Parallelizing Clique and Quasi-Clique Detection over Graph Data

Qun Chen [#1], Zhanhuai Li [#2], Zachary G. Ives [*3], Li You [#4]

[#] *School of Computing, Northwestern Polytechnical University Xi'an, China*

[1,2,4] {chenbenben,lizhh,youli}@nwpu.edu.cn

[*] *Department of Computer and Information Science, University of Pennsylvania Philadelphia, U.S.A.*

[3] zives@cis.upenn.edu

Abstract—In a wide variety of emerging data-intensive applications, such as social network analysis, Web document clustering, entity resolution, and detection of consistently co-expressed genes in systems biology, the detection of *dense subgraphs* (cliques and approximate or *quasi*-cliques) is an essential component. Unfortunately, these problems are NP-Complete and thus computationally intensive at scale. Hence there is a need to come up with techniques for distributing the computation across multiple machines such that the computation, which is too time-consuming on a single machine, can be computed quickly on a sufficiently large machine cluster. In this paper, we propose a new approach for maximal clique and quasi-clique enumeration over graph data and develop corresponding parallel solutions.

We first introduce a novel algorithm which identifies dense subgraphs by recursive graph partitioning. Given a connected graph $G = (V, E)$, the algorithm has a space complexity of $O(|E|)$ and a time complexity of $O(|E|\omega(G)\mu(G))$, where $\omega(G)$ and $\mu(G)$ represent the number of vertices in the maximum clique (quasi-clique) and the number of maximal cliques (quasi-cliques) in G respectively. It achieves so far the best theoretical upperbound on the runtime as a polynomial with respect to the number of maximal cliques (quasi-cliques) under the $O(|E|)$ space constraint. We then demonstrate how graph partitioning enables effective load balancing and efficient parallel implementation by developing corresponding parallel solutions and implementing them on MapReduce, a popular shared-nothing parallel framework. Finally, we evaluate the performance of the proposed approach on real and synthetic graph data and show that it performs considerably better than existing approaches in both centralized and parallel settings. It achieves a speedup of up to 10x over existing approaches on large graphs. In the parallel setting, our proposed approach is implemented and evaluated on MapReduce but its implementation can easily generalize to other shared-nothing or shared-memory parallel frameworks.

1 Introduction

A variety of emerging applications are focused on computations over data modeled as a graph: examples include finding groups of actors or communities in social networks [27], Web mining [31], entity resolution [10], graph mining [38], and detection of consistently co-expressed gene groups in systems biology [23]. For the problems just cited, as well as a number of others, a critical component of the analysis is the detection of cliques (fully connected components), and in some cases highly connected components or *quasi*-cliques, in the structure of the network graph. For instance, for entity resolution, each clique or quasi-clique may represent a block of entities that might be merged.

Clique and quasi-clique detection is NP-Complete. Hence a great deal of effort has been spent on efficient search algorithms [11], [20], [21], [18], [39], [19], [37], [33], [26]. One category of algorithms for maximal clique enumeration is related to the classical algorithm proposed by Bron and Kerbosch (*BK*) [11]. The *BK* algorithm uses a backtracking technique to explore search space and reduces redundant traversals by remembering the search paths it has already visited. In practice, the *BK* algorithm has been widely reported as being faster than its alternatives [22], [17]. Another category of algorithms [37], [33], [26] use a *reverse search* strategy. One key feature of these reverse search algorithms is that it is possible to define an upperbound on their runtime as a polynomial with respect to the number of maximal cliques in a graph.

Data-intensive applications usually require clique and quasi-clique detection to be operated over large graphs, hence there is a need to parallelize it such that the detection, which is too time-consuming on a single machine, can be quickly computed on a sufficiently large machine cluster. There have been a variety of proposals that divide the graph into smaller (sometimes overlapping) subcomponents and exploit parallelism to improve performance [41], [29], [42], [30], [14]. Built on classical sequential algorithms, they have been empirically shown to speed computation in massive networks. However, due to NP-Completeness of clique and quasi-clique computation, the performance of existing parallel approaches is limited by how evenly the graph is partitioned. (In fact, as we show in Section 5.2, their performance is quite sensitive to particular graph characteristics.)

This paper presents a new approach for maximal clique and quasi-clique enumeration. It computes maximal dense subgraphs by recursive graph partitioning. The first key

feature of the new approach is that in a centralized setting, it achieves so far the best theoretical upperbound on the runtime as a polynomial with respect to the number of maximal cliques and quasi-cliques under polynomial space constraints. Secondly, it enables effective load balancing. It recursively divides a graph until each task is sufficiently small to be processed in parallel. Compared with the existing parallel approaches based on the classical search algorithms, our new approach is based on a novel algorithm with provably better complexity bounds and has finer parallelizability. The two features combined result in its better parallel performance.

Two common parallel frameworks for graph data processing are the MapReduce model [4] and the Bulk Synchronous Parallel (BSP) model [24]. The underlying computation models of MapReduce and BSP are essentially isomorphic. Both frameworks enable staged computation where many subtasks are run in parallel, then data is exchanged and new subtasks are added, then another stage works over the output of the previous. The difference of the BSP model from MapReduce is that its computation and communication are centered around graph vertices. The platforms based on MapReduce include the open-source Hadoop [2], Pig [15], Dryad [32] and Spark [6]. Pregel [24] was one of the early BSP platforms. The open-source BSP platforms include Giraph [1] and Hama [3]. Our proposed approach is based on iterative data processing and can work with both MapReduce and BSP platforms. In this paper, we choose MapReduce for parallel implementation and evaluation. However, the implementation can easily generalize to other shared-nothing or shared-memory parallel platforms, such as Pregel and MPI. The major contributions of this paper are summarized as follows:

1) We propose a novel algorithm for maximal clique enumeration. Given a connected graph $G = (V, E)$, the algorithm has a space complexity of $O(|E|)$ and a time complexity of $O(|E|\omega(G)\mu(G))$, where $\omega(G)$ and $\mu(G)$ represent the number of vertices in the maximum clique and the number of maximal cliques in G respectively. It achieves so far the best theoretical upperbound on the runtime as a polynomial with respect to the number of maximal cliques under the $O(|E|)$ space constraint.
2) We develop a parallel solution to maximal clique enumeration by parallelizing the proposed algorithm and implementing the corresponding parallel algorithm based on MapReduce. By using graph partitioning to divide the tasks, the proposed solution can effectively parallelize maximal clique computation with improved load balancing.
3) We extend our techniques to also support maximal k-plex enumeration and achieve similar theoretical and practical results.
4) We experimentally evaluate the performance of our proposed approach on a wide variety of graph data available in open-source. Our extensive experiments demonstrate that it performs considerably better than existing approaches in both centralized and parallel settings. In the parallel setting, our approach achieves a speedup of up to 10x over existing approaches on large graphs.

The rest of this paper is organized as follows: Section 2 provides the background information and the description of the existing approaches. Section 3 presents our new sequential algorithms. Section 4 presents our parallel solutions and their MapReduce implementations. Section 5 empirically evaluates the performance of our approach on real and synthetic datasets. Section 6 discusses related work. Finally, Section 7 concludes this paper with some thoughts on future work.

2 PRELIMINARIES

2.1 Definition: Cliques and K-Plex Quasi-cliques

A clique is a subgraph in which every pair of vertices is connected by an edge. A quasi-clique usually refers to a dense subgraph in which every vertex is directly connected to most of the other vertices. In this paper, we focus on a type of quasi-clique called *k-plex*, whose formal definition is stated as follows:

Definition 1. *An induced subgraph G_i consisting of a set of vertices V in G is a k-plex if $\forall v \in V$, $deg(v) \geq (|V| - k)$, in which $deg(v)$ represents the degree of vertex v in G_i.*

Obviously 1-plex corresponds to clique. The k-plexes with low values of k (e.g., k= 2 or 3) provide good relaxations of clique that closely resemble the cohesive subgroups existing in real networks. The definitions of a *maximum* clique (k-plex) and a *maximal* clique (k-plex) are as follows:

Definition 2. *A maximum clique (k-plex) in a graph G is a clique (k-plex) with the largest number of vertices.*

Definition 3. *A maximal clique (k-plex) in a graph G is a clique (k-plex) not contained by any other clique (k-plex) in G.*

The problem of maximal clique and k-plex enumeration refers to identifying all the maximal cliques and k-plexes in a given graph G. Since each connected component in G can be processed independently, we assume that G is a connected graph in this paper.

2.2 Classical Sequential Algorithms

For maximal clique enumeration, the BK algorithm [11] has been widely reported as being faster in practice than its alternatives [22], [17]. In this subsection, we sketch a variant of the BK algorithm proposed in [20] by a running example. It is based on the BK algorithm, employing the same pruning methods, and has the optimal worst-case time complexity $O(3^{n/3})$.

The algorithm is in essence a depth-first search augmented with pruning tricks. For each vertex v in a graph G, it iteratively identifies the maximal cliques containing v by constructing and traversing a tree. Note that any

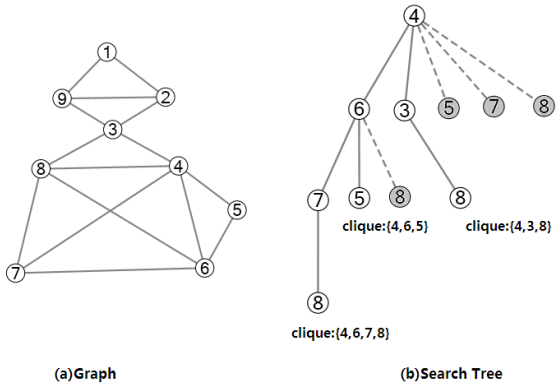


Fig. 1. **An Example Graph and The Search Tree of Vertex 4**

TABLE 1
Traversing Search Tree

step	v	N(v)	N(N(v))
I	4	{3,5,6,7,8}	N(3)={2,8,9},N(5)={6},N(6)={5,7,8}, N(7)={6,8},N(8)={3,6,7}
II	6	{5,7,8}	N(5)={},N(7)={8},N(8)={7}
III	7	{8}	N(8)={}, output clique {4,6,7,8} and backtrack to process the vertex 5

maximal clique containing v is a subgraph of the induced subgraph in G consisting of v and its neighbouring vertices. Consider Vertex 4 in Figure 1(a). Its two-hop information has been recorded in Table 1, in which $N(v)$ and $N(N(v))$ denote v' neighboring vertices and its neighbors' neighbors respectively. The whole search process is sketched in Figure 1(b). The algorithm first identifies the vertex v in $N(4)$ whose $N(v)$ has the biggest intersection set with $N(4)$. This is Vertex 6, and their corresponding intersection set is {5,7,8}. Next, it updates $N(6)$ and $N(N(6))$ by $N(6)$=$N(6) \cap N(4)$ and $N(N(6))$=$N(N(6)) \cap N(6)$ and continues to process Vertex 6 in the same way as Vertex 4. As a result, Vertex 7 is chosen. After $N(7)$ and $N(N(7))$ are updated, $N(8)$ becomes empty. The algorithm therefore outputs the clique consisting of the vertexes {4,6,7,8} and backtracks to the tree level of Vertex 7. Next, Vertex 5 is chosen and processed to output the clique consisting of the vertexes {4,5,6}. The algorithm traverses the whole tree in a recursive process involving the operations of vertex selection, set update, clique generation and backtracking.

