Fast Algorithms for Maximal Clique Enumeration with Limited Memory

James Cheng
Nanyang Technological University
Singapore
j.cheng@acm.org

Linhong Zhu
Institute for Infocomm Research
Singapore
lzhu@i2r.a-star.edu.sg

Yiping Ke
Institute of High Performance Computing
Singapore
keyp@ihpc.a-star.edu.sg

Shumo Chu
Nanyang Technological University
Singapore
shumo.chu@acm.org

ABSTRACT

Maximal clique enumeration (*MCE*) is a long-standing problem in graph theory and has numerous important applications. Though extensively studied, most existing algorithms become impractical when the input graph is too large and is disk-resident. We first propose an efficient partition-based algorithm for MCE that addresses the problem of processing large graphs with limited memory. We then further reduce the high cost of CPU computation of MCE by a careful nested partition based on a cost model. Finally, we parallelize our algorithm to further reduce the overall running time. We verified the efficiency of our algorithms by experiments in large real-world graphs.

Categories and Subject Descriptors

H.2.8 [**DATABASE MANAGEMENT**]: Database Applications—*Data mining*; G.2.2 [**DISCRETE MATHEMATICS**]: Graph Theory—*Graph algorithms*

General Terms

Algorithms, Experimentation, Performance

Keywords

Maximal clique enumeration, I/O efficient, parallel algorithm, massive networks, sparse graphs

1. INTRODUCTION

Maximal clique enumeration (**MCE**) [3, 9] is one of the fundamental problems in graph theory. It is closely related to many other important graph problems, such as maximal independent sets (or minimal vertex covers), graph coloring, maximal common induced subgraphs, maximal common edge subgraphs, etc. Besides graph theory, MCE has numerous other applications, such as social network analysis [20], hierarchy detection through email networks [16], study of structures in behavioral and cognitive networks [5], statistical analysis of financial networks [7], clustering in dynamic networks [28], the detection of emergent patterns in terrorist networks [6], etc. Moreover, MCE is also closely tied to various applications in computational biology [1], including the detection of protein-protein interaction complex and clustering protein sequences.

The problem of MCE is NP-hard theoretically but because of its significance in real applications, many practical algorithms [1, 3, 9, 10, 18, 19, 21, 23, 24, 25, 26, 28, 29, 30, 31] have been proposed. However, many of these algorithms have become impractical today due to the fast growing size of graphs in the real world. For example, the Web graph has over 1 trillion webpages (by Google in 2008); most social networks (e.g., Facebook, LinkedIn, Twitter, Google+) and communication networks (e.g., MSN, phone, SMS, and email networks) have up to billions of users; other networks such as transportation networks, citation networks, stock-market networks, etc., are also massively large.

The existing algorithms are *in-memory* algorithms, which require space that is asymptotically linear in the size of the input graph. However, the massive size of many graphs has outpaced the advance in the memory available on commodity hardware. MCE computation accesses vertices in a rather arbitrary manner and this results in random access. When memory is insufficient and the graph has to be resident on disk, random disk access incurs high I/O cost.

To process such large graphs, Cheng et al. proposed efficient algorithms [12, 13] which enumerate maximal cliques in local subgraphs that fit in main memory as to eliminate the high I/O cost due to random access. However, MCE intrinsically has a high CPU time complexity and thus reducing the I/O cost alone does not fully address the problem.

To address the intensity of MCE computation, several parallel algorithms [17, 27] have been proposed. However, they still require a copy of the entire input graph to be resident in memory at each computing node, which ends at the same predicament faced by the existing in-memory sequential MCE algorithms.

MapReduce has been popularly used to handle massive data. A MapReduce algorithm [33] was proposed recently for MCE. The MapReduce model, however, may not be efficient for enumeration tasks such as MCE and triangle listing [14, 15] in large graphs due to the huge amount of intermediate data produced in the shuffling phase between Mappers and Reducers.

In this paper, we propose new algorithms for MCE that achieve the following three objectives.

1. We reduce the I/O cost by a partition-based strategy, which avoids random access for MCE in large graphs that cannot fit in memory.
2. We reduce the cost of CPU computation by investigating a neglected (by existing work) but non-negligible cost in MCE; that is, most of the operations at each step of MCE require only constant time but some set intersections require linear time. We devise a cost model to reduce this cost which significantly speeds up the entire MCE process.
3. We further reduce the overall running time by parallelizing MCE based on our partitioning strategy. Instead of requiring to keep the entire input graph at each computing node as in the existing parallel algorithms [17, 27], our algorithm requires only limited memory at any computing node.

Extensive experiments on large real-world datasets verify the effectiveness of our algorithms in reducing both the I/O cost and CPU cost. The results show that our algorithms significantly outperform the state-of-the-art MCE algorithms [12, 13, 18, 19, 29], especially in the case when the input graph is too large to fit in main memory.

Paper organization. The remaining of the paper is organized as follows. Section 2 defines the problem and basic notations. Section 3 describes a conventional algorithm for MCE and points out its weaknesses. Section 4 discusses the details of our algorithms. Section 5 reports the experimental results. Section 6 discusses the related work and Section 7 concludes the paper.

2. NOTATIONS AND PROBLEM DEFINITION

Let $G = (V, E)$ be a simple undirected graph. We define the *size* of G, denoted by $|G|$, as $|G| = (|V| + |E|)$. Given a subset of vertices $S \subseteq V$, we define the *induced subgraph* of G by S as $G_S = (V_S, E_S)$, where $V_S = S$ and $E_S = \{(u, v) : u, v \in S, (u, v) \in E\}$. We define the set of *adjacent vertices* of a vertex v in G as $adj(v) = \{u : (u, v) \in E\}$, and the *degree* of v in G as $deg(v) = |adj(v)|$. Similarly, we define $adj(v, G_S) = \{u : (u, v) \in E_S\}$ and $deg(v, G_S) = |adj(v, G_S)|$.

We assume that all graphs are stored (whether in memory or on disk) in adjacency list representation, where vertices are assigned unique IDs and ordered according to their IDs.

A *clique* C in a graph G is a subset of vertices, where $C \subseteq V$, such that G_C is a complete subgraph of G. C is called a *maximal clique* in G if there exists no clique C' in G such that $C' \supset C$.

Problem definition. The problem of **MCE** is: *given a graph G, find the set of all maximal cliques in G.*

We focus on *sparse graphs*, for which $|E| \ll B \cdot |V|$, and B is the disk block size. Most large and many fast growing real-world networks are sparse. For processing these large sparse graphs, random disk access in the process of MCE is prohibitively expensive since data transferred from/to disk is in blocks, while the exact amount of data used in each step of MCE is much less than the amount of data transferred for each random access, since $deg(v) \ll B$ for most $v \in V$.

Algorithm 1 *IM-MCE*

Input: a graph $G = (V, E)$
Output: all maximal cliques in G

```
1. IM-MCE-Step($\emptyset$, $V$, $\emptyset$);
```

Procedure 2 *IM-MCE-Step*$(C, todo, done)$

```
1. if($todo = \emptyset$ and $done = \emptyset$)
2.     output $C$ as a maximal clique;
3. else
4.     choose a pivot vertex $v_p$ from $(todo \cup done)$;
5.     $doing := todo \backslash adj(v_p)$;
6.     for each $v \in doing$ do
7.         $todo := todo \backslash \{v\}$;
8.         IM-MCE-Step($C \cup \{v\}$, $todo \cap adj(v)$, $done \cap adj(v)$);
9.         $done := done \cup \{v\}$;
```

3. CONVENTIONAL MCE ALGORITHM

We first discuss the conventional algorithms for MCE and their shortcomings. Algorithm 1 and Procedure 2 sketch a general algorithm framework for MCE, which is adopted by most existing MCE algorithms [9, 23, 29]. The state-of-the-art parallel MCE algorithm [27] is also developed based on this framework. This framework is also used in later sections to explain our algorithms.

