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 \( \gamma \), a \( \gamma \)-quasi-clique is a subgraph \( g = (V_g, E_g) \) where each vertex \( v \in V_g \) connects to at least \( \gamma \) fraction of the other vertices (i.e., \( \gamma \cdot (|V_g| - 1) \)) 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 201x runtime speedup when mining a graph with 3.77M vertices and 16.5M edges in a 16-node cluster.

1 INTRODUCTION

Given a degree threshold \( \gamma \) and an undirected graph \( G \), a \( \gamma \)-quasi-clique is a subgraph of \( G \), denoted by \( g = (V_g, E_g) \), where each vertex connects to at least \( \gamma \cdot (|V_g| - 1) \) 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:
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, for the graph with 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-Mehr et al. [32] noticed that if \( y' \)-quasi-cliques (\( y' > y \)) are mined first using Quick which are faster to find, then it is more efficient to expand these “kernels” to generate \( y \)-quasi-cliques than to mine them from the original graph. Their kernel expansion is conducted only on those largest \( y' \)-quasi-cliques extracted by postprocessing, in order to find big \( y \)-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 \( y' \)-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.

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 vertices. We will elaborate on these weaknesses in Section 6.

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 \( v_4 \) to denote Vertex \( \hat{a} \) (the same for other vertices), thus we have \( N(v_4) = \{v_a, v_e, v_c, v_h, v_i\} \) and \( d(v_4) = 5 \). Also, let \( S = \{v_a, v_h, v_c, v_d, v_e\} \), 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 vertices of degree at least $k$ that are 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 $B(v) = N^*_2(v)$.

To illustrate using Figure 1, we have $N(u_v) = \{u_a, u_b, u_c, u_d\}$, $B(u_v) = \{u_f, u_g, u_h, u_i\}$, and $B(u_v)$ consisting of all vertices.

**Problem Definition.** We next formally define our problem.

**Definition 1 ($\gamma$-quasi-clique).** A graph $G = (V, E)$ is a $\gamma$-quasi-clique (0 ≤ $\gamma$ ≤ 1) if $G$ is connected, and for every vertex $v \in V$, its degree $d(v)$ is greater than $\gamma \cdot |V| - 1$.

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

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

To illustrate using Figure 1, consider $S_1 = \{u_a, u_b, u_c, u_d\}$ (i.e., vertices in red) and $S_2 = \{u_f, u_g, u_h, u_i\}$. If we set $\gamma = 0.6$, then both $S_1$ and $S_2$ are $\gamma$-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 $S_1 \subseteq S_2$, $G(S_1)$ is not a maximal $\gamma$-quasi-clique.

In the literature of dense subgraph mining, researchers usually only strive to find 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 $\gamma$, and so is an edge with its two endpoints. (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, 26]. Following [26], we use a minimum size threshold $t_{size}$ 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 $t_{size}$, we aim to find all the vertex sets $S$ such that $G(S)$ is a maximal $\gamma$-quasi-clique of $G$, and that $|S| \geq t_{size}$.

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 $b$ since $b < c$; 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 $(S, ext(S))$, 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 $\gamma$-quasi-clique. As we shall see, many vertices cannot form a $\gamma$-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 $S' \supseteq S$. Note that since $ext(S') \subset ext(S)$, the subgraph induced by nodes of a child task $(S', ext(S'))$ 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 pseudo-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 have 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 (cf. Column “Task Time”), and the time unit is millisecond (ms).
Table 1: Features of 10 Most Expensive Tasks on YouTube

| V  | E  | Max Degree | |/|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 | 31.94 | 49 | 13,623 | 1,565.75 |
| 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 | 1,024.41 |
| 3,177 | 101,008 | 1,890 | 31.80 | 49 | 15,267 | 1,521.73 |
| 3,321 | 95,084 | 1,330 | 23.74 | 38 | 15,584 | 529.81 |
| 3,723 | 133,828 | 1,849 | 30.58 | 46 | 16,881 | 1,745.78 |
| 26,235 | 694,686 | 7,105 | 26.48 | 51 | 3,645,905 | 529.81 |

