Shared-Memory Parallel Maximal Clique Enumeration

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

***Abstract*—We present shared-memory parallel methods for Maximal Clique Enumeration (MCE) from a graph. MCE is a fundamental and well-studied graph analytics task, and is a widely used primitive for identifying dense structures in a graph. Due to its computationally intensive nature, parallel methods are imperative for dealing with large graphs. However, surprisingly, there do not yet exist scalable and parallel methods for MCE on a shared-memory parallel machine. In this work, we present efficient 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 that they can scale to a large number of processors, and (3) our implementations on a multicore machine shows a good speedup and scaling behavior with increasing number of cores, and are substantially faster than prior shared-memory parallel algorithms for MCE.**

I. Introduction

We study the problem of 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'. Maximal cliques are perhaps the most fundamental dense subgraphs, and MCE has been widely used in diverse research areas, such as clustering and community detection in social and biological networks [1] and in genomics [2]. It has also applications in finding common substructures in chemical compounds [3], mining from biological data [4], [5], [6], [7], [8], [9], and inference from graphical models [10].

MCE is a computationally hard problem since it is harder than the problem of finding the *maximum* clique, which is a classical NP-hard combinatorial problem. 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) may itself be very large, in the worst case. In particular, Moon and Moser [11] showed that a graph on n vertices can have as many as $3^{n/3}$ maximal cliques, which is proven to be a tight bound. Real-world networks typically do not have cliques of such high complexity and it is possible to enumerate maximal cliques from large graphs. The literature is rich on sequential algorithms for MCE. Bron and Kerbosch [12] introduced a backtracking search method to enumerate maximal cliques. Tomita et. al [13] used 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. [14], who used a degeneracy-based vertex ordering scheme 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. [13]. 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 attractive 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 with a single shared copy of the graph (2) Subproblems generated during MCE are often irregular, and it is hard to predict which subproblems are small and which are large, while initially dividing the problem into subproblems. With a shared memory method, it is easy to further subdivide subproblems and process them in parallel. With a distributed memory method, handling such irregularly sized subproblems in a load-balanced manner requires greater coordination and is more complex.

Prior works on parallel MCE have largely focused on distributed memory algorithms [15], [16], [17], [18], [19]. There are a few works on shared-memory parallel algorithms [20], [21], [22]. 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 [21] or they need to generate a large number of non-maximal cliques as in [20], [22].

A. Our Contributions

We design shared-memory parallel algorithms for enumerating all maximal cliques in a simple graph. Our contributions are as follows:

Theoretically Efficient Parallel Algorithm: We present a shared-memory parallel algorithm `ParTTT` that takes as input a graph G and enumerates all maximal cliques in G. `ParTTT` is an efficient parallelization of the algorithm due to Tomita et al. [13]. Our analysis of `ParTTT` using a work-depth model [23] of computation shows that it is work-efficient when compared with [13] 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 the following ideas to further improve the practical performance of `ParTTT`, leading to Algorithm `ParMCE`. First, instead of starting with a single task that spawns recursive subtasks as it proceeds, which leads to a lack of parallelism at the top level of recursion, we start with multiple parallel subtasks. To achieve this, we consider per-vertex parallelization, where a separate subproblem is created for each vertex and the different subproblems are processed in parallel. Each subproblem is required to enumerate cliques that contain the assigned vertex, where care is taken to prevent overlap between subproblems, and to balance the load between subproblems. Each per-vertex subproblem is further processed in parallel using `ParTTT`. This additional (recursive) level of parallelism is useful since the different per-vertex subproblems may have significantly different computational costs, having each run as a separate sequential task may lead to uneven load balance. To further address load balance, we consider different methods for ranking the vertices, so that the ranking functions can be used in creating subproblems that are balanced as much as possible. For ranking the vertices, we use metrics such as degree, triangle count, and degeneracy number of the vertices.

Experimental Evaluation: We experimentally evaluate our algorithm and show that `ParMCE` is **15x-31x** faster than an efficient sequential algorithm (due to Tomita et al. [13]) on a multicore machine with 32 physical cores and 256G RAM. For example, on the `orkut` network with around 3M vertices, 117M edges, and 2B maximal cliques [1], `ParTTT` achieves a **14x** parallel speedup over the sequential algorithm, and the optimized `ParMCE` achieves a **16x** speedup. In contrast, prior shared memory parallel algorithms for MCE [20], [21], [22] failed to handle the input graphs that we considered, and either ran out of memory ([20], [22]) or did not complete in 5 hours ([21]).

Roadmap. The rest of the paper is organized as follows. We present preliminaries in Section III, followed by a description of the algorithm and analysis in Section IV, an experimental evaluation in Section V, and conclusions in Section VI.

[1]M and B stand for million and billion respectively.

