Parallelizing Clique and Quasi-Clique Detection over Graph Data
Qun Chen, Li You, Zhanhuai Li and Zachary G. Ives

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 efficiently performed on a machine cluster given that it is large enough.

In this paper, we first propose a new approach for maximal clique and quasi-clique enumeration, which identifies dense subgraphs by recursive graph partitioning. Given a connected graph \( G = (V, E) \), it has a space complexity of \( O(|E|) \) and a time complexity of \( O(|E|\mu(G)) \), where \( \mu(G) \) represents the number of different cliques (quasi-cliques) existing in \( G \). It recursively divides a graph until each task is sufficiently small to be processed in parallel. We then develop parallel solutions and demonstrate how graph partitioning can enable effective load balancing. 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. In the parallel setting, it can achieve the speedups of up to 10x over existing approaches on large graphs. Our parallel algorithms are implemented and evaluated on MapReduce, a popular shared-nothing parallel framework, but can easily generalize to other shared-nothing or shared-memory parallel frameworks.

Index Terms—Maximal Clique Enumeration, Maximal K-plex Enumeration, Parallel Graph Processing, MapReduce

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 [41], [32], Web mining [36], entity resolution [10], graph mining [30], [50], 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.

Maximal clique and quasi-clique enumeration is NP-Complete. Hence a great deal of effort has been spent on efficient search algorithms [11], [20], [21], [18], [45], [19]. Most of existing algorithms for maximal clique and quasi-clique enumeration are based on the classical algorithm proposed by Bron and Kerbosch (BK) [11], which uses a backtracking technique to explore search space and limits the size of its search space by remembering the search paths it has already visited. A variant [20] of the BK algorithm also provides a worst-case-optimal solution. In practice, the BK algorithm has been widely reported as being faster than its alternatives [22], [16].

Data-intensive applications usually require clique and quasi-clique detection to be operated over large graphs, hence there is a need to parallelize it on a sufficiently large machine cluster. There have been a variety of proposals that divide the graph into smaller subcomponents and exploit parallelism to improve performance [47], [34], [49], [35], [14]. They have been empirically shown to speed computation in massive networks. However, built on classical sequential algorithms, 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. Versus prior work in this area, its key insight is to exploit iterative decomposition during the computation. It recursively divides a graph until each task is sufficiently small to be processed in parallel. As a result, its computation can be effectively parallelized across a machine cluster such that the computation, which may be too time-consuming on a single machine, can be efficiently performed in parallel.

Two common parallel frameworks for graph data processing are the MapReduce model [3] 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. 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 evaluation due to the maturity and wide availability of its implementations. However, the implementation can easily generalize to other shared-nothing or shared-memory parallel architectures, such as BSP and MPI. The major contributions of this paper are summarized as follows:

1) We propose a novel and efficient approach for maximal clique enumeration. Given a connected graph \( G = (V, E) \), it has a space complexity of \( O(|E|) \) and a time complexity of \( O(|E|\mu(G)) \), where \( \mu(G) \) represents the number of distinct cliques in \( G \).

2) We develop a parallel solution to maximal clique enumeration by parallelizing the proposed algorithms and implementing the corresponding parallel algorithms 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 over a wide variety of graph data available in open-source. Our extensive experiments demonstrate that it performs considerably better than existing techniques in both centralized and parallel settings. In the parallel setting, our approach achieves the speedups 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 techniques. Section 3 presents our new sequential algorithms for maximal clique and k-plex enumeration. Section 4 presents our parallel solutions to maximal clique and k-plex enumeration 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-plex 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-plex 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 a 1-plex corresponds to a 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 problems of maximal clique and k-plex enumeration refer 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 [16], [35]. In this subsection, we sketch a variant of the BK algorithm proposed in [20]. It employs the same pruning methods as BK, but has the optimal worst-case time complexity of \( O(3^{n/3}) \).

![An Example Graph and Search Tree](image)

The algorithm is in essence a depth-first search, augmented with pruning tricks. Given a current vertex \( v \) and a set of candidate vertices \( S \), it iteratively chooses a vertex \( u \) in \( S \) such that \( N(u) \) has the biggest intersection set with \( S \), in which \( N(u) \) represents the set of \( u \)'s neighboring vertices in \( S \). When the candidate set \( S \) becomes empty, the algorithm outputs corresponding cliques and backtracks. It recursively traverses a search tree, performing the operations of vertex selection, set update, clique generation and backtracking. An example of the search process is shown in Figure 1(a), which computes the maximal cliques containing vertex 4.

