define a user-specified threshold τ_{split} so that if a task $t = \langle S, ext(S)\rangle$ has a subgraph with potentially more than τ_{split} vertices to check, it is appended to Q_{global}; otherwise, it is appended to Q_{local} of the current thread. Here, it is difficult to decide the subgraph size of t as it is changing. 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_{split}, which prioritizes its execution to construct the potentially big subgraph early; while when t is mining its constructed subgraph, we consider t as a big task iff $|ext(S)| > \tau_{split}$, since there are $|ext(S)|$ vertices to check to expand S.

In the original G-thinker, each thread loops two operations:

- Algorithm 1 Lines 4-6 "pop": to fetch a task t from Q_{local} and to feed its requested vertices; if any remote vertex is not in the vertex cache, t will be suspended to wait for data;
- Algorithm 1 Lines 7-9 "push": to fetch a task from the thread's local ready-buffer B_{local} for computation, which is then appended to Q_{local} if further processing is needed.

"Pop" is only done if there is enough space left in vertex cache and task containers, otherwise only "push" is conducted to process partially computed tasks so that their requested vertices can be released to make room, which is necessary to keep tasks flowing.

Task refill is conducted right before "pop" if the number of tasks in Q_{local} < task batch size C, with the priority order of getting a task batch from $\mathcal{L}_{small}$, then from B_{local}, and then spawning from vertices in the local vertex table that have not spawned tasks yet. This order is to digest old/spilled tasks before spawning new tasks.

In our redesigned G-thinker engine, we prioritize big tasks for execution and the procedure in Algorithm 1 has three major changes.

The first change is with "push": a mining thread keeps flowing those tasks that have their requested data ready to compute, 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 either to Q_{global} or the thread's Q_{local}. (ii) If B_{global} is, however, found to be empty, the thread will instead fetch a small task from its B_{local} for computing.

The second change is with "pop": a computing thread always fetches a task from Q_{global} first. If (I) Q_{global} is locked by another thread (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) if Q_{global} is successfully locked, if its number of tasks is below a batch size C, the thread will try to refill a batch of tasks from $\mathcal{L}_{big}$. We do not check B_{global} for refill since it is shared by all mining threads which will incur frequent locking overheads. Note that "push" already keeps flowing big tasks with data ready.

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

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

If both $\mathcal{L}_{small}$ and B_{local} are still empty, the computing thread 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.

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 $\mathcal{L}_{big}$) in each machine is periodically collected by a master (every 1 second), which computes the average and generates stealing plans to make the number of big tasks on every machine close to this 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) at most C tasks are moved at each machine.

6 PROPOSED QUICK+ ALGORITHM

The key to an efficient set-enumeration search is the pruning strategies that are applied to remove entire branches from consideration [23]. Without pruning, the search space is exponential. Quick [26] uses the most complete set of pruning rules for mining maximal quasi-cliques. To further improve the efficiency, this section presents our Quick+ algorithm that integrates Quick with new pruning rules. We also fix some missed boundary cases.

Due to space limit, we only briefly overview Quick+ and present those rules necessary for understanding our improvements, and leave the complete details and proofs in our technical report [19].

Recall the set-enumeration tree in Figure 2, where each node represents a mining task $t_S = \langle S, ext(S)\rangle$ which mines the set-enumeration subtree T_S: it assumes that vertices in S are already included in a result quasi-clique to find, and continues to expand $G(S)$ with vertices of $ext(S)$ into a valid quasi-clique. Task t_S can be recursively decomposed into tasks mining the subtrees $\{T_{S'}\}$ where $S' \supset S$ are child nodes of node S. Quick+ examines the set-enumeration search tree in depth-first order, while the parallel algorithm in the next section will utilize the concurrency among child nodes $\{S'\}$ of node S in the set-enumeration tree.

We consider two types of pruning rules: **Type I: Pruning ext(S):** in such a rule, if a vertex $u \in ext(S)$ satisfies certain conditions, u can be pruned from $ext(S)$ since there must not exist a vertex set S' such that $(S \cup u) \subseteq S' \subseteq (S \cup ext(S))$ and $G(S')$ is a γ-quasi-clique. **Type II: Pruning S:** in such a rule, if a vertex $v \in S$ satisfies certain conditions, there must not exist a vertex set S' such that $S \subseteq S' \subseteq (S \cup ext(S))$ and $G(S')$ is a γ-quasi-clique, and thus there is no need to extend S further. Type-II pruning invalidates the entire T_S. A variant invalidates $G(S')$, $S \subset S' \subseteq (S \cup ext(S))$ from being a valid quasi-clique, but node S is not pruned (i.e., $G(S)$ may be a valid quasi-clique).

We identify 7 groups of pruning rules [19] summarized as (P1)–(P7) where each rule either belongs to Type I, or Type II, or both.

(P1) Graph-Diameter Based Pruning. Theorem 1 of [31] defines the upper bound of the diameter of a γ-quasi-clique as a function $f(\gamma)$, and $f(\gamma) = 2$ if $\gamma \geq 0.5$. W.l.o.g, we use 2 as the diameter upper bound in our algorithm description, but it is straightforward to generalize to the case $\gamma < 0.5$ by considering vertices $f(\gamma)$ hops away. Since a vertex $u \in ext(S)$ must be within 2 hops from any $v \in S$, we have $ext(S) \subseteq \bigcap_{v \in S} \mathbb{B}(v)$. This is a Type-I pruning rule since if $u \notin \bigcap_{v \in S} \mathbb{B}(v)$, u can be pruned from $ext(S)$.

(P2) Size-Threshold Based Pruning. A valid γ-quasi-clique $Q \subseteq V$ should contain at least τ_{size} vertices (i.e., $|Q| \geq \tau_{size}$), and therefore for any $v \in Q$, its degree $d(v) \geq \lceil \gamma \cdot (|Q|-1) \rceil \geq \lceil \gamma \cdot (\tau_{size}-1) \rceil$. We thus can prune any vertex u with $d(u) < \lceil \gamma \cdot (\tau_{size}-1) \rceil$ from G. Let $k = \lceil \gamma \cdot (\tau_{size}-1) \rceil$, then this rule shrinks G into its k-core, i.e., the maximal subgraph of G where every vertex has degree $\geq k$. The k-core of a graph $G = (V, E)$ can be computed in $O(|E|)$ time using a peeling algorithm [6]. We thus shrink a graph G into its k-core before mining, which effectively reduces the search space.

(P3) Degree-Based Pruning. Four kinds of degrees are frequently used by pruning rules: (1) SS-degrees: $d_S(v)$ for all $v \in S$; (2) SE-degrees: $d_S(u)$ for all $u \in ext(S)$; (3) ES-degrees: $d_{ext(S)}(v)$ for all $v \in S$; and (4) EE-degrees: $d_{ext(S)}(u)$ for all $u \in ext(S)$.

Three groups of pruning rules utilize these degrees: (i) degree-based pruning that solely uses the degrees of a vertex itself, (ii) upper-bound based pruning and (iii) lower-bound based pruning that look at the degrees of multiple (or even all) vertices in S and $ext(S)$. Each of the three groups contains two rules: one Type-I rule and one Type-II rule. Please see our technical report [19] for these rules.

(P4 & P5) Upper- and Lower-Bound Based Pruning. For each task $t_S = (S, ext(S))$, Quick [26] defined an upper bound U_S (resp. lower bound L_S) on the number of vertices in $ext(S)$ to be added to S to form a γ-quasi-clique, based on the above mentioned degrees. We found that additional Type-II pruning not considered by Quick can happen when computing U_S (resp. L_S) (see [19] for details).