Algorithm 1 calls Procedure 2. In Procedure 2, we use the notations ***todo***, ***doing***, and ***done*** to represent the set of vertices *to be processed*, *being processed*, and *already processed*, respectively.

The algorithm starts from the set of all vertices in G, i.e., $todo = V$, and $done = \emptyset$, by calling Procedure 2. Procedure 2 recursively calls itself to grow a clique C in a depth-first manner until C becomes maximal; that is, when $todo = \emptyset$, i.e., there is no more vertex for C to grow with, and $done = \emptyset$, i.e., C has not been enumerated before (if $done \neq \emptyset$, then $\exists C' \supset C$ such that C' has been found as a maximal clique).

The depth-first MCE process essentially constructs a search tree, or more precisely a search forest joined by a (virtual) root. At each call of Procedure 2, we first pick a *pivot* vertex v_p, which is used to prune all the neighbors of v_p, because any maximal clique containing a vertex $u \in adj(v_p)$ can be enumerated from either v_p or another vertex $v \in adj(u)$, where $v \notin adj(v_p)$. Therefore, all the search subtrees rooted at each $u \in adj(v_p)$ can be pruned.

We then construct $doing$, which is the set of vertices being processed in the current call of Procedure 2. For each vertex $v \in doing$, we grow the current clique C by adding v to C. To further grow $(C \cup \{v\})$ to a bigger clique in the subsequent recursive call, we need to make sure that every vertex in $todo$ must be a neighbor of v. After the recursive call returns, we add v to $done$ to indicate that v has been processed in the current search subtree.

Shortcomings. Algorithm 1 requires $O(3^{|V|/3})$ time in the worst case, but has shown to be the optimal worst-case complexity [29]. In practice, the existing algorithms that adopt the framework of Algorithm 1 are reasonably fast for processing relatively small real-world graphs. However, these algorithms are in-memory algorithms and require $\Omega(|V| + |E|)$ memory space. If memory is not sufficient or the input graph is disk-resident, the process immediately becomes impractical. As we observe in Line 8 of Procedure 2, the MCE process requires accesses to the neighbor set of each vertex $v \in doing$, i.e., $adj(v)$, which may scatter in different locations in the graph resident on disk. Thus, random disk access leads to huge I/O cost and severely degrades the performance of the in-memory algorithm. In addition, some branches of the search tree constructed by Algorithm 1 can be potentially parallelized. Intuitively, the search subtrees rooted at each of the siblings of the search tree can be constructed in parallel with careful design. We address these issues in Section 4.

4. FASTER MCE ALGORITHMS

In this section, we address the shortcomings of the conventional algorithm for MCE. In particular, we have the following three main objectives: *(1) reducing the I/O cost, (2) reducing the CPU cost, and (3) further reducing the overall running time by parallelization.*

4.1 Reducing the I/O Cost

When the input graph cannot fit in main memory, reducing the I/O cost by avoiding random disk accesses becomes the key to the efficiency of MCE. The main idea of our algorithm is to repeatedly extract a subgraph that fits in memory and compute the maximal cliques locally from the subgraph. However, we should also preserve both the *global* correctness and completeness of the results computed from the *local* subgraphs.

We first define the subgraph to be extracted for MCE as follows.

Definition 1 (SEED VERTICES/SUBGRAPH). *Given a graph $G = (V, E)$, denote S to be a set of* ***seed vertices*** *selected from V, where $S \subseteq V$. The* ***seed subgraph*** *of G, denoted by $G_S = (V_S, E_S)$, is defined as the induced subgraph of G by S.*

Using the seed subgraph G_S alone is not sufficient to ensure the completeness of MCE since some seed vertices may form a clique with vertices not in G_S. To address this, we extend G_S as follows.

Definition 2 (EXTENDED VERTICES/SUBGRAPH). *Given a set of seed vertices S of a graph $G = (V, E)$, the set of* ***extended seed vertices****, or simply* ***extended vertices****, denoted by S^+, is defined as $S^+ = S \cup \{v : v \in adj(u), u \in S\}$. The* ***extended seed subgraph****, or simply* ***extended subgraph****, denoted by G_{S^+}, is defined as the induced subgraph of G by S^+.*

Note that the above definition of subgraph is different from the one used in [12, 13], where they do not include the edges among the vertices in $(S^+ \backslash S)$. This difference renders the design of the algorithm totally different, since the subgraphs in [12, 13] still need to access the input graph for MCE while ours do not. Compared with the algorithms in [12, 13], our algorithm is simpler, more efficient, and natural to be parallelized.

The following example illustrates the concepts.

Example 1. Consider the graph G shown in Figure 1. If we choose $S = \{0, 1, 2\}$ as the set of seed vertices, we obtain the seed subgraph G_S as shown in Figure 2 (left). It is easy to see that the seed subgraph alone cannot produce any maximal clique of G. So we extend the seed vertices to be $S^+ = \{0, 1, 2, 3, 4, 5, 7\}$ and obtain the extended subgraph G_{S^+}, as given in Figure 2 (right). The extended subgraph makes possible the local enumeration of global maximal cliques, i.e., $\{0, 1, 2, 3\}, \{1, 4, 5\}, \{2, 7\}$, since G_{S^+} captures fully the neighborhood information of seed vertices. □

With the above formulation of extended subgraph, we first prove that the maximal cliques computed locally in each extended subgraph are also globally maximal.

Let $\mathcal{M}(G)$ and $\mathcal{M}(G_{S^+})$ be the set of maximal cliques in G and G_{S^+}, respectively. Let $\mathcal{M}_S(G) = \{C : C \in \mathcal{M}(G), C \cap S \neq \emptyset\}$ and $\mathcal{M}_S(G_{S^+}) = \{C : C \in \mathcal{M}(G_{S^+}), C \cap S \neq \emptyset\}$ be the set of maximal cliques, respectively in G and G_{S^+}, that contain at least one vertex in S.

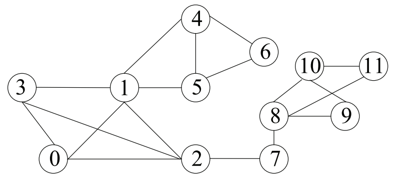

Figure 1: An Example of Input Graph, G

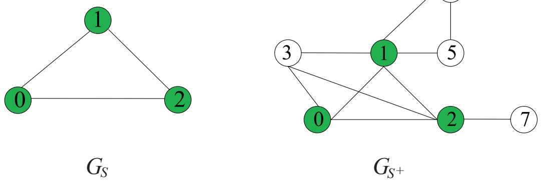

Figure 2: Seed Subgraph and Extended Subgraph

LEMMA 1. $\mathcal{M}_S(G_{S^+}) = \mathcal{M}_S(G)$.

