Shared-Memory Parallel Maximal Clique Enumeration from Static and Dynamic Graphs*

APURBA DAS, Iowa State University

SEYED-VAHID SANEI-MEHRI, Iowa State University

SRIKANTA TIRTHAPURA, Iowa State University

Maximal Clique Enumeration (MCE) is a fundamental graph mining problem, and is useful as a primitive in identifying dense structures in a graph. Due to the high computational cost of MCE, parallel methods are imperative for dealing with large graphs. We present shared-memory parallel algorithms for MCE, with the following properties: (1) the parallel algorithms are provably work-efficient relative to a state-of-the-art sequential algorithm (2) the algorithms have a provably small parallel depth, showing they can scale to a large number of processors, and (3) our implementations on a multicore machine show good speedup and scaling behavior with increasing number of cores, and are substantially faster than prior shared-memory parallel algorithms for MCE; for instance, on certain input graphs, while prior works either ran out of memory or did not complete in 5 hours, our implementation finished within a minute using 32 cores. We also present work-efficient parallel algorithms for maintaining the set of all maximal cliques in a dynamic graph that is changing through the addition of edges.

1 INTRODUCTION

We study Maximal Clique Enumeration (MCE) from a graph, which requires to enumerate all cliques (complete subgraphs) in the graph that are maximal. A clique C in a graph $G = (V, E)$ is a (dense) subgraph such that every pair of vertices in C are directly connected by an edge. A clique C is said to be maximal when there is no clique C' such that C is a proper subgraph of C' (see Figure 1). Maximal cliques are perhaps the most fundamental dense subgraphs, and MCE has been widely used in diverse research areas, e.g. the works of Palla et al. [44] on discovering protein groups by mining cliques in a protein-protein network, of Koichi et al. [29] on discovering chemical structures using MCE on a graph derived from large-scale chemical databases, various problems on mining biological data [8, 21, 22, 24, 39, 46, 63], overlapping community detection [62], location-based services on spatial databases [64], and inference in graphical models [30]. MCE is important in static as well as dynamic networks, i.e. networks that change over time due to the addition and deletion of vertices and

---

Authors' addresses: Apurba Das, Iowa State University, adas@iastate.edu; Seyed-Vahid Sanei-Mehri, Iowa State University, vas@iastate.edu; Srikanta Tirthapura, Iowa State University, snt@iastate.edu.

(a) Input Graph G **(b)** Non-Maximal Clique in G **(c)** Maximal Clique in G
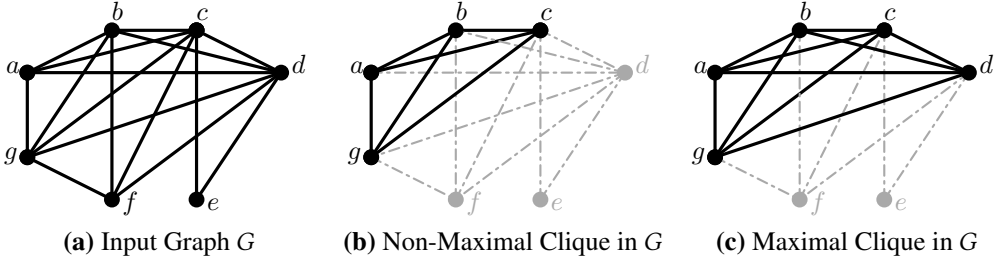
Fig. 1. Maximal clique in a graph

edges. For example, Chateau et al. [7] use MCE in studying changes in the genetic rearrangements of bacterial genomes, when new genomes are added to the existing genome population. Maximal cliques are building blocks for computing approximate common intervals of multiple genomes. Duan et al. [17] use MCE on a dynamic graph to track highly interconnected communities in a dynamic social network.

MCE is a computationally hard problem since it is harder than the *maximum* clique problem, a classical NP-complete combinatorial problem that asks to find a clique of the largest size in a graph. Note that maximal clique and maximum clique are two related, but distinct notions. A maximum clique is also a maximal clique, but a maximal clique need not be a maximum clique. The computational cost of enumerating maximal cliques can be higher than the cost of finding the maximum clique, since the output size (set of all maximal cliques) can itself be very large. Moon and Moser [41] showed that a graph on n vertices can have as many as $3^{n/3}$ maximal cliques, which is proved to be a tight bound. Real-world networks typically do not have cliques of such high complexity and as a result, it is feasible to enumerate maximal cliques from large graphs. The literature is rich on sequential algorithms for MCE. Bron and Kerbosch [5] introduced a backtracking search method to enumerate maximal cliques. Tomita et. al [56] introduced the idea of "pivoting" in the backtracking search, which led to a significant improvement in the runtime. This has been followed up by further work such as due to Eppstein et al. [18], who used a degeneracy-based vertex ordering on top of the pivot selection strategy.

Sequential approaches to MCE can lead to high runtimes on large graphs. Based on our experiments, a real-world network `Orkut` with approximately 3 million vertices, 117 million edges requires approximately 8 hours to enumerate all maximal cliques using an efficient sequential algorithm due to Tomita et al. [56]. Graphs that are larger and/or more complex cannot be handled by sequential algorithms with a reasonable turnaround time, and the high computational complexity of MCE calls for parallel methods.

In this work, we consider shared-memory parallel methods for MCE. In the shared-memory model, the input graph can reside within globally shared memory, and multiple threads can work in parallel on enumerating maximal cliques. Shared-memory parallelism is of high interest today since machines with tens to hundreds of cores and hundreds of Gigabytes of shared-memory are readily available. The advantage of using shared-memory approach over a distributed memory approach are: (1) Unlike distributed memory, it is not necessary to divide the graph into subgraphs and communicate the subgraphs among processors. In shared-memory, different threads can work concurrently on a single shared copy of the graph (2) Sub-problems generated during MCE are often highly imbalanced, and it is hard to predict which sub-problems are small and which are large, while initially dividing the problem into sub-problems. With a shared-memory method, it is possible to further subdivide sub-problems and process them in parallel. With a distributed memory method, handling such imbalances in sub-problem sizes requires greater coordination and is more complex.
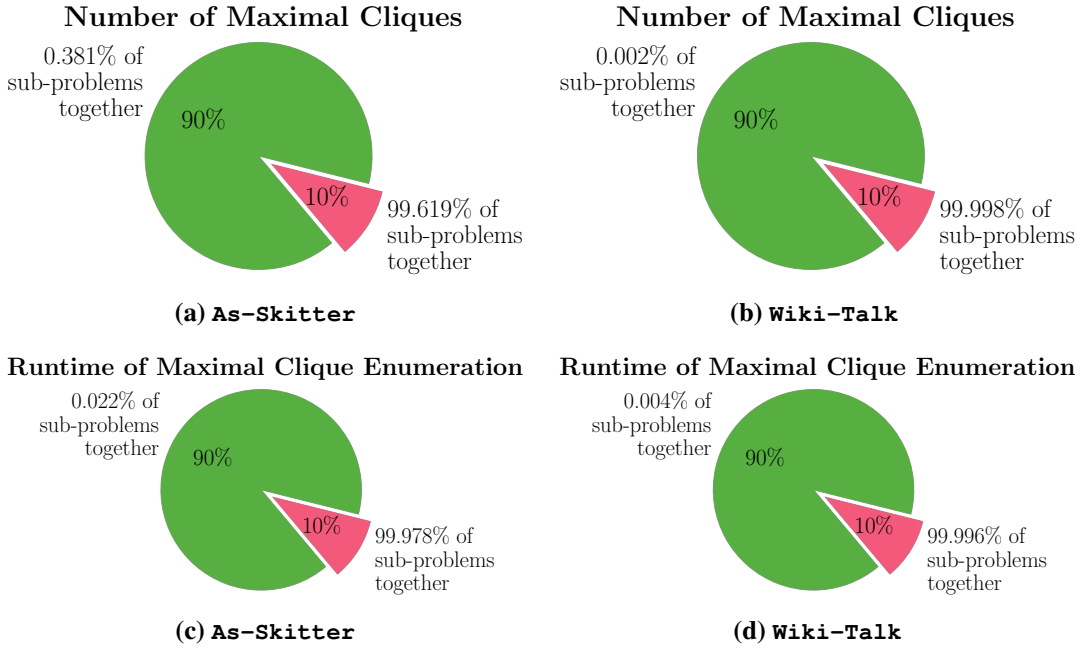
Number of Maximal Cliques

0.381% of
sub-problems
together

90%

10%

99.619% of
sub-problems
together

(a) As-Skitter

Number of Maximal Cliques

0.002% of
sub-problems
together

90%

10%

99.998% of
sub-problems
together

(b) Wiki-Talk

Runtime of Maximal Clique Enumeration

0.022% of
sub-problems
together

90%

10%

99.978% of
sub-problems
together

(c) As-Skitter

Runtime of Maximal Clique Enumeration

0.004% of
sub-problems
together

90%

10%

99.996% of
sub-problems
together

(d) Wiki-Talk

Fig. 2. Imbalanced in sizes of sub-problems for MCE, where each sub-problem corresponds to the maximal cliques of a single vertex in the given graph. (a) **As-Skitter**: 0.3% of sub-problems form 90% of total number of maximal cliques. (b) **Wiki-Talk**: only 0.002% of sub-problems yield 90% of all maximal cliques. (c) **As-Skitter**: 0.02% of sub-problems take 90% of total runtime of MCE. (d) **Wiki-Talk**: only 0.004% of sub-problems take 90% of total runtime of MCE

To show how imbalanced the sub-problems can be, in Figure 2, we show data for two real-world networks **As-Skitter** and **Wiki-Talk**. These two networks have millions of edges and tens of millions of maximal cliques (for statistics on these networks, see Section 6). Consider a division of the MCE problem into per-vertex sub-problems, where each sub-problem corresponds to the set of all maximal cliques containing a single vertex in a network and suppose these sub-problems were solved independently, while taking care to prune out search for the same maximal clique multiple times. For **As-Skitter**, we observed that 90% of total runtime required for MCE is taken by only 0.022% of the sub-problems and less than 0.4% of all sub-problems yield 90% of all maximal cliques. Even larger skew in sub-problem sizes is observed in the **Wiki-Talk** graph. This data demonstrates that load balancing is a central issue for parallel MCE.

Prior works on parallel MCE have largely focused on distributed memory algorithms [37, 51, 55, 60]. There are a few works on shared-memory parallel algorithms [16, 34, 65]. However, these algorithms do not scale to larger graphs due to memory or computational bottlenecks – either the algorithms miss out significant pruning opportunities as in [16], or they need to generate a large number of non-maximal cliques as in [34, 65].

1.1 Our Contributions

We make the following contributions towards enumerating all maximal cliques in a simple graph.

Theoretically Efficient Parallel Algorithm: We present a shared-memory parallel algorithm ParTTT, which takes as input a graph G and enumerates all maximal cliques in G. ParTTT is an efficient parallelization of the algorithm due to Tomita, Tanaka, and Takahashi [56]. Our analysis of

Table 1. Summary of shared-memory parallel algorithms for MCE.

Algorithm	Type	Description
ParTTT	Static	A work-efficient parallel algorithm for MCE on a static graph
ParMCE	Static	A practical parallel algorithm for MCE on a static graph dealing with load imbalance
ParIMCE	Dynamic	ParIMCENew: A work-efficient parallel algorithm for enumerating new maximal cliques in a dynamic graph.
		ParIMCESub: A work-efficient parallel algorithm for enumerating subsumed maximal cliques in a dynamic graph.

ParTTT using a work-depth model of computation [53] shows that it is work-efficient when compared with [56] and has a low parallel depth. To our knowledge, this is the first shared-memory parallel algorithm for MCE with such provable properties.

Optimized Parallel Algorithm: We present a shared-memory parallel algorithm ParMCE that builds on ParTTT and yields improved practical performance. Unlike ParTTT, which starts with a single task at the top level that spawns recursive subtasks as it proceeds, leading to a lack of parallelism at the top level of recursion, ParMCE spawns multiple parallel tasks at the top level. To achieve this, ParMCE uses per-vertex parallelization, where a separate sub-problem is created for each vertex and different sub-problems are processed in parallel. Each sub-problem is required to enumerate cliques which contain the assigned vertex and care is taken to prevent overlap between sub-problems. Each per-vertex sub-problem is further processed in parallel using ParTTT – this additional (recursive) level of parallelism using ParTTT is important since different per-vertex sub-problems may have significantly different computational costs, and having each run as a separate sequential task may lead to uneven load balance. To further address load balance, we use a vertex ordering in assigning cliques to different per-vertex sub-problems. For ordering the vertices, we use various metrics such as degree, triangle count, and the degeneracy number of the vertices.