II. 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 [12] presented an algorithm for MCE based on depth-first-search. Following their work, a number of algorithms have been presented [24], [25], [13], [26], [27], [28], [14]. The algorithm of Tomita et al. [13] 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 $O(3^{\frac{n}{3}})$ [11]. Eppstein et al. [14], [29] 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 [25], [27], [24], which can be viewed as instances of a general paradigm called "reverse-search" [30]. The output-sensitive algorithm due to Makino and Uno [27] provides the best theoretical worst-case time complexity among output-sensitive algorithms. In terms of practical performance, the best output-sensitive algorithms [25], [27] are not as efficient as the best depth-first-search based algorithms such as [13], [14]. Other sequential methods for MCE include algorithms due to Kose et al. [31], Johnson et al. [32], and Cheng et al. [33]. There have also been works on maintaining maximal cliques in a dynamic graph [34], [35], [36].

Parallel MCE: There are multiple prior works on parallel algorithms for MCE [20], [37], [15], [16], [17], [38], [18], [19]. We first discuss shared memory algorithms and then distributed memory algorithms.

Zhang et al. [20] presented 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. [37] 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. [16]. Lessley et al. [22] 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 [22] 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 [22], [20], [37] in later sections.

Distributed memory parallel algorithms for MCE include works due to Wu et al. [15], designed for the MapReduce framework, Lu et al. [17], which is based on the sequential algorithm due to Tsukiyama et al. [24], Xu et al. [19], and Svendsen et al. [18].

Other works on parallel algorithms for enumerating dense subgraphs from a massive graph include parallel algorithms for enumerating k-cores [39], [40], [41], [42], k-trusses [42], [43], [44], nuclei [42], and distributed memory algorithms for enumerating bicliques [45].

III. 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 $\mathcal{C}(G)$ denote the set of all maximal cliques in G.

Sequential Algorithm `TTT`**:** The algorithm due to Tomita, Tanaka, and Takahashi. [13], 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 $q \in \texttt{cand}$ is added to K and a new recursive call is made with parameters $K \cup \{q\}$, $\texttt{cand}_q$, and $\texttt{fini}_q$ for generating all maximal cliques of G that extend $K \cup \{q\}$ but do not contain any vertices from $\texttt{fini}_q$. The sets $\texttt{cand}_q$ and $\texttt{fini}_q$ 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 [12] is the use of a "pivot" where a vertex $u \in \texttt{cand} \cup \texttt{fini}$ is selected that maximizes $|\texttt{cand} \cap \Gamma(u)|$. Once the pivot u is computed, it is sufficient to iterate over all the vertices of $\texttt{cand} \setminus \Gamma(u)$, 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 [46], 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 [23]. 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 *work-efficient* if its total work is of the same order as the work due to the best sequential algorithm[2]. We aim for work-efficient algorithms with a low depth, ideally poly-logarithmic in the size of the input. Using Brent's theorem [46], 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 $p = O(n/d)$.

Algorithm 1: $\texttt{TTT}(\mathcal{G}, K, \texttt{cand}, \texttt{fini})$

```
Input: $\mathcal{G}$ - The input graph
       $K$ - a clique to extend,
  cand - Set of vertices that can be used extend $K$,
  fini - 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 from fini
1  if $(\texttt{cand} = \emptyset)$ & $(\texttt{fini} = \emptyset)$ then
2      Output $K$ and return
3  pivot ← $(u \in \texttt{cand} \cup \texttt{fini})$ such that $u$ maximizes the
     size of $\texttt{cand} \cap \Gamma_{\mathcal{G}}(u)$
4  ext ← $\texttt{cand} - \Gamma_{\mathcal{G}}(\texttt{pivot})$
5  for $q \in$ ext do
6      $K_q \leftarrow K \cup \{q\}$
7      $\texttt{cand}_q \leftarrow \texttt{cand} \cap \Gamma_{\mathcal{G}}(q)$
8      $\texttt{fini}_q \leftarrow \texttt{fini} \cap \Gamma_{\mathcal{G}}(q)$
9      cand ← $\texttt{cand} - \{q\}$
10     fini ← $\texttt{fini} \cup \{q\}$
11     $\texttt{TTT}(\mathcal{G}, K_q, \texttt{cand}_q, \texttt{fini}_q)$
```

IV. Parallel MCE Algorithms

In this section, we present shared-memory parallel algorithms for MCE. We first describe a parallel algorithm `ParTTT` and an analysis of its theoretical properties, where `ParTTT` is a parallel version of the `TTT` algorithm. Then, we discuss practical bottlenecks in `ParTTT`, leading us to another algorithm `ParMCE` with a better practical runtime performance.

A. *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

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

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 intersection $\texttt{cand} \cap \Gamma(u)$ is computed in parallel for each vertex $u \in \texttt{cand} \cup \texttt{fini}$. 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.

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

Proof: If the sets cand and $\Gamma(w)$ are stored as hashsets, then for vertex w the size $t_w = |\texttt{intersect}(\texttt{cand}, \Gamma(w))|$ can be computed sequentially in time $O(\min\{|\texttt{cand}|, |\Gamma(w)|\})$ – 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 $\texttt{intersect}(S_1, S_2)$ by executing the searches elements in 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 logarithmic depth. 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 $O(\min\{|\texttt{cand}|, |\Gamma(w)|\})$ and depth $O(\log n)$. Once the different t_w are computed, $argmax(\{t_w : w \in \texttt{cand} \cup \texttt{fini}\})$ can be computed using additional work $|\texttt{cand} \cup \texttt{fini}|$ and depth $O(\log n)$. Hence, the total work of ParPivot is $O(\sum_{w \in \texttt{cand} \cup \texttt{fini}}(\min\{|\texttt{cand}|, |\Gamma(w)|\})$. Since the size of cand, fini, and $\Gamma(w)$ are bounded by n, this is $O(n^2)$, but typically much smaller. ■

