The VLDB Journal
https://doi.org/10.1007/s00778-021-00712-2

REGULAR PAPER

Parallel mining of large maximal quasi-cliques

Jalal Khalil[1] · Da Yan[1] · Guimu Guo[1] · Lyuheng Yuan[1]

Received: 4 March 2021 / Revised: 13 August 2021 / Accepted: 19 October 2021

Abstract
Given a user-specified minimum degree threshold γ, a γ-quasi-clique is a subgraph where each vertex connects to at least γ fraction of the other vertices. Quasi-clique is a natural definition for dense structures, so finding large and hence statistically significant quasi-cliques is useful in applications such as community detection in social networks and discovering significant biomolecule structures and pathways. However, mining maximal quasi-cliques is notoriously expensive, and even a recent algorithm for mining large maximal quasi-cliques is flawed and can lead to a lot of repeated searches. This paper proposes a parallel solution for mining maximal quasi-cliques that is able to fully utilize CPU cores. Our solution utilizes divide and conquer to decompose the workloads into independent tasks for parallel mining, and we addressed the problem of (i) drastic load imbalance among different tasks and (ii) difficulty in predicting the task running time and the time growth with task-subgraph size, by (a) using a timeout-based task decomposition strategy, and by (b) utilizing a priority task queue to schedule long-running tasks earlier for mining and decomposition to avoid stragglers. Unlike our conference version in PVLDB 2020 where the solution was built on a distributed graph mining framework called G-thinker, this paper targets a single-machine multi-core environment which is more accessible to an average end user. A general framework called T-thinker is developed to facilitate the programming of parallel programs for algorithms that adopt divide and conquer, including but not limited to our quasi-clique mining algorithm. Additionally, we consider the problem of directly mining large quasi-cliques from dense parts of a graph, where we identify the repeated search issue of a recent method and address it using a carefully designed concurrent trie data structure. Extensive experiments verify that our parallel solution scales well with the number of CPU cores, achieving 26.68× runtime speedup when mining a graph with 3.77M vertices and 16.5M edges with 32 mining threads. Additionally, mining large quasi-cliques from dense parts can provide an additional speedup of up to 89.46×.

Keywords Quasi-clique · Parallel · Divide and conquer · T-thinker

1 Introduction

Quasi-clique is a natural generalization of clique (i.e., complete subgraph) useful in various applications, such as finding protein complexes or biologically relevant functional groups [2,6,8,30], and social communities [21,25] that can be cyber-criminals [40] and botnets [37,40].

Mining maximal quasi-cliques is notoriously expensive [36], and the state-of-the-art algorithms [26,33,58] were only tested on small graphs. In fact, [36] showed that even the problem of detecting whether a given quasi-clique in a graph is maximal is NP-hard. Recently, two efforts aim to scale quasi-clique mining to bigger graphs. (1) Our conference version in PVLDB 2020 [19] proposed a distributed divide-and-conquer solution that is able to fully utilize CPU cores. (2) Sanei-Mehri et al. [36] propose to directly mine large quasi-cliques, by first mining dense parts that are faster to find, and then expanding these "kernels" to generate the desired large quasi-cliques. This approach is faster than directly mining large quasi-cliques from the original graph. However, we find that the expansion-searches from different

Jalal Khalil, Da Yan and Guimu Guo have contributed equally to this work.

✉ Da Yan
yanda@uab.edu

Jalal Khalil
jalalk@uab.edu

Guimu Guo
guimuguo@uab.edu

Lyuheng Yuan
lyuan@uab.edu

1 Department of Computer Science, University of Alabama at Birmingham, Birmingham, AL 35233, USA

Published online: 26 November 2021

kernels have a lot of overlaps in their search spaces, causing wasted computations.

To make our solution more accessible to an average user without access to a distributed cluster, in this paper, (i) we first design a general-purpose programming framework called T-thinker for writing divide-and-conquer programs for parallel execution in a multi-core computer and then implement our Quick+ [19] algorithm for mining maximal γ-quasi-cliques on top; (ii) we also improve the algorithm of [36] for mining large maximal γ-quasi-cliques, by eliminating redundant searches using a novel concurrent trie data structure that tracks the mining progress for all threads.

The contributions of this work are listed as follows:

– We implement the efficient parallel quasi-clique mining algorithm proposed in our conference version into a single-machine environment more accessible to an average end user. The new parallel solution inherits all desirable features such as the prioritized scheduling of long-running tasks for mining and decomposition, and a timeout mechanism for task decomposition to avoid stragglers.
– Our parallel solution is built on top of a new general-purpose parallel framework, T-thinker, for divide-and-conquer algorithms in general, including the maximal quasi-clique mining problem this paper studies.
– For kernel-expansion-based approach to mining large dense subgraphs [36], we propose a novel concurrent trie data structure that tracks the mining progress for all mining threads to avoid redundant searches. We remark that even in a serial kernel-expansion algorithm, our trie tracking method is still essential in avoiding redundant searches. Our novel trie structure also supports concurrent access which is essential to enable efficient parallel kernel expansions.

Before our solution, parallel dense subgraph mining was not easy: [36] made 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.” This work breaks new ground by providing such a general efficient programming framework called T-thinker and provide redundancy-avoidance solution for parallel kernel-expansion algorithms that mines large dense subgraphs.

The efficiency of our parallel solution has been extensively verified over various real graph datasets. When running with 32 mining threads, we are able to obtain 26.88× speedup when mining 0.89-quasi-cliques on the *Patent* graph with 3.77M vertices and 16.5M edges: the total serial mining time of 25,369 s are computed by our parallel solution in 943.86 s. Additionally, mining large quasi-cliques from dense kernels can provide an additional speedup of up to 89.46×.

The rest of this paper is organized as follows. Section 2 reviews those related work closely related to the mining of quasi-cliques and other pseudo-clique structures, as well as recent parallel solutions to dense subgraph mining. Section 3 formally defines our notations, the general divisible algorithmic framework for dense subgraph mining which is also adopted by our Quick+, and which is amenable to parallelization in T-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 describes the design of our T-thinker framework, including how it prioritizes long-running tasks for execution. Section 6 then outlines our Quick+ algorithm, and Sect. 7 presents its adaptation on T-thinker including a timeout-based task decomposition for load balancing. Section 8 then reviews the kernel-expansion approach to find large dense subgraphs, introduces the design of our concurrent trie data structure, and presents how it is combined with our Quick+ algorithm to avoid redundant searches. Finally, Sect. 9 reports our experiments and Sect. 10 concludes this paper.

2 Related work

Algorithms for Quasi-Clique Mining A few seminal works devised branch-and-bound subgraph searching algorithms for mining quasi-cliques, such as Crochet [22,33] and Cocain [58] which finally led to the Quick algorithm [26] that integrated all previous search space pruning techniques and added new effective ones. However, we found that some pruning techniques are not fully utilized by Quick, and Quick does not handle a few boundary cases properly, which led to our improved algorithm Quick+ [19]. Quick+ will be introduced in Sect. 6. Yang et al. [57] studied the problem of mining a set of diversified temporal subgraph patterns from a temporal graph, where each subgraph is associated with the time interval that the pattern spans. The dense subgraph definition uses γ-quasi-cliques, and the algorithm is essentially adapted from Quick to include the temporal aspects.

Sanei-Mehri et al. [36] noticed that if γ'-quasi-cliques ($\gamma' > \gamma$) are mined first using Quick which are faster to find, then it is more efficient to expand these “kernels” to generate γ-quasi-cliques than to mine them from the original graph. Their kernel expansion is conducted only on those largest γ'-quasi-cliques extracted by post-processing, in order to find big γ-quasi-cliques as opposed to all of them to keep the running time tractable. However, this work does not fundamentally address the scalability issue: (1) it only studies the problem of enumerating k big maximal quasi-cliques containing kernels rather than all valid ones, and these subgraphs can be clustered in one region (e.g., they overlap on a smaller clique) while missing results on other parts of the data graph, compromising result diversity; (2) the method still needs to first find some γ'-quasi-cliques to grow from and this first step is still using Quick; and (3) the method is not guaranteed to return exactly the set of top-k maximal quasi-cliques. We remark that the kernel-expansion-based acceleration is orthogonal to a parallel dense subgraph mining algorithm and can be easily incorporated; however, as Sect. 8 shall show, the current kernel-expansion solution suffers from heavy redundancy in search space exploration and is inefficient. We address this issue in Sect. 8 using a novel concurrent trie structure.

Other than [36], quasi-cliques have seldom been considered in a big graph setting. Quick [26] was only tested on two small graphs, one with 4,932 vertices and 17,201 edges, and the other with 1,846 vertices and 5,929 edges. In fact, earlier works [22,33,58] 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 only find those quasi-cliques that contain a particular vertex or a set of query vertices [12,14,24] to aggressively narrow down the search space by sacrificing result diversity. Our prior distributed solution [19] implemented on top of G-thinker [49,50] is the first exact parallel algorithm that scales to graphs with tens of millions of vertices and edges, and it achieves a good speedup with the number of CPU cores used.

There is another definition of quasi-clique based on edge density [1,14,32] 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. [7] further consider both vertex degree and edge density. There are also many other definitions of dense subgraphs [5,10,15–17,27,28,60], and they all follow a similar divide-and-conquer algorithmic framework as Quick (c.f. Sect. 3).

Think-Like-A-Task (TLAT) Model The term *think-like-a-task* (or, T-thinker) refers to a task-based programming model to process big data in parallel in order to achieve high CPU utilization [51]. The target problems are usually solved by a divide-and-conquer algorithm with a high time complexity, so that a big task can be decomposed into sub-tasks over smaller data subsets to balance the per-task cost of task creation (e.g., data fetching) and subsequent computation.

This is in contrast to existing big data frameworks such as MapReduce and Pregel-like systems [29,41–46,52,53,55,59] that promote a think-like-a-vertex (TLAV) programming paradigm, whose execution is IO-bound. Those frameworks are only suitable for problems with a low time complexity. For example, in the PageRank application of Google's Pregel, a vertex needs to first receive a value from its in-neighbor, and then simply add it to the current PageRank value.

When applied to problems with a relatively high time complexity, the performance is often a catastrophe. For example, even for triangle counting whose time complexity is only $O(|E|^{1.5})$ where E is the set of edges in an undirected graph, [13] reported that the state-of-the-art MapReduce algorithm for triangle counting uses 1,636 machines and takes 5.33 min on a small graph, on which [13]'s single-threaded algorithm uses less than 0.5 min. Several works including [34,47] have noticed that TLAV frameworks are only efficient for iterative computations where each iteration has $O(n)$ cost and there are constant or at most $O(\log n)$ iterations, giving a time complexity upper bound of $O(n \log n)$ where n is the number of data items. McSherry et al. [9,31] have noticed that existing Big Data systems are comparable and sometimes slower than a single-threaded program since the aggregate IO throughput of disks/network is not beyond that of a single CPU core.

Therefore, a novel compute-intensive framework is essential for dense subgraph mining problems that have a high time complexity. Following our recent paradigm proposal called T-thinker [48,51], we advocate the TLAT computing paradigm and have developed 2 parallel frameworks under this paradigm: (1) G-thinker [49,50] for mining subgraph instances that satisfy certain requirements from a big data graph; and (2) PrefixFPM [54,55] for frequent pattern mining. Since G-thinker already targets a distributed cluster environment, this work develops a single-machine parallel counterpart in order to be accessible to a broader range of end users.

Other Related Work on Parallel Graph Mining Besides the above-mentioned TLAV and TLAT works for parallel graph mining, we hereby review several other interesting related works. 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 (e.g., in the TLAV paradigm) 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. Sect. 4) and dense subgraph mining in general, which adopt divide-and-conquer (and often recursive) algorithms, calling for a new solution for effective task workload partitioning. Besides quasi-clique mining, k-plex mining is another problem that has recently gained a lot of attention [5,15,16,60]. Here, k-plex is another kind of clique relaxation definition which allows each vertex to miss at most $(k-1)$ links to the other vertices in a dense subgraph. Notably, both [16] and [60] follow the set-enumeration search tree framework to be reviewed in Sect. 3.2, so our solution can also be applied to parallelize their algorithms.

Realizing the limitation of TLAV for mining problems that operate on subgraphs rather than individual vertices, many parallel and distributed systems were designed to directly operate on subgraphs. However, while their subgraph-based APIs are more convenient to write subgraph mining algorithms, their execution engines are still IO-bound. Specifically, NScale [35] constructs candidate subgraphs using multiple rounds of MapReduce, leading to large amounts of data shuffling and HDFS (Hadoop Distributed File System) read/write overheads. Arabesque [38] constructs and examines subgraphs iteratively from small ones to large ones, which materializes a huge number of candidate subgraphs. G-Miner [11] has a poor task scheduling scheme that causes huge numbers of subgraph-tasks being buffered on disks when an input graph is large and uses a list for remote vertex caching which is a bottleneck of task concurrency. RStream [39] is a single-machine out-of-core system that utilizes relational joins for evaluation and is thus not very multi-core friendly. Nuri [23] is for single-threaded execution and may become IO-bound. Please refer to Sect. II of [50] for a detailed comparison of these systems with G-thinker which is the winner. Our T-thinker framework designed in this paper inherits the benefits of G-thinker (c.f. Table I of [50]).

3 Preliminaries

This section first prepares the notations for subsequent algorithmic description and formally defines our problem. We then provide the intuition of the set-enumeration tree framework for search space partitioning.

3.1 Notations and problem definition

Notations. Table 1 summarizes the set of notations used throughout this paper for a quick reference.

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 neighbor set 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 $N_{V'}(v) = \{u \,|\, (u, v) \in E, u \in V'\}$, i.e., $N_{V'}(v)$ is the set of v's neighbors inside V', and we also define $d_{V'}(v) = |N_{V'}(v)|$.

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

Table 1 Notations

Symbol	Meaning
$G = (V, E)$	Input graph G with vertex (edge) set V (E)
$V(G)$	Vertex set of a graph G
$ext(S)$	Set of vertices that can extend a vertex set S into a valid γ-quasi-clique
T_S	Set-enumeration subtree rooted at a node with vertex set S
$G(S)$	Subgraph of G induced by a vertex set S
$\|S\|$	Number of elements in set S
v	A vertex, or a singleton vertex set $\{v\}$
$N(v)(, N_S(v))$	Set of neighbor vertices of v (inside S)
$d(v)(, d_S(v))$	Degree of vertex v (inside S)
$\mathbb{B}(v)$	Set of vertices in G within two hops from v
$\mathbb{B}_{>v}(v)$	Set of vertices in $\mathbb{B}(v)$ that are ordered after v
$C_S(u)$	Set of vertices in $ext(S)$ that are covered by u
$U_S(, L_S)$	Upper (lower) bound U_S (L_S) on the number of vertices in $ext(S)$ to be added to S to form a valid γ-quasi-clique
γ, τ_{size}	A valid γ-quasi-clique has at least τ_{size} vertices
τ_{big}	The minimum number of vertices in $ext(S)$ for a task to be considered "big"
τ_{time}	Task timeout duration threshold
n_θ	Number of computing threads in T-thinker
data_array	Data items loaded by T-thinker's main thread
next_position	Next position in *data_array* to spawn a new task
task_queue	T-thinker task queue (including Q_{big} and Q_{reg})
Q_{reg}, Q_{big}	Regular task queue, big task queue
C_{reg}, C_{big}	Capacity (maximum # of tasks) of Q_{reg}, Q_{big}
$\tau_{reg}^{min}, \tau_{big}^{min}$	Task refill is triggered when # of tasks in Q_{reg}, Q_{big} is less than or equal to τ_{reg}^{min}, τ_{big}^{min}
$n_{reg}^{file}, n_{big}^{file}$	# of tasks in Q_{reg}, Q_{big} to be spilled as a file
$\mathcal{L}_{reg}, \mathcal{L}_{big}$	File lists of tasks spilled from Q_{reg}, Q_{big}

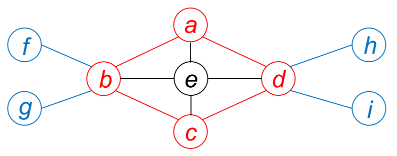


Fig. 1 An illustrative graph

Given two vertices $u, v \in V$, we define $\delta(u, v)$ as the number of edges on the shortest path between u and v. We call G as connected if $\delta(u, v) < \infty$ for any $u, v \in V$. We further define $N_k(v) = \{u \,|\, \delta(u, v) = k\}$ 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 but not in $(k-1)$ hops. Then, we have $N_0(v) = v$ and $N_1(v) = N(v)$, and $N_k^+(v) = N_0(v) + N_1(v) + \ldots + N_k(v)$. For 2-hop neighbors, we define $B(v) = N_2(v)$ and $\mathbb{B}(v) = N_2^+(v)$.

