Parallel Clique-like Subgraph Counting and Listing

Yi Yang[1], Da Yan[2], Shuigeng Zhou[1], and Guimu Guo[2]

[1] Shanghai Key Lab of Intelligent Information Processing, and
School of Computer Science, Fudan University, Shanghai 200433, China
{yyang1,sgzhou}@fudan.edu.cn
[2] Department of Computer Science, The University of Alabama at Birmingham
{yanda,guimuguo}@uab.edu

Abstract. Cliques and clique-like subgraphs (e.g., quasi-cliques) are important dense structures whose counting or listing are essential in applications like complex network analysis and community detection. These problems are usually solved by divide and conquer, where a task over a big graph can be recursively divided into subtasks over smaller subgraphs whose search spaces are disjoint. This divisible algorithmic paradigm brings enormous potential for parallelism, since different subtasks can run concurrently to drastically reduce the overall running time.
In this paper, we explore this potential by proposing a unified framework for counting and listing clique-like subgraphs. We study how to divide and distribute the counting and listing tasks, and meanwhile, to balance the assigned workloads of each thread dynamically. Four applications are studied under our parallel framework, i.e., triangle counting, clique counting, maximal clique listing and quasi-clique listing. Extensive experiments are conducted which demonstrate that our solution achieves an ideal speedup on various real graph datasets.

Keywords: Dense Subgraph Mining · Parallel Computation · Unified Framework.

1 Introduction

Dense subgraphs of a network often contain important information about the communities or modules in the network, and as a result, counting and listing dense subgraphs has received a lot of interest from the research community in the last decade [8, 9, 11, 13, 22, 25]. One example of a dense subgraph is a clique, where every pair of vertices are connected by an edge.

However, these problems have a high computational complexity [17], often NP-hard due to the reduction to the maximum clique problem [18]. Recent studies focus on speeding up the computation by parallel computing [10, 12, 23, 26]. Since the original problem has an exponential computational complexity, after dividing it into multiple subproblems, either the number of subproblems or the time cost of an individual subproblem must be exponential. Accordingly, existing works can be categorized into two aspects.

1. Polynomial Number of Subproblems. This kind of work often uses simple task prepartitioning (e.g., on top of MapReduce [10, 23, 24] or Pregel-like systems [15, 27]). Since the time cost of individual subproblems are exponential, they often suffer from imbalanced workload distribution (e.g., the last-reducer problem) [21] especially on power-law graphs with a heavily skewed degree distribution. However, since the number of subproblems is polynomial, it is not time costly to rearrange the subproblems in order (e.g., vertex ordering schemes are applied [11, 25, 26]). After rearrangements, the time costly subproblems will be handled in parallel firstly, and the less costly subproblems will be handled in parallel lastly. As a result, the workloads are almost balanced, with the differences between the running time of the last few subproblems.

2. Exponential Number of Subproblems. This kind of work often uses dynamic task partitioning [19] and dynamic load balancing [7, 20], or they will suffer from exponential memory consumption storing the fully partitioned subproblems [8]. The dynamics is usually ensured by recursive partitioning, i.e. a subproblem can be further divided into smaller subproblems. Since the number of final subproblems is exponential, the granularity of the partitioning should be carefully chosen (e.g. coarse-grained partitioning leads more load differences and fine-grained partitioning leads more communication cost). The optimal solution usually lays out between extremes, which requires a proper cost model for both communication and computation, and requires an optimization method to find out the optimal solution of the cost model [8].

Contributions. In this paper, we present a unified framework for counting and listing clique-like subgraph in parallel, which takes both of the advantages of the previous two aspects. More specifically, on one hand, we propose a new task partitioning method called *pivot path partitioning*, which gradually partition the original task into a polynomial number of subtasks, then three vertex ordering schemes are compared. On the other hand, the pivot path partitioning method automatically partitions the original task into a proper granularity, without dealing with a cost model and its optimization. More over, the pivot path partitioning method enables general clique counting, which is little explored previously. We focus on the setting of multi-threaded computation within a single machine (i.e., a shared-memory environment). Our contribution can be summarized as follows.

- A unified parallel framework is proposed.
- The pivot path partitioning method is developed.
- First attempt on general clique counting.

2 Preliminaries

This section first defines our graph notations and terminology, and then introduces the clique-like problems that we solve on top of our parallel framework.

2.1 Notations and Terminology

Graphs. Let $G = (V, E)$ be a simple undirected graph with a vertex set V and an edge set E, and let $n = |V|$ and $m = |E|$ be the number of vertices and edges in G, respectively. We use $N(v)$ to denote the set of the neighbors of vertex v, i.e. the set of vertices each of which has an edge connecting to v. We also use $d(v) = |N(v)|$ to denote the degree of vertex v.