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

```
Input: $K$ - a clique in $G$ that may be further extended
  $\mathcal{G}$ - Input graph
  cand - Set of vertices that may extend $K$
  fini - vertices that have been used to extend $K$
Output: pivot vertex $u \in \texttt{cand} \cup \texttt{fini}$
1 for $w \in \texttt{cand} \cup \texttt{fini}$ do in parallel
2     In parallel, compute
        $t_w \leftarrow |\texttt{intersect}(\texttt{cand}, \Gamma_{\mathcal{G}}(w))|$
3 In parallel, find $v \leftarrow argmax(\{t_w : w \in \texttt{cand} \cup \texttt{fini}\})$
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 of 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 to be processed in sequence. Each vertex $v_i \in \texttt{ext}$, once added to K, should be removed from further consideration from cand. To ensure this, instead of incrementally updating cand and fini with v_i as in TTT, 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. This way, the parameters of the ith iteration are computed independently of prior iterations.

Algorithm 3: $\texttt{ParTTT}(\mathcal{G}, K, \texttt{cand}, \texttt{fini})$

```
Input: $\mathcal{G}$ - 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: Set of all maximal cliques of $G$ containing $K$
        and vertices from cand but not containing any
        vertex from fini
1  if $(\texttt{cand} = \emptyset)$ & $(\texttt{fini} = \emptyset)$ then
2      Output $K$ and return
3  pivot $\leftarrow$ ParPivot($\mathcal{G}$, cand, fini)
4  ext$[1..\kappa] \leftarrow$ cand $- \Gamma_{\mathcal{G}}$(pivot) // in parallel
5
6  for $i \in [1..\kappa]$ do in parallel
7      $q \leftarrow$ ext$[i]$
8      $K_q \leftarrow K \cup \{q\}$
9      cand$_q \leftarrow$ intersect(cand $\setminus$ ext$[1..i-1]$, $\Gamma_{\mathcal{G}}(q)$)
10     fini$_q \leftarrow$ intersect(fini $\cup$ ext$[1..i-1]$, $\Gamma_{\mathcal{G}}(q)$)
11     ParTTT($\mathcal{G}$, $K_q$, cand$_q$, fini$_q$)
```

Next, we present an analysis of the total work and depth of ParPivot and ParTTT algorithms.

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 maximum clique in G.*

Proof: First, we analyze the total work. Note that the computational tasks in ParTTT is different from TTT at Line 9 and Line 10 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 9 and Line 10 of Algorithm 1). Therefore, in ParTTT, additional $O(n)$ work is required due to independent computations of $\texttt{cand}_q$ and $\texttt{fini}_q$. The total work, excluding the call to ParPivot is $O(n^2)$. Adding up the work of ParPivot, which requires $O(n^2)$ work, requires $O(n^2)$ total work for each single call of ParTTT excluding further recursive calls

(Algorithm 3, Line 11), which is same as in original sequential algorithm TTT (Section 4, [13]). Hence, using Lemma 2 and Theorem 3 of [13], 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 6 to Line 11. According to Lemma 1, the depth of ParPivot is $O(\log n)$. The depth of computing ext is $O(\log n)$ because it takes $O(1)$ time to check whether an element in cand is in the neighborhood of pivot by doing a set membership check on the set of vertices that are adjacent to pivot. Similarly, the depth of computing $\texttt{cand}_q$ and $\texttt{fini}_q$ at Line 8 and Line 9 are $O(\log n)$ each. The remaining is the depth of the call of ParTTT at Line 10. Observe 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 is increased by 1. Thus, the overall depth of ParTTT is $O(M \log n)$. ■

Corollary 1. *Using P parallel processors that share 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-efficient as long as $P = O(\frac{3^{n/3}}{M \log n})$, and also work-optimal.*

Proof: The parallel time follows from using Brent's theorem [46], 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 [11]). ■

B. Algorithm ParMCE

While ParTTT is a theoretically work-efficient parallel algorithm, we note that it is not that efficient in practice. One of the reasons is the implementation of ParPivot. While the worst case work complexity of ParPivot matches that of the pivoting routine in TTT, in practice, it may have a higher overhead, since the pivoting routine in TTT may take time less than $O(n^2)$. This can cause ParTTT to have greater work than TTT, resulting 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 that 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 the parameter K (clique so far) is empty. During this iteration, the set of vertices still to be considered, $\texttt{cand} \cup \texttt{fini}$, 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 the set K has at least one element in it, the number of the vertices in $\texttt{cand} \cup \texttt{fini}$ still to be considered, drops down 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 (it is smaller than the smallest degree of a vertex in K). Problem instances with K set to a single vertex can be seen as subproblems and on each of these subproblems, the overhead of ParPivot 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 that 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 subproblem G_v, it is important to not enumerate maximal cliques that are being enumerated elsewhere, in other subproblems. 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. The subgraphs G_v for each vertex v are 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, so that each maximal clique is enumerated in exactly one subproblem. 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. The specific ordering that is used influences the total work of the algorithm, as well as the load balance of the parallel implementation.

Load Balancing: Observe 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, and (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 subproblems that deal with a large number of maximal cliques or maximal cliques of a large size are more expensive than others.

In order to maintain the size of the subproblems approximately balanced, we use an idea from PECO [18], 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. By giving a higher rank to v than w, we are decreasing the complexity of the subproblem G_v, since the subproblem at G_v need not enumerate maximal cliques that involve any vertex whose rank is less than v. Hence, 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 subproblems. 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 so that the subgraphs and subproblems have to be explicitly copied across the network, and (2) In `ParMCE`, the vertex specific subproblem, dealing with G_v is itself handled through a parallel algorithm, `ParTTT`. However, in PECO, the subproblem 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 complexity 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 $\texttt{rank}(v) = (d(v), id(v))$ where $d(v)$ and $id(v)$ are degree and identifier of v respectively. For two vertices v and w, $\texttt{rank}(v) > \texttt{rank}(w)$ 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 $\texttt{rank}(v) = (t(v), id(v))$ where $t(v)$ is the number of triangles containing vertex v. This is more expensive to compute than degree based ranking, but may yield a better estimate of the complexity of maximal cliques within G_v.
- **Degeneracy Based Ranking [14]:** For a vertex v, define $\texttt{rank}(v) = (degen(v), id(v))$ 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 with minimum degree of each vertex k in that subgraph. A computational overhead of using this ranking is due to computing the degeneracy of the 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 rankings are called as `ParMCEDegree`, `ParMCETri`, `ParMCEDegen` respectively.

Algorithm 4: `ParMCE`($\mathcal{G}$)