Note that for the BK algorithm and its variants, effective pruning methods can be employed to reduce redundant traversals. In the example of Figure 1(b), the sub-trees rooted at the shaded vertices can be pruned and saved from searching. In general, a vertex u directly under another vertex v can be pruned if the search algorithm has traversed a path from v to u in the subtree rooted at the vertex v. In Figure 1(b), the vertex 8 directly under the vertex 6 is pruned because in the subtree rooted at the vertex 6, there exists a path $6 \rightarrow 7 \rightarrow 8$ that has been traversed. For more details on the algorithm and its pruning methods, please refer to [20].

The existing sequential algorithms for maximal k-plex enumeration [8], [7] are usually extensions of the classical algorithms for maximal clique enumeration. They use a depth-first search strategy and similar pruning methods to reduce redundant traversals.

2.3 Background: MapReduce

The MapReduce model processes distributed data across many nodes via three basic phases. In the *Map* phase, it takes an input and produces a list of intermediate key/value pairs without communication between nodes. Next, the *Shuffle* phase repartitions these intermediate pairs according to their keys across nodes. Finally, the *Reduce* phase aggregates the intermediate pairs it receives to produce final results. This process can be repeated by invoking an arbitrary number of additional *Map-Shuffle-Reduce* cycles as necessary.

In this paper, we use Hadoop for parallel evaluation and develop a MapReduce implementation for our approach, in which recursive graph partitioning is programmed in a *Reduce* phase. If implemented on BSP platforms, it can be similarly programmed in a *superstep*. Detailed implementation of our approach on BSP platforms is however beyond the scope of this paper.

2.4 Existing Parallel Approaches

The typical parallel approaches [41], [29], [42], [30] enumerate maximal cliques and k-plexes for different vertices in a graph in parallel. In the rest of this subsection, we describe the idea behind the typical parallel approach for maximal clique enumeration based on MapReduce. The parallel approach for maximal k-plex enumeration [40] is similar except that it invokes centralized k-plex search instead of clique search.

Given a graph G and a vertex v in G, the maximal cliques (k-plexes) of the vertex v refer to the maximal cliques (k-plexes) containing v in G. According to [25], a vertex v's maximal cliques and k-plexes (with low k values) are the induced subgraphs consisting of v and the vertices at most 2 hops away from v in G. In other words, their vertices are members of the vertex set consisting of v, $N(v)$ and $N(N(v))$. Note that even though v's maximal clique only consists of v and its neighboring vertices, clique detection needs the connectivity information between its neighboring vertices.

In the first step, the parallel approach computes each vertex's 2-hop information. It is completed by a MapReduce cycle:

1) In the *Map* phase, it transforms the adjacency list of each vertex u, $(u, \{v_1, v_2, \ldots, v_m\})$, into m triplets with the format of $(v_i, u, \{v_1, \ldots, v_{i-1}, v_{i+1}, \ldots, v_m\})$;
2) In the *Shuffle* phase, it redistributes the triplets according to the key v_i across computing nodes;
3) In the *Reduce* phase, it aggregates the triplets by the key v_i.

The resulting group of triplets with the key v_i includes the complete two-hop information of the vertex v_i in G. In the second step, it proceeds to search for each vertex's maximal cliques in parallel. For the computation on an individual vertex, it simply adopts the classical sequential algorithms.

Note that in the typical approach, enumerating the maximal cliques of a vertex is supposed to be performed on a single machine. In case that the computation on an individual vertex is extremely time-consuming due to the large number of maximal cliques (as we will show in Section 5), it may become a parellel performance bottleneck. The method proposed in [30] can parallelize maximal clique enumeration on an individual vertex. Based on the BK algorithm, it uses candidate path data structures to record the search progress such that any search subtree can be traversed independently. It achieves better load balancing by allowing a computing node to steal some tasks from others when becoming almost idle. The proposed load balancing technique was implemented by MPI, but can easily generalize to other shared-nothing parallel frameworks such as MapReduce. However, its parallel efficiency may still be limited by unevenness of search subtree sizes and the difficulty to estimate the required runtime on a search subtree.

3 Sequential Algorithms

In this section, we propose novel sequential algorithms for maximal clique and k-plex enumeration, prove their complexity bounds and describe their efficient implementation.

3.1 Idea: Graph Partitioning

We illustrate the idea behind the new sequential algorithms by an example of maximal clique detection. As usual, we search for maximal cliques in a graph G by iteratively computing v's maximal cliques for every vertex v in G. Therefore, in the rest of this subsection, we focus on the general problem of identifying the maximal cliques containing a specific vertex v in G. Suppose that G_v represents the induced subgraph of G consisting of v and its neighboring vertices. Obviously, the maximal cliques of the vertex v are contained by G_v. We have our first Lemma:

Lemma 1. *The subgraph G_v is a clique if and only if the degree of each vertex in G_v is $(n-1)$, where n is the total number of vertices in G_v.*

The main challenge is how to identify maximal cliques in G_v if G_v is not a clique. We illustrate the underlying idea by a subgraph G_4 of the example graph shown in Figure 1 (a), which consists of Vertex 4 and its neighboring vertices (i.e., the vertices {4,3,5,6,7,8}). We randomly choose another vertex in G_4, e.g., Vertex 7, as the partitioning anchor and partition G_4 into two parts G_7^+ and G_7^-. G_7^+ denotes the induced subgraph consisting of Vertex 7 and its neighboring vertices in G_4, {4,6,7,8}. G_7^- denotes the induced subgraph of G_4 consisting of all the vertices not in G_7^+, {3,5}, and their neighboring vertices in G_4, {4,6,8}). The subgraphs

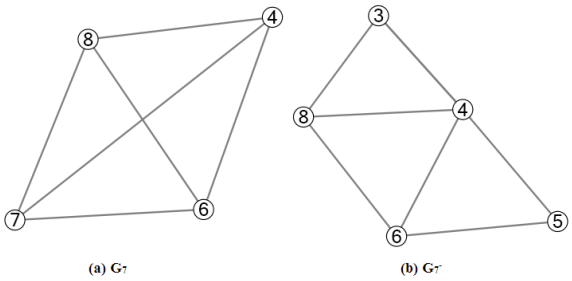


Fig. 2. A Graph Partitioning Example for Clique Detection

G_7^+ and G_7^- are shown in Figure 2 (a) and (b) respectively. We observe that any maximal clique of G_4 is an induced subgraph of either G_7^+ or G_7^-.

Generally, given a graph G_v consisting of vertex v and its neighboring vertices, we randomly select another vertex u in G_v and partition G_v into two subgraphs, G_u^+ and G_u^-. G_u^+ denotes the induced subgraph consisting of vertex u and its neighboring vertices. G_u^- denotes the induced subgraph consisting of all the vertices not in G_u and their neighboring vertices. We have the following Lemma:

Lemma 2. *Any maximal clique of G_v is an induced subgraph of either G_u^+ or G_u^-.*

Proof: If a maximal clique contains the vertex u, it should be an induced subgraph of G_u^+. Otherwise, it should contain at least one vertex not in G_u^+. Suppose that it is the vertex w. As a result, the maximal clique is an induced subgraph of G_w, which consists of vertex w and its neighboring vertices. According to the definition of G_u^-, G_w is obviously an induced subgraph of G_u^-. Therefore, the maximal clique is an induced subgraph of G_u^-. □

According to Lemma 2, maximal clique detection in G_v can be performed by searching for the maximal cliques in G_u^+ and G_u^- independently. The partitioning operation can be recursively invoked until all the resulting subgraphs become cliques. Obviously, all the maximal cliques in the original graph G_v are contained in the set of the resulting cliques. However, a resulting clique generated by the above process of recursive graph partitioning can not be guaranteed to be maximal. Therefore, enumeration algorithms should filter out those which are not maximal.

3.2 Maximal Clique Enumeration

In this subsection, we present the sequential algorithms for maximal clique enumeration, analyze their complexity bounds and give details on their efficient implementation.