Vertex Ordering. We define a total ordering o over the vertices, where $o(v)$ denotes the rank of vertex v, i.e., 1 plus the number of vertices that are before v in the ordering. Obviously, $1 \leq o(v) \leq n$. Given an integer i, we use v_i to denote the vertex with rank i, i.e., $o(v_i) = i$. Given a vertex v, we use $N^-(v) = \{u \mid u \in N(v), o(u) < o(v)\}$ to denote the set of neighbor vertices of v which are ranked before v.

Subgraphs. Given a subset $V' \subseteq V$ of vertices, we define $G(V')$ as the subgraph of G induced by vertex set V', i.e., the edge set of $G(V')$ equals $E' = \{(u, v) \mid u \in V', v \in V', (u, v) \in E\}$. We use $G(v)$ (resp. $G^-(v)$) to denote the subgraph of G induced by vertex set $N(v)$ (resp. $N^-(v)$), i.e., $G(v) = G(N(v))$ (resp. $G^-(v) = G(N^-(v))$).

Cliques. If $G(V')$ is a complete graph where every pair of vertices are connected (i.e. $|E'| = |V'| \cdot (|V'| - 1)/2$), we call $G(V')$ a clique in G, and call $|V'|$ the size of the clique. In particular, if $|V'| = 3$, then we call $G(V')$ a triangle. We use $c(G)$ to denote the size of the largest clique in G.

Maximal Cliques. Let C be a clique in G, and we also abuse the notation C to denote the vertex set of this clique. If there does not exist another clique C' in G, such that $C' \supset C$, then we say that clique C is a maximal clique in G.

Quasi-Cliques. Given a density threshold $\gamma \leq 1$ and a subgraph $G(V')$ of G, if for every vertex $v' \in V'$, its degree in the subgraph $G(V')$ satisfies $d(v') \geq \gamma \cdot (|V'| - 1)$, then we say that $G(V')$ is a γ-quasi-clique in G. To ensure the γ-quasi-clique to be a connected graph, we require $\gamma \geq 0.5$.

Intuitively, a quasi-clique relaxes the requirement of a clique where every vertex is connected to every other vertex, into that every vertex is connected to the majority of other vertices in V'. It is a more realistic model for a social community.

k-core. The k-core of a graph G is the maximal subgraph such that every vertex has degree at least k. It can be found by keeping removing vertices that have degree less than k. We call $k(G)$ as the core number of G if k is the largest integer such that the k-core of G is not empty.

2.2 Problem Statement

Without loss of generality, this work studies four specific problems on top of our parallel framework, and we use the graph in Figure 1 to illustrate the concepts.

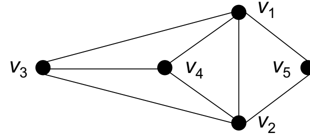

Fig. 1. An Example Input Graph

Triangle Counting. The problem counts the number of triangles in an input graph G. For example, there are 5 triangles $\{v_1, v_2, v_3\}$, $\{v_1, v_2, v_4\}$, $\{v_1, v_2, v_5\}$, $\{v_1, v_3, v_4\}$ and $\{v_2, v_3, v_4\}$ in Figure 1.

Clique Counting. The problem counts the number of cliques in G with different sizes of 1, 2, 3, ..., $c(G)$, respectively. In Figure 1, there are 5 size-1 cliques (i.e., all 5 vertices), 8 size-2 cliques (i.e., all 8 edges), 5 size-3 cliques (i.e., 5 triangles) and 1 size-4 clique (i.e., $\{v_1, v_2, v_3, v_4\}$) in the example graph.

Maximal Clique Listing. The problem lists all the maximal cliques in G. For example, there are 2 maximal cliques $\{v_1, v_2, v_3, v_4\}$ and $\{v_1, v_2, v_5\}$ in Figure 1.

Quasi-Clique Listing. Given a real number $0.5 \leq \gamma \leq 1$ and an integer s, the problem lists all the γ-quasi-cliques in G whose sizes are at least s. For example, there are 3 γ-quasi-cliques $\{v_1, v_2, v_3, v_4\}$, $\{v_1, v_2, v_3, v_5\}$ and $\{v_1, v_2, v_4, v_5\}$ in Figure 1 with $\gamma = 0.6$ and $s = 4$.

