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

Guimu Guo∗, Da Yan†, M. Tamer Özsu‡, Zhe Jiang†, Jalal Majed Khalil∗

Guimu Guo and Da Yan are parallel first authors

∗Department of Computer Science, The University of Alabama at Birmingham
†David R. Cheriton School of Computer Science, University of Waterloo
‡Department of Computer Science, University of Alabama

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 \( γ \cdot |V_g| - 1 \) other vertices in \( V_g \). 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 recent distributed framework targeting divide-and-conquer graph mining algorithms that decomposes the mining into compute-intensive tasks to fully utilize CPU cores. However, 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 and the time growth with task-subgraph size. We address these challenges by redesigning G-thinker’s execution engine to prioritize long-running tasks for mining, and by utilizing a novel timeout strategy to effectively decompose the mining workloads of 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. We improve Quick by integrating new pruning rules, and fixing some missed boundary cases that could lead to missed results. 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.

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 \( γ \cdot |V_g| - 1 \) other vertices in \( V_g \). 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 [8, 28, 5, 10, 20, 34], and social communities [24, 19] that can correspond to cybercriminals [36], botnets [33, 38] and spam/phishing email sources [35, 32].

Mining maximal quasi-cliques is notoriously expensive [31], and the state-of-the-art algorithms [25, 30, 38] were only tested on small graphs. For example, Quick [25], the best among existing algorithms, was only tested on graphs with thousands of vertices [25]. 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, which has been made possible recently by the G-thinker [37] framework for distributed graph mining that avoids the IO bottleneck for data movement that exists in other existing data-intensive systems. In fact, it is found that using conventional IO-bound data-intensive systems could result in a throughput comparable or even less than a single-threaded program [2, 13], making it a must to use a compute-intensive framework like G-thinker.

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 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 [37]. Specifically, [31] showed that even the problem of determining whether a given quasi-clique in a graph is maximal is NP-hard, while [15] showed that “(i) maximal k-plexes are even more numerous than maximal cliques”, and that “(ii) the most efficient algorithms in the literature for computing maximal k-plexes can only be used on small-size graphs”. Unlike those simpler problems considered in [37] where the runtimes of individual tasks are relatively short compared with the entire mining workloads, quasi-clique mining generates tasks of drastically different running time which was not sufficiently handled by the G-thinker engine.

We remark that while the existing execution engine of G-thinker is insufficient, its 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 engine to address the straggler problem. After all, before G-thinker, such parallelization was not easy: [31] makes it a future work “Can the algorithms for quasi-cliques be parallelized effectively?”, while [15] indicated that “We are not aware of parallel techniques for implementing the all.plexes() sub-routine, and we leave this for future work”. Addressing the load balanc-
ing issue of G-thinker would not only benefit parallel quasi-clique mining, but also the parallelization of many other graph-divisible algorithms for mining dense subgraphs [2, 15, 16, 27, 11, 26, 17].

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 achieve effective and early decomposition of big tasks by a novel timeout strategy, without the need to predict task running time which is very difficult.

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

The rest of this paper is organized as follows. Section 2 reviews those related work closely related to 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 also adopted by Quick and our Quick+, and which is amenable to parallelization in 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 another version of it 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 [30, 27], and Cocoon [38] which finally led to the Quick algorithm [25] 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. [31] noticed that if \( \gamma' \)-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 \( \gamma \)-quasi-cliques than to mine them from the original graph. Their kernel expansion is conducted only on those largest \( \gamma' \)-quasi-cliques extracted by post-processing, in order to find big \( \gamma \)-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 find some \( \gamma' \)-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; however, as Section 8 shall show, the performance of this solution is only faster than our exact solution when \( k \) is very small.

Other than [31], quasi-cliques have seldom been considered in a big graph setting. Quick [25] 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 [30, 21, 38] 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 [23, 21, 14] 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, 29, 14] 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 [2, 15, 16, 27, 11, 26, 17], 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 \( N_{V'}(v) = \{ (u, v) \in E, u \in V' \} \). There are also \( S \subseteq \{v_1, v_2, v_3, v_4, v_5\} \), 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 \( d(u, v) \) 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 \( N_{k}(v) = \{ u | \delta(u, v) = k \} \) and define \( N_{k}^{+}(v) = \{ u | \delta(u, v) \leq k \} \). In a nutshell, \( N_{k}(v) \) are the set of vertices reachable from \( v \) within \( k \) hops, and \( N_{k}^{+}(v) \) are the set of vertices reachable from \( v \) in \( k \) hops.
We next formally define our problem.

**Definition 1 (γ-quasi-clique).** A graph \( G = (V, E) \) is a \( \gamma \)-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 \( \gamma \)-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 \( S' \supset S \) such that \( G(S') \) is a \( \gamma \)-quasi-clique.

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

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

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

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. 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 \((S, ext(S))\), 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 \( S' \supset S \). Note that since \( ext(S') \subseteq ext(S) \), 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. For example, when processing task that mines \( T_{<6} \), 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 \( T_{<6} \). 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 1 of Section 3. 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.

To conduct these experiments, we run quasi-clique mining using G-thinker where each task is spawned from a vertex \( v \) and mines the entire set-enumeration subtree \( T_{(v)} \) (i.e., \( S = \{v\} \)) in serial without generating any subtasks. As we shall see from pruning 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 low-degree vertices and vertices beyond \( f(\gamma) \) hops.

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 never a bottleneck: when some tasks are being scheduled or waiting for vertex data needed, other ready-tasks are being mined so almost all mining threads are busy on the actual mining workloads when there are enough tasks to process (i.e., not near the end of a job).
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 5 minutes and 25.4 seconds, but the last task alone takes 2878.7 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.

For Patent, we plot the impacts of |V|, |E|, maximum degree, average degree, and core # in the five subplots in Figure 4. Similar to Figure 3, we can see that for about the same feature values, the time can vary a lot along the vertical direction. The difference is that the task time varies even more where some tasks are so much more time-consuming that most other tasks have their time flatten to be close to 0 along the y-axis. To mitigate this issue, we also plot the diagrams by making the time in log scale. The plots are shown in Figure 5 where we can observe that the time still varies a lot along the vertical direction for similar feature values.

All the relevant analyses are shared as jupyter notebook files at https://github.com/yanlab19870714/gthinkerQC_taskTimeDistribution

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 [17] 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) \( \text{spawn}(v) \) indicating how to spawn a task from each individual vertex of the input graph; (2) \( \text{compute}(t, \text{frontier}) \) indicating how a task \( t \) processes an iteration where \( \text{frontier} \) keeps the adjacency lists of the requested vertices in the previous iteration. In a UDF, users may request for the adjacency lists of the requested vertices in the previous iteration. In a UDF, users may request for the adjacency lists of the requested vertices in the previous iteration. In a UDF, users may request for the adjacency lists of the requested vertices in the previous iteration. In a UDF, users may request for the adjacency lists of the requested vertices in the previous iteration.

UDF \( \text{compute}(t, \text{frontier}) \) 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...
Figure 3: Subgraph Features v.s. Task Time on *YouTube*
Figure 4: Subgraph Features v.s. Task Time on Patent
Figure 5: Subgraph Features vs. Logarithmic Task Time on Patent
can continue to process other tasks; \( t \) will be scheduled to call \( \text{compute}(\cdot) \) again once all its requested data become locally available.

UDF \( \text{compute}(t, \text{frontier}) \) returns \text{true} if the task \( t \) needs to call \( \text{compute}(\cdot) \) 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 6 shows the architecture (components) of G-thinker on a cluster of machines (yellow global task queues are the new additions 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 6 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 6 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 \( \text{compute}(t, \text{frontier}) \).

This allows multiple tasks to share requested vertices to minimize redundancy. In \( \text{compute}(t, \text{frontier}) \), 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 \( \text{compute}(t, \text{frontier}) \) returns, and they may be evicted from the vertex cache.

Our distributed vertex store and cache are designed to allow a graph to be processed even if it cannot fit entirely in the memory of a machine. But if the machine memory is large enough, a pulled vertex will never be evicted, so every vertex will be pulled for at most once, the cost of which is no larger than if we pre-load the entire graph into the memory of every machine.

If \( \text{compute}(t, \text{frontier}) \) returns \text{true}, \( t \) is added to a task queue to be scheduled to call \( \text{compute}(\cdot) \) 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 \( Q_{\text{local}} \) 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 \( B_{\text{local}} \) by the data serving module once all its requested vertices become locally available, to be fetched by the mining thread for calling \( \text{compute}(\cdot) \), and adding it to \( Q_{\text{local}} \) if \( \text{compute}(\cdot) \) 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 6 shows, each machine maintains a list \( L_{\text{small}} \) 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 \( L_{\text{small}} \). 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. Moreover, since the computations over subgraphs are the bottleneck rather than IO for subgraph cre-
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 \( Q_{\text{local}} \) does not have enough tasks then refill \( Q_{\text{local}} \)
4: pop a task \( t \) from \( Q_{\text{local}} \) and provide requested vertices
5: if all vertices are ready, repeat compute(t, frontier) till some vertex is not available
6: if \( t \) is not finished, suspend \( t \) to wait for data
7: obtain a task \( t' \) from \( B_{\text{local}} \)
8: repeat compute(t', frontier) till some vertex is not available
9: if \( t' \) is not finished, append \( t' \) to \( Q_{\text{local}} \)

System Redesign. Recall that a task in pseudo-clique mining can be very time consuming. If we only let each mining thread to 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 \( L_{\text{small}} \) and ready-task buffer \( B_{\text{local}} \)) to keep small tasks only. We similarly maintain a global task queue \( Q_{\text{global}} \) to keep big tasks shared by all computing threads, along with its associated task containers as shown in Figure 6. This order is to digest old/spilled tasks before spawning new tasks.

We define a user-specified threshold \( \tau_{\text{split}} \) so that if a task \( t = (S, \text{ext}(S)) \) has a subgraph with potentially more than \( \tau_{\text{split}} \) vertices to check, it is appended to \( Q_{\text{global}} \); otherwise, it is appended to \( Q_{\text{local}} \) of the current thread. Here, it is difficult to decide the subgraph size of \( t \) as it is changing. 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(\( t \)) is at least \( \tau_{\text{split}} \), 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 \( |\text{ext}(S)| > \tau_{\text{split}} \), 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 \( Q_{\text{local}} \) 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 \( B_{\text{local}} \) for computation, which is then appended to \( Q_{\text{local}} \) if further processing is needed.