3.2.1 A General Algorithm

The general algorithm, as shown in Algorithm 1, enumerates maximal cliques by recursively partitioning a graph. It calls the recursive funtion `enumerateClique(anchor,cand,not)`, which is abstracted in Algorithm 2. The function employs three sets of vertices to record the partitioning progress and prune the subtrees that can not generate maximal cliques:

Algorithm 1 A Sequential Algorithm for Maximal Clique Enumeration

Input: A graph G with vertex set V and edge set E;

1: anchor$\leftarrow \emptyset$;
2: cand$\leftarrow V$;
3: not$\leftarrow \emptyset$;
4: Call enumerateClique(anchor,cand,not);

Algorithm 2 enumerateClique(anchor,cand,not)

1: **if** (G(cand) is a clique) **then**
2: Output the clique G(anchor$\cup$cand);
3: **else**
4: Choose a vertex v in cand;
5: anchor$^-$=anchor;
6: cand$^-$=cand$-\{v\}$;
7: not$^-$=not$\cup\{v\}$;
8: **if** (no vertex in not$^-$ is connected to all the vertices in cand$^-$) **then**
9: enumerateClique(anchor$^-$,cand$^-$,not$^-$);
10: **end if**
11: anchor$^+$=anchor$\cup\{v\}$;
12: cand$^+$=cand$\cap N(v)$;
13: not$^+$=not$\cap N(v)$;
14: **if** (no vertex in not$^+$ is connected to all the vertices in cand$^+$) **then**
15: enumerateClique(anchor$^+$,cand$^+$,not$^+$);
16: **end if**
17: **end if**

1) (anchor set) A set of vertices that have been selected as partitioning anchors and should be contained by the resulting cliques;
2) (cand set) A set of candidate vertices that can serve as partitioning anchors in the following operations;
3) (not set) A set of vertices that are connected to every vertex in the anchor set but could not produce new maximal cliques if combined with the vertices in the anchor set.

We denote the induced subgraph consisting of a set of vertices V_i in G by $G(V_i)$. The recursive function first checks whether the resulting subgraph is a clique (Line 1). If yes, it simply outputs the subgraph. Otherwise, it randomly chooses a vertex v in cand as the partitioning anchor and partitions $G(\texttt{cand})$ into $G(\texttt{cand}^+)$ and $G(\texttt{cand}^-)$. $G(\texttt{cand}^+)$ consists of v and its neighboring vertices in $G(\texttt{cand})$. $G(\texttt{cand}^-)$ consists of all the vertices in $G(\texttt{cand})$ except v. The funtion continues to search for the maximal cliques in $G(\texttt{cand}^-)$ (Lines 5-10) and $G(\texttt{cand}^+)$ (Lines 11-16) respectively. Before executing graph partitioning on $G(\texttt{cand}^-)$ and $G(\texttt{cand}^+)$, the function checks whether the corresponding subgraphs can produce new maximal cliques (in Lines 8 and 14 respectively). It is achieved by inspecting whether there exists a vertex in the current not set that is connected to all the vertices in the current cand set. We have the following theorem:

Theorem 1. *Algorithm 1 exactly returns all the maximal cliques in G.*

Proof: Firstly, if without the pruning operations specified in Lines 8 and 14 of Algorithm 2, all the maximal cliques in G are contained in the set of cliques returned by Algorithm 1.

Secondly, if there exists a vertex in the current not set that is connected to all the vertices in the current cand set, the recursive function can not generate any new maximal clique. Consider a clique C in the induced subgraph $G(\texttt{ancor}\cup\texttt{cand})$ of G. Suppose that a vertex u in the not set is connnected to all the vertices in the cand set. Note that Algorithm 2 ensures that each vertex in not is connected to all the vertices in the anchor set. As a result, u is connected to all the vertices of C. Therefore, the clique C is not maximal.

Finally, any clique returned by Algorithm 1 is maximal. Assume that it returns two cliques, C_1 and C_2, and C_1 is contained by C_2. Suppose that C_1 consists of k vertices, $\{v_1,v_2,\ldots,v_k\}$, and C_2 has an additional vertex u. Also suppose that C_1 is generated by combining the $\texttt{anchor}_1$ set and the $\texttt{cand}_1$ set. Since the vertex u is not in $\texttt{anchor}_1$ but connected to all the vertices in $\texttt{anchor}_1$, its exclusion from $\texttt{cand}_1$ should be a result of a previous graph partitioning operation with u as the partitioning anchor. Therefore, the vertex u should be included in the not set of the corresponding partitioned graph G($\texttt{anchor}^-\cup\texttt{cand}^-$), whose recursive partitioning later generates the clique C_1. Since u is connected to all the vertices in $\texttt{anchor}_1$, Algorithm 2 ensures that it is in the not set of the partitioned graph G($\texttt{anchor}_1\cup\ \texttt{cand}_1$). With u being connected to all the vertices in $\texttt{cand}_1$, Algorithm 2 should have filtered C_1 out. Contradiction. □

A straightforward implementation of Algorithm 2 maintains the vertex sets of anchor, cand and not, as well as the adjacency lists of the vertices in G, by hashtables. Given a connected graph $G=(V,E)$, each line statement in Algorithm 2 can be processed in $O(|E|))$ time. For instance, Line 1 (checking whether G(cand) is a clique) requires to intersect the adjacency list of each vertex in cand with the cand set. It can be achieved by a series of hash look-ups which match the neighboring vertices of each vertex in cand with the entries in the cand hashtable. Lines 8 and 14 can be processed in a similar way. On the space and time complexity of Algorithm 1, we have the following theorem:

Theorem 2. *Given a connected graph G=(V, E), Algorithm 1 has the space complexity of $O(|E|)$ and the time complexity of $O(|E|\omega(G)\mu(G))$, in which $\omega(G)$ and $\mu(G)$ represent the number of vertices in the maximum clique and the number of maximal cliques in G respectivley.*

Proof: We first analyze the space complexity of Algorithm 1. Note that Algorithm 1 processes the $G(\texttt{cand}^-)$ branch before the $G(\texttt{cand}^+)$ branch. Besides a $G(\texttt{cand}^-)$ graph, it also has to store the information of each $G(\texttt{cand}^+)$ on the path from the root to a leaf. Each $G(\texttt{cand}^+)$ results from a partitioning operation with a vertex v_i as partitioning anchor. Since each vertex in the $\texttt{anchor}^+$, $\texttt{cand}^+$ and $\texttt{not}^+$ sets of $G(\texttt{cand}^+)$ (except the vertex v_i itself) should be connected to v_i, the required

space to store $G(\texttt{cand}^+)$ is bound by $O(|E_i|)$, in which E_i represents the set of edges with v_i as one of its end points. It is also observed that each $G(\texttt{cand}^+)$ on the path from the root to a leaf has a distinct partitioning anchor. As a result, the required space to store all $G(\texttt{cand}^+)$ branches is bound by $O(|E|)$. It follows that the space complexity of Algorithm 1 is $O(|E|)$.

Secondly, we analyze its time complexity. Consider the algorithm without pruning (Lines 8 and 14). Obviously, its time complexity is an upperbound on the time complexity of the algorithm with pruning.

The traversal tree generated by the recursive function without pruning is a binary tree, in which each internal node has exactly two children and each leaf node corresponds to a resulting clique. Note that the recursive function returns only distinct cliques. The total number of cliques it returns, which corresponds to the total number of leaves in the binary tree, is at most $O(\omega(G)\mu(G))$. As a result, the size of the binary tree is bounded by $O(\omega(G)\mu(G))$. Accordingly, the total number of invoked graph partitioning operations is bounded by $O(\omega(G)\mu(G))$. Since each invocation of graph partitioning requires $O(|E|)$ time, the time complexity of the recursive function is $O(|E|\omega(G)\mu(G))$. □

Given a connected graph G=(V, E), the known upperbounds on the time complexity with respect to $\mu(G)$ for maximal clique enumeration are given in Table 2, in which n=$|V|$ and m=$|E|$. Except our algorithms, all the results are achieved by reverse search algorithms. In Table 2, $\gamma(G)$ represents the arboricity of G and $M(n)$ represents the time needed to multiply two $(n \times n)$ matrics. The arboricity of a graph is the minimum number of forests into which its edges can be partitioned. Note that for a clique of size n, its arboricity is $\gamma(G)$=$\lceil \frac{n}{2} \rceil$. And for a complete bipartite graph $G_{n,n}$, its size of maximum clique is 2 while its arboricity is $\lceil \frac{n^2}{2n-1} \rceil$. Therefore, we have $O(\omega(G)) < O(\gamma(G))$. The running time of multiplying two $(n \times n)$ matrics, if carried out naively, is $O(n^3)$. The current $O(n^k)$ algorithm with the lowest known exponent k has an asynmptotic complexity of $O(n^{2.3728639})$ [28] . However, the constant coefficient hidden by the **Big O** notation is so large that this algorithm is only worthwhile for matrices that are too large to handle on present-day computers. Our algorithms achieves so far the best theoretical upperbound on time complexity with respect to $\mu(G)$ under the $O(|E|)$ space constraint.

TABLE 2
Complexity Upperbounds for Maximal Clique Enumeration

Algorithms	Runtime	Space
Tsukiyama et.al [37]	$O(nm\mu(G))$	$O(m)$
Chiba and Nishiseki [33]	$O(m\gamma(G)\mu(G)))$	$O(m)$
Makino and Uno [26]	$O(M(n)\mu(G))$	$O(n^2)$
Our Alg.	$O(m\omega(G)\mu(G))$	$O(m)$

3.2.2 Efficient Implementation

The efficiency of Algorithm 2 to a large extent is determined by the size of its partitioning traversal tree, which in turn depends on how to select a partitioning anchor. There are three options: (1) selecting the vertex v with the smallest degree in the current graph G; (2) randomly selecting a vertex; or (3) selecting the vertex with the largest degree. The first strategy would result in a relatively small graph G_v^+, and a large one G_v^-, in which the maximum clique of G is probably located. Generally, G_v^+ would be partitioned into cliques after only a few iterations because of its small size. At the same time, the size of the large graph G_v^- would be effectively reduced as recursive partitioning continues. The alternative of selecting a vertex with the largest degree as anchor would instead result in two large subgraphs G_v^+ and G_v^-, which may share a lot of common cliques. Therefore, it usually produces a much bigger traversal tree. We empirically evaluate the efficiency of these three anchor selection strategies in Section 5.1.3.