3 The Parallel Framework

Overview. In this section, we introduce our parallel framework generic clique-like subgraph counting and listing, which consists of three phases: tasks partitioning, parallel execution and result aggregation, which are described as follows:

- **Task Partitioning.** The framework first loads an input graph G from a file, and then computes a total ordering of the vertices in G. It then divides the computation workloads into multiple tasks each with a bounded cost, and adds them into a concurrent queue [16] to be fetched and processed concurrently by the computing threads.

  We will discuss the vertex ordering schemes in Section 3.1, and introduce our task partitioning method in Section 3.2.
- **Parallel Execution.** Multiple threads are executed concurrently in this phase. Threads are numbered with IDs $i = 1, 2, 3, ..., t$, where t is the total number of threads. The threads keep fetching tasks from the queue for processing, until the task queue becomes empty.

  We will discuss how to compute the tasks in Section 3.3. Once the queue is empty, the idle threads will steal works from the busy threads, so that the workloads are dynamically balanced. The threads will terminate when they all become idle and the task queue is empty.

- **Result Aggregation.** For listing problems, each thread will store the subgraph results in a local buffer of bounded size, and will flush them to disk when the buffer is full, to empty the buffer for keeping more results. After all threads finish running, we can obtain the final results by concatenating the output files of all threads. For counting problems, each thread will maintain its own counter, and their values are summed in the end to get the final count.

3.1 Vertex Ordering Schemes

We assign ranks to vertices using three ordering schemes that are commonly used in existing work [26, 25] as listed below. However, our novelty lies in that we further adjust the resulting ordering by putting a pivot vertex and its neighbors in front of all the other vertices. We will compare all of these ordering schemes in Section 4.

- **Scheme 1: Original Ordering.** This ordering scheme simply assigns the rank of the vertices according to the order that they appear in the input file (which is an edge list). Namely, the two end vertices of the first edge in the input file are ranked with 1 and 2, and the two end vertices of the second edge are ranked with 3 and 4 (if they are different from the first two vertices), and so on.
- **Scheme 2: Static Degree Ordering.** This ordering scheme arranges the vertices by descending order of their original degree in the input graph. Namely, the vertex with the highest degree is ranked with 1, and the vertex with the second highest degree is ranked with 2, and so on.
- **Scheme 3: Dynamic Degree Ordering.** This ordering scheme arranges each vertex v according to its degree in the subgraph induced by v plus those vertices ranked before v. Namely, the vertex with the lowest degree in the original graph is ranked with n, and the vertex with the lowest degree in the subgraph induced by the remaining $n-1$ vertices is ranked with $n-1$, and the vertex with the lowest degree in the subgraph induced by the remaining $n-2$ vertices is ranked with $n-2$, and so on.

Pivot Vertex Reordering. After vertices are ordered as above, we propose to further adjust the order of the vertices by prioritizing a pivot vertex and all of its neighbors to the top. After making this adjustment, a selected pivot vertex v is ranked with 1, and its neighbors are ranked with $2, 3, ..., d(v)+1$, respectively (in an arbitrary order), and then the remaining vertices are ranked with $d(v)+2, d(v)+3, ..., n$ respectively (keeping the same order as in the original ordering scheme). We select the vertex v with the highest degree as the pivot vertex, since v tends to have the heaviest workload and the task partitioning method to be described in Section 3.2 will separate it from the rest of the workloads for further divide-and-conquer to distribute the workload among multiple threads.

3.2 Task Partitioning

For simplicity, let $\mathcal{A}(G', S)$ be the task of counting or listing clique subgraphs in G' given that vertices in S are already assumed to be included in a clique subgraph found by the task. Typically, S refers to those vertices already considered.

Let us denote $V_i = \{v_1, v_2, \ldots, v_i\}$. Then, we can divide the "root" task of computing $\mathcal{A}(G, \emptyset)$ into two subtasks as follows:

$$\begin{aligned} \mathcal{A}(G, \emptyset) &= \mathcal{A}(G(V_n), \emptyset) \\ \to \mathcal{A}(G(V_{n-1}), \emptyset) &\cup \mathcal{A}(G^-(v_n), \{v_n\}). \end{aligned} \tag{1}$$

In other words, we consider two disjoint cases: (1) $\mathcal{A}(G(V_{n-1}), \emptyset)$ finds those clique-like subgraphs that do not contain v_n, where $V_i = \{v_1, v_2, ..., v_i\}$, and (2) $\mathcal{A}(G^-(v_n), \{v_n\})$ finds those clique-like subgraphs that contain v_n.