PROOF. We first prove $\mathcal{M}_S(G_{S^+}) \subseteq \mathcal{M}_S(G)$. We begin by proving that $\forall C \in \mathcal{M}_S(G_{S^+})$, $C \in \mathcal{M}(G)$. Suppose $C \notin \mathcal{M}(G)$, then $\exists C' \in \mathcal{M}(G)$ such that $C' \supset C$. Then $\exists v \in C'$ and $v \notin S^+$ (otherwise $C' \in \mathcal{M}_S(G_{S^+})$ and hence $C \notin \mathcal{M}_S(G_{S^+})$). However, $v \notin S^+$ implies that $(v, u) \notin E$ for some vertex $u \in (C \cap S)$, which contradicts that C' is a clique. Thus, $C \in \mathcal{M}(G)$ and it follows that $C \in \mathcal{M}_S(G)$ since $C \cap S \neq \emptyset$.

We then prove $\mathcal{M}_S(G) \subseteq \mathcal{M}_S(G_{S^+})$. $\forall C \in \mathcal{M}_S(G)$, we have $C \in \mathcal{M}(G)$ and $C \cap S \neq \emptyset$. Let $u \in (C \cap S)$. Then $\forall v \in C \backslash \{u\}$, we have $(u, v) \in E$ and thus $v \in S^+$. Therefore, $C \in \mathcal{M}(G_{S^+})$ and it follows that $C \in \mathcal{M}_S(G_{S^+})$. □

Lemma 1 implies that all maximal cliques in G that contain at least one seed vertex in S can be computed from G_{S^+} alone. This leads to the design of an algorithm that repeatedly extracts a subgraph G_{S^+} that can fit in main memory, computes $\mathcal{M}_S(G_{S^+})$ from G_{S^+} in memory, and then continues to select another set of seed vertices from $(V \backslash S)$, until all vertices in V are processed. The following theorem guarantees the correctness and completeness of the set of maximal cliques so computed.

THEOREM 1. *Let $\mathcal{S} = \{S_1, \ldots, S_k\}$, where $\bigcup_{1 \leq i \leq k} S_i = V$ and $S_i \cap S_j = \emptyset$ for $i \neq j$. Let $\mathcal{M}_{\mathcal{S}} = \bigcup_{1 \leq i \leq k} \mathcal{M}_{S_i}(G_{S_i^+})$. Then, $\mathcal{M}_{\mathcal{S}} = \mathcal{M}(G)$.*

PROOF. First, $\forall S_i$, $\mathcal{M}_{S_i}(G_{S_i^+}) \subseteq \mathcal{M}(G)$ according to Lemma 1. Thus, $\mathcal{M}_{\mathcal{S}} \subseteq \mathcal{M}(G)$. Next, $\forall C \in \mathcal{M}(G)$, $\exists S_i$ such that $C \in \mathcal{M}_{S_i}(G_{S_i^+})$, where $C \cap S_i \neq \emptyset$ (note that the existence of S_i is because $\bigcup_{1 \leq i \leq k} S_i = V$). Thus, $\mathcal{M}(G) \subseteq \mathcal{M}_{\mathcal{S}}$. □

Theorem 1 immediately leads to the design of an efficient partition-based algorithm, as shown in Algorithm 3.

The algorithm sequentially scans the input graph G to select a set of seed vertices S, extracts G_{S^+} by another scan of G, computes $\mathcal{M}_S(G_{S^+})$ from G_{S^+}, and then continues to select another set of seed vertices until all vertices in V are processed as seed vertices.

The algorithm determines the size of G_{S^+} in Line 4, where a parameter ϕ_{deg} is used to estimate the size of G_{S^+}, and $0 < c < 1$

Algorithm 3 *Partition-Based MCE*

Input: a graph $G = (V, E)$
Output: all maximal cliques in G

```
1. $S := \emptyset, S^+ := \emptyset$;
2. for each $v \in V$ do
3.     $S := (S \cup \{v\}), S^+ := (S^+ \cup \{v\} \cup adj(v))$;
4.     if$((\phi_{deg} \cdot |S^+|) \geq cM)$
5.         scan $G$ once to extract $G_{S^+}$;
6.         apply in-memory MCE to $G_{S^+}$;
7.         for each $C \in \mathcal{M}(G_{S^+})$ do
8.             if$((C \cap S) \neq \emptyset$ and $(\nexists u \in C$ s.t. $u \prec S))$
9.                 output $C$ as a maximal clique;
10.        $S := \emptyset, S^+ := \emptyset$;
```

is to ensure the estimated size is less than M[1], where M is the available memory size. Here, ϕ_{deg} can be the average or maximum vertex degree in G. If ϕ_{deg} is the average vertex degree, it is possible that $|G_{S^+}| > M$; however, this can be easily fixed by further splitting G_{S^+} into smaller extended subgraphs with smaller seed vertex sets. Since only some selection of S can lead to the case "$|G_{S^+}| > M$", in practice the total number of extended subgraphs to be extracted remains relatively stable. If ϕ_{deg} is the maximum vertex degree, we can guarantee that $|G_{S^+}| \leq M$. However, this may lead to under-utilization of the available memory. This is also true for splitting a large G_{S^+} into smaller extended subgraphs in the case of setting ϕ_{deg} as the average vertex degree. However, we shall show in Section 4.2 that fully utilizing the available memory may actually lead to an even higher cost in the in-memory computation of MCE.

After computing the set of local maximal cliques in G_{S^+}, i.e., $\mathcal{M}(G_{S^+})$, the algorithm outputs only those cliques containing at least one seed vertex to ensure the correctness according to Lemma 1. However, it is possible that a maximal clique C is enumerated in more than one G_{S^+}, if the vertices in C are in different sets of seed vertices, i.e., $\exists u, v \in C$ such that $u \in S_i$ and $v \in S_j$, where $u \neq v$ and $i \neq j$. To avoid duplicate output of a maximal clique, Line 8 sets another condition, ($\nexists u \in C$ s.t. $u \prec S$). Here, $u \prec S$ means that u is selected as a seed vertex (in an earlier iteration of Lines 5-10), before any vertex in S is selected (in the current iteration). Thus, Lines 7-9 do not output a maximal clique C if C contains a vertex that has already been selected as a seed vertex before S is selected as the seed vertex set.

Greater Pruning. The condition "($\nexists u \in C$ s.t. $u \prec S$)" also implies that we do not need to include into G_{S^+} any edge (u, w), $\forall u, w \in S^+$ and $u, w \prec S$. Furthermore, we can push the condition into the in-memory MCE process to achieve greater pruning as follows. Recall that the search tree (implicitly) constructed by Algorithm 1 is a prefix tree. During the MCE process, we can process the seed vertices in S before the vertices in $S^+ \backslash S$. Then we do not grow the current clique C by any vertex $v \prec S$ in Line 6 of Procedure 2. In this way, we avoid duplicate enumeration of maximal cliques as well as achieve greater pruning effect for MCE.

Complexity Analysis. The following theorem gives the complexity of Algorithm 3. We use the following standard I/O complexity notations [2] in the complexity analysis: M is the main memory size, B is the disk block size, $scan(N) = \Theta(N/B)$ I/Os, where $1 \ll B \leq M/2$ and N is the amount of data being read/written from/to disk.

[1]We assume that if $|S| = 1$, then $|G_{S^+}| \leq M$. In the case of a high-degree vertex v, $|G_{S^+}|$ with $S = \{v\}$ can be large. For sparse real-world graphs, we may effectively reduce $|G_{S^+}|$ by sorting the vertex set V in ascending order of the vertex degree, since we actually do not need to include into G_{S^+} any edge (u, w) if u and w are ordered before v, for all $u, w \in S^+$ (as implied by Line 8 of Algorithm 3).