The vertices and their adjacency lists in the original graph G are stored in a hash table with vertex *id* as the hash key. Clique verification is achieved by checking vertex degrees. We store a `cand` set and the degrees of its vertices in the corresponding subgraph in a hash table with vertex *id* as the hash key. The intersection of two vertex sets are performed by a series of hash look-up operations. Computing the $\texttt{cand}^+$ and $\texttt{not}^+$ sets for the newly partitioned subgraph G_v^+, as shown in Lines 12 and 13 in Algorithm 2, corresponds to intersecting the adjacency set of the anchor vertex v with the current `cand` and `not` sets respectively. The degree of a vertex v_i in the $\texttt{cand}^+$ set of G_v^+ is computed by intersecting the adjacency set of v_i with the $\texttt{cand}^+$ set. For the vertices in the $\texttt{cand}^-$ set of G_v^-, only those connected to v needs to decrease their degrees by 1. Note that to select a partitioning anchor with the minimal degree in the `cand` hash table with vertex id as the hash key requires $O(|\texttt{cand}|)$ time. It has to sequentially scan all the vertices in the hash table. To enable more efficient anchor selection, we also maintain a degree hash table, in which the degrees of the vertices in `cand` are stored as a sorted list while each entry in the sorted list has a hashset consisting of all the vertices with the specified degree. The degree hash table of the G_v^- subgraph is inherited from that of its parent with corresponding updates while the degree hash table of the G_v^+ subgraph is constructed from scratch. With the degree hash table, selecting the vertex with the minimal degree in `cand` only involves picking up a vertex in the hashset of the first entry in the sorted list. It takes only $O(1)$ time.

3.3 Maximal K-plex Enumeration

K-plex detection is more costly than clique detection but allows for relaxed matching criteria, and hence is needed in some applications. In this section, we generalize the sequential algorithms for maximal clique enumeration to handle maximal k-plex enumeration.

3.3.1 A General Algorithm

Similar to the case of maximal clique enumeration, the algorithm for maximal k-plex enumeration selects a partitioning anchor v in a given graph G and partitions it into

Algorithm 3 enumerateKplex(anchor,cand,not)

```
1: if (G(anchorUcand) is a k-plex) then
2:   if (∄u ∈ not:G(anchorUcandU{u}) is a k-plex) then
3:     Output G(anchorUcand);
4:   end if
5: else
6:   Choose a vertex v in cand;
7:   anchor⁻=anchor;
8:   cand⁻=cand − {v};
9:   not⁻=not ∪ {v};
10:  if (∄u ∈ not⁻:u is connected to all the vertices in
     anchor⁻ and cand⁻) then
11:    enumerateKplex(anchor⁻,cand⁻,not⁻);
12:  end if
13:  anchor⁺=anchor ∪ {v};
14:  cand⁺={u|u ∈ cand ∧ G(anchor⁺ ∪{u}) is a k-plex};
15:  not⁺={u|u ∈ not ∧ G(anchor⁺ ∪ {u}) is a k-plex};
16:  if (∄u ∈ not⁺:u is connected to all the vertices in
     anchor⁺ and cand⁺) then
17:    enumerateKplex(anchor⁺,cand⁺,not⁺);
18:  end if
19: end if
```

two subgraphs G_v^+ and G_v^-. The G_v^+ branch corresponds to the case that maximal k-plexes contain the vertex v while the G_v^- branch corresponds to the other case that maximal k-plexes do not contain v. It thereafter proceeds to partition both G_v^+ and G_v^- recursively until they become k-plexes.

The recursive function for maximal k-plex enumeration is shown in Algorithm 3. The anchor, cand and not vertex sets serve the purposes similar to those explained in the algorithm for maximal clique enumeration. Lines 7-9 compute the $\texttt{anchor}^-$, $\texttt{cand}^-$ and $\texttt{not}^-$ sets for G_v^- while Lines 13-15 compute the three vertex sets for G_v^+. Lines 10 and 16 specify the pruning conditions. If there exists a vertex u in $\texttt{not}^+$ such that u is connected to all the vertices in $\texttt{anchor}^+$ and $\texttt{cand}^+$, then any induced k-plex of G_v^+ is not maximal. As a result, the G_v^+ branch can be pruned. The G_v^- branch can be pruned in a similar way. However, this condition does not guarantee to filter out all the non-maximal k-plexes. Therefore, Line 2 checks whether adding any vertex in not to a candidate k-plex would result in a bigger k-plex. If yes , the candidate k-plex is not maximal.

Executing a depth-first search in a traversal tree, Algorithm 3 has to simultaneously maintain the data structures at each node along the path from the root to a leaf. Unlike the case of maximal clique enumeration, Algorithm 3 can not guarantee that the vertices in the $\texttt{anchor}^+$, $\texttt{cand}^+$ and $\texttt{not}^+$ sets of G_v^+ are connected to the partitioning anchor v. As a result, Algorithm 3 requires $O(|V|^2)$ space to store intermediate results. On time complexity, each line of statement in Algorithm 3 can be performed in $O(|E|)$ time. The size of the traversal tree generated by the recursive function is bound by $O(\omega_k(G)\mu_k(G))$, in which $\omega_k(G)$ and $\mu_k(G)$ represent the number of vertices in the maximum k-plex of G and the number of maximal k-plexes in G respectively. Therefore, we have the following theorem:

Theorem 3. *Given a connected graph G=(V, E), Algorithm 3 has the space complexity of $O(|V|^2)$ and the time complexity of $O(|E|\omega_k(G)\mu_k(G))$.*

To achieve the optimal space complexity, we have also proposed a variant of Algorithm 3 which uses a *stack* to track the traversal process on the partitioning tree. It first traverses along the G_v^+ branch, but does not remember the G_v^- subgraphs. However, it is less efficient than Algorithm 3 in practical implementation. More details are omitted here because of space limit. Please refer to our technical report [35] for more information.

3.3.2 A Space Optimal Algorithm

Algorithm 4 A Space Optimal Algorithm for Maximal K-plex Enumeration

```
Input: A graph G with vertex set V and edge set E;
1: Stack S ← ∅;
2: anchor← ∅;
3: cand← V;
4: not← ∅;
5: while (G(anchorUcand) is not a k-plex) or (S ≠ ∅) do
6:   if G(anchorUcand) can not be pruned then
7:     if G(anchorUcand) is a k-plex then
8:       if (∄u ∈ not:G(anchorUcandU{u}) is a k-plex)
         then
9:         Output G(anchorUcand);
10:      end if
11:    else
12:      Choose a vertex v in cand;
13:      S.push(v⁺);
14:      anchor=anchorU{v};
15:      cand={u|u ∈ (cand-{v})∧ G(anchor ∪ {u}) is a
         k-plex};
16:    end if
17:  end if
18:  if (G(anchorUcand) is pruned) or (G(anchorUcand)
     is a k-plex) then
19:    while S.top() == v⁻ do
20:      S.pop();
21:      not=not-{v};
22:    end while
23:    if S.top() == v⁺ then
24:      S.pop();
25:      S.push(v⁻);
26:      anchor=anchor-{v};
27:      not=notU{v};
28:      cand={u|u ∈ (V − not) ∧ G(anchor ∪ {u}) is a
         k-plex};
29:    end if
30:  end if
31: end while
```

To achieve the optimal space complexity, we present a variant of Algorithm 3 (as shown in Algorithm 4), which has the same time complexity $O(|E|\omega(G)\mu(G))$ but requires only $O(|E|)$ space. Besides three vertex sets, anchor, cand and not, to record the current subgraph to be partitioned, it also uses a *stack* data structure S to track the traversal progress of the partitioning search tree. The stack S maintains a series of vertices, each of which is marked as *inclusive* or *exclusive*. The entry of an *inclusive* vertex v in S, denoted by v^+, corresponds to the partitioning branch G_v^+, in which the searched subgraphs

should contain v. In contrast, the entry of an *exclusive* vertex v in S, denoted by v^-, corresponds to the other partitioning branch G_v^-, in which the searched subgraphs do not contain v.

Similar to Algorithm 3, Algorithm 4 executes a depth-first traversal. It first traverses along the G_v^+ branch (Lines 12-15). The anchor v is added to `anchor`. The *inclusive* vertex v, v^+, is pushed into S. `cand` is updated in the same way as in Algorithm 3. Whenever it reaches a leaf node, it backtracks to the last *inclusive* branch (Lines 19-22), which corresponds to the latest *inclusive* vertex pushed into S. Then it continues to traverse along the G_v^- branch (Lines 23-29). The *exclusive* vertex v, v^-, is pushed into S. The vertex v is also removed from `anchor`. Because the algorithm maintains only one `cand` during the partitioning process, the `cand` set of G_v^- has to be constructed from the current `anchor` and `not` sets (Line 28). It is worthy to point out that when traversing along the G_v^+ branch, the algorithm does not update the `not` set. This modification is to facilitate constructing the `not` set of G_v^- branch.

Instead, branch pruning (in Line 6) requires that a vertex in `not` be connected to all the vertices in `anchor` and `cand` . This modification, necessary for constructing the `not` set of G_v^- branch, achieves the same pruning effect as Algorithm 2.

Algorithm 4 only needs to maintain three vertex sets, `anchor`, `cand` and `not`, a stack recording traversal progress, as well as the adjacency lists of vertices. Also note that each line of statement in Algorithm 4 can be performed in $O(|E|)$ time. Therefore, we have the following theorem:

Theorem 4. *Given a connected graph G=(V, E), Algorithm 4 has a space complexity of $O(|E|)$ and a time complexity of $O(|E|\omega(G)\mu(G))$.*

3.3.3 Efficient Implementation

For efficient implementation of Algorithm 3, the vertex with the minimal degree in `cand` is chosen as the partitioning anchor. For each vertex u in `anchor`, `cand` and `not`, we maintain two degrees $deg_a(u)$ and $deg_c(u)$, in which $deg_a(u)$ is the number of vertices in `anchor` connected to u and $deg_c(u)$ is the number of vertices in `cand` connected to u. The condition specified in Line 1 of Algorithm 3 can be verified by adding up $deg_a(u)$ and $deg_c(u)$ for each vertex in `anchor` and `cand`. For the condition specified in Line 2, given a vertex u in `not`, G(`anchor`$\cup$`cand`$\cup\{u\}$) is a k-plex if and only if: (1) the combined degree of u ($deg_a(u)+deg_c(u)$) should be at least (|`anchor`|+|`cand`|-k+1); (2) each vertex w in `anchor` and `cand`, whose combined degree is equal to (|`anchor`|+|`cand`|-k), should be connected u. For the condition specified in Line 10, a given vertex u in `not` is connected to all the vertices in `anchor` and `cand` if and only its combined degree is equal to (|`anchor`|+|`cand`|). The condition specified at Line 16 can be verified in the same way.