```
   Input: $\mathcal{G}$ - The input graph
   Output: $\mathcal{C}(\mathcal{G})$ - set of all maximal cliques of $\mathcal{G}$
1  for $v \in V(\mathcal{G})$ do in parallel
2      Create $\mathcal{G}_v$, the subgraph of $\mathcal{G}$ induced by $\Gamma_{\mathcal{G}}(v) \cup \{v\}$
3      $K \leftarrow \{v\}$
4      cand $\leftarrow \phi$
5      fini $\leftarrow \phi$
6      for $w \in \Gamma_{\mathcal{G}}(v)$ do in parallel
7          if $rank(w) > rank(v)$ then
8              cand $\leftarrow$ cand $\cup \{w\}$
9          else
10             fini $\leftarrow$ fini $\cup \{w\}$
11     ParTTT($\mathcal{G}_v, K$, cand, fini)
```

V. Experiments

In this section, we present results from an experimental evaluation of the performance of parallel algorithms for MCE. For our experiments, we used an Intel Xeon (R) CPU on a Compute Engine in the Google Cloud Platform, with 32 physical cores and 256 GB RAM. We implement all algorithms using java 1.8 with a maximum of 100 GB heap memory for the JVM.

A. Datasets

We use large real world network datasets from KONECT [47], SNAP [48], and Network Repository [49]. Table I contains a summary of the datasets. All networks, used in our experiments, were undirected graphs. Self-loops are removed, and if the input graph was directed, we ignored the direction on the edges to derive an undirected graph.

B. Implementation of the algorithms

In our implementation of `ParTTT` and `ParMCE`, we implement a parallel `For` loop using the primitive `parallelStream()` provided by java 1.8. For computing the intersection of two sets as is required for computing `pivot` and updating `cand` and `fini` in Algorithm 3, we perform a sequential implementation. This is because the sizes of the sets `cand` and `fini` are typically not large so that we can benefit from the use of parallelism. For the garbage collection in java we use flag `-XX:+UseParallelGC` so that parallel garbage collection is run by JVM whenever required.

To compare with prior works in maximal clique enumeration, we implemented some of them [13], [14], [21], [18], [20] in Java, except the sequential algorithm `GreedyBB` [50], and the parallel algorithm `Hashing` [22], for which we used the executables provided by the authors (code written in C++). See Subsection V-D for more details.

We call our implementation of `ParMCE` using degree based vertex ordering as `ParMCEDegree`, using degeneracy based vertex ordering as `ParMCEDegen`, and using triangle count based vertex ordering as `ParMCETri`. 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 in 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, where as degree-based ordering is trivially available, without any additional computation.

We assume that the entire graph is stored in available in shared global memory. The runtime of `ParMCE` consists of (1) 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 vertices and (2) the time required to enumerate all maximal cliques. For `ParMCEDegen` and `ParMCETri` algorithms, the runtime of ranking is also reported. Figures 1 and 2 show the parallel speedup (with respect to

Dataset	$\|V\|$	$\|E\|$	# maximal cliques	avg. size of a maximal clique	size of the maximum clique
dblp-coauthors	540,486	15,245,729	139,340	11	337
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

TABLE I: Undirected graphs used for evaluation, and their properties.

the runtime of TTT) and and the total computation time of ParMCE using different vertex ordering strategies, respectively. Table III shows the breakdown of the runtime into time for ordering and the time for clique enumeration.

C. Performance of Parallel Clique Enumeration Algorithms

The total runtimes of the parallel algorithms with 32 threads are shown in Table II. We observe that ParTTT achieves a speedup of **12x** to **14x** over the sequential algorithm TTT. The three versions of ParMCE, ParMCEDegree, ParMCEDegen, ParMCETri achieve a speedup of **15x** to **31x** with 32 threads, when we consider only the runtime for maximal clique enumeration. This speedup are smaller for ParMCEDegen and ParMCETri when we add up the time taken by ranking strategies (See Figure 1).

