Distributed Maximal Clique Computation

Yanyan Xu, James Cheng, Ada Wai-Chee Fu
Department of Computer Science and Engineering
The Chinese University of Hong Kong
yyxu,jcheng,adafu@cse.cuhk.edu.hk

Yingyi Bu
Department of Computer Science
University of California, Irvine
yingyib@ics.uci.edu

***Abstract*—Maximal cliques are important substructures in graph analysis. Many algorithms for computing maximal cliques have been proposed in the literature; however, most of them are sequential algorithms that cannot scale due to the high complexity of the problem, while existing parallel algorithms for computing maximal cliques are mostly immature and especially suffer from skewed workload. In this paper, we first propose a distributed algorithm built on a share-nothing architecture for computing the set of maximal cliques. We effectively address the problem of skewed workload distribution due to high-degree vertices, which also leads to drastically reduced worst-case time complexity for computing maximal cliques in common real-world graphs. Then, we also devise algorithms to support efficient update maintenance of the set of maximal cliques when the underlying graph is updated. We verify the efficiency of our algorithms for computing and updating the set of maximal cliques with a range of real-world graphs from different application domains.**

***Keywords*-maximal clique enumeration; incremental update**

I. Introduction

Let $G = (V, E)$ be a simple undirected graph. A subset of vertices, $C \subseteq V$, is called a **clique** if every vertex in C is connected to every other vertex in C by an edge in G, and C is called a **maximal clique** if any proper superset of C is not a clique. The problem of **maximal clique enumeration** (**MCE**) is to compute the set of maximal cliques in G.

Maximal cliques are elementary substructures in a graph and instrumental in graph analysis such as the structural analysis of many complex networks, graph clustering and community detection, network hierarchy detection, emerging pattern mining, vertex importance measures, etc. The problem of MCE has been extensively studied and there are three main types of algorithms.

The first type is sequential in-memory algorithms [1], [2], [3], [4], [5], [6], which do not scale well for processing large graphs because of the high complexity of MCE. The second type is sequential I/O-efficient algorithms [7], [8], [9], which focus on reducing the high cost of random disk I/Os for processing graphs that cannot fit in main memory. However, reducing the I/O cost does not solve the main computational issue as MCE is a CPU-intensive task. The third type is parallel and distributed algorithms [9], [10], [11], [12], which aim at reducing the elapsed running time by parallelizing the task of MCE. However, the parallel algorithms in [10], [11] require a copy of the entire input graph to be resident in main memory and cannot handle imbalanced workload. The distributed algorithm in [12] partitions a graph and distribute the subgraphs to workers where MCE is processed locally on the subgraphs. However, these algorithms do not deal with imbalanced workload due to skewed degree distribution, and may also have high communication cost since many unwanted edges may be distributed. Recently, another algorithm was proposed to recursively split a graph into smaller subgraphs and distribute the subgraphs to worker machines for MCE [9]. Their algorithm is also not work-efficient and may also have skewed workload due to high-degree vertices, while the subgraph splitting process can be expensive.

In this paper, we study two main problems: *computing the set of maximal cliques* and *updating the set of maximal cliques*. We highlight the main ideas of our algorithms and the main contributions of our work as follows.

To compute the set of maximal cliques efficiently, we examine the computational bottlenecks that hinder the performance of computing the set of maximal cliques as well as parallelizing MCE. We propose a new parallel algorithm based on the share-nothing architecture to overcome these bottlenecks. Specifically, we significantly reduce the cost of a frequent and most costly operation in the process of MCE, as well as use specific vertex orderings that can achieve $O(\sum_{i=1}^{d} n_i i 3^{i/3})$ worst-case time complexity for processing most common real-world graphs (e.g., d-degenerate graphs, power-law graphs, and sparse graphs), where d is the maximum core number of a graph (which is generally small for real-world graphs, see Table I) and n_i is the number of vertices with core number i [13]. Note that $\sum_{i=1}^{d} n_i = |V|$. This is tremendously smaller the optimal worst-case time complexity of MCE for processing general graphs, which is $O(3^{|V|/3})$ [6]. We give detailed analysis of our algorithm and also show that our algorithm achieves balanced workload and the total amount of work performed by parallelizing the MCE task is asymptotically the same as that by sequentially executing the task.

Real-world graphs often undergo frequent changes. However, the number of maximal cliques is notoriously known to be large even for some small graphs. Thus, it is impractical to recompute the set of all maximal cliques for

978-1-4799-5057-7/14 $31.00 © 2014 IEEE
DOI 10.1109/BigData.Congress.2014.31

every graph update, while any small update (e.g., an edge insertion/deletion) to the graph can cause a considerable amount of updates to the set of maximal cliques. We propose efficient algorithms to incrementally update the set of maximal cliques stored in distributed sites, which poses new challenges.

We evaluate the performance of our algorithms on a set of real world graphs from different domains. Our results show that our parallel MCE algorithm is significantly more efficient than an existing MapReduce algorithm for MCE [12]. Our results also verify the efficiency of our algorithms for update maintenance of the set of maximal cliques.

Paper organization. The remainder of the paper is organized as follows. Section II gives the basic notations and defines the problem. Section III discusses the details of our algorithm for computing the set of maximal cliques. Section IV presents the algorithms for incrementally updating the set of maximal cliques. Section V reports the experimental results. Section VI discusses the related work. Finally, Section VII gives our concluding remarks.

II. Notations and Notions

We study the problem of computing and updating maximal cliques in a simple undirected graph, $G = (V, E)$, where V is the set of vertices and E is the set of edges of G. We keep G in its adjacency list representation. Each vertex $v \in V$ is assigned a unique **vertex ID**, denoted by $ID(v)$, where the vertex ID ranges from 1 to $|V|$. Given any two vertices u and v, we use $ID(u) < ID(v)$ or equivalently $ID(v) > ID(u)$ to denote that u is ordered before v according to the order of their IDs. In the adjacency list representation of a graph, vertices are ordered in ascending order of their IDs.