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 → 7 → 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 also 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 re-partitions 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 Solutions on MapReduce

The typical parallel approaches [47], [34], [35] enumerate maximal cliques and k-plexes for different vertices in a graph in parallel. In 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 [46] 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 [28], 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 \( G \). The parallel search consists of two steps. In the first one, the parallel approach computes each vertex’s 2-hop information. It can be completed by a MapReduce cycle. In the second step, it searches for each vertex’s maximal cliques in parallel. For the computation on an individual vertex, it simply adopts the classical sequential algorithms.

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 parallel performance bottleneck. The method proposed in [35] can parallelize maximal clique enumeration on an individual vertex. 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, as we will show in Section 5, its parallel efficiency may still be limited by unevenness of search subtree sizes.

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, we focus on the general problem of identifying the maximal cliques containing a specific vertex \( v \) in \( G \).

![Fig. 2: A Graph Partitioning Example](image)

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 \( G_v \). The 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 \( G_7^+ \) 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, we have the following theorem:

**Theorem 1.** Given a graph \( G_v \) consisting of vertex \( v \) and its neighboring vertices and a vertex \( u \) in \( G_v \), we partition \( G_v \) into two subgraphs, \( G_u^+ \) and \( G_u^- \), in which \( G_u^+ \) is the induced subgraph consisting of vertex \( u \) and its neighboring vertices and \( G_u^- \) is the induced subgraph consisting of all the vertices not in \( G_u \) and their neighboring vertices. Then,
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 $w$, it should be an induced subgraph of $G^+_w$. Otherwise, it should contain at least one vertex not in $G^+_w$. 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 $G^-_w$. Therefore, the maximal clique is an induced subgraph of $G^-_w$. □

According to Theorem 1, 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 $G_u$ are contained in the set of the resulting cliques. Unfortunately, a resulting clique generated by the above process cannot 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 GP algorithm based on Graph Partitioning, for maximal clique enumeration, analyze its complexity bounds and give details on its efficient implementation.

#### 3.2.1 A General Algorithm

**Algorithm 1** `enumerateClique(anchor, cand, not)`

```plaintext
1: if (G(cand) is a clique) then
2:   Output the clique G(\text{anchor}\cup\text{cand});
3: else
4:   while (G(cand) is NOT a clique) do
5:     Choose a vertex $v$ in cand;
6:     anchor$^+$=anchor $\cup \{v\}$;
7:     cand$^+$=cand $\cap N(v)$;
8:     not$^+$=not $\cap N(v)$;
9:     if ($\exists u \in \text{not}^+: u$ is connected to all the vertices in cand$^+$) then
10:       enumerateClique(\text{anchor}^+, \text{cand}^+, \text{not}^+);
11:     end if
12:     cand=cand $\setminus \{v\}$;
13:     not=not $\cup \{v\}$;
14: end while
15: if ($\exists u \in \text{not}: u$ is connected to all the vertices in cand) then
16:   Output the clique G(\text{anchor}\cup\text{cand});
17: end if
18: end if
```

The algorithm, as shown in Algorithm 1, enumerates maximal cliques by recursively partitioning a graph. The function employs three sets of vertices to record the partitioning progress and prune the subtrees that can not generate maximal cliques:

- (anchor set) A set of vertices that have been selected as partitioning anchors and should be contained by the resulting cliques;
- (cand set) A set of candidate vertices that can serve as partitioning anchors in the following operations;
- (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.