The reason for the higher runtimes of ParTTT when compared with ParMCE is the greater cumulative overhead of computing the pivot and in processing the cand and fini sets in ParTTT. For example, for dblp-coauthors graph, in ParTTT, the cumulative overhead of computing pivot is 248 sec. and cumulative overhead of updating the cand and fini is 38 sec. whereas in ParMCE, these number are 156 sec. and 21 sec. respectively and these reduced cumulative times in ParMCE are reflected in the overall reduction in the parallel enumeration time of ParMCE over ParTTT by a factor of 2.

a) Impact of vertex ordering on overall performance of ParMCE*:* Next we consider the influence of different vertex ordering strategies, degree, degeneracy, and triangle count, on the performance of ParMCE. The total computation time when using different vertex ordering strategies are presented in Table III. Overall, we observe that degree based ordering (ParMCEDegree) usually achieves the smallest (or close to the smallest) runtime for clique enumeration, even when we don't take into account the time to compute the ordering. If we add in the time for computing the ordering, *degree based ordering is clearly better than triangle count or degeneracy based orderings*, since degree based ordering is available for free, while the degeneracy based ordering and triangle based ordering require additional computational overhead.

b) 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, and the speedup as a function of the number of threads is shown in Figure 1 and the runtimes are shown in Figure 2. We see that ParMCEDegree achieves a speedup of more than 15x on all graphs, using 32 threads. On the dblp-coauthors graph, the speedup with 32 threads was nearly 30x.

To get a better understanding of the variation of speedups achieved on different input graphs, we plotted the distribution of the sizes of maximal cliques for different input graphs, see Figure 3. We observe that the speedup of ParMCE is higher on those graphs that have large maximal cliques. For instance, there are many maximal cliques of size in the range 100 to 330 for dblp-coauthors, and we observed the highest speedup, of nearly 30x with 32 threads, for dblp-coauthors. A good speedup of nearly 20x was also observed for the orkut graph, which has a large number of maximal cliques, which are of relatively large sizes (the average size of a maximal clique is 20). Overall, we see that the speedup obtained is roughly correlated with the complexity of the graph, measured in terms of the presence of large maximal cliques, as well as the number of such large maximal cliques.

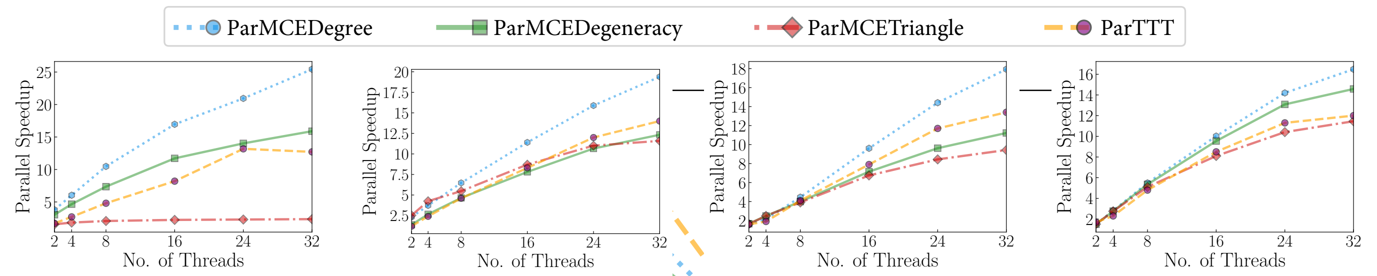
D. Comparison with prior work

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. [50], TTT due to Tomita et al. [13], and BKDegeneracy due to Eppstein et al. [14]. For the comparison with parallel algorithm, we consider algorithm CliqueEnumerator due to Zhang et al. [20], Peamc due to Du et al. [21], PECO due to Svendsen et al. [18], and most recent parallel algorithm Hashing due to Lessley et al. [22]. 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 subproblem construction, and eliminating the need to communicate subgraphs by storing a single copy of the graph in shared memory. We considered 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 IV. We note that ParMCE is significantly better than that of PECO, no matter which ordering strategy was considered.

The comparison of ParMCE with other shared memory algorithms Peamc, CliqueEnumerator, and Hashing is shown in Table V. 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 around 50 secs for enumerating 139K 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

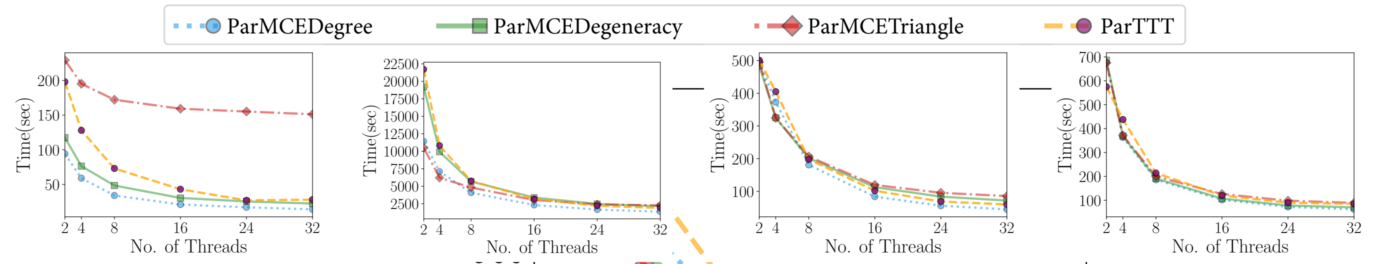
DataSet	TTT	ParTTT	ParMCEDegree	ParMCEDegen	ParMCETri
dblp-coauthors	356	28	14	21.4	152.2
orkut	26,407	1886	1362	2141.1	2278
as-skitter	807	60	45	71.9	85.6