We define the set of **adjacent vertices** of a vertex $v \in V$ as $adj(v) = \{u : (u, v) \in E\}$. We further define $adj(< v) = \{u : u \in adj(v), ID(u) < ID(v)\}$ and $adj(> v) = \{u : u \in adj(v), ID(u) > ID(v)\}$.

A set of vertices, C, where $C \subseteq V$, is a **clique** in G if every $v \in C$ is adjacent to all other vertices in C, i.e., $v \in adj(u)$ for all $u \in (C \setminus \{v\})$. If $\nexists C' \supset C$ such that C' is a clique in G, then C is a **maximal clique**.

We use $\mathcal{M}(G)$ to denote the set of maximal cliques in G. We also use $\mathcal{M}_v$ to denote the set of maximal cliques **starting with** v, i.e., $\mathcal{M}_v = \{C : C \in \mathcal{M}(G), v = argmin_{u \in C} ID(u)\}$, where "$v = argmin_{u \in C} ID(u)$" means "$v \in C$ such that $ID(v) = \min\{ID(u) : u \in C\}$".

The following example illustrates the concepts.

Example 1: Figure 1 shows a graph G with 8 vertices. If we assign the vertex ID in ascending order of the vertex degree, where ties are broken arbitrarily, then we have $ID(h) = 1$, $ID(c) = 2$, $ID(f) = 3$, $ID(a) = 4$, $ID(d) = 5$, $ID(g) = 6$, $ID(b) = 7$, and $ID(e) = 8$. According to this ID assignment and ID ordering, we have

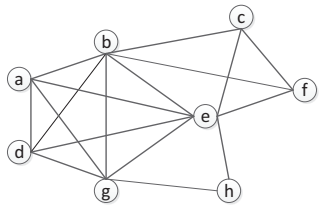


Figure 1. A graph G

$adj(< h) = \emptyset$, $adj(> h) = \{g, e\}$, $adj(< g) = \{h, a, d\}$ and $adj(> g) = \{b, e\}$. There are 3 maximal cliques in G, i.e., $\mathcal{M}(G) = \{\{a, b, d, e, g\}, \{b, c, e, f\}, \{e, g, h\}\}$. Hence, we have $\mathcal{M}_h = \{e, g, h\}$, $\mathcal{M}_c = \{b, c, e, f\}$, $\mathcal{M}_a = \{a, b, d, e, g\}$, and $\mathcal{M}_v = \emptyset$ for $v \in \{b, d, e, f, g\}$.

Problem definition. Given a graph $G = (V, E)$, this paper proposes efficient algorithms for: (1) *Computing the set of maximal cliques, i.e., computing* $\mathcal{M}(G)$; and (2) *Update maintenance of* $\mathcal{M}(G)$ *when* G *is updated.*

III. Computing Maximal Cliques

We first present a parallel algorithm for computing the set of maximal cliques on a *shared-nothing* architecture. The algorithm consists of two phases: *data distribution* and *maximal clique enumeration* (**MCE**), which can be easily implemented in one round of Map and Reduce [14]. We first discuss these two phases, and then we also show how different orderings of vertices can reduce the complexity of MCE in common real-world graphs.

A. Phase I: Data Distribution

The data distribution phase is shown in Lines 1-6 of Algorithm 1. Given a simple undirected graph $G = (V, E)$, the algorithm divides the task of MCE into many sub-tasks to be computed in parallel. The data necessary for MCE at each worker machine is to be distributed according to the following lemma.

Lemma *1:* Computing $\mathcal{M}_v$ requires $ADJ = \{(u, adj(u)) : u \in adj(> v)\} \cup \{(v, adj(v)\}$.

Lemma 1 implies that for each $v \in V$, $(v, adj(v))$ is only needed to compute $\mathcal{M}_u$ for each $u \in (adj(< v) \cup \{v\})$. Thus, we only need to output the key-value pair $\langle ID(u); (v, adj(v))\rangle$ for each $u \in (adj(< v) \cup \{v\})$ instead of $u \in (adj(v) \cup \{v\})$, which will prove to achieve a tremendous reduction in the complexity of MCE for processing common real-world graphs (to be analyzed in Section III-C).

B. Phase II: Maximal Clique Enumeration

The second phase, i.e., MCE, is shown in Lines 7-12 of Algorithm 1. The data for computing $\mathcal{M}_v$ is distributed to an active worker. Note that for each $C \in \mathcal{M}_v$, $(C \setminus \{v\}) \subseteq$

Algorithm 1: Parallel MCE

```
1  Data distribution:
   Input  : ⟨v; adj(v)⟩ for each v ∈ V
2  begin
3      foreach vertex v ∈ V do
4          output ⟨ID(v); (v, adj(v))⟩;
5          foreach vertex u ∈ adj(< v) do
6              output ⟨ID(u); (v, adj(v))⟩;

7  Maximal clique enumeration (MCE):
   Input  : ⟨ID(v); (v, adj(v), {(u, adj(u)) : u ∈ adj(> v)}⟩
            for each v ∈ V
8  begin
9      foreach u ∈ adj(> v) do
10         ADJ_{>v}[u] ← adj(u) ∩ adj(> v);
11         ADJ_v[u] ← adj(u) ∩ adj(v);
12     LocalMCE({v}, adj(> v), adj(< v), ADJ_{>v}, ADJ_v);
```