THEOREM 2. *Algorithm 3 requires $O(k \cdot scan(|G|))$ I/Os, $O(kT)$ CPU time, and $O(M)$ memory space, where $k = min\{\frac{|V|(\phi_{deg})^2}{M}, |V|\}$, and T is the CPU time complexity of the in-memory algorithm for computing MCE in an extended subgraph G_{S^+} with $|G_{S^+}| \leq M$.*

PROOF. Algorithm 3 makes $(k+1)$ scans of G, once in Line 1 and k times in Line 5, where k is the total number of extended subgraphs G_{S^+} extracted. From Line 5 we have $(\phi_{deg} \cdot |S^+|) \geq cM$, but we can easily move the last vertex in S to the next round of seed vertex selection to obtain $|G_{S^+}| \leq (\phi_{deg} \cdot |S^+|) = O(M)$. Thus, we have $((\phi_{deg})^2 \cdot |S|) = O(M)$ since $|S^+| \simeq (\phi_{deg} \cdot |S|)$. Since V is divided into k sets of seed vertices, we have $k = O(\frac{|V|}{|S|}) = O(\frac{|V|(\phi_{deg})^2}{M})$. In the worst case, each S contains only one vertex and thus $k \leq |V|$.

The selection of seed vertices and extraction of extended subgraphs require only linear time; thus, the CPU time is bounded by the number of times the in-memory algorithm is invoked to compute MCE on an extended subgraph G_{S^+}, where the maximum size of any G_{S^+} is M (we refer the readers to [29] for the details on the CPU time complexity analysis on the in-memory algorithm). The memory space requirement is $|G_{S^+}| = O(M)$. □

4.2 Reducing the CPU Cost

In processing large graphs that cannot fit in main memory, the bottleneck of the in-memory algorithms is the huge I/O cost due to random disk access. However, by processing these large graphs using Algorithm 3, the bottleneck may no longer be the I/O cost, but rather the cost of the in-memory computation of MCE in each extracted extended subgraph.

4.2.1 A Neglected but Non-Negligible Cost

The cost of in-memory MCE computation can be broken down into two main types: *the cost of constructing the search tree* and *the cost of set intersection*.

Conventional algorithms focus on reducing the cost of constructing the search tree by employing various pruning techniques to reduce the size of search tree. Although these pruning techniques are vital for efficient MCE, we have observed that we can further reduce the cost of CPU computation as follows.

Among all individual operations in the process of MCE, the most expensive one is the set intersection: two in Line 8 of Procedure 2 (and the set subtraction in Line 5 may also be considered as one). These set intersections can be significantly more costly to process than the other operations that require only constant time.

There are approximation algorithms for set intersection, but for MCE we require an exact answer. To compute $X \cap Y = Z$, if exact set intersection is required, then the cost of the intersection is at least $(|X| + |Y|)$. However, if $|Z|$ is significantly smaller than $|X|$ and $|Y|$, then the majority of the processing is "wasted" on processing *non-contributing* elements in X and Y, i.e., elements in $X \backslash Z$ and $Y \backslash Z$. Thus, the objective here is to reduce the number of non-contributing elements in X and Y.

Reducing the number of non-contributing elements in X and Y in Procedure 2 means removing non-contributing elements in $todo$, $done$, and in particular $adj(v)$ since $adj(v)$ is often significantly larger than $todo$ and $done$ (except in the first call of Procedure 2).

4.2.2 *Cost Reduction of Set Intersection*

To reduce the portion of the non-contributing elements in $todo$, $done$, and $adj(v)$, we transform the search space from G to G_{S^+}. In this way, the search space is immediately reduced from V to S^+ for $todo$ and $done$, and from $adj(v, G)$ to $adj(v, G_{S^+})$. However, which S^+ and hence G_{S^+} should we select in order to minimize the cost?

To avoid expensive random disk access, G_{S^+} should have size smaller than M. For this purpose, Algorithm 3 naturally fulfills the requirement. However, Algorithm 3 attempts to fully utilize the available memory, i.e., $|G_{S^+}|$ is close to M. But the analysis in Section 4.2.1 suggests that a smaller G_{S^+} gives a smaller cost of set intersection.

Extracting a smaller G_{S^+} in Algorithm 3 can be easily done by setting a smaller c in Line 4 of Algorithm 3. However, doing so proportionally increases the number of scans of G in Line 5 and hence the overall I/O cost. The following lemma enables us to totally avoid any extra I/O in extracting smaller G_{S^+}.

LEMMA 2. *Given two sets of seed vertices, S_{big} and S_{small}, if $S_{small} \subseteq S_{big}$, then $G_{S^+_{small}}$ is a subgraph of $G_{S^+_{big}}$.*

PROOF. Since $S_{small} \subseteq S_{big}$, by Definition 2, $S^+_{small} \subseteq S^+_{big}$. Then we also have $E_{S^+_{small}} = \{(u, v) : u, v \in S^+_{small}, (u, v) \in E\} \subseteq E_{S^+_{big}} = \{(u, v) : u, v \in S^+_{big}, (u, v) \in E\}$. Thus, $G_{S^+_{small}}$ is a subgraph of $G_{S^+_{big}}$. $\square$

Lemma 2 shows that it is possible to extract a smaller extended subgraph $G_{S^+_{small}}$ from another bigger extended subgraph $G_{S^+_{big}}$ (which is resident in memory) instead of from the original input graph G (which is resident on disk). However, we still need to determine how small this $G_{S^+_{small}}$ should be.

4.2.3 *A Cost Model*

We propose a cost model for choosing the right size for the extended subgraph. Suppose that we now have a set of extended subgraphs $\mathcal{G}_\beta$ resident in main memory for MCE computation. Each subgraph $G_{S^+_\beta}$ in $\mathcal{G}_\beta$ is roughly of the same size. Initially, $\mathcal{G}_\beta$ contains only one extended subgraph G_{S^+} extracted in Lines 4-5 of Algorithm 3. We determine (iteratively) whether extracting smaller extended subgraphs from $\mathcal{G}_\beta$ can achieve better efficiency for in-memory MCE computation.

For each $G_{S^+_\beta} \in \mathcal{G}_\beta$, we consider to split $G_{S^+_\beta}$ into a set of smaller extended subgraphs $\{G_{S^+_\alpha}\}$, each of them is roughly of the same size. Let $\mathcal{G}_\alpha$ be the set of all such smaller extended subgraphs extracted from $\mathcal{G}_\beta$.

First, having a smaller $adj(v, G_{S^+_\alpha})$, instead of $adj(v, G_{S^+_\beta})$, in the process of MCE leads to a lower cost of set intersection. The *gain* obtained by preferring $\mathcal{G}_\alpha$ over $\mathcal{G}_\beta$ can be estimated as follows.

$$\begin{aligned} Gain &= \sum_{G_{S^+_\beta} \in \mathcal{G}_\beta} |T_{S^+_\beta}| \cdot \overline{deg}(v, G_{S^+_\beta}) \\ &\quad - \sum_{G_{S^+_\alpha} \in \mathcal{G}_\alpha} |T_{S^+_\alpha}| \cdot \overline{deg}(v, G_{S^+_\alpha}) \\ &= \sum_{G_{S^+_\beta} \in \mathcal{G}_\beta} |T_{S^+_\beta}| \cdot (|E_{S^+_\beta}|/|S^+_\beta|) \\ &\quad - \sum_{G_{S^+_\alpha} \in \mathcal{G}_\alpha} |T_{S^+_\alpha}| \cdot (|E_{S^+_\alpha}|/|S^+_\alpha|), \qquad (1) \end{aligned}$$

where $|T_{S^+_\alpha}|$ is the number of nodes in the search tree $T_{S^+_\alpha}$ constructed by Algorithm 1 with $G_{S^+_\alpha}$ as the input, and $\overline{deg}(v, G_{S^+_\alpha})$ is the average degree of a vertex v in $G_{S^+_\alpha}$; and similarly for $|T_{S^+_\beta}|$ and $\overline{deg}(v, G_{S^+_\beta})$.