(P6) Critical-Vertex Based Pruning. Given the above-mentioned lower bound L_S, we call any vertex $v \in S$ as a critical vertex if $d_S(v) + d_{ext(S)}(v) = \lceil \gamma \cdot (|S| + L_S - 1) \rceil$. Quick [26] showed that if v is a critical vertex, then for any vertex set S' such that $S \subset S' \subseteq (S \cup ext(S))$, if $G(S')$ is a γ-quasi-clique, then S' must contain every neighbor of v in $ext(S)$, i.e., $N_{ext(S)}(v) \subseteq S'$. In other words, if we find any $v \in S$ is a critical vertex, we can directly include all vertices in $N_{ext(S)}(v)$ to S for further mining.

(P7) Cover-Vertex Based Pruning. Given a vertex $u \in ext(S)$, Quick [26] defined a vertex set $C_S(u) \subseteq ext(S)$ such that for any γ-quasi-clique Q generated by extending S with vertices in $C_S(u)$, $Q \cup u$ is also a γ-quasi-clique. In other words, Q is not maximal and can thus be pruned. We say that $C_S(u)$ is the set of vertices in $ext(S)$ that are covered by u, and that u is the cover vertex.

To utilize $C_S(u)$ for pruning, we put vertices of $C_S(u)$ after all the other vertices in $ext(S)$ when checking the next level in the set-enumeration tree (see Figure 2), and only check until vertices of $ext(S) - C_S(u)$ are examined (i.e., the extension of S using $V' \subseteq C_S(u)$ is pruned). To maximize the pruning effectiveness, we find $u \in ext(S)$ to maximize $|C_S(u)|$. We refer readers to [19] for the formula of computing $C_S(u)$. As a degenerate special case, initially when $S = \emptyset$, we have $C_S(u) = N(u)$, i.e., we only need to find u as the vertex with the maximum degree. Note that for any γ-quasi-clique Q constructed out of vertices in $C_S(u) = N(u)$, adding u to Q still produces a γ-quasi-clique. We find u as the vertex the maximum degree after k-core pruning by (P2) to avoid high-degree vertex without pruning power (e.g., the center of a sparse star graph).

Iterative Nature of Type-I Pruning. Recall that Type-I pruning shrinks $ext(S)$, which will reduce vertex degrees such as $d_{ext(S)}(v)$ of some $v \in S$, which will in turn update bounds U_S and L_S that are defined on the degrees. This essentially means that Type-I pruning is iterative: each pruned vertex $u \in ext(S)$ may change degrees and bounds, which affects the various pruning rules including Type-I pruning rules themselves; these Type-I pruning rules will thus be checked again and new vertices in $ext(S)$ may be pruned due to Type-I pruning, triggering another iteration.

As this process is repeated, U_S and L_S become tighter until no more vertex can be pruned from $ext(S)$, which consists of 2 cases:

- C1: $ext(S)$ becomes empty. In this case, we only need to check if $G(S)$ is a valid quasi-clique;
- C2: $ext(S)$ is not empty but cannot be shrunk further by pruning rules. Then, we need to check S and its extensions.

The Iterative Pruning Sub-procedure. Algorithm 2 shows how to apply our pruning rules to (1) shrink $ext(S)$ and to (2) determine if S can be further extended to form a valid quasi-clique. This is a pruning sub-procedure of our recursive mining algorithm Quick+.

The iterative pruning caused by Type-I rules that shrink $ext(S)$ is given by the loop of Lines 1–15, which ends if the condition in Line 15 is met, which corresponds to the above 2 cases C1 and C2.

Algorithm 2 returns a boolean tag indicating whether S's extensions (but not S itself) are pruned, and the input $ext(S)$ is passed as a reference and may be shrunk by Type-I pruning by the function.

Since $ext(S)$ can be pruned to become empty, we design this pruning sub-procedure to guarantee that it returns $false$ only if $ext(S) \neq \emptyset$. Therefore, if the loop of Lines 1–15 exits due to $ext(S) = \emptyset$, we have to return $true$ (Line 19) as there is no vertex to extend S, but we need to first examine if $G(S)$ itself is a valid quasi-clique in Lines 17–18. Note that $G(S)$ is not Type-II pruned as otherwise Line 10 would have returned $true$ to exit pruning.

Now let us focus on the loop body in Lines 2–14 about one pruning iteration, which can be divided into 3 parts: (1) Lines 2–7: critical vertex pruning, (2) Lines 8–10: Type-II pruning, and (3) Lines 11–14: Type-I pruning. To keep Algorithm 2 short, we omit some details but they are described in the narrative below.

First, consider Part 1. We compute the degrees in Line 2, which are then used to compute U_S and L_S in Line 3. In Line 3, recall from (P4 & P5) that Type-II pruning may apply when computing U_S and L_S, in which case we return $true$ to prune S's extensions.

Then, Lines 4–6 apply the critical-vertex pruning of (P6). In Quick, each iteration only finds one critical vertex and moves its neighbors from $ext(S)$ to S. We propose to find all critical vertices in S and move their neighbors from $ext(S)$ to S. Such movement will update degrees and bounds which are then updated in Line 7 if a critical vertex is ever found. Similar to Line 3, Line 7 may trigger

Algorithm 2 Iterative Bound-Based Pruning

```
Function: iterative_bounding($S$, $ext(S)$, $\gamma$, $\tau_{size}$)
Output: $true$ iff the case of extending $S$ (excluding $S$ itself) is pruned;
$ext(S)$ is passed as a reference; elements may be pruned after return
 1: repeat
 2:     Compute $d_S(v)$ and $d_{ext(S)}(v)$ for all $v$ in $S$ and $ext(S)$
 3:     Compute upper bound $U_S$ and lower bound $L_S$
 4:     if $\forall$ $v \in S$ that is a critical vertex then
 5:         $I \leftarrow ext(S) \cap N(v)$
 6:         $S \leftarrow S \cup I,\ ext(S) \leftarrow ext(S) - I$
 7:         Update degree values, $U_S$ and $L_S$
 8:     for each vertex $v \in S$ do
 9:         Check Type-II pruning conditions of (P3), (P4) and (P5)
10:         return $true$ if Type-II pruning applies
11:     for each vertex $u \in ext(S)$ do
12:         Check Type-I pruning conditions of (P3), (P4) and (P5)
13:         if some Type-I pruning condition holds for $u$ then
14:             $ext(S) \leftarrow ext(S) - u$
15: until $ext(S) = \emptyset$ or no vertex in $ext(S)$ was Type-I pruned
16: if $ext(S) = \emptyset$ then
17:     if $|S| \geq \tau_{size}$ and $G(S)$ is a $\gamma$-quasi-clique then
18:         Append $S$ to the result file
19:     return $true$
20: return $false$
```

Type-II pruning so that the function returns *true* directly. Since the updates may generate new critical vertices in the updated S, we actually loop Lines 4–7 until there is no more critical vertex in S.

Recall from (P6) that S' in critical-vertex pruning does not include S itself, so it is possible that all extensions of S lead to no valid quasi-cliques, making $G(S)$ itself a maximal quasi-clique. Quick misses this check. To consider this case, we actually first check $G(S)$ as in Lines 17–18 before expanding S (i.e., Line 6). While our algorithm may output S while $G(S)$ is not maximal, the postprocessing step will remove non-maximal quasi-cliques.

Moreover, in actual implementation, if $ext(S)$ is found to be empty after critical vertex pruning (Line 6), we directly exit the loop of Lines 1–15 to skip the execution of Lines 7–14.