Given a connected graph $G=(V,E)$, initially, anchor$=\emptyset$, cand$=V$, and not$=\emptyset$. 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 chooses a vertex $v$ in cand as the partitioning anchor and partitions G(cand) into $G(cand^+)$ and $G(cand^-)$. $G(cand^+)$ consists of $v$ and its neighboring vertices in $G(cand)$ (Line 6-8), $G(cand^-)$ consists of all the vertices in $G(cand)$ except $v$ (Line 12-13). The algorithm recursively processes the subgraph $G(cand^+)$ (Line 9-11). Note that before the recursive function is invoked, the algorithm prunes the search space by inspecting whether there exists a vertex in the current not$^+$ set that is connected to all the vertices in the current cand$^+$ set (Line 9). Updating $G(cand)$ with $G(cand^-)$ (Line 12-13), it then iteratively invokes the partition operation to search for the maximal cliques in $G(cand^+)$ until $G(cand^-)$ becomes a clique (Lines 4-14). After $G(cand^-)$ becomes a clique, the algorithm checks whether it is maximal (Line 15-17).

We have Theorem 2, whose proof is presented in Appendix A.

**Theorem 2.** Algorithm 1 exactly returns all the maximal cliques in $G$.

A straightforward implementation of Algorithm 1 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 1 can be processed in $O(|E|)$ time. On the space and time complexity of Algorithm 1, we have Theorem 3, whose proof is presented in Appendix B. Note that in Theorem 3, different cliques include both maximal and non-maximal cliques contained by maximal cliques.

**Theorem 3.** Given a connected graph $G=(V,E)$, Algorithm 1 has the space complexity of $O(|E|)$ and the time complexity of $O(|E|\mu(G))$, in which $\mu(G)$ represents the number of different cliques in $G$.

#### 3.2.2 Efficient Implementation

The efficiency of Algorithm 1 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; (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 $G^-_v$. 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 other graph $G_\neg^-$ 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.

We store the vertices and their adjacency lists in the original graph $G$ in a hash table with vertex id as the hash key. Similarly, we store $G(cand^+)$ and $G(cand^-)$ by hash tables. The intersection of two vertex sets are performed by hash look-ups. Clique verification is achieved by checking vertex degrees. The degree of a vertex $v_i$ in the cand$^+$ set of $G^+_v$ is computed by intersecting the adjacency set of $v_i$ with the cand$^+$ set. For the vertices in the cand$^-$ set of $G^-_v$, only those connected to $v$ needs to decrease their degrees by 1. Selecting a partitioning anchor with the minimal degree in the cand hash table with vertex id as hash key, however, requires $O(|cand|)$ 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 hashtset 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 hashtset of the first entry in the sorted list. It takes only constant 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 ($k \geq 2$) 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 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 iteratively partition $G^-_v$ until it becomes a k-plex. All the generated $G^+_v$’s are recursively partitioned.

The recursive function for maximal k-plex enumeration is shown in Algorithm 2. The anchor, cand and not vertex sets serve the purposes similar to those explained in the algorithms for maximal clique enumeration. Lines 8-10 compute the three vertex sets for $G^+_v$. Line 11 specifies a pruning condition. If there exists a vertex $u$ in not$^+$ such that $u$ is connected to all the vertices in anchor$^+$ and cand$^+$, then any induced k-plex of $G^+_v$ is not maximal. As a result, the $G^+_v$ branch can be pruned. 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. We have the following theorem, whose proof is similar to the proof of Theorem 2.

Theorem 4. Algorithm 2 exactly returns all the maximal k-plexes in $G$.

Executing a depth-first search in a traversal tree, Algorithm 2 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 2 can not guarantee that the vertices in the anchor$^+$, cand$^+$ and not$^+$ sets of $G^+_v$ are connected to the partitioning anchor $v$. As a result, Algorithm 2 requires $O(|V|^2)$ space to store intermediate results. On time complexity, each line of statement in Algorithm 2 can be performed in $O(|E|)$ time. The size of the traversal tree generated by the recursive function is bound by $O(\mu_k(G))$, in which $\mu_k(G)$ represents the number of different k-plexes in $G$. Therefore, we have the following theorem:

Theorem 5. Given a connected graph $G=(V,E)$, Algorithm 2 has the space complexity of $O(|V|^2)$ and the time complexity of $O(|E|\mu_k(G))$.