“Pop” is only done if there is enough space left in the 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 \( Q_{\text{local}} \) is below task batch size \( C \), with the priority order of getting a task batch from \( L_{\text{small}} \), then from \( B_{\text{local}} \), 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 \( B_{\text{global}} \) for computing. The task may need to be appended back to \( Q_{\text{global}} \) or may be decomposed into smaller tasks to be appended either to \( Q_{\text{global}} \) or the thread’s \( Q_{\text{local}} \). (ii) If \( B_{\text{global}} \) is, however, found to be empty, the thread will instead fetch a small task from its \( B_{\text{local}} \) for computing.

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

In Case (i) if \( Q_{\text{global}} \) 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 \( L_{\text{big}} \). We do not check \( B_{\text{global}} \) 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 \( Q_{\text{local}} \) to pop, before which if the number of tasks therein is below a batch, task refill happens where lies our third change.

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

If both \( L_{\text{small}} \) and \( B_{\text{local}} \) 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 \( Q_{\text{global}} \) (previous tasks are added to \( Q_{\text{local}} \)). 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 \( Q_{\text{global}} \) plus \( L_{\text{big}} \)) 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 \( Q_{\text{global}} \); 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 RECURSIVE ALGORITHM

It has been pointed out that “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 and thus intractable. Different pseudo-clique mining algorithms propose different sophisticated pruning rules, and in the context of quasi-Clue, Quick [25] uses the most complete set of pruning rules. 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 that could lead to missed results in the original Quick algorithm.

6.1 Pruning Rules

Recall the set-enumeration tree in Figure 2 where each node represents a mining task, denoted by \( t_q = (S, \text{ext}(S)) \). Task \( t_q \) 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 \( ext(S) \subseteq (V - S) \) into a valid quasi-clique. Task \( t_S \) that mines \( T_S \) can be recursively decomposed into the mining of the subtrees \( \{T_S'\} \) where \( S' \supseteq S \) are child nodes of node \( S \). Our recursive serial algorithm basically 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.

To reduce search space, we consider two categories of pruning rules that can effectively prune either candidate vertices in \( ext(S) \) from expansion, or simply the entire subtree \( T_S \). Formally, we have

- **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 \( \gamma \)-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 \( S \subseteq S' \subseteq (S \cup ext(S)) \) and \( G(S') \) is a \( \gamma \)-quasi-clique, and thus there is no need to extend \( S \) further.

Type-II pruning invalidates the entire \( T_S \). 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 that are utilized by our algorithm, where each rule either belongs to Type I, or Type II, or sometimes both. Below we summarize these groups as (P1)–(P7), respectively.

(P1) **Graph-Diameter Based Pruning.** Theorem 1 of \( [30] \) defines the upper bound of the diameter of a \( \gamma \)-quasi-clique as a function \( f(\gamma) \). Often, we only consider the case where \( \gamma \geq 0.5 \), in which case the diameter is bounded by 2. To see this, consider any two vertices \( u, v \in V \) in a quasi-clique \( G \) that are not direct neighbors: since both \( u \) and \( v \) can be adjacent to at least \( 0.5 \cdot |V - 1| \) other vertices, they must share a neighbor (and thus are within 2 hops) or otherwise, there exist \( 2 \cdot 0.5 \cdot |V - 1| = |V - 1| \) vertices in \( V \) other than \( u \) and \( v \), leading to a contradiction since there will be more than \( |V| \) vertices in \( G \) when adding \( u \) and \( v \).

Without loss of generality, we use 2 as the diameter upper bound in our algorithm description, but it is straightforward to generalize it 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 \( v \in S \), i.e., \( u \in B(v) \), we obtain the following theorem:

**Theorem 1 (Diameter Pruning).** Given a mining task \( \gamma \), \( ext(S) \), we have \( ext(S) \subseteq \bigcap_{v \in S} B(v) \).

This is a Type-I pruning since if \( u \not\in \bigcap_{v \in S} B(v) \), \( u \) can be pruned from \( ext(S) \).

(P2) **Size-Threshold Based Pruning.** A valid \( \gamma \)-quasi-clique \( Q \subseteq V \) should contain at least \( \tau_{size} \) 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 have:

**Theorem 2 (Size Threshold Pruning).** If a vertex \( u \) has \( d(u) < \lceil \gamma \cdot (\tau_{size} - 1) \rceil \), then \( u \) cannot appear in any quasi-clique \( Q \) with \( |Q| \geq \tau_{size} \).

In other words, we can prune any such vertex \( u \) from \( G \). It is a Type-I pruning as \( u \not\in ext(S) \), and also a Type-II pruning as \( u \not\in S \). Note that a higher \( \tau_{size} \) significantly reduces the search space. Let us define \( k = \lceil \gamma \cdot (\tau_{size} - 1) \rceil \), this rule essentially shrinks \( G \) into its \( k \)-core, which is defined as 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 \( [6] \), which repeatedly deletes vertices with degree \( < k \) until there is no such vertex. We thus always shrink a graph \( G \) into its \( k \)-core before running our mining algorithm, which effectively reduces the search space.

(P3) **Degree-Based Pruning.** There are two degree-based pruning rules, which belong to Type I and Type II, respectively. Recall that \( d_{V'}(v) = |N_{V'}(v)| \), and thus \( d(v) \) denotes the number of \( v \)'s neighbors inside \( S \), and \( d_{ext}(S)(v) \) denotes the number of \( v \)'s neighbors inside \( ext(S) \). These two degrees are frequently used in our pruning rules to be presented subsequently.

**Theorem 3 (Type I Degree Pruning).** Given a vertex \( u \in ext(S) \), if Condition (i): \( d_{u}(u) + d_{ext}(S)(u) < \lceil \gamma \cdot |S| + d_{ext}(S)(u) \rceil \), then \( u \) can be pruned from \( ext(S) \).

This theorem is a result of the following lemma proved by \( [39] \).

**Lemma 1.** If \( a + n < \lceil \gamma \cdot (b + n) \rceil \) then, \( \forall i \in [0,n] \), we have \( a + i < \lceil \gamma \cdot (b + i) \rceil \).

**Theorem 3** follows since for any valid quasi-clique \( Q = S \cup V' \) where \( u \in V' \) and \( V' \subseteq ext(S) \), according to Condition (i) and Lemma 1 we have \( d_{S}(u) + d_{V'}(u) < \lceil \gamma \cdot (|S| + d_{V'}(u)) \rceil \leq \lceil \gamma \cdot (|Q| - 1) \rceil \) (since \( d_{V'}(u) \leq |V' - 1| \) and \( Q = S \cup V' \)), which contradicts with the fact that \( Q \) is a \( \gamma \)-quasi-clique.

**Theorem 4 (Type II Degree Pruning).** Given vertex \( v \in S \), if (i) \( d_{S}(v) < \lceil \gamma \cdot |S| \rceil \) and \( d_{ext}(S)(v) = 0 \), or (ii) if \( d_{S}(v) + d_{ext}(S)(v) < \lceil \gamma \cdot (|S| - 1) + d_{ext}(S)(v) \rceil \), then for any \( S' \) such that \( S \subseteq S' \subseteq (S \cup ext(S)), G(S') \) cannot be a \( \gamma \)-quasi-clique.

If Condition (ii) applies for any \( v \in S \), then for any \( S' \) such that \( S \subseteq S' \subseteq (S \cup ext(S)), G(S') \) cannot be a \( \gamma \)-quasi-clique.

**Theorem 4** (Condition (ii) also follows Lemma 1) \( d_{S}(v) + d_{V'}(v) < \lceil \gamma \cdot (|S| - 1 + d_{V'}(v)) \rceil \leq \lceil \gamma \cdot (|Q| - 1) \rceil \) (since \( d_{V'}(v) \leq |V' - 1| \) and \( Q = S \cup V' \)). Note that as long as we find one such \( v \in S \), there is no need to extend \( S \). If \( d_{ext}(S)(v) = 0 \) in Condition (ii), then we obtain \( d_{S}(v) < \lceil \gamma \cdot (|S| - 1) \rceil \), which is contained in Condition (i). Note that Condition (ii) applies to the case \( S' = S \) since \( i = 0 \) in Lemma 1.

Condition (i) allows more effective pruning and is correct since for any valid quasi-clique \( Q \supseteq S \) extended from \( S \), we have \( d_{S}(v) \leq d_{S}(v) + d_{ext}(S)(v) = d_{S}(v) < \lceil \gamma \cdot (|Q| - 1) \rceil \) (since \( d_{S}(v) \leq |S| \) and \( |S| \leq |Q| - 1 \), which contradicts with the fact that \( Q \) is a \( \gamma \)-quasi-clique. Note that the pruning of Condition (i) does not include the case where \( S' = S \).

(P4) **Upper Bound Based Pruning.** We next define an upper bound on the number of vertices in \( ext(S) \) that can be added to \( S \) concurrently to form a \( \gamma \)-quasi-clique, denoted by \( U_S \). The definition of \( U_S \) is based on \( d_{S}(v) \) and \( d_{ext}(S)(v) \) of all vertices \( v \in S \) and on \( d_{S}(u) \) of vertices \( u \in ext(S) \) as summarized by Figure 7 which we describe next.

We first define \( d_{min} \) as the minimum degree of any vertex in \( S \):
\[ d_{\text{min}} = \min_{v \in S} \{ d_S(v) + d_{\text{ext}(S)}(v) \}. \]  

(1)

Now consider any \( S' \) such that \( S \subseteq S' \subseteq (S \cup \text{ext}(S)) \). For any \( v \in S \), we have \( d_S(v) + d_{\text{ext}(S)}(v) \geq d_S(v) \geq \gamma [(|S'|-1)] \), and therefore, \( d_{\text{min}} \geq \gamma (|S'|-1) \). As a result, \( d_{\text{min}}/\gamma \geq [\gamma (|S'|-1)/\gamma ] \geq [\gamma (|S'|-1)/\gamma ] = |S'|-1 \), which gives the following upper bound on \( |S'| \):

\[ |S'| \leq [d_{\text{min}}/\gamma ] + 1. \]  

(2)

Since \( |S| \) vertices are already included, we obtain an upper bound \( U_{S_{\text{min}}}^\gamma \) on the number of vertices from \( \text{ext}(S) \) that can further extend \( S \) to form a valid quasi-clique:

\[ U_{S_{\text{min}}}^\gamma = [d_{\text{min}}/\gamma ] + 1 - |S|. \]  

(3)

We next tighten this upper bound using vertices in \( \text{ext}(S) = \{u_1, u_2, \ldots, u_n\} \), assuming that the vertices are listed in non-increasing order of \( d_S(u) \). Then, we have:

**Lemma 2.** Given an integer \( k \) such that \( 1 \leq k \leq n \), if \( \sum_{v \in S} d_S(v) + \sum_{1 \leq i \leq k} d_S(u_i) < |S| \cdot \gamma (|S| + k - 1) \), then for any vertex set \( Z \subseteq \text{ext}(S) \) with \( |Z| = k \), \( S \cup Z \) is not a \( \gamma \)-quasi-clique.