Incremental Parallel Algorithm: Next, we present a parallel algorithm ParIMCE that can maintain the set of maximal cliques in a dynamic graph, when the graph is updated due to the addition of new edges. When a batch of edges are added to the graph, ParIMCE can (in parallel) enumerate the set of all new maximal cliques that emerged and the set of all maximal cliques that are no longer maximal (subsumed cliques). ParIMCE consists of two parts: ParIMCENew for enumerating new maximal cliques, and ParIMCESub for enumerating subsumed maximal cliques. We analyze ParIMCE using the work-depth model and show that it is work-efficient relative to an efficient sequential algorithm, and has a low parallel depth. A summary of our algorithms is shown in Table 1.

Experimental Evaluation: We implemented all our algorithms, and our experiments show that ParMCE yields a speedup of **15x-21x** when compared with an efficient sequential algorithm (due to Tomita et al. [56]) on a multicore machine with 32 physical cores and 1 TB RAM. For example, on the **Wikipedia** network with around 1.8 million vertices, 36.5 million edges, and 131.6 million maximal cliques, ParTTT achieves a **16.5x** parallel speedup over the sequential algorithm, and the optimized ParMCE achieves a **21.5x** speedup, and completed in approximately two minutes. In contrast, prior shared-memory parallel algorithms for MCE [16, 34, 65] failed to handle the input graphs that we considered, and either ran out of memory ([34, 65]) or did not complete in 5 hours ([16]).

On dynamic graphs, we observe that ParIMCE gives a **3x-19x** speedup over a state-of-the-art sequential algorithm IMCE [13] on a multicore machine with 32 cores. Interestingly, the speedup of the parallel algorithm increases with the magnitude of change in the set of maximal cliques – the "harder" the dynamic enumeration task is, the larger is the speedup obtained. For example, on a dense graph such as **Ca-Cit-HepTh** (with original graph density of 0.01), we get approximately a **19x** speedup over the sequential IMCE. More details are presented in Section 6.

Techniques for Load Balancing: Our parallel methods can effectively balance the load in solving parallel MCE. As shown in Figure 2, "natural" sub-problems of MCE are highly imbalanced, and, therefore, load balancing is not trivial. In our algorithms, sub-problems of MCE are broken down into smaller sub-problems, according to the search used by the sequential algorithm [56], and this process continues recursively. As a result, the final sub-problem that is solved in a single task is not so large as to create load imbalances. Our experiments in Section 6 demonstrate that the recursive splitting of sub-problems in MCE is essential for achieving a high speedup over existing algorithms [56]. In order to efficiently assign these (dynamically created) tasks to threads at runtime, we utilize a *work stealing scheduler* [3, 4].

    Roadmap. The rest of the paper is organized as follows. We discuss related works in Section 2, we present preliminaries in Section 3, followed by a description of algorithms for a static graph in Section 4, algorithms for a dynamic graph in Section 5, an experimental evaluation in Section 6, and conclusions in Section 7.

2 RELATED WORK

Maximal Clique Enumeration (MCE) from a graph is a fundamental problem that has been extensively studied for more than two decades, and there are multiple prior works on sequential and parallel algorithms. We first discuss sequential algorithms for MCE, followed by parallel algorithms.

Sequential MCE: Bron and Kerbosch [5] presented an algorithm for MCE based on depth-first-search. Following their work, a number of algorithms have been presented [6, 9, 18, 28, 38, 56, 57]. The algorithm of Tomita et al. [56] has a worst-case time complexity $O(3^{\frac{n}{3}})$ for an n vertex graph, which is optimal in the worst-case, since the size of the output can be as large as $O(3^{\frac{n}{3}})$ [41]. Eppstein et al. [18, 19] present an algorithm for sparse graphs whose complexity can be parameterized by the degeneracy of the graph, a measure of graph sparsity.

    Another approach to MCE is a class of "output-sensitive" algorithms whose time complexity for enumerating maximal cliques is a function of the size of the output. There exist many such output-sensitive algorithms for MCE, including [9, 11, 38, 57], which can be viewed as instances of a general paradigm called "reverse-search" [1]. A recent algorithm [11] provides favorable tradeoffs for delay (time between two enumerated maximal cliques), when compared with prior works. In terms of practical performance, the best output-sensitive algorithms [9, 11, 38] are not as efficient as the best depth-first-search based algorithms [18, 56]. Other sequential methods for MCE include algorithms due to Kose et al. [31], Johnson et al. [23], Li et al. [35], on a special class of graphs due to Fox et al. [20], on temporal graph due to Qin et al. [45]. Multiple works have considered sequential algorithms for maintaining the set of maximal cliques [13, 43, 54] on a dynamic graph, and, to our knowledge, the most efficient algorithm is the one due to Das et al. [13].

Parallel MCE: There are multiple prior works on parallel algorithms for MCE [15, 37, 51, 55, 60, 61, 65]. We first discuss shared-memory algorithms and then distributed memory algorithms. Zhang et al. [65] present a shared-memory parallel algorithm based on the sequential algorithm due to Kose et al. [31]. This algorithm computes maximal cliques in an iterative manner, and, in each iteration, it maintains a set of cliques that are not necessarily maximal and, for each such clique, maintains the set of vertices that can be added to form larger cliques. This algorithm does not provide a theoretical guarantee on the runtime and suffers for large memory requirement. Du et al. [15] present a output-sensitive shared-memory parallel algorithm for MCE, but their algorithm suffers from poor load balancing as also pointed out by Schmidt et al. [51]. Lessley et al. [34] present a shared memory parallel algorithm that generates maximal cliques using an iterative method, where in each iteration, cliques of size $(k-1)$ are extended to cliques of size k. The algorithm of [34] is memory-intensive,

since it needs to store a number of intermediate non-maximal cliques in each iteration. Note that the number of non-maximal cliques may be far higher than the number of maximal cliques that are finally emitted, and a number of distinct non-maximal cliques may finally lead to a single maximal clique. In the extreme case, a complete graph on n vertices has $(2^n - 1)$ non-maximal cliques, and only a single maximal clique. We present a comparison of our algorithm with [15, 34, 65] in later sections.

Distributed memory parallel algorithms for MCE include works due to Wu et al. [60], designed for the MapReduce framework, Lu et al. [37], which is based on the sequential algorithm due to Tsukiyama et al. [57], Svendsen et al. [55], Wang et al. [59], and algorithm for sparse graph due to Yu and Liu [61]. Other works on parallel and sequential algorithms for enumerating dense subgraphs from a massive graph include sequential algorithms for enumerating quasi-cliques [36, 49, 58], parallel algorithms for enumerating k-cores [14, 25, 40, 50], k-trusses [26, 27, 50], nuclei [50], and distributed memory algorithms for enumerating k-plexes [10], bicliques [42].

3 PRELIMINARIES

We consider a simple undirected graph without self loops or multiple edges. For graph G, let $V(G)$ denote the set of vertices in G and $E(G)$ denote the set of edges in G. Let n denote the size of $V(G)$, and m denote the size of $E(G)$. For vertex $u \in V(G)$, let $\Gamma_G(u)$ denote the set of vertices adjacent to u in G. When the graph G is clear from the context, we use $\Gamma(u)$ to mean $\Gamma_G(u)$. Let $C(G)$ denote the set of all maximal cliques in G.

Sequential Algorithm TTT**:** The algorithm due to Tomita, Tanaka, and Takahashi. [56], which we call TTT, is a recursive backtracking-based algorithm for enumerating all maximal cliques in an undirected graph, with a worst-case time complexity of $O(3^{n/3})$ where n is the number of vertices in the graph. In practice, this is one of the most efficient sequential algorithms for MCE. Since we use TTT as a subroutine in our parallel algorithms, we present a short description here.

In any recursive call, TTT maintains three disjoint sets of vertices K, cand, and fini, where K is a candidate clique to be extended, cand is the set of vertices that can be used to extend K, and fini is the set of vertices that are adjacent to K, but need not be used to extend K (these are being explored along other search paths). Each recursive call iterates over vertices from cand and in each iteration, a vertex $q \in$ cand is added to K and a new recursive call is made with parameters $K \cup \{q\}$, $cand_q$, and $fini_q$ for generating all maximal cliques of G that extend $K \cup \{q\}$ but do not contain any vertices from $fini_q$. The sets $cand_q$ and $fini_q$ can only contain vertices that are adjacent to all vertices in $K \cup \{q\}$. The clique K is a maximal clique when both cand and fini are empty.

The ingredient that makes TTT different from the algorithm due to Bron and Kerbosch [5] is the use of a "pivot" where a vertex $u \in$ cand $\cup$ fini is selected that maximizes |cand $\cap \Gamma(u)$|. Once the pivot u is computed, it is sufficient to iterate over all the vertices of cand $\setminus \Gamma(u)$, instead of iterating over all vertices of cand. The pseudo code of TTT is presented in Algorithm 1. For the initial call, K and fini are initialized to an empty set, cand is the set of all vertices of G.

Parallel Cost Model: For analyzing our shared-memory parallel algorithms, we use the CRCW PRAM model [2], which is a model of shared parallel computation that assumes concurrent reads and concurrent writes. Our parallel algorithm can also work in other related models of shared-memory such as EREW PRAM (exclusive reads and exclusive writes), with a logarithmic factor increase in work as well as parallel depth. We measure the effectiveness of the parallel algorithm using the *work-depth* model [53]. Here, the "work" of a parallel algorithm is equal to the total number of operations of the parallel algorithm, and the "depth" (also called the "parallel time" or the "span") is the longest chain of dependent computations in the algorithm. A parallel algorithm is said to be

---

Algorithm 1: $\mathrm{TTT}(G, K, \text{cand}, \text{fini})$

Input: G - the input graph

$\qquad$ K - a clique to extend,

cand - a set of vertices that can be used extend K,

fini - a set of vertices that have been used to extend K

Output: Set of all maximal cliques of G containing K and vertices from cand but not containing any vertex

$\qquad$ from fini

1 **if** (cand $= \emptyset$) & (fini $= \emptyset$) **then**

2 $\quad$ | $\quad$ Output K and return

3 pivot $\leftarrow$ ($u \in$ cand $\cup$ fini) such that u maximizes the size of cand $\cap \Gamma_G(u)$

4 ext $\leftarrow$ cand $- \Gamma_G(\text{pivot})$

5 **for** $q \in$ ext **do**

6 $\quad$ | $\quad$ $K_q \leftarrow K \cup \{q\}$

7 $\quad$ | $\quad$ cand$_q \leftarrow$ cand $\cap \Gamma_G(q)$

8 $\quad$ | $\quad$ fini$_q \leftarrow$ fini $\cap \Gamma_G(q)$

9 $\quad$ | $\quad$ cand $\leftarrow$ cand $- \{q\}$

10 $\quad$ | $\quad$ fini $\leftarrow$ fini $\cup \{q\}$

11 $\quad$ | $\quad$ $\mathrm{TTT}(G, K_q, \text{cand}_q, \text{fini}_q)$

---

work-efficient if its total work is of the same order as the work due to the best sequential algorithm.[1] We aim for work-efficient algorithms with a low depth, ideally poly-logarithmic in the size of the input. Using Brent's theorem [2], it can be seen that a parallel algorithm on input size n with a depth of d can theoretically achieve $\Theta(p)$ speedup on p processors as long as $p = O(n/d)$.

We next restate a result on concurrent hash tables [52] that we use in proving the work and depth bounds of our parallel algorithms.

THEOREM 3.1 (THEOREM 3.15 [52]). *There is an implementation of a hash table, which, given a hash function with expected uniform distribution, performs n_1 insert, n_2 delete and n_3 find operations in parallel using $O(n_1 + n_2 + n_3)$ work and $O(1)$ depth on average.*

4 PARALLEL MCE ALGORITHMS ON A STATIC GRAPH

In this section, we present new shared-memory parallel algorithms for MCE. We first describe a parallel algorithm ParTTT, a parallelization of the sequential TTT algorithm and an analysis of its theoretical properties. Then, we discuss bottlenecks in ParTTT that arise in practice, leading us to another algorithm ParMCE with a better practical performance. ParMCE uses ParTTT as a subroutine – it creates appropriate sub-problems that can be solved in parallel and hands off the enumeration task to ParTTT.

4.1 Algorithm ParTTT

Our first algorithm ParTTT is a work-efficient parallelization of the sequential TTT algorithm. The two main components of TTT (Algorithm 1) are (1) Selection of the pivot element (Line 3) and (2) Sequential backtracking for extending candidate cliques until all maximal cliques are explored (Line 5 to Line 11). We discuss how to parallelize each of these steps.