Next, consider Part 2 on Type-II pruning as in Lines 8–10. These lines assume that Type-II pruning rules prune the entire subtree T_S if any vertex $v \in S$ satisfies the rule conditions. There exists a Type-II rule of (P3) where we can only prune extensions of S but not S itself, but it is not utilized by Quick. Quick+ properly utilizes it [19]: only when all other Type-II pruning conditions fail to prune T_S will we consider S as a candidate for a valid quasi-clique (checked as in Lines 17–18) and check this special rule that prunes all extensions of S (i.e., to return *true* if conditions are met).

Finally, Part 3 on Type-I pruning checks every prune $u \in ext(S)$ and tries to u using a Type-I pruning condition as shown in Lines 11–14, which may create new pruning opportunities for next iteration.

To summarize, Quick+ improves Quick in 3 aspects. (1) In Quick, each iteration only finds one critical vertex and moves its neighbors from $ext(S)$ to S, while we find all critical vertices to move their neighbors to S. (2) Type-II pruning may apply when computing U_S and L_S (c.f., (P4 & P5)), and Quick+ handles these boundary cases and returns *true* to prune S's extensions. (3) in both critical vertex pruning and degree-based Type-II pruning, $G(S)$ itself should not be pruned which is properly handled by Quick+, but not Quick.

Algorithm 3 Mining Valid Quasi-Cliques Extended from S

```
Function: recursive_mine($S$, $ext(S)$, $\gamma$, $\tau_{size}$)
Output: $true$ iff a valid quasi-clique $Q \supset S$ is found
 1: $\mathcal{T}_{Q\_found} \leftarrow false$
 2: Find cover vertex $u \in ext(S)$ with the largest $C_S(u)$
 3: {If not found, $C_S(u) \leftarrow \emptyset$}
 4: Move vertices of $C_S(u)$ to the tail of the vertex list of $ext(S)$
 5: for each vertex $v$ in the sub-list $(ext(S) - C_S(u))$ do
 6:     if $|S| + |ext(S)| < \tau_{size}$ then
 7:         return $\mathcal{T}_{Q\_found}$
 8:     if $G(S \cup ext(S))$ is a $\gamma$-quasi-clique then
 9:         Append $S \cup ext(S)$ to the result file
10:         return $true$
11:     $S' \leftarrow S \cup v,\ ext(S) \leftarrow ext(S) - v$
12:     $ext(S') \leftarrow ext(S) \cap \mathbb{B}(v)$
13:     if $ext(S') = \emptyset$ then
14:         if $|S'| \geq \tau_{size}$ and $G(S')$ is a $\gamma$-quasi-clique then
15:             $\mathcal{T}_{Q\_found} \leftarrow true$
16:             Append $S'$ to the result file
17:     else
18:         $\mathcal{T}_{pruned} \leftarrow$ iterative_bounding($S'$, $ext(S')$, $\gamma$, $\tau_{size}$)
19:         {here, $ext(S')$ is Type-I-pruned and $ext(S') \neq \emptyset$}
20:         if $\mathcal{T}_{pruned} = false$ and $|S'| + |ext(S')| \geq \tau_{size}$ then
21:             $\mathcal{T}_{found} \leftarrow$ recursive_mine($S'$, $ext(S')$, $\gamma$, $\tau_{size}$)
22:             $\mathcal{T}_{Q\_found} \leftarrow \mathcal{T}_{Q\_found}$ or $\mathcal{T}_{found}$
23:             if $\mathcal{T}_{found} = false$ and $|S'| \geq \tau_{size}$ and $G(S')$ is a
                $\gamma$-quasi-clique then
24:                 $\mathcal{T}_{Q\_found} \leftarrow true$
25:                 Append $S'$ to the result file
26: return $\mathcal{T}_{Q\_found}$
```

The Recursive Main Algorithm. Algorithm 3 shows our Quick+ main algorithm for mining valid quasi-cliques extended from S (including $G(S)$ itself). This algorithm is recursive (see Line 21) and starts by calling *recursive_mine*(v, $\mathbb{B}_{>v}(v)$, γ, τ_{size}) on every $v \in V$ where $\mathbb{B}_{>v}(v)$ denotes those vertices in $\mathbb{B}(v)$ (i.e., within 2 hops from v) whose IDs $> v$, as according to Figure 2, we should not consider the other vertices in $\mathbb{B}(v)$ to avoid double counting.

Recall from (P7) that we have a degenerate cover-vertex pruning method that finds the vertex v_{max} with the maximum degree, so that any quasi-clique generated from only v_{max}'s neighbors cannot be maximal (as it can be extended with v_{max}). To utilize this pruning rule, we need to recode the vertex IDs so that v_{max} has ID 0, while vertices of $N(v_{max})$ have larger IDs than all other vertices, i.e., they are listed at the end in the first level of the set-enumeration tree illustrated in Figure 2 (as they only extend with vertices in $N(v_{max})$). If we enable ID recoding, *recursive_mine*(v, $\mathbb{B}_{>v}(v)$, γ, τ_{size}) only needs to be called on every $v \in V - N(v_{max})$.

Algorithm 3 keeps a boolean tag $\mathcal{T}_{Q\_found}$ to return (see Line 26), which indicates whether some valid quasi-clique Q extended from S (but $Q \neq S$) is found. Line 1 initializes $\mathcal{T}_{Q\_found}$ as $false$, but it will be set as *true* if any valid quasi-clique Q is later found.

Algorithm 3 examines S, and it decomposes this problem into the sub-problems of examining $S' = S \cup v$ for all $v \in ext(S)$, as described by the loop in Line 5. Before the loop, we first apply the cover-vertex pruning of (P7) in Lines 2–4 to compute a cover set $C_S(u)$ so that those vertices in $C_S(u)$ can be skipped in Line 5. Note that we move $C_S(u)$ to the tail of the vertex list of $ext(S)$ (Line 4), and since Line 11 excludes an already examined v from

$ext(S)$, the loop in Line 5 with v scanning $C_S(u)$ corresponds to the case of extending S' using $ext(S') \subseteq ext(S) \subseteq C_S(u)$ (see Lines 11-12) which should be pruned. If we cannot find a cover vertex (see Line 3), then Line 5 iterates over all vertices of $ext(S)$.

Now consider the loop body of Lines 6–25. Line 6 first checks if S, when extended with every vertex in $ext(S)$, can generate a subgraph larger than τ_{size}; if so, the current and future iterations (where $ext(S)$ further shrinks) cannot generate a valid quasi-clique and are thus pruned, and Line 7 directly returns $\mathcal{T}_{Q\_found}$ which indicates if a valid quasi-clique is found by previous iterations.

For a vertex $v \in ext(S)$, the current iteration creates $S' = S \cup v$ for examination in Line 11. Before that, Lines 8–10 first checks if S extended with the entire current $ext(S)$ creates a valid quasi-clique; if so, this is a maximal one and is thus output in Line 9, and further examination can be skipped (Line 10). This pruning is called the look-ahead technique in Quick [26]. Note that $G(S \cup ext(S))$ must satisfy the size threshold requirement since Line 6 is passed, and thus Line 8 does not need to check that condition again.

If the look-ahead technique does not prune the search, then Line 11 creates $S' = S \cup v$, and excludes v from $ext(S)$. The latter also has a side effect of excluding v from $ext(S)$ of all subsequent iterations, which matches exactly how the set-enumeration tree illustrated in Figure 2 avoids generating redundant nodes.