Note that if \( \gamma (|S'| - 1) \) is a \( \gamma \)-quasi-clique, then \( d_S(v) \geq \gamma (|S'|-1) \) for any \( v \in S' \), and therefore, for any \( S \subseteq S' \), we have \( \sum_{v \in S} d_S(v) \geq |S| \cdot \gamma (|S'| - 1) \). Thus, to prove Lemma 2, we only need to show that \( \sum_{v \in S} d_S(u) < |S| \cdot \gamma (|S| + |Z| - 1) \), which is because:

\[ \sum_{v \in S} d_S(u) = \sum_{v \in S} d_S(v) + \sum_{v \in S} d_S(u) \]
\[ = \sum_{v \in S} d_S(v) + \sum_{u \in Z} d_S(u) \]
\[ \leq \sum_{v \in S} d_S(v) + \sum_{1 \leq i \leq |Z|} d_S(u_i) \]
\[ < |S| \cdot \gamma (|S| + |Z| - 1) \].

Based on Lemma 2, we define a tightened upper bound \( U_S \) as follows:

\[ U_S = \max \left\{ t \mid \left( 1 \leq t \leq U_{S_{\text{min}}}^\gamma \right) \land \left( \sum_{v \in S} d_S(v) + \sum_{1 \leq i \leq t} d_S(u_i) \geq |S| \cdot \gamma (|S| + t - 1) \right) \right\}. \]  

(4)

If such a \( t \) cannot be found, then \( S \) cannot be extended to generate a valid quasi-clique, which is a Type II pruning. Otherwise, we further consider two pruning rules based on \( U_S \).

**Theorem 5.** (Type I Upper Bound Pruning). Given a vertex \( u \in \text{ext}(S) \), if \( d_S(u) + U_S < \gamma \cdot (|S| + U_S - 1) \), then \( u \) can be pruned from \( \text{ext}(S) \).

Consider any valid quasi-clique \( Q = S \cup V' \) where \( u \in V' \) and \( V' \subseteq \text{ext}(S) \). If the condition in Theorem 5 holds, i.e., \( d_S(u) + U_S < \gamma \cdot (|S| + U_S - 1) \), then based on Lemma 1 and the fact that \( |V'| \leq U_S \), we have:

\[ d_Q(u) + |V'|-1 < \gamma \cdot (|S'|+|V'|-1) = \gamma \cdot (|Q|-1) \].

Then, we have the following theorem:

**Figure 8: Lower Bound Derivation**

**Theorem 6.** (Type II Upper Bound Pruning). Given a vertex \( v \in S \), if \( d_S(v) + U_S < \gamma \cdot (|S| + U_S - 1) \), then for any \( S' \) such that \( S \subseteq S' \subseteq (S \cup \text{ext}(S)) \), \( G(S') \) cannot be a \( \gamma \)-quasi-clique.

**Theorem 7.** (Type I Lower Bound Pruning). Given a vertex set \( u \in \text{ext}(S) \), if \( d_S(u) + d_{\text{ext}(S)}(u) < \gamma \cdot (|S| + U_S - 1) \), then \( u \) can be pruned from \( \text{ext}(S) \).

**Theorem 8.** (Type II Lower Bound Pruning). Given a vertex \( v \in S \), if \( d_S(v) + d_{\text{ext}(S)}(v) < \gamma \cdot (|S| + U_S - 1) \), then for any \( S' \) such that \( S \subseteq S' \subseteq (S \cup \text{ext}(S)) \), \( G(S') \) cannot be a \( \gamma \)-quasi-clique.

**Definition 4.** (Critical Vertex). Let \( S \) be a vertex set. If there exists a vertex \( v \in S \) such that \( d_S(v) + d_{\text{ext}(S)}(v) = \gamma \cdot (|S| + U_S - 1) \), then \( v \) is called a critical vertex of \( S \).
Theorem 9 (Critical Vertex Pruning). If $v \in S$ 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., $N_{\text{ext}(S)}(v) \subseteq S'$.

This is because if $u \in N_{\text{ext}(S)}(v)$ is not in $S'$, then $d_S(v) < d_{S'}(v) + d_{\text{ext}(S)}(v) = \lceil |(S' + L_S) - 1| \rceil \leq \lceil |(S'| - 1)\rceil$, which contradicts the fact that $S'$ is a $\gamma$-quasi-clique. Therefore, when extending $S$, if we find $v \in S'$ is a critical vertex, we can directly add all vertices in $N_{\text{ext}(S)}(v)$ to $S$ for further mining.

(P7) Cover-Vertex Based Pruning. Given a vertex $u \in \text{ext}(S)$, we will define a vertex set $C_S(u) \subseteq \text{ext}(S)$ such that 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 compute $C_S(u)$ as the intersection of (1) $\text{ext}(S)$, (2) $N(u)$, and (3) $N(v)$ of any $v \in S$ that is not a neighbor of $u$:
\[
C_S(u) = N_{\text{ext}(S)}(u) \cap \bigcap_{v \in S \land v \not\in N(u)} N(v)
\]

We compute $C_S(u)$ only if $d_S(u) \geq \lceil \gamma \cdot |S| \rceil$ and for any $v \in S$ that are not adjacent to $u$, it holds that $d_S(v) \geq \lceil \gamma \cdot |S| \rceil$; otherwise, we deem this pruning inapplicable as they are pruned by Theorems 3 and 4.

For any $\gamma$-quasi-clique $Q$ that extends $S$ with vertices in $C_S(u)$, we now explain why $Q \cup u$ is also a $\gamma$-quasi-clique by showing that for any vertex $v \in Q \cup u$, it holds that $d_{Q \cup u}(v) \geq \lceil (Q \cup u) - 1 \rceil = \lceil \gamma \cdot |Q| \rceil$. There are 4 cases for $v$: (1) $v = u$: then since $u$ is adjacent to all the vertices in $C_S(u)$ and we require $d_S(u) \geq \lceil \gamma \cdot |S| \rceil$, we have $d_{Q \cup u}(u) = d_S(u) + |Q| - |S| \geq \lceil \gamma \cdot |S| \rceil + |Q| - |S| \geq \lceil \gamma \cdot |Q| \rceil$; (2) $v \in S$ and $v \not\in N(u)$: then since $v$ is adjacent to all the vertices in $C_S(u)$ and we require $d_S(v) \geq \lceil \gamma \cdot |S| \rceil$, we have $d_{Q \cup u}(v) = d_S(v) + |Q| - |S| \geq \lceil \gamma \cdot |S| \rceil + |Q| - |S| \geq \lceil \gamma \cdot |Q| \rceil$; (3) $v \in S$ and $v \in N(u)$: then we have $d_{Q \cup u}(v) = d_S(v) + 1 \geq \lceil \gamma \cdot (|S| - 1) \rceil + 1 \geq \lceil \gamma \cdot |Q| \rceil$; (4) $v \not\in Q$ - $S$: then we have $d_{Q \cup u}(v) = d_S(v) + 1 \geq \lceil \gamma \cdot (|Q| - 1) \rceil + 1 \geq \lceil \gamma \cdot |Q| \rceil$. In summary, $Q \cup u$ is a $\gamma$-quasi-clique and $Q$ is not maximal.

As a degenerate special case, initially when $S = \emptyset$, Eq 2 is essentially $C_S(u) = N_{\text{ext}(S)}(u) = N(u)$, 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) above, since otherwise, we may find a high-degree vertex without much pruning power (e.g., the center of a sparse star graph).

6.2 The Recursive Algorithm

We have summarized 7 categories of pruning rules (P1)–(P7). Next, we present our recursive algorithm for mining maximal quasi-cliques in topics (T1)–(T6) below, which effectively utilizes the pruning rules.

(T1) Size Threshold Pruning as a Preprocessing. First consider the size-threshold based pruning established by Theorem 2, which says that any vertex with degree less than $k = \lceil \gamma \cdot (\text{size} - 1) \rceil$ cannot be in a valid quasi-clique. This rule essentially shrinks an input graph $G$ into its $k$-core, which is defined as the maximal subgraph of $G$ where every vertex has degree $\geq k$. The $k$-core of $G$ can be computed in $O(|E|)$ time using a peeling algorithm [9], which repeatedly deletes vertices with degree $< k$ until there is no such vertex. We thus always shrink a graph $G$ into its $k$-core before running the mining algorithm to be described next, and since the $k$-core of $G$ is much smaller than itself, this pruning effectively reduces the search space.

(T2) Degree Computation. Since we are growing $G(S)$ into a valid quasi-clique by including more vertices in $\text{ext}(S)$, when we say we maintain $S$, we actually maintain $G(S)$: every vertex $v \in S$ is associated with an adjacency list in $G(S)$. Whenever we add a new vertex $u \in \text{ext}(S)$ to $G(S)$, for each $v \in N(u) \cap S$, we add $u$ (resp. $v$) to $v$’s (resp. $u$’s) adjacency list in $G(S)$.

Recall that our pruning rules use 4 kinds of vertex degrees:

- SS-degrees: $d_S(v)$ for all $v \in S$;
- SE-degrees: $d_S(u)$ for all $u \in \text{ext}(S)$;
- ES-degrees: $d_{\text{ext}(S)}(v)$ for all $v \in S$;
- EE-degrees: $d_{\text{ext}(S)}(u)$ for all $u \in \text{ext}(S)$.

As Figure 7 shows, computing $U_S$ requires the first 3 kinds of degrees; and as Figure 8 shows, computing $L_S$ requires the first 2 kinds of degrees. The EE-degrees are only used by Type I pruning rules of Theorems 3 and 7.

SS-degrees can be obtained from the adjacency list sizes of $G(S)$, SE-degrees and ES-degrees can be calculated together: for each $u \in \text{ext}(S)$, and for each $v \in N(u) \cap S$, $(u, v)$ is an edge crossing $S$ and $\text{ext}(S)$ and thus we increment both $d_{\text{ext}(S)}(u)$ and $d_{\text{ext}(S)}(v)$.