Algorithm 2: LocalMCE(C, $cand$, $prev$, $ADJ_{>v}$, ADJ_v)

```
1  if cand = ∅ and prev = ∅ then
2      output C as a maximal clique;
3  else if cand ≠ ∅ then
4      let u_p be the vertex in cand that maximizes
       |cand ∩ ADJ_{>v}[u_p]|;
5      U ← cand \ ADJ_{>v}[u_p];
6      sort U in descending order of |ADJ_{>v}[u]| for all u ∈ U;
7      foreach u ∈ U do
8          cand ← cand \ {u};
9          cand' ← cand ∩ ADJ_{>v}[u];
10         foreach w ∈ cand' do
11             ADJ'_{>v}[w] ← ADJ_{>v}[w] ∩ cand';
12             ADJ'_v[w] ← ADJ_v[w] ∩ prev;
13         LocalMCE(C ∪ {u}, cand', prev ∩ ADJ_v[u],
           ADJ'_{>v}, ADJ'_v);
14         prev ← prev ∪ {u};
```

$adj(> v)$. Thus, to enumerate the maximal cliques in $\mathcal{M}_v$, we only need $adj(u) \cap adj(> v)$, denoted by $ADJ_{>v}[u]$, for each $u \in adj(> v)$. However, to check maximality of the cliques, we also need $adj(u) \cap adj(v)$, denoted by $ADJ_v[u]$, for each $u \in adj(> v)$. Then, the algorithm invokes the procedure "*LocalMCE*" to compute $\mathcal{M}_v$ locally at the worker, as shown in Algorithm 2.

We first explain some notations used in Algorithms 2. We use C to denote the clique currently being enumerated, ***cand*** to denote the set of candidate vertices that can be used to expand or form a clique, and ***prev*** to denote a set of vertices that are in some other maximal cliques (either enumerated previously by the same worker or enumerated by another worker) so that C is maximal only if $prev = \emptyset$. We also use $ADJ_{>v}$ and ADJ_v to denote the sets $\{ADJ_{>v}[u] : u \in adj(> v)\}$ and $\{ADJ_v[u] : u \in adj(> v)\}$, respectively.

The LocalMCE algorithm starts from a set C initially consisting of a single vertex, and repeats the process "find a candidate vertex $u \in cand$ that is a common neighbor of all vertices in the current C and then add u to C" until there exists no common neighbor of the current C, in which case $cand = \emptyset$, and C is returned as a maximal clique if $prev = \emptyset$. When we grow the current clique C to $C' = (C \cup \{u\})$, we refine $cand$ by intersecting it with $ADJ_{>v}[u]$ because any candidate vertex that can grow C' must be in $ADJ_{>v}[u]$. We also refine $prev$ by intersecting it with $ADJ_v[u]$ because if another maximal clique C'' exists such that C' cannot be grown into a maximal clique in the end, then $(C'' \setminus C')$ must be a subset of $ADJ_v[u]$. For the same reasons, we also refine $ADJ_{>v}[w]$ and $ADJ_v[w]$ for each new candidate vertex $w \in cand'$, by intersecting them with $cand'$ and $prev$, respectively. Then, LocalMCE is invoked recursively to further grow C'.

The following lemma shows that computing $\mathcal{M}_v$ for each $v \in V$ gives the complete set of maximal cliques, $\mathcal{M}$, and no redundant maximal clique is generated.

LEMMA *2:* $\mathcal{M}(G) = \bigcup_{v \in V} \mathcal{M}_v$, and $\mathcal{M}_u \cap \mathcal{M}_v = \emptyset$ for all $u, v \in V$ and $u \neq v$.

A comparison between LocalMCE and classic MCE algorithms. The LocalMCE algorithm is similar to the classic MCE algorithms that apply pruning by pivot vertex (i.e., u_p in Line 4 of Algorithm 2) [1], [2], [6]. Compared with the classic algorithm, we make the following improvements.

When growing the current clique C to a new clique $(C \cup \{u\})$, the classic algorithm refines $cand$ and $prev$ by intersecting each of them with $adj(u)$. During the processing of MCE, these set intersections are the most costly operations, especially because $adj(u)$ can be very large for those high-degree vertices in a power-law graph. Since the number of cliques enumerated (i.e., those 'C's in the intermediate steps of MCE) can be significantly larger than the number of maximal cliques and high-degree vertices are contained in many cliques, we can tremendously reduce the running time of MCE if we can reduce the cost of the set intersections.

Cheng et al. [9] proposed to reduce the cost of set intersection by extracting subgraphs and then performing MCE in the subgraphs. They used a cost model to find the optimal subgraphs, but computing the cost model is NP-hard.

We propose a much simpler but effective mechanism, which is also more suitable for parallel MCE. We observe that by ordering the vertices we only need $ADJ_{>v}[u]$ for growing the current clique and $ADJ_v[u]$ for checking maximality. Thus, we always refine $cand$ by intersecting it with $ADJ_{>v}[u]$ instead of with the whole set $adj(u)$, and we will show in Section III-C that $ADJ_{>v}[u]$ is small even for high-degree vertices in common real-world graphs. Furthermore, whenever we add a new candidate vertex u to C, we refine $ADJ_{>v}[w]$ and $ADJ_v[w]$ for each new candidate vertex w, so that in the subsequent recursive steps we can intersect with the smaller $ADJ_{>v}[w]$ and $ADJ_v[w]$

instead of with $adj(w)$.

C. Ordered MCE and Complexity

The time complexity of the classic algorithm for MCE [1], [2], [6] is $O(3^{|V|/3})$, which is proved to be optimal for processing general graphs [6]. However, we show how different orderings of vertices can reduce the complexity of MCE in many graphs such as power-law graphs and d-degenerate graphs, which are prevalent in real world [15], [3]. We consider the following types of ordering: (1) ordering by vertex degree, (2) ordering by degeneracy number [3], and (3) ordering by the core number of the vertices.