Now we provide the details on how $deg_a(u)$ and $deg_c(u)$ can be efficiently maintained in the partitioning process. We first consider the G_v^- subgraph. The vertex degrees of G_v^- can be inherited from its parent graph with only minor updates. For a vertex u in `anchor`$^-$ or `cand`$^-$, its $deg_a(u)$ value remains unchanged. Its $deg_c(u)$ value should be decreased by 1 if it is connected to the anchor v. Similarly, for a vertex u in `not`$^-$ but not the vertex v, its $deg_a(u)$ value remains unchanged and its $deg_c(u)$ value should be decreased by 1 if it is connected to the anchor v. For the vertex v in `not`$^-$, both of its $deg_a(u)$ and $deg_c(u)$ values remain unchanged. The required updates on the G_v^+ subgraph are more involved. For a vertex u in the original `anchor`, its $deg_a(u)$ value should be increased by 1 if it is connected to the anchor v; otherwise it remains unchanged. The $deg_a(u)$ value of the new anchor v remains unchanged. The $deg_c(u)$ value of each vertex in `anchor`$^+$ is computed by intersecting its adjacency vertex set and the `cand`$^+$ set. The case for a vertex w in `cand`$^+$ or `not`$^+$ is similar. Its $deg_a(u)$ value should be increased by 1 if it is connected to the anchor v; otherwise it remains unchanged. Its $deg_c(u)$ value is computed by intersecting its adjacency vertex set and the `cand`$^+$ set.

4 Parallel Algorithms

In this section, we present the parallel algorithms for maximal clique and k-plex enumeration based on recursive graph partitioning and describe their MapReduce implementations.

4.1 Maximal Clique Enumeration

4.1.1 Parallel Algorithm

The parallel algorithm consists of two steps. In the first step, for every vertex v in the graph G, it retrieves v's neighboring vertices and the connectivity information between them. In the second step, it uses recursive graph partitioning to compute the maximal cliques of every vertex. In the parallel setting, both subgraph retrieval and clique computation on individual vertices are distributed across multiple computing nodes.

Consider the induced subgraph G_v of G consisting of v and its neighbouring vertices. Obviously, all the maximal cliques containing v are induced subgraphs of G_v. Maximal clique enumeration on the vertex v can performed by the sequential algorithm presented in Algorithm 2. It recursively chooses a partitioning anchor in a graph and partitions the graph into two subgraphs until all the resulting subgraphs become clique. Note that in the parallel setting, some maximal cliques may be repeatedly computed because they are contained by more than one induced subgraphs G_v in G. To eliminate redundant cliques, we assign each vertex a unique serial number. On a vertex v and its induced subgraph G_v, we delete the neighboring vertices of v with smaller serial numbers than v's from its `cand` set and insert them into its `not` set. With this initialization, each clique output by the parallel algorithm is maximal and unique.

Unfortunately, computational cost on individual vertices may be unbalanced. Maximal clique computation on some vertices may be more expensive than on others because their partitioning traversal trees have larger sizes. As a result,

the computation on an individual vertex may become a parallel performance bottleneck if it is too time-consuming. The good news is that our proposed algorithm enables an easy and effective load balancing mechanism. Since the resulting subgraphs G_u^+ and G_u^- after partitioning G_v are independent, maximal clique computation on the vertex v (the graph G_v) can be easily parallelized. In practice, the recursive function usually takes only a few iterations (no more than 5-6 iterations in our experiments in Section 5.2) to transform a big G_v into many sufficiently small subgraphs. With sufficiently small tasks, effective load balancing can be achieved by sending some tasks on the computing nodes with heavy workload to others with lighter one.

In a parallel setting, workload can be balanced across computing nodes by repeatedly invoking the *compute* and *shuffle* cycle. In the *compute* phase, every computing node performs partitioning operations on the graphs it has received; in the *shuffle* phase, all the non-clique graphs on the nodes are reshuffled so that every node receives roughly the same number of unfinished graphs. The workload limit of each *compute* phase can be quantified by the number of partitioning operations executed or CPU time consumed.

[Summary.] The parallel algorithm for maximal clique detection with load balancing consists of the following two steps:

1) **Subgraph retrieval:** retrieve the induced graph G_v consisting of v and its neighboring vertices for every vertex v in the graph G in parallel;
2) **Iterative clique computation:**
   - ***compute* phase:** for each computing node, compute maximal cliques of the graphs assigned to it by recursive graph partitioning;
   - ***shuffle* phase:** evenly reshuffle all the unfinished graphs across the nodes;

4.1.2 MapReduce Implementation

It is obvious that subgraph retrieval can be built on 2-hop retrieval as described in Subsection 2.4. Unfortunately, 2-hop retrieval has the shortcoming that it may generate huge intermediate data between phases, which can result in poor performance. Based on the observation that non-trivial clique consists of triangles, we propose to use the technique of triangle enumeration proposed in [21] to implement the process of subgraph retrieval. Its MapReduce implementation consists of the following two cycles:

1) In the first cycle, it begins with the edge pairs (v_i, v_j), which means that there is an edge between the vertices v_i and v_j, and emits open triads with the format of $(\{v_i, v_j\}, v_k)$, which means that both v_i and v_j are directly connected to v_k;
2) In the second cycle, it maps both the edge pairs (v_i, v_j) and the open triads $(\{v_i, v_j\}, v_k)$, and aggregates them by the key (v_i, v_j). It finally emits the open triads with close edges, which correspond to triangles.

As in [21], we represent each triangle with a vertex as its key and the other two vertices as its value. For instance, the triangle consisting of the vertices $\{1, 2, 3\}$ is represented by the key-value pair of $(1, \{2, 3\})$. A set of key-value pairs with the same key corresponds to a triangle unit. In the rest of this subsection, we describe the implementation details beginning with the retrieved triangles.

Algorithm 5 Maximal Clique Computation in *Reducer*

Input: A queue of unfinished subgraphs Q_c;

```
1:  while (Q_c is not empty) and (workload limit is not reached) do
2:    Dequeue a subgraph G_u from Q_c;
3:    while (G_u can not be pruned) and (G_u is not a clique) do
4:      Choose the vertex w with the minimal degree in G_u as the anchor;
5:      Partition G_u into G_w^+ and G_w^-;
6:      if |cand(G_w^+)| ≤ k then
7:        Recursively partition G_w^+ using Algorithm 2 to the end;
8:      else
9:        if G_w^+ can not be pruned then
10:         if G_w^+ is a clique then
11:           Output G_w^+;
12:         else
13:           Enqueue G_w^+ into Q_c;
14:         end if
15:       end if
16:     end if
17:     G_u=G_w^-;
18:   end while
19:   if G_u can not be pruned then
20:     Output the clique G_u;
21:   end if
22: end while
```

The *Map* phase of the first cycle reads the triangles into memory and maps them to corresponding reducers according to their keys $\{v_1, v_2, \ldots, v_n\}$, in which v_i is a vertex in the graph G. The *Reduce* phase first aggregates the triangles with the same key as a triangle unit. It then sequentially computes the maximal cliques of v_i on the induced subgraph G_{v_i}.

The vertices and their adjacency lists in the G_{v_i} subgraph are stored in a hashtable. Suppose that these induced G_{v_i} subgraphs are maintained by a queue Q_c. The process of maximal clique computation in the *Reduce* phase is sketched in Algorithm 5. It dequeues a G_u subgraph from Q_c and iteratively partitions it in a depth-first manner. If the resulting G_w^+ has a small size, which means that its maximal clique computation can be finished in short time, it is recursively partitioned to the end (Lines 6-7). Otherwise, it is temporarily enqueued into Q_c if it is not a clique (Line 13). It then continues to partition G_w^- in the same manner as G_u (Line 17). Each subgraph in the queue is represented by its `anchor`, `cand` and `not` sets. A *Reducer* iteratively dequeues a subgraph from Q_c and processes it until Q_c becomes empty or it reaches the predefined workload limit. Finally, it writes all the left subgraphs in Q_c (if any) to disk. A new MapReduce cycle is then triggered to process the unfinished subgraphs. Note that partitioning an induced subgraph G_u of G_{v_i} requires its `anchor`, `cand` and `not` sets and the hashtable of G_{v_i}. As a result, between MapReduce cycles, besides the `anchor`,

`cand` and `not` sets of each subgraph in Q_c, the hashtable of its corresponding G_{v_i} should also been written to disk and transferred to the next cycle. However, at most one copy of the hashtable of G_{v_i} is needed for each *reducer*.

In the second MapReduce cycle, *Mappers* reads the unfinished subgraphs from the first cycle into memory and randomly map them to reducers such that each reducer receives roughly the same number of subgraphs. Each *Reducer* then processes its assigned subgraphs as described in Algorithm 5. The MapReduce cycle is iteratively invoked until no unfinished subgraph is left.

4.2 Maximal K-plex Enumeration

The parallel algorithm for maximal k-plex enumeration similarly consists of two steps, subgraph retrieval and iterative k-plex computation.

In the first step of subgraph retrieval, each vertex v_i in G retrieves its 2-hop neighbors in parallel. In the second step of iterative k-plex computation, the algorithm computes the maximal k-plexes of each vertex by recursive graph partitioning in parallel. Similar to the case of maximal clique enumeration, the algorithm iteratively partitions a subgraph G_{v_i} in a depth-first manner and transforms G_{v_i} into many sufficiently small subgraphs in the *compute* phase. In the *shuffle* phase, the small subgraphs are evenly redistributed to achieve load balancing among computing nodes. The cycle of *shuffle* and *compute* phases is iteratively invoked until the k-plex computations on all the subgraphs retrieved in the first step are finished.