Finally, EE-degrees can be computed from adjacency lists of vertices in $\text{ext}(S)$, and since it is only needed by Type I pruning rather than computing $U_S$ and $L_S$, we can delay its computation to right before checking Type I pruning rules.

(T3) Type II Pruning Rules. We have described 3 major Type II pruning rules in Theorems 3, 6, and 8, which share the following common feature: every vertex $v \in S$ is checked and if the pruning condition is met for any $v$, $S$ along with any of its extensions cannot be a valid quasi-clique and are thus pruned.

The only exception is Theorem 4 Condition (ii), which prunes $S$’s extensions but not $S$ itself. Of course, if any of the other Type II pruning condition is met, $S$ is also pruned. Therefore, only when all Type II pruning conditions except for Theorem 4 Condition (i) are not met, will we consider $S$ as a candidate for a valid quasi-clique.

Also note that the computation of bounds $U_S$ and $L_S$ may also trigger Type II pruning. For example, in Eq 4, if a valid $t$ cannot be found, then any extension of $S$ can be pruned though $G(S)$ is still a candidate to check. In contrast, in Eq 7, if a valid $t$ cannot be found (including $t = 0$), then $S$ and its extensions are pruned; this also applies to Eq 8.

(T4) Iterative Nature of Type I Pruning. Recall that we have 3 major Type I pruning rules in Theorems 3, 6, and 7 which share the following common feature: every vertex $v \in \text{ext}(S)$ is checked and if the pruning condition is met for $u$, $u$ is pruned from $\text{ext}(S)$.

Note that removing a vertex $u$, from $\text{ext}(S)$ reduces $d_{\text{ext}(S)}(v)$ of every $v \in N(u) \cap S$, which will further update $U_S$ (see Figure 7), as well as $L_S$ (see Eq 8). This essentially means that the Type I pruning is iterative: each pruned $u$ may change degrees and bounds, which affects the various pruning rules (including Type I ones), which should be checked again and new vertices in $\text{ext}(S)$ may be pruned due to Type I pruning. As this process is repeated, $U_S$ and $L_S$ become tighter until no more vertex can be pruned from $\text{ext}(S)$, which consists of 2 cases:
Algorithm 2 Iterative Bound-Based Pruning

Function: iterative bounding(S, ext(S), γ, τsize)

Output: true iff the case of extending S (excluding S itself) is pruned; ext(S) is passed as a reference, and some elements may be pruned when the function returns

1: repeat
2: Compute dS(v) and dext(S)(v) for all v ∈ S and ext(S)
3: Compute upper bound US and lower bound LS (Type II pruning may apply)
4: if ∃ v ∈ S that is a critical vertex then
5: I ← ext(S) ∩ N(v)
6: S ← S ∪ I
7: ext(S) ← ext(S) − I
8: Update degree values, US and LS (Type II pruning may apply)
9: for each vertex v ∈ S do
10: Check Type II pruning conditions: Theorems[4,6,8] and S
11: if some condition other than Theorem 4 Condition (i) holds for v then
12: return true
13: if Theorem 4 Condition (i) holds for some v ∈ S then
14: if |S| ≥ τsize and G(S) is a γ-quasi-clique then
15: Append S to the result file
16: return true
17: for each vertex u ∈ ext(S) do
18: Check Type I pruning conditions: Theorems[4,6,8] and S
19: if some Type I pruning condition holds for u then
20: ext(S) ← ext(S) − u
21: until ext(S) = ∅ or no vertex in ext(S) was Type-I-pruned
22: if ext(S) = ∅ then
23: if |S| ≥ τsize and G(S) is a γ-quasi-clique then
24: Append S to the result file
25: return true
26: return false

- 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.

(T5) The Iterative Pruning Subprocedure. Given a vertex set S, and the set of vertices ext(S) to extend S into valid quasi-cliques, 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. In Algorithm 2, the return value is of a boolean type 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 when the function returns.

As (T4) indicates, the application of pruning rules is intrinsically iterative since the shrinking of ext(S) may trigger more pruning. This iterative process is described by Lines 1–21, and the loop ends if the condition in Line 21 is met which corresponds to the two cases C1 and C2 described in (T4).

We design function iterative bounding(S, ext(S), γ, τsize) to guarantee that it returns false only if ext(S) ̸= ∅. Therefore, if the loop of Lines 1–21 exits due to ext(S) becoming ∅, we have to return true (Line 25) 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 23–24; note that here, G(S) is not pruned by Type II pruning as otherwise, the loop will directly return true (see Lines 10–12).

Now let us focus on the loop body in Lines 2–20 about one pruning iteration, which can be divided into 3 parts: (1) Lines 2–8: critical vertex pruning, (2) Lines 9–16: Type II pruning, and (3) Lines 17–20: Type I pruning. To keep Algorithm 2 short, we omit some details but they are included in our descriptions.

First, consider Part 1. We compute the degrees in Line 2, which are then used to compute US and LS in Line 3. In Line 2, we do not need to compute EE-degrees since they are only used by Type I pruning; we actually compute it right before Part 3, since if any Type II pruning applies, the function returns and the computation of EE-degrees is saved. In Line 3, Type II pruning may apply when computing US and LS (see the paragraphs below Eqs (4) and (8), respectively), in which case we return true to prune S’s extensions. Note that for US’s case, we still need to examine G(S), and the actions are the same as in Lines 23–25. In Line 3, after we obtain US and LS, if US < LS we also directly return true to prune S and its extensions; note that since LS ≥ 1, S is not a valid quasi-clique as it needs to add at least LS vertices to be valid.

Then, Lines 4–7 then apply the critical-vertex pruning of Theorem 4. Line 4 first checks the condition of a critical vertex in Definition 4 which uses LS. Lines 5–7 then performs the movement of N(v) ∩ ext(S), which will change the degrees and hence bounds and so they are recomputed in Line 8. Similar to Line 3, Line 8 may trigger type II pruning so that the function returns true. Also similar to Line 3, after we obtain US and LS in Line 8, if US < LS we also directly return true to prune S and its extensions.

In our actual implementation, if ext(S) is found to be empty after running Line 7, we directly exit the loop of Lines 1–21, to skip the execution of Lines 8–21.

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 in Line 8 which may generate new critical vertices in the updated S, therefore, we actually loop Lines 4–8 until there is no more critical vertex in S.

Recall that Theorem 5 does not prune S itself, and it is possible that the expanded S leads to no valid quasi-cliques, making G(S) a maximal quasi-clique. We therefore actually first check G(S) as in Lines 23–24 before expanding S with N(v) ∩ ext(S). The original Quick does not examine G(S) and thus may miss results. While our algorithm may output S while G(S) is not maximal, but just like in Quick, we require a postprocessing phase to remove non-maximal quasi-cliques anyway.

Next, consider Part 2 on Type II pruning. Lines 9–12 first check the pruning conditions of Theorems 6, 8, and 11 on every vertex v ∈ S. If any condition other than Theorem 4 Condition (i) applies, S along with its extensions are pruned and thus Line 12 returns true. Otherwise, if Theorem 4 Condition (i) applies for some v ∈ S, then extensions of S are pruned but G(S) itself is not, and it is examined in Lines 14–16.

Finally, Part 3 on Type I pruning checks every vertex u ∈ ext(S) and tries to prune u using a condition of Theorems 6, 8, and 11 as shown in Lines 17–20. The shrinking of ext(S) may create new pruning opportunities for the next iteration.

To summarize, Quick+ improves Quick for iterative bounding 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 to improve pruning. (2) Type-II pruning may apply when computing US and LS (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.

(T6) The Recursive Main Algorithm. Given a vertex set S, and
Algorithm 3 Mining Valid Quasi-Cliques Extended from S

Function: recursive_mine(S, ext(S), γ, τ_{size})

Output: true iff some valid quasi-clique Q ⊇ S is found

1. T_{Q, found} ← false
2. Find cover vertex u ∈ ext(S) with the largest C₅(u)
3. if not found, C₅(u) ← ∅
4. Move vertices of C₅(u) to the tail of the vertex list of ext(S)
5. for each vertex v in the sub-list (ext(S) − C₅(u)) do
6. if (|S| + |ext(S)|) < τ_{size} then
7. return T_{Q, found}
8. if (G(S ∪ ext(S)) is a γ-quasi-clique then
9. Append S ∪ ext(S) to the result file
10. return true
11. S′ ← S ∪ v, ext(S) ← ext(S) − v
12. ext(S′) ← ext(S) ∩ B(v)
13. if ext(S′) = ∅ then
14. if |S′| ≥ τ_{size} and G(S′) is a γ-quasi-clique then
15. T_{Q, found} ← true
16. Append S′ to the result file
17. else
18. T_{pruned} ← iterative_bounding(S′, ext(S′), γ, τ_{size})
19. if T_{pruned} = false and ext(S′) ≠ ∅ then
20. T_{Q, found} ← recursive_mine(S′, ext(S′), γ, τ_{size})
21. T_{Q, found} ← T_{pruned} or T_{Q, found}
22. if T_{Q, found} = true and |S′| ≥ τ_{size} and G(S′) is a γ-quasi-clique then
23. T_{Q, found} ← true
24. Append S′ to the result file
25. return T_{Q, found}

the set of vertices ext(S) to extend S into valid quasi-cliques. Algorithm 3 shows our 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, B_≻(v), γ, τ_{size}) on every v ∈ V where B_≻(v) denotes those vertices in B(v) whose IDs are larger than v, as according to Figure 1 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, recursive_mine(v, B_≻(v), γ, τ_{size}) only needs to be called on every v ∈ 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 is found. Line 1 initializes T_{Q, found} as false, but it will be set as true if any valid quasi-clique Q is found.

Algorithm 3 examines S, and it decomposes this problem into the subproblems of examining S′ = S ∪ v for all v ∈ ext(S), as described by the loop in Line 5. Before the loop, we first apply cover vertex pruning as described in (P7) of Section 6.1 for the selected cover vertex u ∈ ext(S) (Line 2), we move its cover set C₅(u) to the tail of the vertex list of ext(S) (Line 4), so that the loop in Line 5 ends when v reaches a vertex in C₅(u). This is correct since Line 11 excludes an already examined v from ext(S) and so the loop in Line 5 with v scanning C₅(u) corresponds to the case of extending S′ using ext(S′) ⊆ ext(S) ⊆ C₅(u) (see Lines 11–12) which should be pruned. If we cannot find a cover vertex (see Line 2), then Line 5 iterates over all vertices of ext(S).

Note that in Line 2, we need to check every u ∈ ext(S) and keep the current maximum value of |C₅(u)|: if for a vertex u we find when evaluating Eq. (3) that |N_{ext}(u)| is already less than the current maximum, u can be skipped without further checking N(v) for v ∈ S − N(u).