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

Problem Definition. We formally define our problem:

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

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

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

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

In the literature of dense subgraph mining, researchers usually only strive to find big dense subgraphs, such as the largest dense subgraph [15,24,27,28], the top-k largest ones [36], and those larger than a predefined size threshold [15,16,26]. There are two reasons. (i) Small dense subgraphs are common and thus statistically insignificant and not interesting. For example, a single vertex itself is a quasi-clique for any γ, and so is an edge with its two end-vertices. (ii) The number of dense subgraphs grows exponentially with the graph size and is thus intractable unless we focus on finding large ones. In fact, it has been shown that even the problem of detecting if a given quasi-clique is maximal is NP-hard [36]. Following [26], we use a minimum size threshold τ_{size} to return only large quasi-cliques as described below.

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

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

3.2 Subgraph mining by set-enumeration search tree

In a general pseudo-clique mining problem (such as our quasi-clique problem), the giant search space of a graph $G = (V, E)$, i.e., V's power set, can be organized as a set-enumeration tree [26]. Figure 2 shows the set-enumeration tree T for a graph G with four vertices $\{a, b, c, d\}$ where we assume an order $a < b < c < d$. Each tree node represents a vertex set S, and only those vertices larger than the largest vertex in S are used to extend S. For example, in Fig. 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.

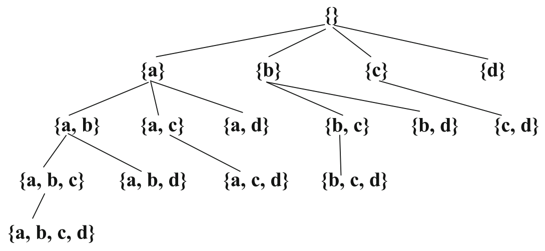


Fig. 2 Set-enumeration tree

Let us denote T_S as the subtree of 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, for any pseudo-clique Q found in T_S, $Q \supseteq S$. We represent the task of mining T_S as a pair $\langle S, ext(S) \rangle$, where S is the set of vertices assumed to be already included, and $ext(S) \subseteq (V - S)$ keeps those vertices that can extend S further into a γ-quasi-clique. As we shall see, many vertices cannot form a γ-quasi-clique together with S and can thus be safely pruned from $ext(S)$, making $ext(S)$ much smaller than $(V - S)$.

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

This set-enumeration approach typically requires post-processing to remove non-maximal pseudo-cliques from the set of valid pseudo-cliques found [26]. For example, consider Fig. 2: when processing a task that mines $T_{\{b\}}$, vertex a is not considered and thus, the task has no way to determine that $\{b, c, d\}$ is not maximal, even if $\{b, c, d\}$ is invalidated by $\{a, b, c, d\}$ which happens to be a valid pseudo-clique, since $\{a, b, c, d\}$ is processed by the task mining $T_{\{a\}}$. But this post-processing 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* that are summarized in Table 4 of Sect. 9. We show that (1) the running time of tasks spans a wide range, (2) even those tasks with subgraphs of similar size-

Table 2 Features of 10 most expensive tasks on *YouTube*

$\|V\|$	$\|E\|$	Max degree	$\|E\|/\|V\|$	Core #	Task time	Predicted time
2570	72,678	1583	28.28	43	13,033	899.67
3588	82,727	1417	23.06	37	13,407	1128.13
3228	100,177	2,127	31.04	49	13,623	1505.75
2646	75,747	1646	28.63	44	13,893	969.04
2755	78,375	1597	28.45	45	15,011	1,028.77
5074	162,249	2721	31.98	50	15,015	1924.41
3177	101,008	1850	31.80	49	15,267	1521.73
2321	55,094	1320	23.74	38	15,584	529.61
3723	113,828	1849	30.58	46	16,881	1745.78
26,235	694,686	7105	26.48	51	3,645,905	1015.08

Table 3 Features of 10 most expensive tasks on *Patent*

$\|V\|$	$\|E\|$	Max degree	$\|E\|/\|V\|$	Core #	Task time	Predicted time
109	4232	93	38.83	64	729,769	5.53
93	3197	80	34.38	60	1,006,208	3.84
104	3914	88	37.63	64	1,053,326	4.99
95	3332	82	35.07	60	1,083,755	4.07
69	1786	65	25.88	43	1,198,085	1.48
78	2282	69	29.26	48	1,220,241	2.32
72	1950	66	27.08	45	1,411,622	1.75
79	2346	69	29.70	49	1,757,738	2.43
88	2873	75	32.65	55	2,658,704	3.32
76	2167	68	28.51	47	2,878,700	2.11

and degree-related features can have drastically different running time, and hence (3) expensive tasks cannot be effectively predicted using regression models in machine learning.

Specifically, we ran quasi-clique mining using T-thinker where each task is spawned from a vertex v and mines the entire set-enumeration subtree $T_{\{v\}}$ in serial without generating any subtasks. As shall be clear from rules (P1) and (P2) in Sect. 6.3, vertices with low degrees can be pruned using a k-core algorithm, and vertices in $ext(S)$ have to be within $f(\gamma)$ hops from v. Our reported experiments have applied these pruning rules so that (i) low-degree vertices are directly pruned without generating tasks, (ii) the subgraphs have been pruned not to include vertices pruned by (P1) and (P2).

Also, we only report the actual time of mining $T_{\{v\}}$ for each task, not including any system-level overheads for task creation and scheduling, though the latter cost is not a bottleneck. Table 2 (resp. Table 3) shows the task-subgraph features of the top-10 longest-running tasks on *YouTube* with $\tau_{size} = 18$ and $\gamma = 0.9$ (resp. *Patent* with $\tau_{size} = 20$ and $\gamma = 0.89$) including the number of vertices and edges, the maximum and average vertex degrees, the k-core number (aka. degeneracy) of the subgraph, the actual serial mining time on the subgraph, along with the predicted time using support vector regression (SVR). The tasks are listed in ascending order of running time (c.f. Column "Task Time"), and the time unit is millisecond (ms).

In Table 2, the last task takes more than 1 h (3645.9 s) to complete, much longer than all the other tasks. 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 contrast, our T-thinker program with proper task decomposition only takes 17 min and 25 s to complete this job with 32 mining threads.

In Table 3, the last 9 tasks all take more than 1000 s, so unlike *YouTube* with one particularly expensive task, *Patent* has a few of them, so the computing thread that gets assigned most of those tasks will become a straggler. In fact, on both graphs, there are tasks taking less than 1 ms, so the task time spans 8 orders of magnitude! In contrast, our T-thinker program with proper task decomposition only takes 15 min and 19 s to complete this job with 32 mining threads.

Note that in the tables, we already have size- and degree-based features of a task-subgraph, as well as the more advanced feature of subgraph degeneracy that reflects graph density. We have extensively tested the various machine learning models for task-time regression using those subgraph features along with the top-10 highest vertex degrees

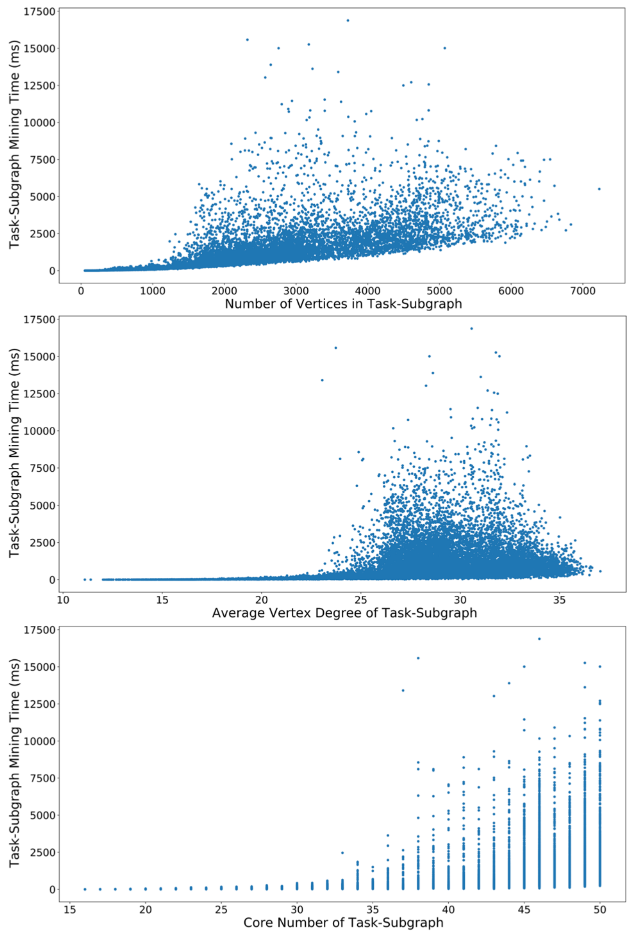


Fig. 3 Subgraph features versus task time on *YouTube*

and top-10 vertex core indices, but none of the models can effectively predict the time-consuming tasks. In both Tables 2 and 3, the last column shows the predicted time using an SVM (RBF kernel) trained with all the task statistics, and we can see that the predicted times are way off the ground truth.

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

To visualize how each subgraph feature impacts the task running time, we plot the impacts of vertex number, average degree, and core number in Fig. 3 for the *YouTube* graph, where we excluded the sole straggler task that takes 3645.9 s which would otherwise flatten other points to near 0 on the y-axis. We 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. Complete plots on both graphs can be found in the technical report [20] of our PVLDB 2020 version.

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 Sect. 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 first in Sect. 5 below.

5 The T-thinker framework

To parallelize our Quick+ algorithm [19], we develop a general-purpose task-based programming framework called T-thinker and implement the parallel Quick+ on top. This section first describes the programming interface of T-thinker and then, introduces the design of its execution engine. Section 6 will then describe our Quick+ algorithm, and Sect. 7 will introduce how we use T-thinker's API to parallelize Quick+.

We remark that the design of T-thinker has its own merit by providing a generic framework to parallelize any algorithms that adopt divide and conquer.

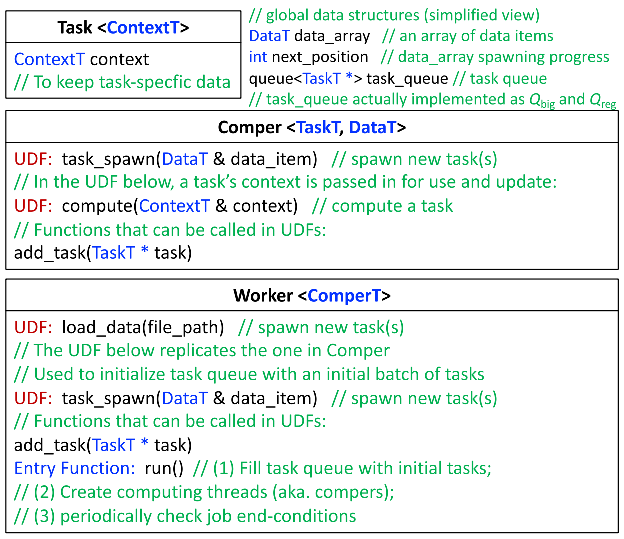
5.1 T-thinker programming interface

T-thinker is written as a set of C++ header files defining some base classes and their virtual functions for users to inherit in their subclasses and to specify the application logic. We call these virtual functions as user-defined functions (UDFs). The base classes also contain C++ template arguments for users to specify with the proper data structures.

Figure 4 shows the API of these base classes along with some global variables that a programmer needs to be aware of when implementing parallel algorithms in T-thinker. We now introduce them as follows.

There are three base classes, *Task*, *Comper* and *Worker*. (1) A *Task* object is the basic unit of computing, with its content specified by type <*ContextT*> which keeps the variables related for task computation. (2) A *Worker* object implements the main thread of a T-thinker program, which creates a pool of n_θ computing threads (aka. **compers**) for computing tasks. (3) A *Comper* object implements a computing thread that keeps fetching and computing tasks.

There are also a few global variables. As Fig. 4 shows, (1) a T-thinker program maintains an array *data_array* of data items that are loaded by the main thread initially from a data file. Tasks are spawned from individual data items, and (2) the global variable *next_position* tracks the progress of



```
Task <ContextT>
ContextT context
// To keep task-specfic data

// global data structures (simplified view)
DataT data_array  // an array of data items
int next_position  // data_array spawning progress
queue<TaskT *> task_queue // task queue
// task_queue actually implemented as Q_big and Q_reg

Comper <TaskT, DataT>
UDF:  task_spawn(DataT & data_item)   // spawn new task(s)
// In the UDF below, a task's context is passed in for use and update:
UDF:  compute(ContextT & context)   // compute a task
// Functions that can be called in UDFs:
add_task(TaskT * task)

Worker <ComperT>
UDF:  load_data(file_path)   // spawn new task(s)
// The UDF below replicates the one in Comper
// Used to initialize task queue with an initial batch of tasks
UDF:  task_spawn(DataT & data_item)   // spawn new task(s)
// Functions that can be called in UDFs:
add_task(TaskT * task)
Entry Function:  run()   // (1) Fill task queue with initial tasks;
// (2) Create computing threads (aka. compers);
// (3) periodically check job end-conditions
```

Fig. 4 The T-thinker programming API

task spawning, i.e., the next position in *data_array* to spawn the next task (all previous data items have spawned tasks). Finally, (3) a task queue *task_queue* is maintained for the computing threads (or, compers) to fetch tasks for computing.

Both *data_array* and *task_queue* are initialized by the main thread (i.e., the *Worker* object) when a job begins, where types <*DataT*> and <*TaskT*> are obtained from the user-defined *Comper* subclass which is specified in *Worker*'s template argument <*ComperT*>.

To use the base classes, users need to subclass *Comper* and *Worker* with their UDFs specified by the application logic; but since *Task*<*ContextT*> has no UDF, users only need to specify the concrete type for template argument <*ContextT*> and to rename the new type using "typedef" for ease of use. For example, in our set-enumeration search tree-based mining introduced in Sect. 3.2, a task that mines the subtree T_S can be denoted by t_S, which keeps content such as S, $ext(S)$ and the subgraph induced by $S \cup ext(S)$ for checking vertex connectivity; so <*ContextT*> should be a class that keeps these fields.