In Equation (1), the gain obtained from the reduction in the cost of set intersection with a smaller $adj(v, G_{S^+_\alpha})$ instead of $adj(v, G_{S^+_\beta})$ is reflected by the average degree $\overline{deg}(v, G_{S^+_\alpha})$ and $\overline{deg}(v, G_{S^+_\beta})$. And we multiply the average degree by the number of nodes in the search tree because at each node in the search tree, we need to perform the set intersection as shown in Line 8 of Procedure 2.

If choosing $\mathcal{G}_\alpha$ instead of $\mathcal{G}_\beta$ only results in a gain for MCE, then the optimal choice is by setting $S_\alpha = \{v\}$ for each single vertex $v \in V$. However, preferring $\mathcal{G}_\alpha$ over $\mathcal{G}_\beta$ may also result in some loss due to the extraction of a large number of smaller extended subgraphs and the construction of their search trees during the process of MCE.

First, if we further split $G_{S^+_\beta}$ into a set of smaller $G_{S^+_\alpha}$, we have extra extraction costs. Second, the size of the search tree constructed from $G_{S^+_\beta}$ is smaller than the total size of the search trees constructed from the set $\{G_{S^+_\alpha}\}$ extracted from $G_{S^+_\beta}$. The *loss* of preferring $\mathcal{G}_\alpha$ over $\mathcal{G}_\beta$ is given as follows.

$$\begin{aligned} Loss &= \Big(\sum_{G_{S^+_\beta} \in \mathcal{G}_\beta} \sum_{G_{S^+_\alpha} \subset G_{S^+_\beta}} (|G_{S^+_\alpha}| + |G_{S^+_\beta}|) \Big) \\ &\quad + \Big(\sum_{G_{S^+_\alpha} \in \mathcal{G}_\alpha} |T_{S^+_\alpha}| - \sum_{G_{S^+_\beta} \in \mathcal{G}_\beta} |T_{S^+_\beta}| \Big). \qquad (2) \end{aligned}$$

In Equation (2), the first half gives the cost of extracting all $G_{S^+_\alpha}$ from each $G_{S^+_\beta} \in \mathcal{G}_\beta$. Since we scan each $G_{S^+_\beta}$ at most once to extract each $G_{S^+_\alpha}$, we add $|G_{S^+_\beta}|$ to the total cost. The second half is the difference in the total size of the search trees constructed from all $G_{S^+_\alpha}$ and from all $G_{S^+_\beta}$.

Equation (1) and Equation (2) represent a tradeoff: we trade off *(1) the cost of set intersection* with *(2) the cost of subgraph extraction and that of search tree construction*; that is, using smaller extended subgraphs reduces (1) but increases (2), while using larger extended subgraphs results in the opposite. We quantify this tradeoff by the following cost model.

$$TotalGain = Gain - Loss. \qquad (3)$$

Finally, in Equation (1) and Equation (2), $|G_{S^+_\alpha}|$ and $|G_{S^+_\beta}|$ are known after we extract the subgraphs, while $|T_{S^+_\alpha}|$ and $|T_{S^+_\beta}|$ can be estimated by a variation of Knuth's method [22] for estimating the size of a backtracking tree of MCE proposed in [12].

4.2.4 *An Improved Algorithm for MCE*

With the results of the previous subsections, we propose an improved partition-based algorithm for MCE, as shown in Algorithm 4. We name this algorithm as **SeqMCE** (for **Seq**uential **MCE**), to distinguish from the parallel algorithm proposed in Section 4.3.

As shown in Algorithm 4 and Procedure 5, the extraction of ex-

Algorithm 4 *SeqMCE*

```
Input: a graph $G = (V, E)$
Output: all maximal cliques in $G$

Replace Lines 6-9 of Algorithm 3 by: MCE-Step($G_{S^+}$);
```

Procedure 5 *MCE-Step*$(H = (V_H, E_H))$

```
1. $\mathcal{G}_\alpha := \{H\}$;
2. $TotalGain := \infty$;
3. while($TotalGain > 0$)
4.     $\mathcal{G}_\beta := \mathcal{G}_\alpha, \mathcal{G}_\alpha := \emptyset$;
5.     for each $G_{S_\beta^+} \in \mathcal{G}_\beta$ do
6.         divide $S_\beta$ into two equal halves: $S_{\alpha 1}$ and $S_{\alpha 2}$;
7.         extract $G_{S_{\alpha 1}^+}$ and $G_{S_{\alpha 2}^+}$ from $G_{S_\beta^+}$;
8.         $\mathcal{G}_\alpha := \mathcal{G}_\alpha \cup \{G_{S_{\alpha 1}^+}, G_{S_{\alpha 2}^+}\}$;
9.     compute $TotalGain$ from $\mathcal{G}_\alpha$ and $\mathcal{G}_\beta$;
10. $\mathcal{G}_\alpha := \mathcal{G}_\beta$;
11. for each $G_{S_\alpha^+} \in \mathcal{G}_\alpha$ do
12.     apply in-memory MCE to $G_{S_\alpha^+}$;
```

tended subgraphs is processed at two levels, one aiming to minimize the I/O cost while the other aiming to reduce the CPU cost.

The first level is to extract bigger extended subgraphs, which is done in a similar way as in Algorithm 3 but by replacing Lines 6-9 with a call to Procedure 5. At this level, the size of each G_{S+} should be close to M to minimize the number of scans of G and hence the I/O cost.

Then, at the second level as shown in Procedure 5, we extract smaller extended subgraphs from each bigger extended subgraph H obtained at the first level. Whether or not to further extract smaller extended subgraphs is determined by the cost model given by Equation (3).

Computing the optimal $\mathcal{G}_\alpha$ to maximize $TotalGain$ is similar to finding the optimal graph partition from H, which is APX-hard [4]. Thus, this process alone would dominate the overall cost of MCE. However, if we do not select the seed vertices from H randomly but rather select them sequentially, the selection process can be made much more efficient.

Another challenge of applying Equation (3) is that Equation (3) consists of Equations (1) and (2), both of which require the entire sets of extended subgraphs, $\mathcal{G}_\alpha$ and $\mathcal{G}_\beta$, to be known. Even if we select the seed vertices sequentially, there are still exponentially many different permutations of $\mathcal{G}_\alpha$ and $\mathcal{G}_\beta$.

To avoid the costly intermediate process of subgraph extraction, we consider the search space as a binary tree, where the set of nodes at each level of the tree is a set of extended subgraphs. We construct the binary tree level-wise (Lines 1-10). The first level consists of only one extended subgraph, i.e., H itself. The two children of any internal node in the tree are constructed by dividing the set of seed vertices of the parent sequentially into two sets of seed vertices with equal size, and then extract the corresponding extended subgraphs from the parent (the correctness of the extraction is guaranteed by Lemma 2). Then at each level, we compute $TotalGain$. We stop constructing a new level of the binary tree when there is no longer a gain. Then (in Lines 10-12), we take the corresponding $\mathcal{G}_\alpha$ as the set of extended subgraphs, and process MCE in each $G_{S_\alpha^+} \in \mathcal{G}_\alpha$ as is done before in Algorithm 3.