Now let us focus on the loop body in Lines 6–25. Line 6 first checks if S extended with every vertex not yet considered in ext(S) can generate a subgraph larger than τ_{size} (note that already-considered vertices v are removed from ext(S) by Line 11 in previous iterations which automatically guarantees the ID-based deduplication illustrated in Figure 2; if so, current and future iterations 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 lookahead technique in [25]. Note that G(S ∪ ext(S)) must satisfy the size threshold requirement as Line 6 is passed, and thus Line 8 does not need to check that condition again.

Now assume that lookahead technique does not prune the search, then Line 11 creates S′ = S ∪ v (the implementation actually updates G(S) into G(S′)), 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 for S.

Then, Line 12 shrinks ext(S) into ext(S′) by ruling out vertices more than 2 hops away from v according to (P1) of Section 6.1, 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 candidate for a valid quasi-clique and is thus examined in Lines 14–16. We remark that [25]’s original Quick algorithm misses this check and thus may miss results.

If ext(S′) ≠ ∅, Line 18 then calls iterative_bounding(S′, ext(S′), γ, τ_{size}) (i.e., 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 22–25 and 14–16 in Algorithm 2).

If Line 18 decides that S′ can be further extended (i.e., 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 T_{found} indicating if some valid maximal quasi-cliques Q ⊇ S′ are found (and output). If T_{found} = true, Line 22 will update the return value T_{Q, found} as true, but G(S′) is not maximal. Otherwise (i.e., 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.
Algorithm 4 UDF task_spawn(v)

Define \( k = \lceil \gamma (\tau_{\text{size}} - 1) \rceil \).

1. if \( |N(v)| \geq k \) then
2. Create a task \( t \)
3. \( t.\text{iteration} \leftarrow 1 \)
4. \( t.\text{root} \leftarrow v \) \{spawning vertex\}
5. \( t.S \leftarrow v \)
6. for each \( u \in N(v) \) with \( u > v \) do
7. \( t.\text{pull}(u) \)
8. add_task(t)

Algorithm 5 UDF compute(t, frontier)

Define \( k = \lceil \gamma (\tau_{\text{size}} - 1) \rceil \)

1. if \( t.\text{iteration} = 1 \) then
2. iteration_1(t, frontier)
3. else if \( t.\text{iteration} = 2 \) then
4. iteration_2(t, frontier)
5. else
6. iteration_3(t)

7. To maximize the success probability of the lookahead technique in Lines 8–10 of Algorithm 4, what 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 22 so that high-degree vertices tend to appear in \( ext(S) \) of more set- enumeration tree nodes.

7. PARALLEL G-THINKER ALGORITHMS

Divide-and-Conquer Algorithm. We next adapt Algorithm 3 to run on the redesigned G-thinker, where a big task (judged by \( |ext(S)| \)) is divided into smaller subtasks for concurrent processing. If a task \( t = \langle S, ext(S) \rangle \) is spawned from a vertex \( v \), we only pull vertices with \( ID > v \) into \( S \) and \( ext(S) \), which avoids redundancy (recall Figure 2). Whenever we say a task \( t \) pulls a vertex \( u \), we implicitly mean that we only do so when \( u > v \) that spawned \( t \).

Recall from Theorem 2 that any vertex with degree less than \( k = \lceil \gamma (\tau_{\text{size}} - 1) \rceil \) cannot be in a valid quasi-clique. Therefore, our implementation shrinks the subgraph \( g \) of any task \( t \) into the \( k \)-core of \( g \) before mining. We adopt the \( O(|E|) \)-time peeling algorithm 6 for this purpose.

Recall that users write a G-thinker program by implementing two UDFs, and here we spawn a task from each vertex \( v \) by pulling vertices within two hops from \( v \), to construct \( v \)'s two-hop ego-network from \( B(v) \). Of course, we only pull vertices with \( ID > v \) here and prune vertices with degree \( < k \), so that the resulting subgraph to mine is effectively a \( k \)-core. 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 \( v_{\text{max}} \) 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 lookahead pruning. We can then use the old-to-new ID mapping table to recode the IDs in the adjacency lists.

We first consider UDF task_spawn(v) as given by Algorithm 4. Specifically, we only spawn a task for a vertex \( v \) if its degree \( \geq k \) (Lines 1–2). The task is initialized to be at iteration 1 (Line 3, to be used by Line 1 of Algorithm 5 later), with spawning vertex \( v \) (Line 4, recorded so that future iterations only pull vertices larger than it) and \( S = \{ v \} \) (Line 5). The task then pulls the adjacency lists of \( v \)'s neighbors (Lines 6–7) and gets itself added to the system for further processing (Line 8).

Algorithm 6 iteration_1(t, frontier)

1. \( v \leftarrow t.\text{root} \)
2. \( t.N \leftarrow V(\text{frontier}) \cup v \)
3. \( V_1 \leftarrow \) vertices in \( \text{frontier} \) with degree \( \geq k \)
4. \( V_2 \leftarrow \) vertices in \( \text{frontier} \) with degree \( < k \)
5. Construct subgraph \( t.g \) to include vertices \( V_1 \cup v \)
6. for each vertex \( u \) in \( t.g \) do
7. for each vertex \( w \in N(u) \) do
8. if \( w > v \) and \( w \notin V_2 \) then
9. Add \( u \) to \( t.g \) if \( u \)'s adjacency list in \( t.g \)
10. \( t.g \leftarrow k\text{-core}(t.g) \)
11. if \( v \notin V(t.g) \) then return false
12. for each vertex \( u \) in \( t.g \) do
13. for each vertex \( w \in N(u) \) do
14. if \( w > v \) and \( w \notin t.N \) then
15. \( t.\text{pull}(w) \)
16. iteration_1 \( \leftarrow 2 \)
17. return true \{continue iteration 2\}

Next, UDF compute(t, frontier) runs 3 iterations as shown in Algorithm 5. The first iteration adds the pulled first-hop neighbors of \( v \) into the task’s subgraph \( t.g \) with proper size-threshold based pruning, and then pulls the second-hop neighbors of \( v \). The second iteration adds the pulled second-hop neighbors into \( t.g \) with proper size-threshold based pruning, and since \( t \) does not need to pull any more vertices, \( t \) will not be suspended but rather run the third iteration immediately. The third iteration then mines quasi-cliques from \( t.g \) using our recursive algorithm (Algorithm 3), but if the task is big, it will create smaller subtasks for concurrent computation. We next present the algorithms of Iterations 1–3, respectively.

The algorithm of Iteration 1 is given by Algorithm 6 where \( v \) is the task-spawning vertex (Line 1). In Line 2, we collect \( v \) and its neighbors already pulled inside \( \text{frontier} \) into a set \( N \) which records all vertices within 1 hop to \( v \), which will be used in Line 14 to filter them when pulling the second-hop neighbors. Then, we divide the pulled vertices into two sets: \( V_1 \) containing those with degree \( \geq k \) (Line 3) and \( V_2 \) containing those with degree \( < k \) (Line 4) which should be pruned.

We then construct the task’s subgraph \( t.g \) to include vertices \( V_1 \cup v \) in Line 5, and Lines 6–9 prune the adjacency lists of vertices in \( t.g \) by removing a destination \( w \) if it is smaller than \( v \) or if it is in \( V_2 \) (i.e., has degree \( < k \)). Note that the adjacency list of a vertex \( u \) in \( t.g \) may contain a destination \( w \) that is 2 hops away from \( v \); since we do not have \( N(u) \) yet, we cannot compare the degree of \( w \) with \( k \) for pruning.

After the adjacency list pruning, a vertex \( u \) in \( t.g \) may have its adjacency list shorter than \( k \), and therefore we run the peeling algorithm over \( t.g \) to shrink \( t.g \) into its \( k \)-core (Line 10); here, a destination \( w \) that is 2 hops away from \( v \) in an adjacency list stays untouched and we only remove vertices in \( V_1 \cup v \) (though \( w \) is counted for degree checking). If \( v \) becomes pruned from \( t.g \), compute(t, frontier) returns false to terminate \( t \) since \( t \) is to find quasi-cliques that contain \( v \) (Line 11).

Next, Lines 12–15 pull all second-hop vertices (away from \( v \)) in the adjacency lists of vertices of \( t.g \). Note that Line 14 makes sure that a vertex \( w \) to pull is not within 1 hop (i.e., \( w \notin N \) and \( w > v \)). In the actual implementation, we add all such vertices into a set and then pull them to avoid pulling the same vertex twice when checking \( N(v_a) \) and \( N(v_b) \) of different \( v_a, v_b \in V(t.g) \). Finally, Line 16 sets \( t.\text{iteration} \leftarrow 2 \) so that when compute(t, frontier) is called again, it will execute iteration_2(t, frontier).

Algorithm 7 gives the computation in Iteration 2. Line 2 first collects \( B \) as all vertices within 2 hops from \( v \), which is used in
Algorithm 7 iteration_2(t, frontier)

1: \( v \leftarrow t.\text{root} \)
2: \( B \leftarrow V(\text{frontier}) \cup t.N \)
3: for each vertex \( u \) in \( \text{frontier} \) do
4: \( \text{if } |N(u)| \geq k \) then
5: \( \text{Add } u \) into \( t.g \)
6: for each vertex \( w \in N(u) \) do
7: \( \text{if } w > v \) and \( w \in B \) then
8: \( \text{Add } w \) to \( u \)'s adjacency list in \( t.g \)
9: \( t.g \leftarrow k-core(t.g) \)
10: \( \text{if } v \notin t.g \text{ then return false} \)
11: \( t.\text{iteration} \leftarrow 3 \)
12: \( t.S \leftarrow \{v\} \), \( t.\text{ext}(S) \leftarrow V(g) - v \)
13: return \text{true} \ (\text{continue iteration 3})

Algorithm 8 iteration_3(t)

1: \( \text{if } |t.\text{ext}(S)| \leq \tau_{\text{split}} \) then
2: \( \text{recursive_mine}(t.S, t.\text{ext}(S), \gamma, \tau_{\text{size}}) \)
3: else
4: \( \text{Find cover vertex } u \in t.\text{ext}(S) \text{ with the largest } C_S(u) \)
5: \( \{\text{If not found, } C_S(u) = \emptyset\} \)
6: \( \text{Move vertices of } C_S(u) \text{ to the tail of vertex list } t.\text{ext}(S) \)
7: for each vertex \( v \) in the sub-list \( t.\text{ext}(S) - C_S(u) \) do
8: \( \text{if } |t.S| + |t.\text{ext}(S)| < \tau_{\text{size}} \) then return false
9: \( \text{if } G(t.S \cup t.\text{ext}(S)) \text{ is a } \gamma\text{-quasi-clique} \) then
10: \( \text{Append } t.S \cup t.\text{ext}(S) \) 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: \( t'.\text{ext}(S) \leftarrow t.\text{ext}(S) \cap B(u) \)
15: \( \text{if } |t'.S| \geq \tau_{\text{size}} \) and \( G(t'.S) \) is a \( \gamma\text{-quasi-clique} \) then
16: \( \text{Append } t'.S \) to the result file
17: \( T_{\text{pruned}} \leftarrow \text{iterative_bounding}(t'.S, t'.\text{ext}(S), \gamma, \tau_{\text{size}}) \)
18: \( \text{if } T_{\text{pruned}} = \text{false} \) 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: \( \text{Delete } t' \)
24: return false \ (\text{task is done})

Line 7 to filter out adjacency list items of those vertices in \( \text{frontier} \) that are 3 hops from \( v \). Recall that \( t.N \) is collected in Line 2 of Algorithm 8 to contain the vertices within 1 hop from \( v \), and that we are finding \( \gamma\text{-quasi-cliques} \) with \( \gamma \geq 0.5 \) and hence the quasi-clique diameter is upper bounded by 2.