Then, Line 12 shrinks $ext(S)$ into $ext(S')$ by ruling out vertices that are more than 2 hops away from v according to (P1) diameter-based pruning, which is then used to extend S'. If $ext(S') = \emptyset$ after shrinking, then S' has nothing to extend, but $G(S')$ itself may still be a valid quasi-clique and is thus examined in Lines 14–16. We remark that Quick misses this check and may miss results [26].

If $ext(S') \neq \emptyset$, Line 18 then calls Algorithm 2 to apply the pruning rules. Recall that the function either returns $\mathcal{T}_{pruned} = false$ indicating that we need to further extend S' using its shrunk $ext(S')$; or it returns $\mathcal{T}_{pruned} = true$ to indicate that the extensions of S' should be pruned, which will also output $G(S')$ if it is a valid quasi-clique (see Lines 16–19 in Algorithm 2).

If Line 18 decides that S' can be further extended (i.e., $\mathcal{T}_{pruned} = false$) and extending S' with all vertices in $ext(S')$ still has the hope of generating a subgraph with τ_{size} vertices or larger (Line 20), we then recursively call our algorithm to examine S' in Line 21, which returns $\mathcal{T}_{found}$ indicating if some valid maximal quasi-cliques $Q \supset S'$ are found (and output). If $\mathcal{T}_{found} = true$, Line 22 will update the return value $\mathcal{T}_{Q\_found}$ as $true$, but $G(S')$ is not maximal. Otherwise (i.e., $\mathcal{T}_{found} = false$), $G(S')$ is a candidate for a valid maximal quasi-clique and is thus examined in Lines 23–25.

Finally, as in Quick, Quick+ also requires a postprocessing step to remove non-maximal quasi-cliques from the results of Algorithm 3. Also, we only run Quick+ after the input graph is shrunk by the k-core pruning of (P2). To summarize, besides Quick's cover vertex pruning, Quick+ also supports a top-level degenerate pruning using v_{max} as mentioned in (P7), and checks if $G(S')$ is a valid quasi-clique when $ext(S')$ becomes empty after the diameter-based pruning of (P1). Quick misses this check and may miss results.

Additionally, we find that the vertex order in $ext(S)$ matters (Algorithm 3 Line 7) and can significantly impact the running time. To maximize the success probability of the lookahead technique in Lines 8–10 of Algorithm 3 that effectively prunes the entire T_S, we propose to sort the vertices in $ext(S)$ in ascending order of $d_S(v)$ (tie broken by $d_{ext(S)}(v)$) following [23] so that high-degree vertices tend to appear in $ext(S)$ of more set-enumeration tree nodes.

7 PARALLELIZATION ON G-THINKER

Divide-and-Conquer Algorithm. We next adapt Algorithm 3 to run on our redesigned G-thinker, where a big task is divided into smaller subtasks for concurrent processing. Recall that users write a G-thinker program by implementing two UDFs, one for spawning a task from each vertex v, and another, *compute*(t, *frontier*), to pull vertices within two hops (or $f(\gamma)$ hops in general) from v to construct v's two-hop ego-network g from $\mathbb{B}(v)$, and then mines g.

Here, if a task $t = \langle S, ext(S)\rangle$ is spawned from a vertex v, we only pull vertices with ID $> v$ into g to be added to S and $ext(S)$, which avoids redundancy (recall Figure 2). Moreover, if we would like to use the initial degenerate cover-vertex pruning described in (P7), we need to recode the vertex IDs. Specifically, we load the ID and degree (or, $|N(v)|$) into memory, find v_{max} and assign it ID 0, and assign vertices in $N(v)$ IDs $(|V| - |N(v)|), \cdots, (|V| - 2), (|V| - 1)$; for the other vertices, we sort them in ascending order of degree and assign IDs $1, 2, \cdots, (|V| - |N(v)| - 1)$ to allow effective look-ahead pruning. We can then use the old-to-new ID mapping table to recode the IDs in the adjacency lists.

In UDF *task_spawn*(v), we only spawn a task for a vertex v if its degree $\geq k$ and $v_{max} \notin N(v)$, and pull the adjacency list of v's neighbors. Then UDF *compute*$(t, frontier)$ runs 3 iterations. The first iteration uses the pulled first-hop neighbors to construct v's one-hop ego-network $t.g$, and continues to pull the second-hop neighbors tracked by the first-hop neighbors' adjacency lists. Then, the second iteration uses the pulled second-hop neighbors to expand $t.g$ into v's two-hop ego-network. Since t does not need to pull any more vertices, t will not be suspended but rather run the third iteration immediately, which mines quasi-cliques from $t.g$ using Quick+. While pulling vertices, we effectively skip those vertices whose degree cannot be at least k, and due to space limit, we refer interested readers to our technical report [19] for the details.

Now that $t.g$ contains the k-core of the spawning vertex's 2-hop ego-network, Algorithm 4 describes the computation in Iteration 3 which mines quasi-cliques from $t.g$. Since the task can be prohibitive when $t.g$ and $ext(S)$ are big, we only directly process the task using Algorithm 3 when $|ext(S)|$ is small enough (Lines 1–2); otherwise, we divide it into smaller subtasks to be scheduled for further processing (Lines 3–23) (execution is similar to Algorithm 3).

Recall that Algorithm 3 is recursive where Line 21 extends S with another vertex $v \in ext(S)$ for recursive processing, and here we will instead create a new task t' with $t'.S = t.S \cup v$ (Lines 12–13). However, we still want to apply all our pruning rules to see if t' can be pruned first; if not, we will add t' to the system (Line 21) with $t'.iteration = 3$ so that when t' is scheduled for processing, it will directly enter iteration_3(t'). Here, we shrink t''s subgraph to be induced by $t'.S \cup t'.ext(S)$ so that the subtask is on a smaller graph, and since $t'.ext(S)$ shrinks (due to pruning) at each recursion and $t'.g$ also shrinks, the computation cost becomes smaller.

Another difference is with Line 23 of Algorithm 3, where we only check if $G(S')$ is a valid quasi-clique when $\mathcal{T}_{found} = false$, i.e., the recursive call in Line 21 verifies that S' fails to be extended to produce a valid quasi-clique. In Algorithm 4, however, the recursive

Algorithm 4 iteration_3(t)

```
1:  if $|t.ext(S)| \le \tau_{split}$ then
2:      $recursive\_mine(t.S, t.ext(S), \gamma, \tau_{size})$
3:  else
4:      Find cover vertex $u \in t.ext(S)$ with the largest $C_S(u)$
5:      {If not found, $C_S(u) \leftarrow \emptyset$}
6:      Move vertices of $C_S(u)$ to the tail of vertex list $t.ext(S)$
7:      for each vertex $v$ in the sub-list $(t.ext(S) - C_S(u))$ do
8:          if $|t.S| + |t.ext(S)| < \tau_{size}$ then    return $false$
9:          if $G(t.S \cup t.ext(S))$ is a $\gamma$-quasi-clique then
10:             Append $t.S \cup t.ext(S)$ to the result file
11:             return $false$
12:         Create a task $t'$
13:         $t'.S \leftarrow t.S \cup v$,  $t.ext(S) \leftarrow t.ext(S) - v$
14:         $t'.ext(S) \leftarrow t.ext(S) \cap \mathbb{B}(v)$
15:         if $|t'.S| \ge \tau_{size}$ and $G(t'.S)$ is a $\gamma$-quasi-clique then
16:             Append $t'.S$ to the result file
17:         $\mathcal{T}_{pruned} \leftarrow iterative\_bounding(t'.S, t'.ext(S), \gamma, \tau_{size})$
18:         if $\mathcal{T}_{pruned} = false$ and $|t'.S| + |t'.ext(S)| \ge \tau_{size}$ then
19:             $t'.g \leftarrow$ subgraph of $t.g$ induced by $t'.S \cup t'.ext(S)$
20:             $t'.iteration \leftarrow 3$
21:             $add\_task(t')$
22:         else
23:             Delete $t'$
24: return $false$    {task is done}
```