Recall that *Comper* is a class that implements a computing thread which fetches and computes tasks from *task_queue*. A *Comper* subclass requires a programmer to specify the type of a data item, <*DataT*>, and the type of a task, <*TaskT*> (whose <*ContextT*> has been specified). *Comper* provides two UDFs: (i) *task_spawn*(o), where users may create tasks from a data item o (from *data_array*), and call *add_task*(t) to add each created task t to *task_queue*. (ii) *compute*($t.context$), which specifies how a task t (from *task_queue*) is computed; the computation operates on the variables in $t.context$. If a task t is expensive, a user can decompose it into smaller tasks in *compute*(.) and call *add_task*(.) to add them to *task_queue* for parallel processing.

A T-thinker program is initiated by subclassing the *Worker* base class, calling its UDF *load_data*(.) to load data from the input file into *data_array* (and to conduct proper preprocessing), and then calling its *run*() function to execute the mining program.

5.2 Execution engine design overview

In *Worker*::*run*(), **(i)** the main thread actually first fills *task_queue* with C_{reg} tasks by spawning from the first C_{reg} data items in *data_array* (if available), where C_{reg} is capacity of a regular task queue Q_{reg} (see Sect. 5.3 for more details on Q_{reg}), i.e., the maximum number of tasks allowed in Q_{reg} to keep memory consumption bounded. This is exactly why UDF *task_spawn*(.) is replicated from *Comper* to *Worker* again, so that the main thread can call it to spawn initial tasks. The initial task creation from the C_{reg} data items can be conducted using a parallel-for loop (e.g., in OpenMP) to fully utilize multi-cores. Then, **(ii)** the main thread creates n_θ compers to fetch tasks from *task_queue* for concurrent processing. Since *task_queue* has abundant initial tasks, the compers can be kept busy. Finally, **(iii)** the main thread periodically (every 0.1 s) checks the job end-condition and sets an end flag to notify compers to terminate if it finds that all tasks have been processed. Since the main thread sleeps most of the time, its CPU occupancy is low so CPU cores are fully utilized by the compers to compute tasks in parallel.

During the job execution, each comper repeats the following 4 operations: it (1) refills *task_queue* if it finds that *task_queue* is about to become empty, (2) fetches a task t from *task_queue*, (3) calls UDF *compute*($t.context$) to complete its computation and then (4) deletes the task object to release its memory resources. Note that operation (1) keeps *task_queue* with abundant tasks to keep all compers (and hence CPU cores) busy unless there are not enough remaining tasks to refill.

A comper is suspended if there is no task to fetch from *task_queue*, and the T-thinker job terminates only if all compers are suspended. This condition is probed by the main thread every 0.1 s: (i) if it holds, the main thread will set the end flag and then wake up all compers to terminate (as they will see the end flag and exit rather than repeat the 4 operations); otherwise, (ii) if there are tasks in *task_queue*, the main thread will wake up all compers to continue task processing; otherwise, (iii) while *task_queue* is empty, at least one comper is still processing tasks and may add more tasks to *task_queue* due to task decomposition, but since now *task_queue* is empty and need no more compers to fetch tasks, the main thread will suspend itself immediately to wait for next round of probe to occur 0.1 s later.

5.3 Task queues and system parameters

Big versus Regular Tasks It is desirable to schedule those tasks that tend to be long-running (called big tasks) early, so that they can be computed and decomposed earlier to improve load balancing. For example, in our set-enumeration search (c.f. Sect. 3.2), a task $t_S = \langle S, ext(S)\rangle$ tends to be more expensive if $|ext(S)|$ is large. To enable big task prioritization, we implement *task_queue* as two task queues: Q_{big} to hold big tasks, and Q_{reg} to hold regular tasks. Our *Comper* class actually has another UDF *is_bigTask*$(t.context)$ which is specified to return if a task t is big or not (e.g., if $|ext(S)| \geq \tau_{big}$ where τ_{big} is a user-specified threshold). This UDF is used by Comper to decide which queue (Q_{big} or Q_{reg}) to add a task t when *add_task*(t) is called. If users do not specify this UDF, the default implementation returns *false* so all tasks are regular.

Task Spilling and Refill Now, we explain how tasks are added to and fetched from *task_queue* which consists of two queues Q_{big} and Q_{reg}. We require Q_{big} (resp. Q_{reg}) to have a maximum capacity of C_{big} (resp. C_{reg}) tasks, where $C_{big} = 16$ (resp. $C_{reg} = 512$) by default. However, it is possible for a big task to generate many decomposed tasks to be inserted into Q_{big} and Q_{reg}, causing either queue to overflow. To keep the number of in-memory tasks bounded, if Q_{big} (resp. Q_{reg}) is full but a new task is to be inserted, we spill a batch of n_{big}^{file} (resp. n_{reg}^{file}) tasks at the end of Q_{big} (resp. Q_{reg}) as a file to local disk to make room, where $n_{big}^{file} = 4$ (resp. $n_{reg}^{file} = 32$) by default. Note that tasks spilled from Q_{big} (resp. Q_{reg}) are written to the disk (and loaded back later) in batches of size n_{big}^{file} (resp. n_{reg}^{file}) each, to achieve serial disk IO. We use a file list $\mathcal{L}_{big}$ (resp. $\mathcal{L}_{reg}$) to track those files spilled from Q_{big} (resp. Q_{reg}) to be loaded back to Q_{big} (resp. Q_{reg}) later when it needs a task refill. Task spilling is automatically handled by the *add_task*(t) function in T-thinker.

Also, whenever a comper that checks Q_{big} (resp. Q_{reg}) for task fetching finds that there are less than τ_{big}^{min} (resp. τ_{reg}^{min}) tasks in Q_{big} (resp. Q_{reg}), it will refill tasks either from a task file, or by spawning new tasks. By default, $\tau_{big}^{min} = 8$ (resp. $\tau_{reg}^{min} = 128$). We shall return to the refill details later.

Since Q_{big} (resp. Q_{reg}) needs to be refilled from the head of the queue and to spill tasks from the tail by all compers, we implement Q_{big} (resp. Q_{reg}) as a deque protected by a mutex. Also, to spill tasks, a comper first locks Q_{big} (resp. Q_{reg}) and fetches n_{big}^{file} (resp. n_{reg}^{file}) tasks at the tail of the queue and then unlocks the queue so that other compers can access it; the comper then serializes the fetched tasks to a file and deletes those tasks from memory, without holding queue lock.

Recall that before a comper fetches a task, it first checks if *task_queue* needs a refill. The sources for task refill include $\mathcal{L}_{big}$, $\mathcal{L}_{reg}$ and *data_array* (for spawning new tasks), and we always prioritize $\mathcal{L}_{big}$ and $\mathcal{L}_{reg}$ before *data_array* for refill so that the number of tasks kept on disk is minimized (i.e., new tasks are spawned only if both $\mathcal{L}_{big}$ and $\mathcal{L}_{reg}$ are empty). Also, a comper always checks Q_{big} for task fetching, and only if Q_{big} is empty will the comper fetch a task from Q_{reg}. We next detail the steps for a comper to fetch a task.

Task Fetching by a Comper There are 6 cases. A comper first **(1)** checks if Q_{big} has less than τ_{big}^{min} tasks; if so, it refills Q_{big} by a task file from $\mathcal{L}_{big}$ and obtains a big task from Q_{big} for computation. **(2)** But if $\mathcal{L}_{big}$ is found to be empty when the comper is conducting refill, it will just fetch a big task from Q_{big} to compute. **(3)** if Q_{big} has no task to fetch, the comper will then check if Q_{reg} has less than τ_{reg}^{min} tasks; if so, it refills Q_{reg} by a task file from $\mathcal{L}_{reg}$ and obtains a task from Q_{reg} for computation. **(4)** But if $\mathcal{L}_{reg}$ is found to be empty when the comper is conducting refill, it will then spawn up to n_{spawn} ($= 32$ by default) tasks from *data_array* for refill (if enough unspawned data items are available). During this process, if any spawned task t is found to be big, the task spawning stops and the comper fetches t to compute. This avoids generating many big tasks out of one refill (that may cause cascaded task spilling from Q_{big}), since a big task may itself be decomposed into many big subtasks to be added into Q_{big}. **(5)** Otherwise, the comper should have spawned n_{spawn} regular tasks and refilled them into Q_{reg}, it then fetches a task from Q_{reg} to compute. **(6)** if there is no task to fetch in Q_{reg} (e.g., there is no more task to spawn in *data_array*), the comper then suspends itself to be awakened by the main thread during its next probe.

The default system parameters described so far (which are illustrated in Fig. 5) have been extensively tuned to consistently deliver the near-optimal performance across all datasets we tested. For example, a comper can only refill $n_{big}^{file} = 4$ tasks to Q_{big} with capacity $C_{big} = 16$ at a time, while a comper can only refill $n_{reg}^{file} = 32$ or $n_{spawn} = 32$ tasks to Q_{reg} with capacity $C_{reg} = 512$ at a time. This makes sense since multiple compers may detect the need of a queue refill due to low queue occupancy, and simultaneously conduct task refill. Our scheme ensures that even in such cases, the collectively refilled tasks are not likely to cause task spilling which would otherwise cause the thrashing issue where some task gets repeatedly refilled and spilled.

5.4 Discussion of T-thinker contributions

Compared with G-thinker [50] which is tailor-made for subgraph finding, T-thinker has a more general API for parallelizing any algorithm that adopts divide and conquer. Also, T-thinker only requires a single machine for execution rather than a computer cluster as in G-thinker, so T-thinker is accessible to more users.

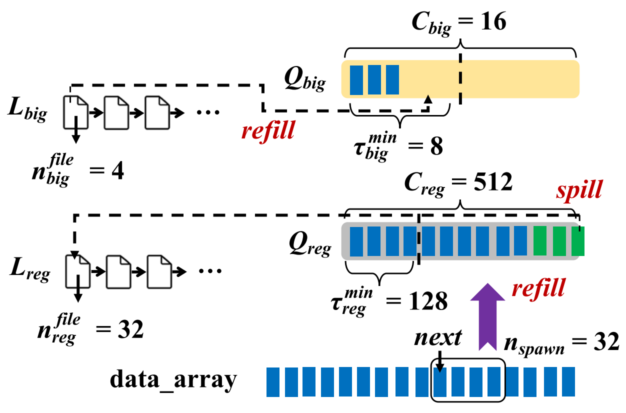


Fig. 5 Illustration of T-thinker's Default System Parameters

As we shall see in Sect. 9, T-thinker is memory-efficient and a typical modern server with 64 GB memory is more than sufficient in vast majority of the graph datasets. This is thanks to T-thinker's task spilling and refill design that keeps a low memory usage by tasks; in fact, most memory space is used to hold the large input graph. While a distributed system like G-thinker can partition a large input graph using distributed memory, most modern graphs can already fit in the memory of a server. Moreover, quasi-clique mining is very expensive and graphs larger than the memory capacity of a server are not computationally tractable anyway, which is in contrast to less expensive problems such as triangle counting and maximum clique finding that G-thinker also targets [50], so T-thinker is a proper and sufficient platform for mining maximal quasi-cliques.

Finally, while G-thinker also uses separate queues for big and regular tasks for effective prioritization of big tasks, T-thinker's design makes proper adaptions to be efficient in a single-machine multi-core setting. Specifically, since the slower machine-wise data and task transmissions are totally eliminated, a task does not need to wait for remote data, so the UDF *compute*(.) is just called once for each task to complete task computation, rather than called for multiple iterations as in G-thinker to allow data waiting between iterations. **Secondly,** instead of letting compers spawn tasks from local data items as in G-thinker, T-thinker lets the main thread generates a pool of C_{reg} tasks initially to ensure sufficient tasks to keep compers busy. Notably, while initial tasks are generated by one main thread, it can use OpenMP's parallel for-loop to utilize multiple CPU cores, so there is no sacrifice in parallelism. **Thirdly,** since tasks are generated and processed in a faster speed than in G-thinker, the task queues need to support higher parallelism, which leads to some changes in the system design and parameter choice. For example, in G-thinker, a thread refills $C_{reg}/3$ (resp. $C_{big}/3$) tasks to Q_{big} (resp. Q_{reg}) at a time, but we find that this design will slow down T-thinker's processing. This is because when the task queue is locked by a comper to refill a relatively large task batch, other compers cannot fetch tasks from the queue. To prevent undesirable comper stall, T-thinker uses a much smaller batch size for task spilling and refill (i.e., n_{big}^{file} or n_{reg}^{file}) compared with queue capacity C_{big} or C_{reg}, so that the queue locking time is significantly reduced, while even if multiple compers detect the need of refill, their consecutive refill operations are not likely to cause task thrashing.

6 The Quick+ Algorithm

The next two sections present our Quick+ algorithm [19,20] for maximal γ-quasi-clique mining, and how to parallelize it using T-thinker. Notably, to tackle the load balancing issue described earlier in Sect. 4, a timeout mechanism is proposed in our T-thinker algorithm to break down long-running tasks for parallel execution.

6.1 Pruning rules

The key to an efficient set-enumeration search is the pruning strategies that are applied to remove entire branches from consideration [4]. Without pruning, the search space is exponential. Quick [26] uses the most complete set of pruning rules for mining maximal quasi-cliques. Our PVLDB 2020 conference version [19] further improves Quick with new pruning rules and fixes some missed boundary cases. Since our main focus is to parallelize Quick+ on T-thinker, and the complete Quick+ has a lot of details, this section only briefly overviews Quick+ and presents those rules necessary for understanding our parallelization. The complete details and proofs are in our Quick+ technical report [20].

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

We consider two types of pruning rules: **Type I: Pruning ext(S)**: in such a rule, if a vertex $u \in ext(S)$ satisfies certain conditions, u can be pruned from $ext(S)$ since there must not exist a vertex set S' such that $(S \cup u) \subseteq S' \subseteq (S \cup ext(S))$ and $G(S')$ is a γ-quasi-clique. **Type II: Pruning S**: in such a rule, if a vertex $v \in S$ satisfies certain conditions, there must not exist a vertex set S' such that $S \subseteq S' \subseteq (S \cup ext(S))$ and $G(S')$ is a γ-quasi-clique, and thus, there is no need to extend S further. Type-II pruning invalidates the entire T_S. A variant of Type-II pruning 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 [20] summarized below as (P1)–(P7), where each rule either belongs to Type I, or Type II, or both.

(P1) Graph-Diameter-Based Pruning Theorem 1 of [33] defines the diameter upper bound of a γ-quasi-clique as a function $f(\gamma)$, and $f(\gamma) = 2$ if $\gamma \geq 0.5$. Without loss of generality, we use 2 as the diameter upper bound in our algorithm description, but it is straightforward to generalize to the case $\gamma < 0.5$ by considering vertices $f(\gamma)$ hops away. Since a vertex $u \in ext(S)$ must be within 2 hops from any $v \in S$, we have $ext(S) \subseteq \bigcap_{v \in S} \mathbb{B}(v)$. This is a Type-I pruning rule since if $u \notin \bigcap_{v \in S} \mathbb{B}(v)$, u can be pruned from $ext(S)$.

(P2) Size-Threshold-Based Pruning A valid γ-quasi-clique $Q \subseteq V$ should contain at least τ_{size} vertices (i.e., $|Q| \geq \tau_{size}$), and therefore for any $v \in Q$, its degree $d(v) \geq \lceil \gamma \cdot (|Q|-1) \rceil \geq \lceil \gamma \cdot (\tau_{size}-1) \rceil$. We thus can prune any vertex u with $d(u) < \lceil \gamma \cdot (\tau_{size}-1) \rceil$ from G. Let $k = \lceil \gamma \cdot (\tau_{size}-1) \rceil$, then this rule shrinks G into its k-core, i.e., the maximal subgraph of G where every vertex has degree $\geq k$. The k-core of a graph $G = (V, E)$ can be computed in $O(|E|)$ time using a peeling algorithm [3]. We thus shrink a graph G into its k-core before mining, which effectively reduces the search space.