The above scheme for subgraph extraction, though heuristic, is effective and efficient for the following important reasons: first, it fits seamlessly into the design of the partition-based algorithm; second, it supports effective parallelization; lastly, it avoids extra overhead in re-assigning the vertex order or relabeling the vertices,

Algorithm 6 *ParMCE*

```
Input: a graph $G = (V, E)$
Output: all maximal cliques in $G$

1. $S := \emptyset, S^+ := \emptyset$;
2. for each $v \in V$ do
3.     $S := (S \cup \{v\}), S^+ := (S^+ \cup \{v\} \cup adj(v))$;
4.     if(($\phi_{deg} \cdot |S^+|) \geq cM$)
5.         scan $G$ once to extract $G_{S^+}$;
6.         while(no idle computing node)   wait;
7.         distribute the task MCE-Job($G_{S^+}$)
               to any idle computing node;
8.         $S := \emptyset, S^+ := \emptyset$;
```

Procedure 7 *MCE-Job*$(H = (V_H, E_H))$

```
1. $\mathcal{G}_\alpha := \{H\}$;
2. $TotalGain := \infty$;
3. while(($TotalGain > 0$) or
          (# of idle computing nodes $> |\mathcal{G}_\beta|$))
4.     $\mathcal{G}_\beta := \mathcal{G}_\alpha, \mathcal{G}_\alpha := \emptyset$;
5.     for each $G_{S_\beta^+} \in \mathcal{G}_\beta$ do
6.         divide $S_\beta$ into two equal halves: $S_{\alpha 1}$ and $S_{\alpha 2}$;
7.         extract $G_{S_{\alpha 1}^+}$ and $G_{S_{\alpha 2}^+}$ from $G_{S_\beta^+}$;
8.         $\mathcal{G}_\alpha := \mathcal{G}_\alpha \cup \{G_{S_{\alpha 1}^+}, G_{S_{\alpha 2}^+}\}$;
9.     compute $TotalGain$ from $\mathcal{G}_\alpha$ and $\mathcal{G}_\beta$;
10. $\mathcal{G}_\alpha := \mathcal{G}_\beta$;
11. for each $G_{S_\alpha^+} \in \mathcal{G}_\alpha$ do
12.     while(no idle computing node)   wait;
13.     distribute $G_{S_\alpha^+}$ to an idle computing node, $X$, and
        compute in-memory MCE in $G_{S_\alpha^+}$, further parallelizing
        the task among multiple cores or threads within $X$,
        or any other idle computing node(s), if any;
```

which is necessary for the task of MCE when vertices are selected out of order into $G_{S_\alpha^+}$.

4.3 Parallel Maximal Clique Enumeration

Existing parallel algorithms for MCE [17, 27] are in-memory algorithms that require the entire graph to be resident in main memory of each computing node, and thus are not scalable as the graph size increases and main memory is not sufficient to hold the graph.

We adopt our partition-based MCE framework and propose a parallel algorithm for MCE that requires only limited memory at each computing node. We describe the algorithm, named as **ParMCE** (for **Par**allel **MCE**), in Algorithm 6 and Procedure 7.

Algorithm 6 uses one computing node as the *master node*. The master node reads the input graph G from disk, extracts extended subgraphs from G, and distributes them to the idle nodes, if any. To utilize parallelism (e.g., multi-cores, hyper threading, etc.) within the master node, we create two threads in the master node. One thread extracts extended subgraphs and pushes it into the *pool* of un-computed subgraphs, and the other one takes the subgraphs from the *pool* and sends them to any available idle nodes. In case that the total number of available computing nodes becomes too large so that the computation at the master node might become a bottleneck, our algorithm easily allows us to split the input graph and create multiple master nodes to serve the increasing number of computing nodes.

Procedure 7 then describes that when a computing node receives an extended subgraph H (distributed by Algorithm 6), it further extracts smaller extended subgraphs to improve the efficiency of MCE, as done in Procedure 5. If we have extra idle computing nodes, we may further divide extended subgraphs into smaller ones to maximize parallelization (note that the bottleneck is at the MCE computation rather than at the subgraph extraction, both of which are now performed in-memory). This task itself can also be easily parallelized, by simply distributing the extended subgraphs in $\mathcal{G}_\beta$ to new computing nodes and call Procedure 7. But since this process is not the bottleneck, it should be parallelized only when computing nodes are excessive.

After we obtain the set of extended subgraphs $\mathcal{G}_\alpha$, we distribute each small extended subgraph $G_{S_\alpha^+} \in \mathcal{G}_\alpha$ to an idle computing node, X. Then at X, we apply the in-memory algorithm to process MCE in $G_{S_\alpha^+}$. The fine-grained parallelization of MCE proposed in [27] can be further applied if X has multiple cores or threads, or if there are more idle computing nodes available.

5. EXPERIMENTAL EVALUATION

In this section, we evaluate the performance of our algorithms. We compared with the algorithm that has the optimal time complexity for MCE in memory proposed by Tomita et al. [29] (denoted by *TomitaTT*), a recent algorithm for MCE in d-degenerate graphs proposed by Eppstein et al. [18, 19] (denoted by *EppsteinLS*), and the I/O-efficient MCE algorithm by Cheng et al. [13] (denoted by *ChengKFYZ*).

For all the sequential programs, we ran the experiments on a machine with an Intel Xeon 2.67GHz CPU and 4GB RAM, running CentOS 5.4 (Linux). For the parallel program (using the MPI), we ran the experiments on a cluster, with each computing nodes having 2.93GHz CPU and 4GB RAM.

Datasets. We use the following four datasets: `blog`, `LJ`, `Web`, and `BTC`. The `blog` network is collected from the top-15 popular queries published by Technorati (technorati.com) every three hours from Nov 2006 to Mar 2008. For each query, the top-50 results are retrieved. In the `blog` network, vertices are blogs and edges indicate that two blogs appear in the search result of the same query. `LJ` is the free online community called *LiveJournal*, where vertices are members and edges represent friendship between members. The `LJ` dataset is available from snap.stanford.edu. The `Web` graph is obtained from the YAHOO webspam dataset (barcelona.research.yahoo.net/webspam), where vertices are webpages and edges are hyperlinks. The `BTC` dataset is a semantic graph converted from the *Billion Triple Challenge 2009 RDF* dataset (http://vmlion25.deri.ie), where each vertex represents an object such as a person, a document, and an event, and each edge represents the relationship between two vertices such as "as-author", "links-to", and "has-title". We list some details of the datasets (number of vertices and edges, physical storage size) in Table 1.

Table 1: Datasets (M = 1,000,000)

	`blog`	`LJ`	`Web`	`BTC`
$\|V\|$	1M	4.8M	52.9M	165M
$\|E\|$	6.5M	43M	274.8M	773M
Storage size	186MB	1310MB	5GB	14.5GB

Among the four datasets, `blog` and `LJ` are two relatively smaller graphs, while `Web` and `BTC` are two larger graphs. We use the smaller graphs to compare the performance of our algorithm with the in-memory algorithms, while we use the larger graphs to evaluate the performance of our algorithms when memory of a single machine is insufficient to hold the input graph.