Degeneracy ordering. An undirected graph G is *k-degenerate* if for every subgraph G' of G, there exists some vertex in G' that has k or fewer neighbors within G'. The degeneracy of G is the smallest value of k for which G is k-degenerate. If the degeneracy of G is d, then G has a degeneracy ordering such that if we assign $ID(v)$ according to the degeneracy ordering (i.e., $ID(v) = i$ if v is at the i-th position by the degeneracy ordering), then $|adj(> v)| \leq d$ for all $v \in V$.

Eppstein et al. [3] prove the following complexity of MCE for a d-degenerate graph.

THEOREM *1:* Let $G = (V, E)$ be a graph where the degeneracy of G is d. When the vertices in G are ordered by degeneracy ordering, applying Algorithm 2 to compute $\mathcal{M}_v$ for all $v \in V$ uses $O(|V|d3^{d/3})$ time.

The proof in [3] is for sequential algorithms, while in our case for parallel computation, each worker uses $O(d3^{d/3})$ time for computing $\mathcal{M}_v$.

Since the degeneracy d is quite small for most real-world graphs, especially for sparse graphs and power-law graphs, the above complexity is a significant reduction to the $O(3^{|V|/3})$ complexity for processing general graphs.

Degree ordering. For each $v \in V$, we assign $ID(v) = i$ if v is at the i-th position when the vertices in V are ordered in ascending order of their degree, where ties are broken arbitrarily. Let h be the maximum value of h such that there are h vertices with degree at least h. We give the following complexity analysis.

THEOREM *2:* Given a graph $G = (V, E)$, when the vertices are ordered by their degree, applying Algorithm 2 to compute $\mathcal{M}_v$ for all $v \in V$ uses $O(|V|h3^{h/3})$ time.

Proof: We first show that $|adj(> v)| \leq h$ for all $v \in V$. Suppose on the contrary that there exists a vertex $v \in V$ such that $|adj(> v)| > h$. Since $|adj(> v)| > h$, there are at least $(h+1)$ vertices that are ordered after v, i.e., they have degree at least as large as v. Since $|adj(v)| \geq |adj(> v)| > h$, v has degree at least $(h+1)$ and hence each $u \in adj(> v)$ has degree at least $(h + 1)$. This means that there are at least $(h + 1)$ vertices that have degree at least $(h + 1)$, which contradicts to the fact that h is the maximum value of h such that there are h vertices with degree at least h. Thus, $|adj(> v)| \leq h$ for any $v \in V$.

If $|adj(> v)| \leq h$, then following a similar analysis to Theorem 2 of [3] we can show that computing $\mathcal{M}_v$ by Algorithm 2 uses at most $O(h3^{h/3})$ time. ■

For a typical power-law graph, $h \leq |V|^{0.4}$ [7]. Thus, for processing large power-law graphs, our algorithm can be much faster than the classic algorithms [1], [2], [6].

Core number ordering. Another interesting vertex ordering we can use is based on *k-core* [13]. The k-core of a graph G is the largest subgraph $C_k = (V_{C_k}, E_{C_k})$ of G such that $\forall v \in V_{C_k}$, the degree of v in C_k is at least k. The *core number* of a vertex $v \in V$, denoted by $c(v)$, is defined as the largest k such that v is in C_k.

Let d be the maximum core number of a vertex in G. Note that the degeneracy of G is equal to d. However, we can give an ordering of the vertices by their core number and show that this ordering achieves better time complexity than that by degeneracy ordering given in Theorem 1.

LEMMA *3:* There exists an ordering of the vertices in G s.t. (1) for all $v \in V$, $ID(v) = i$ if v is at the i-th position by the ordering, (2) for all $u, v \in V$, if $ID(u) < ID(v)$, then $c(u) \leq c(v)$, and (3) for all $v \in V_i$, $|adj(> v)| \leq i$.

By the ordering given in the proof of Lemma 3, we obtain the following new complexity. Let V_i be the set of vertices with core number i and $n_i = |V_i|$. Thus, $n_i = |V_{C_i} \setminus V_{C_{i+1}}|$ for $1 \leq i < d$ and $n_d = |V_{C_d}|$.

THEOREM *3:* Given a graph $G = (V, E)$, when the vertices are ordered by core number ordering as given in Lemma 3, applying Algorithm 2 to compute $\mathcal{M}_v$ for all $v \in V$ uses $O(\sum_{i=1}^{d} n_i i 3^{i/3})$ time.

The proof of Theorem 3 is similar to that of Theorem 2.

Since in most real-world graphs, the number of vertices with core number i decreases rapidly when i increases, the complexity using core number ordering can be much smaller than that using degeneracy ordering. The following theorem further shows that the complexity of MCE given in Theorem 3 is close to the lower bound.

THEOREM *4:* The maximum possible number of maximal cliques in a graph G, where the degeneracy of G is d, is given by $\sum_{i=1}^{d-1} n_i 3^{i/3} + (n_d - d)3^{d/3}$, and such a graph exists.

Proof: It is shown that the maximum possible number of maximal cliques in a graph with n vertices is bounded by $3^{n/3}$ [16]. For all $v \in V_i$, $|adj(> v)| \leq i$ according to Lemma 3, which means that the subgraph of G induced by $adj(> v)$ gives at most $3^{i/3}$ maximal cliques. Since any maximal clique in $\mathcal{M}_v$ is a subset of $(\{v\} \cup adj(> v))$, v forms at most $3^{i/3}$ maximal cliques with vertices in $adj(> v)$ and hence $|\mathcal{M}_v| \leq 3^{i/3}$. Thus, $\sum_{v \in V_i} |\mathcal{M}_v| \leq n_i 3^{i/3}$.