We can similarly divide the first subtask $\mathcal{A}(G(V_{n-1}), \emptyset)$ as follows:

$$\begin{aligned} &\mathcal{A}(G(V_{n-1}), \emptyset) \\ &\to \mathcal{A}(G(V_{n-2}), \emptyset) \cup \mathcal{A}(G^-(v_{n-1}), \{v_{n-1}\}). \end{aligned} \tag{2}$$

In general, we can keep recursively dividing the first subtask $\mathcal{A}(G(V_{n-i}), \emptyset)$ as:

$$\begin{aligned} &\mathcal{A}(G(V_{n-i}), \emptyset) \\ &\to \mathcal{A}(G(V_{n-i-1}), \emptyset) \cup \mathcal{A}(G^-(v_{n-i}), \{v_{n-i}\}). \end{aligned} \tag{3}$$

In the end, we will obtain:

$$\begin{aligned} &\mathcal{A}(G, \emptyset) \\ &\to \mathcal{A}(G(V_{d(v_1)+1}), \emptyset) \cup \bigcup_{i=d(v_1)+2}^{n} \mathcal{A}(G^-(v_i), \{v_i\}). \end{aligned} \tag{4}$$

If v_1 is the pivot vertex with the highest degree, and its $d(v_1)$ neighbors are also ordered right after v_1 as done by our pivot vertex reordering approach, then task $\mathcal{A}(G(V_{d(v_1)+1}), \emptyset)$ in Equation (4) essentially performs the original listing/counting task on the 1-ego network of v_1. We call this task as the pivot task, and call $\bigcup_{i=d(v_1)+2}^{n} \mathcal{A}(G^-(v_i), \{v_i\})$ the list of minor tasks.

When pivot vertex reordering is used together with our vertex ordering Scheme 3 "dynamic degree ordering", we can show that the size of a minor task is bounded by $k(G)$, the core number of G. This result is formalized by Lemma 1 below.

Lemma 1. *Given a graph $G^-(v_i)$ $(i > d(v_1) + 1)$ of a minor task, the number of vertices in the graph is bounded by $k(G)$.*

Table 1. Maximum Degree v.s. Core Number

Dataset	n	m	$d(v_1)$	$k(G)$
Google	875713	4322051	6353	44
Youtube	1134890	2987624	28754	51
Patents	3774768	16518947	793	64
Flixster	2523386	7918801	1474	68
Skitter	1696415	11095298	35455	111
Wiki	2394385	4659565	100032	131

Proof: According to our scheme of dynamic pivot vertex reordering, vertex v_n has the lowest degree, followed by v_{n-1}, and so on. Therefore, our recursive division steps are equivalent to iteratively removing a vertex with the lowest degree at a time. This corresponds exactly to the algorithm of k-core decomposition [14], and thus at any step, $d(v_i) \leq k(G)$ and thus $G^{-}(v_i)$ has no more than $k(G)$ vertices.

In a real graph, the maximum degree $d(v_1)$ is often much larger than $k(G)$ (which is typically not much larger than 100), and thus most workload is attributed to the pivot task. In other words, the pivot task may need further divide-and-conquer to distribute its workload; while a minor task can directly be processed by a single thread. We show the maximum vertex degree and the core number of the 6 real graphs used in our experiments in Table 1, where we see that $d(v_1) \gg k(G)$.

Even though Lemma 1 may not always hold for the other two vertex ordering schemes, the pivot task is the same and is always the bottleneck of computing workloads, and thus our solution of partitioning the pivot task is still valid. For quasi-cliques, the task partitioning method still works but $G^{-}(v)$ should be computed using v's two-hop neighborhood (rather than one-hop).

Pivot Task Partitioning. Refer back to Equation (4) again. For maximal clique/quasi-clique counting/listing, the pivot task $\mathcal{A}(G(V_{d(v_1)+1}), \emptyset)$ is equivalent to $\mathcal{A}(G(V_{d(v_1)+1} - \{v_1\}), \{v_1\})$, i.e., assuming v_1 is in a clique/quasi-clique found and continues to examine the subgraph induced by v_1's neighbors. This is because v_1 is connected to every vertex in $G(V_{d(v_1)+1})$ and thus any clique/quasi-clique in it without v_1 cannot be maximal.

Similarly, for triangle counting, the pivot task $\mathcal{A}(G(V_{d(v_1)+1}), \emptyset)$ can be divided into two cases: (1) count the triangles that contain v_1, which is essentially the number of edges in $G(V_{d(v_1)+1} - \{v_1\})$ (as the two end vertices of an edge also connects to v_1); (2) count the triangles that do not contain v_1, which essentially counts triangles in $G(V_{d(v_1)+1} - \{v_1\})$, another triangle counting task that can be recursively solved. The overall count is just their sum.

To summarize, unlike in Equation (1) where a task generates two subtasks, a pivot task $\mathcal{A}(G(V_{d(v_1)+1}), \emptyset)$ only generates one subtask computed over the graph $G(V_{d(v_1)+1} - \{v_1\})$.