Parallel Pivot Selection: Within a single recursive call of ParTTT, the pivot element is computed in parallel using two steps, as described in ParPivot (Algorithm 2). In the first step, the size of the

---

[1]Note that work-efficiency in the CRCW PRAM model does not imply work-efficiency in the EREW PRAM model

intersection $\text{cand} \cap \Gamma(u)$ is computed in parallel for each vertex $u \in \text{cand} \cup \text{fini}$. In the second step, the vertex with the maximum intersection size is selected. The parallel algorithm for selecting a pivot is presented in Algorithm 2. The following lemma proves that the parallel pivot selection is work-efficient with logarithmic depth:

LEMMA 1. *The total work of* ParPivot *is* $O(\sum_{w \in \text{cand} \cup \text{fini}}(\min\{|\text{cand}|, |\Gamma(w)|\}))$, *which is* $O(n^2)$, *and depth is* $O(\log n)$.

PROOF. If sets cand and $\Gamma(w)$ are stored as hashsets, then, for vertex w, the size $t_w = |\text{intersect}(\text{cand}, \Gamma(w))|$ can be computed sequentially in time $O(\min\{|\text{cand}|, |\Gamma(w)|\})$ – the intersection of two sets S_1 and S_2 can be found by considering the smaller set among the two, say S_2, and searching for its elements within the larger set, say S_1. It is possible to parallelize the computation of $\text{intersect}(S_1, S_2)$ by executing the search elements of S_2 in parallel, followed by counting the number of elements that lie in the intersection, which can also be done in parallel in a work-efficient manner using $O(1)$ depth using Theorem 3.1. Since computing the maximum of a set of n numbers can be accomplished using work $O(n)$ and depth $O(\log n)$, for vertex w, t_w can be computed using work $O(\min\{|\text{cand}|, |\Gamma(w)|\})$ and depth $O(\log n)$. Once t_w (i.e. $|\text{cand} \cap \Gamma(w)|$) is computed for every vertex $w \in \text{cand} \cup \text{fini}$, $argmax(\{t_w : w \in \text{cand} \cup \text{fini}\})$ can be obtained using additional work $O(|\text{cand} \cup \text{fini}|)$ and depth $O(\log n)$. Hence, the total work of ParPivot is $O(\sum_{w \in \text{cand} \cup \text{fini}}(\min\{|\text{cand}|, |\Gamma(w)|\}))$. Since the size of cand, fini, and $\Gamma(w)$ are bounded by n, this is $O(n^2)$, but typically much smaller in practice.   □

Algorithm 2: ParPivot$(G, K, \text{cand}, \text{fini})$

Input: G - the input graph; K - a clique in G that may be further extended; cand - a set of vertices that may extend K; fini - a set of vertices that have been used to extend K.

Output: Pivot vertex $v \in \text{cand} \cup \text{fini}$.

1 **for** $w \in \text{cand} \cup \text{fini}$ **do in parallel**
2   In parallel, compute $t_w \leftarrow |\text{intersect}(\text{cand}, \Gamma_G(w))|$

3 In parallel, find $v \leftarrow argmax(\{t_w : w \in \text{cand} \cup \text{fini}\})$
4 **return** v

Parallelization of Backtracking: We first note that there is a sequential dependency among the different iterations within a recursive call of TTT. In particular, the contents of the sets cand and fini in a given iteration are derived from the contents of cand and fini in the previous iteration. Such sequential dependence of updates to cand and fini restricts us from calling the recursive TTT for different vertices of ext in parallel. To remove this dependency, we adopt a different view of TTT which enables us to make the recursive calls in parallel. The elements of ext, the vertices to be considered for extending a maximal clique, are arranged in a predefined total order. Then, we unroll the loop and explicitly compute the parameters cand and fini for recursive calls.

Suppose $\langle v_1, v_2, ..., v_\kappa \rangle$ is the order of vertices in ext. Each vertex $v_i \in \text{ext}$, once added to K, should be removed from further consideration in cand. To ensure this, in ParTTT, we explicitly remove vertices $v_1, v_2, ..., v_{i-1}$ from cand and add them to fini, before making the recursive calls. As a consequence, parameters of the ith iteration are computed independently of prior iterations.

Here, we prove the work efficiency and low depth of ParTTT in the following lemma:

LEMMA 2. *Total work of* ParTTT *(Algorithm 3) is* $O(3^{n/3})$ *and depth is* $O(M \log n)$ *where n is the number of vertices in the graph, and M is the size of a maximum clique in G.*

Algorithm 3: ParTTT(G, K, cand, fini)

Input: G - the input graph
K - a non-maximal clique to extend
cand - set of vertices that may extend K
fini - vertices that have been used to extend K
Output: A set of all maximal cliques of G containing K and vertices from cand but not containing any
vertex from fini

1 **if** (cand = ∅) & (fini = ∅) **then**
2   Output K and **return**

3 pivot ← ParPivot(G, cand, fini)
4 ext$[1..\kappa]$ ← cand − Γ_G(pivot) // in parallel
5 **for** $i \in [1..\kappa]$ **do in parallel**
6   q ← ext$[i]$
7   K_q ← $K \cup \{q\}$
8   cand$_q$ ← intersect(cand \ ext$[1..i-1], \Gamma_G(q)$)
9   fini$_q$ ← intersect(fini ∪ ext$[1..i-1], \Gamma_G(q)$)
10   ParTTT(G, K_q, cand$_q$, fini$_q$)

PROOF. First, we analyze the total work. Note that the computational tasks in ParTTT is different from TTT at Line 8 and Line 9 of ParTTT where at an iteration i, we remove all vertices $\{v_1, v_2, ..., v_{i-1}\}$ from cand and add all these vertices to fini as opposed to the removal of a single vertex v_{i-1} from cand and addition of that vertex to fini as in TTT (Line 8 and Line 9 of Algorithm 1). Therefore, in ParTTT, additional $O(n)$ work is required due to independent computations of cand$_q$ and fini$_q$. The total work, excluding the call to ParPivot is $O(n^2)$. Adding up the work of ParPivot, which requires $O(n^2)$ work and $O(n^2)$ total work for each single call of ParTTT excluding further recursive calls (Algorithm 3, Line 10), which is the same as in original sequential algorithm TTT (Section 4, [56]). Hence, using Lemma 2 and Theorem 3 of [56], we infer that the total work of ParTTT is the same as the sequential algorithm TTT and is bounded by $O(3^{n/3})$.

Next, we analyze the depth of the algorithm. The depth of ParTTT consists of the (sum of the) following components: (1) Depth of ParPivot, (2) Depth of computation of ext, (3) Maximum depth of an iteration in the **for** loop from Line 5 to Line 10. According to Lemma 1, the depth of ParPivot is $O(\log n)$. The depth of computing ext is $O(\log n)$ since it takes $O(1)$ time to check whether an element in cand is in the neighborhood of pivot. Similarly, the depth of computing cand$_q$ and fini$_q$ at Line 8 and Line 9 are $O(\log n)$. The remaining is the depth of the call of ParTTT at Line 10. Notice that the recursive call of ParTTT continues until there is no further vertex to add for expanding K, and this depth can be at most the size of the maximum clique which is M because, at each recursive call of ParTTT, the size of K increases by 1. Thus, the overall depth of ParTTT is $O(M \log n)$. □

COROLLARY 1. *Using P parallel processors, which are shared-memory,* ParTTT *(Algorithm 3) is a parallel algorithm for MCE and can achieve a worst case parallel time of* $O\left(\frac{3^{n/3}}{M \log n} + P\right)$ *using P parallel processors. This is work-optimal and work-efficient as long as* $P = O(\frac{3^{n/3}}{M \log n})$.

PROOF. The parallel time follows Brent's theorem [2], which states that the parallel time using P processors is $O(w/d + P)$, where w and d are the work and the depth of the algorithm respectively. If the number of processors $P = O\left(\frac{3^{n/3}}{M \log n}\right)$, then using Lemma 2, the parallel time

is $O\left(\max\{\frac{3^{n/3}}{P}, M\log n\}\right) = O\left(\frac{3^{n/3}}{P}\right)$. The total work across all processors is $O(3^{n/3})$, which is worst-case optimal, since the size of the output can be as large as $3^{n/3}$ maximal cliques (Moon and Moser [41]). □

4.2 Algorithm ParMCE

While ParTTT is a theoretically work-efficient parallel algorithm, we note that its runtime can be further improved. While the worst case work complexity of ParPivot matches that of the pivoting routine in TTT, in practice, the work in ParTTT can be higher, since computation of $cand_q$ and $fini_q$ has additional and growing overhead as the sizes of the $cand_q$ and $fini_q$ increase. This can result in a lower speedup than the theoretically expected one.

We set out to improve on this to derive a more efficient parallel implementation through a more selective use of ParPivot. In this way, the cost of pivoting can be reduced by carefully choosing many pivots in parallel instead of a single pivot element as in ParTTT at the beginning of the algorithm. We first note that the cost of ParPivot is the highest during the iteration when set K (the clique in the search space) is empty. During this iteration, the number of vertices in $cand \cup fini$ can be high, as large as the number of vertices in the graph. To improve upon this, we can perform the first few steps of pivoting, when K is empty, using a sequential algorithm. Once set K contains at least one element, the number of the vertices in $cand \cup fini$ is decremented to no more than the size of the intersection of neighborhoods of all vertices in K, which is typically a number much smaller than the number of vertices in the graph (this number is smaller than the smallest degree of a vertex in K). Problem instances with K, assigned to a single vertex, can be seen as sub-problems, and, on each of these sub-problems, the overhead of ParPivot computation is much smaller since the number of vertices that have to be dealt with is also much smaller.

Based on this observation, we present a parallel algorithm ParMCE which works as follows. The algorithm can be viewed as considering, for each vertex $v \in V(G)$, a subgraph G_v that is induced by the vertex v and its neighborhood $\Gamma_G(v)$. It enumerates all maximal cliques from each subgraph G_v in parallel using ParTTT. While processing sub-problem G_v, it is important to not enumerate maximal cliques that are being enumerated elsewhere, in other sub-problems. To handle this, the algorithm considers a specific ordering of all vertices in V such that v is the least ranked vertex in each maximal clique enumerated from G_v. Subgraph G_v for each vertex v is handled in parallel – these subgraphs need not be processed in any particular order. However, the ordering allows us to populate the cand and fini sets accordingly such that each maximal clique is enumerated in exactly one sub-problem. The order in which the vertices are considered is defined by a "rank" function **rank**, which indicates the position of a vertex in the total order. This global ordering on vertices has impact on the total work of the algorithm, as well as the load balance of the distribution of workloads across sub-problems.

Load Balancing: Notice that the sizes of the subgraphs G_v may vary widely because of two reasons: (1) The subgraphs themselves may be of different sizes, depending on the vertex degrees. (2) The number of maximal cliques and the sizes of the maximal cliques containing v can vary widely from one vertex to another. Clearly, the sub-problems that deal with a large number of maximal cliques or maximal cliques of a large size are more computationally expensive than others.

In order to maintain the size of the sub-problems approximately balanced, we use an idea from PECO [55], where we choose the rank function on the vertices in such a way that for any two vertices v and w, **rank**$(v) > $ **rank**(w) if the complexity of enumerating maximal cliques from G_v is higher than the complexity of enumerating maximal cliques from G_w. Indeed, by giving a higher rank to v than w, we are decreasing the complexity of the sub-problem G_v since the sub-problem at G_v need not enumerate maximal cliques that involve any vertex whose rank is less than v. Therefore, the higher

the rank of vertex v, the lower is its "share" (of maximal cliques it belongs to) of maximal cliques in G_v. We use this idea for approximately balancing the workload across sub-problems. The additional enhancements in ParMCE when compared with the idea from PECO are as follows: (1) In PECO, the algorithm is designed for distributed memory such that the subgraphs and sub-problems have to be explicitly copied across the network. (2) In ParMCE, the vertex specific sub-problem, dealing with G_v is itself handled through a parallel algorithm, ParTTT, while, in PECO, the sub-problem for each vertex was handled through a sequential algorithm.

Note that it is computationally expensive to accurately count the number of maximal cliques within G_v, and, hence, it is not possible to compute the rank of each vertex exactly according to the complexity of handling G_v. Instead, we estimate the cost of handling G_v using some easy-to-evaluate metrics on the subgraphs. In particular, we consider the following:

- **Degree Based Ranking:** For vertex v, define $\mathsf{rank}(v) = (d(v), id(v))$ where $d(v)$ and $id(v)$ are degree and identifier of v, respectively. For two vertices v and w, $\mathsf{rank}(v) > \mathsf{rank}(w)$ if $d(v) > d(w)$ or $d(v) = d(w)$ and $id(v) > id(w)$, $rank(v) < rank(w)$ otherwise.
- **Triangle Count Based Ranking:** For vertex v, define $\mathsf{rank}(v) = (t(v), id(v))$ where $t(v)$ is the number of triangles, which vertex v is a part of. Note that ranking based on triangle count is more expensive to compute than degree based ranking but may yield a better estimate of the complexity of maximal cliques within G_v.[2]
- **Degeneracy Based Ranking [18]:** For a vertex v, define $\mathsf{rank}(v) = (degen(v), id(v))$ where $degen(v)$ is the degeneracy of a vertex v. A vertex v has degeneracy number k when it belongs to a k-core but no $(k + 1)$-core, where a k-core is a maximal induced subgraph such that the minimum degree of each vertex in the subgraph is k. A computational overhead of using this ranking is due to computing the degeneracy of vertices which takes $O(n + m)$ time, where n is the number of vertices and m is the number of edges.