Lines 3–8 then add all second-hop vertices in \( \text{frontier} \) with degree \( \geq k \) into \( t.g \) (Lines 4–5), but prunes a destination \( w \) in an adjacency list if \( w < v \) or \( w \) is not within 2 hops from \( v \) (i.e., \( w \notin B \)). Since adjacency lists may become shorter than \( k \) after pruning, Line 9 then shrinks \( t.g \) into its \( k\)-core, and if \( v \) is no longer in \( t.g \), compute(\( t, \text{frontier} \)) returns false to terminate the task (Line 10). Finally, Line 11 sets \( t.\text{iteration} \) to 3 so that when compute(\( t, \text{frontier} \)) is called again, it will execute iteration_3(\( t \)) which we present next. Since \( t \) does not pull any vertex in iteration 2, G-thinker will schedule \( t \) to run Iteration 3 right away.

Now that \( t.g \) contains the \( k\)-core of the spanning vertex’s 2-hop ego-network, Algorithm 8 gives the computation in Iteration 3 which mines quasi-cliques from \( t.g \). Since the task can be prohibitive when \( t.g \) and \( t.\text{ext}(S) \) are big, we only directly process the task using Algorithm 3 when \( |t.\text{ext}(S)| \) is small enough (Lines 1–2); otherwise, we divide it into smaller subtasks to be scheduled for further processing (Lines 3–23), though the execution flow is very similar to Algorithm 3.

Recall that Algorithm 8 is recursive where Line 21 extends \( S \) with another vertex \( v \in t.\text{ext}(S) \) 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 \( t'.\text{iteration} = 3 \) 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 \( t'.S \cup t'.\text{ext}(S) \) so that the subtask is on a smaller graph, and since \( t'.\text{ext}(S) \) 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 8 where we only check if \( G(S') \) is a valid quasi-clique when \( T_{\text{pruned}} = \text{false} \), i.e., the recursive call in Line 21 verifies that \( S' \) fails to be extended to produce a valid quasi-clique. In Algorithm 8 however, the recursive 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 found.

Timeout-Based Task Decomposition. So far, we decompose a task \( (S, t.\text{ext}(S)) \) as long as \( |t.\text{ext}(S)| > \tau_{\text{split}} \) 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 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 9 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 \( t_i \) times out; we then wrap \{a, b, c, d\} 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.

Algorithm 9 iteration_3(t) with Timeout Strategy

1: \( \text{time_delayed}(t.S, t.\text{ext}(S), \text{initial_time}) \)
2: return false \ (\text{task is done})
Table 3: Graph Datasets

| Data            | | | | Max Degree | URL                                                                 |
|-----------------|---|---|---|-------------|----------------------------------------------------------------------|
| CX_GSE1730      | 998 | 5,006 | 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         |
| Ce-GroC         | 5,342 | 14,496 | 2.77 | 81  | https://snap.stanford.edu/data/ca-GroC.html                      |
| Enron           | 36,892 | 183,831 | 5.01 | 1,382 | https://snap.stanford.edu/data/email-Enron.html                  |
| Amazon          | 334,633 | 925,872 | 2.76 | 54  | https://snap.stanford.edu/data/com-Amaazon.html                    |
| Hyves           | 1,402,673 | 2,777,419 | 1.98 | 31,883 | http://konect.uni-koblenz.de/networks/hyves/                |
| YouTube         | 1,134,800 | 2,987,624 | 2.63 | 28,754 | https://snap.stanford.edu/data/com-Youtube.html                |
| Patent          | 3,774,768 | 16,518,947 | 4.38 | 790  | http://konect.uni-koblenz.de/networks/it-Patent.html            |
| kmer            | 67,716,231 | 69,389,281 | 1.02 | 35  | http://graphchallenge.s3.amazonaws.com/synthetic/old/ua-km.tsv  |
| USA Road        | 23,947,347 | 28,854,312 | 1.20 | 0   | http://pers.dia.unimib.it/challenge9/download.shtml             |

(a) Statistics of graph datasets

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

Algorithm 10 **time_delayed(S, ext(S), initial_time)**

1: \( T_{Q_{found}} \leftarrow \text{false} \)
2: Find cover vertex \( u \in ext(S) \) with the largest \( C_S(u) \)
3: If not found, \( C_S(u) \leftarrow 0 \)
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 \( |S| + |ext(S)| < \tau_time \) then: return \( false \)
   7: if \( G(S \cup ext(S)) \) is a \( \gamma \)-quasi-clique then: return \( false \)
   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: \( ext(S') \leftarrow ext(S) \cap B(v) \)
  11: if \( ext(S') = \emptyset \) then
  12: if \( |S'| \geq \tau_{size} \) and \( G(S') \) is a \( \gamma \)-quasi-clique then
  13: \( T_{Q_{found}} \leftarrow true \)
  14: Append \( S' \) to the result file
  15: else
  16: \( T_{pruned} \leftarrow \text{iterative_bounding}(S', ext(S'), \gamma, \tau_{size}) \)
  17: if not \( \{ here, ext(S') \} \) is Type-I-pruned and \( ext(S') \neq \emptyset \)
  18: if \( \text{current_time} - \text{initial_time} > \tau_time \) then
  19: \( T_{pruned} = \text{false} \) and \( |S'| + |ext(S')| \geq \tau_{size} \) then
  20: Create a task \( t' \); \( t'.S \leftarrow S \)
  21: \( t'.ext(S) \leftarrow ext(S'), t'.iteration \leftarrow 3 \)
  22: \( \text{add_task}(t') \)
  23: if \( |S'| \geq \tau_{size} \) and \( G(t'.S) \) is a \( \gamma \)-quasi-clique then
  24: Append \( t'.S \) to the result file
  25: else if \( T_{pruned} = \text{false} \) and \( |S'| + |ext(S')| \geq \tau_{size} \) then
  26: \( T_{Q_{found}} \leftarrow \text{time_delayed}(S', ext(S'), \text{initial_time}) \)
  27: \( T_{Q_{found}} \leftarrow T_{Q_{found}} \) or \( T_{Q_{found}} \)
  28: if \( T_{Q_{found}} = \text{false} \) and \( |S'| \geq \tau_{size} \) and \( G(S') \) is a \( \gamma \)-quasi-clique then
  29: \( T_{Q_{found}} \leftarrow true \)
  30: Append \( S' \) to the result file
  31: return \( T_{Q_{found}} \)

With the timeout strategy, the third iteration of our UDF compute(t, frontier) is given by Algorithm 9. Line 1 calls our recursive backtracking function time_delayed(S, ext(S), initial_time) detailed in Algorithm 10 where initial_time is the time when iteration 3 begins. Line 2 then returns false to terminate this task.

Algorithm 10 now considers 2 cases. (1) Lines 18–24: if timeout happens, we wrap \( (S', ext(S')) \) into a task \( t' \) to be added for processing just like in Algorithm 8 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 8 where we recursively call time_delayed(\( t' \)) to process \( (S', ext(S')) \) 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].

Datasets. We used 10 real graph datasets as Table 3(a) shows: biological networks CX_GSE1730 and CX_GSE10158, arXiv collaboration network Ca-GroC, 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 \( |ext(S)| \) with size threshold \( \tau_{split} \) (c.f. Algorithm 8); (3) \( A_{time} \): one that splits tasks based on timeout threshold \( \tau_{time} \) (c.f. Algorithm 9). Note that even \( A_{time} \) and \( A_{base} \) need \( \tau_{split} \) 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 [27]’s experiments using our new engine as shown in Table 4, where column “G-thinker” refers to the old engine while “G-thinker++” refers to our redesigned engine. We observe improvements of our redesigned engine compared with using the old engine in most cases, and in the remaining cases the performance is similar; also, G-thinker is much faster than all prior systems.