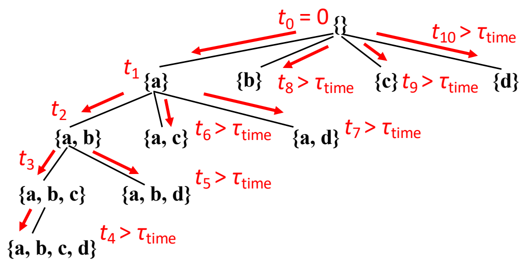

Figure 5: Timeout-Based Divide and Conquer

call now becomes an independent task t' in Line 12, and the current task t has no clue of its results. Therefore, we check if $G(t'.S)$ is a valid quasi-clique right away in Line 15 in order to not miss it. A subtask may later find a larger quasi-clique containing $t'.S$, rendering $G(t'.S)$ not maximal, and we resort to the postprocessing phase to remove non-maximal quasi-cliques.

Due to cover-vertex pruning, a task t can generate at most $|t.ext(S) - C_S(u)|$ subtasks (see Line 7) where u is the cover vertex.

Timeout-Based Task Decomposition. So far, we decompose a task $\langle S, ext(S)\rangle$ as long as $|ext(S)| > \tau_{split}$ but due to the large time variance caused by the many pruning rules, some of those tasks might not be worth splitting as they are fast to compute, while others might not be sufficiently decomposed and need an even smaller τ_{split}. We, therefore, improve our UDF *compute*(t, *frontier*) further by a timeout strategy where we guarantee that each task spends at least a duration of τ_{time} on the actual mining of its subgraph by backtracking (which does not materialize subgraphs) before dividing the remaining workloads into subtasks (which needs to materialize their subgraphs). Figure 5 illustrates how our algorithm works. The algorithm recursively expands the set-enumeration tree in depth-first order, processing 3 tasks until entering $\{a, b, c, d\}$ for which we find the entry time t_4 times out; we then wrap $\{a, b, c, d\}$

Algorithm 5 iteration_3(t) with the Timeout Strategy

```
1: $time\_delayed(t.S, t.ext(S), initial\_time)$
2: return $false$    {task is done}
```

Algorithm 6 *time_delayed*(S, *ext*(S), *initial_time*)

```
1:  $\mathcal{T}_{Q\_found} \leftarrow false$
2:  Find cover vertex $u \in ext(S)$ with the largest $C_S(u)$
3:  {If not found, $C_S(u) \leftarrow \emptyset$}
4:  Move vertices of $C_S(u)$ to the tail of the vertex list of $ext(S)$
5:  for each vertex $v$ in the sub-list $(ext(S) - C_S(u))$ do
6:      if $|S| + |ext(S)| < \tau_{size}$ then:    return $false$
7:      if $G(S \cup ext(S))$ is a $\gamma$-quasi-clique then
8:          Append $S \cup ext(S)$ to the result file;    return $false$
9:      $S' \leftarrow S \cup v$,  $ext(S) \leftarrow ext(S) - v$
10:     $ext(S') \leftarrow ext(S) \cap \mathbb{B}(v)$
11:     if $ext(S') = \emptyset$ then
12:         if $|S'| \ge \tau_{size}$ and $G(S')$ is a $\gamma$-quasi-clique then
13:             $\mathcal{T}_{Q\_found} \leftarrow true$
14:             Append $S'$ to the result file
15:     else
16:         $\mathcal{T}_{pruned} \leftarrow iterative\_bounding(S', ext(S'), \gamma, \tau_{size})$
17:         {here, $ext(S')$ is Type-I-pruned and $ext(S') \ne \emptyset$}
18:         if $current\_time - initial\_time > \tau_{time}$ then
19:             if $\mathcal{T}_{pruned} = false$ and $|S'| + |ext(S')| \ge \tau_{size}$ then
20:                 Create a task $t'$;  $t'.S \leftarrow S'$
21:                 $t'.ext(S) \leftarrow ext(S')$;  $t'.iteration \leftarrow 3$
22:                 $add\_task(t')$
23:             if $|t'.S| \ge \tau_{size}$ and $G(t'.S)$ is a $\gamma$-quasi-clique then
24:                 Append $t'.S$ to the result file
25:         else if $\mathcal{T}_{pruned} = false$ and $|S'| + |ext(S')| \ge \tau_{size}$ then
26:             $\mathcal{T}_{found} \leftarrow time\_delayed(S', ext(S'), initial\_time)$
27:             $\mathcal{T}_{Q\_found} \leftarrow \mathcal{T}_{Q\_found}$ or $\mathcal{T}_{found}$
28:             if $\mathcal{T}_{found} = false$ and $|S'| \ge \tau_{size}$ and $G(S')$ is a $\gamma$-quasi-
                clique then
29:                 $\mathcal{T}_{Q\_found} \leftarrow true$
30:                 Append $S'$ to the result file
31: return $\mathcal{T}_{Q\_found}$
```

as a subtask to be added to our system, and backtrack the upper-level nodes to also add them as subtasks (due to timeout). Note that subtasks are at different granularity and not over-decomposed.

With the timeout strategy, the third iteration of our UDF *compute*(t, *frontier*) is given by Algorithm 5. Line 1 calls our recursive backtracking function *time_delayed*(S, *ext*(S), *inital_time*) detailed in Algorithm 6, where *inital_time* is the time when Iteration 3 begins. Line 2 then returns $false$ to terminate this task.

Algorithm 6 now considers 2 cases. (1) Lines 18–24: if timeout happens, we wrap $\langle S', ext(S')\rangle$ into a task t' to be added for processing just like in Algorithm 4, and since the current task cannot track whether t' will find a valid quasi-clique that extends S', we have to check if $G(S')$ itself is a valid quasi-clique (Lines 23–24) in order not to miss it if it is maximal. (2) Lines 25–30: we perform regular backtracking just like in Algorithm 3, where we recursively call *time_delayed*(.) to process $\langle S', ext(S')\rangle$ in Line 26.

8 EXPERIMENTS

This section reports our experiments. We have released the code of our redesigned G-thinker and quasi-clique algorithms on GitHub [3].