Table 2: Features of 10 Most Expensive Tasks on Patent

| V  | E  | Max Degree | |/|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  | 3,382 | 82 | 35.07 | 60 | 1,083,755 | 4.07 |
| 69  | 1,786 | 65 | 25.88 | 43 | 1,198,086 | 1.48 |
| 78  | 2,262 | 69 | 29.30 | 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 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 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
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 \( \text{true} \) if the task \( t \) needs to call \( compute() \) for more iterations for further processing; it returns \( \text{false} \) 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 \( \text{frontier} \) to UDF \( compute(t, \text{frontier}) \). This allows multiple tasks to share requested vertices to minimize redundancy. In \( compute(t, \text{frontier}) \), task \( t \) is supposed to save the needed vertices and edges in \( \text{frontier} \) into its subgraph, as G-thinker releases \( t \)'s hold of those vertices in \( \text{frontier} \) right after \( compute(t, \text{frontier}) \) 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 \( compute(t, \text{frontier}) \) returns \( \text{true} \), \( t \) is added to a task queue to be scheduled to call \( compute() \) for more iterations; while if it returns \( \text{false} \), \( t \) 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 \( \text{true} \),

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 \( L_{small} \) 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 \( L_{small} \). 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 pseudo-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 \( L_{small} \) 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\((t, \text{frontier})\)
6. if \( t \) is not finished, suspend \( t \) to wait for data
7. obtain a task \( t' \) from \( B_{local} \)
8. repeat compute\((t', \text{frontier})\) 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 \( L_{big} \) 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 \( \tau_{split} \) so that if a task \( t = (S, \text{ext}(S)) \) has a subgraph with potentially more than \( \tau_{split} \) 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 \( \tau_{split} \), which prioritizes its execution to construct the potentially big subgraph early. While \( t \) is mining its constructed subgraph, we consider \( t \) as a big task iff \( \text{ext}(S) > \tau_{split} \), since there are \( \text{ext}(S) \) 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 computing, 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 \( Q_{local} \) is less than task batch size \( C \), with the priority order of getting a task batch from \( L_{small} \), 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.

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 \( L_{big} \). 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 \( L_{small} \), and then from its \( B_{local} \) in this prioritized order to minimize the number of partially processed tasks buffered on local disk tracked by \( L_{small} \).

If both \( L_{small} \) 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 \( L_{big} \)) 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+ 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 \( t_S = (S, \text{ext}(S)) \) 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 \( \text{ext}(S) \) into a valid quasi-clique. Task \( t_S \) can be recursively decomposed into tasks mining the subtrees \( \{T_S'\} \) where \( S' \subseteq S \) 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 \( \text{ext}(S) \): in such a rule, if a vertex \( u \in \text{ext}(S) \) satisfies certain conditions, \( u \) can be pruned from \( \text{ext}(S) \) since there must not exist a vertex set \( S' \) such that \( S \cup u \not\subseteq S' \subseteq (S \cup \text{ext}(S)) \) and \( G(S') \) is a \( \gamma \)-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 \( S \subseteq S' \subseteq (S \cup \text{ext}(S)) \) and \( G(S') \) is a \( \gamma \)-quasi-clique, and thus there is
no need to extend $S$ further. Type-II pruning invalidates the entire $T_3$. A variant invalidates $G(S')$, $S \subseteq S' \subseteq (S \cup \text{ext}(S))$ 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 $\gamma$-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 $u \in \text{ext}(S)$ must be within 2 hops from any $v \in S$, we have $\text{ext}(S) \subseteq \bigcap_{v \in S} B(v)$. This is a Type-I pruning rule since if $u \notin \bigcap_{v \in S} B(v)$, $u$ can be pruned from $\text{ext}(S)$.