We remark that \( (\tau_{split}, \tau_{time}) \) are algorithm parameters for parallelization. We also have the quasi-clique definition parameters \( (\gamma, \tau_{size}) \) (recall Definition 5) at the first place. Interestingly, small value perturbations of \( (\gamma, \tau_{size}) \) 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 or even billions of results and run for a long time. Table 5 (resp. Table 6) shows the number of results \(|\{\text{Results}\}|\) found by \( A_{base} \) and the maximal ones after postprocessing \(|\{\text{Maximal}\}|\) along with the job time spent when we vary \( \gamma \) (resp. \( \tau_{size} \)) slightly, where we can see that the result number is quite sensitive to the parameters. For example, when changing \( (\gamma, \tau_{size}) \) from (20, 0.9) to (20, 0.89) on Patent, the result number increases from 256 to over 44 million; and when changing \( (\gamma, \tau_{size}) \) from (23, 0.9) to (21, 0.9) on Hyves, the result number increases from 114 to 11,087. Since our goal is to find the pool of largest valid subgraphs for prioritized examination, trials of different parameters are necessary and it is important that each trial should run an efficient algorithm like ours.
Table 4: System Comparison for [37]'s Experiments

| Dataset   | Triangle Counting (TC) | Maximum Clique Size | Subgraph Matching (GM) |
|-----------|------------------------|----------------------|------------------------|
| Youtube   | 88.88 s / 2.9 GB       | 59.40 s / 6.9 GB     | 19.74 s / 0.5 GB       |
| Skitter   | 133.48 s / 4.8 GB      | 141.42 s / 0.7 GB    | 98.98 s / 1.4 GB       |
| Orkut     | 783.51 s / 17.7 GB     | 691.00 s / 2.5 GB    | 24 hr / 7.3 GB         |
| BTC       | x                      | x                    | > 24 hr / 7.3 GB       |
| Friendster| x                      | x                    | 10,915 s / 9.2 GB      |

(b) Maximum Clique Finding (MCF):

| Dataset   | Youtube | Skitter | Orkut   | BTC     | Friendster |
|-----------|---------|---------|---------|---------|------------|
| Time (sec)| 12.07   | 141.42  | 691.00  | 691.00  | 10,915     |
| # Results | 6.74    | 5.02    | 71.45   | 326.80  | 354.23     |
| # Maximal | 0       | 0       | 0       | 0       | 0          |

(c) Subgraph Matching (GM):

| Dataset   | Youtube | Skitter | Orkut   | BTC     | Friendster |
|-----------|---------|---------|---------|---------|------------|
| Time (sec)| 19.74   | 20.78   | 7.91    | 8.41    | 3,669.00   |
| # Results | 5.51    | 7.84    | 7.84    | 116.17  | 42,000.00  |
| # Maximal | 0       | 0       | 0       | 0       | 0          |

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

Table 5: Effect of γ

| Dataset   | Tsize | γ   | Time (sec) | # Results | # Maximal |
|-----------|-------|-----|------------|-----------|-----------|
| Patent    | 20    | 0.91| 41.48      | 0         | 0         |
|           | 0.9   | 911.06       | 256       | 256       |
|           | 0.89  | 3,386.37     | 44,083,840| 44,080,758|
| Hyves     | 22    | 0.92| 7.44       | 0         | 0         |
|           | 0.91  | 7.40         | 6         | 6         |
|           | 0.9   | 7.45         | 2,349     | 1,480     |
|           | 0.89  | 7.45         | 2,347     | 1,480     |
|           | 0.88  | 8.45         | 2,375     | 1,433     |
|           | 0.87  | 9.41         | 2,455     | 1,139     |
|           | 0.86  | 14.44        | 71,558    | 34,274    |

| Enron     | 23    | 0.92| 4.38       | 0         | 0         |
|           | 0.91  | 4.39         | 15        | 15        |
|           | 0.9   | 5.38         | 336       | 200       |
|           | 0.89  | 7.32         | 339       | 200       |
|           | 0.88  | 20.40        | 352       | 191       |
|           | 0.87  | 39.48        | 1,250     | 740       |

We also remark that the post-processing cost of removing nonmaximal results is negligible compared with the job running time, by using a prefix tree organization of the result vertex sets. For example, post-processing the 256 results of Patent when γ = 0.9 takes 0.002 seconds, while post-processing the over 44 million results when γ = 0.89 takes 282.38 seconds.

Table 3(b) shows the default values of (γ, τsize) 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 = ⌈γ · (τsize − 1)⌉. We additionally prune any vertex whose two-hop neighbor set has size < τsize, 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 $A_{base}$, $A_{split}$ and $A_{time}$. Table 7 shows the performance of our three G-thinker algorithm variants on all the datasets using the default ($\gamma, \tau_{size}$) 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 $A_{base}$, i.e., YouTube and Patent, $A_{split}$ significantly speeds it up, which is in turn further accelerated by $A_{time}$. For example, on Patent, $A_{base}$, $A_{split}$ and $A_{time}$ takes 91, 98.68 and 36.66 seconds, 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, $A_{split}$ can be much slower than $A_{base}$ as on USA Road due to excessive task decomposition, but $A_{time}$ 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), $A_{base}$, $A_{split}$ and $A_{time}$ take 3,386.37, 194.54, and 126.19 seconds, 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 ($\tau_{split}$, $\tau_{time}$). We have tested the various pairs of values for ($\tau_{split}$, $\tau_{time}$) on our datasets, and the results are shown in Tables S(a)-(j). We can see that (50, 5 sec) consistently delivers either the best or close to the best performance for $A_{time}$ across all our datasets. 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 seconds, respectively, the job running time is 743.94, 561.82, 419.77, 179.59, 71.61 seconds, respectively; while if we fix $\tau_{time}$ = 5 sec and vary $\tau_{split}$ = 1000, 500, 200, 100, 50 seconds, respectively, the job running time is 419.77, 448.78, 426.75, 490.81, 36.66 seconds, respectively.

Comparison with [31]. Recall from Section 2 that [31] first mines quasi-cliques with $\gamma > \gamma'$, then finds the top-k largest result subgraphs as “kernels” which are then expanded to generate $\gamma$-quasi-cliques and return top-k maximal ones from the results. Thus, a job of [31] takes a parameter quadruple ($\gamma'$, $k'$, $\gamma$, $k$). We use their code [1] for comparison, and set $k'$ = 3k following [31]’s setting.

We observe that they cannot find the exact top-k quasi-cliques. For example, on GSE10158, with ($\gamma'$, $k'$, $\gamma$, $k$) = (0.9, 30, 0.8, 10), the maximum subgraph found has 31 vertices while the true one has 32. If we reduce $\gamma'$ = 0.85 to include more results, it finds only 5 subgraphs with 32 vertices, but there are actually 6 maximal 0.8-quasi-cliques with 32 vertices. This happens even if we reduce $\gamma'$ to 0.81 (very close to $\gamma$). As another study, on Amazon, with ($\gamma'$, $k'$, $\gamma$, $k$) = (0.501, 300, 0.5, 100), only 9 subgraphs are returned with 6 with 13 vertices, and 3 with 12 vertices. In contrast, there are actually 13 0.5-quasi-cliques with 12 vertices or more.

Their program is also slower than our G-thinker’s solution. For example, running their program on YouTube and Hyves with exactly the same parameters as in [31] (where $\tau_{size}$ = 5), YouTube takes 11,985.84 seconds just to get the top-100 results while even our slowest $A_{base}$ runs for only 3,690.13 seconds to find all the 750 results (247 of which are maximal); Hyves takes 2,836.35 seconds to get the top-100 results while even our slowest $A_{base}$ runs for only 7.45 seconds to find all the 2,349 results (1,480 of which are maximal). In fact, even the serial Quick+ takes only 348.49 seconds on Hyves to find those results, thanks to our new degenerate cover-vertex pruning technique.

The other datasets we use were not considered in [31]. Here, we try different parameters and find that even with smaller parameters ($k'$ = 30, $k$ = 10) to allow faster running time, the program is not faster than our slowest exact programs $A_{base}$ as reported in Table 7. The results are reported in Table S where we set $\gamma' = \gamma + 0.05$ in most cases since a larger $\gamma'$ leads to zero results in our tests.

We can speed up the approach of [31] by parallelization in G-thinker with minor system revision. Specifically, we revise the maximum clique mining program of G-thinker [37] to find top-k largest cliques (instead of only one biggest clique). We then revise G-thinker so that each machine initially loads a portion of clique “kernels” $S$ to construct tasks $t_S = \langle S, ext(S) \rangle$ for mining, which are initially loaded to the global queue. The difference here is that we no longer have a spawning vertex $v$ so we will pull 2-hop neighbors of all vertices in $S$ with $k$-core pruning to construct $ext(S)$, and then mine task subgraph $G(S \cup ext(S))$ with proper task decomposition. Each machine no longer spawns tasks from individual vertices in the local vertex table.

Note, however, that each task $t_S$ 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 a vertex with a smaller ID, but [31] seems still does so to allow faster mining. So if $k$ is large, we will have a lot of redundant search space exploration by different “kernel”, even degrading the performance. Of course, we can compromise the result maximality requirement and only pull vertices with ID larger than those in $S$ to eliminate redundancy, but our current implementation is not considering this.

Table 11(a) shows the result when we use top-1 kernel to expand quasi-cliques with different $\tau_{size}$. Table 11(b) shows the result when we use top-1 kernel to expand quasi-cliques with different $\gamma$. 

### Table 7: Performance of $A_{base}$, $A_{split}$ and $A_{time}$ on All Datasets

| Dataset      | $\tau_{split}$ | 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      | 1,602    | 0.026 sec           |
| CX_GSE10158  | 100           | 5          | 3.30     | 0.237     | 0          | 3.23     | 0.24      | 0          | 3.24     | 0.24      | 312      | 0.010 sec           |
| CaGrQc       | 1,000         | 0.1        | 3.32     | 0.238     | 0          | 3.32     | 0.15      | 0          | 3.23     | 0.25      | 43,399   | 1.198 sec           |
| Amazon       | 100           | 10         | 3.31     | 0.24      | 0          | 3.26     | 0.24      | 0          | 3.24     | 0.24      | 13       | 0.001 sec           |
| Hyves        | 50            | 20         | 7.45     | 0.409     | 0          | 8.40     | 0.60      | 0          | 7.31     | 0.45      | 1,480    | 0.015 sec           |
| YouTube      | 500           | 0.01       | 3,690.13 | 9.44      | 0          | 552.36   | 6.05      | 0          | 506.48   | 16.02     | 274      | 0.010 sec           |
| Patent       | 50            | 5          | 911.06   | 0.371     | 0          | 98.68    | 0.42      | 1.26       | 36.66    | 0.34      | 256      | 0.002 sec           |
| kmer         | 100           | 1          | 17.09    | 0.927     | 0          | 16.46    | 0.91      | 0          | 16.37    | 0.91      | 63       | 0.001 sec           |
| USA Road     | 1,000         | 10         | 16.21    | 0.87      | 0          | 26.21    | 0.74      | 0          | 17.30    | 0.74      | 16       | 0.001 sec           |


Table 8: Effect of $(\tau_{\text{split}}, \tau_{\text{time}})$ on All Datasets

| Dataset          | $\tau_{\text{split}}$ | $\tau_{\text{time}}$ | $\gamma$ | $\gamma'$ | Time (sec) | #{Results} | Time of Abase |
|------------------|------------------------|----------------------|----------|---------|-----------|-----------|-------------|
| CX_GSE1730       | 30                     | 10                   | 0.9      | 0.95    | 34.66     | 10        | 3.14        |
| CX_GSE10158      | 29                     | 10                   | 0.8      | 0.85    | 7.02      | 10        | 3.30        |
| Ca-GrQc          | 10                     | 10                   | 0.8      | 0.85    | >24 h     | N/A       | 3.32        |
| Enron            | 30                     | 10                   | 0.9      | 0.95    | 84.77     | 0         | 5.38        |
| Amazon           | 12                     | 10                   | 0.5      | 0.55    | 16.41     | 0         | 3.31        |
| Hyves            | 5                      | 100                  | 0.75     | 300     | 2,836.35  | 100       | 7.45        |
| YouTube          | 5                      | 100                  | 0.8      | 300     | 11,985.84 | 100       | 3,690.13    |