The different implementations of ParMCE using degree, triangle, and degeneracy based rankings are called as ParMCEDegree, ParMCETri, ParMCEDegen respectively.

Algorithm 4: ParMCE(G)

Input: G - the input graph.
Output: $C(G)$ - a set of all maximal cliques of G.

1 **for** $v \in V(G)$ **do in parallel**
2   Create G_v, the subgraph of G induced by $\Gamma_G(v) \cup \{v\}$
3      $K \leftarrow \{v\}$, cand $\leftarrow \phi$, fini $\leftarrow \phi$
4   **for** $w \in \Gamma_G(v)$ **do in parallel**
5   **if** $rank(w) > rank(v)$ **then** cand $\leftarrow$ cand $\cup \{w\}$
6   **else** fini $\leftarrow$ fini $\cup \{w\}$
7   ParTTT(G_v, K, cand, fini)

---

5 PARALLEL MCE ALGORITHM ON A DYNAMIC GRAPH

When the graph changes over time due to addition of edges, the maximal cliques of the updated graph also change. The update in the set of maximal cliques consists of (1) The set of new maximal cliques – the maximal cliques that are newly formed (2) The set of subsumed cliques – maximal cliques of the original graph that are subsumed by the new maximal cliques. The combined set of

---

[2]A triangle is a cycle of length three.

(a) Input Graph G **(b)** A new edge (e, d) **(c)** Three new edges: $(a, c), (a, d), (c, e)$
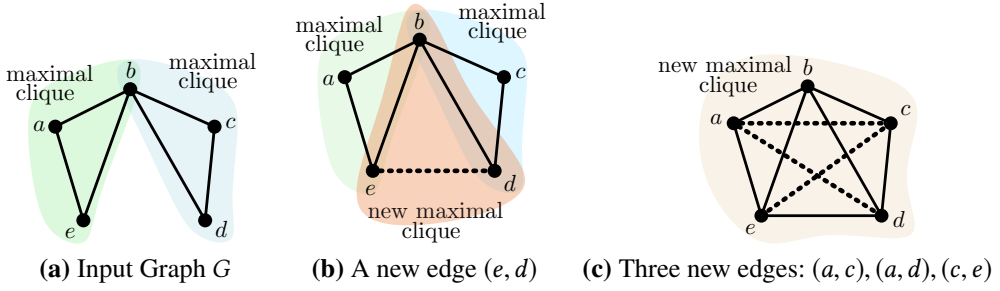
Fig. 3. Maximal cliques change upon addition of new edges. (a) Two subgraphs $\{a, b, e\}$ and $\{b, c, d\}$ are maximal cliques in the original graph G. (b) A new maximal clique (i.e. $\{b, d, e\}$) is created when edge (e, d) is added to G. (c) When three more edges $(a, c), (a, d)$, and (c, e) are added, the entire graph turns into a maximal clique, which subsumes all maximal cliques in prior steps.

new and subsumed maximal cliques is called the set of changes, and the size of this set refers to the size of change in the set of maximal cliques (see Figure 3).

Algorithm 5: ParIMCENew(G, H)

Input: G - the input graph
H - a set of ρ edges being added to G
Output: Cliques in $\Lambda^{new} = C(G + H) \setminus C(G)$

1   $G' \leftarrow G + H$
2   Consider edges of H in an arbitrary order $e_1, e_2, \ldots, e_\rho$
3   **for** $i \leftarrow 1, 2, \ldots, \rho$ **do in parallel**
4      $e \leftarrow e_i = (u, v)$
5      $V_e \leftarrow \{u, v\} \cup \{\Gamma_{G'}(u) \cap \Gamma_{G'}(v)\}$
6      $G'' \leftarrow$ Graph induced by V_e on G'
7      $K \leftarrow \{u, v\}$
8   cand $\leftarrow V_e \setminus \{u, v\}$; fini $\leftarrow \emptyset$
9      $S \leftarrow$ ParTTTExcludeEdges$(G'', K, \text{cand}, \text{fini}, \{e_1, e_2, ..., e_{i-1}\})$
10     $\Lambda^{new} \leftarrow \Lambda^{new} \cup S$

Note that the size of change can be as small as $O(1)$ or as large as exponential in the size of the graph upon addition of a new single edge. For example, consider a graph of size n which is missing a single edge from being a clique. The size of change is only 3 when that missing edge is added to the graph because there will be only one new maximal clique of size n and two subsumed cliques, each of size $(n-1)$. On the other hand, consider a Moon-Moser graph [41] of size n. Addition of a single edge to this graph makes the size of the changes in the order of $O(3^{n/3})$.

In a previous work, we presented a sequential algorithm IMCE [13], which efficiently tackles the problem of updating the set of maximal cliques of a dynamic graph in an incremental model when new edges are added at a time. IMCE consists of FastIMCENewClq for computing new maximal cliques and IMCESubClq for computing subsumed cliques. However, IMCE is still unable to update the set of maximal cliques when the size of change is large. For instance, it takes IMCE around 9.4 hours to update the set of maximal cliques, when the first $90K$ edges of graph **Ca-Cit-HepTh** are added incrementally (with the original graph density 0.01). The high computational cost of IMCE calls for parallel methods.
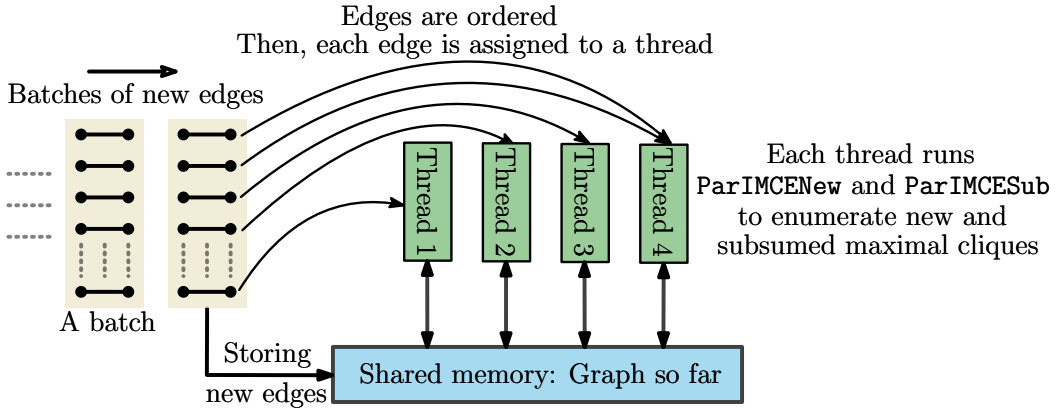
Fig. 4. Our shared-memory set up for processing a dynamic graph.

In this section, we present parallel algorithms for enumerating the set of new and subsumed cliques when an edge set $H = \{e_1, e_2, ..., e_\rho\}$ is added to a graph G. Our parallel algorithms are based on IMCE [13]. In this work, we focus on (1) processing new edges in parallel, (2) enumerating new maximal cliques using ParTTT, and (3) Parallelizing IMCESubClq [13]. First, we describe an efficient parallel algorithm for generating new maximal cliques, i.e. the maximal cliques in $G + H$, which are not present in G and then an efficient parallel algorithm for generating subsumed maximal cliques, i.e. the cliques which are maximal in G but not maximal in $G + H$. We present a shared-memory parallel algorithm ParIMCE for the incremental maintenance of maximal cliques. ParIMCE consists of (1) algorithm ParIMCENew for enumerating new maximal cliques and (2) algorithm ParIMCESub for enumerating subsumed cliques. A brief description of the algorithms is discussed in Table 2. Figure 4 also sketches the parallel shared-memory setup for enumerating maximal cliques in a dynamic graph upon addition of new batches.

5.1 Parallel Enumeration of New Maximal Cliques

Here, we present a parallel algorithm ParIMCENew for enumerating the set of new maximal cliques when a set of edges is added to the graph. The idea is that we iterate over new edges in parallel and at each (parallel) iteration we construct a subgraph of the original graph and enumerate the set of all maximal cliques within the subgraph. We present an efficient parallel algorithm ParTTTExcludeEdges (Algorithm 6) for this enumeration. The description of ParIMCENew is presented in Algorithm 5.

Note that ParIMCENew is based upon an existing sequential algorithm FastIMCENewClq [13] for enumerating new maximal cliques using TTTExcludeEdges (Algorithm 8) [13], which lists all new maximal cliques without any duplication. ParTTTExcludeEdges avoids the duplication in maximal clique enumeration similar to the technique used in TTTExcludeEdges and uses a parallelization approach similar to of ParTTT. More specifically, ParTTTExcludeEdges follows a global ordering

Table 2. Brief description of the incremental algorithms in this work.

Objective	Sequential Algorithm [13]	Parallel Algorithm (this work)	Overview of Parallel Algorithms
Enumerating new maximal cliques	FastIMCENewClq	ParIMCENew	(1) Process new edges in parallel. (2) Enumerate maximal cliques using ParTTTExcludeEdges.
Enumerating subsumed cliques	IMCESubClq	ParIMCESub	(1) Generate candidates in parallel (executing inner for loop of IMCESubClq in parallel). (2) Process each candidate in parallel (executing candidate processing step of IMCESubClq in parallel).

Algorithm 6: ParTTTExcludeEdges($G, K,$ cand, fini, $\mathcal{E}$)

Input: G - the input graph
K - Set of vertices forming a clique
cand - a set of vertices that may extend K
fini - vertices that are not used to extend K, but are connected to all vertices of K
$\mathcal{E}$ - a set of edges to ignore

1 **if** (cand $= \emptyset$) & (fini $= \emptyset$) **then**
2   Output K // K is a maximal clique
3   **return**
4 pivot $\leftarrow$ ParPivot($G,$ cand, fini)
5 ext$[1..\kappa] \leftarrow$ cand $- \Gamma_G($pivot$)$ // in parallel
6 **for** $i \in [1..\kappa]$ **do in parallel**
7      $q \leftarrow$ ext$[i]$
8      $K_q \leftarrow K \cup \{q\}$
9   **if** $K_q \cap \mathcal{E} \neq \emptyset$ **then**
10   **return**
11   cand$_q \leftarrow$ intersect(cand $\setminus$ ext$[1..i-1], \Gamma_G(q))$
12   fini$_q \leftarrow$ intersect(fini $\cup$ ext$[1..i-1], \Gamma_G(q))$
13   ParTTTExcludeEdges($G, K_q,$ cand$_q,$ fini$_q, \mathcal{E}$)

---

of the new edges to avoid redundancy in the enumeration process. Note that the correctness of ParIMCENew is followed by the correctness of the sequential algorithm FastIMCENewClq. Following lemma shows the work efficiency and depth of ParIMCENew.

LEMMA 3. *Given a graph G and a new edge set H,* ParIMCENew *is work-efficient, i.e, the total work is of the same order of the time complexity of* FastIMCENewClq*. The depth of* ParIMCENew *is $O(\Delta^2 + M \log \Delta)$, where Δ is the maximum degree and M is the size of a maximum clique in $G + H$.*

We prove this lemma in Appendix A by showing the equivalence of the operations of FastIMCENewClq and ParIMCENew.

5.2 Parallel Enumeration of Subsumed Cliques

In this section, we present a parallel algorithm ParIMCESub based on the sequential algorithm IMCESubClq [13] for enumerating subsumed cliques. In ParIMCESub, we perform parallelization in the following order: (1) Removing a single new edges from all the candidates in parallel and (2) Checking for the candidacy of the subsumed cliques in parallel. We present ParIMCESub in Algorithm 7. In the following lemma, we show the work efficiency and depth of ParIMCESub:

LEMMA 4. *Given a graph G and a new edge set H,* ParIMCESub *is work-efficient; the total work is of the same order of the time complexity of* IMCESubClq*. The depth of* ParIMCENew *is $O(min\{M^2, \rho\})$ for processing each new maximal clique, where M is the size of a maximum clique in $G + H$ and ρ is the size of H.*

PROOF. First, note that the procedure of ParIMCESub is exactly the same as the procedure of IMCESubClq except for the parallel loops at Line 6 and Line 13 of ParIMCESub whereas these loops are sequential in IMCESubClq. Since all the computations in ParIMCESub is exactly the same as the computations in IMCESubClq, except for the loop parallelization, ParIMCESub is work-efficient.