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

Three groups of pruning rules utilize these degrees: (i) degree-based pruning that solely uses the degrees of a vertex itself, (ii) upper-bound-based pruning and (iii) lower-bound-based pruning that look at the degrees of multiple (or even all) vertices in S and $ext(S)$. Each of the three groups contains two rules: one Type-I rule and one Type-II rule. Please refer to [20] for these rules.

(P4 & P5) Upper- and Lower-Bound-Based Pruning For each task $t_S = (S, ext(S))$, Quick [26] defined an upper bound U_S (resp. lower bound L_S) on the number of vertices in $ext(S)$ to be added to S to form a γ-quasi-clique, based on the above-mentioned degrees. We found that additional Type-II pruning not considered by Quick can happen when computing U_S (resp. L_S), and please see [20] for details.

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

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

To utilize $C_S(u)$ for pruning, we put vertices of $C_S(u)$ after all the other vertices in $ext(S)$ when checking the next level in the set-enumeration tree (see Fig. 2) and only check until all vertices of $ext(S) - C_S(u)$ are examined (i.e., the extension of S using $V' \subseteq C_S(u)$ is pruned). To maximize the pruning effectiveness, we find $u \in ext(S)$ that tends to have a large $|C_S(u)|$. We refer readers to [20] for the formula of computing $C_S(u)$.

As a degenerate special case, initially when $S = \emptyset$, we have $C_S(u) = N(u)$, i.e., we only need to find u as the vertex with the maximum degree. Note that for any γ-quasi-clique Q constructed out of vertices in $C_S(u) = N(u)$, adding u to Q still produces a γ-quasi-clique. We find u as the vertex with the maximum degree after k-core pruning by (P2) to avoid selecting a high-degree vertex without pruning power (e.g., center of a star).

6.2 The iterative pruning sub-procedure

Iterative Nature of Type-I Pruning Recall that Type-I pruning shrinks $ext(S)$, which will reduce vertex degrees such as $d_{ext(S)}(v)$ of some $v \in S$, which in turn will update bounds U_S and L_S that are defined on the degrees. This essentially means that Type-I pruning is iterative: each pruned vertex $u \in ext(S)$ may change degrees and bounds, which affects the various pruning rules including Type-I pruning rules themselves; these Type-I pruning rules will thus be checked again, and new vertices in $ext(S)$ may be pruned due to Type-I pruning, triggering another iteration. As this process repeats, U_S and L_S become tighter until no more vertex can be pruned from $ext(S)$, which has 2 cases:

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

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

Algorithm 1 Iterative Bound-Based Pruning

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

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

Algorithm 1 returns a Boolean tag indicating whether S's extensions (but not S itself) are pruned, and the input argument $ext(S)$ is passed as a reference and may be shrunk due to Type-I pruning by the function.

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

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

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

Then, Lines 4–6 apply the critical-vertex pruning of (P6). We find all critical vertices in S and move their neighbors from $ext(S)$ to S. Such movement will update degrees and bounds which are then updated in Line 7 if a critical vertex is ever found. Similar to Line 3, Line 7 may trigger Type-II pruning so that the function returns $true$ directly. Since the updates may generate new critical vertices in the updated S, we actually loop Lines 4–7 until there is no more critical vertex in S.

Quick+ also performs the following pruning: if any neighbor $u \in ext(S)$ of a critical vertex $v_1 \in S$ is found to be more than 2 hops away from another vertex $v_2 \in S$, or from another neighbor $u' \in ext(S)$ of v_1, then Type-II pruning is triggered so that the function returns $true$ directly. This is because neither $\{u, v_1, v_2\}$ nor $\{u, v_1, u'\}$ can co-exist in a valid quasi-clique, but since v_1 is a critical vertex, u and u' have to be included into S, leading to a contradiction.

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

Next, consider Part 2 on Type-II pruning as in Lines 8–10. These lines assume that Type-II rules prune the entire subtree T_S if any vertex $v \in S$ satisfies the rule conditions. Finally, Part 3 on Type-I pruning checks every $u \in ext(S)$ and tries to prune u using a Type-I pruning condition as shown in Lines 11–14, which may create new pruning opportunities for next iteration.

6.3 The recursive main mining algorithm

Algorithm 2 shows our Quick+ main algorithm for mining valid quasi-cliques extended from S (including $G(S)$ itself). This algorithm is recursive (see Line 21) and starts by calling *recursive_mine*(v, $\mathbb{B}_{>v}(v)$, γ, τ_{size}) on every $v \in V$ where $\mathbb{B}_{>v}(v)$ denotes those vertices in $\mathbb{B}(v)$ (i.e., within 2 hops from v) that are ordered after v (e.g., in Fig. 2, we have the order $a < b < c < d$). Note that according to Fig. 2, we should not consider the other vertices in $\mathbb{B}(v)$ to avoid double counting.

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

Algorithm 2 keeps a Boolean tag $\mathcal{T}_{Q\_found}$ to return (see Line 26), which indicates whether some valid quasi-clique

Algorithm 2 Extending S for Valid Quasi-Cliques

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

Q extended from S (but $Q \neq S$) is found. Line 1 initializes $\mathcal{T}_{Q\_found}$ as $false$, but it will be set as $true$ if any valid quasi-clique Q is later found.

Algorithm 2 examines S, and it decomposes this problem into sub-problems of examining $S' = S \cup v$ for all $v \in ext(S)$, as described by the loop in Line 5. Before the loop, we first apply the cover-vertex pruning of (P7) in Lines 2–4 to compute a cover set $C_S(u)$ so that those vertices in $C_S(u)$ can be skipped in Line 5. If we cannot find a cover vertex (see Line 3), then Line 5 iterates over all vertices of $ext(S)$.

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

For a vertex $v \in ext(S)$, the current iteration creates $S' = S \cup v$ for examination in Line 11. Before that, Lines 8–10 first checks if S extended with the entire current $ext(S)$ creates a valid quasi-clique; if so, this is a maximal one and is thus output in Line 9, and further examination can be skipped (Line 10). This pruning is called the look-ahead technique in Quick [26]. Note that $G(S \cup ext(S))$ must satisfy the size threshold requirement since Line 6 is passed.

The look-ahead technique is important since if $G(S \cup ext(S))$ is dense, its subgraphs tend to be dense and node $S \cup ext(S)$ can generate a large set-enumeration subtree if not pruned. If the look-ahead technique does not prune the search, then Line 11 creates $S' = S \cup v$ and excludes v from $ext(S)$. The latter also has a side effect of excluding v from $ext(S)$ of all subsequent iterations, which matches exactly how the set-enumeration tree (c.f. Fig. 2) avoids generating redundant nodes.

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

If $ext(S') \neq \emptyset$, Line 18 then calls Algorithm 1 to apply the pruning rules. Recall that the function either returns $\mathcal{T}_{pruned} = false$, indicating that we need to further extend S' using its shrunk $ext(S')$; or it returns $\mathcal{T}_{pruned} = true$ to indicate that the extensions of S' should be pruned, which will also output $G(S')$ if it is a valid quasi-clique (see Lines 16–19 in Algorithm 1).

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

Finally, as in Quick, Quick+ also requires a post-processing step to remove non-maximal quasi-cliques from the results of Algorithm 2. Also, we only run Quick+ after the input graph is shrunk by the k-core pruning of (P2). Additionally, we find that the vertex order in $ext(S)$ matters (Algorithm 2 Line 5) and can significantly impact the running time. To maximize the success probability of the look-ahead technique in Lines 8–10 of Algorithm 2 that effectively prunes the entire T_S, we propose to sort the vertices in $ext(S)$ in ascending order of $d_S(v)$ (tie broken by $d_{ext(S)}(v)$) following [4] so that high-degree vertices tend to appear in $ext(S)$ of more set-enumeration tree nodes.

7 Parallel Quick+ implementation in T-thinker

We next parallelize Algorithm 2 on T-thinker, where a long-running task is divided into smaller subtasks for concurrent processing. Recall from Sect. 5.1 that users write a T-thinker program by implementing 3 UDFs: (1) *load_data(file)* for loading data (i.e., vertices of a graph) into *data_array*; (2) *task_spawn*(v) for spawning a task from each vertex v (i.e., a data item in *data_array*); (3) *compute*$(t.context)$ for computing a task t. We next describe how we specify them for Quick+.

UDF load_data(file). When a T-thinker program begins, the main thread will call *load_data(file)* to load vertices into *data_array*, where each data item corresponds to a vertex $v \in V$. To provide an intuition using Fig. 2, UDF *load_data(file)* basically loads vertices a, b, c, and d at Level 1 of the set-enumeration tree into *data_array*, so that they can spawn initial tasks t_S for $S = \{a\}, \{b\}, \{c\}, \{d\}$.

In our implementation of *load_data(file)*, after loading the graph G from an input file, we additionally compute the k-core of G where $k = \lceil \gamma \cdot (\tau_{size} - 1) \rceil$ according to (P2), to prune those vertices that cannot be in any valid quasi-clique of size at least τ_{size}. We only add the remaining vertices into *data_array* so that less tasks will be spawned, and $ext(S)$ of a task will also not include those vertices with degree less than k. We also sort vertices in *data_array* in ascending order of degree to allow effective look-ahead pruning. Finally, we also conduct the initial cover-vertex-based pruning as introduced in (P7) to mask out those vertices $v \in N(u)$ where u is the vertex with the maximum degree in G, so that they will not call *task_spawn*(v) to spawn tasks.

UDF task_spawn (v). Let us denote each initial task as t_v that mines subtree $T_{\{v\}}$. We would like to construct a much smaller graph into $t_v.context$ to be memory-efficient since *task_queue* will contain a pool of tasks. Recall that according to (P1), t_v only needs to consider vertices within 2 hops from v, i.e., $\mathbb{B}(v)$. Additionally, among those vertices in $\mathbb{B}(v)$, t_v does not need to consider any vertex whose position is before v in *data_array*, since according to Fig. 2, $T_{\{v\}}$ does not contain any vertex before v (e.g., the subtree $T_{\{c\}}$ does not contain a and b). Here, we are using the position of vertices in *data_array* to determine their total order, and to facilitate such vertex pruning (from $\mathbb{B}(v)$), *load_data(file)* actually also scans *data_array* to build a reverse map from their vertex IDs to their positions in *data_array*, so that for any vertex $u \in \mathbb{B}(v)$, we can obtain the positions of u and v in $O(1)$ time to check if $u > v$. Only those vertices in $\mathbb{B}_>(v)$ are copied from G into $t_v.context$ for task computation.

In our implementation, if v is not masked by the initial cover-vertex-based pruning, we then create a task t_v with its context containing $S = \{v\}$ and graph $G(\mathbb{B}_>(v))$, and call *add_task*(t_v) to add the task to *task_queue*. Recall that *task_spawn*(v) is called in two places, one place is by worker to generate and fill an initial batch of C_{reg} tasks into *task_queue* (big tasks are added to Q_{big}, while other tasks are added to Q_{reg}), and the other place is by compers during their refill of *task_queue*.

Additionally, if *task_spawn*(v) finds that all vertices in *data_array* have finished calling *task_spawn*(.), then the input graph G is no longer needed and thus, *task_spawn*(v) frees G from memory to save memory space.

Timeout Mechanism to Avoid Stragglers To avoid long-running tasks that we demonstrated in Sect. 4, a timeout strategy is used to limit the time that any task can perform its computation without being decomposed into smaller tasks. Specifically, when a task t_S begins its mining, we record its starting time t_0. During the depth-first recursive mining of set-enumeration subtree T_S, if timeout happens, i.e., $t_{cur} - t_0 > \tau_{time}$ where t_{cur} is the current time and τ_{time} is the timeout duration threshold, we create a new task $t_{S'}$ for each vertex set S' to be checked, rather than continue to mine it recursively as in Algorithm 2 Line 21.

Figure 6 illustrates how the timeout strategy works for task t_a. The algorithm recursively expands the set-enumeration tree in depth-first order, processing 2 tasks until entering $\{a, b, c, d\}$ for which the entry time t_3 times out; we then wrap $\{a, b, c, d\}$ as a subtask and add it to *task_queue* 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. Also note that this strategy guarantees that each task spends at least a duration of τ_{time} on the actual mining by backtracking (which does not materialize subgraphs) before dividing the remaining workloads into subtasks (which needs to materialize their subgraphs), and we will show that the time spent by a task on subgraph materialization for its subtasks is very small compared with the time spent on the actual mining, so only a small overhead is incurred to improve load balancing and prevent straggler tasks.

UDF compute$(t.context)$. Algorithm 3 details our UDF *compute*$(t.context)$, which calls a recursive function *time_delayed*$(S, ext(S), t_0)$ by giving input $\langle t.S, t.ext(S), t_{cur} \rangle$.

Here, *time_delayed*$(S, ext(S), t_0)$ mines T_S recursively as in Algorithm 2, but decomposes the task if timeout occurs. Specifically, *time_delayed*(.) in Algorithm 3 now considers 2 cases. (1) Lines 18–24: if timeout happens, $\langle S', ext(S') \rangle$ is wrapped into a new task t' to add to *task_queue*; (2) Lines 25–30: we perform backtracking as in Algorithm 2, where we recursively call *time_delayed*(.) to process $\langle S', ext(S') \rangle$ in Line 26.