Table 9: Performance of [31]

| Dataset          | $\tau_{\text{Time}}$ | $k$ | $\gamma$ | $k'$ | $\gamma'$ | Time (sec) | #{Results} | Time of Abase |
|------------------|-----------------------|-----|----------|------|----------|-----------|-----------|-------------|
| CX_GSE1730       | 30                     | 10  | 0.9      | 0.95 | 34.66    | 10        | 3.14      |
| CX_GSE10158      | 29                     | 10  | 0.8      | 0.85 | 7.02     | 10        | 3.30      |
| Ca-GrQc          | 10                     | 10  | 0.8      | 0.85 | >24 h    | N/A       | 3.32      |
| Enron            | 23                     | 10  | 0.9      | 0.95 | 84.77    | 0         | 5.38      |
| Amazon           | 12                     | 10  | 0.5      | 0.55 | 16.41    | 0         | 3.31      |
| Hyves            | 5                      | 100 | 0.75     | 300  | 2,836.35 | 100       | 7.45      |
| YouTube          | 5                      | 100 | 0.8      | 300  | 11,985.84| 100       | 3,690.13  |

20
and hence more subgraph materialization overheads are generated. The cost of which is not part of the original mining workloads. If a task timeout happens, we need to generate subtasks with smaller overlapping subgraphs (see Lines 18-22), the subgraph materialization overheads are generated.

Recall from Algorithm 10 that if a task timeout happens, we need to generate subtasks with smaller overlapping subgraphs (see Lines 18-22), the subgraph materialization overheads are generated. The smaller $\tau_{time}$ is, the more often task decomposition is triggered and hence more subgraph materialization overheads are generated.

Cost of Task Decomposition. Recall from Algorithm 10 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.

Table 10: Scalability of $A_{time}$

(a) Scalability on Patent

| Thread # | Time (sec) | RAM (GB) | Disk (GB) |
|----------|------------|----------|-----------|
| 1        | 572.84     | 0.30     | 0.00      |
| 2        | 240.64     | 0.30     | 0.00      |
| 4        | 125.55     | 0.30     | 0.00      |
| 8        | 71.59      | 0.31     | 0.00      |
| 16       | 46.53      | 0.32     | 0.00      |
| 32       | 36.66      | 0.34     | 0.02      |

(b) Scalability on Hyves

| Thread # | Time (sec) | RAM (GB) | Disk (GB) |
|----------|------------|----------|-----------|
| 1        | 25.48      | 0.35     | 0.00      |
| 2        | 16.45      | 0.36     | 0.00      |
| 4        | 11.43      | 0.36     | 0.00      |
| 8        | 9.29       | 0.38     | 0.00      |
| 16       | 7.36       | 0.40     | 0.00      |
| 32       | 7.31       | 0.45     | 0.00      |

(c) Scalability on Enron

| Thread # | Time (sec) | RAM (GB) | Disk (GB) |
|----------|------------|----------|-----------|
| 1        | 23.49      | 0.30     | 0.00      |
| 2        | 21.32      | 0.30     | 0.00      |
| 4        | 11.38      | 0.29     | 0.00      |
| 8        | 8.32       | 0.30     | 0.00      |
| 16       | 7.40       | 0.30     | 0.00      |
| 32       | 7.40       | 0.30     | 0.00      |

Table 11: Kernel Expansion in G-thinker

(a) Top-1 Kernel on YouTube (Effect of $\tau_{size}$)

| $\tau_{size}$ | Time (sec) | #Results | #Maximal |
|---------------|------------|----------|----------|
| 20            | 572.83     | 11       | 11       |
| 19            | 649.42     | 12       | 11       |
| 18            | 769.91     | 20       | 15       |
| 17            | 903.47     | 53       | 39       |

(b) Top-1 Kernel on YouTube (Effect of $\gamma$)

| $\gamma$ | Time (sec) | #Results | #Maximal |
|----------|------------|----------|----------|
| 0.85     | 769.91     | 20       | 15       |
| 0.8 > 1 hr (cut) | N/A | N/A     | N/A     |

Table 12: Mining vs. Subgraph Materialization on Patent

| Time (sec) | Job Time | Total Task Mining Time | Total Subgraph Materialization Time | Mining-v.s.-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               | 0.04                               | 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                                |

Our tests show that the additional time spent on task materialization is not significant compared with the actual mining workloads. For example, Table 12 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. Tables 13 and 14 show the profiling results on YouTube and Hyves where we observe similar results and hence conclusion.

**Quick+ vs. Quick.** We have compared our Quick+ with the original Quick algorithm on all the datasets in the single-threaded setting, the results of which are reported in Table 15 where we can observe that Quick+ improves over Quick for up to over 4× w.r.t. running time.

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.

In terms of how the costs of different phases of Quick+ distribute, we consider 4 important phases related to pruning rules: (1) the check by lookahead pruning, (2) the check by cover-vertex pruning, (3) the check by critical-vertex pruning, and (4) the check by lower- and upper-bound pruning. The results are shown in Table 16 for 6 graphs, where we can see that cover-vertex pruning and critical-vertex pruning consumes a lot of the time, while the other two prunings are very fast. However, our test shows that it is still well worth to conduct cover-vertex pruning and critical-vertex pruning as otherwise, the increased search space adds significantly more time to the overall mining than the cost needed by the pruning rule checking.

**9. CONCLUSION**

This paper proposed an algorithm-system codeignition 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.

**Acknowledgment.** This work was supported by NSF OAC-1755464, NSF DGE-1723250 and the NSERC of Canada. Guimu Guo acknowledges financial support from the Alabama Graduate Research Scholars Program (GRSP) funded through the Alabama Commission for Higher Education and administered by the Alabama EPSCoR.

**10. REFERENCES**

[1] Code of the BigData 2018 Paper on Large Quasi-Clique Mining. https://github.com/Beginner1010/topk-quasi-clique-enumeration

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

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

[4] J. Abello, M. G. C. Resende, and S. Sudarsky. Massive quasi-clique detection. In *LATIN*, volume 2286 of * Lecture Notes in Computer Science*, pages 598–612. Springer, 2002.

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

[6] V. Batagelj and M. Zaversnik. An o(m) algorithm for cores decomposition of networks. *CoRR*, cs.DS/0310049, 2003.

[7] D. Berlowitz, S. Cohen, and B. Kimmel. Efficient enumeration of maximal k-plexes. In T. K. Sellis, S. B. Davidson, and Z. G. Ives, editors, *SIGMOD*, pages 431–444. ACM, 2015.

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

[9] M. Brunato, H. H. Hoos, and R. Battiti. On effectively finding maximal quasi-cliques in graphs. In *International conference on learning and intelligent optimization*, pages 41–55. Springer, 2007.
[10] D. Bu, Y. Zhao, L. Cai, H. Xue, X. Zhu, H. Lu, J. Zhang, S. Sun, L. Ling, N. Zhang, et al. Topological structure analysis of the protein–protein interaction network in budding yeast. Nucleic acids research, 31(9):2443–2450, 2003.

[11] L. Chang, J. X. Yu, L. Qin, X. Lin, C. Liu, and W. Liang. Efficiently computing k-edge connected components via graph decomposition. In K. A. Ross, D. Srivastava, and D. Papadí, editors, SIGMOD, pages 205–216. ACM, 2013.

[12] Y. H. Chou, E. T. Wang, and A. L. P. Chen. Finding maximal quasi-cliques containing a target vertex in a graph. In DATA, pages 5–15. SciTePress, 2015.

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

[14] P. Conde-Cespedes, B. Ngom, and E. Viennot. 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):20, 2018.

[15] A. Conte, D. Firmani, C. Mordente, M. Patrignani, and R. Torlone. Fast enumeration of large k-plexes. In SIGKDD, pages 115–124. ACM, 2017.

[16] A. Conte, T. D. Matteis, D. D. Sensi, R. Grossi, A. Marino, and L. Versari. D2K: scalable community detection in massive networks via small-diameter k-plexes. In Y. Guo and F. Farooq, editors, SIGKDD, pages 1272–1281. ACM, 2018.

[17] W. Cui, Y. Xiao, H. Wang, Y. Lu, and W. Wang. Online search of overlapping communities. In K. A. Ross, D. Srivastava, and D. Papadí, editors, SIGMOD, pages 277–288. ACM, 2013.

[18] W. Fan, R. Jin, M. Liu, P. Lu, X. Luo, R. Xu, Q. Yin, W. Yu, and J. Zhou. Application driven graph partitioning. In SIGMOD, 2020.

[19] J. Hopcroft, O. Khan, B. Kulis, and B. Selman. Tracking evolving communities in large linked networks. Proceedings of the National Academy of Sciences, 101(suppl 1):5249–5253, 2004.

[20] H. Hu, X. Yan, Y. Huang, J. Han, and X. J. Zhou. Mining coherent dense subgraphs across massive biological networks for functional discovery. Bioinformatics, 21(suppl 1):i213–i221, 2005.

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

[22] R. J. B. Jr. Efficiently mining long patterns from databases. In L. M. Haas and A. Tiwary, editors, SIGMOD, pages 85–93. ACM Press, 1998.

[23] P. Lee and L. V. S. Lakshmanan. Query-driven maximum quasi-clique search. In SDM, pages 522–530. SIAM, 2016.

[24] J. Li, X. Wang, and Y. Cui. Uncovering the overlapping community structure of complex networks by maximal cliques. Physica A: Statistical Mechanics and its Applications, 415:398–406, 2014.

[25] G. Liu and L. Wong. Effective pruning techniques for mining quasi-cliques. In W. Daelemans, B. Goethals, and K. Morik, editors, ECML/PKDD, volume 5212 of Lecture Notes in Computer Science, pages 33–49. Springer, 2008.

[26] C. Lu, J. X. Yu, H. Wei, and Y. Zhang. Finding the maximum clique in massive graphs. Proc. VLDB Endow., 10(11):1538–1549, 2017.

[27] B. Lyu, L. Qin, X. Lin, Y. Zhang, Z. Qian, and J. Zhou. Maximum biclique search at billion scale. Proc. VLDB Endow., 13(9):1359–1372, 2020.