Algorithm 7: ParIMCESub(G, H, C, Λ^{new})

Input: G - the input Graph

H - an edge set being added to G

C - a set of maximal cliques in G

Λ^{new} - a set of new maximal cliques in $G + H$

Output: All cliques in $\Lambda^{del} = C(G) \setminus C(G + H)$

1 $\Lambda^{del} \leftarrow \emptyset$

2 **for** $c \in \Lambda^{new}$ **do in parallel**

3 $\quad$ $S \leftarrow \{c\}$

4 $\quad$ **for** $e = (u, v) \in E(c) \cap H$ **do**

5 $\quad\quad$ $S' \leftarrow \phi$

6 $\quad\quad$ **for** $c' \in S$ **do in parallel**

7 $\quad\quad\quad$ **if** $e \in E(c')$ **then**

8 $\quad\quad\quad\quad$ $c_1 = c' \setminus \{u\}$; $c_2 = c' \setminus \{v\}$

9 $\quad\quad\quad\quad$ $S' \leftarrow S' \cup c_1$; $S' \leftarrow S' \cup c_2$

10 $\quad\quad\quad$ **else**

11 $\quad\quad\quad\quad$ $S' \leftarrow S' \cup c'$

12 $\quad\quad$ $S \leftarrow S'$

13 $\quad$ **for** $c' \in S$ **do in parallel**

14 $\quad\quad$ **if** $c' \in C$ **then**

15 $\quad\quad\quad$ $\Lambda^{del} \leftarrow \Lambda^{del} \cup c'$

16 $\quad\quad\quad$ $C \leftarrow C \setminus c'$

For proving the parallel depth of ParIMCESub, first note that all the elements of S at Line 6 of ParIMCESub are processed in parallel, and the total cost of executing Lines 7 to 11 is $O(1)$. In addition, the depth of the operation at Line 12 of ParIMCESub is $O(1)$ using the concurrent hashtable. Therefore, the overall depth of the procedures from Line 4 to Line 12 is the number of new edges in a new maximal clique c, processed at Line 2 of ParIMCESub which is $O(\min\{M^2, \rho\})$. Next, the depth of Lines 14 to 16 is $O(1)$ because it only takes $O(1)$ to execute Lines 15 and 16 using Theorem 3.1. As a result, for each new maximal clique, the depth of the procedure for enumerating all subsumed cliques within the new maximal clique is $O(\min\{M^2, \rho\})$. □

5.3 Decremental Case

We have so far discussed incremental maintenance of new and subsumed maximal cliques when the stream only contains new edges. We note that our algorithm can also support the deletion of edges through a reduction to the incremental case, this is similar to the methods used for sequential algorithms. Please see Sections 4.4 and 4.5 of [13] for further details.

6 EVALUATION

In this section, we experimentally evaluate the performance of our shared-memory parallel static (ParTTT and ParMCE) and dynamic (ParIMCE) algorithms for MCE on static and dynamic (real world and synthetic) graphs to show the parallel speedup and scalability of our algorithms over efficient sequential algorithms TTT and IMCE respectively. We also compare our algorithms with state-of-the-art parallel algorithms for MCE to show that the performance of our algorithm has substantially improved over the prior works. We run the experiments on a computer configured with Intel Xeon

(R) CPU E5-4620 running at 2.20GHz , with 32 physical cores (4 NUMA nodes each with 8 cores) and 1TB RAM.

6.1 Datasets

We use eight different real-world static and dynamic networks from publicly available repositories KONECT [32], SNAP [33], and Network Repository [47] for doing the experiments. Dataset statistics are summarized in Table 3. For our experiments, we convert these networks to simple undirected graphs by removing self-loops, edge weights, parallel edges, and edge directions.

We consider networks **DBLP-Coauthor**, **As-Skitter**, **Wikipedia**, **Wiki-Talk**, and **Orkut** for the evaluation of the algorithms on static graphs and networks **DBLP-Coauthor**, **Flickr**, **Wikipedia**, **LiveJournal**, and **Ca-Cit-HepTh** for the evaluation of the algorithms on dynamic graphs.

DBLP-Coauthor shows the collaboration of authors of papers from DBLP computer science bibliography. In this graph, vertices represent authors, and there is an edge between two authors if they have published a paper [47]. **As-Skitter** is an Internet topology graph which represents the autonomous systems, connected to each other on the Internet [32]. **Wikipedia** is a network of English Wikipedia in 2013, where vertices represent pages in English Wikipedia, and there is an edge between two pages p and q if there is a hyperlink in page p to page q [33]. **Wiki-Talk** contains users of Wikipedia as vertices where each edge between two users in this graph indicates that one of the users has edited the "page talk" of the other user on Wikipedia [33]. **Orkut** is a social network where each vertex represents a user in the network, and there is an edge if there is a friendship relation between two users [32]. Similar to **Orkut**, **Flickr** and **LiveJournal** are also social networks where a vertex represents a user and an edge represents the friendship between two users. **Ca-Cit-HepTh** is a citation network of high energy physics theory where a vertex represents a paper and there is an edge between paper u and paper v if u cites v.

For the evaluation of ParTTT and ParMCE, we give the entire graph as input to the algorithm in the form of an edge list and for the evaluation of the algorithms on dynamic graphs, we start with an empty graph that contains all vertices but no edges, and, at a time, we add a set of edges in an increasing order of timestamps for computing the changes in the set of maximal cliques. For ParIMCE, we use real dynamic graphs, where each edge has a timestamp. In addition, we evaluate ParIMCE on **LiveJournal**, which is a static graph. We convert this graph to a dynamic graph through randomly permuting the edges and processing edges in that ordering.

To understand the datasets better, we illustrated the frequency distribution of sizes of maximal cliques in Figure 5. The size of maximal cliques in **DBLP-Coauthor** can be as large as 100 vertices. Although the number of such large maximal cliques are small, the depth of the search space might increase exponentially for discovering such large maximal cliques. As shown in Figure 5, most of the graphs contain more than tens of millions of maximal cliques. For example, **Orkut** contains more than two billion maximal cliques. That being said, the depth and breadth of the search space makes MCE a challenging problem to solve.

6.2 Implementation of the Algorithms

In our parallel implementations of ParTTT, ParMCE, and ParIMCE, we utilize **parallel_for** and **parallel_for_each**, developed by the Intel TBB parallel library [4]. We also utilize **concurrent_hash_map** for atomic operations on hashtable. We use C++11 standard for the implementation of the algorithms and compile all the sources using Intel ICC compiler version 18.0.3 with optimization level '-O3'. We use the command 'numactl -i all' for balancing the memory in a NUMA machine. System level load balancing is performed using a dynamic work stealing scheduler [4] in TBB.
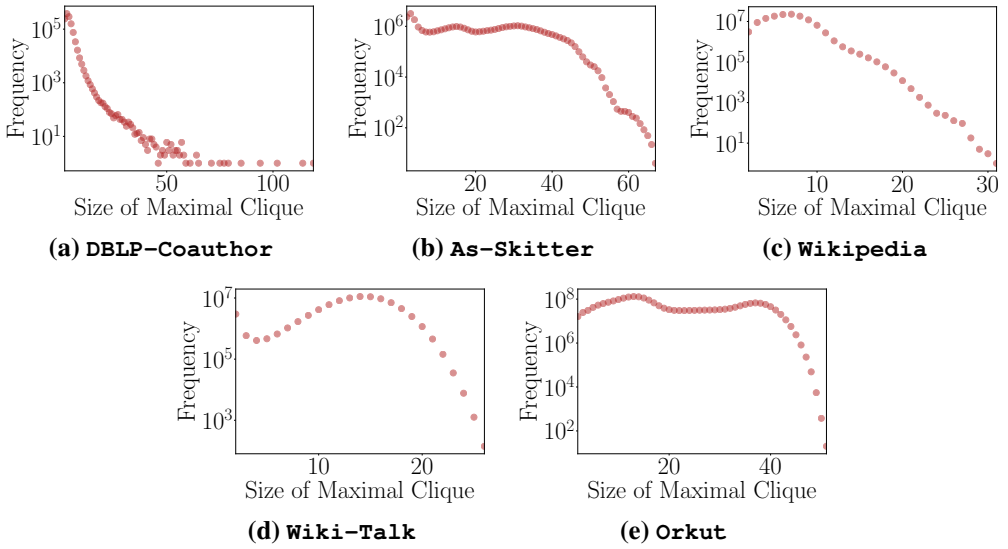
(a) DBLP-Coauthor **(b) As-Skitter** **(c) Wikipedia**

(d) Wiki-Talk **(e) Orkut**

Fig. 5. Frequency distribution of sizes of maximal cliques across different input graphs.

To compare with prior works on MCE, we implement some of them such as [16, 18, 55, 56, 65] in C++, and we use the executable of the C++ implementations for the rest of the algorithms (GreedyBB [48], Hashing [34], and GP [59]), provided by the respective authors. See Subsection 6.4 for more details.

We compute the degeneracy number and triangle count for each vertex using sequential procedures. While the computation of per-vertex triangle counts and the degeneracy ordering could be potentially parallelized, implementing a parallel method to rank vertices based on their degeneracy number or triangle count is itself a non-trivial task. We decided not to parallelize these routines since the degeneracy- and triangle-based ordering did not yield significant benefits when compared with degree-based ordering, whereas the degree-based ordering is trivially available, without any additional computation.

We assume that the entire graph is available in a shared global memory. The Total Runtime (TR) of ParMCE consists of (1) Ranking Time (RT): the time required to rank vertices of the graph based on the ranking metric used in the algorithm, i.e. degree, degeneracy number, or triangle count of

Table 3. Static and Dynamic Networks, used for evaluation, and their properties. For some of the graphs (e.g. **Flickr** and **Ca-Cit-HepTh**) used for evaluating the incremental algorithms, we could not report the information about maximal cliques as they did not finish within 10 days, even using parallel algorithms.

Dataset	#Vertices	#Edges	#Maximal Cliques	Average Size of Maximal Cliques	Size of Largest Clique
DBLP-Coauthor	1,282,468	5,179,996	1,219,320	3	119
Orkut	3,072,441	117,184,899	2,270,456,447	20	51
As-Skitter	1,696,415	11,095,298	37,322,355	19	67
Wiki-Talk	2,394,385	4,659,565	86,333,306	13	26
Wikipedia	1,870,709	36,532,531	131,652,971	6	31
LiveJournal	4,033,137	27,933,062	38,413,665	29	214
Flickr	2,302,925	22,838,276	> 400 billion	-	-
Ca-Cit-HepTh	22,908	2,444,798	> 400 billion	-	-

vertices and (2) Enumeration Time (ET): the time required to enumerate all maximal cliques. For `ParMCEDegen` and `ParMCETri` algorithms, the runtime of ranking is also reported. Figures 6 and 7 show the parallel speedup (with respect to the runtime of TTT) and the enumeration time of `ParMCE` using different vertex ordering strategies. Table 5 shows the breakdown of the Total Runtime (TR) into Ranking Time (RT) and Enumeration Time (ET).

The runtime of `ParIMCE` consists of (1) the computation time of `ParIMCENew` and (2) the computation time of `ParIMCESub`. In the implementation of `ParIMCENew`, we follow the design of `ParMCE` and instead of executing `ParTTTExcludeEdges` on the entire (sub)graph for an edge as in Line 9 of `ParIMCENew`, we run `ParTTTExcludeEdges` on per-vertex sub-problems in parallel. For dealing with load balance, we use degree based ordering of the vertices of G'' in creating the sub-problems. We decided to implement `ParIMCENew` in this way as we observed a significant improvement in the performance of `ParMCE` over `ParTTT` when we used a degree-based vertex ordering. For experiment on dynamic graphs, we use the batch size of 1000 edges for all the graphs except for an extremely dense graph `Ca-Cit-HepTh` (with original graph density 0.01) where we use batch size of 10.

6.3 Discussion of the Results

Here, we empirically evaluate our parallel algorithms and prior methods. First, we show the parallel speedup (with respect to the sequential algorithms) and scalability (with respect to the number of cores) of our algorithms. Next, we compare our works with the state-of-the-art sequential and parallel algorithms for MCE. The results demonstrate that our solutions are faster than prior methods with significant margins.

6.3.1 Parallel MCE on Static Graphs. The total runtime of the parallel algorithms with 32 threads are shown in Table 4. We observe that `ParTTT` achieves a speedup of **5x**-**14x** over the sequential algorithm TTT. The three versions of `ParMCE`, i.e. `ParMCEDegree`, `ParMCEDegen`, `ParMCETri`, achieve a speedup of **15x**-**21x** with 32 threads, when we consider the runtime of enumeration task alone. The speedup ratio decreases in `ParMCEDegen` and `ParMCETri` if we include the time taken by ranking strategies (See Figure 6).