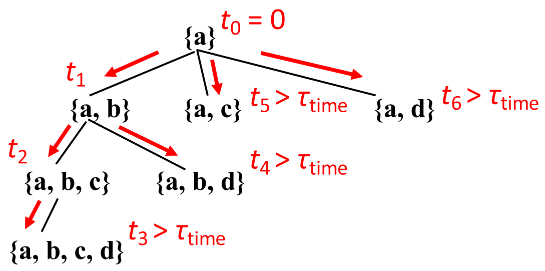


Fig. 6 Timeout-based divide and conquer

Algorithm 3 UDF $compute(t.context)$

```
UDF: $compute(t.context)$
1: $time\_delayed(t.S, t.ext(S), current\_time)$
   $\{t.t_0 = current\_time\}$
Function: $time\_delayed(S, ext(S), initial\_time)$
1: $\mathcal{T}_{Q\_found} \leftarrow false$
2: Find cover vertex $u \in ext(S)$ with the largest $C_S(u)$
3: {If not found, $C_S(u) \leftarrow \emptyset$}
4: Move vertices of $C_S(u)$ to the tail of the vertex list of $ext(S)$
5: for each vertex $v$ in the sub-list $(ext(S) - C_S(u))$ do
6:   if $|S| + |ext(S)| < \tau_{size}$ then:   return $false$
7:   if $G(S \cup ext(S))$ is a $\gamma$-quasi-clique then
8:     Append $S \cup ext(S)$ to the result file;   return $false$
9:   $S' \leftarrow S \cup v$, $ext(S) \leftarrow ext(S) - v$
10:  $ext(S') \leftarrow ext(S) \cap \mathbb{B}(v)$
11:  if $ext(S') = \emptyset$ then
12:    if $|S'| \geq \tau_{size}$ and $G(S')$ is a $\gamma$-quasi-clique then
13:      $\mathcal{T}_{Q\_found} \leftarrow true$
14:      Append $S'$ to the result file
15:  else
16:    $\mathcal{T}_{pruned} \leftarrow iterative\_bounding(S', ext(S'), \gamma, \tau_{size})$
17:    {here, $ext(S')$ is Type-I-pruned and $ext(S') \neq \emptyset$}
18:    if $current\_time - initial\_time > \tau_{time}$ then
19:      if $\mathcal{T}_{pruned} = false$ and $|S'| + |ext(S')| \geq \tau_{size}$ then
20:        Create a task $t'$
21:        $t'.S \leftarrow S'$;  $t'.ext(S) \leftarrow ext(S')$
22:        $add\_task(t')$
23:      if $|t'.S| \geq \tau_{size}$ and $G(t'.S)$ is a $\gamma$-quasi-clique then
24:        Append $t'.S$ to the result file
25:    else if $\mathcal{T}_{pruned} = false$ and $|S'| + |ext(S')| \geq \tau_{size}$ then
26:      $\mathcal{T}_{found} \leftarrow time\_delayed(S', ext(S'), initial\_time)$
27:      $\mathcal{T}_{Q\_found} \leftarrow \mathcal{T}_{Q\_found}$ or $\mathcal{T}_{found}$
28:      if $\mathcal{T}_{found} = false$ and $|S'| \geq \tau_{size}$ and $G(S')$ is a
         $\gamma$-quasi-clique then
29:        $\mathcal{T}_{Q\_found} \leftarrow true$
30:        Append $S'$ to the result file
31: return $\mathcal{T}_{Q\_found}$
```

Recall that Algorithm 2 is recursive where Line 21 extends S with another vertex $v \in ext(S)$ for recursive processing. Now in *time_delayed*(.) when timeout happens, we will instead create a new task t' with $t'.S = t.S \cup v$ (Lines 20–21), which is essentially $t_{S'}$ and add it to *task_queue* (Line 22). However, we still want to apply all our pruning rules to see if $T_{S'}$ can be pruned to avoid creating t' at the first place (Lines 6–17); if not, we will add t' to T-thinker in Line 22 so that when a comper becomes available, t' is scheduled for processing by calling $time\_delayed(S', ext(S'), t_0)$. Here, we shrink t''s subgraph to be induced by $S' \cup ext(S')$ so that the subtask is on a smaller graph, and since $t'.ext(S)$ shrinks (due to pruning) as the node level becomes deeper and $t'.g$ also shrinks, the computation cost becomes smaller.

Another difference is with Line 23 of Algorithm 2, where we only check if $G(S')$ is a valid quasi-clique when $\mathcal{T}_{found} = false$, i.e., the recursive call in Line 21 verifies that S' fails to be extended to produce a valid quasi-clique. Here in *time_delayed*(.), however, the recursive call now becomes an independent task t' (Lines 20–22), and the current task t has no clue of the result from t'. Therefore, 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. A subtask may later find a larger quasi-clique containing $t'.S = S'$, making $G(S')$ non-maximal, but post-processing will safely remove it.

Memory Optimization Recall that in UDF *task_spawn*(v), we create a task t_v with its context containing $S = \{v\}$ and graph $G(\mathbb{B}_>(v))$ and then, calls *add_task*(t_v) to add the task to *task_queue*. However, such an implementation has a problem: the worker initially generates and fills an initial batch of $C_{reg} = 512$ tasks into *task_queue*, and each task t_v is associated with a big initial subgraph $G(\mathbb{B}_>(v))$, consuming a lot of memory.

To avoid this issue, we delay the subgraph creation for the initial vertex-spawned tasks to UDF $compute(t.context)$ rather than in UDF *task_spawn*(v). That is, if a task t finds in $compute(t.context)$ that $|t.S| = 1$ ($t.S$ is obtained from $t.context$), then t is an initial tasks (rather than a decomposed one) and we will (1) first create its subgraph $G(\mathbb{B}_>(v))$ where v is the only vertex in $t.S$, (2) then increment the atomic counter for tracking how many initial tasks have created their subgraphs (for proper deletion of G), and (3) proceed to the original task computation shown in Algorithm 3.

Assume that we have 32 compers, then in the worst case, every comper is computing an initial vertex-spawned task with the task subgraph created in memory, leading to at most 32 large initial subgraphs rather than $C_{reg} = 512$ ones, hence much more memory-efficient.

8 Parallel kernel expansion

Recall from Sect. 2 that [36] proposed a method to find k large maximal γ-quasi-cliques by kernel expansion. Specifically, the method first **(1)** mines γ'-quasi-cliques ($\gamma' > \gamma$) using Quick, and then **(2)** selects the k' largest γ'-quasi-cliques as “kernels.” For each kernel S, it then **(3)** expands S into γ-quasi-cliques that are maximal locally in $G(S \cup ext(S))$, i.e., it mines maximal γ-quasi-cliques in the set-enumeration subtree T_S. **(4)** For all the γ-quasi-cliques found by expanding the k' kernels, the top-k largest results are then returned. In a nutshell, a kernel-expansion job can be represented by a parameter quadruple (γ', k', γ, k) besides the minimum size threshold τ_{size}, and it finds k large γ-quasi-cliques with at least τ_{size} vertices (if available).

This kernel expansion method can be generalized to other dense subgraph mining problems, where we first mine subgraphs denser than are actually required, so that only nodes in the top levels of a set-enumeration search tree are examined leading to faster mining; then, mining of the targeted dense subgraphs can begin only from the top-k' largest nodes representing the k' denser kernel subgraphs, skipping most other tree branches that tend to generate smaller dense subgraphs.

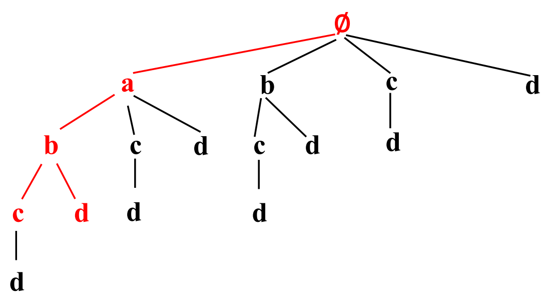


Fig. 7 The trie that incorporates all sets in Fig. 2

Referring to Fig. 2 again, we can first mine $k' = 2$ kernels like $S_1 = \{a, c\}$ and $S_2 = \{b, c\}$, and then continue to mine γ-quasi-cliques in T_{S_1} and T_{S_2}, which leads to result subgraphs such as $\{a, c, d\}$ and $\{b, c, d\}$.

However, expansions from different kernels may search the same set-enumeration tree nodes repeatedly, causing the total nodes searched to be even more than the entire set-enumeration search tree shown in Fig. 2 without redundancy. As we observed in [20], this kernel expansion solution can be even less efficient than mining all maximal γ-quasi-cliques directly using Quick+.

To illustrate why the problem arises, let us consider the following two scenarios. **Scenario 1:** there is only one kernel $S_1 = \{a, c\}$ found. In this case, if we use the node ordering $a < b < c < d$ as in Fig. 2, we will only find quasi-clique $\{a, c, d\}$ but will miss $\{a, b, c\}$ even if the latter is a large valid result. To find $\{a, b, c\}$ from S_1, we need to assume that a and c are before all other nodes in ordering, so $c < b$ and b should be included in $ext(S_1)$ unlike in Fig. 2 where we assume $b < c$. While this assumption avoids missing results, it can lead to redundant search as we explain next. **Scenario 2:** there are two kernels $S_1 = \{a, c\}$ and $S_2 = \{b, c\}$. According to our discussion above, when expanding S_1 we are assuming $a < b$ (i.e., $a, c \in S_1$ and $b \in ext(S_1)$) so S_1 can be expanded with b, but when expanding S_2 we are assuming $b < a$ (i.e., $b, c \in S_2$ and $a \in ext(S_2)$) so S_2 can be expanded with a. As a result, the expansions from both S_1 and S_2 will reach node $\{a, b, c\}$, meaning that both of them search for the entire set-enumeration subtree $T_{\{a,b,c\}}$! And such redundancy is pretty common among other vertex pairs beyond (a, b)!

Our proposed solution is to maintain a concurrent data structure $\mathcal{T}$ to keep all those nodes that have been explored by compers, so that, for example, if the expansion from S_1 has visited node $\{a, b, c\}$, it will be detected by the expansion from S_2, so this expansion will skip node $\{a, b, c\}$ and hence, the entire $T_{\{a,b,c\}}$, which is already being explored by the expansion from S_1. Note that since we are only expanding from k' kernels, we expect the number of nodes being searched to be acceptable so that $\mathcal{T}$ can keep all of them in memory.

A trivial solution is to keep $\mathcal{T}$ as a mutex-protected set, but this is inefficient since elements of $\mathcal{T}$ are themselves vertex sets such as $\{a, b, c\}$ and $\{a, b, d\}$, so here a and b are kept twice. Trie (i.e., prefix tree) is a more compact data structure choice that avoids such redundancy. For example, Fig. 7 shows a trie that encodes all the sets in Fig. 2, and we can see that fewer elements are saved compared with in Fig. 2. In fact, the red parts in Fig. 7 are those resulted after inserting both $\{a, b, c\}$ and $\{a, b, d\}$ into trie $\mathcal{T}$, and we can see that a and b are kept only once for these 2 sets. Note that each trie node x is also associated with a flag $\sigma(x)$ indicating if $S \in \mathcal{T}$, where S is a set represented by the path from the root to x. For example, if we consider the red subtree in Fig. 7 resulted from inserting $\{a, b, c\}$ and $\{a, b, d\}$ into an empty $\mathcal{T}$, then we have $\sigma(c) = \sigma(d) = true$, but $\sigma(a) = \sigma(b) = false$ since $\{a\}, \{a, b\} \notin \mathcal{T}$. If later $\{a, b\}$ is inserted into $\mathcal{T}$, $\sigma(b)$ will be set to $true$.

We remark that a trie basically executes a fixed vertex ordering. For example, in Fig. 7 we assume $a < b < c < d$. While different kernel expansions may use different vertex orders, when they check a node S against $\mathcal{T}$ for redundancy avoidance, S is basically reordered using the vertex ordering of trie $\mathcal{T}$.

However, since a T-thinker job searches the set-enumeration trees from kernel expansions in parallel, we need a thread-safe trie structure that supports high concurrency, but we are not aware of any such off-the-shelf concurrent trie library. In the sequel, we first present our T-thinker algorithm for the kernel expansion method to find large maximal γ-quasi-cliques, assuming that $\mathcal{T}$ is a thread-safe trie that supports high concurrency; we will then introduce how we implement such a concurrent trie data structure.

8.1 Kernel expansion algorithm in T-thinker

The kernel expansion algorithm can be implemented with two T-thinker jobs. **(1)** The first job runs exactly our parallel Quick+ program that was described in Sect. 7, to output maximal γ'-quasi-cliques from which we select the top-k' largest ones (if available) as the kernels. These kernels are saved in a kernel file. **(2)** The second job then reads the k' kernels and use them to create initial tasks for further expansion to find maximal γ-quasi-cliques, from which the k largest ones are the final results. We next describe the second job.

One way to implement the second job is to put the k' kernels in *data_array* to spawn initial kernel tasks for further expansion. The benefit is that when k' is large, T-thinker will spawn and refill *task_queue* on demand to keep memory consumption bounded. However, it is usually not necessary since k' is expected to be small (since the main goal is to reduce the number of nodes searched by the second job). So, we choose to directly create all the k' kernel tasks into *task_queue* in UDF *load_data*(.) by reading the kernel file, and since *data_array* is empty, UDF *task_spawn*(o) will never be called and can be left empty. As for UDF *compute*$(t.context)$, the logic is almost the same as in Algorithm 3, except that after Line 9, we need to add a new line "**if** $S' \in \mathcal{T}$ **then continue**" to skip S' if another kernel expansion has visited it.

Note that UDF *load_data*(.) still needs to load the input graph G initially and use it to create a task t_S for each kernel S, and then call *add_task*(t_S). The creation of different kernel tasks can be parallelized using OpenMP's parallel for-loop. Finally, *load_data*(.) frees G from memory.

For each kernel S, *load_data*(.) generates task $t_S = \langle S, ext(S)\rangle$ as follows. We construct graph $t_S.G = G(\bigcap_{v \in S} \mathbb{B}(v))$ to be added to $t_S.context$ based on (P1) diameter-based pruning. Then, $t_S.G$ is updated into its k-core where $k = \lceil \gamma \cdot (\tau_{size} - 1)\rceil$, and if some vertex in S is pruned, we directly delete t_S rather than add it to *task_queue*. Otherwise, we add those remaining vertices that are not in S into $ext(S)$, sort vertices in $ext(S)$ in ascending order of degree to allow effective look-ahead pruning. Other pruning rules mentioned in Sect. 6.3 can then be applied over $\langle S, ext(S)\rangle$ for further pruning.

8.2 Concurrent trie implementation

We now introduce how our concurrent trie $\mathcal{T}$ is implemented. Figure 8a shows the data structure of a trie node, and Fig. 8b shows a trie when four vertex sets S_1, S_2, S_3 and S_4 have been inserted, assuming that we define an order $a < b < c < d$. In the actual implementation, we directly order vertices by their IDs when inserting the vertex sets of a node S into $\mathcal{T}$.

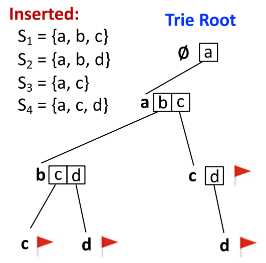
Note that to insert a set S into $\mathcal{T}$, we actually treat S as a sequence with its elements ordered and scan the elements one at a time and match each element to a trie node starting from the root node of the trie $\mathcal{T}$.

As shown in Fig. 8a, each trie node η maintains two fields. (1) The field $\eta.flag$ indicating whether $S \in \mathcal{T}$ where S is a vertex set that consists of the elements on the path from root to η. For example, in Fig. 8b, the flag is not set for the trie node b below a since $\{a, b\} \notin \mathcal{T}$, while the flag is set for the node c below a since $\{a, c\} = S_3 \in \mathcal{T}$. (2) The field $\eta.children$ is a table where entry $\eta.children[\eta_c]$ keeps a pointer to a child trie node of η (if it exists). For example, in Fig. 8b, the trie node b below a has a table *children* containing two child nodes (actually pointers to them) which are created when inserting S_1 and S_2, while the node c below a has a table *children* containing only one child node d which is created when inserting S_4.

Note from Fig. 8a that $\eta.flag$ is protected by a read-write lock $\eta.flag\_lock$, while $\eta.children$ is protected by another read-write lock $\eta.lock$. We use read-write lock rather than mutex for high concurrency. For example, consider the trie node b below a in Fig. 8b, it will be frequently visited as a

```
TrieNode <T>
Fields:
read-write lock lock
map<T, TrieNode <T>*> children
read-write lock flag_lock
bool flag
```

(a) Trie Node Data Structure



(b) Trie Structure

```
bool insert(const vector<T> & seq)
1  curr ⟵ trie root
2  for each e in seq
3    curr -> lock.read_lock()
4    parent ⟵ curr
5    if e is not in curr -> children
6      curr -> lock.unlock()
7      curr -> lock.write_lock()
8      if e is not in curr -> children
9        curr -> children[e] ⟵ new trie node
10   curr ⟵ curr -> children[e]
11   parent -> lock.unlock()
12   if e is the last element in seq
13     curr -> flag_lock.read_lock()
14     if curr -> flag = true
15       curr -> flag_lock.unlock()
16       return false
17     curr -> flag_lock.unlock()
18     curr -> flag_lock.write_lock()
19     if curr -> flag = false
20       curr -> flag ⟵ true
21       curr -> flag_lock.unlock()
22       return true
23     curr -> flag_lock.unlock()
24     return false
```

(c) Algorithm of Trie Insertion

Fig. 8 The concurrent trie data structure

bridging node by different compers if they insert (or search for the existence of) sets that contain elements a and b, and we would like these visits to happen in parallel without blocking each other, which is possible if a node η protects its fields (i.e., for thread safety) using read-write locks.