wiki-talk	1022	85	62	70.1	89.2

TABLE II: Comparison of total computation time (in sec.) of ParMCE (with degree based, degeneracy based, and triangle count based vertex ordering) and computation time ParTTT (with 32 threads) with TTT.



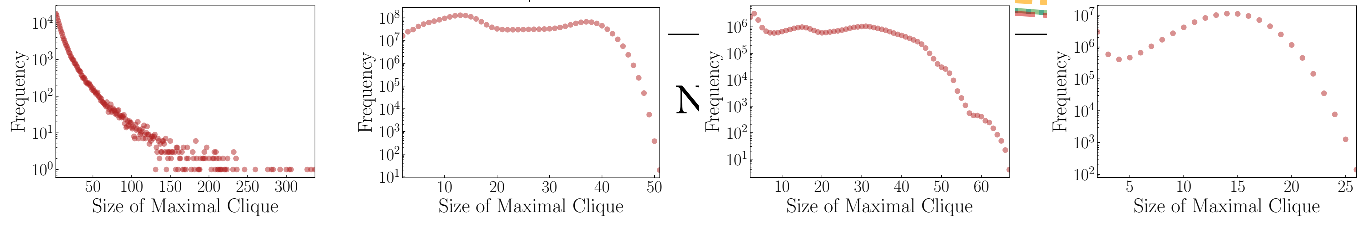
(a) dblp-coauthor (b) orkut (c) as-skitter (d) wiki-talk

Fig. 1: Parallel speedup of ParMCEDegree, ParMCEDegen, ParMCETri, and ParTTT with respect to TTT as a function of the number of threads. We use total computation time (time for computing ranking + parallel enumeration time) for measuring the speedup.



(a) dblp-coauthor (b) orkut (c) as-skitter (d) wiki-talk

Fig. 2: Total computation time of ParMCEDegree, ParMCEDegen, ParMCETri, and ParTTT as a function of the number of threads.



(a) dblp-coauthor (b) orkut (c) as-skitter (d) wiki-talk

Fig. 3: Frequency distribution of sizes of maximal cliques across different input graphs.

DataSet	ParMCEDegree	ParMCEDegen			ParMCETri		
		RT	ET	TT	RT	ET	TT
dblp-coauthors	**14**	8.4	13	**21.4**	138.2	14	**152.2**
orkut	**1362**	599.1	1542	**2141.1**	786.7	1492	**2278**
as-skitter	**45**	26.9	45	**71.9**	43.6	42	**85.6**
wiki-talk	**62**	8.1	62	**70.1**	30.2	59	**89.2**

TABLE III: Total computation time (in sec.) of ParMCEDegree, ParMCEDegen, and ParMCETri. "RT" stands for time for computing the vertex ranking, "ET" stands for parallel enumeration time, and "TT" stands for total computation time.

DataSet	PECODegree	ParMCEDegree	PECODegen	ParMCEDegen	PECOTri	ParMCETri
dblp-coauthors	73	**14**	78	**14**	74	**13**
orkut	2001	1362	7502	1542	2500	1492
as-skitter	272	**45**	450	**45**	267	**42**
wiki-talk	1423	**62**	1776	**62**	1534	**59**

TABLE IV: Comparison of parallel enumeration time (in sec.) of `ParMCE` with `PECO` (modified to use shared memory), using 32 threads. Three different variants are considered for each algorithm, based on the ordering strategy used.

DataSet	ParMCEDegree	Hashing	Clique Enumerator	Peamc
dblp-coauthors	14	run out of memory in 3 min.	run out of memory in 10 min.	did not complete in 5 hours.
orkut	1362	run out of memory in 7 min.	run out of memory in 20 min.	did not complete in 5 hours.
as-skitter	45	run out of memory in 5 min.	run out of memory in 10 min.	did not complete in 5 hours.
wiki-talk	62	run out of memory in 10 min.	run out of memory in 20 min.	did not complete in 5 hours.

TABLE V: Comparison of total computation time (in sec.) of `ParMCE` with `Hashing`.

the maximality of a clique in the search space is not efficient. The `CliqueEnumerator` algorithms run 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.

Next, we compare the performance of `ParMCE` with that of sequential algorithms `BKDegeneracy` and a recent sequential algorithm `GreedyBB` – results are in Table VI. 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.

E. Summary of Experimental Results

We found that both `ParTTT` and `ParMCE` yield significant speedups over the sequential algorithm `TTT`, sometimes as much as 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.

VI. 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. [13], and `ParMCE` is a practical adaptation of `ParTTT` that has more opportunities for parallelization and better load balancing. Our algorithms are significant improvements compared with the current state-of-the-art on MCE. Our experiments show that `ParMCE` has a speedup of up to 31x (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. 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] G. Palla, I. Derényi, I. Farkas, and T. Vicsek, "Uncovering the overlapping community structure of complex networks in nature and society," *Nature*, vol. 435, no. 7043, pp. 814–818, 2005.

[2] O. Rokhlenko, Y. Wexler, and Z. Yakhini, "Similarities and differences of gene expression in yeast stress conditions," *Bioinformatics*, vol. 23, no. 2, pp. e184–e190, 2007.

[3] S. Koichi, M. Arisaka, H. Koshino, A. Aoki, S. Iwata, T. Uno, and H. Satoh, "Chemical structure elucidation from 13c nmr chemical shifts: Efficient data processing using bipartite matching and maximal clique algorithms," *Journal of chemical information and modeling*, vol. 54, no. 4, pp. 1027–1035, 2014.