The runtime of `ParTTT` is higher than the runtime of `ParMCE`, due to a heavier cumulative overhead of pivot computation and of processing sets cand and fini in `ParTTT`. For example, in `DBLP-Coauthor` graph, when we run `ParTTT`, the cumulative overhead of computing pivot is 248 seconds, and the cumulative overhead of updating cand and fini is 38 seconds while, in `ParMCE`, these runtimes are 156 and 21 seconds, respectively. This helps `ParMCE` accelerate the entire process, which achieves 2x speedup over `ParTTT`.

Impact of vertex ordering on overall performance of `ParMCE`**.** Here, we study the impact of different vertex ordering strategies, i.e. degree, degeneracy, and triangle count, on the overall performance of `ParMCE`. Table 5 presents the total computation time when we use different orderings.

Table 4. Runtime (in sec.) of TTT, `ParTTT`, and `ParMCE` with different vertex orderings on 32 cores. The numbers exclude the time taken for vertex ordering. Note that the best algorithm, which uses degree based vertex ordering, has zero additional cost for computing the vertex ordering.

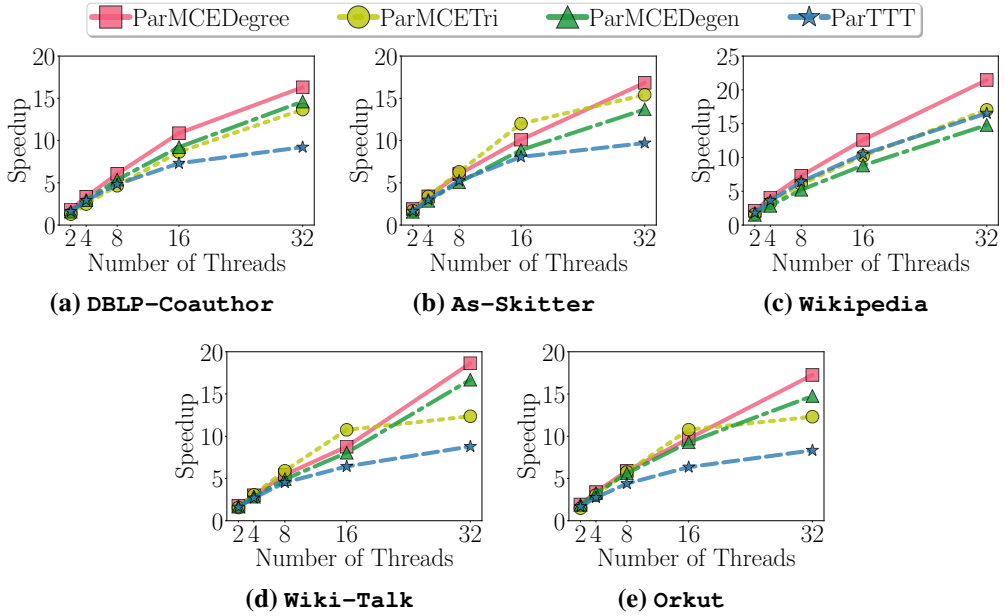| **Dataset**	TTT	ParTTT	ParMCEDegree	ParMCEDegen	ParMCETri
DBLP-Coauthor	42	4	2	3	3
Orkut	28923	3472	1676	2350	1959
As-Skitter	660	68	39	43	48
Wiki-Talk	961	109	52	78	58
Wikipedia	2646	160	123	155	179

Fig. 6. Parallel speedup when compared with TTT (sequential algo. due to Tomita et al. [56]) as a function of the number of threads.
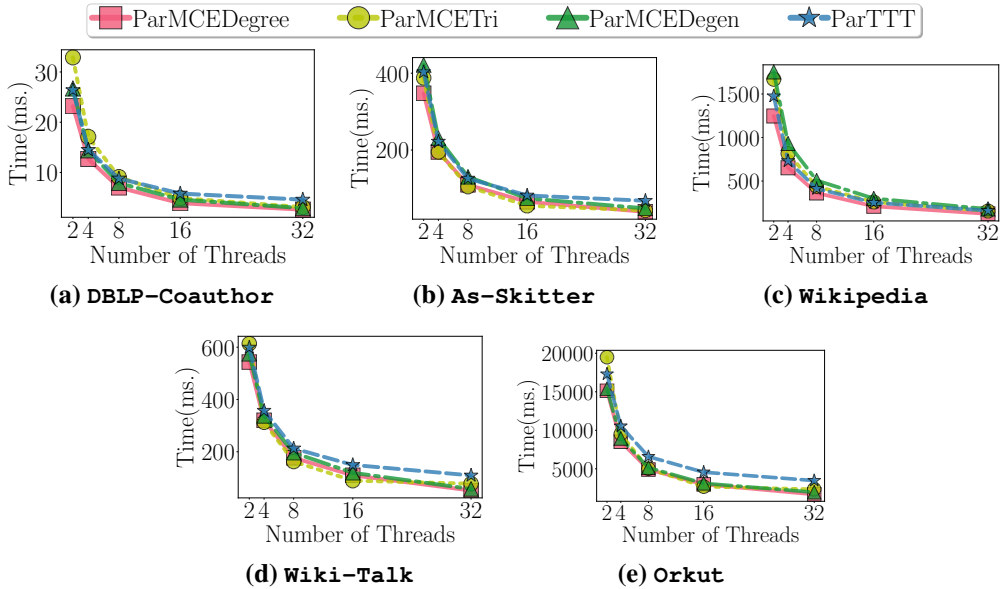

Fig. 7. The total runtime of parallel algorithms ParMCEDegree, ParMCETri, ParMCEDegen, and ParTTT in milliseconds as a function of the number of threads.

We observe that degree-based ordering (ParMCEDegree) in most cases is the fastest strategy for clique enumeration, even when we do not consider the computation time for generating degeneracy- and triangle-based orderings. If we add up the runtime for the ordering step, *degree-based ordering is obviously better than degeneracy- or triangle-based orderings* since degree-based ordering is

Fig. 8. Parallel speedup of `ParIMCE` over `IMCE` as a function of the size of the change in the set of maximal cliques. The size of the change is measured by the total number of new maximal cliques and subsumed maximal cliques when a batch of edges is added to the graph.
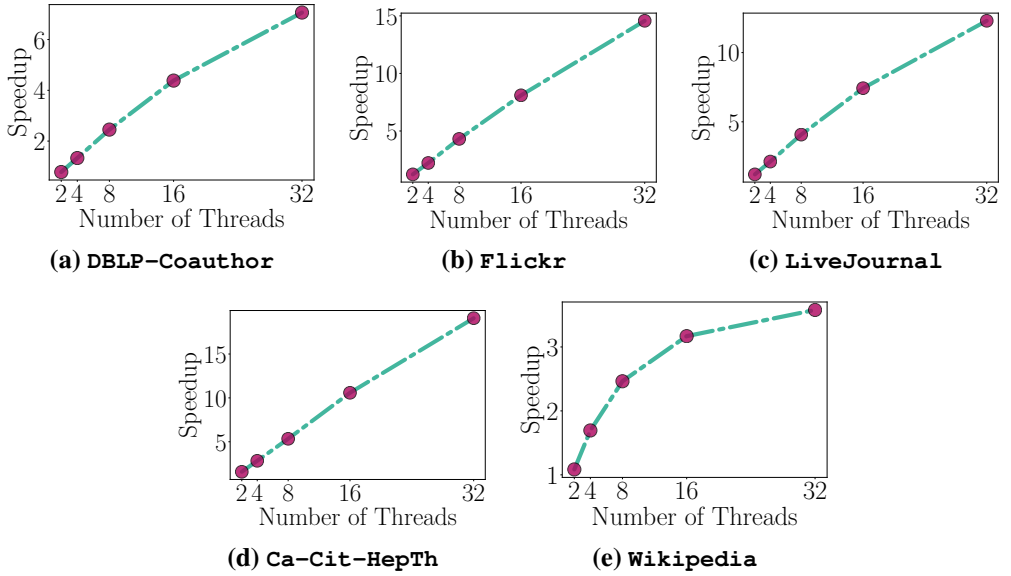

Fig. 9. Parallel speedup of `ParIMCE` over `IMCE` as a function of number of threads, using the cumulative time of `ParIMCE` and of `IMCE` for processing all batches of edges.

available for free when the input graph is read, while the degeneracy- and triangle-based orderings require additional computations.

Scaling up with the degree of parallelism. As the number of threads (and the degree of parallelism) increases, the runtime of `ParMCE` and of `ParTTT` decreases. Figure 6 presents the speedup factor of `ParMCE` over the state-of-the-art sequential algorithm, i.e. TTT, as a function of the number of threads, and Figure 7 presents the runtime of `ParMCE`. `ParMCEDegree` achieves a speedup of more than **15x** on all graphs, when 32 threads are used.

6.3.2 Parallel MCE on Dynamic Graphs. The cumulative runtime of `IMCE` and `ParIMCE` are presented in Table 6 which shows that the speedup achieved by `ParIMCE` is **3.6x-19.1x** over `IMCE`. This wide spectrum of speedups is mainly due to the variations in the size of the changes in the set of maximal cliques (number of new maximal cliques + number of subsumed maximal cliques) in the course of the incremental computation which can be observed in Figure 8. From this plot, we can see that the speedup increases with the increase in the size of the changes in the set of maximal cliques. This trend is as expected because the effect of parallelism will be prominent whenever the number of parallel tasks will become sufficiently large. This happens when the number of new and subsumed maximal cliques are large.

Scalability. The degree of parallelism increase with the increase in the number of threads. From Figure 9 we can see that the speedup increases linearly with the number of threads. This behavior shows the scalability of our parallel algorithm `ParIMCE`. When the size of the changes will become large, scalability will become prominent because otherwise, most of the processors will remain idle when there will not be large amount of parallel tasks to fully utilize all the available processors. This is observed in **Wikipedia** (Figure 9) where the cumulative size of change is relatively small. Additionally, the speedup observed on different input graphs vary with the similar size of change. For example, as we can see in Figure 8, **LiveJournal** gives more speedup than **Wikipedia** with the size of change around 10^5. To explain this, we first note that higher parallel speedup is observed when the cost of maintenance (which is proportional to the size of change in the

Table 5. Total Runtime (in sec.) of `ParMCE` with different vertex orderings (using 32 threads). Total Runtime (TR) = Ranking Time (RT) + Enumeration Time (ET).

Dataset	ParMCEDegree	ParMCEDegen			ParMCETri		
		RT	ET	TR	RT	ET	TR
DBLP-Coauthor	3	25	3	28	42	3	45
Orkut	1676	928	2350	3278	2166	1959	4125
As-Skitter	39	41	43	84	122	48	170
Wiki-Talk	52	23	78	101	74	58	132
Wikipedia	123	244	155	399	950	179	1129

Table 6. Cumulative runtime (in sec.) over the incremental computation across all edges, with `IMCE` and `ParIMCE` using 32 threads. The total number of edges that are processed is also presented.

Dataset	#Edges Processed	IMCE	ParIMCE	Parallel Speedup
DBLP-Coauthor	5.1M	6608	933	**7x**
Flickr	4.1M	35238	2416	**14.6x**
Wikipedia	36.5M	9402	2614	**3.6x**
LiveJournal	19.2M	30810	2497	**12.3x**
Ca-Cit-HepTh	93.8K	33804	1767	**19.1x**

set of maximal cliques) on a single batch is high. Following this, we see that in adding around 19K batches of new edges (with batch size of 1000 edges) starting from the empty graph, **LiveJournal** has more frequent (around 2000) size of change in the order of 10^5 compared to **Wikipedia** (around 100 size of changes in the order of 10^5).

6.4 Comparison with prior work

We compare the performance of ParMCE with prior sequential and parallel algorithms for MCE. We consider the following sequential algorithms: GreedyBB due to Segundo et al. [48], TTT due to Tomita et al. [56], and BKDegeneracy due to Eppstein et al. [18]. For the comparison with parallel algorithm, we consider algorithm CliqueEnumerator due to Zhang et al. [65], Peamc due to Du et al. [16], PECO due to Svendsen et al. [55], and the most recent parallel algorithm Hashing due to Lessley et al. [34]. The parallel algorithms CliqueEnumerator, Peamc, and Hashing are designed for the shared-memory model, while PECO is designed for distributed memory. We modified PECO to work with shared-memory, by reusing the method for sub-problem construction and eliminating the need to communicate subgraphs by storing a single copy of the graph in shared-memory. We consider three different ordering strategies for PECO, which we call PECODegree, PECODegen, and PECOTri. The comparison of performance of ParMCE with PECO is presented in Table 7. We note that ParMCE is significantly better than that of PECO, no matter which ordering strategy was considered. We also compare the performance of ParMCE with a recent shared-nothing parallel algorithm GP to show that ParMCE outperforms GP with resources equivalent to our shared memory setting.