As Fig. 8c shows, there is only one function *insert*(S) for compers to call: if $S \notin \mathcal{T}$, S will be inserted and the function returns *true* to signal insertion success; otherwise, the function returns *false* to indicate that S is already in $\mathcal{T}$, so that a comper will prune the set-enumeration subtree T_S.

We next explain the insertion algorithm of Fig. 8c, and note that S has been encoded into a sequence *seq* for path traversal in $\mathcal{T}$. Starting from the root node (Line 1), we match the next element e in *seq* to the trie nodes one by one as specified by the for-loop in Line 2. Specifically, we want to see if a child trie node e has already been created due to a previous insertion (Line 5); here, to safely visit the *children* table of the current node, we add read-lock for *children* first in Line 3, and if e is in *children*, we directly move to the child node (Line 10), and unlock the previous read-lock (Lines 4 and 11).

While if node e has never been created before (Line 5), we upgrade the lock on *children* to a write-lock in Lines 6–7 to exclusively hold *children*, so that a new trie node is created for e and its pointer is added to *children* (Line 9), after which we move to this new node (Line 10) and unlock *children* for other compers to visit (Line 11). Note that we need to recheck if e has already been created again in Line 8 right after gaining the write-lock in Line 7, since it is possible that another comper θ first gained the write-lock on *children* in Line 7 and already updated *children* as in Line 9, and the current comper was waiting on Line 7 till θ releases its write-lock in Line 11. Without this recheck, a new trie node copy of e will overwrite the one created by θ, losing everything under that old trie node.

Once we reach the end of seq (Line 12), we need to check if the current node is already flagged which requires a read-lock on the flag to allow concurrent checks (Lines 13–16). If not, we need to run Lines 17–22 to upgrade the flag-lock to a write-lock and then set the flag to indicate that $S \in \mathcal{T}$. Similar to locking $children$, here we also need to recheck the flag after gaining the write-lock (Lines 18–19) since the write-lock could have been gained by another comper θ for setting S's flag (in which case T_S will be mined by θ rather than the current thread), and if this is the case, Lines 23–24 are executed instead to skip the flag setting and return *false* indicating that T_S can be skipped by the current task.

9 Experiments

This section reports our experiments. We have released the code of T-thinker and quasi-clique mining on GitHub at: https://github.com/yanlab19870714/Tthinker.

Datasets. We used 12 real graph datasets as shown in Table 4: biological networks *CX_GSE1730* and *CX_GSE10158*, arXiv collaboration network *Ca-GrQc*, email communication network *Enron*, product co-purchasing network *Amazon*, social networks *Hyves* and *YouTube*, patent citation network *Patent*, protein k-mer graph *kmer*, USA road network *USA Road*, a temporal Q&A interaction network *StackOverflow*, and friendship network *Pokec*. These graphs are selected to cover different graph type, size, and degree characteristics.

Notably, *StackOverflow* and *Pokec* are very dense with an average degree of 10.91 and 13.66, respectively. We have extensively tested various public graph datasets and find that such average degree values are the largest for datasets to be tractable on our server. We plan to mine denser and larger graphs using supercomputing as a future work. On the other hand, the absolute size of a graph is not a hurdle for processing as long as the memory space required to hold the graph does not exceed our server's RAM limit, as can be seen from *USA Road* and *kmer* which have over one order of magnitude more vertices but a low average degree $|E|/|V|$.

Algorithms & Parameters. By default, we use the T-thinker algorithm described in Algorithm 3 with the timeout mechanism that decomposes tasks running beyond a time period of τ_{time}, and we denote this algorithm by $\mathcal{A}_{time}$. We also consider another baseline without task decomposition (i.e., $\tau_{time} = \infty$) for comparison purpose, and we denote this variant by $\mathcal{A}_{base}$.

We use the tuned default system parameters for T-thinker as described in Sect. 5.3 in our experiments, which are summarized by Fig. 5. We also have a pair of algorithm parameters $(\tau_{big}, \tau_{time})$, where a task is considered a big one iff $|ext(S)| \geq \tau_{big}$, which is used by the *add_task*(t) function

Table 4 Graph datasets

Dataset	$\|V\|$	$\|E\|$	$\|E\|/\|V\|$	Max degree	τ_{size}	γ	#{Maximal}	URL
CX_ GSE1730	998	5098	5.11	197	30	0.85	79,356	https://www.ncbi.nlm.nih.gov/geo/query/acc.cgi?acc=GSE1730
CX_ GSE10158	1621	7079	4.37	110	29	0.77	17,245	https://www.ncbi.nlm.nih.gov/geo/query/acc.cgi?acc=GSE10158
Ca-GrQc	5,242	14,496	2.77	81	10	0.80	43,399	https://snap.stanford.edu/data/ca-GrQc.html
Enron	36,692	183,831	5.01	1383	23	0.84	50,676	https://snap.stanford.edu/data/email-Enron.html
Amazon	334,863	925,872	2.76	549	12	0.50	13	https://snap.stanford.edu/data/com-Amazon.html
Hyves	1,402,673	2,777,419	1.98	31,883	22	0.82	490,610	https://snap.stanford.edu/data/com-Youtube.html
YouTube	1,134,890	2,987,624	2.63	28,754	18	0.89	2,556	https://snap.stanford.edu/data/com-Youtube.html
Patent	3,774,768	16,518,947	4.38	793	20	0.90	256	https://snap.stanford.edu/data/cit-Patents.html
kmer	67,716,231	69,389,281	1.02	35	10	0.50	16	https://graphchallenge.s3.amazonaws.com/synthetic/gc6/U1a.tsv
USA Road	23,947,347	28,854,312	1.20	9	7	0.50	63	https://users.diag.uniroma1.it/challenge9/download.shtml
StackOverflow*	2,584,164	28,183,518	10.91	44,065	92	0.90	11	https://snap.stanford.edu/data/sx-stackoverflow.html
Pokec	1,632,803	22,301,964	13.66	14,854	29	0.95	6	http://konect.cc/networks/soc-pokec-relationships

of T-thinker. As we shall demonstrate later in this section, our tuned default algorithm parameters $(\tau_{big}, \tau_{time}) = (200, 1\text{ s})$ work consistently near-optimal and are thus adopted in this section unless otherwise stated.

While many graph-parallel systems have been proposed, none of them have ever implemented quasi-clique mining as an application on top, likely due to the complication of a quasi-clique mining algorithm such as Quick [26]. The latest work implementing Quick is published in IEEE BigData 2018 [36], so we name the implementation by *BigData'18*. Their implementation is released on GitHub[1], containing both an implementation of the basic Quick algorithm, as well as the kernel expansion technique to find large quasi-cliques. Since *BigData'18* is single-threaded and does not support parallel execution, we also compare our single-threaded Quick+ algorithm with *BigData'18* in this section.

Regarding the quasi-clique definition parameters (γ, τ_{size}) (recall Definition 3), we will study their effects next. In a nutshell, if we set their values too large, we cannot find any valid quasi-cliques, while if we set their values too small, there would be overwhelmingly many results for examining. We tested various value combinations of (γ, τ_{size}) on our datasets and obtained the default ones as summarized in Table 4, which lead to selective results for users to examine, and meanwhile, have sufficient mining workloads to run for a reasonable amount of time that is worth parallel computation.

Notably, we find that *StackOverflow* is too expensive to mine from scratch due to the high average degree of 10.91 and maximum degree of 44,065. Fortunately, mining its maximum clique (with 55 vertices) using our G-thinker program proposed in [50] is efficient with a running time of merely 48.88 s, so we only report experiments that expand large quasi-cliques from that maximum clique found by G-thinker as the only kernel. To emphasize this special setting, we append the name *StackOverflow* with a star in our tables, i.e., *StackOverflow**. All the other datasets are mined from scratch unless otherwise stated.

Experimental Setup. Our experiments were run on a server equipped with IBM POWER8 CPU (32 cores, 3491 MHz) and 1TB RAM. Unless otherwise stated, our T-thinker program was run with 32 compers. All reported results were averaged over 3 repeated runs. T-thinker requires only a very small amount of disk space to buffer tasks in all experiments thanks to our prioritizing disk-buffered tasks over spawning new tasks for task refill, so we omit disk consumption in most reported results.

We remark that while we used a powerful server with 1TB RAM, most of our experiments only need a small fraction of RAM that can easily fit in a typical modern server with 64 GB RAM or less. The only exception is *StackOverflow** which consumes most of our RAM, but as we have previously explained, this dataset is very challenging to mine quasi-cliques upon; for such datasets or even larger ones, we plan to explore more scalable solutions using supercomputing in the future.

Effect of Problem Parameters (γ, τ_{size}). We next demonstrate how the mining time of $\mathcal{A}_{time}$ varies with quasi-clique problem parameters γ and τ_{size}. Without loss of generality, we use three datasets *Patent*, *Hyves* and *Enron* for illustration. We fix one parameter and vary the other with four different values to show its effect.

Table 5 shows the number of results (denoted by column #{Results}) found by $\mathcal{A}_{time}$ and the number of maximal ones after post-processing (denoted by column #{Maximal}), along with the job running time and the peak memory usage, when we fix τ_{size} and change γ. We only tested 3 values for γ on *Patent* since the number of results already vary from 0 to over 44 million.

From Table 5, we obtain the following observations:

- A small change of γ can have a significant impact on the result number: for example, when changing (γ, τ_{size}) from (20, 0.9) to (20, 0.89) on *Patent*, the result number increases from 256 to over 44 million; and when changing (γ, τ_{size}) from (23, 0.88) to (23, 0.86) on *Enron*, the result number increases from 191 to over 100,000.
- As γ decreases, we see a stepped increase in the result number, where #{Maximal} remains in the same level for a few changes and then jumps to a next level with a number one order of magnitude larger (or more). This shows that real graphs often have different density levels similar to the 1-core, 2-core, 3-core, $\cdots$, in core decomposition, but not exactly the same as we explain next.
- In the same #{Maximal} level, the result number may even slightly drop as γ decreases. For example, as γ changes from 0.90 to 0.88 and then to 0.86 on *Enron*,

Table 5 Effect of γ

Dataset	τ_{size}	γ	Runtime (s)	#{Results}	#{Maximal}
Patent	20	0.91	28.537	0	0
		0.90	274.632	256	256
		0.89	943.855	44,083,823	44,080,758
Hyves	22	0.92	15.634	0	0
		0.88	18.608	2,352	1,433
		0.84	35.285	555,387	148,333
		0.80	548.472	51,570,125	6,343,942
Enron	23	0.92	7.754	0	0
		0.90	11.078	335	200
		0.88	33.792	351	191
		0.86	226.534	308,915	107,451

[1] https://github.com/beginner1010/topk-quasi-clique-enumeration

the result number first reduces from 200 to 191 and then increases to 107,451. This happens because two or more valid subgraphs may merge into a larger valid quasi-clique, while the reduction of γ is not yet enough to bring a lot of new results in the next density level.

We remark that the post-processing cost of removing non-maximal results is small 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 $\gamma = 0.9$ takes 0.002 s, while post-processing the over 44 million results when $\gamma = 0.89$ takes 282.38 s.

In reality, users are unlikely to check 44 million results, so we allow users to specify a threshold τ_{max} for the maximum number of results allowed. If our T-thinker program has found more than τ_{max} results by all compers collectively, the main thread will set the end flag to terminate the job so that users can adjust (γ, τ_{size}) with larger values to find denser (and fewer) structures that are more selective. When the result number is reasonable, the post-processing time is always negligible so we omit it in our report.

So far, we have discussed the effect of γ. The effect of τ_{size} is similar, and we show the results in Table 6. For example, when changing (γ, τ_{size}) from (24, 0.9) to (21, 0.9) on *Hyves*, the result number jumps from 6 to 11,087, which crosses multiple density levels.

Since the mining goal is to find a selective pool of largest valid subgraphs for prioritized examination, trials of different parameter values of (γ, τ_{size}) are necessary given that different graphs have different characteristics, and therefore, it is important that each trial should run efficiently like using our T-thinker program.

Default Problem and Algorithm Parameters. As users need trials to obtain a proper pair of problem parameters (γ, τ_{size}) that can find the most selective dense quasi-cliques for each dataset, we tune them to contain abundant but not too many results while ensuring that the job runs for a reasonable amount of time so that parallel computing helps. We

Table 6 Effect of τ_{size}

Dataset	τ_{size}	γ	Runtime (s)	#{Results}	#{Maximal}
Patent	22	0.9	73.847	0	0
	21		273.223	256	256
	19		282.965	256	256
	17		321.248	640	640
Hyves	24	0.9	15.323	6	6
	23		17.096	168	114
	22		17.539	2345	1,480
	21		19.198	20,431	11,087
Enron	24	0.9	10.962	15	15
	23		11.078	335	200
	22		14.811	4616	2,424
	21		19.233	47,031	20,742

Table 7 Effect of $(\tau_{big}, \tau_{time})$

τ_{time}	τ_{big}				
	1000	500	200	100	50
(a) Running time (second) on YouTube					
20	1249.35	1254.63	1255.42	1254.45	1248.32
10	1243.52	1252.54	1245.02	1244.91	1236.73
5	1234.78	1224.00	1239.66	1236.86	1241.58
1	888.77	1183.15	1176.19	891.54	901.23
0.1	885.96	902.69	899.87	901.81	909.91
0.01	909.03	910.03	906.31	912.90	915.84
(b) Peak Memory (MB) on YouTube					
20	58,546	57,937	56,946	57,571	56,266
10	58,473	58,464	56,300	56,240	57,114
5	57,751	58,444	56,383	57,953	58,210
1	58,479	58,147	57,226	57,402	58,268
0.1	59,099	59,608	57,625	56,879	56,148
0.01	59,095	59,871	57,926	55,641	56,593
(c) Running time (second) on enron					
20	1125.16	1144.77	1134.06	1146.88	1136.37
10	1129.27	1131.18	1137.26	1142.78	1143.27
5	1132.16	1125.65	1148.78	1138.57	1142.17
1	1161.39	1173.48	1199.71	1449.56	1327.47
0.1	2145.76	2029.60	2287.66	1988.69	2801.24
0.01	5676.18	5191.41	5990.86	6854.88	7424.53
(d) Peak Memory (MB) on Enron					
20	2235	1387	799	669	609
10	2529	1178	743	665	747
5	2232	1301	793	711	655
1	2041	1409	802	686	659
0.1	1766	1624	758	725	716
0.01	1259	1153	982	807	792
(e) Running time (second) on CX_GSE1730					
20	18.617	18.634	18.628	18.629	18.628
10	12.226	12.117	12.226	12.223	12.217
5	7.221	7.322	7.217	7.322	7.223
1	2.514	2.512	2.419	2.421	2.517
0.1	1.021	1.116	1.124	1.115	1.121
0.01	1.017	1.118	1.124	1.115	1.016
(f) Peak Memory (MB) on CX_GSE1730					
20	22	13	16	16	16
10	15	15	10	25	20
5	11	13	11	11	19
1	8	20	13	8	11
0.1	21	16	14	7	7
0.01	18	4	13	16	24

fix the tuned default problem parameters in the subsequent experiments, which are summarized in Table 4.