(P2) **Size-Threshold Based Pruning.** A valid $\gamma$-quasi-clique $Q \subseteq V$ should contain at least $\tau_{\text{size}}$ vertices (i.e., $|Q| \geq \tau_{\text{size}}$), and therefore for any $\gamma \in Q$, its degree $d(\gamma) \geq \gamma \cdot (|Q| - 1) \geq \gamma \cdot (\tau_{\text{size}} - 1)$. We thus can prune any vertex with $d(u) < \gamma \cdot (\tau_{\text{size}} - 1)$ from $G$. Let $k = \gamma \cdot (\tau_{\text{size}} - 1)$, 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 $u \in \text{ext}(S)$; (3) ES-degrees: $d_{\text{ext}(S)}(v)$ for all $v \in S$; and (4) EE-degrees: $d_{\text{ext}(S)}(u)$ for all $u \in \text{ext}(S)$.

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 $\text{ext}(S)$. 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 $T_\gamma = (S, \text{ext}(S))$, Quick [26] defined an upper bound $U_\gamma$ (resp. lower bound $L_\gamma$) on the number of vertices in $\text{ext}(S)$ to be added to $S$ to form a $\gamma$-quasi-clique, based on the above mentioned degrees. We found that additional Type-II pruning not considered by Quick can happen when computing $U_\gamma$ (resp. $L_\gamma$) (see [19] for details).

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

(P7) **Cover-Vertex Based Pruning.** Given a vertex $u \in \text{ext}(S)$, Quick [26] defined a vertex set $C_S(u) \subseteq \text{ext}(S)$ such that for any $\gamma$-quasi-clique $Q$ generated by extending $S$ with vertices in $C_S(u)$, $Q \cup u$ is also a $\gamma$-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 $\text{ext}(S)$ 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 $\text{ext}(S)$ when checking the next level in the set-enumeration tree (see Figure 2), and only check until vertices of $\text{ext}(S) - C_S(u)$ are examined (i.e., the extension of $S$ using $V' \subseteq C_S(u)$ is pruned). To maximize the pruning effectiveness, we find $u \in \text{ext}(S)$ 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 $\gamma$-quasi-clique $Q$ constructed out of vertices in $C_S(u) = N(u)$, adding $u$ to $Q$ still produces a $\gamma$-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 $\text{ext}(S)$, which will reduce vertex degrees such as $d_{\text{ext}(S)}(v)$ of some $v \in S$, which will in turn update bounds $U_\gamma$ and $L_\gamma$ that are defined on the degrees. This essentially means that Type-I pruning is iterative: each pruned vertex $u \in \text{ext}(S)$ 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 $\text{ext}(S)$ may be pruned due to Type-I pruning, triggering another iteration.

As this process is repeated, $U_\gamma$ and $L_\gamma$ become tighter until no more vertex can be pruned from $\text{ext}(S)$, which consists of 2 cases:

- **C1:** $\text{ext}(S)$ becomes empty. In this case, we only need to check if $G(S)$ is a valid quasi-clique;
- **C2:** $\text{ext}(S)$ 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 $\text{ext}(S)$ and (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 $\text{ext}(S)$ 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 $\text{ext}(S)$ is passed as a reference and may be shrunk by Type-I pruning by the function.

Since $\text{ext}(S)$ can be pruned to become empty, we design this pruning sub-procedure to guarantee that it returns false only if $\text{ext}(S) \neq \emptyset$. Therefore, if the loop of Lines 1–15 exits due to $\text{ext}(S) = \emptyset$, 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 $U_\gamma$ and $L_\gamma$ in Line 3. In Line 3, recall from (P4 & P5) that Type-II pruning may apply when computing $U_\gamma$ and $L_\gamma$, 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 $\text{ext}(S)$ to $S$. We propose to find all critical vertices in $S$ and move their neighbors from $\text{ext}(S)$ 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: \text{iterative\_bounding}(S, ext(S), y, t_{size})