Its MapReduce implementation first uses a MapReduce cycle to retrieve each vertex's 2-hop neighbors in parallel. Then, it initiates a new MapReduce cycle to perform maximal k-plex computation on each vertex in parallel. Similar to the MapReduce implementation of maximal clique enumeration, it expoits a queue to store the unfinished subgraphs. A resulting G_w^+ would be recursively partitioned to the end if its size does not exceed a predefined threshold; otherwise, it is temporarily enqueued into the queue for later processing. If each computing node reaches its workload limit and there exists any unfinished subgraph, a new MapReduce cycle will be initiated to redistribute all the unfinished subgraphs across computing nodes and execute k-plex computation on them. The iteration of MapReduce cycle ends if and only if there is no unfinished subgraph left. More details on its MapReduce implementation are omitted here because it is very similar to the MapReduce implementation of maximal clique enumeration described in Subsection 4.1.1.

5 Experimental Evaluation

We empirically evaluate the performance of our new approach by a comparative study. For maximal clique enumeration, we compare our approach with a variant of the BK algorithm proposed in [20], which employs the same pruning methods as BK but has been reported to be faster than BK. In the parallel setting, besides the typical BK approach that confines the computation on an individual vertex to a computing node, we also compare our approach with the parallel BK approach enhanced with the dynamic load balancing technique proposed in [30]. The proposed technique was implemented by MPI in [30]. We have instead implemented a MapReduce version. Each reducer is set to have a predefined workload limit. After every reducer reaches its workload limit, the unfinished subgraphs are evenly redistributed across computing nodes. As Algorithm 5, a BK search subtree will be processed to the end if its size is no larger than a predefined threshold. Otherwise, it will be temporarily buffered for later processing.

Our evaluation for maximal k-plex enumeration is conducted in a similar way. In the centralized setting, we compare our sequential algorithm with a variant of the BK search algorithm for maximal k-plex enumeration proposed in [40] because of its effective pruning tricks. In the parallel setting, we compare our approach with the typical BK approach and the BK approach enhanced with the dynamic load balancing technique. In this section, we refer to the new algorithms based on graph partitioning as GP algorithms and the variants of the classical BK algorithm as BK algorithms.

Our experiments are conducted on both real and synthetic graph datasets. The evaluation on real datasets can show the efficiency of the proposed algorithms in real applications while the evaluation on synthetic datasets can clearly demonstrate their sensitivity to varying graph characteristics. The real graph data are selected from [5], which are in various domains including email communication networks, social networks, web graphs and Wiki communication networks. The synthetic datasets are generated by the SSCA#2 generator and the power-law generator R-MAT from the popular graph generator suite GTgraph [16]. A SSCA#2 graph is directed, and made up of random-sized cliques, with a hierarchical inter-clique distribution of edges based on a distance metric. We vary the values of the *TotVertices* and *MaxCliqueSize* parameters, which specify the number of vertices and the size of the maximum clique respectively. The R-MAT generator applies the Recursive Matrix (R-MAT) graph model [12] to produce the graphs with power-law degree distributions and small-world characteristics, which are common in many real life graphs. We vary two parameter values, the number of vertices and the number of edges. Note that both the SSCA#2 and R-MAT graphs are directed. We transform them into undirected graphs by discarding edge directions. The details of the real and synthetic graph datasets are summarized in Table 3.

The experiments are executed on a ten-machine cluster. Each machine runs the Ubuntu Linux (version 10.04), has a memory size of 16G, disk storage of 160G and four Intel Xeon E5502 CPUs with the frequency of 1.87GHz. The evaluation of sequential algorithms is conducted on a JVM (Java Virtual Machine) running on a machine. In the parallel evaluation, one machine functions as master and up to 9 machines as slaves. Each slave machine runs 4 JVMs (Java Virtual Machine) and each JVM heap is allocated a maximum 3G RAM. We have implemented the parallel approaches on the Hadoop framework (version

TABLE 3
Details of the Real and Synthetic Graph Datasets

Dataset	Data Description	Number of Vertexes	Number of Edges
D_1	Email Network from a EU Research Institution	265,214	364,481
D_2	Web graph from Google	875,713	4,322,051
D_3	Web graph of Berkeley and Stanford	685,230	7,600,595
D_4	Wikipedia communication network	1,928,669	3,494,674
D_5	Pokec online social network	1,632,803	30,622,564
D_6	Social circles from Twitter	11,316,811	85,331,846
R-MAT	Synthetic graphs with power-law degree distributions and small-world characteristics	Two parameters used: the number of vertices and the ratio of edges to vertices	
SSCA#2	Synthetic graphs with a hierarchical inter-clique distribution of edges based on a distance metric	Two parameters used: the number of vertices and the size of maximum clique	

0.20.2). Each experiment was run three times and its running time averaged. In case that processing the entire graphs listed in Table 3 is beyond the capability of a single machine or even the ten-machine cluster (e.g., maximal clique enumeration over the Twitter dataset and maximal k-plex enumeration), we randomly select some vertices and compute their maximal cliques and k-plexes over the entire graphs. However, the observations coming from the experiments can be applied to the entire graphs. All the experiments generate non-trivial maximal cliques and k-plexes with sizes larger than 3.

5.1 Evaluation of Sequential Algorithms

In this subsection, we evaluate the performance of the sequential algorithms for maximal clique and k-plex enumeration on both real and synthetic graphs. Their performance are measured by two metrics, the size of search tree and the runtime. While the runtime of an algorithm depends on its implementation details, search tree size accurately measures search space and is independent of algorithmic implementations. By default, the GP algorithm always selects the vertex with the mimimal degree in the current graph as the partition anchor. We evaluate its sensitivity to different selection strategies of partitioning anchor in Subsection 5.1.3, which clearly demonstrates that the strategy of selecting the vertex with the minimal degree produces the smallest search trees and thus achieves the best performance.

5.1.1 Maximal Clique Enumeration

The evaluation results on the real graphs (D_1-D_5) are presented in Figure 3 (a) and (b). Note that running the dataset D_6 is beyond the capability of a single computing node. Therefore, they will be later used for parallel evaluation. On search tree size, the GP algorithm consistently outperforms the BK algorithm by big margins. Even though GP achieves considerably smaller search space than BK, it has to check whether the partitioned subgraphs are cliques or not. GP thus takes more time per traversal than BK. As a result, on runtime, GP still performs better than BK but by smaller margins.

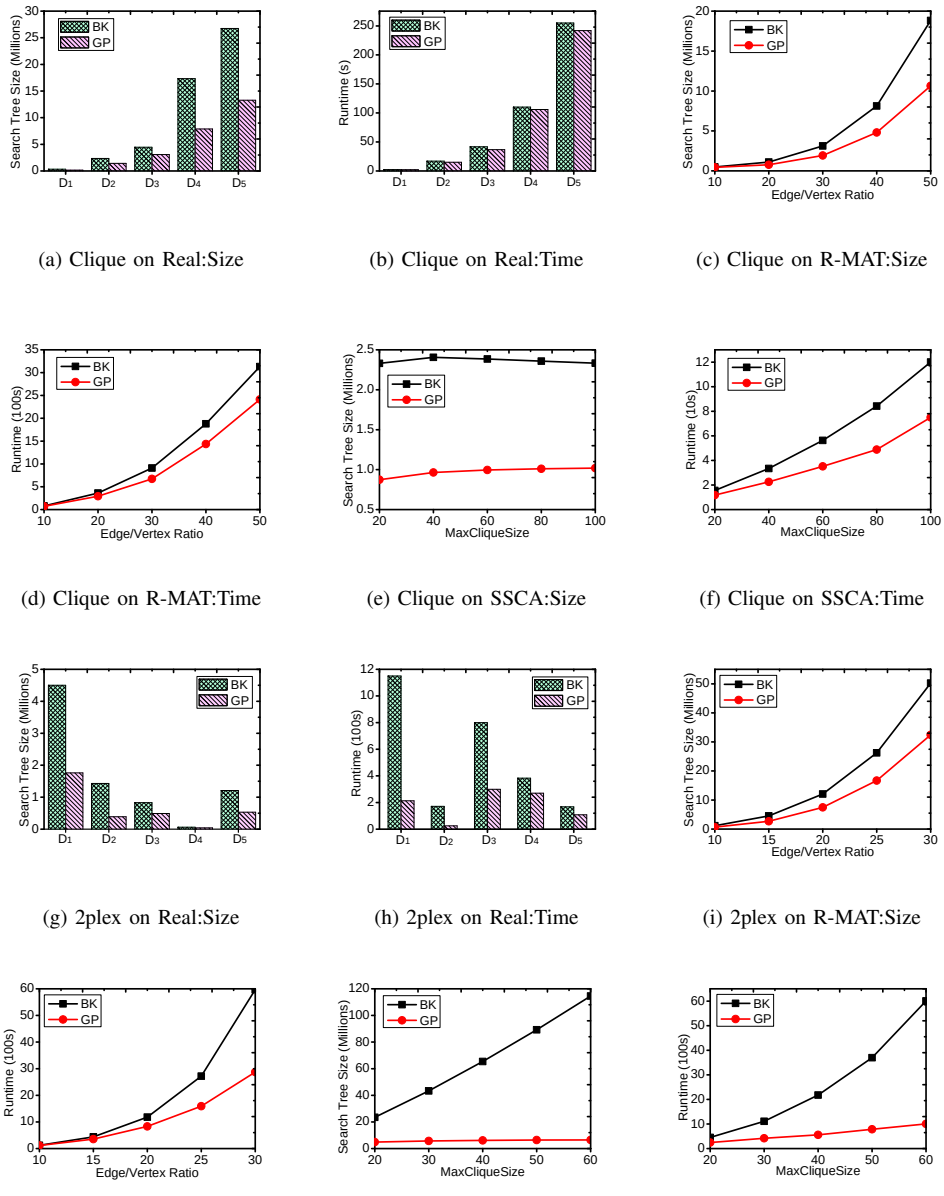
We also evaluate their performance on synthetic graphs to investigate how their performance vary with different graph characteristics and densities. For R-MAT graphs, the number of vertices is set to be 1 million and the edge-to-vertex ratio varies from 10 to 50. The results are shown in Figure 3 (c) and (d). Similar to what were observed on real graphs, GP outperforms BK on both search space size and runtime and its performance advantage on search space size is more significant. It is interesting to observe that the performance gap between BK and GP steadily increases with graph density. The evaluation results on the SSCA graphs are also shown in Figure 3 (e) and (f). We set the number of vertices to be 2^{20} and vary the size of the maximum clique from 20 to 100. It can be observed that compared with the results on real and R-MAT graphs, GP outperforms BK by the largest margins on the SSCA graphs. The SSCA graphs have larger-sized maximal cliques. As a result, GP is able to achieve bigger save on search space size. Similar to what were observed on R-MAT graphs, the performance advantage of GP steadily increases with the sizes of maximal cliques.