Table 3: Graph Datasets

Data	$\|V\|$	$\|E\|$	$\|E\|/\|V\|$	Max Degree	URL
CX_GSE1730	998	5,096	5.11	197	https://www.ncbi.nlm.nih.gov/geo/query/acc.cgi?acc=GSE1730
CX_GSE10158	1,621	7,079	4.37	110	https://www.ncbi.nlm.nih.gov/geo/query/acc.cgi?acc=GSE10158
Ca-GrQc	5,242	14,496	2.77	81	https://snap.stanford.edu/data/ca-GrQc.html
Enron	36,692	183,831	5.01	1,383	https://snap.stanford.edu/data/email-Enron.html
Amazon	334,863	925,872	2.76	549	https://snap.stanford.edu/data/com-Amazon.html
Hyves	1,402,673	2,777,419	1.98	31,883	http://konect.cc/networks/hyves/
YouTube	1,134,890	2,987,624	2.63	28,754	https://snap.stanford.edu/data/com-Youtube.html
Patent	3,774,768	16,518,947	4.38	793	https://snap.stanford.edu/data/cit-Patents.html
kmer	67,716,231	69,389,281	1.02	35	https://graphchallenge.s3.amazonaws.com/synthetic/gc6/U1a.tsv
USA Road	23,947,347	28,854,312	1.20	9	http://users.diag.uniroma1.it/challenge9/download.shtml

(a) Statistics of graph datasets

Data	τ_{size}	γ	k	$\|V\|$	$\|E\|$	$\|E\|/\|V\|$	Max Degree
CX_GSE1730	30	0.9	27	55	1,168	21.24	54
CX_GSE10158	29	0.8	23	45	784	17.42	44
Ca-GrQc	10	0.8	8	405	4,674	11.54	64
Enron	23	0.9	20	2,276	68,430	30.06	623
Amazon	12	0.5	6	173	667	3.86	13
Hyves	22	0.9	19	2,700	72,242	26.75	586
YouTube	18	0.9	16	26,901	707,751	26.31	7,109
Patent	20	0.9	18	19,078	359,840	18.86	471
kmer	10	0.5	5	55	177	3.22	12
USA Road	7	0.5	3	1,937	3,107	1.60	8

(b) Default parameters and graph statistics after k-core pruning

Table 4: Illustration of the Effect of (γ, τ_{size})

τ_{size}	γ	Time (sec)	#{Results}	#{Maximal}
	0.91	41.48	0	0
20	0.9	911.06	256	256
	0.89	3,386.37	44,083,840	44,080,758

(a) Effect of γ on *Patent*

τ_{size}	γ	Time (sec)	#{Results}	#{Maximal}
23		7.43	161	114
22	0.9	7.45	2,349	1,480
21		8.30	20,662	11,087

(b) Effect of τ_{min} on *Hyves*

Datasets. We used 10 real graph datasets as Table 3(a) shows: biological networks *CX_GSE1730* and *CX_GSE10158*, arXiv collaboration network *Ca-GrQc*, email communication network *Enron*, product co-purchasing network *Amazon*, social networks *Hyves* and *YouTube*, patent citation network *Patent*, protein k-mer graph *kmer* and USA road network *USA Road*. These graphs are selected to cover different graph type, size and degree characteristics.

Algorithms & Parameters. We test our 3 algorithms: (1) $\mathcal{A}_{base}$: one where a task spawned from a vertex mines its set-enumeration subtree in serial without decomposition; (2) $\mathcal{A}_{split}$: one that splits tasks by comparing $|ext(S)|$ with size threshold τ_{split} (c.f. Algorithm 4); (3) $\mathcal{A}_{time}$: one that splits tasks based on timeout threshold τ_{time} (c.f. Algorithm 5). Note that even $\mathcal{A}_{time}$ and $\mathcal{A}_{base}$ need τ_{split} which is used by *add_task*(t) to decide whether a task t is be put to the global queue or a local queue. We have repeated G-thinker paper [38]'s experiments using our new engine, and observe improvements compared with the old engine (see [19] for details).

We remark that $(\tau_{split}, \tau_{time})$ are algorithm parameters for parallelization. We also have the quasi-clique definition parameters (γ, τ_{size}) (recall Definition 3) at the first place. Interestingly, small value perturbations of (γ, τ_{size}) can have significant impact on the result number: if the parameters are too large, there will be 0 results; while if the parameter is too small, there can be millions of or more results and run for a long time. Table 4(a) (resp. Table 4(b)) shows the number of results (#{Results}) found by $\mathcal{A}_{base}$ and the maximal ones after postprocessing (#{Maximal}) along with the job time spent when we vary γ (resp. τ_{size}) slightly, where we can see that the result number is quite sensitive to the parameters. For example, when changing (γ, τ_{size}) from $(20, 0.9)$ to $(20, 0.89)$ on *Patent*, the result number increases from 256 to over 44 million. More experiments on the effect of (γ, τ_{size}) can be found in [19].

Also, the post-processing to remove non-maximal results is fast. For example, post-processing the 256 (resp. 44 million) results of *Patent* when $\gamma = 0.9$ (resp. 0.89) takes 0.002 sec (resp. 282.38 sec).

Table 3(b) shows the default values of (γ, τ_{size}) for each dataset that we find to return a reasonable number of result subgraphs for human examination. Note that this immediately allows k-core pruning of the input graphs where $k = \lceil \gamma \cdot (\tau_{size} - 1) \rceil$. We additionally prunes any vertex whose two-hop neighbor set has size $< \tau_{size}$, and statistics of the resulting dense graphs after pruning are shown in Table 3(b) where *YouTube* and *Patent* are still large.

Experimental Setup. 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.

Comparison of $\mathcal{A}_{base}$, $\mathcal{A}_{split}$ and $\mathcal{A}_{time}$. Table 5 shows the performance of our three G-thinker algorithm variants on all the datasets using the default (γ, τ_{size}) values in Table 3(b), and $(\tau_{split}, \tau_{time})$ tuned to achieve the best performance. There, we report the job running time, and the peak memory and disk usage on a machine. We can see that for graphs that are time-consuming to mine with $\mathcal{A}_{base}$, i.e., *YouTube* and *Patent*, $\mathcal{A}_{split}$ significantly speeds it up, which is in turn further accelerated by $\mathcal{A}_{time}$. For example, on *Patent*, $\mathcal{A}_{base}$, $\mathcal{A}_{split}$ and $\mathcal{A}_{time}$ takes 911, 98.68 and 36.66 sec, respectively. This shows the need to task decomposition to handle the straggler problem, and the advantage of our timeout strategy. In fact, if there is no straggler, $\mathcal{A}_{split}$ can be much slower than $\mathcal{A}_{base}$ as on *USA Road* due to excessive task decomposition, but $\mathcal{A}_{time}$ does not suffer from this issue. We also tested other parameters and the results are similar; for example, when mining *Patent* with $(\gamma, \tau_{size}) = (0.89, 20)$, $\mathcal{A}_{base}$, $\mathcal{A}_{split}$ and $\mathcal{A}_{time}$ take 3,386.37, 194.54, and 126.19 sec, respectively.

Also, the RAM usage is small, in fact less than 1GB except for on *YouTube*. There is also almost no task spilling on disk, with the exception of *Patent* where a machine may keep up to 1.28GB spilled tasks, potentially due to a lot of decomposed tasks generated at some point of time. Overall, space is not an issue.

Effect of $(\tau_{\mathrm{split}}, \tau_{\mathrm{time}})$. We have tested many value pairs for $(\tau_{split}, \tau_{time})$ and find that (50, 5 sec) consistently delivers either the best or almost the best performance for $\mathcal{A}_{time}$ on all datasets, and the complete results can be found in [19]. However, other settings may lead to significant increase in time. For example, on *Patent*, when fixing $\tau_{split} = 1,000$ and varying $\tau_{time} = 20, 10, 5, 1, 0.1$ sec, respectively, the job running time is 743.94, 561.82, 419.77, 179.59, 71.61 sec, respectively; while if we fix $\tau_{time} = 5$ sec and vary $\tau_{split} = 1000, 500, 200, 100, 50$ sec, respectively, the job running time is 419.77, 448.78, 426.75, 490.81, 36.66 sec, respectively.