When $i = d$, the subgraph induced by V_d, i.e., C_d, has degeneracy d. By Theorem 3 of [3], the maximum possible number of maximal cliques in a graph with degeneracy d is given by $(n_d - d)3^{d/3}$.

Next we show that there exists a graph G that has $\sum_{i=1}^{d-1} n_i 3^{i/3} + (n_d - d)3^{d/3}$ maximal cliques. The graph G is formed by $d/3$ components (for simplicity, we assume that d is a multiple of 3) and a main component. The main component is a Turan's graph $T(d, d/3)$, forming $d/3$ independent sets (each of size 3), and with edges from each vertex in each independent set connected to all vertices in the other independent sets. The other $d/3$ components are $L_1, ..., L_{d/3}$, where L_i contains m_i vertices and there is no edge among vertices in L_i, i.e., L_i is an independent set in G (for simplicity, we assume that $m_i \geq 3$). Each vertex in L_i is connected to all vertices in i independent sets in $T(d, d/3)$. Obviously, vertices in L_i have core number $3i$, and hence $m_i = n_{3i}$ for $i < d/3$, and $n_d = m_{d/3} + d$ since the d vertices in $T(d, d/3)$ are also in the d-core. For $1 \leq i \leq d$, if $i\%d \neq 0$, then $n_i = 0$ (note that no maximal clique is formed by vertices with core number i in this case). Now consider the number of maximal cliques that can be formed by vertices in L_i for $1 \leq i < d/3$. Note that each vertex in L_i is linked to i sets of independent sets $\{I_1, ..., I_i\}$ in $T(d, d/3)$. A maximal clique can be formed by picking one vertex from L_i, and one vertex from each of $I_1, ..., I_i$. Hence the number of maximal cliques that can be formed by vertices in L_i is given by $m_i 3^i = n_{3i} 3^i$. Since $n_d = m_{d/3} + d$, the number of maximal cliques that can be formed by vertices in $L_{d/3}$ is given by $m_{d/3} 3^{d/3} = (n_d - d)3^{d/3}$. Summing up we have a graph with $\sum_{i=1}^{d-1}(n_i)3^{i/3} + (n_d - d)3^{d/3}$ maximal cliques. ■

Theorem 4 implies that the worst-case time complexity of MCE in a graph with degeneracy d is at least $O(\sum_{i=1}^{d-1} n_i 3^{i/3} + (n_d - d)3^{d/3})$ since there are so many maximal cliques in the worst case. Thus, the worst-case time complexity of our algorithm for computing $\mathcal{M}_v$ for all $v \in V_i$ is only a factor of i greater than the lower bound time complexity. Importantly, i is small when n_i is large, and n_d is usually small for most real-world graphs. We further verify the efficiency of MCE by each ordering by experiments.

D. Work Efficiency and Workload Balancing

The following theorem shows that the total amount of work performed by all the machines is asymptotically the same as that by sequentially executing Algorithm 2 to compute $\mathcal{M}_v$ for all $v \in V$, which achieves the best known time complexity (as given in Theorem 3) for computing maximal cliques in a graph with degeneracy d (the previous best complexity is given by [3] as shown in Theorem 1). In addition, it also shows that the workload at each worker is bounded by d instead of $|V|$ or the degree of the vertices.

THEOREM *5:* The total amount of work performed by all the workers for computing the set of maximal cliques by core number ordering, as well as the total amount of data being communicated, is $O(\sum_{i=1}^{d} n_i i 3^{i/3})$. The amount of work done by any worker is $O((|V|/\phi)n_d d 3^{d/3})$, where ϕ is the number of workers.

Note that the same analysis can also be applied if the vertices are ordered by degeneracy ordering or degree ordering.

IV. UPDATING MAXIMAL CLIQUES

We consider two update operations: edge insertion and edge deletion. Note that vertex deletion can be considered as a series of edge deletion followed by the deletion of an isolated vertex, while the deletion (and insertion) of an isolated vertex (i.e., a maximal clique of size 1) is trivial.

We first need to present the data structure that we use to store the maximal cliques. When we apply Algorithm 2 to compute $\mathcal{M}_v$, for each $v \in V$, the process naturally constructs a prefix tree, denoted by T_v, such that the root of T_v is v and each root-to-leaf path represents a maximal clique in $\mathcal{M}_v$. We use the prefix tree structure because it can effectively save storage space by sharing the common subsets among maximal cliques. The prefix trees T_v for each $v \in V$ are stored in a distributed file system.

In the following discussion, we process the insertion and deletion of an edge (u, v), and without loss of generality we assume $ID(u) < ID(v)$.

A. Edge Insertion

When an edge (u, v) is inserted into G, we have two cases to handle: (1) some existing maximal cliques become non-maximal, which need to be deleted; and (2) new maximal cliques appear in G, which need to be inserted. We process the two cases as shown in Algorithm 3.

For Case (1), an existing maximal clique C becomes non-maximal only if C contains either u or v, and $C \subset (adj(u) \cap adj(v)) \cup \{u, v\}$, since now $C \cup \{v\}$ or $C \cup \{u\}$ is a new maximal clique containing C. The following observation shows where C can be found.

OBSERVATION *1:* Let C be a maximal clique before inserting (u, v), where $C \subset (adj(u) \cap adj(v)) \cup \{u, v\}$ and either $u \in C$ or $v \in C$. Then, C is represented by a root-to-leaf path in T_w, where $w \in ((adj(> u) \cap adj(< v)) \cup \{v\})$ or $w \in ((adj(< u) \cap adj(< v)) \cup \{u\})$.