Regarding algorithm parameters, as we shall present next, we find that $(\tau_{big}, \tau_{time}) = (200, 1\text{ s})$ consistently delivers near-optimal performance across all our tested datasets and thus, are adopted by default, including experiments already reported in Tables 5 and 6.

Effect of Algorithm Parameters $(\tau_{big}, \tau_{time})$. We have tested the various pairs of values for $(\tau_{big}, \tau_{time})$ on all our datasets. Since we have 12 datasets in total, to save space, without loss of generality, we demonstrate the runtime and peak memory for *YouTube*, *Enron*, *CX_GSE1730* in Table 7.

From Tables 7(a), (c) and (e), we can see that using $(\tau_{big}, \tau_{time}) = (200, 1\text{ s})$ consistently delivers either the fastest performance or close to the fastest for $\mathcal{A}_{time}$. Other settings may slightly reduce the running time on some datasets, but can lead to a significant increase in time on others and are thus not stable. For example, Table 7(a) shows that smaller τ_{time} such as 0.1 s or 0.01 s delivers better performance on *YouTube*, but Table 7(c) shows that they can lead to many times slower performance on *Enron*.

Comparison of $\mathcal{A}_{base}$ and $\mathcal{A}_{time}$. To see how effective our timeout mechanism can improve load balancing and eliminate stragglers, we compare $\mathcal{A}_{time}$ with $\mathcal{A}_{base}$ by mining quasi-cliques using the default parameters shown in Table 4. Recall that $\tau_{time} = 1$ second in $\mathcal{A}_{time}$ while $\tau_{time} = \infty$ in $\mathcal{A}_{base}$.

Tables 8 and 9 show the results of $\mathcal{A}_{base}$ and $\mathcal{A}_{time}$, respectively, on all our datasets, where we report the job running time, the peak memory and disk usage. We can see that for graphs that are time-consuming to mine with $\mathcal{A}_{base}$, our $\mathcal{A}_{time}$ algorithm significantly speeds up the mining process.

For example, on *Enron*, $\mathcal{A}_{base}$ takes 17,797 s while $\mathcal{A}_{time}$ takes 1,172.80 s, a 15.2× speedup! As another example, on *StackOverflow**, $\mathcal{A}_{base}$ takes 11,035.98 s while $\mathcal{A}_{time}$ takes 1,202.85 s, a 9.2× speedup. This also holds similarly for *GX_GSE1730*, *YouTube*, *Patent* and *Hyves*, which shows the need of task decomposition to handle the straggler problem, and the advantage of our timeout strategy.

For those cases where there are no stragglers (e.g., on *kmer* and *USA Road*), $\mathcal{A}_{time}$ and $\mathcal{A}_{base}$ have a similar running time, showing that task creation caused by task decomposition does not add much overhead, thanks to our timeout mechanism that does not over-decompose a task beyond the necessary level (c.f., Fig. 6). We will show with additional experiments soon that the cost of task creation is small compared with the cost of task computation.

The RAM usage is also very scalable thanks to T-thinker's task spilling and refill design. For example, as Table 9 shows, $\mathcal{A}_{time}$ uses only 55.55 GB memory even when processing the big graph *YouTube*, and only 10.66 GB when processing the dense graph *Hyves*, and less than 10 GB when processing all the other graphs except for *StackOverflow**. In other words, except for *StackOverflow**, all other graphs can be easily processed with a modern server with 64 GB or less memory space.

As we have indicated earlier, *StackOverflow* is a very challenging dataset and mining it from scratch runs out of memory even with our server that has 1TB RAM space, because of its high average and maximum vertex degree. As Table 9 shows, even when mining *StackOverflow* by expanding from its maximum clique, $\mathcal{A}_{time}$ consumes 785.80 GB RAM and takes 1,202.85 s to find only 11 valid 0.9-quasi-cliques with size $\geq \tau_{size} = 92$. This shows the necessity of the kernel expansion method when processing graphs that are both dense and large.

Task spilling is mostly light. One exception is $\mathcal{A}_{time}$ on *Enron* where the peak disk space used by spilled tasks is given by 36.44 GB, which is because big tasks are kept being decomposed and spilled. However, this overhead is

Table 8 Performance of $\mathcal{A}_{base}$ with default (γ, τ_{size})

Dataset	τ_{big}	τ_{time}	Runtime (s)	#{Results}	#{Maximal}	Memory (MB)	Disk (MB)
CX_ GSE1730	200	∞	18.62	460,461	79,356	25	60
CX_ GSE10158	200	∞	1.01	29,580	17,245	12	2
Ca-GrQc	200	∞	0.62	115,627	43,399	13	8
Enron	200	∞	17,797.00	358,629	50,676	308	46
Amazon	200	∞	0.56	16	13	100	0
Hyves	200	∞	142.27	1,729,934	490,610	10,944	259
YouTube	200	∞	5464.80	4072	2556	68,698	8
Patent	200	∞	767.49	256	256	3071	2
kmer	200	∞	53.03	201	63	6742	0
USA Road	200	∞	20.62	16	16	2659	0
StackOverflow*	200	∞	11,035.98	11	11	720,341	4
Pokec	200	∞	2169.42	10	6	21,453	11

Table 9 Performance of $\mathcal{A}_{time}$ with default (γ, τ_{size})

Dataset	τ_{split}	τ_{time}	Runtime (s)	#{Results}	#{Maximal}	Memory (MB)	Disk (MB)
CX_ GSE1730	200	1	2.51	460,461	79,356	17	51
CX_ GSE10158	200	1	1.01	29,580	17,245	7	2
Ca-GrQc	200	1	0.62	115,627	43,399	4	11
Enron	200	1	1172.80	358,629	50,676	812	37,318
Amazon	200	1	0.56	13	13	42	0
Hyves	200	1	76.35	1,729,934	490,610	10,911	674
YouTube	200	1	1,191.21	4072	2556	56,884	1233
Patent	200	1	273.79	256	256	3075	2251
kmer	200	1	52.71	201	63	7031	0
USA Road	200	1	21.25	16	16	2,672	0
StackOverflow*	200	1	1202.85	11	11	806,446	23,551
Pokec	200	1	2070.15	10	6	21,205	11

Table 10 *BigData'18* versus Quick+ (single-threaded)

Dataset	BigData'18			Quick+ Single-Threaded		
	Runtime (s)	Memory (MB)	#{Maximal}	Runtime (s)	Memory (MB)	#{Maximal}
CX_ GSE1730	11,477.37	460	4475	17.6	14	79,356
CX_ GSE10158	633.00	417	9656	1.0	7	17,245
Ca-GrQc	>12h	N/A	N/A	1.1	4	43,399
Enron	>12h	N/A	N/A	30,654.3	370	50,676
Amazon	18.13	162	13	1.5	13	13
Hyves	20,309.94	1251	47,001	1923.8	11,093	490,610
YouTube	>12h	N/A	N/A	21,613.8	56,560	2556
Patent	>12h	N/A	N/A	6781.5	2403	256
kmer	1230.04	27,847	63	64.5	7290	63
USA Road	476.12	10,114	16	27.7	2860	16
StackOverflow*	>12h	N/A	N/A	15,150.6	777,282	11
Pokec	>12h	N/A	N/A	35,070.9	22,313	6

Table 11 Total versus subgraph materialization time on Patent

τ_{time}	Runtime (s)	Total l	Total subgraph materialization time	Total versus materialization time ratio
50	284.264	8195.601	20.228	405.160
20	288.414	8212.309	21.574	380.662
10	277.950	8,220.621	24.320	338.014
1	273.403	8,205.839	75.433	108.784
0.5	276.608	8,158.877	183.625	44.432
0.1	298.851	7,847.789	620.876	12.640
0.01	952.488	7,139.440	4,145.621	1.722

acceptable and the better load balancing resulted in a 15.17× speedup over $\mathcal{A}_{base}$ which only uses 46 MB disk space at its peak. Another exception is *StackOverflow** where $\mathcal{A}_{time}$ uses 20.71 GB disk space at its peak, but this is understandable given the expensive cost of mining the large and dense *StackOverflow* graph as we have previously explained. The disk space consumed by all the other experiments is negligible, which shows that our task refilling strategy that prioritizes spilled tasks for refill is effective in reducing the chance and data volume of task spilling.

Finally, consider the number of subgraphs outputted before post-processing (to remove non-maximal ones), as

Table 12 Total versus subgraph materialization time on *Hyves*

τ_{time}	Runtime (s)	Total time of all tasks	Total subgraph materialization time	Total versus materialization time ratio
50	91.267	1952.653	44.454	43.925
20	78.834	1985.455	46.899	42.334
10	77.604	1976.065	51.642	38.264
1	76.656	1931.759	109.286	17.676
0.5	77.244	1905.852	145.241	13.122
0.1	80.709	1915.606	237.618	8.062
0.01	89.116	1801.483	447.759	4.023

indicated by Column #{Results} in Tables 8 and 9. The subgraph numbers in Table 8 are almost the same as those in Table 9, but slightly higher in some cases, such as 16 versus 13 on *Amazon*. Recall that in Algorithm 3, the recursive call of *time_delayed*(.) now becomes an independent task t' (Lines 20–22), and the current task t has no clue of the result from t', i.e., whether t' will find a valid quasi-clique that extends S'. Therefore, 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, which could result in more number of outputted subgraphs than $\mathcal{A}_{base}$. Fortunately, in all our experiments, we find that the number of such additional non-maximal outputs are very limited and mostly 0, as Tables 8 and 9 have demonstrated.

Quick+ versus BigData'18 Since *BigData'18* [26] implements the serial Quick algorithm [36], we also compare our single-threaded Quick+ algorithm with *BigData'18*, by repeating the experiments in Tables 8 and 9. The results are shown in Table 10, where we can see that our Quick+ is much faster than *BigData'18*, often by one to two orders of magnitude. We believe that besides better implementation, the improved pruning rules of Quick+ also plays a part in achieving better performance. In fact, *BigData'18* cannot finish in 12 h on 6 out of the 12 datasets, and on the other 6 it misses results on 3 of them, possibly due to some bugs in their implementation. For example, Table 10 shows that on *Hyves*, *BigData'18* only finds 47,001 of the 490,610 results using 20,309.94 s, while Quick+ finds all results in 1923.8 s; moreover, Table 9 shows that using 32 compers, $\mathcal{A}_{time}$ is able to reduce the time to 76.35 s!

Cost of Task Decomposition Recall from Algorithm 3 that if a timeout happens, we need to generate subtasks with smaller overlapping subgraphs (see Lines 18-22), the subgraph materialization cost of which is not part of the original mining workloads. The smaller τ_{time} is, the more often task decomposition is triggered, and hence more subgraph materialization overheads are generated. Our tests show that the additional time spent on task materialization is not significant compared with the actual mining workloads, especially when we use the default timeout threshold $\tau_{time} = 1$ s.

For example, Table 11 shows the profiling results on *Patent*, including the job running time, the sum of task processing time spent by all tasks, the sum of subgraph materialization time spent by all tasks, and a ratio of the latter two. We can see that decreasing τ_{time} does increase the fraction of cumulative time spent on subgraph materialization due to more frequent occurrences of task decompositions, but this cost is very small compared with the total task processing time: for example, with the default setting of $\tau_{time} = 1$ s, the materialization overhead accounts for only 1/108.8 of the total task processing time, which can be reduced significantly if we increase τ_{time} further. So, only a small cost is paid for better load balancing, and recall from Tables 8 and 9 that such a timeout mechanism reduces the runtime from 142.27 s to only 76.35 s.

As another example using *Hyves* which is shown by Table 12, the default setting of $\tau_{time} = 1$ s gives a materialization overhead that accounts for only 1/17.7 of the total task processing time. So, only a small cost is paid for better load balancing, and recall from Tables 8 and 9 that such a timeout mechanism reduces the runtime from 132.37 s to only 70.24 s.

Vertical Scalability As an illustration of the scalability of our T-thinker program with the number of CPU cores used, Figs. 9 and 10 show the vertical scalability of $\mathcal{A}_{time}$ on *Enron* and *Patent*, respectively. We can see that the speedup ratio is close to ideal: when there are 32 compers, the speedup ratio is 26.06× and 24.81× on *Enron* and *Patent*, respectively, which is not far from 32. More importantly, when there are 16 compers, the speedup ratio is ideal (i.e., nearly 16×) in all our datasets, meaning that our T-thinker program is able to fully utilize all CPU cores in a 16-core machine.

Figures 9 and 10 also show that the memory cost slightly increases with the number of compers, since more compers means more concurrent tasks (with subgraphs maintained) being generated (e.g., by refill) and processed. Note that the RAM usage increases slowly while we double the number of compers, since the task queue capacity is fixed, and there is a base RAM cost required to hold the input graph.

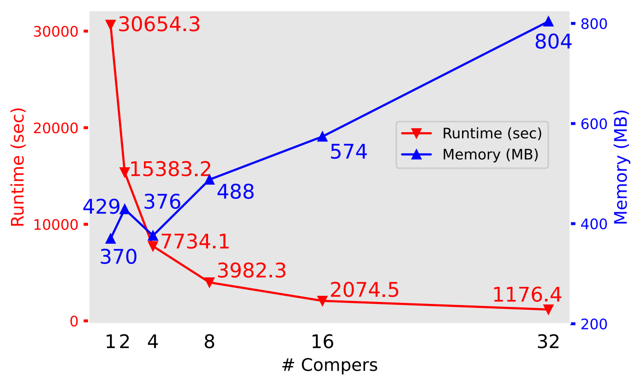


Fig. 9 Vertical scalability on *Enron*

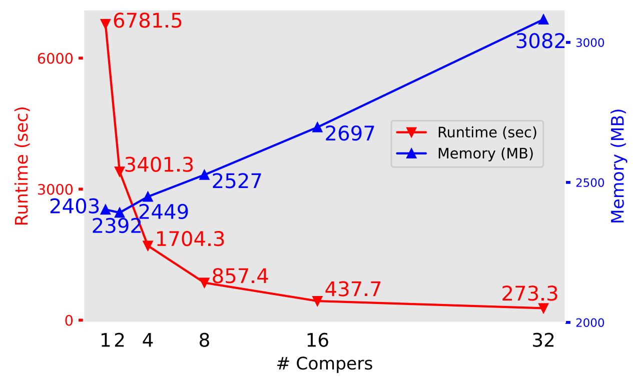


Fig. 10 Vertical scalability on *Patent*

Effectiveness of Kernel Expansion Recall from Sect. 2 that [36] first mines quasi-cliques with $\gamma' > \gamma$, then finds the top-k' largest result subgraphs as "kernels" which are then expanded to generate γ-quasi-cliques and return top-k maximal ones from the results. Thus, a job of [36] takes a parameter quadruple (γ', k', γ, k). We use their default setting $k' = 3k$ and $k = 100$ for experiments. We conducted some experiments where we choose values of τ_{size}, γ and γ' such that mining γ'-quasi-cliques is significantly faster than directly mining γ-quasi-cliques. Table 13 shows the performance of the two stages running with our T-thinker program, where we denote the time for γ'-quasi-clique kernel generation by t_1 and the time for kernel expansion by t_2. As a comparison, we also show t_1 and t_2 used by *BigData'18* [36], which are two or more orders of magnitude longer than the time by T-thinker. Notably, *BigData'18* cannot finish Stage 1 for kernel generation on *Patent* within 12 h, so Stage 2 cannot run and t_2 is thus N/A. Interestingly, while the kernel expansion approach of *BigData'18* could miss results as we explained in Sect. 8, this does not occur in our tested experiments: the top-k results found by our T-thinker program and *BigData'18* are all the same.