Table 5: Performance of $\mathcal{A}_{base}$, $\mathcal{A}_{split}$ and $\mathcal{A}_{time}$ on All Datasets

			Abase			Asplit			Atime				
	τ_{split}	τ_{time}	Time (sec)	RAM (GB)	Disk (GB)	Time (sec)	RAM (GB)	Disk (GB)	Time (sec)	RAM (GB)	Disk (GB)	#{Maximal}	Postprocessing Time
CX_GSE1730	500	20	3.14	0.235	0	3.28	0.25	0	3.24	0.15	0	1,602	0.026 sec
CX_GSE10158	100	5	3.30	0.237	0	3.23	0.24	0	3.24	0.24	0	312	0.010 sec
Ca-GrQc	1,000	0.1	3.32	0.238	0	3.32	0.15	0	3.23	0.25	0	43,399	1.198 sec
Enron	1,000	20	5.38	0.313	0	8.49	0.46	0	7.40	0.3	0	200	0.002 sec
Amazon	100	10	3.31	0.24	0	3.26	0.24	0	3.24	0.24	0	13	0.001 sec
Hyves	50	20	7.45	0.409	0	8.40	0.60	0	7.31	0.45	0	1,480	0.015 sec
YouTube	500	0.01	3,690.13	9.44	0	552.36	6.05	0	506.48	16.02	0.27	274	0.010 sec
Patent	50	5	911.06	0.371	0	98.68	0.42	1.26	36.66	0.34	0.02	256	0.002 sec
kmer	100	1	17.09	0.927	0	16.46	0.91	0	16.37	0.91	0	63	0.001 sec
USA Road	1,000	10	16.21	0.87	0	26.21	0.74	0	17.30	0.74	0	16	0.001 sec

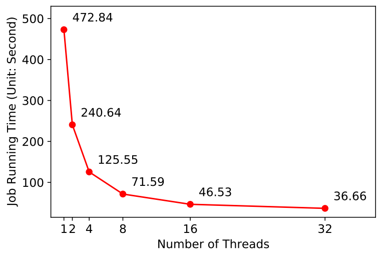

Figure 6: Vertical Scalability on *Patent*

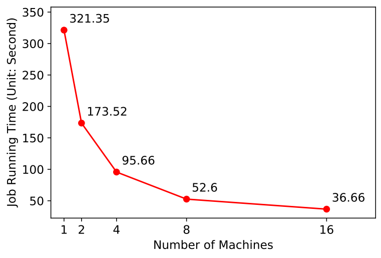

Figure 7: Horizontal Scalability on *Patent*

Comparison with [32]. Recall from Section 2 that [32] first mines quasi-cliques with $\gamma' > \gamma$, then finds the top-k' largest result subgraphs as "kernels" which are then expanded to generate γ-quasi-cliques and return top-k maximal ones from the results. Thus, a job of [32] takes a parameter quadruple (γ', k', γ, k). We use their code [1] for comparison, and set $k' = 3k$ following [32]'s setting. We observe that they cannot find the exact top-k quasi-cliques, and their program is also slower than our solution. See [19] for details.

We can speed up the approach of [32] by parallelization in G-thinker with minor system revision as detailed in [19]. However, we find that the performance is not improved much and even degrades when k' increases beyond 1, because of the need of redundant computation. Note that we can no longer only pull vertices with ID larger than those in S, or we will miss maximal results that can be obtained by expanding a "kernel" with vertices with smaller IDs.

Scalability. Figure 6 shows the vertical scalability of $\mathcal{A}_{time}$ on *Patent* where we use all 16 machines but change the number of threads on each machine, and Figure 7 shows the horizontal scalability of $\mathcal{A}_{time}$ on *Patent* where we run all 32 threads on each machine but change the number of machines. We can see that $\mathcal{A}_{time}$ scales well along both directions, which verifies that our solution is able to utilize the computing power of all machines in a cluster. Results on other datasets can be found in our technical report [19].

Table 6: Mining v.s. Subgraph Materialization on *Patent*

τ_{time}	Job Time	Total Task Mining Time	Total Subgraph Materialization Time	Mining-vs.-Materialization Time Ratio
50	128.65	7,831.56	0.30	26,417.73
20	65.56	8,303.44	0.62	13,403.82
10	43.53	9,005.62	1.23	7,310.96
1	34.63	9,260.19	9.04	1,024.39
0.5	39.70	9,245.71	18.31	504.85
0.1	53.57	9,661.53	78.84	122.55
0.01	57.58	10,721.14	334.36	32.06

Cost of Task Decomposition. Recall from Algorithm 6 that if a timeout happens, we need to generate subtasks with smaller overlapping subgraphs (see Lines 18-22), the subgraph materialization cost of which is not part of the original mining workloads. The smaller τ_{time} is, the more often task decomposition is triggered and hence more subgraph materialization overheads are generated.

Our tests show that the additional time spent on task materialization is not significant compared with the actual mining workloads. For example, Table 6 shows the profiling results on *Patent*, including the job running time, the sum of mining time spent by all tasks, the sum of subgraph materialization time spent by all tasks, and a ratio of the latter two. We can see that decreasing τ_{time} does increase the fraction of cumulative time spent on subgraph materialization due to more occurrences of task decompositions, but even with $\tau_{time} = 0.01$ sec, the materialization overhead is still only 1/32 of that for mining, so only a small cost is paid for better load balancing. Results on more graphs are in our technical report [19].

Quick+ v.s. Quick. We have compared our Quick+ with the original Quick algorithm on all the datasets in the single-threaded setting, and observe speedup that can be up to over 4×, the table of which can be found in our technical report [19]. Also, Quick did miss results although rare. For example, on *CX_GSE1730* (resp. *Ca-GrQc*), Quick finds 1,601 of the 1,602 valid quasi-cliques (resp. 43,398 of the 43,499 valid quasi-cliques), i.e., misses 1 result.

9 CONCLUSION

This paper proposed an algorithm-system codesign solution to fully utilize CPU cores in a cluster for mining maximal quasi-cliques. We provided effective load-balancing techniques such as timeout-based task decomposition and big task prioritization.

ACKNOWLEDGMENTS

This work was supported by NSF OAC-1755464, NSF DGE-1723250 and the NSERC of Canada.

REFERENCES

[1] [n.d.]. Code of the BigData 2018 Paper on Large Quasi-Clique Mining. https://github.com/beginner1010/topk-quasi-clique-enumeration.

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

[3] [n.d.]. Our code. https://github.com/yanlab19870714/gthinkerQC.

[4] James Abello, Mauricio G. C. Resende, and Sandra Sudarsky. 2002. Massive Quasi-Clique Detection. In *LATIN (Lecture Notes in Computer Science)*, Vol. 2286. Springer, 598–612.

[5] Gary D Bader and Christopher WV Hogue. 2003. An automated method for finding molecular complexes in large protein interaction networks. *BMC bioinformatics* 4, 1 (2003), 2.

[6] Vladimir Batagelj and Matjaz Zaversnik. 2003. An O(m) Algorithm for Cores Decomposition of Networks. *CoRR* cs.DS/0310049 (2003). http://arxiv.org/abs/cs.DS/0310049

[7] Devora Berlowitz, Sara Cohen, and Benny Kimelfeld. 2015. Efficient Enumeration of Maximal k-Plexes. In *SIGMOD*, Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives (Eds.). ACM, 431–444.

[8] Malay Bhattacharyya and Sanghamitra Bandyopadhyay. 2009. Mining the largest quasi-clique in human protein interactome. In *2009 International Conference on Adaptive and Intelligent Systems*. IEEE, 194–199.