Output: true if 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: \quad Compute \( d_s(v) \) and \( d_{ext(S)}(v) \) for all \( v \) in \( S \) and \( ext(S) \)
3: \quad Compute upper bound \( U_S \) and lower bound \( L_S \)
4: \quad if \( \forall v \in S \) that is a critical vertex then
5: \quad \qquad \( I \leftarrow ext(S) \land N(v) \)
6: \quad \qquad \( S \leftarrow S \cup I, \text{ext}(S) \leftarrow ext(S) - I \)
7: \quad \qquad Update degree values, \( U_S \) and \( L_S \)
8: \quad for each vertex \( v \in S \) do
9: \quad \qquad \text{Check Type-II pruning conditions of (P3), (P4) and (P5)}
10: \quad \qquad \text{return true if Type-II pruning applies}
11: \quad for each vertex \( u \in ext(S) \) do
12: \quad \qquad \text{Check Type-I pruning conditions of (P3), (P4) and (P5)}
13: \quad \qquad if some Type-I pruning condition holds for \( u \) then
14: \quad \qquad \text{\( ext(S) \leftarrow ext(S) - u \)}
15: \quad \quad until \( ext(S) = \emptyset \) or no vertex in \( ext(S) \) was Type-I pruned
16: \quad if \( ext(S) = \emptyset \) then
17: \quad \quad \text{if \( |S| \geq t_{size} \) and \( G(S) \) is a \( y \)-quasi-clique then}
18: \quad \quad \quad Append \( S \) to the result file
19: \quad \quad return true
20: \quad 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 \( G(S) \) in Lines 17–18 before expanding \( S \) (i.e., Line 6). While our algorithm may output \( S \) while \( G(S) \) is not maximal, the post-processing 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 \( T_S \) we will 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: \text{recursive\_min}(S, ext(S), y, t_{size})

Output: true if a valid quasi-clique \( Q \) \( \supset S \) is found

1: \( T_{Q\_found} \leftarrow \text{false} \)
2: \quad Find cover vertex \( u \in ext(S) \) with the largest \( C_S(u) \)
3: \quad if not found, \( C_S(u) = \emptyset \)
4: \quad Move vertices of \( C_S(u) \) to the tail of the vertex list of \( ext(S) \)
5: \quad for each vertex \( v \in \) in the sub-list (\( ext(S) - C_S(u) \)) do
6: \quad \quad if \( |S| + |ext(S)| < t_{size} \) then
7: \quad \quad \quad \( T_{Q\_found} \leftarrow \text{false} \)
8: \quad \quad \text{Append } S \cup ext(S) \text{ to the result file}
9: \quad \quad return true
10: \quad S' \leftarrow S \cup v, \text{ext}(S) \leftarrow ext(S) - v
11: \quad ext(S') \leftarrow ext(S) \land \bar{v}(u)
12: \quad if \( ext(S') = \emptyset \) then
13: \quad \quad if \( |S'| \geq t_{size} \) and \( G(S') \) is a \( y \)-quasi-clique then
14: \quad \quad \quad \( T_{Q\_found} \leftarrow \text{true} \)
15: \quad \quad \quad Append \( S' \) to the result file
16: \quad else
17: \quad \quad \( T_{pruned} \leftarrow \text{iterative\_bounding}(S', ext(S'), y, t_{size}) \)
18: \quad \quad if \( \text{[here, ext(S') is Type-I pruned and ext(S') \neq \emptyset]} \)
19: \quad \quad \quad if \( T_{pruned} = \text{false} \) and \( |S'| + |ext(S')| > t_{size} \) then
20: \quad \quad \quad \quad \( T_{found} \leftarrow \text{recursive\_min}(S', ext(S'), y, t_{size}) \)
21: \quad \quad \quad \quad \( T_{Q\_found} \leftarrow T_{Q\_found} \lor T_{found} \)
22: \quad \quad \quad \text{if } T_{found} = \text{false} \text{ and } |S'| \geq t_{size} \text{ and } G(S') \text{ is a } y \text{-quasi-clique then}
23: \quad \quad \quad \quad \( T_{Q\_found} \leftarrow \text{true} \)
24: \quad \quad \quad Append \( S' \) to the result file
25: \quad \quad return \( T_{Q\_found} \)
26: \quad return \( T_{Q\_found} \)

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 \( \text{recursive\_min}(v, B_{\geq y}(v), y, t_{size}) \) on every \( v \in V \) where \( B_{\geq y}(v) \) denotes those vertices in \( B(v) \) (i.e., within 2 hops from \( v \)) whose IDs \( > y \), as according to Figure 2, we should not consider the other vertices in \( B(v) \) 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, \( \text{recursive\_min}(v, B_{> y}(v), y, t_{size}) \) only needs to be called on every \( v \in V - N(v_{max}) \).