[4] H. M. Grindley, P. J. Artymiuk, D. W. Rice, and P. Willett, "Identification of tertiary structure resemblance in proteins using a maximal common subgraph isomorphism algorithm," *J. Mol. Biol.*, vol. 229, no. 3, pp. 707–721, 1993.

[5] M. Hattori, Y. Okuno, S. Goto, and M. Kanehisa, "Development of a chemical structure comparison method for integrated analysis of chemical and genomic information in the metabolic pathways," *J. Am. Chem. Soc.*, vol. 125, no. 39, pp. 11 853–11 865, 2003.

[6] S. Mohseni-Zadeh, P. Brézellec, and J.-L. Risler, "Cluster-c, an algorithm for the large-scale clustering of protein sequences based on the extraction of maximal cliques," *Comp. Biol. Chem.*, vol. 28, no. 3, pp. 211–218, 2004.

[7] Y. Chen and G. M. Crippen, "A novel approach to structural alignment using realistic structural and environmental information," *Protein science*, vol. 14, no. 12, pp. 2935–2946, 2005.

DataSet	BKDegeneracy	GreedyBB	ParMCEDegree	ParMCEDegen	ParMCETri
dblp-coauthors	231	did not finish in 30 min.	14	21.4	152.2
orkut	19,958	run out of memory in 5 min.	1362	2141.1	2278
as-skitter	588	out of memory in 10 min.	45	71.9	85.6
wiki-talk	844	run out of memory in 10 min.	62	70.1	89.2

TABLE VI: Total computation time (sec.) of ParMCE (with 32 threads) and sequential BKDegeneracy and GreedyBB.

[8] P. F. Jonsson and P. A. Bates, "Global topological features of cancer proteins in the human interactome," *Bioinformatics*, vol. 22, no. 18, pp. 2291–2297, 2006.

[9] B. Zhang, B.-H. Park, T. Karpinets, and N. F. Samatova, "From pull-down data to protein interaction networks and complexes with biological relevance," *Bioinformatics*, vol. 24, no. 7, pp. 979–986, 2008.

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

[11] J. W. Moon and L. Moser, "On cliques in graphs," *Israel J. Math.*, vol. 3, no. 1, pp. 23–28, 1965.

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

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

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

[15] B. Wu, S. Yang, H. Zhao, and B. Wang, "A distributed algorithm to enumerate all maximal cliques in mapreduce," in *Frontier of Computer Science and Technology, 2009. FCST'09. Fourth International Conference on*. IEEE, 2009, pp. 45–51.

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

[17] L. Lu, Y. Gu, and R. Grossman, "dmaximalcliques: A distributed algorithm for enumerating all maximal cliques and maximal clique distribution," in *Data Mining Workshops (ICDMW), 2010 IEEE International Conference on*. IEEE, 2010, pp. 1320–1327.

[18] M. Svendsen, A. P. Mukherjee, and S. Tirthapura, "Mining maximal cliques from a large graph using mapreduce: Tackling highly uneven subproblem sizes," *Journal of Parallel and distributed computing*, vol. 79, pp. 104–114, 2015.

[19] Y. Xu, J. Cheng, and A. W.-C. Fu, "Distributed maximal clique computation and management," *IEEE Transactions on Services Computing*, vol. 9, no. 1, pp. 110–122, 2016.

[20] Y. Zhang, F. N. Abu-Khzam, N. E. Baldwin, E. J. Chesler, M. A. Langston, and N. F. Samatova, "Genome-scale computational approaches to memory-intensive applications in systems biology," in *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*. IEEE Computer Society, 2005, p. 12.

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

[22] B. Lessley, T. Perciano, M. Mathai, H. Childs, and E. W. Bethel, "Maximal clique enumeration with data-parallel primitives," in *2017 IEEE 7th Symposium on Large Data Analysis and Visualization (LDAV)*. IEEE, 2017, pp. 16–25.

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

[24] S. Tsukiyama, M. Ide, H. Ariyoshi, and I. Shirakawa, "A new algorithm for generating all the maximal independent sets," *SIAM J. Comput.*, vol. 6, no. 3, pp. 505–517, 1977.

[25] N. Chiba and T. Nishizeki, "Arboricity and subgraph listing algorithms," *SIAM J. Comput.*, vol. 14, pp. 210–223, 1985.