5.1 Effectiveness of Cost Model

We first examine whether the reduction of set intersection cost by extracting smaller extended subgraphs, rather than extracting extended subgraphs that fill the available memory, can indeed improve the efficiency of MCE computation. We also assess the effectiveness of our cost model, given in Section 4.2.3, for choosing the right size for the extended subgraphs to be extracted. We test the settings on *SeqMCE*, i.e., Algorithm 4.

Figure 3 reports the running time of *SeqMCE* for `blog` and `Web`. We extract extended subgraphs of size from 0.001% to 10% of the available memory size at the second level of Algorithm 4, i.e., each $G_{S_\alpha^+} \in \mathcal{G}_\alpha$ in Lines 11-12 of Procedure 5 is of size xM, where $x \in \{0.001\%, 0.01\%, 0.1\%, 1\%, 10\%\}$ and M is the available memory size, which is set to 1 GB in this experiment. We also report the running time of *SeqMCE* that extracts extended subgraphs by our cost model, represented by "**CM**" on the x-axis. From Figure 3, the sizes of the extended subgraphs extracted by "**CM**" are around 2% for `blog` and 0.05% for `Web`.

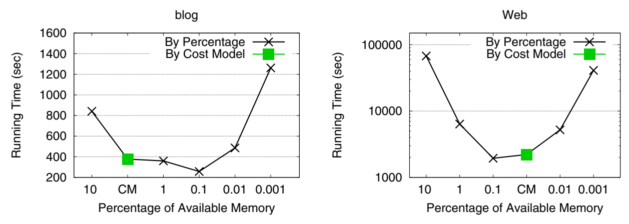

Figure 3: Running time (wall-clock time in seconds) of *SeqMCE* for MCE with varying sizes of extended subgraphs

The results show that for both graphs, the efficiency of MCE is first improved significantly when smaller extended subgraphs are used at the second level of the algorithm, but there is an optimal point after which further decreasing the size of the extended subgraphs reports a significantly deteriorated performance. The trend of the running time for different sizes of extended subgraphs can be explained by our analysis of the gain and loss, i.e., Equation (1) and Equation (2), in Section 4.2.3, which represent a tradeoff between *(1) the cost of set intersection* and *(2) the cost of subgraph extraction and search tree construction*. The same tradeoff can be also be clearly observed from the `LJ` and `BTC` graphs (details omitted due to lack of space).

Having shown the tradeoff, we show that the results obtained by the cost model is near the optimal point. Relatively speaking, the running time reported for the cost model for `blog` is further away from the optimal point than that for `Web`. This is because the `blog` graph is small and hence determining the right subgraph size by the cost model takes a considerable portion of the total time. When the input graph is large, the cost of computation involving the cost model becomes negligible compared to the cost of MCE.

In the subsequent experiments, the algorithm *SeqMCE* always extracts extended subgraphs by the cost model.

5.2 Performance of Sequential Algorithms

We now compare our algorithm *SeqMCE* with other state-of-the-art sequential algorithms for MCE.

Table 2 reports the running time of the four algorithms. For the smaller datasets, `blog` and `LJ`, *EppsteinLS* is the fastest. The *EppsteinLS* algorithm uses a d-degeneracy ordering[2] to limit the depth

[2]A graph is d-degenerate if it has a maximum k-core number of d [18].

Table 2: Running time (wall-clock time in seconds)

	`blog`	`LJ`	`Web`	`BTC`
TomitaTT [29]	11,876	150,122	–	–
EppsteinLS [19]	68	67	–	–
ChengKFYZ [13]	329	6,941	22,840	134,951
SeqMCE	378	175	2,209	28,410

Table 3: Peak memory consumption (in GB)

	`blog`	`LJ`	`Web`	`BTC`
TomitaTT [29]	0.2	0.9	–	–
EppsteinLS [19]	1.3	4.0	–	–
ChengKFYZ [13]	0.1	1.0	1.0	1.0
SeqMCE	0.1	1.0	1.0	1.0

of recursive calls in the *TomitaTT* algorithm to d, where d is not large for many real-world sparse graphs. Our algorithm *SeqMCE* can effectively reduce the cost of set intersection as well as limit the depth of recursive calls; however, it is a semi-streaming algorithm designed for processing large datasets, while *EppsteinLS* is an in-memory algorithm. Thus, *SeqMCE* needs to scan the input graph many times, which explains why it is slower than *EppsteinLS*. However, *EppsteinLS* uses significantly more memory than *SeqMCE* and ran out of memory for the larger graphs, `Web` and `BTC`. On the contrary, with the available memory set at only 1 GB, *SeqMCE* still processes MCE efficiently on both large graphs, and is up to an order of magnitude faster than the state-of-the-art I/O-efficient MCE algorithm, *ChengKFYZ*.

Compared with the in-memory algorithm *TomitaTT*, which also ran out of memory for `Web` and `BTC`, *SeqMCE* is up to two orders of magnitude faster for processing `blog` and `LJ`. Since the smaller datasets can be processed in memory, the main difference between *SeqMCE* and *TomitaTT* for in-memory processing is the new adoption of cost reduction on set intersection, we conclude that the performance improvement of *SeqMCE* over *TomitaTT* is mainly due to our proposal of cost reduction on set intersection. This result thus further demonstrates the effectiveness of the CPU cost reduction method proposed in Section 4.2.

5.3 Performance of Parallel Algorithm

We now evaluate the performance of the parallel algorithm (i.e., Algorithm 6), denoted by *ParMCE*. There are some recent algorithms for parallel MCE [17, 27, 33], but we were not able to obtain their code for comparison.

We report the running time of *ParMCE* in Figure 4. To show that our method is truly effective in handling graphs that cannot fit in memory, we set the available memory to only 64 MB for `blog` and `LJ`, and 256 MB for the two large graphs `Web` and `BTC`.

The figures show that when two machines are used, the running time is nearly halved for all the datasets. The efficiency continues to improve when more machines are used, although the speed-up becomes milder when more machines are used. With only eight machines, we can finish computing MCE in the largest dataset in about an hour.

6. RELATED WORK

Maximal clique enumeration has been studied extensively and comprehensive reviews can be found in [10] and [8] (the latter also discusses maximum clique finding). The first algorithms were the *backtracking* method [3, 9] that use $O(|V|^2)$ memory space. Then effective pruning strategy by selecting good *pivots* was employed

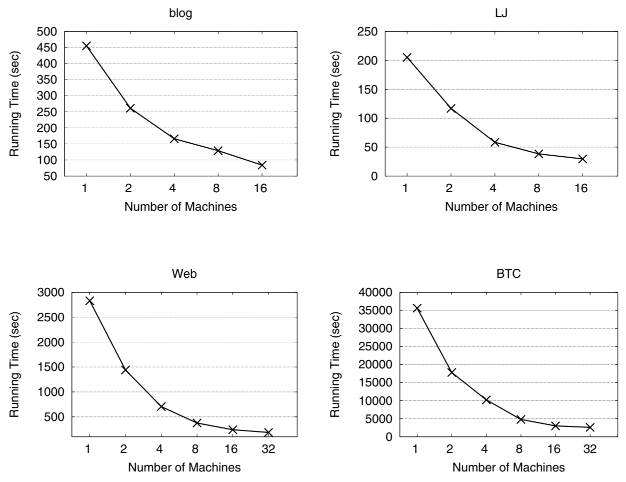

Figure 4: Running time (wall-clock time in seconds) of *ParMCE*

to further reduce the search space [10, 23, 29]. Algorithm for d-degenerate graphs was also proposed [18, 19], which achieves a time complexity of $O(d|V|3^{d/3})$ by utilizing a d-degeneracy ordering. However, all these studies did not focus on reducing the memory complexity and require $\Omega(|V| + |E|)$ memory space, which may not be practical for large disk-resident graphs.