Algorithm 3 keeps a boolean tag \( T_{Q\_found} \) 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 \( T_{Q\_found} \) as \( \text{false} \), but it will be set as \( \text{true} \) 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(v) corresponds to the case of extending S' using ext(S') ⊆ ext(S) ⊆ C_S(v) (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 rsize; 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 T_Q.found which indicates if a valid quasi-clique is found by previous iterations.

For a vertex v ∈ ext(S), the current iteration creates S' = S ∪ 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 ∪ 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 ∪ 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-premption 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') = ∅ 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') ≠ ∅, Line 18 then calls Algorithm 2 to apply the pruning rules. Recall that the function either returns T_pruned = false indicating that we need to further extend S' using its shrink ext(S') or it returns 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., $T_{pruned} = \text{false}$) and extending S' with all vertices in ext(S') still has the hope of generating a subgraph with $r_{size}$ vertices or larger (Line 20), we then recursively call our algorithm to examine S' in Line 21, which returns $T_{found}$ indicating if some valid maximal quasi-cliques $Q ⊆ S'$ are found (and output). If $T_{found} = \text{true}$, Line 22 will update the return value $T_{Q.found}$ as true, but $G(S')$ is not maximal. Otherwise (i.e., $T_{found} = \text{false}$), $G(S')$ 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 $f(v)$ hops in general) from v to construct v’s two-hop ego-network g from $B(v)$, and then mines g.

Here, if a task $t = (S, ext(S))$ 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 ($|V| - |N(v)|, \ldots, |V| - 2, |V| - 1$); for the other vertices, we sort them in ascending order of degree and assign IDs 1, 2, $\ldots$, $|V| - |N(v)| - 1$ 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 ∈ 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 $t'$ to the 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 $T_{found} = \text{false}$, 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 \(|\{t.\text{ext}(S)\} \leq \tau_{\text{split}}\) then
2: \text{recursive\_mine}(t.S, t.\text{ext}(S), \gamma, \tau_{\text{size}})
3: else
4: Find cover vertex \(u \in t.\text{ext}(S)\) 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 \(t.\text{ext}(S)\)
7: \text{for each} vertex \(v \in \text{the sub-list (}t.\text{ext}(S) - C_S(u)\)) do
8: if \(|t.S + |t.\text{ext}(S)| < \tau_{\text{size}}\) then \text{return false}
9: if \(G(t.S \cup t.\text{ext}(S))\) is a \(\gamma\)-quasi-clique then
10: Append \(t.S \cup t.\text{ext}(S)\) to the result file
11: return false
12: Create a task \(t'\)
13: \(t'.S \leftarrow t.S \cup v, t.\text{ext}(S) \leftarrow t.\text{ext}(S) - v\)
14: \(t'.\text{ext}(S) \leftarrow t.\text{ext}(S) \cap \big(\text{ext}(v)\big)\)
15: if \(|t'.S| \geq \tau_{\text{size}}\) and \(G(t'.S)\) is a \(\gamma\)-quasi-clique then
16: Append \(t'.S\) to the result file
17: \(T_{\text{pruned}} \leftarrow \text{iterative\_bounding}(t'.S, t'.\text{ext}(S), \gamma, \tau_{\text{size}})\)
18: if \(T_{\text{pruned}} = \text{false and } |t'.S| + |t'.\text{ext}(S)| \geq \tau_{\text{size}}\) then
19: \(t'.g \leftarrow \text{subgraph of } t.g\) induced by \(t'.S \cup t'.\text{ext}(S)\)
20: \(t'.\text{iteration} \leftarrow 3\)
21: \(\text{add\_task}(t')\)
22: else
23: Delete \(t'\)
24: return \text{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 \(|t.\text{ext}(S) - C_S(u)|\) subtasks (see Line 7) where \(u\) is the cover vertex.