Following Observation 1, first Lines 3, 5 and 6 of Algorithm 3 call the "DeleteNode" procedure (i.e., Algorithm 4) to delete an existing maximal clique C from T_w, if C will become non-maximal after inserting (u, v), i.e., $C \subset cand = (adj(u) \cap adj(v)) \cup \{u, v\}$ (checked in Line 3 of Algorithm 4). Since C is represented by a root-to-leaf path in T_w but a prefix subpath of C may be shared by some other maximal cliques in $\mathcal{M}_w$, we only remove the part of

Algorithm 3: Insert(u, v)

```
1  cand ← (adj(u) ∩ adj(v)) ∪ {u, v};
2  foreach w ∈ ((adj(> u) ∩ adj(< v)) ∪ {v}) do
3      DeleteNode(v, T_w, cand);
4  foreach w ∈ ((adj(< u) ∩ adj(< v)) ∪ {u}) do
5      DeleteNode(v, T_w, cand);
6      DeleteNode(u, T_w, cand);
7      newcand ← (adj(> w) ∩ cand) \ {u, v};
8      newprev ← adj(< w) ∩ cand;
9      foreach w' ∈ newcand do
10         ADJ_{>w}[w'] ← adj(w') ∩ adj(> w);
11         ADJ_w[w'] ← adj(w') ∩ adj(w);
12     invoke LocalMCE({u, v, w}, newcand, newprev,
       ADJ_{>w}, ADJ_w) to generate new maximal cliques
       containing {u, v} and {w} and insert them into T_w;
```

Algorithm 4: DeleteNode$(x, T_w, cand)$

```
1  foreach occurrence of x in T_w do
2      foreach maximal clique C represented by each
       root-to-leaf path via x in T_w do
3          if C ⊂ cand then
4              starting from the leaf node and moving along
               the path up to the root, recursively delete from
               T_w any node that has no child (note that initially
               only the leaf node has no child);
```

the path that is not shared by any other maximal cliques (Line 4 of Algorithm 4).

Now we process Case (2). The following observation shows where a new maximal clique C should be inserted.

OBSERVATION *2:* Let C be a new maximal clique after inserting (u, v) into G. Then, C should be inserted into T_w, where $w \in ((adj(< u) \cap adj(< v)) \cup \{u\})$.

Following Observation 2, we first generate all new maximal cliques that contain $\{u, v, w\}$, for each $w \in ((adj(< u) \cap adj(< v)) \cup \{u\})$, which is processed by the algorithm "LocalMCE" (i.e., Algorithm 2), and we also insert the new maximal cliques into T_w (Lines 7-12 of Algorithm 3).

B. Edge Deletion

The deletion of an edge (u, v) is processed in essentially the reverse way of how we process the edge insertion. As shown in Algorithm 5, we first delete all the existing maximal cliques that contain both u and v, where such maximal cliques appear in T_w, where w is defined in Observation 2. Then, we general all new maximal cliques that contain only u or v, and insert them into T_w, where w is defined in Observation 1.

V. EXPERIMENTAL EVALUATION

We evaluate the performance of our algorithms on a cluster of 64 computing nodes, each with a 2.0GHz and 4GB

Algorithm 5: Delete(u, v)

```
1  cand ← (adj(u) ∩ adj(v)) ∪ {u, v};
2  foreach w ∈ ((adj(< u) ∩ adj(< v)) ∪ {u}) do
3      delete unshared nodes along any path in T_w that
       represents a maximal clique containing both u and v by
       a procedure similar to Algorithm 4;
4      compute all new maximal cliques that contain
       C = {w, u} and insert them into T_w by a procedure
       similar to Lines 7-12 of Algorithm 3; and similarly for
       C = {w, v};
5  foreach w ∈ ((adj(> u) ∩ adj(< v)) ∪ {v}) do
6      compute all new maximal cliques that contain
       C = {w, v} and insert them into T_w by a procedure
       similar to Lines 7-12 of Algorithm 3;
```

Table I
DATASET STATISTICS

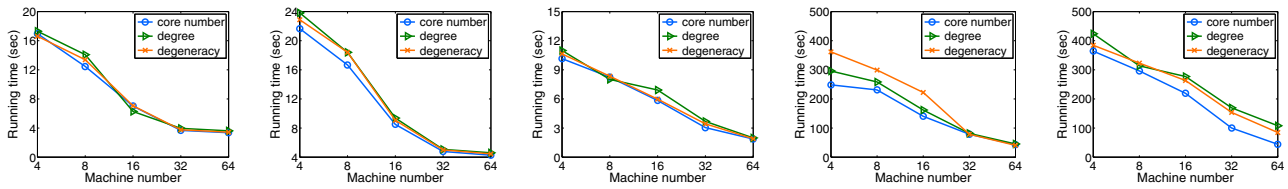
	Youtube	Patents	Google	Skitter	Wiki
$\|V\|$	1134890	3774767	875713	1696415	2394385
$\|E\|$	2987624	16518948	4322051	11059298	4659562
deg_{max}	28754	793	6332	35455	100029
d	51	58	43	111	131
α	17	11	44	67	26
$\|\mathcal{M}(G)\|$	3265953	14787028	1417580	37322351	86333297

RAM. The communication speed between the computing nodes in the cluster is 1Gbps.