[26] I. Koch, "Enumerating all connected maximal common subgraphs in two graphs," *Theoretical Computer Science*, vol. 250, no. 1, pp. 1–30, 2001.

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

[28] F. Cazals and C. Karande, "A note on the problem of reporting maximal cliques," *Theoretical Computer Science*, vol. 407, no. 1, pp. 564–568, 2008.

[29] D. Eppstein and D. Strash, "Listing all maximal cliques in large sparse real-world graphs," in *Experimental Algorithms*, ser. LNCS, P. Pardalos and S. Rebennack, Eds., 2011, vol. 6630, pp. 364–375.

[30] D. Avis and K. Fukuda, "Reverse search for enumeration," *Discrete Applied Mathematics*, vol. 65, pp. 21–46, 1993.

[31] F. Kose, W. Weckwerth, T. Linke, and O. Fiehn, "Visualizing plant metabolomic correlation networks using clique-metabolite matrices." *Bioinformatics*, vol. 17, no. 12, pp. 1198–1208, 2001.

[32] D. S. Johnson, M. Yannakakis, and C. H. Papadimitriou, "On generating all maximal independent sets," *Information Processing Letters*, vol. 27, no. 3, pp. 119–123, 1988.

[33] J. Cheng, Y. Ke, A. W.-C. Fu, J. X. Yu, and L. Zhu, "Finding maximal cliques in massive networks," *TODS*, vol. 36, no. 4, p. 21, 2011.

[34] A. Das, M. Svendsen, and S. Tirthapura, "Change-sensitive algorithms for maintaining maximal cliques in a dynamic graph," *CoRR*, vol. abs/1601.06311, 2016. [Online]. Available: http://arxiv.org/abs/1601.06311

[35] V. Stix, "Finding all maximal cliques in dynamic graphs," *Comput. Optim. Appl.*, vol. 27, no. 2, pp. 173–186, 2004.

[36] T. J. Ottosen and J. Vomlel, "Honour thy neighbour: clique maintenance in dynamic graphs," in *PGM*, 2010, pp. 201–208.

[37] N. Du, B. Wu, L. Xu, B. Wang, and X. Pei, "A parallel algorithm for enumerating all maximal cliques in complex network," in *Data Mining Workshops, 2006. ICDM Workshops 2006. Sixth IEEE International Conference on*. IEEE, 2006, pp. 320–324.

[38] J. Cheng, L. Zhu, Y. Ke, and S. Chu, "Fast algorithms for maximal clique enumeration with limited memory," in *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2012, pp. 1240–1248.

[39] A. Montresor, F. De Pellegrini, and D. Miorandi, "Distributed k-core decomposition," *IEEE Transactions on parallel and distributed systems*, vol. 24, no. 2, pp. 288–300, 2013.

[40] N. S. Dasari, R. Desh, and M. Zubair, "Park: An efficient algorithm for k-core decomposition on multicore processors," in *Big Data (Big Data), 2014 IEEE International Conference on*. IEEE, 2014, pp. 9–16.

[41] H. Kabir and K. Madduri, "Parallel k-core decomposition on multicore platforms," in *Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2017 IEEE International*. IEEE, 2017, pp. 1482–1491.

[42] A. E. Sariyuce, C. Seshadhri, and A. Pinar, "Parallel local algorithms for core, truss, and nucleus decompositions," *arXiv preprint arXiv:1704.00386*, 2017.

[43] H. Kabir and K. Madduri, "Shared-memory graph truss decomposition," *arXiv preprint arXiv:1707.02000*, 2017.

[44] ——, "Parallel k-truss decomposition on multicore systems," in *High Performance Extreme Computing Conference (HPEC), 2017 IEEE*. IEEE, 2017, pp. 1–7.

[45] A. P. Mukherjee and S. Tirthapura, "Enumerating maximal bicliques from a large graph using mapreduce," *IEEE Trans. Services Computing*, vol. 10, no. 5, pp. 771–784, 2017.

[46] G. E. Blelloch and B. M. Maggs, "Parallel algorithms," in *Algorithms and theory of computation handbook*, 2010, pp. 25–25.

[47] J. Kunegis, "Konect: the koblenz network collection," in *Proceedings of the 22nd International Conference on World Wide Web*. ACM, 2013, pp. 1343–1350.

[48] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, 2014.

[49] R. A. Rossi and N. K. Ahmed, "The network data repository with interactive graph analytics and visualization," in *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015. [Online]. Available: http://networkrepository.com

[50] P. San Segundo, J. Artieda, and D. Strash, "Efficiently enumerating all maximal cliques with bit-parallelism," *Computers & Operations Research*, vol. 92, pp. 37–46, 2018.