3.3.2 A Space Optimal Algorithm

To achieve the optimal space complexity, we present a variant of Algorithm 2 (as shown in Algorithm 3), which has the same time complexity $O(|E|\mu_k(G))$ but requires only $O(|E|)$ space. Besides three vertex sets, anchor, cand and not, 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

```
Algorithm 2 enumerateKplex(anchor, cand, not)
1: if (G(anchorUCand) is a k-plex) then
2: if (\exists u \in not:G(anchorUCand\cup\{u\}) is a k-plex) then
3: Output G(anchorUCand);
4: end if
5: else
6: while (G(anchorUCand) is NOT a k-plex) do
7: Choose a vertex v in cand;
8: anchor+=anchor \cup \{v\};
9: cand+=\{u|u \in cand \land G(anchor+ \cup \{u\}) is a k-plex\};
10: not+=\{u|u \not\in not \land G(anchor+ \cup \{u\}) is a k-plex\};
11: if (\exists u \not\in not:u is connected to all the vertices in anchor+ and cand+) then
12: enumerateKplex(anchor+, cand+, not+);
13: end if
14: cand=cand \setminus \{v\};
15: not=not \cup \{v\};
16: end while
17: if (\exists u \not\in not:G(anchorUCand\cup\{u\}) is a k-plex) then
18: Output G(anchorUCand);
19: end if
20: end if
```
Algorithm 3 A Space Optimal Algorithm for Maximal K-plex Enumeration

Input: A graph $G$ with vertex set $V$ and edge set $E$;
1: Stack $S$ ← $\emptyset$;
2: anchor ← $\emptyset$;
3: cand ← $V$;
4: not ← $\emptyset$;
5: while (G(anchor∪cand) is not a k-plex) or ($S \neq \emptyset$) do
6:  if G(anchor∪cand) can not be pruned then
7:     if G(anchor∪cand) is a k-plex then
8:        Output G(anchor∪cand);
9:     end if
10:    end if
11:  else
12:     Choose a vertex $v$ in cand;
13:     S.push($v^+$);
14:     anchor$\leftarrow$anchor∪{$v$};
15:     cand=$\{u|u \in$ (cand-$\{v\}) \land G(\text{anchor} \cup \{u\})$ is a k-plex$\}$;
16:     end if
17:  end while
18:  end if
19: end while

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 2, Algorithm 3 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 2. 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 $G_-$ 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 $G_-$ branch.

Algorithm 3 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 3 can be performed in $O(|E|)$ time. Therefore, we have the following theorem:

**Theorem 6.** Given a connected graph $G=(V, E)$, Algorithm 3 has a space complexity of $O(|E|)$ and a time complexity of $O(|E|\mu_k(G))$.

### 3.3.3 Efficient Implementation

Even though Algorithm 3 achieves the same time complexity as Algorithm 2 with the optimal space complexity, it does not remember the $G_+^v$ subgraphs while performing graph partitioning operations. Neither does it filter out the unnecessary vertices in the not sets while traversing along the $G_+^v$ branches. These properties make it substantially less efficient than Algorithm 2 in practical implementation.

For efficient implementation of Algorithm 2, 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 2 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, $G(\text{anchor} \cup \text{cand} \cup \{u\})$ is a k-plex if and only if: (1) the combined degree of $u$ ($\deg_a(u) + \deg_c(u)$) should be at least $(|\text{anchor}| + |\text{cand}| - k + 1)$; (2) each vertex $w$ in anchor and cand, whose combined degree is equal to $(|\text{anchor}| + |\text{cand}| - k)$, should be connected to $u$. For the condition specified in Line 11, 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 $(|\text{anchor}| + |\text{cand}|)$.

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 $\text{anchor}^-$ or $\text{cand}^-$, 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 $\text{not}^-$ 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 $\text{not}^-$, 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 $v$ 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 $\text{anchor}^+$ is computed by intersecting its adjacency vertex set and the cand$^+$ set. The cases for a vertex $w$ in cand$^+$ or not$^+$ are 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 SOLUTIONS

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 Parallel Algorithms

The parallel algorithms consist of two phases. In the first phase, for every vertex v in the graph G, it retrieves an induced subgraph of G whose vertices are relevant to the computation of v’s maximal cliques or k-plexes. In the second step, it performs recursive graph partitioning on every vertex. Both subgraph retrieval and clique computation on individual vertices are distributed across multiple computing nodes.