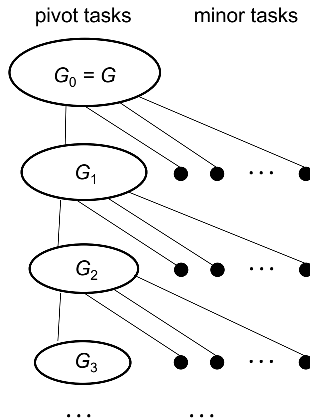

Fig. 2. Task Partitioning

Since a pivot task can be time-consuming, we can recursively divide the pivot task over $G' = G(V_{d(v_1)+1} - \{v_1\})$ into a new pivot task and a list of minor tasks. Specifically, we can perform dynamic pivot vertex reordering in G'; let v_1' be the new pivot vertex with the maximum degree in G', then we can obtain a level-2 pivot task for v_1' and a list of minor tasks. The pivot task for v_1' can be further partitioned if its graph is still too big.

In general, for a task with a set S of already-selected vertices, we call $|S| = \ell$ the level of the task. The pivot task at level-ℓ can generate a level-$(\ell+1)$ pivot task and a list of level-$(\ell+1)$ minor tasks. While a minor task is always processed by a single thread.

Task Representation. After phase 1 "task partitioning" we obtain at most one pivot task at each level, as well as a list of minor tasks. Since there are many minor tasks (e.g., up to n at level-1), it is costly to enqueue them one by one to the task queue, and for computing threads to fetch them one at a time. To reduce this cost, we represent a group of minor tasks at each level by a range; for example, the group of all minor tasks $\bigcup_{i=d(v_1)+2}^{n} \mathcal{A}(G^-(v_i), \{v_i\})$ can be simply denoted by $[d(v_1) + 2, n]$.

Generally, for a minor task at level $\ell = |S|$ over graph G' with n' vertices and v_1' be the pivot vertex, the complete group of tasks at this level can be denoted by range $[d(v_1') + 2, n']$, where we abuse $d(.)$ to measure vertex degree in G'.

While a minor task is usually efficient to compute, a group of them may contain much workload and may need to be distributed to multiple cores for concurrently processing. In this case, we may evenly split the task range into subranges and assign them to different computing threads.

Therefore, in the task queue, it is sufficient to use $\langle \ell, [a, b] \rangle$ to denote a batch of tasks at level-ℓ and with range $[a, b]$, and this is the basic unit to be fetched by computing threads for processing (though the batch can be split to create new batches for fetching when another thread is idle).

Table 2. Dataset Sizes and Properties

Dataset	n	m	$d(v_1)$	$k(G)$	$c(G)$
Google	875713	4322051	6353	44	44
Youtube	1134890	2987624	28754	51	17
Patents	3774768	16518947	793	64	11
Flixster	2523386	7918801	1474	68	31
Skitter	1696415	11095298	35455	111	67
Wiki	2394385	4659565	100032	131	26

Task Initialization. From level 1, we keeps partition the pivot task at each level until when a newly-generated pivot task has an empty graph. Let the last level be ℓ_{max}, then this task initialization approach generates ℓ_{max} group of minor tasks. Figure 2 illustrates this task generation process.

3.3 Task Computation

As long as the task queue Q is not empty, a computing thread will keep dequeueing a batch of minor tasks $\langle \ell, [a, b] \rangle$ from Q for computation. The thread will compute the tasks within the range $[a, b]$ one by one. Each task is computed by a serial recursive counting/list algorithm, which consumes a stable and small amount of memory as the search space tree is traversed in a depth-first manner.

Subgraph Construction. The pivot tasks' subgraphs G_1, G_2, ..., $G_{\ell_{max}}$ are kept in the memory after task partitioning, so that computing threads can read them whenever needed. The subgraph of a level-ℓ (minor) task can thus be incrementally constructed from $G_{\ell-1}$ for serial processing, rather than constructed from scratch from G. We remark that the former is faster since we only need to examine a smaller graph (and thus less edges/adjacency list items).

4 Performance Evaluation

This section evaluates the performance of the various algorithms on top of parallel framework using large real graph datasets. All the experiments were run on a Linux server with 40 3GHz CPU cores and 32GB memory. The programs were written in C++ and compiled with GCC.

For quasi-clique listing, we use parameters $\gamma = 0.8$ and $s = k(G)$ (i.e., G's core number), which essentially finds quasi-cliques from the $\gamma(s-1)$-core of G. We report computation time in the unit of seconds.