[9] Mauro Brunato, Holger H Hoos, and Roberto Battiti. 2007. On effectively finding maximal quasi-cliques in graphs. In *International conference on learning and intelligent optimization*. Springer, 41–55.

[10] Dongbo Bu, Yi Zhao, Lun Cai, Hong Xue, Xiaopeng Zhu, Hongchao Lu, Jingfen Zhang, Shiwei Sun, Lunjiang Ling, Nan Zhang, et al. 2003. Topological structure analysis of the protein–protein interaction network in budding yeast. *Nucleic acids research* 31, 9 (2003), 2443–2450.

[11] Lijun Chang, Jeffrey Xu Yu, Lu Qin, Xuemin Lin, Chengfei Liu, and Weifa Liang. 2013. Efficiently computing k-edge connected components via graph decomposition. In *SIGMOD*, Kenneth A. Ross, Divesh Srivastava, and Dimitris Papadias (Eds.). ACM, 205–216.

[12] Yuan Heng Chou, En Tzu Wang, and Arbee L. P. Chen. 2015. Finding Maximal Quasi-cliques Containing a Target Vertex in a Graph. In *DATA*. SciTePress, 5–15.

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

[14] Patricia Conde-Cespedes, Blaise Ngonmang, and Emmanuel Viennet. 2018. An efficient method for mining the maximal α-quasi-clique-community of a given node in complex networks. *Social Network Analysis and Mining* 8, 1 (2018), 20.

[15] Alessio Conte, Donatella Firmani, Caterina Mordente, Maurizio Patrignani, and Riccardo Torlone. 2017. Fast Enumeration of Large k-Plexes. In *SIGKDD*. ACM, 115–124.

[16] Alessio Conte, Tiziano De Matteis, Daniele De Sensi, Roberto Grossi, Andrea Marino, and Luca Versari. 2018. D2K: Scalable Community Detection in Massive Networks via Small-Diameter k-Plexes. In *SIGKDD*, Yike Guo and Faisal Farooq (Eds.). ACM, 1272–1281.

[17] Wanyun Cui, Yanghua Xiao, Haixun Wang, Yiqi Lu, and Wei Wang. 2013. Online search of overlapping communities. In *SIGMOD*, Kenneth A. Ross, Divesh Srivastava, and Dimitris Papadias (Eds.). ACM, 277–288.

[18] Wenfei Fan, Ruochun Jin, Muyang Liu, Ping Lu, Xiaojian Luo, Ruiqi Xu, Qiang Yin, Wenyuan Yu, and Jingren Zhou. 2020. Application Driven Graph Partitioning. In *SIGMOD*.

[19] Guimu Guo, Da Yan, M. Tamer Özsu, and Zhe Jiang. 2020. Scalable Mining of Maximal Quasi-Cliques: An Algorithm-System Codesign Approach. *CoRR* abs/2005.00081 (2020).

[20] John Hopcroft, Omar Khan, Brian Kulis, and Bart Selman. 2004. Tracking evolving communities in large linked networks. *Proceedings of the National Academy of Sciences* 101, suppl 1 (2004), 5249–5253.

[21] Haiyan Hu, Xifeng Yan, Yu Huang, Jiawei Han, and Xianghong Jasmine Zhou. 2005. Mining coherent dense subgraphs across massive biological networks for functional discovery. *Bioinformatics* 21, suppl_1 (2005), i213–i221.

[22] Daxin Jiang and Jian Pei. 2009. Mining frequent cross-graph quasi-cliques. *ACM Trans. Knowl. Discov. Data* 2, 4 (2009), 16:1–16:42.

[23] Roberto J. Bayardo Jr. 1998. Efficiently Mining Long Patterns from Databases. In *SIGMOD*, Laura M. Haas and Ashutosh Tiwary (Eds.). ACM Press, 85–93.

[24] Pei Lee and Laks V. S. Lakshmanan. 2016. Query-Driven Maximum Quasi-Clique Search. In *SDM*. SIAM, 522–530.

[25] Junqiu Li, Xingyuan Wang, and Yaozu Cui. 2014. Uncovering the overlapping community structure of complex networks by maximal cliques. *Physica A: Statistical Mechanics and its Applications* 415 (2014), 398–406.

[26] Guimei Liu and Limsoon Wong. 2008. Effective Pruning Techniques for Mining Quasi-Cliques. In *ECML/PKDD (Lecture Notes in Computer Science)*, Walter Daelemans, Bart Goethals, and Katharina Morik (Eds.), Vol. 5212. Springer, 33–49.

[27] Can Lu, Jeffrey Xu Yu, Hao Wei, and Yikai Zhang. 2017. Finding the maximum clique in massive graphs. *Proc. VLDB Endow.* 10, 11 (2017), 1538–1549.

[28] Bingqing Lyu, Lu Qin, Xuemin Lin, Ying Zhang, Zhengping Qian, and Jingren Zhou. 2020. Maximum Biclique Search at Billion Scale. *Proc. VLDB Endow.* 13, 9 (2020), 1359–1372.

[29] Hideo Matsuda, T. Ishihara, and Akihiro Hashimoto. 1999. Classifying Molecular Sequences Using a Linkage Graph With Their Pairwise Similarities. *Theor. Comput. Sci.* 210, 2 (1999), 305–325.

[30] Jeffrey Pattillo, Alexander Veremyev, Sergiy Butenko, and Vladimir Boginski. 2013. On the maximum quasi-clique problem. *Discret. Appl. Math.* 161, 1-2 (2013), 244–257.

[31] Jian Pei, Daxin Jiang, and Aidong Zhang. 2005. On mining cross-graph quasi-cliques. In *SIGKDD*. ACM, 228–238.

[32] Seyed-Vahid Sanei-Mehri, Apurba Das, and Srikanta Tirthapura. 2018. Enumerating Top-k Quasi-Cliques. In *IEEE BigData*. IEEE, 1107–1112.

[33] Steve Sheng, Brad Wardman, Gary Warner, Lorrie Cranor, Jason Hong, and Chengshan Zhang. 2009. An empirical analysis of phishing blacklists. In *6th Conference on Email and Anti-Spam (CEAS)*. Carnegie Mellon University.

[34] Brian K. Tanner, Gary Warner, Henry Stern, and Scott Olechowski. 2010. Koobface: The evolution of the social botnet. In *eCrime*. IEEE, 1–10.

[35] Duygu Ucar, Sitaram Asur, Umit Catalyurek, and Srinivasan Parthasarathy. 2006. Improving functional modularity in protein-protein interactions graphs using hub-induced subgraphs. In *European Conference on Principles of Data Mining and Knowledge Discovery*. Springer, 371–382.

[36] Chun Wei, Alan Sprague, Gary Warner, and Anthony Skjellum. 2008. Mining spam email to identify common origins for forensic application. In *ACM Symposium on Applied Computing*, Roger L. Wainwright and Hisham Haddad (Eds.). ACM, 1433–1437.

[37] Daniel Weiss and Gary Warner. 2015. Tracking criminals on Facebook: A case study from a digital forensics REU program. In *Proceedings of Annual ADFSL Conference on Digital Forensics, Security and Law*.

[38] Da Yan, Guimu Guo, Md Mashiur Rahman Chowdhury, Tamer Özsu, Wei-Shinn Ku, and John C.S. Lui. 2020. G-thinker: A Distributed Framework for Mining Subgraphs in a Big Graph. In *ICDE*.

[39] Zhiping Zeng, Jianyong Wang, Lizhu Zhou, and George Karypis. 2006. Coherent closed quasi-clique discovery from large dense graph databases. In *SIGKDD*. ACM, 797–802.