Timeout-Based Task Decomposition. So far, we decompose a task \((S, \text{ext}(S))\) as long as \(|\text{ext}(S)| > \tau_{\text{split}}\) 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 \(\tau_{\text{split}}\). We, therefore, improve our UDF \text{compute}(t, \text{frontier}) further by a timeout strategy where we guarantee that each task spends at least a duration of \(\tau_{\text{time}}\) 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; then we wrap \([a, b, c, d]\) as a subtask to be added to our system, and backtracking 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 \text{compute}(t, \text{frontier}) is given by Algorithm 5. Line 1 calls our recursive backtrack function \text{time\_delayed}(S, \text{ext}(S), \text{initial\_time}) detailed in Algorithm 6, where \text{initial\_time} is the time when Iteration 3 begins. Line 2 then returns \text{false} to terminate this task.

Algorithm 6 \text{time\_delayed}(S, \text{ext}(S), \text{initial\_time})
1: \(\mathcal{T}_{Q_{\text{found}}^*} \leftarrow \text{false}\)
2: Find cover vertex \(u \in \text{ext}(S)\) 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 \(\text{ext}(S)\)
5: \text{for each} vertex \(v \in \text{the sub-list (}\text{ext}(S) - C_S(u)\)) do
6: if \(|S| + |\text{ext}(S)| < \tau_{\text{size}}\) then \text{return false}
7: if \(G(S \cup \text{ext}(S))\) is a \(\gamma\)-quasi-clique then
8: Append \(S \cup \text{ext}(S)\) to the result file; \text{return false}\n9: \(S' \leftarrow S \cup v, \text{ext}(S) \leftarrow \text{ext}(S) - v\)
10: \(\text{ext}(S') \leftarrow \text{ext}(S) - \big(\text{ext}(v)\big)\)
11: if \(\text{ext}(S') = \emptyset\) then
12: if \(|S'| \geq \tau_{\text{size}}\) and \(G(S')\) is a \(\gamma\)-quasi-clique then
13: \(\mathcal{T}_{Q_{\text{found}}^*} \leftarrow \text{true}\)
14: Append \(S'\) to the result file
15: else
16: \(T_{\text{pruned}} \leftarrow \text{iterative\_bounding}(S', \text{ext}(S'), \gamma, \tau_{\text{size}})\)
17: \{here, \text{ext}(S')\) is Type-I-pruned and \(\text{ext}(S') \neq \emptyset\)\}
18: if \(\text{current\_time} - \text{initial\_time} > \tau_{\text{time}}\) then
19: if \(T_{\text{pruned}} = \text{false and } |S'| + |\text{ext}(S')| \geq \tau_{\text{size}}\) then
20: Create a task \(t'\); \(t'.S \leftarrow S'\)
21: \(t'.\text{ext}(S) \leftarrow \text{ext}(S')\); \(t'.\text{iteration} \leftarrow 3\)
22: \(\text{add\_task}(t')\)
23: if \(|t'.S| \geq \tau_{\text{size}}\) and \(G(t'.S)\) is a \(\gamma\)-quasi-clique then
24: Append \(t'.S\) to the result file
25: else if \(T_{\text{pruned}} = \text{false and } |S'| + |\text{ext}(S')| \geq \tau_{\text{size}}\) then
26: \(\mathcal{T}_{Q_{\text{found}}^*} \leftarrow \text{time\_delayed}(S', \text{ext}(S'), \text{initial\_time})\)
27: \(\mathcal{T}_{Q_{\text{found}}^*} \leftarrow \mathcal{T}_{Q_{\text{found}}^*} \text{ or } \mathcal{T}_{Q_{\text{found}}^*}\)
28: if \(\mathcal{T}_{Q_{\text{found}}^*} = \text{false and } |S'| \geq \tau_{\text{size}}\) and \(G(S')\) is a \(\gamma\)-quasi-clique then
29: \(\mathcal{T}_{Q_{\text{found}}^*} \leftarrow \text{true}\)
30: Append \(S'\) to the result file
31: return \(\mathcal{T}_{Q_{\text{found}}^*}\)