Datasets. We used 6 graph datasets in our experiments as shown in Table 2, which correspond to different types of real-world networks.

Specifically, *Google* [1] is a web graph of Google; *Youtube* [2] is the social network of Youtube users and their connections; *Patents* [3] is the US Patent citation network; *Flixster* [4] is the social network of a movie rating site; *Skitter* [5]

Table 3. The Computation Cost

Problem	Dataset	Original	Static	Dynamic
Triangle Counting	Google	52.86	42.76	**40.03**
	Youtube	64.74	41.07	**38.23**
	Patents	223.32	214.82	**182.26**
	Flixster	181.86	135.34	**119.70**
	Skitter	312.54	210.47	**189.48**
	Wiki	174.88	86.70	**78.90**
Clique Counting	Google	66.72	67.84	**63.75**
	Youtube	47.53	47.26	**45.64**
	Patents	190.76	173.27	**155.68**
	Flixster	285.04	238.15	**189.00**
	Skitter	3052.21	2868.00	**2759.43**
	Wiki	2949.47	2823.36	**2179.24**
Maximal Clique Listing	Google	210.37	**203.82**	211.82
	Youtube	188.20	**185.88**	186.74
	Patents	593.28	**601.10**	606.32
	Flixster	1201.00	1055.19	**905.90**
	Skitter	8287.91	8348.78	**7590.01**
	Wiki	10619.76	10160.41	**7973.61**
Quasi-Clique Listing	Google	47840.84	**47532.52**	48344.17
	Youtube	**62716.50**	75432.17	65563.62
	Wiki	126553.08	94453.64	**82456.81**

is an Internet topology graph; and finally, *Wiki* [6] is a user communication network from Wikipedia.

In Table 2, $d(v_1)$ means the maximum degree, $k(G)$ means the core number, and $c(G)$ means the size of the maximum clique. The datasets are listed in ascending order of their core numbers $k(G)$.

4.1 Experiments on Vertex Ordering Schemes

We first conduct experiments to compare different vertex ordering schemes. For each problem and each dataset, we run a single-threaded program with each of our 3 proposed ordering schemes "original order", "static degree", and "dynamic degree", respectively (c.f. Section 3.1). The ordering is adjusted by moving the pivot vertex and its neighbors ahead.

Table 3 reports the computation time of various problems on various datasets using the 3 schemes. We do not report the time of quasi-clique listing on *Patents*, *Flixster* and *Skitter* since they ran over 48 hours due to the giant search space [17] and are thus killed.

From Table 3, we can see that the dynamic degree ordering scheme (along with pivot vertex reordering) is a clear and consistent winner, and therefore we adopt this vertex ordering scheme in the following experiments.

Comparison with Existing Work on Clique Counting. The works of [10] and [11] proposed an algorithm called FFF_k for counting cliques with small

Table 4. Comparison on Clique Counting

Dataset	FFF_7	Our Algorithm	Speedup
Google	**48.62**	70.82	0.69
Youtube	**38.33**	54.10	0.71
Patents	**105.62**	186.21	0.57
Flixster	458.68	**246.38**	1.86
Skitter	5491.22	**2890.53**	2.20
Wiki	4220.07	**2832.20**	1.49

Table 5. Comparison on Maximal Clique Listing

Dataset	Existing Approach	Our Approach	Speedup
Google	361.07	**244.30**	1.48
Youtube	236.90	**217.01**	1.09
Patents	**769.63**	780.18	0.99
Flixster	1146.16	**972.42**	1.18
Skitter	9571.65	**7693.29**	1.24
Wiki	9676.89	**8017.03**	1.21

sizes k. The ordering scheme they used is essentially the static degree ordering defined in our Section 3.1 (without pivot vertex reordering).

To explore whether pivot vertex reordering reduces the workload, we compare our clique counting algorithm with FFF_k. Since the largest k used in their experiments is $k = 7$, we run our clique counting algorithm with static degree ordering plus pivot vertex reordering as an equivalence to FFF_7. Note that we are counting cliques with very large sizes up to $c(G) = 67$. The results are reported in Table 4, where we can see that our computation time is comparable to FFF_7 when the computation time is short, but much faster when the computation time is long, which justifies the need of pivot vertex reordering.

Comparison with Existing Work on Maximal Clique Listing. The work of [26] and [25] proposed an algorithm for maximal clique listing. They explored the static and dynamic degree ordering schemes described in our Section 3.1. They decompose the task into subtasks $M(G^{-}(v)) \oplus \{v\}$ for all $v \in V$, and for each subtask, they solve it using pivot vertex reordering. However, pivot vertex reordering is not performed on the root task $M(G)$, and it is interesting to see how this affects the amount of overall workloads.