We selected five datasets from five different domains in the Stanford Large Network Dataset Collection (`http://snap.stanford.edu/data/`). The Youtube dataset comes from the community networks, where users form friendship each other. The Patents dataset is from the citation networks. The Google dataset is a Web graph from Google and it is selected from the category of Web graphs. The Skitter dataset is an Internet topology graph and it is selected from the category of Systems graphs. The Wiki dataset is a Wikipedia talk (communication) network from the Wikipedia networks. Some of the graphs are directed and we ignore the edge direction to study maximal cliques in these graphs. Table I gives some statistical information about the datasets, including the number of vertices ($|V|$), the number of edges ($|E|$), the maximum vertex degree (deg_{max}), the maximum core number or degeneracy of the graph (d), the size of the maximum clique (α), and the number of maximal cliques ($|\mathcal{M}(G)|$).

A. Results of Computing Maximal Cliques

We evaluate the performance of computing the set of maximal cliques, compared with an existing MapReduce MCE algorithm [12], denoted by **Wu et al.** in Table III. Wu et al.'s algorithm is implemented using Hadoop. We also use the Hadoop distributed file system to store the graph data and the prefix trees that represent the set of maximal cliques, but implemented our own version of Map and Reduce phases for both data distribution (see Section



(a) YouTube (b) Patents (c) Google (d) Skitter (e) Wiki

Figure 2. Running time (in sec) of MCE with different vertex orderings

III-A) and MCE computation (see Section III-B). We ran the algorithms on 4, 8, 16, 32, and 64 machines, respectively, and recorded the elapsed running time (in seconds).

Effect of vertex orderings. We first report the elapsed running time of our algorithm for MCE using different vertex orderings by **core number**, **degree**, and **degeneracy**, respectively, as shown in Figure 2. Note that the running time includes the time to compute the orderings. The results show that for all the datasets, when more machines are used, there is a significant decrease in the elapsed running time. On average, we record 1.60 times reduction in the elapsed running time when the number of machines is doubled for both core number ordering and degeneracy order, while for degree ordering the reduction is 1.54 times. Among the three vertex orderings, core number ordering achieves the best performance consistently in most of the cases. The result thus verifies our analysis that core number ordering gives the lowest time complexity for MCE in Section III-C. The performance of degree ordering is comparable with that of degeneracy ordering in most cases, except for processing the Skitter dataset its performance is considerably worse when 4 to 16 machines are used.

We also measured the data distribution time of our algorithm, and found that the data distribution time is almost the same regardless of which vertex ordering is used. We thus report the data distribution time of core number ordering when different number of machines are used in Table II. The result shows that the data distribution time does not decrease significantly when more than 16 machines are used, which may be due to the fact that the communication time also increases when more machines are used, and the amount of data to be distributed is not large. Compared with the running time shown in Figure 2, we can see that the overall data distribution time is only a small portion of the elapsed running time, especially for the Skitter and Wiki datasets. Thus, it is important to have an efficient algorithm for the main MCE process.

Comparison with Wu et al. [12]. We now compare the performance of our algorithm (by core number ordering) with that of Wu et al.'s algorithm. We report the running time for 16 machines only due to limited space, as the running time of Wu et al.'s algorithm is orders of magnitude larger than ours in all cases while our algorithm also shows a better scalability in terms of number of machines used. As shown in Table III, Wu et al.'s algorithm uses up to orders of magnitude more time than our algorithm, which clearly demonstrates the efficiency of our algorithm.

Table II
DATA DISTRIBUTION TIME (IN SEC) FOR 4 TO 64 MACHINES

	Youtube	Patents	Google	Skitter	Wiki
4	4.8286	4.3470	4.4218	4.1530	4.3232
8	3.7168	3.5472	3.6615	3.5509	3.6869
16	2.8447	2.6010	2.6174	2.9907	2.7394
32	2.4480	2.6241	2.5171	2.3189	2.4165
64	2.5729	2.3870	2.4692	2.4641	2.5176

Table III
RUNNING TIME (IN SEC) OF MCE

	Our algorithm	Wu et al.
Youtube	7.0162	982
Patents	8.4981	2868
Google	5.8394	1178
Skitter	140.3460	> 10000
Wiki	219.2920	> 10000

B. Results of Updating Maximal Cliques

We now evaluate the performance of updating the set of maximal cliques, $\mathcal{M}(G)$. For edge insertion, we randomly generated 1000 edges that are not in G, and insert them into G. For edge deletion, we randomly selected 1000 existing edges in G to be deleted. Tables IV and V report the average elapsed running time for each update operation, running on 4, 8, 16, 32, and 64 machines, respectively.

The results show that updating $\mathcal{M}(G)$ for both edge insertion and deletion is fast for all the datasets, but it is not easy to see a trend when more machines are used. In general, when more machines are used, the running time decreases but the time reduction is not significant. We examined the details and found that when an edge (u, v) is inserted or deleted, most updates are operated on T_u and/or T_v. Since the updates on T_u and/or T_v take much longer time, the time taken by the worker that processes T_u and/or T_v is longer than that by other workers. Since we measure the elapsed time, the finishing time of the last worker determines the running time, and thus using more machines does not help much in this situation. However, we emphasize that the use of vertex ordering has significantly limited the size

Table IV
UPDATE TIME (IN SEC) FOR EDGE INSERTION

	Youtube	Patents	Google	Skitter	Wiki
4	0.0669	0.0623	0.0493	0.0830	0.0188
8	0.0410	0.0423	0.0333	0.0532	0.1304
16	0.0311	0.0299	0.0285	0.0436	0.0702
32	0.0269	0.0286	0.0353	0.0387	0.0727
64	0.0269	0.0254	0.0228	0.0336	0.0421

Table V
UPDATE TIME (IN SEC) FOR EDGE DELETION

	Youtube	Patents	Google	Skitter	Wiki
4	0.0780	0.0731	0.2082	0.0784	0.2459
8	0.0461	0.0486	0.1636	0.0539	0.2606
16	0.0283	0.0409	0.0294	0.0455	0.2375
32	0.0220	0.0247	0.0346	0.0393	0.2062
64	0.0261	0.0243	0.0226	0.0342	0.1529

of any T_v to $O(3^{d/3})$. Without the ordering, the size of T_v is bounded by $O(3^{n/3})$ or $O(3^{|adj(v)|/3})$ in practice, where $O(3^{|adj(v)|/3})$ is still drastically larger than $O(3^{d/3})$ for high-degree vertices (see Table I). Thus, our method has already achieved a good bounded balanced workload.