The comparison of ParMCE with other shared-memory algorithms Peamc, CliqueEnumerator, and Hashing is shown in Table 8. The performance of ParMCE is seen to be much better than that of any of these prior shared-memory parallel algorithms. For the graph **DBLP-Coauthor**, Peamc did not finish within 5 hours whereas ParMCE takes at most around 50 secs for enumerating 1.2 million maximal cliques. The poor running time of Peamc is due to two following reasons: (1) The algorithm does not apply efficient pruning techniques such as pivoting, used in TTT, and (2) The method to determine the maximality of a clique in the search space is not efficient. The CliqueEnumerator algorithm runs out of memory after a few minutes. The reason is that CliqueEnumerator maintains a bit vector for each vertex that is as large as the size of the input graph and, additionally, needs to store intermediate non-maximal cliques. For each such non-maximal clique, it is required to maintain a bit vector of length equal to the size of the vertex set of the original graph. Therefore, in CliqueEnumerator, a memory issue is inevitable for a graph with millions of vertices.

A recent parallel algorithm in the literature, Hashing, also has a significant memory overhead and ran out of memory on the input graphs that we considered. The reason for its high memory requirement is that Hashing enumerates intermediate non-maximal cliques before finally outputting maximal cliques. The number of such intermediate non-maximal cliques may be very large, even for graphs with few number of maximal cliques. For example, a maximal clique of size c contains $2^c - 1$ non-maximal cliques.

Table 7. Comparison of parallel runtime (in sec.) of ParMCE, excluding the time for computing vertex ranking, with a version of PECO that is modified to use shared-memory with 32 threads. Three different variants are considered for each algorithm based on the vertex ordering strategy.

Dataset	PECODegree	ParMCEDegree	PECODegen	ParMCEDegen	PECOTri	ParMCETri
DBLP-Coauthor	6.4	2.6	6.9	3.1	6.8	2.9
Orkut	2050.7	1676.4	2183.4	2350	2361.9	1959.3
As-Skitter	261.5	39.2	331.8	42.8	260.9	48.2
Wiki-Talk	1729.7	51.6	1728.2	77.8	1720	57.6
Wikipedia	8982.5	123.3	9110.4	155.3	8938	178.8

Table 8. Comparison of runtimes (in sec.) of ParMCE with prior works on shared-memory algorithms for MCE (with 32 threads).

Dataset	ParMCEDegree	Hashing	CliqueEnumerator	Peamc
DBLP-Coauthor	2.6	Out of memory in 3 min.	Out of memory in 10 min.	Not complete in 5 hours.
Orkut	1676.4	Out of memory in 7 min.	Out of memory in 20 min.	Not complete in 5 hours.
As-Skitter	39.2	Out of memory in 5 min.	Out of memory in 10 min.	Not complete in 5 hours.
Wiki-Talk	51.6	Out of memory in 10 min.	Out of memory in 20 min.	Not complete in 5 hours.
Wikipedia	123.3	Out of memory in 10 min.	Out of memory in 20 min.	Not complete in 5 hours.

Table 9. Speedup factor of ParMCEDegree over GP and PECODegree. A speedup factor greater than 1 indicates that our algorithm ParMCEDegree is faster than the prior methods. 8★ indicates that 8 threads for ParMCEDegree and 8 MPI nodes for GP are used (★ next to other numbers is interpreted similarly). ✗ indicates that GP runs out of memory.

Dataset	Speedup factor of ParMCEDegree over GP					Speedup factor of ParMCEDegree over PECODegree				
	2★	4★	8★	16★	32★	2 threads	4 threads	8 threads	16 threads	32 threads
DBLP-Coauthor	0.83	1.21	1.76	2.9	4.1	2.5	2.85	2.4	2.33	2.5
Orkut	1.92	1.68	1.73	✗	✗	1.2	1.11	1.04	0.85	1.48
As-Skitter	1.66	1.64	1.56	1.71	1.35	4.47	4.33	3.92	4.1	3.26
Wiki-Talk	1.38	1.32	1.15	1.3	0.84	2.48	2.37	4.65	9.07	10.9
Wikipedia	1.5	1.45	1.44	1.86	1.69	9.6	9.74	14.07	25.34	29.6

Next, we compare the performance of ParMCE with that of sequential algorithms BKDegeneracy and a recent sequential algorithm GreedyBB – results are in Table 10. For large graphs, the performance of BKDegeneracy is almost similar to TTT whereas GreedyBB performs much worse than TTT. Since our ParMCE algorithm outperforms TTT, we can conclude that ParMCE is significantly faster than other sequential algorithms.

Next, we compare with GP [59], a recent distributed algorithm for MCE based on MPI. This method first assigns each vertex v to an MPI worker, and then the worker constructs subproblems of vertex v by constructing the sets cand and fini for enumerating the maximal cliques which vertex v is part of. Through an iterative computation, each sub-problem is broken down into smaller sub-problems, till a maximal clique is obtained. Subproblems within an MPI worker might be sent to another worker, where the receiver MPI process is chosen randomly. We compare the runtime performance of our work ParMCEDegree with of GP. Note that ParMCEDegree is a shared-memory method while GP is implemented in a distributed memory model.

We ran GP and ParMCEDegree on a machine configured with two 18-core Intel Skylake 6140 Xeon processors. In every experiment, we use 192GB as the total memory. For a fair comparison, we use the same number of threads for ParMCEDegree as the number of MPI processes for GP. In Table 9, we report the speedup of ParMCEDegree over GP. In most cases ParMCEDegree outperforms GP. In **DBLP-Coauthor**, we observed that GP cannot achieve a better runtime while the number of MPI workers increases. For a better understanding of this case, we measured the runtime performance of exchanged sub-problems among MPI nodes and the runtime of clique enumeration. We observed that the enumeration takes at most one second while the overhead for exchanging sub-problems among

Table 10. Total runtime (sec.) of parallel algorithm ParMCE (with different vertex ranking, with 32 threads) and sequential algorithms BKDegeneracy and GreedyBB.

Dataset	BKDegeneracy	GreedyBB	ParMCEDegree	ParMCEDegen	ParMCETri
DBLP-Coauthor	53.6	Not finish in 30 min.	2.6	28.1	44.3
Orkut	29812.3	Out of memory in 5 min.	1676.4	3278	4125.3
As-Skitter	641.7	Out of memory in 10 min.	39.2	83.8	170.2
Wiki-Talk	1003.2	Out of memory in 10 min.	51.6	100.8	131.2
Wikipedia	2243.6	Out of memory in 10 min.	123.3	399	1128

workers is huge and skewed towards a few MPI nodes, which leads to an unbalanced workload. As shown in Table 9, there are two cases that GP performs faster than ParMCEDegree – in both cases, the difference in runtimes is small, about 8 seconds.

6.5 Summary of Experimental Results

We found that both ParTTT and ParMCE yield significant speedups over the sequential algorithm TTTnearly linear in the number of cores available. ParMCE using the degree-based vertex ranking always performs better than ParTTT. The runtime of ParMCE using degeneracy/triangle count based vertex ranking is sometimes worse than ParTTT due to the overhead of sequential computation of vertex ranking – note that this overhead is not needed in ParTTT. The parallel speedup of ParMCE is better when the input graph has many large sized maximal cliques. Overall, ParMCE consistently outperforms prior sequential and parallel algorithms for MCE. For a dynamic graph we found that ParIMCE consistently yields a substantial speedup over the efficient sequential algorithm IMCE. Further, the speedup of ParIMCE improves as the size of the change (to be enumerated) becomes larger.

7 CONCLUSION

We presented shared-memory parallel algorithms for enumerating maximal cliques from a graph. ParTTT is a work-efficient parallelization of a sequential algorithm due to Tomita et al. [56], and ParMCE is an adaptation of ParTTT that has more opportunities for parallelization and better load balancing. Our algorithms obtain significant improvements compared with the current state-of-the-art on MCE. Our experiments show that ParMCE has a speedup of up to 21x (on a 32 core machine) when compared with an efficient sequential baseline. In contrast, prior shared-memory parallel methods for MCE were either unable to process the same graphs in a reasonable time or ran out of memory. We also presented a shared-memory parallel algorithm ParIMCE that can enumerate the change in the set of maximal cliques when new edges are added to the graph.

Many questions remain open: (1) Can these methods scale to even larger graphs and to machines with larger numbers of cores (2) How can one adapt these methods to other parallel systems such as a cluster of computers with a combination of shared and distributed memory or GPUs?

REFERENCES

[1] David Avis and Komei Fukuda. 1993. Reverse Search for Enumeration. *Discrete Applied Mathematics* 65 (1993), 21–46.

[2] Guy E Blelloch and Bruce M Maggs. 2010. Parallel algorithms. In *Algorithms and theory of computation handbook*. 25–25.

[3] Robert David Blumofe. 1995. *Executing multithreaded programs efficiently*. Ph.D. Dissertation. Massachusetts Institute of Technology.

[4] Robert D Blumofe and Charles E Leiserson. 1999. Scheduling multithreaded computations by work stealing. *Journal of the ACM (JACM)* 46, 5 (1999), 720–748.

[5] Coen Bron and Joep Kerbosch. 1973. Algorithm 457: finding all cliques of an undirected graph. *CACM* 16, 9 (1973), 575–577.

[6] Frédéric Cazals and Chinmay Karande. 2008. A note on the problem of reporting maximal cliques. *TCS* 407, 1 (2008), 564–568.

[7] Annie Chateau, Pierre Riou, and Eric Rivals. 2011. Approximate common intervals in multiple genome comparison. In *Bioinformatics and Biomedicine (BIBM), 2011 IEEE International Conference on*. IEEE, 131–134.

[8] Yu Chen and Gordon M Crippen. 2005. A novel approach to structural alignment using realistic structural and environmental information. *Protein science* 14, 12 (2005), 2935–2946.

[9] Norishige Chiba and Takao Nishizeki. 1985. Arboricity and subgraph listing algorithms. *SIAM J. Comput.* 14 (1985), 210–223. Issue 1.

[10] Alessio Conte, Tiziano De Matteis, Daniele De Sensi, Roberto Grossi, Andrea Marino, and Luca Versari. 2018. D2K: Scalable Community Detection in Massive Networks via Small-Diameter k-Plexes. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. ACM, 1272–1281.

[11] Alessio Conte, Roberto Grossi, Andrea Marino, and Luca Versari. 2016. Sublinear-space bounded-delay enumeration for massive network analytics: maximal cliques. In *43rd International Colloquium on Automata, Languages, and Programming (ICALP 2016)*, Vol. 148. 1–148.

[12] Apurba Das, Seyed-Vahid Sanei-Mehri, and Srikanta Tirthapura. 2018. Shared-memory parallel maximal clique enumeration. In *2018 IEEE 25th International Conference on High Performance Computing (HiPC)*. IEEE, 62–71.

[13] Apurba Das, Michael Svendsen, and Srikanta Tirthapura. 2019. Incremental maintenance of maximal cliques in a dynamic graph. *The VLDB Journal* (Apr 2019).

[14] Naga Shailaja Dasari, Ranjan Desh, and Mohammad Zubair. 2014. ParK: An efficient algorithm for k-core decomposition on multicore processors. In *Big Data*. IEEE, 9–16.

[15] Nan Du, Bin Wu, Liutong Xu, Bai Wang, and Xin Pei. 2006. A parallel algorithm for enumerating all maximal cliques in complex network. In *ICDM Workshop*. IEEE, 320–324.

[16] Nan Du, Bin Wu, Liutong Xu, Bai Wang, and Pei Xin. 2009. Parallel algorithm for enumerating maximal cliques in complex network. In *Mining Complex Data*. Springer, 207–221.

[17] Dongsheng Duan, Yuhua Li, Ruixuan Li, and Zhengding Lu. 2012. Incremental K-clique clustering in dynamic social networks. *Artificial Intelligence Review* (2012), 1–19.

[18] David Eppstein, Maarten Löffler, and Darren Strash. 2010. Listing all maximal cliques in sparse graphs in near-optimal time. In *ISAAC*. 403–414.

[19] David Eppstein and Darren Strash. 2011. Listing All Maximal Cliques in Large Sparse Real-World Graphs. In *Experimental Algorithms*. Vol. 6630. 364–375.

[20] James Fox, Tim Roughgarden, C Seshadhri, and Fan Wei. 2018. Finding Cliques in Social Networks: A New Distribution-Free Model. *SIAM journal on computing* (2018).

[21] Helen M Grindley, Peter J Artymiuk, David W Rice, and Peter Willett. 1993. Identification of tertiary structure resemblance in proteins using a maximal common subgraph isomorphism algorithm. *JMB* 229, 3 (1993), 707–721.

[22] Masahiro Hattori, Yasushi Okuno, Susumu Goto, and Minoru Kanehisa. 2003. Development of a chemical structure comparison method for integrated analysis of chemical and genomic information in the metabolic pathways. *JACS* 125, 39 (2003), 11853–11865.

[23] David S Johnson, Mihalis Yannakakis, and Christos H Papadimitriou. 1988. On generating all maximal independent sets. *Inform. Process. Lett.* 27, 3 (1988), 119–123.