Table 5 reports the computation time of our algorithm with/without root-task pivot vertex reordering, and we can see that after applying the pivot vertex reordering for the root task, the computation time is consistently improved.

Another problem with [26] and [25] is that they do not support recursive partitioning of a pivot task which usually contains a lot of workloads needing parallel computation.

Table 6. Parallel Computation on Wiki Dataset

Problem	# of Threads	1	2	4	8	16	32
Triangle	Time	78.89	39.49	21.86	11.09	7.35	5.83
Counting	Speedup	1.00	2.00	3.61	7.12	10.74	13.53
Clique	Time	2179.24	1206.20	612.21	323.43	181.88	119.99
Counting	Speedup	1.00	1.81	3.56	6.74	11.98	18.16
Maximal	Time	7973.61	4608.41	2311.60	1228.39	657.33	443.84
Clique	Speedup	1.00	1.73	3.45	6.49	12.13	17.96
Quasi-	Time	82500.23	45493.54	24127.00	14223.90	8202.75	5139.22
Clique	Speedup	1.00	1.81	3.42	5.80	10.06	16.05

4.2 Experiments on Parallel Computation

We now explore how our parallel framework scales up with the number of threads using various applications and datasets. We run our programs with 1, 2, 4, 8, 16 and 32 threads, respectively, for testing vertical scalability. We report both the computation time and speedup ratio (w.r.t. single-threaded execution). Dynamic degree ordering plus pivot vertex reordering is used for task partitioning in all the following experiments. Due to the space limitation, we only report the results on *Wiki* dataset.

Table 6 shows the computation time and the corresponding speedup ratio w.r.t. single-threaded execution. We can see that the speedup ratio increases near-linearly with the number of threads all the way till 16, but the increment of the speedup trend slows down when we run 32 threads, possibly because the overheads of task generation and fetching stand out compared with the significantly amortized task computation time. The increment of the speedups are similar for clique counting and quasi-clique listing. Overall, our framework demonstrates a near-optimal speedup.

Compared with maximal clique listing where the vertical scalability slows down at 32 threads, triangle counting slows down earlier at 16 threads mainly because the computing workloads of triangle counting is much lower than that of maximal clique listing, and thus the other overheads stand out sooner.

5 Conclusion

In this paper, we proposed a framework of task partitioning and workload balancing for triangle counting, clique counting, maximal clique listing and quasi-clique listing. For task partitioning, we proposed pivot path partitioning, which recursively explores a node in the search tree which has the most heavily workload. For workload distribution, we dynamically balanced the workload by a work stealing strategy. Our experiments showed that our pivot path partitioning strategy reduced the total amount of work to be computed. We also demonstrated that our parallel executions have a almost ideal speedup ratio for up to 32 threads.

6 Acknowledgement

Yang and Zhou were supported by National Natural Science Foundation of China (NSFC) under grant No. U1636205, Yan and Guo were partially supported by NSF OAC-1755464 and NSF DGE-1723250.

References

1. https://snap.stanford.edu/data/web-Google.html
2. https://snap.stanford.edu/data/com-Youtube.html
3. https://snap.stanford.edu/data/cit-Patents.html
4. http://konect.uni-koblenz.de/networks/flixster
5. https://snap.stanford.edu/data/as-skitter.html
6. https://snap.stanford.edu/data/wiki-Talk.html
7. Blumofe, R.D., Leiserson, C.E.: Scheduling multithreaded computations by work stealing. J. ACM **46**(5), 720–748 (1999). https://doi.org/10.1145/324133.324234, http://doi.acm.org/10.1145/324133.324234
8. Cheng, J., Zhu, L., Ke, Y., Chu, S.: Fast algorithms for maximal clique enumeration with limited memory. In: The 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '12, Beijing, China, August 12-16, 2012. pp. 1240–1248 (2012), http://doi.acm.org/10.1145/2339530.2339724
9. Du, N., Wu, B., Xu, L., Wang, B., Pei, X.: A parallel algorithm for enumerating all maximal cliques in complex network. In: Workshops Proceedings of the 6th IEEE International Conference on Data Mining (ICDM 2006), 18-22 December 2006, Hong Kong, China. pp. 320–324 (2006), https://doi.org/10.1109/ICDMW.2006.17
10. Finocchi, I., Finocchi, M., Fusco, E.G.: Counting small cliques in mapreduce. CoRR **abs/1403.0734** (2014), http://arxiv.org/abs/1403.0734
11. Finocchi, I., Finocchi, M., Fusco, E.G.: Clique counting in mapreduce: Algorithms and experiments. ACM Journal of Experimental Algorithmics **20**, 1.7:1–1.7:20 (2015). https://doi.org/10.1145/2794080, http://doi.acm.org/10.1145/2794080
12. Kanewala, T.A., Zalewski, M., Lumsdaine, A.: A distributed algorithm for γ-quasi-clique extractions in massive graphs. In: Communications in Computer and Information Science. pp. 241:422–431 (2011)
13. Kumpula, J.M., Kivela, M., Kaski, K., , Saramaki, J.: Sequential algorithm for fast clique percolation. In: Physical Review E Statistical Nonlinear and Soft Matter Physics. p. 78(2):026109 (2008)
14. Matula, D.W., Beck, L.L.: Smallest-last ordering and clustering and graph coloring algorithms. J. ACM **30**(3), 417–427 (1983). https://doi.org/10.1145/2402.322385, http://doi.acm.org/10.1145/2402.322385
15. McCune, R.R., Weninger, T., Madey, G.: Thinking like a vertex: A survey of vertex-centric frameworks for large-scale distributed graph processing. ACM Comput. Surv. **48**(2), 25:1–25:39 (2015). https://doi.org/10.1145/2818185, http://doi.acm.org/10.1145/2818185
16. Michael, M.M., Scott, M.L.: Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In: Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing, Philadelphia, Pennsylvania, USA, May 23-26, 1996. pp. 267–275 (1996), http://doi.acm.org/10.1145/248052.248106