To evaluate the result quality of the kernel expansion technique, we define the concept of "top-k recall" as the fraction of the exact top-k largest γ-quasi-cliques that are within the

Table 13 Effectiveness of Kernel expansion

Dataset	τ_{size}	Stage 1: Kernel Generation						Stage 2: Kernel Expansion					
		k'	γ'	Runtime (s)	#{ Results}	# {Maximal}	BigData'18 Time	k	γ	Runtime (s)	# {Results}	#{Maximal}	BigData'18 Time
CX_GSE1730	30	300	0.90	0.407	3690	1602	608.48	100	0.80	0.117	16,194	2237	1.52
CX_GSE10158	29	300	0.82	0.108	2170	1405	32.58	100	0.74	0.110	7612	925	0.5
Hyves	22	300	0.90	18.679	2,345	1480	1828.06	100	0.80	10.970	66,573	15,116	14.37
Enron	23	300	0.90	12.690	335	200	984.95	100	0.84	0.695	1418	401	3.00
YouTube	18	300	0.93	720.817	662	473	28703.5	100	0.86	47.026	1245	314	63.01
Patent	17	300	0.90	323.438	640	640	>12h	100	0.89	13.767	398	298	N/A

Table 14 Effectiveness of Kernel expansion

Dataset	Kernel-Technique Runtime (s)	Top-50 Recall (%)	Top-100 Recall (%)	Top-200 Recall (%)	Without Kernel Runtime (s)	Speedup Ratio
CX_ GSE1730	0.524	100	97	98.5	31.23	59.60
CX_ GSE10158	0.218	100	100	100	1.81	8.29
Hyves	29.649	58	79	89.5	546.16	18.42
Enron	13.385	30	65	48.5	1197.41	89.46
YouTube	767.843	94	90	45	2062.01	2.69
Patent	337.205	100	100	100	1097.58	3.25

top-k largest quasi-cliques found by Stage 2. Intuitively, top-k recall measures how many large quasi-cliques are missed when using kernel expansion rather than exact mining. Note that the result precision is always 100% since Stage 2 always finds valid quasi-cliques that meet the (γ, τ_{size}) requirement.

Table 14 reports the top-k recall for $k = 50, 100, 150$. We can see that the recall is highly data-dependent, with some reaching 100% (on *CX_GSE1730* and *Patent*) while others reaching as low as 30% (on *Enron*). Note that the recall value is not monotonic to k, since as k changes, so is the set of top-k exact largest quasi-cliques that determine the numerator of the recall ratio computation (although the denominator increases with k). Table 14 also reports the total time of our kernel-based technique (i.e., $(t_1 + t_2)$) as well as the time t_3 to mine γ-quasi-cliques directly (Column "Without Kernel Runtime"). The speedup ratio is thus given by $t_3/(t_1 + t_2)$. From Table 14, we can see that kernel expansion can be quite effective on some datasets: for example, on *Enron* the time drops from $t_3 = 1,197.41$ s to only 13.39 s (89.46×). From a few times up to 59.60× speedup can be reached on the other datasets.

10 Conclusion

We proposed a task-based parallel framework called T-thinker that is able to fully utilize CPU cores, and implement maximal quasi-clique mining on top. We also proposed a novel trie-based solution for redundancy-free kernel expansion. Compared with our distributed quasi-clique mining solution in our prior PVLDB 2020 paper, the current program is more accessible to users as multi-core machines are readily available. Extensive experiments showed that our parallel solution achieves excellent speedups on various real graph datasets.

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

References

1. Abello, J., Resende, M.G.C., Sudarsky, S.: Massive quasi-clique detection. In: LATIN, volume 2286 of Lecture Notes in Computer Science, pp. 598–612. Springer (2002)
2. Bader, G.D., Hogue, C.W.: An automated method for finding molecular complexes in large protein interaction networks. BMC Bioinform. **4**(1), 2 (2003)
3. Batagelj, V., Zaversnik, M.: An o(m) algorithm for cores decomposition of networks. CoRR, cs.DS/0310049 (2003)
4. Bayardo Jr, R.J.: Efficiently mining long patterns from databases. In: SIGMOD Conference, pp. 85–93. ACM Press (1998)
5. Berlowitz, D., Cohen, S., Kimelfeld, B.: Efficient enumeration of maximal k-plexes. In: SIGMOD Conference, pp. 431–444. ACM (2015)
6. Bhattacharyya, M., Bandyopadhyay, S.: Mining the largest quasi-clique in human protein interactome. In: 2009 International Conference on Adaptive and Intelligent Systems, pp. 194–199. IEEE (2009)
7. Brunato, M., Hoos, H.H., Battiti, R.: On effectively finding maximal quasi-cliques in graphs. In: International Conference on Learning and Intelligent Optimization, pp. 41–55. Springer, Berlin (2007)
8. Bu, D., Zhao, Y., Cai, L., Xue, H., Zhu, X., Lu, H., Zhang, J., Sun, S., Ling, L., Zhang, N., et al.: Topological structure analysis of the protein-protein interaction network in budding yeast. Nucleic Acids Res. **31**(9), 2443–2450 (2003)
9. COST in the Land of Databases. https://github.com/frankmcsherry/blog/blob/master/posts/2017-09-23.md
10. Chang, L., Yu, J.X., Qin, L., Lin, X., Liu, C., Liang, W.: Efficiently computing k-edge connected components via graph decomposition. In: SIGMOD Conference, pp. 205–216. ACM (2013)
11. Chen, H., Liu, M., Zhao, Y., Yan, X., Yan, D., Cheng, J.: G-miner: an efficient task-oriented graph mining system. In: EuroSys, pp. 32:1–32:12. ACM (2018)
12. Chou, Y.H., Wang, E.T., Chen, A.L.P.: Finding maximal quasi-cliques containing a target vertex in a graph. In: DATA, pp. 5–15. SciTePress (2015)
13. Chu, S., Cheng, J.: Triangle listing in massive networks. TKDD **6**(4), 17:1–17:32 (2012)
14. Conde-Cespedes, P., Ngonmang, B., Viennet, E.: An efficient method for mining the maximal α-quasi-clique-community of a given node in complex networks. Soc. Netw. Anal. Min. **8**(1), 20 (2018)
15. Conte, A., Firmani, D., Mordente, C., Patrignani, M., Torlone, R.: Fast enumeration of large k-plexes. In: SIGKDD, pp. 115–124. ACM (2017)
16. Conte, A., Matteis, T.D., Sensi, D.D., Grossi, R., Marino, A., Versari, L.: D2K: scalable community detection in massive networks

via small-diameter k-plexes. In: SIGKDD, pp. 1272–1281. ACM (2018)
17. Cui, W., Xiao, Y., Wang, H., Lu, Y., Wang, W.: Online search of overlapping communities. In: SIGMOD Conference, pp. 277–288. ACM (2013)
18. Fan, W., Jin, R., Liu, M., Lu, P., Luo, X., Xu, R., Yin, Q., Yu, W., Zhou, J.: Application driven graph partitioning. In: SIGMOD Conference, pp. 1765–1779. ACM (2020)
19. Guo, G., Yan, D., Özsu, M.T., Jiang, Z., Khalil, J.: Scalable mining of maximal quasi-cliques: an algorithm-system codesign approach. Proc. VLDB Endow. **14**(4), 573–585 (2020)
20. Guo, G., Yan, D., T. Özsu, M., Jiang, Z., Khalil, J.: Scalable mining of maximal quasi-cliques: An algorithm-system codesign approach. CoRR, arXiv:2005.00081 (2020)
21. Hopcroft, J., Khan, O., Kulis, B., Selman, B.: Tracking evolving communities in large linked networks. Proc. Natl. Acad. Sci. **101**(suppl 1), 5249–5253 (2004)
22. Jiang, D., Pei, J.: Mining frequent cross-graph quasi-cliques. ACM Trans. Knowl. Discov. Data **2**(4), 16:1–16:42 (2009)
23. Joshi, A., Zhang, Y., Bogdanov, P., Hwang, J.: An efficient system for subgraph discovery. In: IEEE Big Data, pp. 703–712 (2018)
24. Lee, P., Lakshmanan, L.V.S.: Query-driven maximum quasi-clique search. In: SDM, pp. 522–530. SIAM (2016)
25. Li, J., Wang, X., Cui, Y.: Uncovering the overlapping community structure of complex networks by maximal cliques. Physica A Stat. Mech. Appl. **415**, 398–406 (2014)
26. Liu, G., Wong, L.: Effective pruning techniques for mining quasi-cliques. In: ECML/PKDD, volume 5212 of Lecture Notes in Computer Science, pp. 33–49. Springer, Berlin (2008)
27. Lu, C., Yu, J.X., Wei, H., Zhang, Y.: Finding the maximum clique in massive graphs. Proc. VLDB Endow. **10**(11), 1538–1549 (2017)
28. Lyu, B., Qin, L., Lin, X., Zhang, Y., Qian, Z., Zhou, J.: Maximum biclique search at billion scale. Proc. VLDB Endow. **13**(9), 1359–1372 (2020)
29. Malewicz, G., Austern, M.H., Bik, A.J.C., Dehnert, J.C., Horn, I., Leiser, N., Czajkowski, G.: Pregel: a system for large-scale graph processing. In: SIGMOD Conference, pp. 135–146 (2010)
30. Matsuda, H., Ishihara, T., Hashimoto, A.: Classifying molecular sequences using a linkage graph with their pairwise similarities. Theor. Comput. Sci. **210**(2), 305–325 (1999)
31. McSherry, F., Isard, M., Murray, D.G.: Scalability! but at what cost? In: HotOS (2015)
32. Pattillo, J., Veremyev, A., Butenko, S., Boginski, V.: On the maximum quasi-clique problem. Discrete Appl. Math. **161**(1–2), 244–257 (2013)
33. Pei, J., Jiang, D., Zhang, A.: On mining cross-graph quasi-cliques. In: SIGKDD, pp. 228–238. ACM (2005)
34. Qin, L., Yu, J.X., Chang, L., Cheng, H., Zhang, C., Lin, X.: Scalable big graph processing in mapreduce. In: SIGMOD Conference, pp. 827–838. ACM (2014)
35. Quamar, A., Deshpande, A., Lin, J.: Nscale: neighborhood-centric large-scale graph analytics in the cloud. VLDB J. 1–26 (2014)
36. Sanei-Mehri, S., Das, A., Tirthapura, S.:Enumerating top-k quasi-cliques. In: IEEE BigData, pp. 1107–1112. IEEE (2018)
37. Tanner, B.K., Warner, G., Stern, H., Olechowski, S.: Koobface: The evolution of the social botnet. In: eCrime, pp. 1–10. IEEE (2010)
38. Teixeira, C.H.C., Fonseca, A.J., Serafini, M., Siganos, G., Zaki, M.J., Aboulnaga, A.: Arabesque: a system for distributed graph mining. In: SOSP, pp. 425–440 (2015)
39. Wang, K., Zuo, Z., Thorpe, J., Nguyen, T.Q., Xu, G.H.: Rstream: Marrying relational algebra with streaming for efficient graph mining on A single machine. In: OSDI, pp. 763–782 (2018)
40. Weiss, D., Warner, G.: Tracking criminals on facebook: a case study from a digital forensics reu program. In: Proceedings of Annual ADFSL Conference on Digital Forensics, Security and Law (2015)
41. Yan, D., Bu, Y., Tian, Y., Deshpande, A.: Big graph analytics platforms. Found. Trends Databases **7**(1–2), 1–195 (2017)
42. Yan, D., Bu, Y., Tian, Y., Deshpande, A., Cheng, J.: Big graph analytics systems. In: SIGMOD Conference, pp. 2241–2243. ACM (2016)
43. Yan, D., Cheng, J., Chen, H., Long, C., Bangalore, P.: Lightweight fault tolerance in pregel-like systems. In: ICPP, pp. 69:1–69:10. ACM (2019)
44. Yan, D., Cheng, J., Lu, Y., Ng, W.: Blogel: a block-centric framework for distributed computation on real-world graphs. Proc. VLDB Endow. **7**(14), 1981–1992 (2014)
45. Yan, D., Cheng, J., Lu, Y., Ng, W.: Effective techniques for message reduction and load balancing in distributed graph computation. In: WWW, pp. 1307–1317 (2015)
46. Yan, D., Cheng, J., Özsu, M.T., Yang, F., Lu, Y., Lui, J.C.S., Zhang, Q., Ng, W.: A general-purpose query-centric framework for querying big graphs. Proc. VLDB Endow. **9**(7), 564–575 (2016)
47. Yan, D., Cheng, J., Xing, K., Lu, Y., Ng, W., Bu, Y.: Pregel algorithms for graph connectivity problems with performance guarantees. PVLDB **7**(14), 1821–1832 (2014)
48. Yan, D., Guo, G.: Systems and algorithms for massively parallel graph mining. In: BigData. IEEE (2020)
49. Yan, D., Guo, G., Chowdhury, M.M.R., Özsu, M.T., Ku, W., Lui, J.C.S.: G-thinker: a distributed framework for mining subgraphs in a big graph. In: ICDE, pp. 1369–1380. IEEE (2020)
50. Yan, D., Guo, G., Khalil, J. et al. G-thinker: a general distributed framework for finding qualified subgraphs in a big graph with load balancing. The VLDB Journal (2021). https://doi.org/10.1007/s00778-021-00688-z
51. Yan, D., Guo, G., Chowdhury, M.M.R., Özsu, M.T., Lui, J.C.S., Tan, W.: T-thinker: a task-centric distributed framework for compute-intensive divide-and-conquer algorithms. In: PPoPP, pp. 411–412. ACM (2019)
52. Yan, D., Huang, Y., Liu, M., Chen, H., Cheng, J., Wu, H., Zhang, C.: Graphd: Distributed vertex-centric graph processing beyond the memory limit. IEEE Trans. Parallel Distrib. Syst. **29**(1), 99–114 (2018)
53. Yan, D., Liu, H.: Parallel graph processing. In: Encyclopedia of Big Data Technologies. Springer (2019)
54. Yan, D., Qu, W., Guo, G., Wang, X.: Prefixfpm: A parallel framework for general-purpose frequent pattern mining. In: ICDE, pp. 1938–1941. IEEE (2020)
55. Yan, D., Qu, W., Guo, G. et al.: PrefixFPM: a parallel framework for general-purpose mining of frequent and closed patterns. The VLDB Journal (2021). https://doi.org/10.1007/s00778-021-00687-0
56. Yan, D., Tian, Y., Cheng, J.: Systems for Big Graph Analytics. Springer Briefs in Computer Science. Springer (2017)
57. Yang, Y., Yan, D., Wu, H., Cheng, J., Zhou, S., Lui, J.C.S.: Diversified temporal subgraph pattern mining. In: SIGKDD, pp. 1965–1974. ACM (2016)
58. Zeng, Z., Wang, J., Zhou, L., Karypis, G.: Coherent closed quasi-clique discovery from large dense graph databases. In: SIGKDD, pp. 797–802. ACM (2006)
59. Zhang, Q., Yan, D., Cheng, J.: Quegel: A general-purpose system for querying big graphs. In: SIGMOD Conference, pp. 2189–2192. ACM (2016)
60. Zhou, Y., Xu, J., Guo, Z., Xiao, M., Jin, Y.: Enumerating maximal k-plexes with worst-case time guarantee. In: AAAI, pp. 2442–2449. AAAI Press (2020)