Our experiments show that the sequential GP algorithm has an overall better performance than the BK algorithm and its performance advantage increases with graph density and the sizes of maximal cliques.

5.1.2 Maximal K-plex Enumeration

We evaluate the performance of the sequential GP algorithm for maximal k-plex enumeration with low values of k (k=2). Maximal 2-plex enumeration is more costly than maximal clique enumeration. Enumerating all the maximal 2-plexes in the real graphs listed in Table 3 is beyond the capability of a computing node in our experimental setting. Therefore, we randomly select 10% of the vertices in D_1, 0.1% of vertices in D_2-D_5, and compute their maximal k-plexes on the entire graphs. Compared with the synthetic graphs used for maximal clique enumeration, the test SSCA and R-MAT graphs also have smaller sizes. The R-MAT graphs have 10,000 vertices and edge-to-vertex ratios varying from 10 to 30. The SSCA graphs have 2^{20} vertices and have the maximum clique sizes varying from 20 to 60.

The evaluation results on the real, synthetic R-MAT and SSCA graphs are presented in Figure 3 (g)-(l). They are similar to what were reported on maximal clique enumeration except that GP ourperforms BK by even larger margins. The maximal 2-plexes of a graph have larger sizes than its maximal cliques. BK traversal on a maximal 2-plex search



(a) Clique on Real:Size (b) Clique on Real:Time (c) Clique on R-MAT:Size

(d) Clique on R-MAT:Time (e) Clique on SSCA:Size (f) Clique on SSCA:Time

(g) 2plex on Real:Size (h) 2plex on Real:Time (i) 2plex on R-MAT:Size

(j) 2plex on R-MAT:Time (k) 2plex on SSCA:Size (l) 2plex on SSCA:Time

Fig. 3. **Evaluation of sequential algorithms: (1) on maximal clique enumeration, GP performs considerably better than BK; (2) on maximal 2-plex enumeration, GP outperforms BK by even more significant margins.**

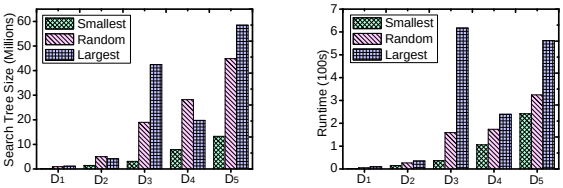
tree is also less efficient than on a maximal clique search tree. As a result, the performance advantage of GP over BK becomes more considerable. On the R-MAT and SSCA graphs, we also observe that the performance gap between BK and GP increases with graph density and the sizes of maximal cliques.

5.1.3 Sensitivity of Anchor Selection

In this subsection, we empirically study the impact of different partitioning anchor selection strategies on the performance of the GP approach. We runs the GP algorithm on the real graphs (D_1-D_5) listed in Table 3. We evaluate their performance with three different anchor selection strategies, as described in Section 3.2.2: (1) choosing the vertex with the smallest degree in the current graph; (2) randomly choosing a vertex; (3) choosing the vertex with the largest degree.

The evaluation results for maximal clique enumeration in terms of traversal tree size and runtime are shown in Figure 4. We observe that the first strategy consistently achieves the best performance. The comparative results on traversal tree size reveal that the second and third strategies produce many more branches not containing any maximal clique; redundant computation leads to poor performance. Between the second and third strategies, the strategy of randomly selecting the partitioning anchor has an overall better performance.

The evaluation results for maximal 2-plex are similar,

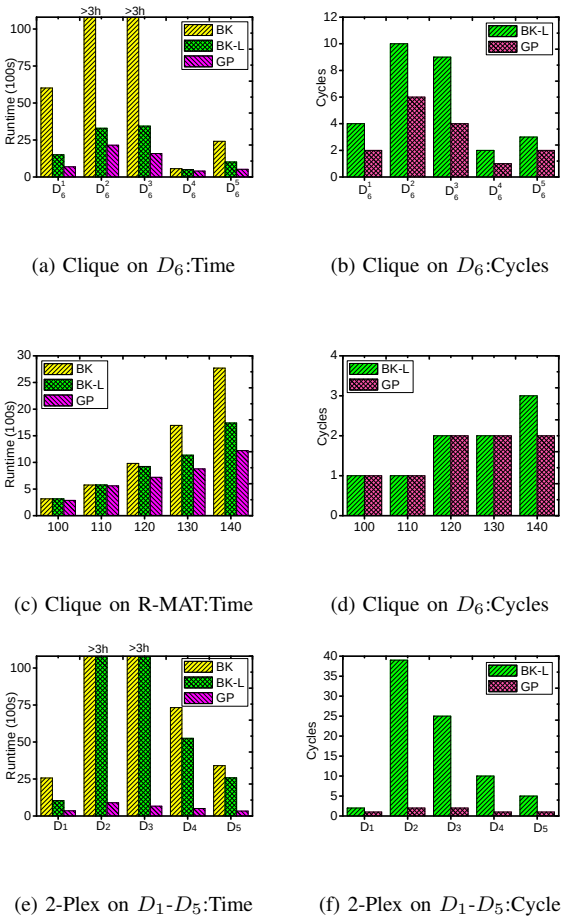


(a) Search Space (b) Runtime

Fig. 4. **Sensitivity evaluation of anchor selection strategies: the strategy of selecting the vertex with the smallest degree achieves the best performance.**

thus ommited here. Our experiments demonstrate that the strategy of choosing the vertex with the smallest degree results in the smallest search space, and thus achieves the best performance.

5.2 Evaluation of Parallel Algorithms



(a) Clique on D_6:Time (b) Clique on D_6:Cycles

(c) Clique on R-MAT:Time (d) Clique on D_6:Cycles

(e) 2-Plex on D_1-D_5:Time (f) 2-Plex on D_1-D_5:Cycle

Fig. 5. **Parallel evaluation: (1) the BK approach w/o dynamic load balancing may perform very poorly; (2) the GP approach performs considerably better than the BK-L approach; (3) the performance advantage of GP over BK and BK-L is more significant on maximal 2-plex enumeration than on maximal clique enumeration.**

In this subsection, we compare the parallel approach based on GP against those based BK search on performance. We refer to the typical BK approach as BK and the BK approach with dynamic load balancing as BK-L in this subsection. Since all the parallel approaches use the same method of subgraph retrieval, we exclude its cost from performance evaluation in our comparative study. We use triangle enumeration and 2-hop retrieval to implement subgraph retrieval for maximal clique enumeration and maximal 2-plex enumeration respectively.

5.2.1 Maximal Clique Enumeration

The GP approach always selects the vertex with the smallest degree in the current graph as the partitioning anchor. We specify that a graph with no more than 50 vertices is recursively partitioned to the end within a phase, or $k = 50$ in Algorithm 5. The maximal execution time per *Reduce* phase is set to be 300 seconds. The parameter m and the workload limit of execution time per *Reduce* phase are similarly set for the BK-L approach.

Since the real graphs (D_1-D_5) in Table 3 can be efficiently processed on a single worker, we do not evaluate the performance of parallel approaches on them. Processing the entire graph of the Twitter dataset (D_6) takes too long on our ten-machine cluster. Therefore, we generate 5 random test tasks, denoted by D_6^1,. . .,D_6^5, by randomly choosing 1% of the vertices in the graph and enumerating their maximal cliques over the entire graph. On SSCA graphs, even the BK approach manages to evenly distribute the workload across workers. As a result, the parallel evaluation results on SSCA graphs are similar to what were observed in sequential evaluation. We do not present their results here either.

The comparative results on the Twitter and R-MAT graphs are presented in Figure 5 (a)-(d). On the Twitter graph, Figure 5 (a) shows that the typical BK approach performs very poorly in some cases (e.g., D_6^2 and D_6^3). It can not finish computation within the 3-hour runtime limit. In both cases, there exist some vertices heavily connected to other vertices. The maximal clique computation on these vertices are extremely expensive and thus become parallel performance bottleneck if without dynamic load balancing. The experiments show that both the BK-L and GP approaches can effectively break the performance bottleneck by redistributing the computation on an individual vertex across multiple computing nodes. For instance on D_6^2, both of them reduce the runtime to less than 1 hour. However, it can be observed that the GP approach achieves overall better parallel performance than the BK-L approach. Compared with BK-L, GP generates much smaller traversal trees and is able to partition big graphs into sufficiently small subgraphs with less iterations. With more effective load balancing mechanism, GP achieves better performance than BK-L in terms of number of required MapReduce cycles, as shown in Figure 5 (b). The evaluation results on the R-MAT graphs, as shown in Figure 5 (c) and (d), are similar except that the performance difference among the three approaches appears less considerable. Compared

with the Twitter graph, a R-MAT graph has more balanced edge distribution among its vertices. As a result, the effect of dynamic load balancing becomes less dramatic. On the denser graphs (e.g., when the edge/vertex ratio is equal to 140), GP still outperforms BK-L by considerable margins (more than 30%) in terms of runtime. It is also worthy to point out that similar to what was observed in sequential evaluation, the performance advantage of GP over BK-L increases with the density of R-MAT graphs.