VI. RELATED WORK

The classic algorithms for MCE are the *backtracking* methods [1], [2], [6], which employ effective pruning by selecting good *pivots* to reduce the search space, and give an optimal worst-case time complexity of $O(3^{|V|/3})$ for processing general graphs [6]. For processing d-degenerate graphs, an extension of the algorithm that uses degeneracy ordering achieves a time complexity of $O(d|V|3^{d/3})$ [3]. Various output-sensitive MCE algorithms whose processing time is proportional to the number of maximal cliques were also studied [4]. Algorithms for enumerating maximal cliques of size larger than a threshold were also studied [5]. All these algorithms are sequential in-memory algorithms, which do not scale well due to the high complexity of MCE. To process graphs that are too large to fit in main memory, I/O-efficient algorithms that recursively extract a core part of the input graph for local MCE computation were proposed [7], [8], [9]. I/O-efficient algorithms for enumerating triangles (i.e., smallest cliques) were proposed in [17], [18] and related application was given in [19]. Recently, several parallel or distributed algorithms were proposed [9], [10], [11], [12], which we have discussed in Section I. Apart from algorithms for computing the maximal cliques, a recent work proposed a concise summary of the set of maximal cliques, which ensures that for every maximal clique C, there is a good portion of C that is represented by some maximal clique in the summary [20]. This summary, however, is not a lossless representation of the set of maximal cliques.

VII. CONCLUSIONS

We studied efficient algorithms for computing and updating the set of maximal cliques. Existing parallel algorithms for computing maximal cliques are still immature and we showed that our parallel algorithm is orders of magnitude faster than the existing MapReduce algorithm for MCE [12]. Update maintenance for the set of maximal cliques on distributed sites has not been well studied in the past, but we verified that our algorithm is efficient for a range of real-world datasets from different domains.

REFERENCES

[1] C. Bron and J. Kerbosch, "Algorithm 457: finding all cliques of an undirected graph," *Commun. ACM*, vol. 16, no. 9, pp. 575–577, 1973.

[2] F. Cazals and C. Karande, "A note on the problem of reporting maximal cliques," *Theor. Comput. Sci.*, vol. 407, no. 1-3, pp. 564–568, 2008.

[3] D. Eppstein, M. Löffler, and D. Strash, "Listing all maximal cliques in sparse graphs in near-optimal time," in *ISAAC (1)*, 2010, pp. 403–414.

[4] K. Makino and T. Uno, "New algorithms for enumerating all maximal cliques," in *SWAT*, 2004, pp. 260–272.

[5] N. Modani and K. Dey, "Large maximal cliques enumeration in large sparse graphs," in *COMAD*, 2009.

[6] E. Tomita, A. Tanaka, and H. Takahashi, "The worst-case time complexity for generating all maximal cliques and computational experiments," *Theor. Comput. Sci.*, vol. 363, no. 1, pp. 28–42, 2006.

[7] J. Cheng, Y. Ke, A. W.-C. Fu, J. X. Yu, and L. Zhu, "Finding maximal cliques in massive networks by h*-graph," in *SIGMOD Conference*, 2010, pp. 447–458.

[8] ——, "Finding maximal cliques in massive networks," *ACM Trans. Database Syst.*, vol. 36, no. 4, p. 21, 2011.

[9] J. Cheng, L. Zhu, Y. Ke, and S. Chu, "Fast algorithms for maximal clique enumeration with limited memory," in *KDD*, 2012, pp. 1240–1248.

[10] N. Du, B. Wu, L. Xu, B. Wang, and P. Xin, "Parallel algorithm for enumerating maximal cliques in complex network," in *Mining Complex Data*, 2009, pp. 207–221.

[11] M. C. Schmidt, N. F. Samatova, K. Thomas, and B.-H. Park, "A scalable, parallel algorithm for maximal clique enumeration," *J. Parallel Distrib. Comput.*, vol. 69, no. 4, pp. 417–428, 2009.

[12] B. Wu, S. Yang, H. Zhao, and B. Wang, "A distributed algorithm to enumerate all maximal cliques in mapreduce," in *Proceedings of the International Conference on Frontier of Computer Science and Technology*, 2009, pp. 45–51.

[13] S. B. Seidman, "Network structure and minimum degree," *Social Networks*, vol. 5, no. 3, pp. 269–287, 1983.

[14] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *OSDI*, 2004, pp. 137–150.

[15] S. N. Dorogovtsev and J. F. F. Mendesand, "Evolution of networks: From biological nets to the internet and www," *Oxford University Press*, 2003.

[16] J. Moon and L. Moser, "On cliques in graphs," *Israel J. Math*, vol. 3(1), pp. 23–28, 1965.

[17] S. Chu and J. Cheng, "Triangle listing in massive networks and its applications," in *KDD*, 2011, pp. 672–680.

[18] ——, "Triangle listing in massive networks," *TKDD*.

[19] J. Wang and J. Cheng, "Truss decomposition in massive networks," *PVLDB*, vol. 5, no. 9, pp. 812–823, 2012.

[20] J. Wang, J. Cheng, and A. W.-C. Fu, "Redundancy-aware maximal cliques," in *KDD*, 2013, pp. 122–130.