The algorithms perform the computation on every vertex by Algorithm 1 and 2. Unfortunately, computational cost on individual vertices may be unbalanced. The computations on some vertices may be more expensive than on others because they have larger partitioning traversal trees. In case that the computation on a vertex is too time-consuming, it becomes a parallel performance bottleneck. A good property of our proposed approach is that it enables easy and effective load balancing. Since the resulting subgraphs are independent, load balancing can be achieved by sending some tasks on computing nodes with heavy workload to others with lighter one.

Generally, workload can be balanced across computing nodes by repeatedly invoking the compute-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.

In summary, the parallel algorithm for maximal clique (or k-plex) detection with load balancing consists of the following two phases:

1) **Subgraph Retrieval:** For every vertex v in the graph G, retrieve the induced graph G_v whose vertices are relevant to v’s maximal clique (k-plex) computations in parallel;

2) **Iterative Computation:**
   - **compute phase:** For each computing node, compute maximal cliques (k-plexes) of the graphs assigned to it by recursive graph partitioning;
   - **shuffle phase:** Evenly reshuffle all the unfinished graphs across the nodes;

4.2 MapReduce Implementation

In this subsection, we first describe the implementation of the parallel solution for maximal clique enumeration based on MapReduce in detail and then briefly describe the implementation for maximal k-plex enumeration, which is similar.

4.2.1 Maximal Clique Enumeration

It is obvious that subgraph retrieval for maximal clique enumeration can be built on 2-hop retrieval. 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 cliques consist 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.

The Map phase of the first cycle reads the triangles into memory and maps them to corresponding reducers according to their keys {v_i, v_2, ..., 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 4. It dequeues a G_{v_i} subgraph from Q_c and iteratively partitions it in a depth-first manner. If the resulting G_{v_i}^+ 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_{v_i}^+ in the same manner as G_{v_i} (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 be 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 4. The MapReduce cycle is iteratively invoked until no unfinished subgraph is left.

### 4.2.2 maximal k-plex enumeration

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

Given a connected graph $G = (V, E)$, let $d(u, v)$ denote the number of edges in the shortest path between two vertices $u$ and $v$ in $G$. The diameter of $G$, denoted by $\text{diam}(G)$, is defined as $\text{diam}(G) = \max_{u,v \in V} d(u, v)$. A connected graph $G$ is $\gamma$-quasi-complete if every vertex in $G$ has a degree at least $\gamma \cdot (|V| - 1)$. According to [28], the relationship between the diameter of a $\gamma$-quasi-complete graph and $\gamma$ can be established as follows:

**Theorem 7.** Let $G$ be a $\gamma$-quasi-complete graph such that $n = |V| \geq 1$. If $\frac{1}{2} \leq \gamma \leq \frac{n-2}{n-1}$, then $\text{diam}(G) \leq 2$.

According to Lemma 7, with low k values ($k \leq (|V| - 1)$), any maximal k-plex of $v$ consists of $v$ and its 2-hop neighbors.

The 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 exploits 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.

### 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. The typical parallel BK approach confines the computation on an individual vertex to a computing node. We also compare the GP approach with the parallel BK approach enhanced with the dynamic load balancing technique (denoted by BK-L), which was proposed in [35]. It was implemented by MPI in [35]. 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 4, 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.

On maximal k-plex enumeration, we compare our sequential algorithm with a variant of the BK search algorithm proposed in [46] because of its effective pruning methods. In the parallel setting, we compare the GP approach with the BK and BK-L approach.

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 [4], 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 [15]. A SSCA#2 graph is directed, and made up of random-sized cliques,
TABLE 1: Details of the Real and Synthetic Graph Datasets

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Data Description</th>
<th>Number of Vertexes</th>
<th>Number of Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>D1</td>
<td>Email Network from a EU Research Institution</td>
<td>265,214</td>
<td>364,481</td>
</tr>
<tr>
<td>D2</td>
<td>Web graph from Google</td>
<td>875,713</td>
<td>4,322,051</td>
</tr>
<tr>
<td>D3</td>
<td>Web graph of Berkeley and Stanford</td>
<td>685,230</td>
<td>7,600,595</td>
</tr>
<tr>
<td>D4</td>
<td>Wikipedia communication network</td>
<td>1,928,669</td>
<td>3,494,674</td>
</tr>
<tr>
<td>D5</td>
<td>Pokec online social network</td>
<td>1,632,803</td>
<td>30,622,564</td>
</tr>
<tr>
<td>D6</td>
<td>Social circles fromTwitter</td>
<td>11,316,811</td>
<td>85,331,846</td>
</tr>
</tbody>
</table>

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 |

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.