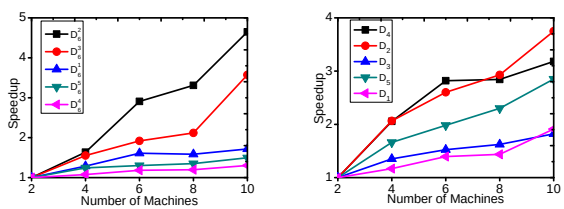
5.2.2 Maximal K-plex Enumeration

This subsection comparatively evaluates the performance of the BK, BK-L and GP approaches for parallel maximal 2-plex enumeration. Maximal 2-plex enumeration is computationally more expensive than maximal clique enumeration. As a result, processing the entire graphs of some real datasets listed in Table 3 takes too long on our ten-machine cluster. On D_1 and D_2, we process the entire graphs. On D_3, D_4 and D_5, we instead randomly select 1% of the vertices in the graphs and compute their maximal 2-plexes over the entire graphs. Our programs specify that any graph with no more than 50 vertices is recursively processed to the end without being redistributed. The maximal execution time per *Reduce* phase is set to be 500 seconds.

The comparative results on real graphs are presented in Figure 5 (e) and (f). As in maximal clique enumeration, we limit the running time to 3 hours. It is observed that without load balancing, the performance of the BK approach is volatile; it can perform very poorly in some cases (e.g. D_2 and D_3). Between BK-L and GP, GP ourperforms BK-L by considerable margins. Again, GP traverses considerably smaller search space and is able to partition big graphs into sufficiently small subgraphs with less iterations. Compared to what were observed in parallel clique computation, the performance advantage of GP appears more considerable.

5.2.3 Parallelizability Evaluation

To evaluate the parallelizability of the GP approach, we run it on the machine clusters of different sizes and track its performance variation. We set up 5 cluster configurations which have 2,4,6,8 and 10 slave machines respectively. The performance is measured in terms of runtime.



(a) Clique on D_6 (b) 2-Plex on D_1-D_5

Fig. 6. **Parallizability evaluation for maximal clique and 2-plex enumeration: the GP approach achieves desired parallelizability**

6 Related Work

Even though efficient clique and k-plex detection have been studied extensively in the literature [20], [21], [18], [34], [39], [13], [36], [8], [7], most work focused on centralized search algorithms. They usually rely on global state and cannot be easily implemented in the parallel setting.

Existing parallel approaches [41], [29], [42], [40], [30], [14] for clique and k-plex detection were based on either MPI or MapReduce. They distribute the vertices across computing nodes and compute every vertex's maximal cliques and k-plexes in parallel. Most of them used the BK algorithm or its variants to compute maximal maximal cliques and k-plexes containing an individual vertex in a graph. The dynamic load balancing technique of redistributing subtree searches was first proposed in [30].

In this paper, we choose the popular open-source MapReduce platform, Hadoop [2], to implement our parallel solutions. Obviously, our approach can also be easily implemented on other parallel platforms modeled after the MapReduce architecture, e.g., Pig [15] and Dryad [32]. It can be similarly implemented on BSP platforms, e.g., Pregel [24], Giraph [1] and Hama [3].

Since the load balancing process of our approach has to redistribute intermediate graphs across multiple computing nodes, implementing our approach on the platforms with efficient main-memory processing and caching mechanisms, such as HaLoop[9] and Spark [6], would further reduce I/O cost and thus improve efficiency.

7 Conclusion

In this paper, we have proposed a novel approach based on recursive graph partitioning to address the problem of maximal clique and k-plex enumeration over graph data. It achieves so far the best theoretical upperbound on the runtime as a polynomial with respect to the number of maximal cliques and k-plexes under polynomial space constraints. It is also inherently more parallelizable than previous approaches. Its better parallelizability enables more effective load balancing and ultimately results in more efficient parallel performance. Our extensive experiments have validated its efficacy.

On future work, it is interesting to investigate whether our approach based on recursive graph partitioning can be applied to address parallel processing of other NP-Complete graph problems (e.g., minimum Steiner tree and graph pattern matching).

References

[1] Giraph project. In *http://giraph.apache.org/*.
[2] Hadoop: An open-source implementation of mapreduce. In *http://hadoop.apache.org/*.
[3] Hama project. In *http://hama.apache.org/*.
[4] Mapreduce. In *http://en.wikipedia.org/wiki/MapReduce*.
[5] Real graph datasets. In *http://snap.stanford.edu/data/*.
[6] Spark project. In *http://spark.apache.org/*.
[7] B.McClosky and I.V.Hicks. The co-2-plex polytope and integral systems. *IAM Journal on Discrete Mathematics*, (3):1135–1148, 2009.

[8] B.McClosky and I.V.Hicks. Combinatorial algorithms for the maximum k-plex problem. *Journal of Combinatorial Optimization*, (23):29–49, 2012.

[9] Y. Bu, B. Howe, and et al. Haloop: Iterative data processing on large clusters. In *VLDB*, 2010.

[10] B.Won.On, E.Elmacioglu, and et al. Improving grouped-entity resolution using quasi-cliques. In *Data Mining, 2006. ICDM '06. Sixth International Conference on*, pages 1008 –1015, dec. 2006.

[11] C.Bron and J.Kerbosch. Algorithm 457: Finding all cliques of an undirected graph. *Communications of the ACM*, 16(9):575–577, 1973.

[12] D. Chakrabarti, Y.Zhan, and C.Faloutsos. R-mat: A recursive model for graph mining. In *In Proc. 4th SIAM Intl. Conf. on Data Mining*, 2004.

[13] J. Cheng, Y. Ke, and et al. Finding maximal cliques in massive networks by h*-graph. In *SIGMOD*, 2010.

[14] J. Cheng, L. Zhu, and et al. Fast algorithms for maximal clique enumeration with limited memory. In *KDD*, 2012.

[15] C.Olston, B.Reed, and et al. Pig latin: A not-so-foreign language for data processing. In *SIGMOD*, 2008.

[16] D.A.Bader and K.Madduri. Gtgraph: A synthetic graph generator suite. In *http://www.cse.psu.edu/ madduri/software/GTgraph/*, 2006.

[17] D.Eppstein and D.Strash. Listing all maximal cliques in large sparse real-world graphs. In *10th International Symposium on Experimental Algorithms*, 2011.

[18] E.A.Akkoyunlu. The enumeration of maximal cliques of large graphs. *SIAM Journal on Computing*, 2(1), 1973.

[19] D. Eppstein, M. Löffler, and et al. Listing all maximal cliques in sparse graphs in near-optimal time. In *ISAAC(1)*, pages 403–414, 2010.

[20] E.Tomita, A.Tanaka, and H.Takahashi. The worst-case time complexity for generating all maximal cliques and computational experiments. *Theoretical Computer Science*, 363(1):28–42, 2006.

[21] F.Cazals and C.Karande. A note on the problem of reporting maximal cliques. *Theoretical Computer Science*, 407(1-3):564–568, 2008.

[22] F.Cazals and C.Karande. A note on the problem of reporting maximal cliques. *Theoretical Computer Science*, 407(1):564–568, 2008.

[23] G.A.Pavlopoulos, M.Secrier, and et al. Using graph theory to analyze biological networks. *BioData Mining*, 4(10), 2011.

[24] G.Malewicz, M.H.Austern, and et al. Pregel: A system for large-scale graph processing. In *SIGMOD*, 2010.

[25] J.Pei, D.Jiang, and et al. On mining cross-graph quasi-cliques. In *KDD*, 2005.

[26] K.Makino and T.Uno. New algorithms for enumerating all maximal cliques. In *SWAT, Lecture Notes in Computer Science 3111*, pages 260–272, 2004.

[27] J. Leskovec, K. J.Lang, and et al. Statistical properties of community structure in large social and information networks. In *WWW*, pages 695–704, 2008.

[28] L.G.Francois. Powers of tensors and fast matrix multiplication. In *Proceedings of the 39th International Symposium on Symbolic and Algebraic Computation (ISSAC)*, 2014.

[29] L. Lu, Y. Gu, and et al. dmaximalcliques: A distributed algorithm for enumerating all maximal cliques and maximal clique distribution. In *IEEE International Conference on Data Mining Workshops*, pages 1320–1327, 2010.

[30] M.C.Schmidt, N.F.Samatova, K.Thomas, and B.H.Park. A scalable, parallel algorithm for maximal clique enumeration. *Journal of Parallel and Distributed Computing*, (69):417–428, 2009.

[31] M.Haraguchi and Y.Okubo. A method for pinpoint clustering of web pages with pseudo-clique search. In K.Jantke, A.Lunzer, and etc., editors, *Federation over the Web*, volume 3847 of *Lecture Notes in Computer Science*, pages 59–78.

[32] M.Isard, M.Budiu, and et al. Dryad: Distributed data-parallel programs from sequential building blocks. In *EuroSys*, 2007.

[33] N.Chiba and T.Nishizeki. Arboricity and subgraph listing algorithms. *SIAM Journal on Computing*, 14(1):210–223, 1985.

[34] N.Modani and K.Dey. Large maximal cliques enumeration in sparse graphs. In *CIKM*, pages 1377–1378, 2008.

[35] Q.Chen, Z.H.Li, Y.Li, and Z.G.Ives. Parallelizing clique and quasi-clique detection over graph data. In *http://www.wowbigdata.com.cn/clique/parallelclique.pdf*, 2014.

[36] S.B.Seidman and B.L.Foster. A graph theoretic generalization of the clique concept. *Journal of Mathematical Sociology*, (6):139–154, 1978.

[37] S.Tsukiyama, M.Ide, and I.Shirakawa. A new algorithm for generating all the maximal independent sets. *SIAM Journal on Computing*, 6(3):505–517, 1977.

[38] T.Washio and H.Motoda. State of the art of graph-based data mining. *SIGKDD Explorations Newsletter*, 5(1):59–68, 2003.

[39] V.Stix. Finding all maximal cliques in dynamic graphs. *Computational Optimization and Applications*, (27):173–186, 2004.

[40] B. Wu and X. Pei. A parallel algorithm for enumerating all the maximal k-plexes. In *PAKDD*, pages 476–483, 2007.

[41] B. Wu, S. Yang, and et al. A distributed algorithm to enumerate all maximal cliques in mapreduce. In *International Conference on Frontier of Computer Science and Technology*, pages 45–51, 2009.

[42] S. Yang, B. Wang, and et al. Efficient dense structure mining using mapreduce. In *IEEE International Conference on Data Mining Workshops*, pages 332–337, 2009.