8 EXPERIMENTS
This section reports our experiments. We have released the code of our redesigned G-thinker and quasi-clique algorithms on GitHub [3].
Datasets. We used 10 real graph datasets as Table 3(a) shows: biological networks CX_GSE1730 and CX_GSE10158, arXiv collaboration network Ca-GhQc, 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) $A_{base}$: one where a task spawned from a vertex mines its set-enumeration subtree in serial without decomposition; (2) $A_{split}$: one that splits tasks by comparing $\text{ext}(S)$ with size threshold $\tau_{split}$ (c.f. Algorithm 4); (3) $A_{time}$: one that splits tasks based on timeout threshold $\tau_{time}$ (c.f. Algorithm 5). Note that even $A_{time}$ and $A_{base}$ need $\tau_{split}$ which is used by $\text{add\_task}(t)$ 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-cliche definition parameters $(γ, τ_{size})$ (recall Definition 3) at the first place. Interestingly, small value perturbations of $γ$ (or $τ_{size}$) 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 results; while if the parameter is too large, there will be 0 results; while if the parameter is too small, there can be millions of results; while if the parameter is too large, there will be 0 results.

Algorithms & Parameters. We test our 3 algorithms: (1) $A_{base}$: one where a task spawned from a vertex mines its set-enumeration subtree in serial without decomposition; (2) $A_{split}$: one that splits tasks by comparing $\text{ext}(S)$ with size threshold $\tau_{split}$ (c.f. Algorithm 4); (3) $A_{time}$: one that splits tasks based on timeout threshold $\tau_{time}$ (c.f. Algorithm 5). Note that even $A_{time}$ and $A_{base}$ need $\tau_{split}$ which is used by $\text{add\_task}(t)$ 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-cliche definition parameters $(γ, τ_{size})$ (recall Definition 3) at the first place. Interestingly, small value perturbations of $γ$ (or $τ_{size}$) 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 results; while if the parameter is too large, there will be 0 results; while if the parameter is too small, there can be millions of results; while if the parameter is too large, there will be 0 results.

Algorithms & Parameters. We test our 3 algorithms: (1) $A_{base}$: one where a task spawned from a vertex mines its set-enumeration subtree in serial without decomposition; (2) $A_{split}$: one that splits tasks by comparing $\text{ext}(S)$ with size threshold $\tau_{split}$ (c.f. Algorithm 4); (3) $A_{time}$: one that splits tasks based on timeout threshold $\tau_{time}$ (c.f. Algorithm 5). Note that even $A_{time}$ and $A_{base}$ need $\tau_{split}$ which is used by $\text{add\_task}(t)$ 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-cliche definition parameters $(γ, τ_{size})$ (recall Definition 3) at the first place. Interestingly, small value perturbations of $γ$ (or $τ_{size}$) 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 results; while if the parameter is too large, there will be 0 results; while if the parameter is too small, there can be millions of results; while if the parameter is too large, there will be 0 results.

Algorithms & Parameters. We test our 3 algorithms: (1) $A_{base}$: one where a task spawned from a vertex mines its set-enumeration subtree in serial without decomposition; (2) $A_{split}$: one that splits tasks by comparing $\text{ext}(S)$ with size threshold $\tau_{split}$ (c.f. Algorithm 4); (3) $A_{time}$: one that splits tasks based on timeout threshold $\tau_{time}$ (c.f. Algorithm 5). Note that even $A_{time}$ and $A_{base}$ need $\tau_{split}$ which is used by $\text{add\_task}(t)$ 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-cliche definition parameters $(γ, τ_{size})$ (recall Definition 3) at the first place. Interestingly, small value perturbations of $γ$ (or $τ_{size}$) 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 results; while if the parameter is too large, there will be 0 results; while if the parameter is too small, there can be millions of results; while if the parameter is too large, there will be 0 results.