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 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 1.

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 0.20.2) [2]. Each experiment was run three times and its running time averaged. In case that processing the entire graphs listed in Table 1 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 [27] and compute their maximal cliques and k-plexes over the entire graphs. The tested sets include both the vertices having large degrees and the vertices having small degrees. The observations coming from the experiments can be applied to the entire graph dataset.
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 minimal 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 performs considerably better 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 1 is beyond the capability of a computing node in our experimental setting. Therefore, we randomly select 10% of the vertices in \( D_1 \), 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 4. They are similar to what were reported on maximal clique enumeration except that GP outperforms 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 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. On SSCA graphs, GP achieves a speedup of up to 6x over BK.

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 run the GP algorithm on the real graphs \((D_1 - D_5)\) listed in Table 1. 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 5. 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, thus omitted 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

In this subsection, we compare the GP approach against the BK and BK-L approaches. 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.

The rest of this subsection is organized as follows: Subsection 5.2.1 evaluates the performance of the parallel approach for maximal clique enumeration; Subsection 5.2.2 evaluates the performance of the parallel approach for maximal k-plex enumeration; Subsection 5.2.3 evaluates the parallelizability of our GP approach, how its performance varies as the size of machine cluster increases.

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 the parameter \( m \) in Algorithm 4 by the number of vertices contained by a graph. It is set to be 50. 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 1 can be efficiently processed on a single worker, we do not evaluate the performance of parallel approaches on them. 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. Also note that processing the entire graph of the Twitter dataset \((D_6)\) takes too long on our ten-machine cluster. We therefore generate 5 random test tasks, denoted by \( D_6^1, \ldots, D_6^5 \), by randomly choosing 1% of the vertices in the graph and enumerating their maximal cliques over the entire graph.

The comparative results on the Twitter and R-MAT graphs are presented in Figure 6 (a)-(d). On the Twitter graph, Figure 6 (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. Closer scrutiny reveals that there exist some vertices heavily

![Fig. 5: Sensitivity evaluation of anchor selection strategies: the strategy of selecting the vertex with the smallest degree achieves the best performance.](image-url)
connected to other vertices in these graphs. 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 $D_8^2$, 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 usually 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 6 (b). The evaluation results on the R-MAT graphs, as shown in Figure 6 (c) and (d), are similar except that the performance difference among the three approaches appears less significant. 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 k-plex enumeration. We test their performance in the case of $k = 2$. Note that maximal 2-plex enumeration is computationally more expensive than maximal clique enumeration and processing the entire graphs of some real datasets listed in Table 1 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 in terms of runtime and MapReduce cycles on real graphs are presented in Figure 6 (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. 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 is more considerable. It achieves the speedups of over 10x on $D_2$ and $D_3$. 
5.2.3 Parallelizability Evaluation

![Graphs and charts illustrating parallelizability evaluation](image)

Fig. 7: Parallelizability evaluation for maximal clique and 2-plex enumeration: the GP approach achieves desired parallelizability

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.

The evaluation results of maximal clique enumeration on the Twitter graph are presented in Figure 7 (a). We present the results of 5 sample test tasks on the Twitter graph, $D_6^0$, $D_6^4$, $D_6^6$, $D_6^8$, and $D_6^{10}$, with increasing runtime. It can be observed that on all the tasks, increasing cluster size results in improved performance. We also observe that the speedup increases with runtime. On the task with the longest runtime ($D_6^{10}$), the GP approach achieves the best speedup: its performance on 10-machine cluster achieves a speedup close to 5 compared with the result on 2-machine cluster. The evaluation results of maximal 2-plex enumeration, as shown in Figure 7 (b), are similar. We present the results of 5 test tasks on real graphs, $D_1$, $D_5$, $D_3$, $D_2$ and $D_4$, with increasing runtime. On $D_1$ and $D_2$, we process the entire graphs. On $D_3$-$D_5$, we only process 1% of vertices in the graphs. Similar to what was observed in maximal clique enumeration, more machines lead to improved parallel performance and the speedup increases with computational load (runtime).