Recently, several parallel algorithms were proposed for MCE. Most of these parallel algorithms [17, 27] still require the entire input graph to be resident in main memory of each computing node, and thus are not practical for processing large graphs that cannot fit in memory. A MapReduce algorithm [33] was also proposed for MCE. The MapReduce model, however, is slow in practice for enumeration tasks such as MCE and triangle listing in large graphs, mainly because of the huge amount of intermediate data produced in the shuffling phase between Mappers and Reducers.

Algorithm for output-sensitive MCE was also introduced which is based on reverse search [30]. The time delay was reduced to $O(d_{max}^4)$ for sparse graphs using matrix multiplication [25], where d_{max} is the maximum degree of a graph; but the algorithm requires $O(|V| \cdot |E|)$ preprocessing time. Other algorithms, such as computing a k-clique by joining two $(k-1)$-cliques [24], by making use of triangles [31], and enumerating maximal cliques of size larger than a threshold [26], were also studied. However, all these algorithms require memory space at least $\Omega(|V| + |E|)$.

Stix [28] proposed a streaming algorithm that updates the set of maximal cliques upon each edge insertion, but the update is expensive and it also requires to keep all maximal cliques in memory, which has been verified in [12].

Recently, Cheng et al. [12, 13] proposed an efficient algorithm that recursively extracts a core part of the input graph for local MCE computation. This idea is similar to the algorithm described in Section 4.1, but a fundamental difference in the formulation of the subgraphs to be extracted, i.e., their subgraphs still need to access the input graph for MCE while ours do not, making it nontrivial to parallelize their algorithm as what we do. We have also verified in Section 5 that our algorithm is significantly more efficient.

Other related works that also avoid random disk access by extracting small subgraphs include core/truss decomposition [11, 32], triangle listing [14, 15]. However, the subgraphs used in these works cannot guarantee the maximality of the cliques computed and their algorithm frameworks are also not suitable for the task of MCE.

7. CONCLUSIONS

We presented efficient algorithms to reduce both the I/O cost and CPU cost of MCE in massive networks. We verified the performance of our algorithms comparing with the state-of-the-art algorithms for MCE [12, 13, 18, 19, 29]. Our results first demonstrate that, by reducing the cost of set intersection in MCE, we are able to achieve significant speed-ups in both cases when the input graph can fit in main memory or cannot fit in main memory. Then, we also showed that our parallel algorithm for MCE significantly speeds up the MCE computation compared with the sequential algorithm.

8. ACKNOWLEDGMENTS

The authors would like to thank the reviewers of this paper for their constructive comments. This research is supported in part by the A*STAR Thematic Strategic Research Programme Grants (102 158 0034 and 112 172 0013).

9. REFERENCES

[1] F. N. Abu-Khzam, N. E. Baldwin, M. A. Langston, and N. F. Samatova. On the relative efficiency of maximal clique enumeration algorithms, with applications to high-throughput computational biology. In *International Conference on Research Trends in Science and Technology*, 2005.

[2] A. Aggarwal and S. Vitter, Jeffrey. The input/output complexity of sorting and related problems. *Commun. ACM*, 31(9):1116–1127, 1988.

[3] E. A. Akkoyunlu. The enumeration of maximal cliques of large graphs. *SIAM J. Comput.*, 2(1):1–6, 1973.

[4] K. Andreev and H. Racke. Balanced graph partitioning. In *SPAA*, pages 120–124, 2004.

[5] H. R. Bernard, P. D. Killworth, and L. Sailer. Informant accuracy in social network data iv: a comparison of clique-level structure in behavioral and cognitive network data. *Social Networks*, 2(3):191–218, 1979.

[6] N. M. Berry, T. H. Ko, T. Moy, J. Smrcka, J. Turnley, and B. Wu. Emergent clique formation in terrorist recruitment. In *The AAAI-04 Workshop on Agent Organizations: Theory and Practice*, 2004.

[7] V. Boginski, S. Butenko, and P. M. Pardalos. Statistical analysis of financial networks. *Computational Statistics & Data Analysis*, 48(2):431–443, 2005.

[8] I. M. Bomze, M. Budinich, P. M. Pardalos, and M. Pelillo. The maximum clique problem. In *Handbook of Combinatorial Optimization*, pages 1–74. Kluwer Academic Publishers, 1999.

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

[10] F. Cazals and C. Karande. A note on the problem of reporting maximal cliques. *Theor. Comput. Sci.*, 407(1-3):564–568, 2008.

[11] J. Cheng, Y. Ke, S. Chu, and M. T. Özsu. Efficient core decomposition in massive networks. In *ICDE*, pages 51–62, 2011.

[12] 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*, pages 447–458, 2010.

[13] J. Cheng, Y. Ke, A. W.-C. Fu, J. X. Yu, and L. Zhu. Finding maximal cliques in massive networks. *ACM Trans. Database Syst.*, 36(4):21, 2011.

[14] S. Chu and J. Cheng. Triangle listing in massive networks. *To appear in ACM Transactions on Knowledge Discovery from Data, 2012.*

[15] S. Chu and J. Cheng. Triangle listing in massive networks and its applications. In *KDD*, pages 672–680, 2011.

[16] G. Creamer, R. Rowe, S. Hershkop, and S. J. Stolfo. Segmentation and automated social hierarchy detection through email network analysis. In *WebKDD/SNA-KDD*, pages 40–58, 2007.

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

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

[19] D. Eppstein and D. Strash. Listing all maximal cliques in large sparse real-world graphs. In *SEA*, pages 364–375, 2011.

[20] K. Faust and S. Wasserman. Social network analysis: Methods and applications. *Cambridge University Press*, 1995.

[21] K. Gouda and M. J. Zaki. Efficiently mining maximal frequent itemsets. In *ICDM*, pages 163–170, 2001.

[22] D. E. Knuth. Estimating the efficiency of backtrack programs. *Mathematics of Computation*, 29(129):121–136, 1975.

[23] I. Koch. Enumerating all connected maximal common subgraphs in two graphs. *Theor. Comput. Sci.*, 250(1-2):1–30, 2001.

[24] F. Kose, W. Weckwerth, T. Linke, and O. Fiehn. Visualizing plant metabolomic correlation networks using clique-metabolite matrices. *Bioinformatics*, 17(12):1198–1208, 2001.

[25] K. Makino and T. Uno. New algorithms for enumerating all maximal cliques. In *SWAT*, pages 260–272, 2004.

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

[27] M. C. Schmidt, N. F. Samatova, K. Thomas, and B.-H. Park. A scalable, parallel algorithm for maximal clique enumeration. *J. Parallel Distrib. Comput.*, 69(4):417–428, 2009.

[28] V. Stix. Finding all maximal cliques in dynamic graphs. *Computational Optimization and applications*, 27:173–186, 2004.

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

[30] S. Tsukiyama, M. Ide, H. Ariyoshi, and I. Shirakawa. A new algorithm for generating all the maximal independent sets. *SIAM J. Comput.*, 6(3):505–517, 1977.

[31] L. Wan, B. Wu, N. Du, Q. Ye, and P. Chen. A new algorithm for enumerating all maximal cliques in complex network. In *ADMA*, pages 606–617, 2006.

[32] J. Wang and J. Cheng. Truss decomposition in massive networks. *To appear in Proceedings of the Very Large Database Endowment, 2012.*

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