[24] Pall F Jonsson and Paul A Bates. 2006. Global topological features of cancer proteins in the human interactome. *Bioinformatics* 22, 18 (2006), 2291–2297.

[25] Humayun Kabir and Kamesh Madduri. 2017. Parallel k-core decomposition on multicore platforms. In *IPDPS Workshop*. IEEE, 1482–1491.

[26] Humayun Kabir and Kamesh Madduri. 2017. Parallel k-truss decomposition on multicore systems. In *HPEC*. 1–7.

[27] Humayun Kabir and Kamesh Madduri. 2017. Shared-memory graph truss decomposition. In *High Performance Computing (HiPC), 2017 IEEE 24th International Conference on*. IEEE, 13–22.

[28] Ina Koch. 2001. Enumerating all connected maximal common subgraphs in two graphs. *TCS* 250, 1 (2001), 1–30.

[29] Shungo Koichi, Masaki Arisaka, Hiroyuki Koshino, Atsushi Aoki, Satoru Iwata, Takeaki Uno, and Hiroko Satoh. 2014. Chemical structure elucidation from 13C NMR chemical shifts: Efficient data processing using bipartite matching and maximal clique algorithms. *JCIM* 54, 4 (2014), 1027–1035.

[30] D Koller and N Friedman. 2009. *Probabilistic Graphical Models: Principles and Techniques*. MIT Press.

[31] Frank Kose, Wolfram Weckwerth, Thomas Linke, and Oliver Fiehn. 2001. Visualizing plant metabolomic correlation networks using clique-metabolite matrices. *Bioinformatics* 17, 12 (2001), 1198–1208.

[32] Jérôme Kunegis. 2013. Konect: the koblenz network collection. In *WWW*. ACM, 1343–1350.

[33] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

[34] Brenton Lessley, Talita Perciano, Manish Mathai, Hank Childs, and E Wes Bethel. 2017. Maximal Clique Enumeration with Data-Parallel Primitives. In *LDAV*. IEEE, 16–25.

[35] Yinuo Li, Zhiyuan Shao, Dongxiao Yu, Xiaofei Liao, and Hai Jin. 2019. Fast Maximal Clique Enumeration for Real-World Graphs. In *International Conference on Database Systems for Advanced Applications*. Springer, 641–658.

[36] Guimei Liu and Limsoon Wong. 2008. Effective pruning techniques for mining quasi-cliques. In *Joint European conference on machine learning and knowledge discovery in databases*. Springer, 33–49.

[37] Li Lu, Yunhong Gu, and Robert Grossman. 2010. dmaximalcliques: A distributed algorithm for enumerating all maximal cliques and maximal clique distribution. In *ICDM Workshop*. IEEE, 1320–1327.

[38] Kazuhisa Makino and Takeaki Uno. 2004. New algorithms for enumerating all maximal cliques. In *SWAT*. 260–272.

[39] S Mohseni-Zadeh, Pierre Brézellec, and J-L Risler. 2004. Cluster-C, an algorithm for the large-scale clustering of protein sequences based on the extraction of maximal cliques. *Comp. Biol. Chem.* 28, 3 (2004), 211–218.

[40] Alberto Montresor, Francesco De Pellegrini, and Daniele Miorandi. 2013. Distributed k-core decomposition. *TPDS* 24, 2 (2013), 288–300.

[41] John W Moon and Leo Moser. 1965. On cliques in graphs. *Israel J. Math.* 3, 1 (1965), 23–28.

[42] Arko Provo Mukherjee and Srikanta Tirthapura. 2017. Enumerating Maximal Bicliques from a Large Graph Using MapReduce. *IEEE Trans. Services Computing* 10, 5 (2017), 771–784.

[43] Thorsten J Ottosen and Jiří Vomlel. 2010. Honour thy neighbour: clique maintenance in dynamic graphs. In *PGM*. 201–208.

[44] Gergely Palla, Imre Derényi, Illés Farkas, and Tamás Vicsek. 2005. Uncovering the overlapping community structure of complex networks in nature and society. *Nature* 435, 7043 (2005), 814–818.

[45] Hongchao Qin, Rong-Hua Li, Guoren Wang, Lu Qin, Yurong Cheng, and Ye Yuan. 2019. Mining Periodic Cliques in Temporal Networks. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 1130–1141.

[46] Oleg Rokhlenko, Ydo Wexler, and Zohar Yakhini. 2007. Similarities and differences of gene expression in yeast stress conditions. *Bioinformatics* 23, 2 (2007), e184–e190.

[47] Ryan A. Rossi and Nesreen K. Ahmed. 2015. The Network Data Repository with Interactive Graph Analytics and Visualization. In *AAAI*. 4292–4293.

[48] Pablo San Segundo, Jorge Artieda, and Darren Strash. 2018. Efficiently enumerating all maximal cliques with bit-parallelism. *Computers & Operations Research* 92 (2018), 37–46.

[49] Seyed-Vahid Sanei-Mehri, Apurba Das, and Srikanta Tirthapura. 2018. Enumerating top-k quasi-cliques. In *2018 IEEE International Conference on Big Data (Big Data)*. IEEE, 1107–1112.

[50] Ahmet Erdem Sariyuce, C Seshadhri, and Ali Pinar. 2017. Parallel local algorithms for core, truss, and nucleus decompositions. *arXiv preprint arXiv:1704.00386* (2017).

[51] Matthew C Schmidt, Nagiza F Samatova, Kevin Thomas, and Byung-Hoon Park. 2009. A scalable, parallel algorithm for maximal clique enumeration. *JPDC* 69, 4 (2009), 417–428.

[52] Ori Shalev and Nir Shavit. 2006. Split-ordered lists: Lock-free extensible hash tables. *Journal of the ACM (JACM)* 53, 3 (2006), 379–405.

[53] Julian Shun. 2017. *Shared-memory parallelism can be simple, fast, and scalable*. Morgan & Claypool.

[54] Volker Stix. 2004. Finding all maximal cliques in dynamic graphs. *Comput. Optim. Appl.* 27, 2 (2004), 173–186.

[55] Michael Svendsen, Arko Provo Mukherjee, and Srikanta Tirthapura. 2015. Mining maximal cliques from a large graph using mapreduce: Tackling highly uneven subproblem sizes. *JPDC* 79 (2015), 104–114.

[56] Etsuji Tomita, Akira Tanaka, and Haruhisa Takahashi. 2006. The worst-case time complexity for generating all maximal cliques and computational experiments. *TCS* 363, 1 (2006), 28–42.

[57] Shuji Tsukiyama, Mikio Ide, Hiromu Ariyoshi, and Isao Shirakawa. 1977. A new algorithm for generating all the maximal independent sets. *SIAM J. Comput.* 6, 3 (1977), 505–517.

[58] Takeaki Uno. 2010. An efficient algorithm for solving pseudo clique enumeration problem. *Algorithmica* 56, 1 (2010), 3–16.

[59] Zhuo Wang, Qun Chen, Boyi Hou, Bo Suo, Zhanhuai Li, Wei Pan, and Zachary G Ives. 2017. Parallelizing maximal clique and k-plex enumeration over graph data. *J. Parallel and Distrib. Comput.* 106 (2017), 79–91.

[60] Bin Wu, Shengqi Yang, Haizhou Zhao, and Bai Wang. 2009. A distributed algorithm to enumerate all maximal cliques in mapreduce. In *FCST*. IEEE, 45–51.

[61] Ting Yu and Mengchi Liu. 2019. A Memory Efficient Clique Enumeration Method for Sparse Graphs with a Parallel Implementation. *Parallel Comput.* (2019).

[62] Long Yuan, Lu Qin, Wenjie Zhang, Lijun Chang, and Jianye Yang. 2017. Index-based densest clique percolation community search in networks. *IEEE Transactions on Knowledge and Data Engineering* 30, 5 (2017), 922–935.

[63] Bing Zhang, Byung-Hoon Park, Tatiana Karpinets, and Nagiza F Samatova. 2008. From pull-down data to protein interaction networks and complexes with biological relevance. *Bioinformatics* 24, 7 (2008), 979–986.

[64] Chen Zhang, Ying Zhang, Wenjie Zhang, Lu Qin, and Jianye Yang. 2019. Efficient Maximal Spatial Clique Enumeration. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 878–889.

[65] Yun Zhang, Faisal N Abu-Khzam, Nicole E Baldwin, Elissa J Chesler, Michael A Langston, and Nagiza F Samatova. 2005. Genome-scale computational approaches to memory-intensive applications in systems biology. In *SC*. IEEE Computer Society, 12.

A PROOF OF WORK EFFICIENCY OF PARALLEL MCE ON DYNAMIC GRAPH

Here, we show the work efficiency of ParIMCE by proving Lemma 3.

Algorithm 8: TTTExcludeEdges($G, K, \mathrm{cand}, \mathrm{fini}, \mathcal{E}$)

Input: G - The input graph

K - a non-maximal clique to extend

cand - Set of vertices that may extend K

fini - vertices that have been used to extend K

$\mathcal{E}$ - set of edges to ignore

1 **if** $(\mathrm{cand} = \emptyset)$ & $(\mathrm{fini} = \emptyset)$ **then**

2 $\quad$ Output K and **return**

3 pivot $\leftarrow$ $(u \in \mathrm{cand} \cup \mathrm{fini})$ such that u maximizes the size of the intersection of $\mathrm{cand} \cap \Gamma_G(u)$

4 ext $\leftarrow$ cand $- \Gamma_G(\mathrm{pivot})$

5 **for** $q \in \mathrm{ext}$ **do**

6 $\quad K_q \leftarrow K \cup \{q\}$

7 $\quad$ **if** $K_q \cap \mathcal{E} \neq \emptyset$ **then**

8 $\quad\quad$ cand $\leftarrow$ cand $- \{q\}$

9 $\quad\quad$ fini $\leftarrow$ fini $\cup \{q\}$

10 $\quad\quad$ **continue**

11 $\quad \mathrm{cand}_q \leftarrow \mathrm{cand} \cap \Gamma_G(q)$

12 $\quad \mathrm{fini}_q \leftarrow \mathrm{fini} \cap \Gamma_G(q)$

13 $\quad$ TTTExcludeEdges($G, K_q, \mathrm{cand}_q, \mathrm{fini}_q, \mathcal{E}$)

14 $\quad$ cand $\leftarrow$ cand $- \{q\}$

15 $\quad$ fini $\leftarrow$ fini $\cup \{q\}$

Lemma 3: Given a graph G and a new edge set H, ParIMCENew is work-efficient, i.e, the total work is of the same order of the time complexity of FastIMCENewClq. The depth of ParIMCENew is $O(\Delta^2 + M \log \Delta)$ where Δ is the maximum degree and M is the size of a maximum clique in $G + H$.

PROOF. First, we prove the work efficiency of ParIMCENew followed by the depth of the algorithm. Note that, for proving the work-efficiency we will show that procedure at each line from Line 4 to Line 10 of ParIMCENew is work-efficient. Lines 4, 6, 7, and 8 are work-efficient because, all these procedures are sequential in ParIMCENew. The parallel set operations at Line 5 and Line 10 are work-efficient using Theorem 3.1. Now we will show the work efficiency of ParTTTExcludeEdges as follows. If we disregard Lines 7-10 of TTTExcludeEdges and Lines 9-10 of ParTTTExcludeEdges then the total work of ParTTTExcludeEdges is the same as the time complexity of ParTTT following the work efficiency of ParTTT. Next, we say that the time complexity of Lines 7-10 of TTTExcludeEdges is the same as the time complexity of Lines 9-10 of ParTTTExcludeEdges because in TTTExcludeEdges we use two global hashtables - one for maintaining the adjacent vertices of the currently processing vertex in the set of new edges and another for maintaining the indexes of the new edges that we define before the beginning of the enumeration of new maximal cliques. With these two hashtables, we can check the *if* condition at Line 9 of ParTTTExcludeEdges in parallel with total work $O(n)$ using Theorem 3.1 which is of the same order of the time complexity of performing *if* condition check at Line 7 of TTTExcludeEdges. This completes the proof of work efficiency of ParTTTExcludeEdges. For proving the depth of ParIMCENew, note that the depth is the sum of the depths of procedures at Line 5, 6, 9, 10 of ParIMCENew because the cost of all operations in other lines are $O(1)$ each. The depth of executing intersection in parallel at Line 5 is

$O(1)$ using Theorem 3.1, the depth of the procedure for constructing the graph at Line 6 is $O(\Delta^2)$ as we construct the graph sequentially, the depth `ParTTTExcludeEdges` is $O(M \log \Delta)$ following the depth of `TTTExcludeEdges`, and the depth of Line 10 is $O(1)$ because we can do this operation in parallel using Theorem 3.1. Thus, the overall depth of `ParIMCENew` follows.   □