Our experiments in this subsection demonstrate that our parallel approach for maximal clique and 2-plex enumeration can effectively leverage available machines to achieve improved parallel performance. It achieves desired parallelizability and scalability.

6 RELATED WORK

Maximal clique and k-plex enumeration have been studied extensively in the literature [20], [39], [45], [13], [25], [42]. Focusing on centralized search algorithms, they usually rely on global state and cannot be easily implemented in the parallel setting. Besides the classical BK algorithms, another category of algorithms [44], [37], [31] 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. There are also some work [8], [40] studying the closely related problem of detecting maximum clique. They used variants of the existing algorithms for maximal clique enumeration. The work in [29], [51], [48], [12], [17] extended the definition of clique to dense subgraph structures other than k-plex and studied their applications. The proposed algorithms in these work were centralized and implemented on a single machine.

Existing parallel approaches [47], [34], [46], [38], [35], [14] for maximal clique and k-plex enumeration were based on either MPI or MapReduce. They distribute the vertices across computing nodes and compute every vertex’s dense subgraphs in parallel. They used the BK algorithm or its variants to compute 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 [35]. Recently, [26] proposed a scalable and fault-tolerant parallel solution for maximum clique detection using MapReduce.

Iteration has been the focus of recent cluster data processing systems [9], [5], [43], [24], [1], [33], [6]. Our parallel approach can be similarly implemented on these platforms. Their improved support for iteration can result in more efficient implementations.

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. Compared with previous approaches, it achieves smaller search space and is also inherently more parallelizable. 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


APPENDIX A

PROOF OF THEOREM 2

Proof: Firstly, if without the pruning operations specified in Lines 9 and 15 of Algorithm 1, 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 cannot generate any new maximal clique. Consider a clique C_1 in the graph G(\text{anchor} \cup \text{cand}). Suppose that the vertex u in the not set is connected to all the vertices in the cand set. Note that Algorithm 1 ensures that every 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_1. 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 anchor set and the cand set. Since the vertex u is not in anchor, but connected to all the vertices in anchor, its exclusion from cand 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(\text{anchor} \cup \text{cand}^{-}), whose recursive partitioning later generates the clique C_1. Since u is connected to all the vertices in anchor, Algorithm 1 ensures that it is in the not set of the partitioned graph G(\text{anchor} \cup \text{cand}). With u being connected to all the vertices in cand, Algorithm 1 should have filtered C_1 out. Contradiction.
APPENDIX B
PROOF OF THEOREM 3

Proof: We first analyze the space complexity of Algorithm 1. It iteratively partitions the $G(\text{cand}^-)$ branch until $G(\text{cand}^-)$ becomes a clique. Besides a $G(\text{cand}^-)$ graph, it also has to store the information of each $G(\text{cand}^+)$ along the partitioning path. Each $G(\text{cand}^+)$ results from a partitioning operation with a vertex $v_i$ as partitioning anchor. Since each vertex in the anchor+, cand+ and not+ sets of $G(\text{cand}^+)$ (except the vertex $v_i$ itself) should be connected to $v_i$, the required space to store $G(\text{cand}^+)$ 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 $G(\text{cand}^+)$ on the partitioning path has a distinct partitioning anchor. As a result, the required space to store all $G(\text{cand}^+)$ 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 the pruning trick specified on Line 9. Obviously, its time complexity is an upperbound on the time complexity of Algorithm 1. The traversal tree generated by the recursive function without pruning is a binary tree, in which each internal node has exactly two children. Since the cliques generated by the $G(\text{cand}^+)$ branch are guaranteed to be different from those generated by the $G(\text{cand}^-)$ branch, each leaf node corresponds to a different clique (maximal or non-maximal). Therefore, the size of the binary tree is bounded by $O(\mu(G))$. Accordingly, the total number of invoked graph partitioning operations is bounded by $O(\mu(G))$. Since each invocation of graph partitioning requires $O(|E|)$ time, the time complexity of the recursive function is $O(|E|\mu(G))$. \qed