17. Pardalos, P.M., Rebennack, S.: Computational challenges with cliques, quasi-cliques and clique partitions in graphs. In: Experimental Algorithms, 9th International Symposium, SEA 2010, Ischia Island, Naples, Italy, May 20-22, 2010. Proceedings. pp. 13–22 (2010), https://doi.org/10.1007/978-3-642-13193-6_2
18. Pardalos, P.M., Xue, J.: The maximum clique problem. Journal of global Optimization **4**(3), 301–328 (1994)
19. Ribeiro, P.M.P., Silva, F.M.A., Lopes, L.M.B.: Efficient parallel subgraph counting using g-tries. In: Proceedings of the 2010 IEEE International Conference on Cluster Computing, Heraklion, Crete, Greece, 20-24 September, 2010. pp. 217–226 (2010), https://doi.org/10.1109/CLUSTER.2010.27
20. Schmidt, M.C., Samatova, N.F., Thomas, K., Park, B.: A scalable, parallel algorithm for maximal clique enumeration. J. Parallel Distrib. Comput. **69**(4), 417–428 (2009), https://doi.org/10.1016/j.jpdc.2009.01.003
21. Svendsen, M., Mukherjee, A.P., Tirthapura, S.: Mining maximal cliques from a large graph using mapreduce: Tackling highly uneven subproblem sizes. J. Parallel Distrib. Comput. **79-80**, 104–114 (2015), https://doi.org/10.1016/j.jpdc.2014.08.011
22. Tsourakakis, C.E., Bonchi, F., Gionis, A., Gullo, F., Tsiarli, M.A.: Denser than the densest subgraph: extracting optimal quasi-cliques with quality guarantees. In: The 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD 2013, Chicago, IL, USA, August 11-14, 2013. pp. 104–112 (2013), http://doi.acm.org/10.1145/2487575.2487645
23. Wu, B., Yang, S., Zhao, H., Wang, B.: A distributed algorithm to enumerate all maximal cliques in mapreduce. In: Fourth International Conference on Frontier of Computer Science and Technology, FCST 2009, Shanghai, China, 17-19 December, 2009. pp. 45–51 (2009), https://doi.org/10.1109/FCST.2009.30
24. Xiang, J., Guo, C., Aboulnaga, A.: Scalable maximum clique computation using mapreduce. In: 29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013. pp. 74–85 (2013), https://doi.org/10.1109/ICDE.2013.6544815
25. Xu, Y., Cheng, J., Fu, A.W.: Distributed maximal clique computation and management. IEEE Trans. Services Computing **9**(1), 110–122 (2016), https://doi.org/10.1109/TSC.2015.2479225
26. Xu, Y., Cheng, J., Fu, A.W., Bu, Y.: Distributed maximal clique computation. In: 2014 IEEE International Congress on Big Data, Anchorage, AK, USA, June 27 - July 2, 2014. pp. 160–167 (2014), https://doi.org/10.1109/BigData.Congress.2014.31
27. Yan, D., Bu, Y., Tian, Y., Deshpande, A.: Big graph analytics platforms. Foundations and Trends in Databases **7**(1-2), 1–195 (2017), https://doi.org/10.1561/1900000056