Algorithms & Parameters. We test our 3 algorithms: (1) $A_{base}$: one where a task spawned from a vertex mines its set-enumeration subtree in serial without decomposition; (2) $A_{split}$: one that splits tasks by comparing $\text{ext}(S)$ with size threshold $\tau_{split}$ (c.f. Algorithm 4); (3) $A_{time}$: one that splits tasks based on timeout threshold $\tau_{time}$ (c.f. Algorithm 5). Note that even $A_{time}$ and $A_{base}$ need $\tau_{split}$ which is used by $\text{add\_task}(t)$ 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-cliche definition parameters $(γ, τ_{size})$ (recall Definition 3) at the first place. Interestingly, small value perturbations of $γ$ (or $τ_{size}$) 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 results; while if the parameter is too large, there will be 0 results; while if the parameter is too small, there can be millions of results; while if the parameter is too large, there will be 0 results.

Algorithms & Parameters. We test our 3 algorithms: (1) $A_{base}$: one where a task spawned from a vertex mines its set-enumeration subtree in serial without decomposition; (2) $A_{split}$: one that splits tasks by comparing $\text{ext}(S)$ with size threshold $\tau_{split}$ (c.f. Algorithm 4); (3) $A_{time}$: one that splits tasks based on timeout threshold $\tau_{time}$ (c.f. Algorithm 5). Note that even $A_{time}$ and $A_{base}$ need $\tau_{split}$ which is used by $\text{add\_task}(t)$ 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-cliche definition parameters $(γ, τ_{size})$ (recall Definition 3) at the first place. Interestingly, small value perturbations of $γ$ (or $τ_{size}$) 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 results; while if the parameter is too large, there will be 0 results; while if the parameter is too small, there can be millions of results; while if the parameter is too large, there will be 0 results.

Algorithms & Parameters. We test our 3 algorithms: (1) $A_{base}$: one where a task spawned from a vertex mines its set-enumeration subtree in serial without decomposition; (2) $A_{split}$: one that splits tasks by comparing $\text{ext}(S)$ with size threshold $\tau_{split}$ (c.f. Algorithm 4); (3) $A_{time}$: one that splits tasks based on timeout threshold $\tau_{time}$ (c.f. Algorithm 5). Note that even $A_{time}$ and $A_{base}$ need $\tau_{split}$ which is used by $\text{add\_task}(t)$ 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-cliche definition parameters $(γ, τ_{size})$ (recall Definition 3) at the first place. Interestingly, small value perturbations of $γ$ (or $τ_{size}$) 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 results; while if the parameter is too large, there will be 0 results; while if the parameter is too small, there can be millions of results; while if the parameter is too large, there will be 0 results.
We observe that they cannot find the exact top-$k$ when on other datasets can be found in our technical report [19].

To utilize the computing power of all machines in a cluster. Results but change the number of machines. We can see that $A$ threads on each machine, and Figure 7 shows the horizontal scalability.

We thinker with minor system revision as detailed in [19]. However, we find that the performance is not improved much and even degrades $\gamma$ greater than those in $S$, or we will miss maximal results that can be found in our technical report [19].

Table 5: Performance of $A_{base}$, $A_{split}$ and $A_{time}$ on All Datasets

<table>
<thead>
<tr>
<th>$\tau$</th>
<th>$\tau_{split}$</th>
<th>$\tau_{time}$</th>
<th>$\gamma$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>0.001 sec</td>
<td>0.15 sec</td>
<td>0.25</td>
</tr>
<tr>
<td>0.01</td>
<td>0.001 sec</td>
<td>0.15 sec</td>
<td>0.25</td>
</tr>
<tr>
<td>0.001</td>
<td>0.001 sec</td>
<td>0.15 sec</td>
<td>0.25</td>
</tr>
<tr>
<td>0.0001</td>
<td>0.001 sec</td>
<td>0.15 sec</td>
<td>0.25</td>
</tr>
</tbody>
</table>

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 $\tau_{time}$ 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 $\tau_{time}$ 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


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